h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmn
FACHBEREICH MATHEMATIK
UND NATURWISSENSCHAFTEN

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt

Fachbereiche Mathematik und
Naturwissenschaften & Informatik

Untersuchung von Data Science Projekten auf GitHub

Abschlussarbeit zur Erlangung des
akademischen Grades

Master of Science (M. Sc.)
im Studiengang Data Science

vorgelegt von

Jan Patrick Holler

Referent	:	Prof. Dr. Markus Döhring
Korreferent	:	Prof. Dr. Sebastian Döhler

Ausgabedatum	:	3. Juni 2019
Abgabedatum	:	16. Dezember 2019

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 16. Dezember 2019

Jan Patrick Holler

ABSTRACT

In the age of big data the field of Data Science gets more relevant. A lot of data has to be collected, cleaned and analyzed. But the role of a Data Scientist is broad. The responsibilities reaches from data collecting und cleaning, analyzing, visualizing to Machine Learning and Artificial Intelligence. But how is the work split in Data Science projects? Are specific roles assigned or does everyone work every role? Especially for recruiting this topic is interesting. The result could give insight on recruiting more specifically. Even the job descriptions and titles could get more specific to the employees wishes.
Within the scope of this paper Data Science projects will be analyzed to find role distribution. For this task publicly available projects hosted on GitHub are used as the database. Specifically only IPython files will be analyzed, since it's assumed it's more likely to find Data Science projects there compared to other formats. Based on the commit changes the written source code will be assigned to its author. Using abstract syntax trees used functions are extracted and matched with their module.
The collected data will be grouped by authors. Based on the functions used a topic modeling will be done to find a possible role distribution. A distribution of used functions can be observed, although it isn't possible to deduce subject areas.

ZUSAMMENFASSUNG

Im Zeitalter der großen Datenmengen wird der Bereich Data Science immer relevanter. Viele Daten müssen erhoben, bereinigt und analysiert werden. Doch die Rollen des Data Scientist sind weitreichend. Die Aufgabenbereiche reichen von Datenerhebung, -bereinigung, Analyse und Visualisierung bis hin zu Machine Learning und künstliche Intelligenz. Doch wie erfolgt die Aufgabenverteilung innerhalb eines Data Science Projektes? Werden spezifische Rollen angenommen oder bearbeitet jeder jeden Aufgabenbereich? Gerade fürs Recruiting ist diese Fragestellung interessant, da mit dieser Erkenntnis gezielter nach potenziellen Arbeitnehmern gesucht werden kann. Auch die Stellenbeschreibungen selbst können dadurch spezifischer auf die Arbeitnehmer angepasst werden.
Im Rahmen dieser Arbeit werden Data Science Projekte auf Rollenverteilung untersucht. Hierbei werden öffentliche Repositories von GitHub als Datenbasis verwendet. Es werden ausschließlich IPython Dateien untersucht, da angenommen wird, dass in dieser Umgebung am wahrscheinlichsten Data Science Projekte geschrieben sind. Anhand der Commit Änderungen wird geschriebener Source Code den jeweiligen Autoren zugewiesen. Mit Hilfe von Abstract Syntax Trees (AST) werden im Source Code aufgerufene Funktionen extrahiert und den entsprechenden Modulen zugewiesen.
Die gesammelten Daten werden nach Autoren gruppiert. Anhand der verwendeten Funktionen wird ein Topic Modeling durchgeführt um mögliche Rollen abzuleiten. Es kann eine Verteilung von aufgerufenen Funktionen gebildet werden, diese ist aber nicht eindeutig in feste Themenbereiche unterteilbar.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

Teil I

THESIS

1 EINLEITUNG

1.1 MOTIVATION

Im Zeitalter der großen Datenmengen wird der Bereich Data Science immer relevanter. Viele Daten müssen erhoben, bereinigt und analysiert werden. Doch die Rollen des Data Scientist sind weitreichend. Die Aufgabenbereiche reichen von Datenerhebung, -bereinigung, Analyse und Visualisierung bis hin zu Machine Learning und künstliche Intelligenz. Doch wie erfolgt die Aufgabenverteilung innerhalb eines Data Science Projektes? Werden spezifische Rollen angenommen oder bearbeitet jeder jeden Aufgabenbereich? Gerade fürs Recruiting ist diese Fragestellung interessant, da mit dieser Erkenntnis gezielter nach potenziellen Arbeitnehmern gesucht werden kann. Auch die Stellenbeschreibungen selbst können dadurch spezifischer auf die Arbeitnehmer angepasst werden.

1.2 ZIEL DER ARBEIT

Das Feld Data Science umfasst viele verschiedene Bereiche. Es existieren verschiedene Berufsbezeichnungen für die Rollen, die ein Data Scientist annehmen kann. Die zentrale Frage dieser Arbeit ist die der Rollenverteilung in Data Science Projekten.
Im Rahmen dieser Fragestellung werden spezifische Projekte innerhalb der IPython Umgebung Jupyter betrachtet. Ziel ist es zu untersuchen ob innerhalb eines gemeinsam erstellten Projektes eine Rollenverteilung stattfindet. Hierfür erfolgt eine Zuordnung zwischen Person und geschriebenem Code.
Anschließend wird untersucht, ob Entwickler in Data Science Projekten spezifische Rollen einnehmen, die sie in den Projekten ausüben.

1.3 AUFBAU DER ARBEIT

Im Kapitel 2 wird ein grundlegendes Verständnis über die in dieser Arbeit behandelten Themen geschaffen. Insbesondere werden Grundlagen der Natural Language Processing und Topic Modeling im Detail erklärt.
Im Anschluss wird in Kapitel 3 beschrieben wie mit Hilfe von Abstract Syntax Trees Funktionsaufrufe innerhalb einer Source Code Da-

tei ihren zugehörigen Modulen zugewiesen werden.
In Kapitel 4 kommt dieser Ansatz zum Einsatz, wenn aus Commit Änderungen die Funktionen extrahiert werden. Des weiteren wird der Datensatz, der von der GitHub Webseite generiert wird, beschrieben.
Kapitel 5 widmet sich dem Topic Modeling, welches auf Grundlage des geschriebenen Source Codes durchgeführt wird.
Kapitel 6 bildet den Schluss dieser Arbeit. Dort werden die Ergebnisse und Erkenntnisse dieser Arbeit zusammengefasst.

2

GRUNDLAGEN

In diesem Kapitel werden grundlegende Konzepte vorgestellt, die zum weiteren Verständnis dieser Arbeit notwendig sind.

2.1 IPYTHON

IPython ist eine interaktive Python Umgebung. In dieser Umgebung ist es möglich mehrere Codeblöcke zu erstellen, die getrennt voneinander ausgeführt werden können. Die Ausgabe und Visualisierung von einzelnen Blöcken ist auf der Oberfläche möglich.
Hauptfunktionen der interaktiven Umgebung[8]:

- Selbstüberprüfung von Objekten
- Session Logging und Reloading
- Zugang zur System Shell
- Sogenannte Magic Commands

IPython Dateien werden in der Regel in einem Jupyter Notebook geöffnet. Die Nutzeroberfläche ist in Abbildung 2.1 zu sehen. Neben Code-Blöcken können auch Markdown Blöcke erstellt werden. Dies ermöglicht eine übersichtliche Gliederung und Dokumentation der Notebooks.

IPython Notebooks haben eine ".ipynb" Dateiendung und liegen im JSON-Format vor. Listing 2.1 zeigt die unterliegende Struktur des in Abbildung 2.1 gezeigten Notebooks. Neben Informationen über Python Kernel und Version ist zu jedem Block der Typ des Inhalts angegeben. Bei Code Blöcken ist hier auch der erzeugte Output und Anzahl der Ausführungen gespeichert.

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "# Titel"
   ]
```

```
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import pandas as pd\n",
    "df = pd.DataFrame"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.6.5"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```


Listing 2.1: JSON-Struktur eines simplen Jupyter Notebooks

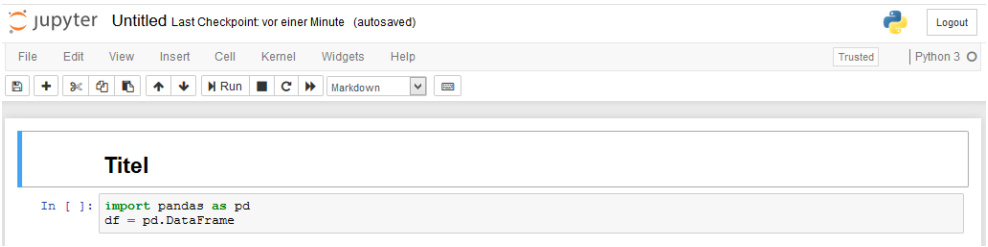


Abbildung 2.1: Nutzeroberfläche Jupyter Notebook

Laut der offiziellen Jupyter Doku[1] sind folgende Anwendungsfelder gegeben:

- Data Science
- Scientific Computing
- Education
- Enterprise

2.2 NATURAL LANGUAGE PROCESSING

Natural Language Processing (NLP) ist ein Bereich des Machine Learnings, der sich mit der maschinellen Verarbeitung von natürlicher Sprache beschäftigt. Hierbei werden Methoden aus der Sprachwissenschaft verwendet und mithilfe von Machine Learning umgesetzt. Im Rahmen dieser Arbeit sind insbesondere die Bereiche Syntaktik und Semantik von Bedeutung. Alle Informationen über NLP sind entweder [4] oder [3] entnommen, sofern nicht anders angegeben.

2.2.1 *Ebenen des Sprachverständnisses*

Im Bereich der Linguistik wird die natürliche Sprache in mehrere Ebenen unterteilt. Jede dieser Ebenen beschreibt eine Eigenschaft der Sprache. Mit Hilfe von externen Informationen können diese Eigenschaften interpretiert werden. (Abbildung 2.2).

Die Phonetik beschäftigt sich mit den Eigenschaften des Klanges der Sprache. Es wird analysiert wie der Ton erzeugt wird und welche Klänge erkennbar sind.
In der Phonologie wird den Klängen eine Bedeutung zugewiesen. Mit Hilfe von Informationen über ein gewähltes Sprachsystem werden die Laute interpretiert und es können Zeichen abgeleitet werden.
Wobei die Morphologie sich mit der Bildung von Wörtern aus diesen Zeichen widmet, werden diese durch die Lexik in einen sprachlichen Kontext gebracht. Dies geschieht mit dem Wissen über den Aufbau der Wörter, bzw. durch Verwendung eines Wörterbuches.

Diese Wörter können mithilfe von Regeln der Grammatik zu zusammenhängenden Sätzen kombiniert werden. Die Struktur der Sätze werden im Syntax analysiert.

1 https://jupyter.readthedocs.io/en/latest/use-cases/content-user.html

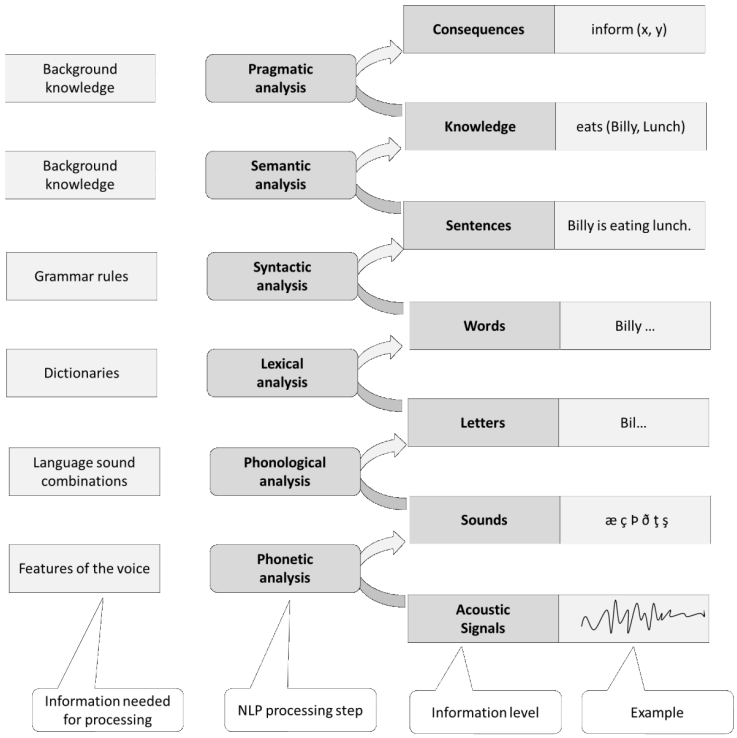


Abbildung 2.2: Sieben Ebenen des Sprachverständnisses [4]

