Universität Leipzig
Fakultät für Mathematik und Informatik
Wirtschaftswissenschaftliche Fakultät

Ermittlung von Konfigurationsoptionen im Source Code mit Fokus auf Machine Learning Bibliotheken in Python

Bachelorarbeit

Marco Jaeger-Kufel
geb. am: 10.05.1995 in Hannover

Matrikelnummer 3731679

1. Gutachter: Prof. Dr. Norbert Siegmund
2. Gutachter: Prof. Dr. Unknown Yet

Datum der Abgabe: 14. Juni 2022

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Leipzig, 14. Juni 2022

.....
Marco Jaeger-Kufel

Zusammenfassung

A short summary.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Moderne Softwaresysteme ermöglichen den Nutzenden ein breites Spektrum unterschiedlicher Konfigurationsoptionen. Anhand dieser Konfigurationsoptionen sind die Nutzenden in der Lage, viele Aspekte der Ausgestaltung einer Software zu steuern. Konfigurationsoptionen können dabei ganz unterschiedliche Funktionen besitzen, die von den Nutzenden nach den eigenen Bedürfnissen angepasst werden können.

Konfigurationsoptionen können in verschiedenen Teilen eines Softwareprojekts verarbeitet, definiert und beschrieben werden, wie beispielsweise in der Konfigurationsdatei, im Source Code und in der Dokumentation [Dong et al., 2016]. Sie werden meist als Key-Value Pair entworfen und gesammelt in einer Konfigurationsdatei gespeichert. Dem Namen der Konfigurationsoption (Key) werden dabei Einstellungsmöglichkeiten beliebigen Typs zugeordnet (Value). Zur Speicherung von Konfigurationsoptionen verwenden einige Systeme, wie das Big Data-Framework Hadoop, strukturierte XML-Formate oder auch JSON-Dateien. Ein einheitliches Schema zur Speicherung als Konfigurationsdatei gibt es jedoch nicht, weshalb sich diese von der Struktur und Syntax unterscheiden [Rabkin and Katz, 2011]. Im Source Code können die Konfigurationsoptionen eingelesen und bearbeitet werden [Dong et al., 2016].

Während die Vielfältigkeit und Individualisierbarkeit der Software größer wird, erhöht sich auch die Komplexität der Software für Entwickler und Entwicklerinnen. Vor allem das Warten und Testen der Konfigurationsmöglichkeiten gestaltet sich nun umfangreicher. Besonders problematisch wird es, wenn Konfigurationsoptionen auf unvorhergesehene Weise interagieren. Solche Abhängigkeiten sind in einem wachsenden Spielraum möglicher Kombinationen

schwer zu entdecken und zu verstehen. Entwickler und Entwicklerinnen müssen die Konfigurationsoptionen innerhalb der Software verfolgen, um festzustellen, welche Codefragmente von einer Option betroffen sind und wo und wie sie mit ihr interagieren. Des Weiteren kann es vorkommen, dass Entwickler und Entwicklerinnen Werte von Konfigurationsoptionen nicht aktualisieren, was dazu führen kann, dass über mehrere Module hinweg, unbemerkt mit falschen Werten gearbeitet wird [Dong et al., 2016]. Die verschiedenen möglichen Ausprägungen einer Konfigurationsoption erschweren hierbei die Nachvollziehbarkeit des Kontrollflusses der Software.

In einer empirischen Studie über Konfigurationsfehler stellen Yin et al. fest, dass in kommerziellen Softwareunternehmen für Speicherlösungen, knapp ein Drittel aller Ursachen von Kundenproblemen auf Konfigurationsfehler zurückzuführen sind (siehe Tabelle 3). Bei Konfigurationsfehler sind Source Code und die Eingabe zwar korrekt, es wird jedoch ein falscher Wert für eine Konfigurationsoption verwendet, sodass sich die Software nicht wie gewünscht verhält. Solche Fehler können dazu führen, dass die Software abstürzt, eine fehlerhafte Ausgabe erzeugt oder nur unzureichend funktioniert [Zhang and Ernst, 2014].

Konfigurationsfehler entstehen zum Beispiel durch die Verwendung unterschiedlicher Versionen einer Software. Im folgendem Codeausschnitt wird die Klasse *LogisticRegression* der Machine Learning-Bibliothek *scikit-learn* initialisiert und der Variable *clf* zugewiesen:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
```

Da keine Parameter angegeben werden, wird die Klasse mit den Default-Werten initialisiert, zum Beispiel *max_iter = 100, verbose = 0, n_jobs = 1.* In der Version *scikit-learn 0.21.3* wird für den Parameter *multi_class* als Default noch der Wert *ovr* zugewiesen (2022). Für alle nachfolgenden Versionen ist der Default-Wert für diesen Parameter jedoch *auto* (2022). Folglich führt die Verwendung dieser Klasse ohne explizite Parameterangabe, nach Verwendung unterschiedlicher Versionen, zu schwer nachvollziehbaren Programmverhalten.

Bei den Parametern eines solchen Machine Learning Algorithmus, die vor Beginn des Lernprozesses vom Nutzenden festgelegt werden können, spricht man dabei von sogenannten *Hyperparametern.* Sie werden als Parameter an die jeweilige Klasse übergeben und bieten den Nutzenden Konfigurationsmöglichkeiten für das Training des Modells. In dem obigen Fall können so beispiels-

weise über den Parameter *max_iter* die Anzahl der Trainingsiterationen der Logistischen Regression festgelegt werden und über *n_jobs* die maximale Anzahl der parallel zu verwendenden CPU-Kernen. Durch das Extrahieren dieser Konfigurationswerte, können die Parameter statisch geprüft werden, was für den Nutzenden eine Zeitersparnis bedeutet, wenn sonst ein falscher Konfigurationswert einen langen Batch-Job zum Scheitern bringt oder das Ergebnis nicht den Erwartungen entspricht.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, Konfigurationsoptionen im Source Code zu erkennen und zu extrahieren.

Es existieren bereits einige Forschungsansätze zur Ermittlung und Verarbeitung von Konfigurationsoptionen im Source Code. Viel zitiert wird dabei der Ansatz von Rabkin and Katz, den sie 2011 publizierten. Wie auch Dong et al. oder Lillack et al. entwickelten sie einen Ansatz, mittels statischer Code-Analyse Konfigurationsoptionen im Source Code zu tracken. Dabei fokussierten sie sich auf die Programmiersprache Java, die nach dem TIOBE-Index jahrelang als beliebteste Programmiersprache galt. Der TIOBE-Index misst die Popularität von Programmiersprachen auf der Grundlage von Suchanfragen auf beliebten Websites und in Suchmaschinen. Im Februar 2022 ist in diesem Ranking Python erstmals zur beliebtesten Programmiersprache aufgestiegen [BV, 2022]. Für Python ist diese Thematik jedoch bislang allerdings noch wenig beleuchtet.

Die Programmiersprache Python ist für ihre Benutzendenfreundlichkeit bekannt und obwohl Python eine interpretierte High-Level-Programmiersprache ist, ist sie in der Lage, bei Bedarf die Leistung von Programmiersprachen auf Systemebene zu nutzen. Insbesondere im Bereich wissenschaftliches Rechnen (Scientific Computing) gewann Python in den letzten Jahren enorm an Popularität, weshalb viele Bibliotheken für maschinelles Lernen auf Python basieren [Raschka et al., 2020].