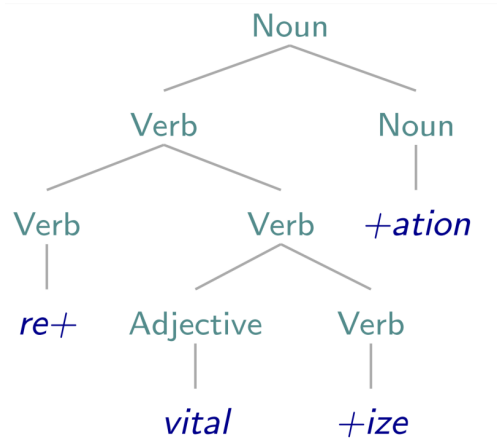


Abbildung 2.3: Morphologie [5]

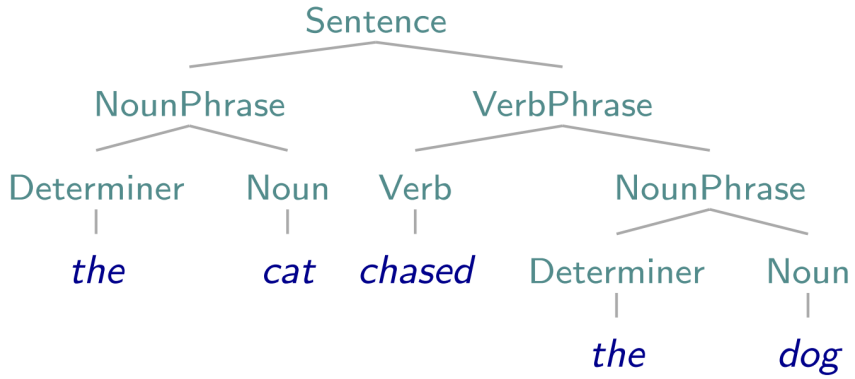


Abbildung 2.4: Syntax [5]

Die Semantik hingegen interpretiert den Inhalt der Sätze und weist diesem Bedeutung zu. Hierfür sind Hintergrundinformationen notwendig.
In der Pragmatik wird die Intention des Satzes analysiert. Auch hierfür werden Hintergrundinformationen für die Analyse benötigt. Die Bedeutung von Wörtern kann sich bei verschiedenen Intentionen des Satzes ändern. Ein gängiges Beispiel hierfür ist die Verwendung von Sarkasmus.

Im Rahmen dieser Arbeit wird hauptsächlich die Syntaktik und Semantik betrachtet.

2.2.2 *Verarbeitungsschritte*

Aus all diesen Ebenen des Sprachverständnisses sind für die meisten Use Cases nur Teilgruppen relevant. Da im Rahmen dieser Arbeit ausschließlich geschriebener Text betrachtet wird, werden die Bereiche Phonetik und Phonologie im weiteren nicht weiter thematisiert.
Um einen geschriebenen Text in verwertbare Bestandteile zu zerlegen gibt es einige Methoden. Im folgenden werden die gängigsten erläutert.

2.2.2.1 *Sentence Splitting*

Um einen Text verarbeiten zu können, ist es vorteilhaft diesen vorher in logische Bereiche zu unterteilen. Diese Bereiche werden je nach Use Case gewählt und können von Kapiteln, Paragraphen, Sätze bis zu Zeichenketten reichen. Eine typische Aufteilung eines Textes erfolgt hier satzweise.

In der Regel wird ein Satz mit einem Punkt (".") beendet. Das Suchen nach diesem Satzzeichen alleine reicht jedoch nicht immer aus, da es auch innerhalb eines Satzes z.B. bei der Darstellung von Zahlen (0.5) oder Abkürzungen (z.B.) verwendet werden kann.
Lösungsansätze reichen von regulären Ausdrücken in unterschiedlicher Komplexität bis hin zu Maschinelles Lernen.

2.2.2.2 *Tokenisierung*

Die Tokenisierung in der Linguistik beschreibt die Gruppierung von textuellen Zeichenketten in linguistische Einheiten. Diese Einheiten, auch Token genannt, sind z.B. Wörter, Zahlen, Satzzeichen, etc..
Die Gruppierung erfolgt mit Hilfe von Hintergrundinformationen über die Regeln der gegebenen Sprache.
Betrachtet wird folgender Beispielsatz:
"My dog also likes eating sausage."
Die korrekte Tokenisierung sieht wie folgt aus. (Abbildung 2.5)

Abbildung 2.5: Beispiel Tokenisierung [4]

2.2.2.3 *Part-of-Speech(POS) Tagging*

Beim Part-of-Speech Tagging werden den vorher gebildeten Token die korrekten grammatikalischen Kategorien zugeordnet. Auch für diesen Schritt der Verarbeitung erfolgt die Zuordnung mit Hilfe von Hintergrundinformationen über die Regeln der gegebenen Sprache. Abbildung 2.6 zeigt das POS Tagging Ergebnis des Beispielsatzes "My dog also likes eating sausage.".

2.2.2.4 *Parsing*

Aufbauend auf den Ergebnissen des POS Tagging kann die Grammatik des Satzes noch weiter analysiert werden. In diesem Schritt werden die Token nicht mehr einzeln, sondern in Bezug zu dem ganzen Satz betrachtet. Das Ergebnis dieses Schrittes wird in der Regel in einer Baum-Struktur angezeigt.
Abbildung 2.7 zeigt das Parsing Ergebnis des Beispielsatzes "My dog also likes eating sausage.".

Eine solche semantische Analyse ist jedoch nicht mit 100% Richtigkeit durchführbar. Wenn ein Mensch einen Satz verarbeitet, wird

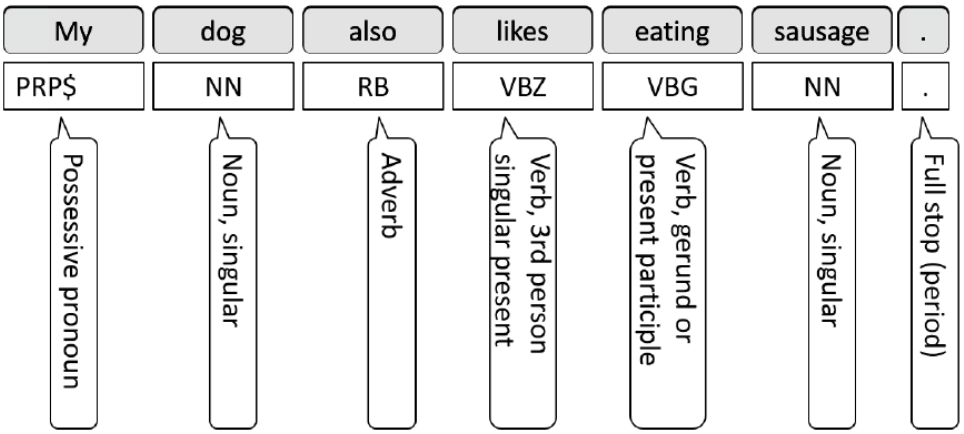


Abbildung 2.6: Beispiel POS Tagging[4]

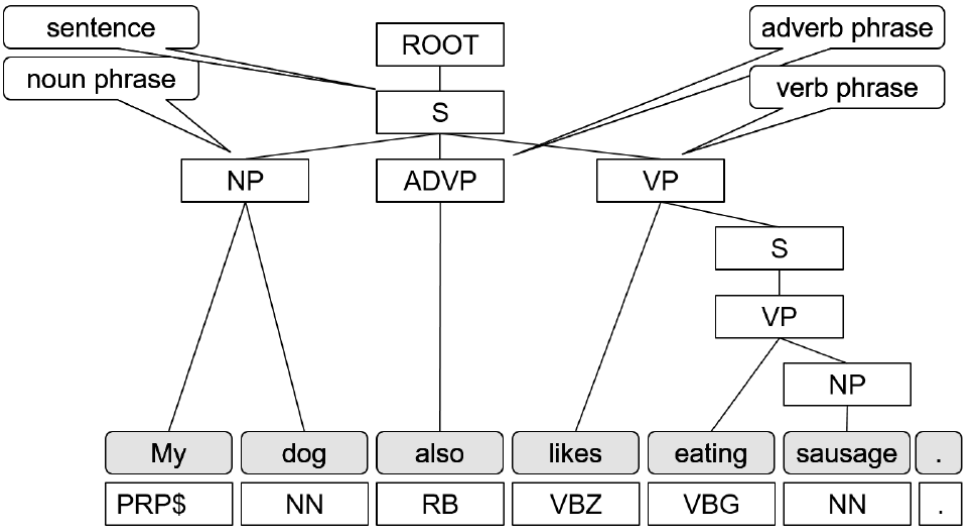


Abbildung 2.7: Beispiel Parsing[4]

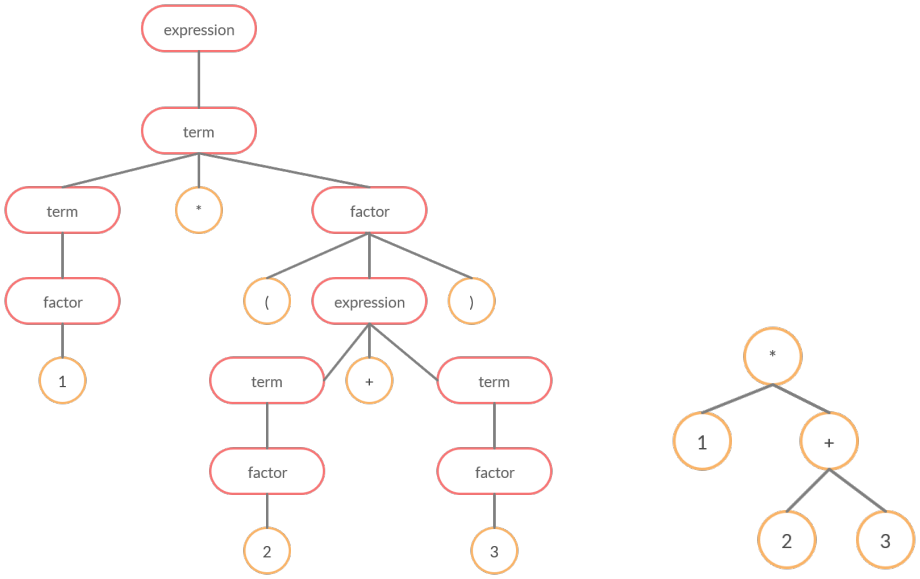
anhand von dem zuvor gesammelten Wissen und Erfahrung entschieden was die wahrscheinlichste Interpretation ist. Für einen Computer stellt diese Aufgabe eine größere Herausforderung dar. Sätze sind nicht immer eindeutig interpretierbar und dementsprechend ist keine klare Zuordnung möglich.

2.3 ABSTRACT SYNTAX TREES

Ähnlich wie ein Text kann auch Source Code mittels Parsing analysiert werden. Hierbei gibt es Unterschiedliche Darstellungen des Source Codes in einer Baumstruktur, Parse Tree und Abstract Syntax Tree.

2.3.1 *Concrete Vs. Abstract Syntax Tree*

Ein Parse Tree, auch Concrete Syntax Tree (CST) genannt, bietet eine genaue Darstellung des Source Codes. Da hier jedes Zeichen unverändert übernommen wird, hängt die Darstellung von der Programmiersprache ab und kann variieren. Nur mit Kenntnis über die verwendete Programmiersprache kann anhand des CST überprüft werden ob der Code die korrekte Syntax hat. Ein Abstract Syntax Tree (AST), auch nur Syntax Tree genannt, hingegen beinhalten nur die signifikanten Bereiche eines Ausdruckes. Alle syntaktischen Details die im CST enthalten sind werden bei der Überführung verworfen und nur die Funktionalität des Source Code wird abstrakt dargestellt. Diese Darstellungsweise ist dadurch viel kompakter als die des CST und es sind keine Details mehr über die Synatx der Programmiersprache enthalten. Diese simplere Darstellung des Source Codes erleichtert die weitere Verarbeitung durch einen Compiler.
Abbildung 2.8 zeigt den direkten Vergleich zwischen einem Parse Tree und einem Syntac Tree für den Ausdruck "$1 * (2 + 3)$". Überflüssige Knoten, die keine Informationen über die Funktionalität beinhalten, werden beim Überführen in den AST verworfen und nur die relevanten Informationen werden dargestellt. Wobei beim CST die Operatorknoten und Faktoren jeweils als Kindknoten dargestellt werden, werden Operatoren beim AST als Elternknoten dargestellt, mit den Faktoren als Kindknoten.



(a) Concrete Syntax Tree (b) Abstract Syntax Tree

Abbildung 2.8: Vergleich der Darstellung des Ausdruckes: "$1 * (2 + 3)$"

2.3.2 *AST in Python*

Die Programmiersprache Python stellt schon bei Installation eine AST[2] Bibliothek zur Generierung und Verarbeitung von Abstract Syntax Trees zur Verfügung.
Mit diesem Modul ist es möglich, Source Code zu einem AST zu parsen. Hierbei kann der Source Code untersucht werden, ohne diesen ausführen zu müssen. Abbildung 2.9 zeigt einen AST zum Ausdruck "$1 * (2 + 3)$", wie er durch das Python AST Modul generiert wurde. Es ist die gleiche Struktur wie im AST aus 2.8 erkennbar. Der Ausdruck hat als obersten Knoten die Binäroperation Multiplikation. Kinderknoten sind jeweils auf der linken Seite die Zahl 1 und auf der rechten Seite die Addition von 2 und 3.

2.4 TOPIC MODELING

Das Ziel des Topic Modeling ist die Annotation von großen Mengen an Dokumenten anhand von Themen. Algorithmen zum Topic Modeling sind statistische Methoden, die Wörter innerhalb von Texten analysieren und zu Themengebieten zusammenzufassen. Diese Algorithmen können auf große Datenmengen angewendet werden und diese

2 https://docs.python.org/3/library/ast.html

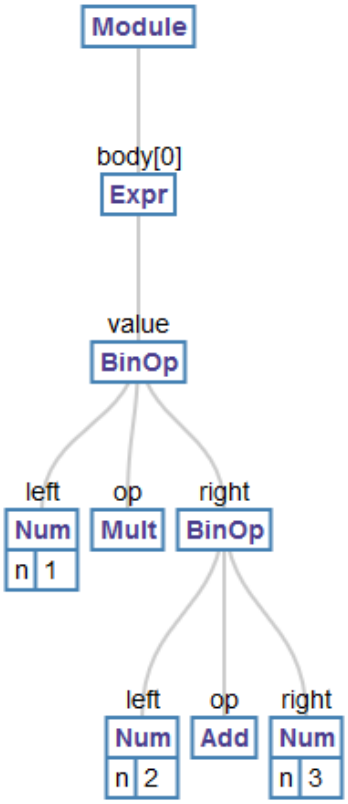


Abbildung 2.9: Python AST Darstellung des Ausdruckes: "$1 * (2 + 3)$"

organisieren. [1]

Eine der einfachsten Topic Model Algorithmen ist LDA.[1] Dieses generative Wahrscheinlichkeitsmodell wurde von Blei et al. [2] vorgestellt. Dokumente werden hier als "Bag of Words" definiert, also als Menge von Wörtern. Der Grundgedanke dieses Algorithmus ist, dass ein Dokument aus mehreren unbekannten Themen besteht, wobei für jedes Thema bestimmte Wörter charakteristisch sind. Obwohl die Menge der betrachteten Dokumente alle den selben Themen zugeordnet werden, ist die Wahrscheinlichkeitsverteilung in jedem Dokument unterschiedlich. Diese Verteilung über den Themen ist von den charakteristischen Wörtern abhängig die im Dokument vorkommen.[1]
Das Modell nimmt an, dass die Themen schon vor den Dokumenten bestehen. Weiterhin wird angenommen, dass die Wörter innerhalb eines Dokuments durch einen drei-stufigen Prozess generiert werden (Abbildung 2.10):

1. Wähle Anzahl an Wörtern N:
   $N \sim Poisson(\xi)$

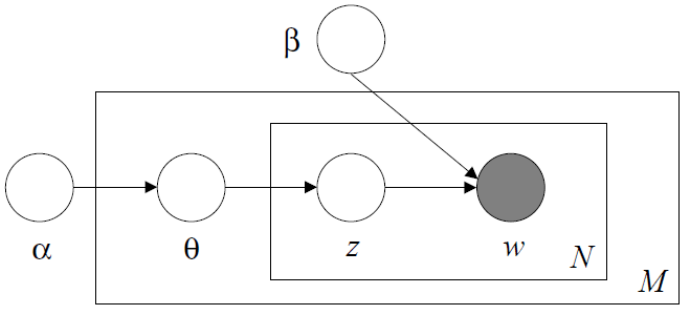


Abbildung 2.10: Graphische Repräsentation des LDA Modells [2]

2. Wähle Variable θ für die Themenverteilung:
   $\theta \sim Dir(\alpha)$

3. Für jedes Wort der N Wörter w_n:
   - Wähle ein Thema z_n:
     $z_n \sim Multinomial(\theta)$
   - Wähle ein Wort w_n aus der Multinomialen Wahrscheinlichkeit abhängig von Thema z_n:
     $p(w_n|z_n, \beta)$

Folgende Annahmen werden in diesem Modell getroffen:

- Die Dimensionalität k der Dirichlet Verteilung wird als festgelegt und bekannt angenommen.
- Die Wort Wahrscheinlichkeiten werden durch eine $k \times V$ Matrix β parametrisiert, wobei $\beta_{ij} = p(w^j = 1|z^i = 1)$ eine feste Größe ist, die es gilt zu schätzen.
- Die Poisson Verteilung wird als keine kritische Annahme gesehen und kann bei Bedarf durch eine andere Verteilung ersetzt werden.

Die einzige bekannte Variable beim LDA ist die Wortverteilung in den Dokumenten. Alle weiteren Variablen zur Themenverteilung und -struktur sind unbekannt und werden geschätzt. Das Problem der Schätzung dieser unbekannten Variablen wird auch als Aposteriori Verteilung bezeichnet. [1]. Die Aposteriori Wahrscheinlichkeit wird wie folgt berechnet:

$$p(\theta, z|w, \alpha, \beta) = \frac{p(\theta, z, w|\alpha, \beta)}{p(w|\alpha, \beta)} \tag{2.1}$$

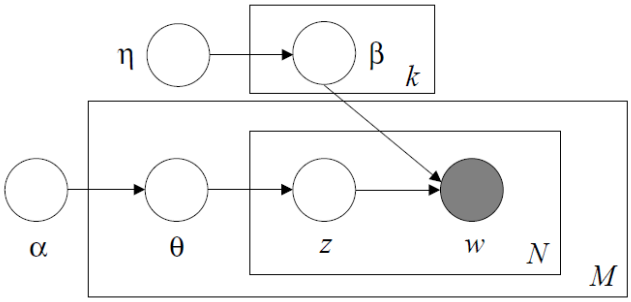


Abbildung 2.11: Graphische Repräsentation des geglätteten LDA Modells [2]

Da diese Verteilung generell unmöglich zu berechnen ist, wird sie durch die Wahrscheinlichkeit $p(w|\alpha,\beta)$ normalisiert[2]:

$$p(w|\alpha,\beta) = \frac{\Gamma(\Sigma_i \alpha_i)}{\Pi_i \Gamma(\alpha_i)} \int (\prod_{i=1}^{k} \theta_i^{\alpha_i - 1})(\prod_{n=1}^{N} \sum_{i=1}^{k} \prod_{j=1}^{V} (\theta_i \beta_{ij})^{\vec{w}_n^j}) d\theta \tag{2.2}$$

Durch die Verknüpfung der Variablen θ und β wird die Funktion unlösbar, was auch Inferenzproblem genannt wird. Dieses Problem entsteht durch die Kanten zwischen θ, z und w (siehe Abbildung 2.10). Werden diese Kanten und der w Knoten entfernt, entsteht ein vereinfachtes Modell mit frei wählbaren Variablen. Diese entstandene Familie von variationalen Verteilungen wird wie folgt charakterisiert:

$$q(\theta, z|\gamma,\phi) = q(\theta|\gamma) \prod_{n=1}^{N} q(z_n|\phi_n) \tag{2.3}$$

Der Dirichlet Parameter γ und die multinomialen Parameter $(\phi_1, .., \phi_N)$ sind hierbei frei wählbar.
Das Modell aus Abbildung 2.10 wird erweitert (Abbildung 2.11). Hierbei wird nun β_i als Zufallsvariable angesehen und der Parameter η wird hinzugefügt. Diese ist ein austauschbarer Skalar für die Wortverteilung je Topic. Durch diese Änderungen entsteht eine Glättung. Ein neues Dokument hat zu hoher Wahrscheinlichkeit auch Wörter, die nicht im Trainings Corpus vorkommen. Normalerweise würden diesen Wörtern eine Null Wahrscheinlichkeit zugewiesen werden. Die Glättung verhindert das jedoch. Das bedeutet, dass auch Wörter, die nicht im Trainingsset vorhanden sind, eine positive Wahrscheinlichkeit haben.

3 MODULE MATCHING

Um eine Aussage über das Nutzerverhalten innerhalb eines Data Science Projektes treffen zu können, ist es notwendig die beigetragene Leistung zu analysieren. Im Rahmen dieser Arbeit erfolgt dies auf Basis des vom einzelnen Nutzern produzierten Source Codes. Im Detail werden die aufgerufenen Funktionen innerhalb eines Jupyter Notebook betrachtet.
Zur syntaktischen Analyse des Python Codes kann wie auch bei literarischen Texten ein Syntax Baum eingesetzt werden. Wie in 2.3.2 beschrieben stellt Python schon ab Installation ein Modul zur Verfügung welches Source Code in einen sogenannten Abstract Syntax Tree (AST) umwandelt.
In dieser Darstellung des Source Codes werden Operationen bestimmten Knotentypen zugewiesen. Unter anderem gibt es auch einen festen Knotentypen für Funktionsaufrufe. Abbildung 3.1 visualisiert die AST Darstellung des Source Codes, welcher in Listing 3.1 gezeigt wird. Der Funktionsaufruf wird durch einen "Call" Knoten repräsentiert. Dem Zugeordnet ist ein "Attribute" Knoten, der Informationen zur aufgerufenen Funktion enthält. Dem folgt ein "Name" Knoten, der das Objekt auf dem die Funktion angewendet wird repräsentiert. Eine Information die jedoch nicht explizit im Call Knoten abgebildet wird, ist das Modul welche die aufgerufene Funktion bereitstellt. Im folgenden wird beschrieben, wie eine solche Verknüpfung zwischen Funktion und Modul hergestellt werden kann. Zudem wird beschrieben wie die Umsetzung eingesetzt werden kann und was die momentanen Limits sind. Die Umsetzung der hier dargestellten Vorgehensweise ist in einem GitHub Repository[1] öffentlich abgelegt.

```
import pandas
df = pandas.DataFrame()
df.head()
```

Listing 3.1: Beispiel Import + Funktionsaufrufe

1 https://github.com/pathol/ipython_project_analyzer in Matchmaker.py

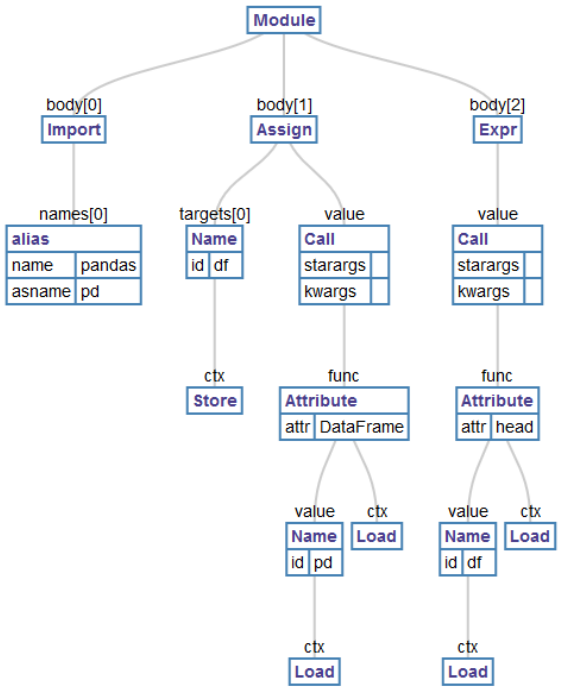


Abbildung 3.1: AST Darstellung eines Import und Funktionsaufruf in Python

3.1 ANFORDERUNGEN

In diesem Abschnitt wird darauf eingegangen, welche Anforderungen an einen Module Matching Ansatz gestellt werden.
Neben dem Import-Aufruf in Abbildung 3.1 gibt es auch weitere Möglichkeiten ein Modul zu importieren. In Abbildung 3.2 werden drei verschiedene Varianten des Imports gezeigt. Im Module Matching müssen sowohl die Imports des Import Knoten als auch des Import-From Knoten erkannt werden. Zudem muss die Verknüpfung auch hergestellt werden, wenn ein Modul unter einem Alias (“import as”) importiert wird.

Auch für Funktionsaufrufe gibt es Unterschiede in der AST-Struktur, wobei der Call Knoten als Indikator die Gemeinsamkeit ist. Abbildung 3.3 zeigt drei einfache Funktionsaufrufe. Beim ersten und letzten Funktionsaufruf ist die Struktur identisch. Unterschiedlich ist jedoch auf welchem Objekt die Funktion aufgerufen wird. Im ersten Aufruf wird direkt auf dem Alias des Pandas Modul die “DataFrame” Funktion aufgerufen. Der Aufruf der “head” Funktion erfolgt jedoch auf dem Resultat der ersten Funktion. Es muss also eine Verknüpfung zwischen den beiden Funktionsaufrufen hergestellt werden um auf das verwendete Modul schließen zu können. Hierfür muss der “As-

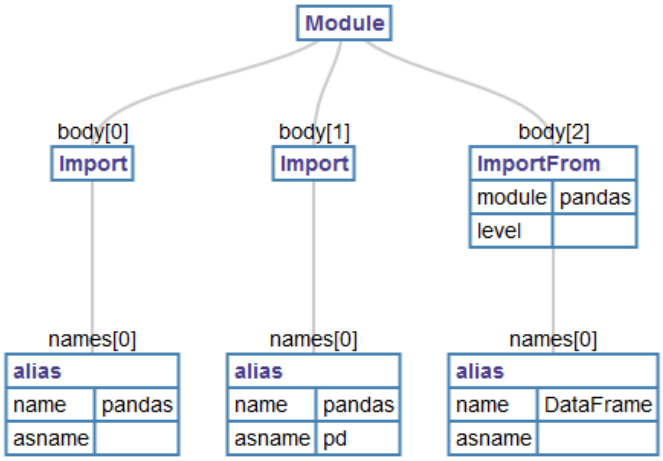


Abbildung 3.2: Verschiedene Formen des Import im AST

sign" Knoten Betrachtet werden. Durch diesen Knoten wird die Variable mit dem Funktionsaufruf verknüpft. Funktionsaufrufe die auf keinem Objekt aufgerufen werden haben eine andere Knoten-Struktur. Der Attribute Knoten fällt weg und der Funktionsname ist im Name Knoten enthalten.

Eine weitere Möglichkeit des Funktionsaufrufes ist ein aneinandergereihter Funktionsaufruf, bei dem direkt an den Rückgabewert der ersten Funktion eine weitere Funktion angewendet wird (Abbildung 3.4). Hierbei folgt auf den Attribute Knoten ein weiter Call Knoten. Erst unten am Baum angekommen ist der Name Knoten abgebildet. Auch in solch einem Fall muss die Struktur richtig verarbeitet und die wesentlichen Informationen extrahiert werden.
Aus einer gegebenen Python Source Code Datei müssen demnach alle Imports und Funktionsaufrufe gesammelt und abgeglichen werden um die Zuordnung durchzuführen. Des weiteren soll eine Liste von allen Verknüpfungen als Endergebnis ausgegeben werden.

Hieraus ergeben sich folgende Anforderungen an das Module Matching:

1. Es müssen verschiedene Formen des Imports korrekt erkannt werden.

2. Es müssen verschiedene Formen des Funktionsaufrufes korrekt erkannt werden.

3. Eine Verknüpfung zwischen importierten Modulen und aufgerufenen Funktionen muss hergestellt werden.

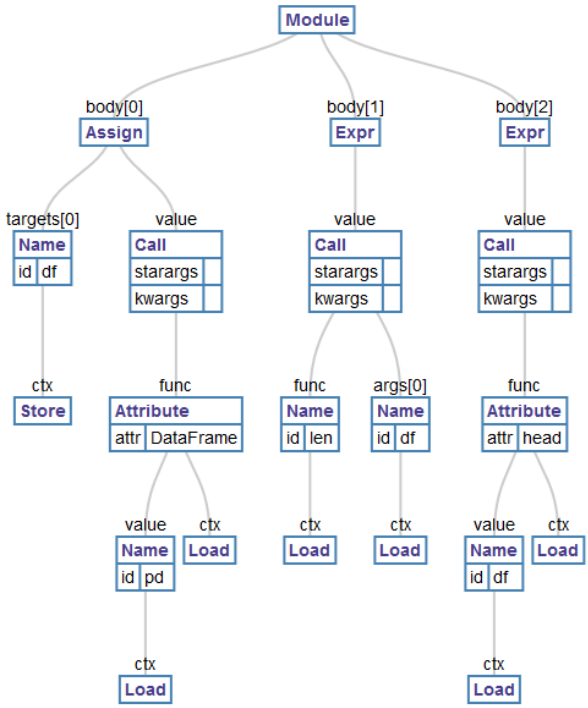


Abbildung 3.3: Verschiedene Formen des Funktionsaufruf im AST

4. Auch Funktionsaufrufe die nicht direkt von dem Import-Objekt aufgerufen werden müssen zugeordnet werden.

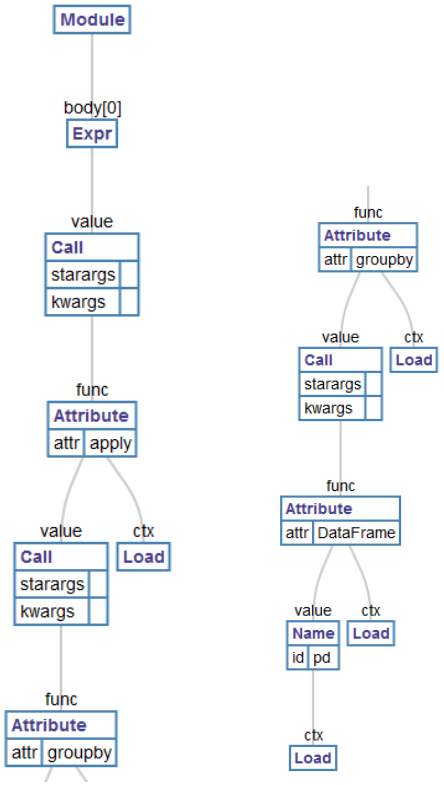
5. Es soll Python Source Code verarbeiten und eine Liste mit den Zuordnungen zurückgeben.

Im folgenden Abschnitt wird der eigene Ansatz vorgestellt und darauf eingegangen wie die gestellten Anforderungen erfüllt werden.

3.2 EIGENER ANSATZ

Das Grundgerüst der Implementierung bildet die "NodeVisitor" Klasse des AST [2] Moduls. Dies ist eine Basis Klasse die den Syntaxbaum entlangläuft und eine generische Visitor Funktion für jeden gefunden Knoten aufruft. Laut der offiziellen Dokumentation[3] ist es vorgesehen, dass eine Subklasse erstellt wird, in der die Visitor Funktion spezifisch nach den eigenen Bedürfnissen implementiert wird. Neben der generischen Visitor Funktion können auch Visitor Funktionen für spezifische Knotentypen erstellt werden.
Bei der Initialisierung der hierfür erstellten Subklasse werden zunächst

2 https://docs.python.org/3/library/ast.html
3 https://docs.python.org/3/library/ast.html#ast.NodeVisitor



(a) Oberer Teil des Aufrufes (b) Unterer Teil des Aufrufes

Abbildung 3.4: AST Darstellung des Aufrufes pd.DataFrame().groupby().apply()

die Objekte initialisiert, die in den jeweiligen Visitor Funktionen befüllt werden sollen. Hierfür werden pandas DataFrames[4] verwendet. Insgesamt werden vier DataFrames initialisiert:

- mods: Verwendete Imports
- calls: Funktionsaufrufe
- assign: Variablenzuweisungen die einen Funktionsaufruf beinhalten
- matchs: Zuordnungen zwischen Funktionsaufruf und Imports

Die spezifischen Visitor Funktionen sind alle nach dem gleichen Schema aufgebaut. Der Name der Funktion folgt dem Schema "visit_<Knotenname>" und bekommt als Variablen einmal die Klasse selbst und den zu betrachteten Knoten übergeben. Um auch die Kinder Knoten des betrachteten Knoten entlangzulaufen wird innerhalb der Funktion am Ende die generische Visitor Funktion aufgerufen.

3.2.1 *Import/ImportFrom*

In den Visitor Funktionen für Import bzw. ImportFrom Knoten werden die Daten über die Imports gesammelt. Sobald ein solcher Knoten gefunden wurde werden sowohl Name als auch falls vorhanden der Alias in dem mods DataFrame gespeichert. Zusätzlich wird bei dem ImportFrom Knoten die Information über die importierte Klasse gespeichert. Hierdurch wird die Anforderung 1 erfüllt.

3.2.2 *Call*

Um die verschiedenen Funktionsaufrufe aus Abbildung 3.3 und 3.4 erkennen zu können, muss in der "visit_Call" Funktion zwischen den Fällen unterschieden werden. Zunächst wird differenziert, ob dem Call Knoten ein Name oder Attribute Knoten folgt. Folgt ein Name Knoten, also die Funktion wird nicht von einem spezifischen Objekt aufgerufen, wird nur der Funktionsname und die Zeile in der die Funktion aufgerufen wird in dem calls DataFrame gespeichert.
Folgt jedoch ein Attribute Knoten soll neben Name und Zeile auch der Name des Objektes auf den die Funktion aufgerufen wird gespeichert werden. Ist der nächste Knoten nun ein Name Knoten wird aus diesem der Objektname entnommen und mit Funktionsname und Zeilennummer im DataFrame abgespeichert. Ist dies aber nicht der Fall,

4 https://github.com/pandas-dev/pandas

wird der Struktur gemäß Abbildung 3.4 gefolgt bis der Name Knoten erreicht wird. Dies ist jedoch nur erfolgreich, wenn diese Struktur beim Funktionsaufruf eingehalten wird. Bei komplexeren Funktionsverkettungen ändert sich die Struktur der Knoten. In dieser Implementierung werden komplexe Funktionsverkettungen jedoch nicht berücksichtigt, da hierfür zu viele verschiedene Fälle abgedeckt werden müssten. Anforderung 2 wird hierdurch dennoch größtenteils erfüllt.

3.2.3 *Assign*

Wobei die Visitor Funktion alle Assign Knoten betrachtet, sind nur diejenigen von Interesse, bei denen das Ergebnis eines Funktionsaufrufes einer Variable zugewiesen wird. Dies wird anhand der Existenz eines Call Knotens überprüft.
Die Informationen über Funktionsnamen und Objekt welches die Funktion aufruft werden über die gleiche Abfrage wie in 3.2.2 erlangt. Zusätzlich neben all den Informationen die auch beim Call Visitor erhoben werden, wird hier zusätzlich der Namen der Variable zu der die Zuweisung erfolgt gespeichert. Mit Hilfe dieser zusätzlichen Information der Zuweisungen kann die Anforderung 4 erfüllt werden.

3.2.4 *Matching*

Nachdem der gesamte AST entlanggelaufen und alle Informationen von Interesse durch die Visitor Funktionen extrahiert wurden, können diese nun miteinander verknüpft werden. Das Matching erfolgt in mehreren Schritten, wobei alle zuvor gesammelten Daten zum Einsatz kommen.
Zunächst werden die einzelnen Funktionsaufrufe mit den Informationen der Imports abgeglichen. Es wird überprüft ob der Name des Objektes der die Funktion aufgerufen hat mit den Modulnamen, Alias oder Namen der importierten Klassen übereinstimmen. Ist dies der Fall, kann die Funktion direkt dem Modul zugewiesen werden.
Ist dies jedoch nicht der Fall werden die Zuweisungen, die vor dem Funktionsaufruf stattgefunden haben, in Betracht gezogen. Hierbei wird der Name des Objektes der die Funktion aufruft mit dem Namen der zugewiesenen Variable abgeglichen. Nun wird beim Funktionsaufruf in der Zuweisung der Name des Objektes mit den Einträgen der Module abgeglichen. Wenn es eine Übereinstimmung gibt, wird das Modul und gegebenenfalls die Klasse mit der Funktion verknüpft und im DataFrame matchs gespeichert.
Dieser Schritt wiederholt sich bis eine Übereinstimmung gefunden

wurde oder alle Objekte überprüft wurden. Sollte kein Matching stattfinden, wird die Funktion ohne Modul- und Klassennamen abgespeichert. Die Anforderung 3 wird somit erfüllt.

3.3 ANWENDUNG

Wie nun die genaue Anwendung der oben beschriebenen Klasse erfolgt, wird im Listing 3.2 aufgeführt. Der zu untersuchende Source Code wird in einer String Variable abgespeichert. Dieser String wird nun über das AST Modul in einen AST geparsed. Nachdem nun die selbst geschriebene Klasse Matchmaker initialisiert wurde, wird der AST durch die "visit" Funktion durchlaufen. Die Visitor Funktionen werden automatisch aufgerufen und die DataFrames werden befüllt. Daraufhin erfolgt das Module Matching durch Aufruf der "matching" Funktion und gibt nach Beendigung des Prozesses den match DataFrame zurück. Anforderung 4 wird somit ebenfalls erfüllt.

```
import ast
source = """import pandas as pd
df = pd.DataFrame()
df.head()"""
tree = ast.parse(source)
mm = Matchmaker()
mm.visit(tree)
matchs = mm.matching()
```

Listing 3.2: Anwendung des Module Matchings

3.4 LIMITS

Wie schon in den Beschreibungen der Funktionen erwähnt, werden in dieser Implementierung nur bestimmte vorgegebenen Funktionen und Imports berücksichtigt. Obwohl in der Regel Imports über einen der in Abbildung 3.2 gezeigten Varianten erfolgen, gibt es noch weitere Möglichkeiten Module zu importieren. Eine weitere Methode ist die Verwendung des Importlib [5] Moduls. Module die hierüber importiert werden, werden nicht berücksichtigt.
Des weiteren werden auch nicht alle Funktionen erkannt. Nur Funkionen die dem Schema aus Abbildung 3.3 und 3.4 folgen werden erfolgreich erkannt. Für komplexere Funktionsaufrufe, die z.B. zwischen den multiplen Aufrufen noch Indizes verwenden, kann kein Eintrag angelegt werden. Im Ausblick auf eine Weiterentwicklung des Mo-

5 https://docs.python.org/3/library/importlib.html

duls können diese Grenzen jedoch überschritten werden, indem noch mehr spezifische Folgen von Knotentypen in Betracht gezogen werden.

4

DATEN

Als Grundlage für diese Arbeit werden öffentlich zugängliche Projekte auf GitHub betrachtet. Hierbei werden speziell nur die untersucht, die Jupyter Notebook Dateien beinhalten. Dies basiert auf der Annahme, dass in Jupyter Notebooks hauptsächlich Data Science relevante Projekte entstehen.[7]
Um die Namen der Repositories zu erhalten, die Jupyter Notebooks enthalten wurde der öffentlich zugängliche GitHub Datensatz auf der Plattform Google BigQuery[1] verwendet. Anhand der Informationen des Datensatzes wurden die relevanten Repositores von GitHub extrahiert und im weiteren analysiert (Abbildung 4.1).

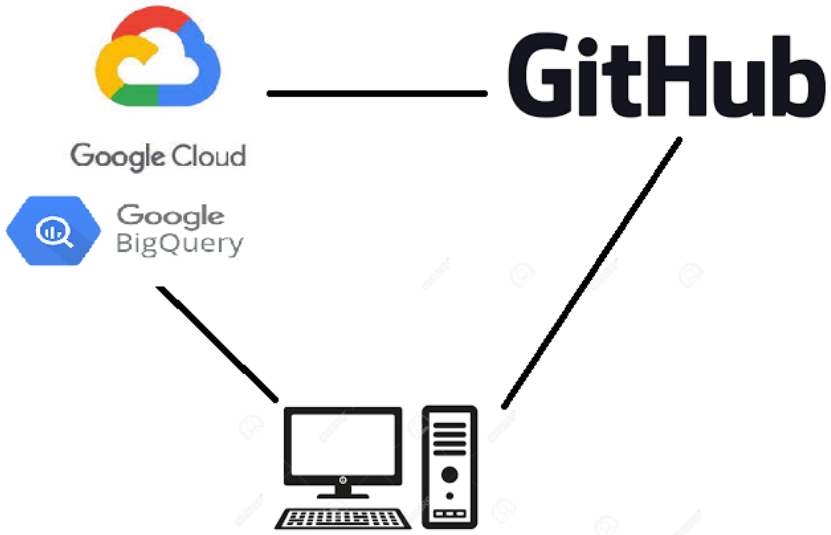


Abbildung 4.1: Datenarchitektur

4.1 GOOGLE BIG QUERY

BigQuery ist ein Dienst der Google Cloud der Speicherung und Verarbeitung von großen Datensätzen ermöglicht. Neben vielen anderen

1 https://bigquery.cloud.google.com/dataset/bigquery-public-data:github_repos?pli=1

öffentlichen Datensätzen, wird einem auch ein GitHub Datensatz mit allen öffentlich zugänglichen Repositories zur Verfügung gestellt.
Die Daten liegen in relationaler Tabellenform vor und können durch die Query-Sprache SQL abgerufen und in ein eigenes Projekt importiert werden.
Durch Ausführen der Query A.1 werden Informationen über Jupyter Notebooks im ".ipynb" Format gesammelt und anschließend in eine eigene Tabelle gespeichert. Anschließend werden die dazugehörigen Commit-Daten durch die Query A.2 gesammelt (Stand 13.06.2019).

Durch die Queries wurden nun zwei Tabellen erstellt (Abbildung 4.2). Die Tabelle "content_ipynb" enthält die Daten aller Jupyter Notebooks vom Datentyp ".ipynb". Die relevanten Variablen sind hierbei "content", "sample_repo_name", "sample_path". In diesen Variablen sind Namen des Repositories, Pfad der Datei und auch der gesamte Inhalt der Datei gespeichert.
Die Tabelle "commits_ipynb" enthält alle Daten der Commits. Alle Variablen des Typen "record" enthalten weitere Untervariablen. In Abbildung 4.2 sind zu Anschauungszwecken nur die Untervariablen von Interesse aufgeführt. Die Variable "commit" beinhaltet den Hash-Wert der für jeden Commit einzigartig ist. Zu jedem Commit ist der Autor und die Commit Nachricht enthalten. Über die Variable "repo_name" und "difference_new_path" bzw. "difference_old_path" ist nachvollziehbar welche Datei in welchem Repository durch diesen Commit geändert wurde. Weitere Variablen werden in dieser Arbeit nicht betrachtet.

Ingesamt werden hierbei 144.578 Jupyter Notebooks betrachtet, die mit 921.546 Commits verändert wurden.

Diese gesammelten Notebooks wurden sowohl von nur einer einzelnen Person erstellt, als auch in Zusammenarbeit. Da im Rahmen dieser Arbeit die Funktionsteilung in Projekten untersucht wird, sollen alle Notebooks herausgefiltert werden, die nicht in Zusammenarbeit entstanden sind.
Hierfür wurden über die Autoren Email Adresse der einzelnen Commits die Anzahl der Autoren pro Jupyter Notebook ermittelt (A.3,A.4). Anschließend wurden alle Commit-Daten von Notebooks mit nur einem Autor verworfen (A.5).
Nach dem Filtern sind nur noch 33.631 Jupyter Notebooks übrig, die durch 317.060 Commits geändert wurden. Die Daten dieser Commits

wurde durch Listing A.6 extrahiert und steht im GitHub Repository[2] zur Verfügung.
Folgende Datenstruktur ergibt sich durch diese Query:

- commit: Hash-Wert des Commits
- author: Email-Adresse des Autors des Commits
- path: Dateipfad des Notebooks
- repo: Name des GitHub Repositories
- contributors: Anzahl an Mitwirkenden des Notebooks

4.2 GITHUB

In den Github Daten auf Google BigQuery sind zwar Informationen über Commits und der Inhalt der Notebooks enthalten, Informationen darüber welcher Autor den jeweiligen Code geschrieben hat existiert aber nicht.
Da in diesem Datensatz keine Informationen über die Änderungen in den Commits enthalten ist, muss diese Information direkt von GitHub extrahiert werden. Um Daten über Commits direkt von GitHub zu extrahieren sind mehrere Möglichkeit geboten. Eine Möglichkeit ist das Extrahieren der Daten über die Weboberfläche, bzw. über die Web-API[3]. Anhand der gegebenen Daten über Commit-Hash, Repository und veränderten Datei, lassen sich über die API via REST die Commit Änderungen extrahieren[4]. Dies hat jedoch den Nachteil, dass zum einen ein Anfrage-Limit pro Stunde gesetzt ist und zum anderen zu große Änderungen nicht dargestellt sind.
Eine weitere Möglichkeit besteht darin, die entsprechenden Repositories lokal zu klonen. Der Vorteil hierbei ist, dass kein Anfrage-Limit vorgegeben ist und auch große Änderungen extrahiert werden können. Aufgrund dieser Vorteile werden die Repositories lokal abgespeichert und im weiteren verarbeitet.

4.2.1 *Pydriller*

Aus den nun lokal gespeicherten Repositories soll nun programmatisch die Informationen über die Commit Änderungen extrahiert wer-

2 https://github.com/pathol/ipython_project_analyzer in /data/commits_bigquery.csv
3 https://developer.github.com/v3
4 https://github.com/pathol/ipython_project_analyzer in /notebooks/commit_data_analysis.ipynb

Abbildung 4.2: Datenmodell Github

den. Um dies mit Python zu erreichen wurde das Modul Pydriller[5] verwendet[9].
Zur Verwendung dieses Moduls wird falls vorhanden auf ein lokal gespeichertes Repository zugegriffen. Falls dies nicht der Fall ist, wird das Repository temporär heruntergeladen. Nach dem Laden des gewünschten Repositories gibt es nun mehrere Möglichkeiten über die vorhandenen Commits zu iterieren. Da der Hash-Wert der Commits bekannt ist, können diese gefiltert werden.
Innerhalb jedes Commits kann nun durch alle geänderten Dateien iteriert werden. Listing 4.1 zeigt wie dies grundlegend aussieht.

```
from pydriller import RepositoryMining
commits = [<list of all commits>]
files = [<list of all files>]
repository = RepositoryMining(repository, only_no_merge=True,
    only_commits=commits)
for commit in repository.traverse_commits():
    for file in commit.modifications:
        if file in files:
                        #extract changes
```

Listing 4.1: Anwendungsbeispiel Pydriller

Innerhalb dieser Iterationen können viele verschiedene Informationen extrahiert werden. Neben dem Source Code vor und nach dem Commit kann auch direkt eine Differenz ausgegeben werden.

4.2.2 *Verarbeitung*

Wie in Abschnitt 2.1 beschrieben liegt eine Ipython Datei im JSON-Format vor. Somit sind in den Commit Änderungen nicht nur reiner Code enthalten, sondern auch geänderte JSON-Struktur, z.B. wenn ein neuer Codeblock hinzugefügt wird. Da in einer Änderungsdatei jedoch nur die geänderten Zeilen sind, ist meistens keine gültige JSON-Struktur gebildet, was die Verarbeitung erschwert.
Um diese JSON-Struktur zu verlieren und nur den Python Code zu erhalten, kann wie in Listing 4.2 gezeigt ein Jupyter Notebook in eine Python Datei umgewandelt werden. Ein weiterer Vorteil dieser Umwandlung ist, dass Ipython spezifische Funktionen auch umgewandelt werden, dass diese in einem AST korrekt interpretiert werden können.

```
from nbformat import reads, NO_CONVERT
```

5 https://github.com/ishepard/pydriller

```
from nbconvert import PythonExporter
def code_extractor(jpt):
    nb = reads(jpt, NO_CONVERT)
    exporter = PythonExporter()
    source, meta = exporter.from_notebook_node(nb)
    return stripComments(source)
```

Listing 4.2: Konvertierung von Ipython zu Python

Da Pydriller den kompletten Source Code vor und nach dem Commit ausgibt, können diese jeweils in Python Source umgewandelt werden. Aus diesen beiden Dateien, kann nun selbst eine Differenz gebildet werden um die durch den Commit hinzugefügten Zeilen zu extrahieren.
Nun kommt das in Kapitel 3 beschriebene Module Matching zum Einsatz. Da hierfür sowohl die geänderten Funktionen als auch importierten Module notwendig sind, wird dies auf dem Source Code nach dem Commit angewendet. Der nun in Python vorliegende Source Code wird in einen AST umgewandelt. Hierauf wird das Module Matching angewendet und alle Funktionen mit den zugeordneten Modulen und Klassen wird ausgegeben.
Diese enthalten jedoch alle Funktionen der gesamten Datei, nicht nur die Änderungen. Daher wird diese Liste mit den geänderten Zeilen abgeglichen und nur die geänderten Funktionen mit Zuordnungen bleiben bestehen. Diese Funktionen die im Commit hinzugefügt wurden werden anschließend mit dem Autor des Commits verknüpft und abgespeichert. Diese komplette Umsetzung ist im GitHub Repository[6] enthalten.

4.2.3 *Datenstruktur*

Die nun erzeugte Zuweisung von Funktionen zu Autoren ist im GitHub Repository[7] vorhanden.
Die daraus entstandenen Daten haben folgende Struktur:

- line: Zeile im Dokument
- function: Name der Funktion die aufgerufen wurde
- package: Name des Moduls zu der die Funktion gehört (falls vorhanden)

6 https://github.com/pathol/ipython_project_analyzer in /notebooks/repo_function_matching.ipynb
7 https://github.com/pathol/ipython_project_analyzer in /data/repo_function_matched.csv

- class: Name der Klasse zu der die Funktion gehört (falls vorhanden)
- author: Email Adresse des Autors der Änderung
- commit: Hash-Wert des Commits
- code: Eigentliche Zeile in der die Funktion aufgerufen wird

Hieraus sind 1.077.910 aufgerufene Funktionen entstanden, wobei 453.636 erfolgreich einem Paket zugeordnet wurden.

5 ANALYSE NUTZERVERHALTEN

In den vorherigen Kapiteln wurde beschrieben wie der Source Code aus den Commit Daten extrahiert und den Autoren zugeordnet wurde. Nun soll anhand dieser Daten Nutzerverhalten in Projekten abgeleitet werden. Hierzu muss festgelegt werden nach welchem Kriterium diese Daten analysiert werden.
Zum Zwecke dieser Analyse wird ein Topic Modeling (2.4) durchgeführt. Die Funktionsnamen werden hierbei als Text zum Clustering verwendet. Es ist festzulegen, wie ein Dokument definiert wird. Dies bestimmt in welchem Kontext die aufgerufenen Funktionen betrachtet werden.

5.1 DOKUMENT DEFINITION

Jupyter Notebooks haben die Besonderheit, dass mehrere Blöcke erstellt werden können. Code in diesen Blöcken kann individuell getrennt von den anderen Blöcken ausgeführt werden. Dies legt die Überlegung nahe, dass Code der gemeinsam in einen Block auftritt zum Ziel des gleichen Use Cases beiträgt. Somit kann ausgesagt werden, wenn die gleichen Funktionen vermehrt miteinander im gleichen Block auftreten, dass sie zu einem Themenbereich gehören.
Dieser Ansatz wurde weiter verfolgt und getestet[1]. Aus allen Jupyter Notebooks wurden die Funktionsnamen extrahiert und nach Codeblöcken getrennt gespeichert. Hierbei wird im Sinne des Topic Modelings ein Codeblock als Dokument definiert. Anhand der Zuweisung von Funktion zu Themen werden Nutzer nach Rollen kategorisiert.
Nach der Extraktion der verwendeten Funktionen ergeben sich 138.944 Blöcke. Doch wie Tabelle 5.1 zeigt, sind in einem Großteil der Blöcke nur ein einzelner Funktionsaufruf.
Selbst wenn alle Blöcke mit nur einem Funktionsaufruf herausgefiltert werden, sind in den übrigen Blöcken nur wenige Funktionsaufrufe (Tabelle 5.2). Da hier pro Dokument nur wenige Funktionen zugeordnet werden, ist hier ein Topic Modeling nicht sinnvoll.

Da dieser Ansatz nicht sinnvoll scheint, wird eine andere Dokumentendefinition verwendet. In Abschnitt 4.2 wird ein Datensatz erzeugt,

1 https://github.com/pathol/ipython_project_analyzer in /notebooks/function_topics.ipynb

min	25%	50%	75%	max
1	1	1	3	194

Tabelle 5.1: Quantil-Verteilung der Anzahl von Funktionen per Block

min	25%	50%	75%	max
2	2	3	5	194

Tabelle 5.2: Quantil-Verteilung der Anzahl von Funktionen per Block mit mehr als einer Funktion

der Funktionsaufrufe Nutzern zuweist. Im Rahmen dieser Arbeit gilt zu untersuchen, ob eine Rollenverteilung in Data Science Projekten stattfindet. Aufbauend auf dieser These kann eine weitere formuliert werden: Wenn ein Nutzer eine feste Rolle in einem Projekt annimmt, nimmt er sie auch in anderen Projekten an.
Somit wird ein Dokument als Autor definiert. Jede vom Nutzer aufgerufene Funktion wird seinem Dokument zugeordnet. Somit ergeben sich 7583 Dokumente mit einem Median von 49 Funktionsaufrufen (Tabelle 5.3).

Des weiteren wird eine neue Variable abgeleitet, die sich aus Funktions-, Modul- und Klassennamen ergibt. Die neue Variable "combined" hat die Struktur "package:class:function", sofern eine Zuweisung zu Modul und/oder Klasse getroffen werden konnte.

5.2 DURCHFÜHRUNG

Das Topic Modeling wirt anhand des Python Moduls Gensim[2] durchgeführt. Zunächst muss ein Wörterbuch mit allen Funktionsnamen hergestellt werden. Bei der Analyse von normalem Text würde in der Regel vor diesem Schritt eine Liste von Stop Words verwendet um diese aus den Dokumenten zu entfernen. Da hier jedoch Funktionsnamen betrachtet werden, wird hierauf verzichtet. Weiterhin werden aus dem Wörterbuch die Extremwerte herausgefiltert. Dies geschieht anhand der Häufigkeit innerhalb der Dokumente. Es kann eingestellt werden in wie vielen Dokumenten eine Funktion vorkommen soll um

2 https://github.com/RaRe-Technologies/gensim

min	25%	50%	75%	max
1	13	49	139	16332

Tabelle 5.3: Quantil-Verteilung der Anzahl von Funktionen per Nutzer

fürs Topic Modeling relevant zu sein. Hierfür wird festgelegt, dass alle Funktionen verworfen werden, die von weniger als 10 Personen verwendet wurden. Des weiteren können zu häufig verwendete Funktionen herausgefiltert werden. Hierfür wird festgelegt, dass nur Funktionen betrachtet werden, die von weniger als 40% aller Personen verwendet wurden. Diese Werte wurden frei basierend von vorherigen Verwendungen ausgewählt. Im Wörterbuch sind nun 2359 Wörter enthalten.
Nachdem das Wörterbuch von Extremwerten befreit wird, wird ein sogenannter Corpus gebildet. Hierbei werden die Dokumente in ein "Bag of Words" (BoW) umgewandelt. Hierbei werden Wörter durch Tupel ersetzt, die die zugehörige Wort-Id aus dem Wörterbuch und die Häufigkeit des Vorkommens enthalten.
Mit dem Wörterbuch und Corpus kann nun ein Topic Modell erstellt werden. Dieses Modell wird mit Hilfe von LDA (Abschnitt 2.4) erstellt. Hierbei können Parameter zum Alpha-Wert und Anzahl der Themen angegeben werden. Der Alpha Wert beschreibt die a-priori Wahrscheinlichkeit für jedes Thema. Wird die Variable nicht gesetzt, wird standardmäßig von einer normalisierten Wahrscheinlichkeit von $\frac{1}{AnzahlTopics}$ ausgegangen. Diese a-priori Wahrscheinlichkeit wird im folgenden verwendet.
Die optimale Anzahl von Topics muss individuell bestimmt werden. Eine Möglichkeit besteht darin, für verschiedene Anzahlen an Topics Modelle zu erstellen und die Verteilung der Wörter und Überschneidungen zu betrachten. Dies erfolgt jedoch subjektiv und nach individueller Präferenz. Eine weitere Möglichkeit ist eine Metrik festzulegen, für eine Reihe an Topic-Nummern LDA-Modelle zu entwickeln und anhand der Metrik eine Entscheidung zu treffen.
Als Metrik des Topic Modeling wird Topic Coherence verwendet. Dies ist ein Gütemaß darüber, wie interpretierbar die Themenverteilung ist. Hierfür wird das Maß der semantischen Ähnlichkeit zwischen der prägnantesten Wörter jedes Topics evaluiert. [6] Um die optimale Anzahl an Topics zu erhalten, werden Topic Modelle mit 2 bis 20 Themen berechnet. Abbildung 5.1 zeigt den Verlauf des Coherence Scores anhand der "c_v" Metrik.

Je höher der Wert, desto besser das Topic Modeling. Wenn jedoch zu viele Themen gewählt werden, besteht ein hohes Risiko, dass die gleichen Wörter prägnant für mehrere Topics sind. Anhand der Werte ist 3 die optimale Anzahl an Themen.

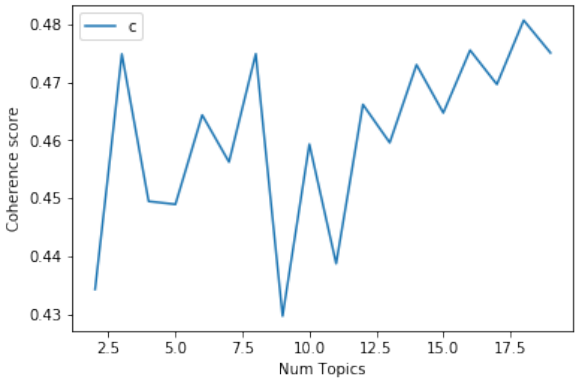


Abbildung 5.1: Graph der Koherenz des Topic Modeling

5.3 AUSWERTUNG

Abbildungen 5.2, 5.3, 5.4 zeigen die Wortverteilungen sortiert nach dem höchsten Vorkommen innerhalb des Topics. Es ist zu sehen, dass Thema 3 sich hauptsächlich durch Funktionen des Moduls Tensorflow auszeichnet. In Themen 1 und 2 sind Funktionen der Pakete Matplotlib, Numpy und Pandas am aussagekräftigsten.

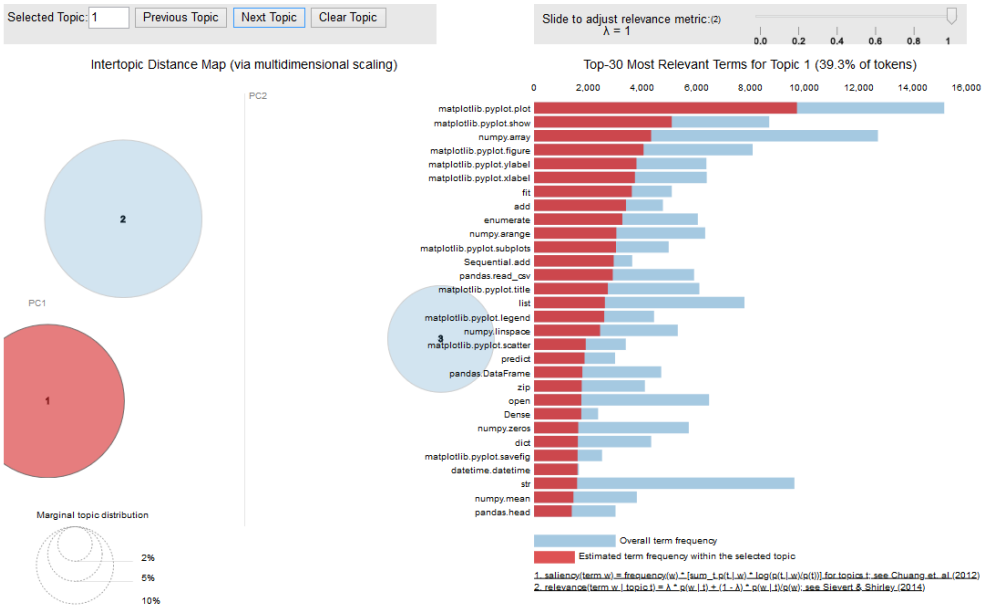


Abbildung 5.2: Wortverteilung für Thema 1

Kommen bestimmte Funktionen generell häufig im Corpus vor, sind sie meistens auch aussagekräftig für mehrere Themen. Verändert man jedoch den Gamma-Wert der Auflistung, sieht man die Funktionen, die hauptsächlich nur in diesem Thema vorkommen. Abbildungen

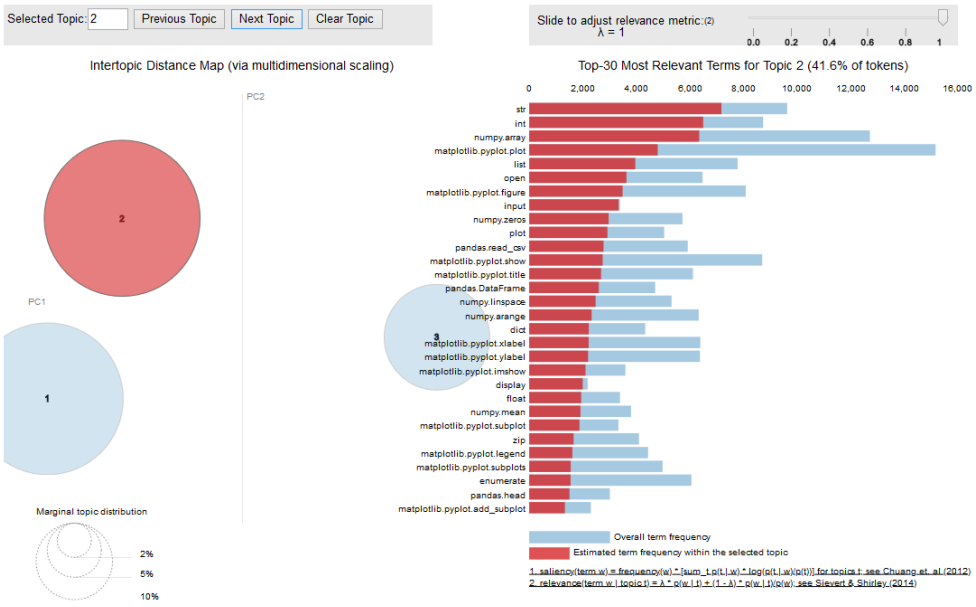


Abbildung 5.3: Wortverteilung für Thema 2

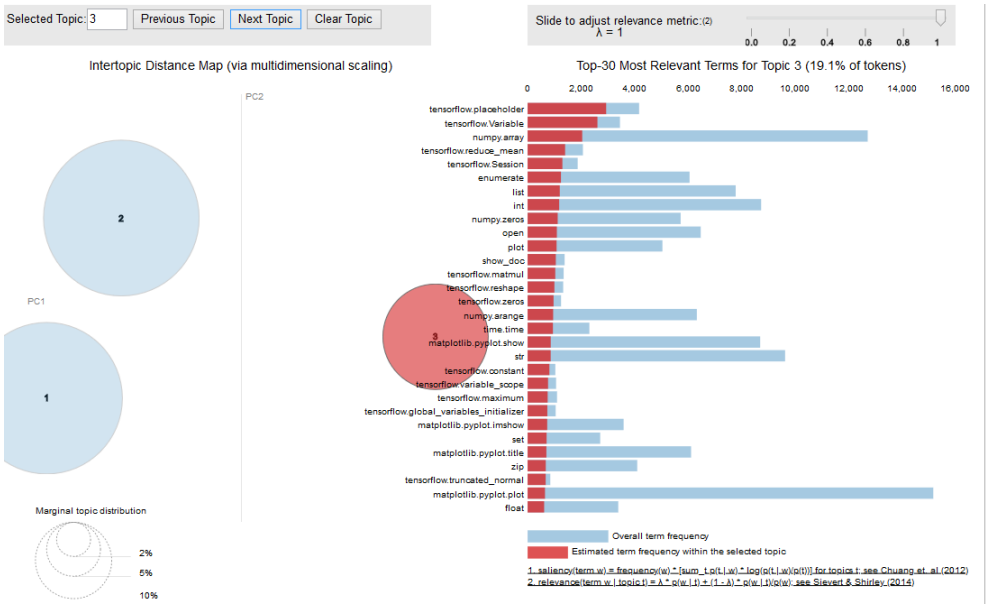


Abbildung 5.4: Wortverteilung für Thema 3

5.5, 5.6, 5.7 zeigen diese Wortverteilungen.
Bestimmte Klassifikatoren, Preprocessing, sowie 2D-Objekte finden sich hier hauptsächlich wieder. In Thema 2 werden hauptsächlich Graphen zugeordnet. Thema 3 ist breit gemischt mit Unit Tests und Helper Funktionen. Das Topic Modeling ist jedoch sehr stark vom Modul Matplotlib geprägt.

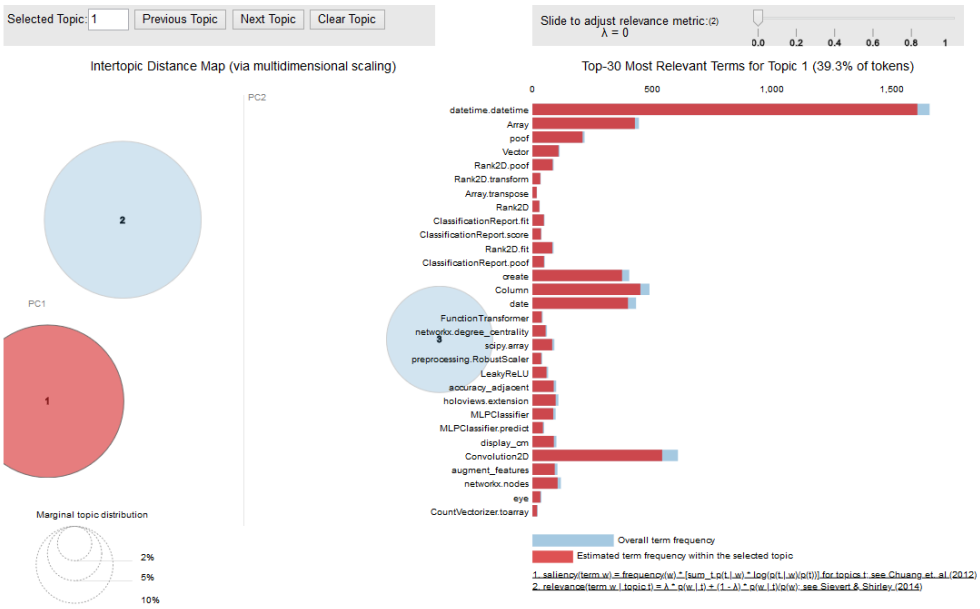


Abbildung 5.5: Wortverteilung für Thema 1 mit verändertem Gamma-Wert

Da Matplotlib die tragende Funktion ist, wird dies herausgefiltert und ein neues Topic Modeling durchgeführt. Auch für diese Wortverteilung wurden Kohärenz Werte gebildet. Abbildung 5.8 zeigt die Verteilung. Aufgrund der Daten wird im folgenden ein Topic Modeling mit 5 Themen durchgeführt. Abbildung 5.9, 5.10, 5.11, 5.12, 5.13 zeigen die Ergebnisse des topic Modeling. Themen 1,4 und 5 sind sehr stark auf die Module Pandas und Numpy fixiert. Thema 3 enthält hauptsächlich Tensorflow Funktionen und Thema 2 ist breit gefächert.

Das Topic Modeling zeigt, dass eine Verteilung bei der Benutzung von Funktionen auftritt. Jedoch ist diese nicht klar in Themen bzw. Rollen zu unterteilen und häufig verwendete Funktionen bilden die Mehrheit an Kriterien für ein Thema.

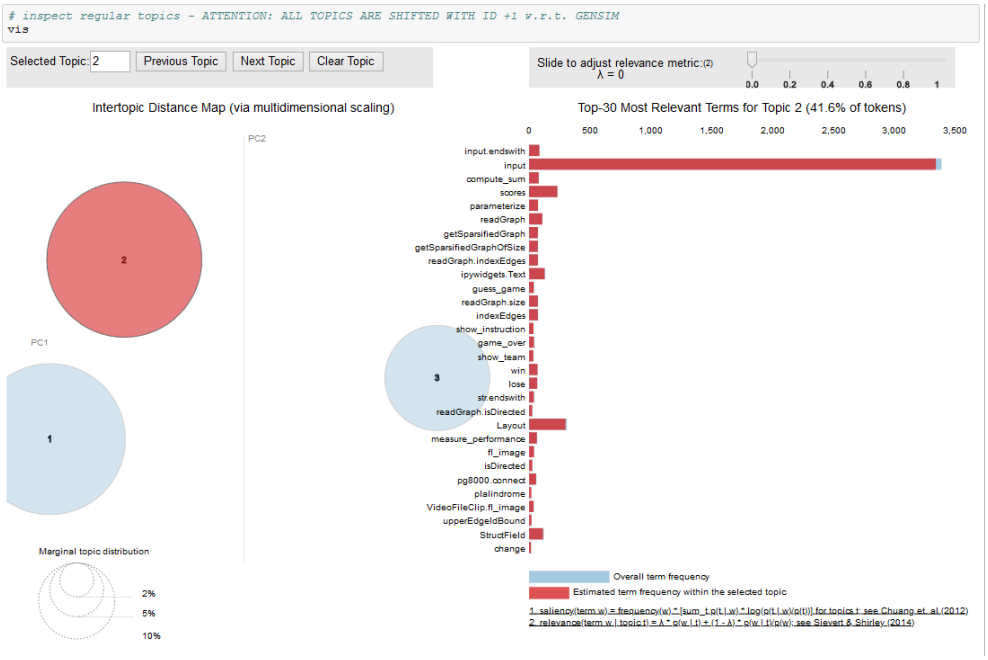


Abbildung 5.6: Wortverteilung für Thema 2 mit verändertem Gamma-Wert

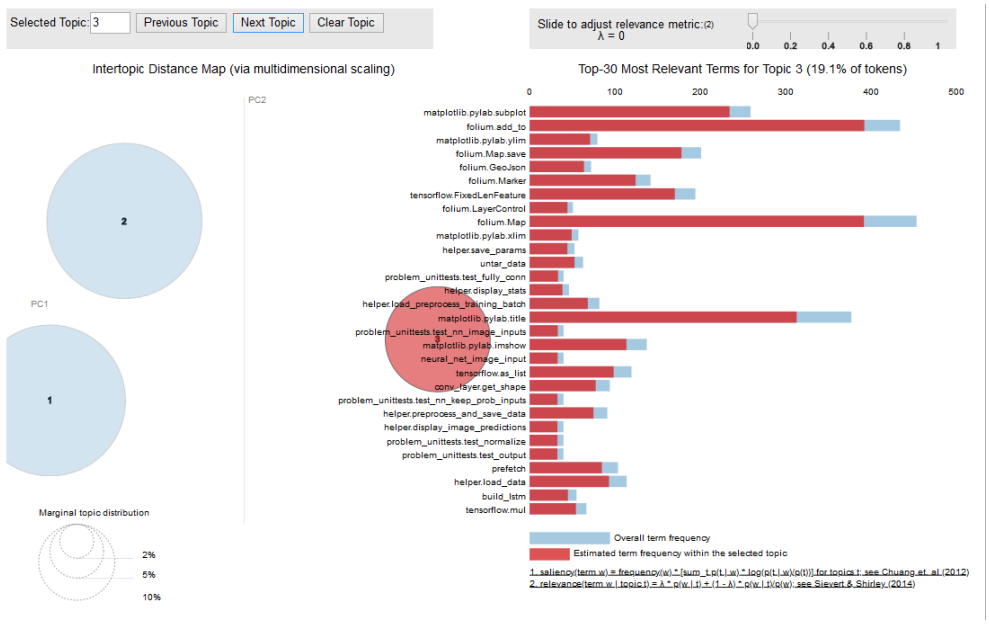


Abbildung 5.7: Wortverteilung für Thema 3 mit verändertem Gamma-Wert

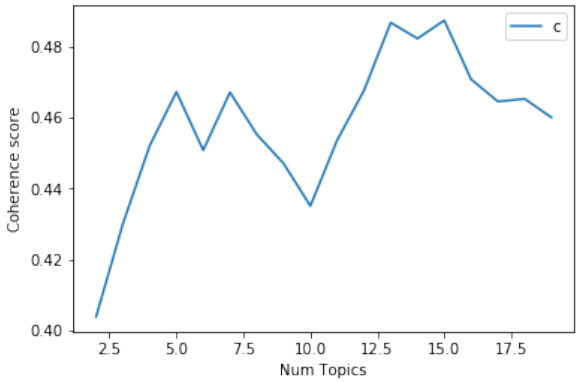


Abbildung 5.8: Graph der Koherenz des Topic Modeling ohne Matplotlib

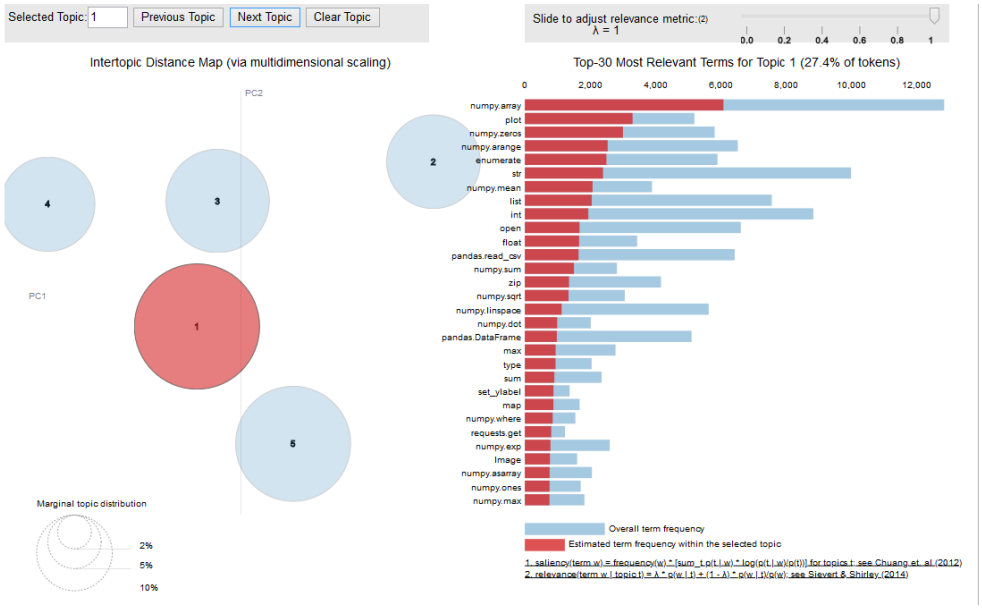


Abbildung 5.9: Wortverteilung für Thema 1 ohne Matplotlib

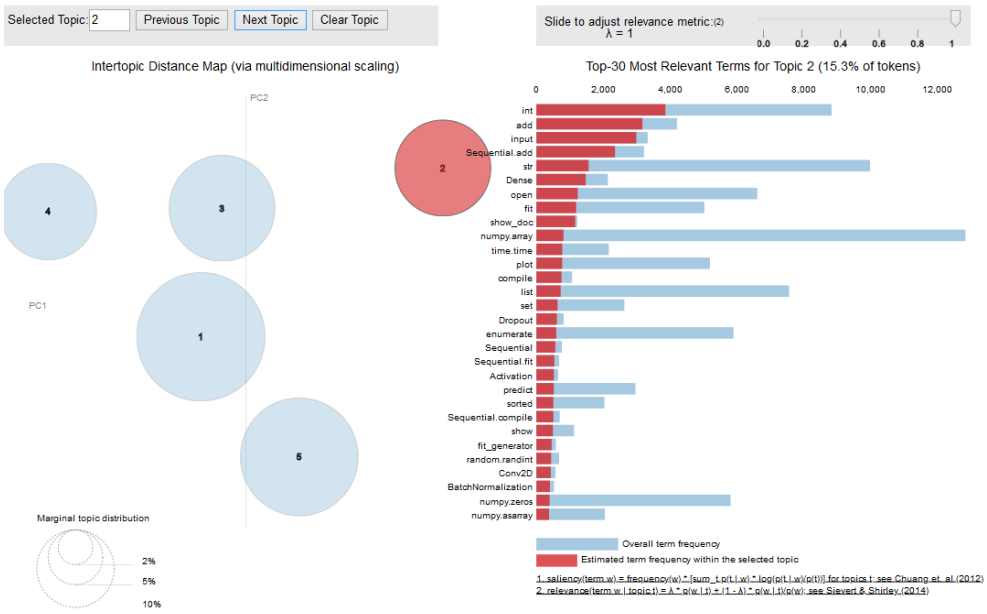


Abbildung 5.10: Wortverteilung für Thema 2 ohne Matplotlib

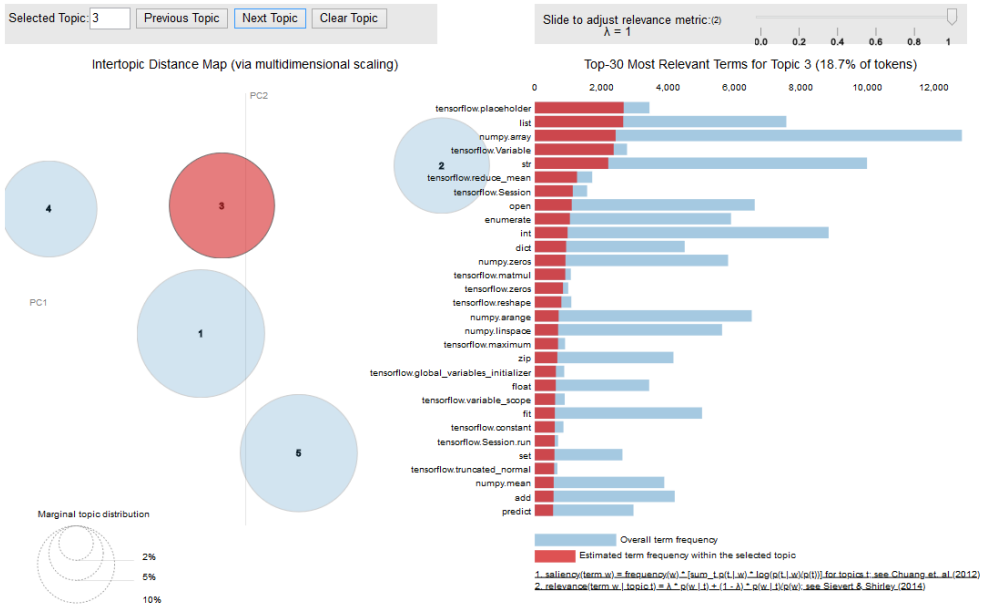


Abbildung 5.11: Wortverteilung für Thema 3 ohne Matplotlib

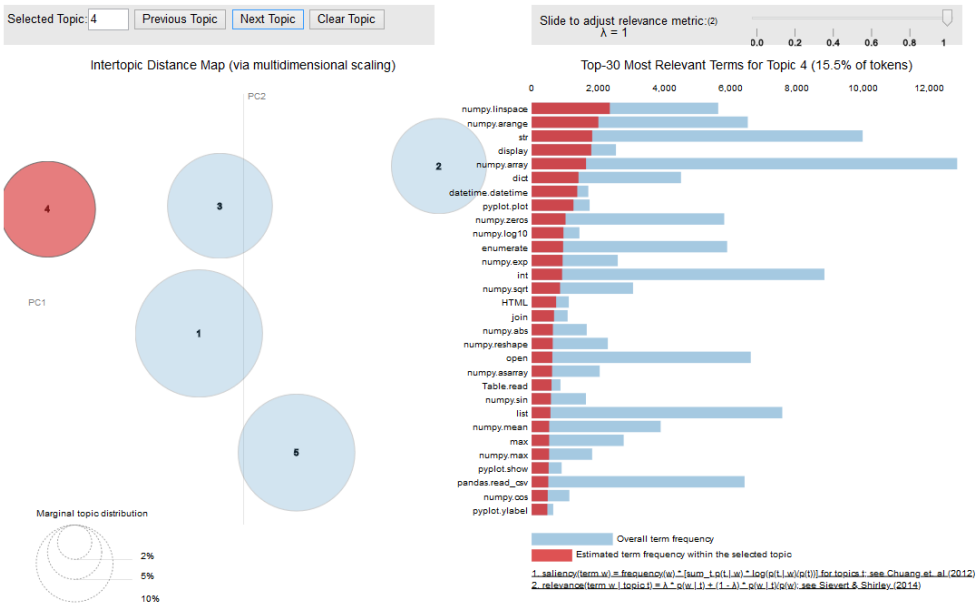


Abbildung 5.12: Wortverteilung für Thema 4 ohne Matplotlib

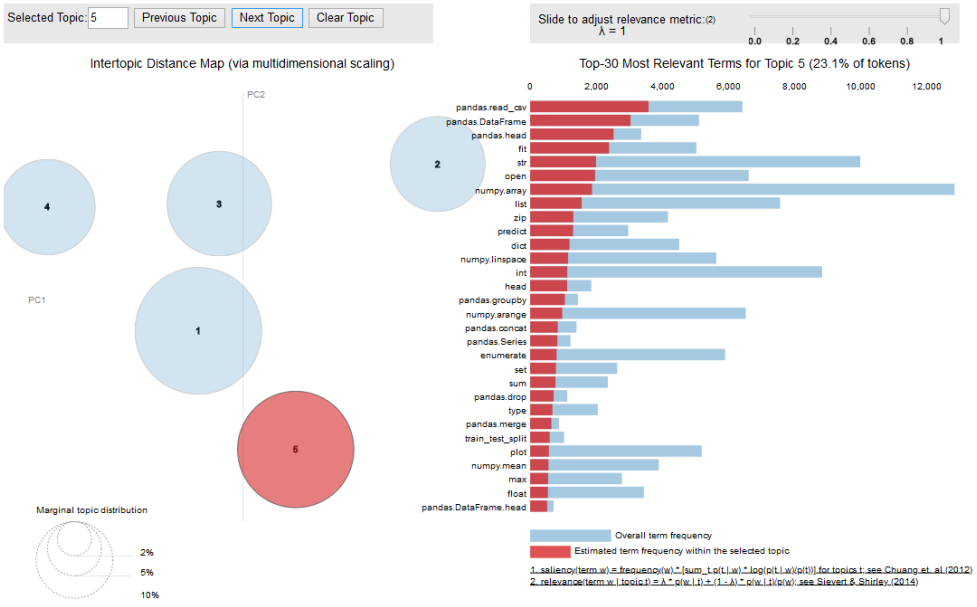


Abbildung 5.13: Wortverteilung für Thema 5 ohne Matplotlib

6 FAZIT UND AUSBLICK

In dieser Arbeit wurden Daten von GitHub gesammelt und bereinigt. Mit Hilfe von Syntaxbäumen konnten Funktionen und dazugehörige Module extrahiert und den Autoren zugeordnet werden. Das Matching von Funktionen zu Modulen erwies sich als eine Herausforderung. Da der Stil des Source Code von den Autoren und den verwendeten Funktionen abhängt, ist es schwer alle möglichen Fälle abzudecken. Durch Analyse von genügend Source Code ist es aber in Ausblick auf weitere Arbeiten durchaus möglich noch mehr Fälle abzudecken.
Durch die spezielle Struktur von Jupyter Notebooks ist es schwieriger diese maschinell zu verarbeiten, verglichen mit Dateitypen die nur reinen Source Code enthalten. Beim Umwandeln von Jupyter Notebooks in Python Dateien gingen einige verloren, da entweder die Struktur Fehler aufwies oder die Versionen nicht kompatibel waren.
Ziel der Arbeit war es eine Rollenverteilung nachzuweisen. Das Topic Modeling zeigt, dass es Differenzen von Funktionen zwischen Autoren gibt. Diese werden jedoch durch oft verwendete Module überlagert. In Ausblick auf weitere Forschung wäre es durchaus interessant zu beobachten, was für Ergebnisse durch andere Filtermethoden entstehen.

Teil II

APPENDIX

A

SQL QUERIES

```
#standardSQL
SELECT a.id id, size, content, binary, copies,
  sample_repo_name, sample_path
FROM (
  SELECT id, ANY_VALUE(repo_name) sample_repo_name, ANY_VALUE(path
      ) sample_path
  FROM `bigquery-public-data.github_repos.files`
  WHERE ENDS_WITH(path,'.ipynb')
  GROUP BY 1
) a
JOIN `bigquery-public-data.github_repos.contents` b
ON a.id=b.id
```

Listing A.1: Erhalte alle Jupyter Notebooks von GitHub

```
#standardSQL
SELECT *
FROM `bigquery-public-data.github_repos.commits`
WHERE commit IN
(SELECT distinct commit FROM `bigquery-public-data.github_repos.
    commits`
left join unnest(repo_name) as single_repo_name
where single_repo_name IN (SELECT DISTINCT(sample_repo_name) FROM
    `githubmining-237707.github_samples.content_ipynb`))
```

Listing A.2: Erhalte alle Commits von Repositories mit Jupyter Notebooks

```
#standardSQL
SELECT *
FROM `githubmining-237707.github_samples.commits_ipynb`
left join unnest(difference) as diff
WHERE ENDS_WITH(diff.new_path,'.ipynb')
```

Listing A.3: Erhalte alle Commits von Jupyter Notebooks

```
#standardSQL
SELECT * EXCEPT(new_path, repo)
FROM `githubmining-237707.github_samples.content_ipynb` a
LEFT JOIN
(SELECT repo, new_path, count(distinct author.email) contributors
```

```
FROM ‘githubmining-237707.github_samples.commits_ipynb_only‘
LEFT JOIN unnest(repo_name) repo
GROUP BY repo,new_path) b
ON a.sample_repo_name = repo
WHERE a.sample_path = b.new_path
```

Listing A.4: Erhalte alle Jupyter Notebooks mit Anzahl der Autoren

```
#standardSQL
SELECT *EXCEPT(repo_name, sample_repo_name,sample_path)
FROM ‘githubmining-237707.github_samples.commits_ipynb_only‘ a
LEFT JOIN unnest(repo_name) as repo
LEFT JOIN (SELECT sample_repo_name, sample_path, contributors FROM
     ‘githubmining-237707.github_samples.
    content_ipynb_contributors‘) b
ON repo = b.sample_repo_name
WHERE new_path = sample_path
AND contributors >1
```

Listing A.5: Erhalte alle Commit Daten von Notebooks mit mehr als 1 Autor

```
SELECT commit, author.email as author, new_path as path, repo,
    contributors
FROM ‘githubmining-237707.github_samples.commits_ipynb_teamwork‘
```

Listing A.6: Erhalte alle relevanten Commit-Daten

LITERATUR

[1] David M. Blei. "Probabilistic topic models". In: *Communications of the ACM* 55.4 (2012), S. 77. ISSN: 00010782. DOI: \url{10.1145/2133806.2133826}.

[2] David M. Blei, Andrew Y. Ng und Michael I. Jordan. "Latent dirichlet allocation". In: *Journal of machine Learning research* 3 (2003), S. 993–1022.

[3] Markus Döhring. "Text- und Webmining: Lecture Script Hochschule Darmstadt". Diss. Darmstadt: Hochschule Darmstadt, 2018.

[4] Bernhard Humm. *Applied Artificial Intelligence: An Engineering Approach*. http://leanpub.com: Leanpub, 2016.

[5] Mark Johnson. "Introduction to COmputational Linguistics and Natural Language Processing". Diss. Australia: Macquarie University, 2014.

[6] Shashank Kapadia. *Evaluate Topic Models: Latent Dirichlet Allocation (LDA)*. 2019. URL: \url{https://towardsdatascience.com/evaluate-topic-model-in-python-latent-dirichlet-allocation-lda-7d57484bb5d0}.

[7] Thomas Kluyver u. a., Hrsg. *Jupyter Notebooks – a publishing format for reproducible computational workflows*. 2016.

[8] Fernando Perez und Brian E. Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3 (2007), S. 21–29. ISSN: 1521-9615. DOI: \url{10.1109/MCSE.2007.53}.

[9] Davide Spadini, Maurício Aniche und Alberto Bacchelli. "PyDriller: Python framework for mining software repositories". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, S. 908–911. ISBN: 9781450355735. DOI: \url{10.1145/3236024.3264598}. URL: \url{http://dl.acm.org/citation.cfm?doid=3236024.3264598}.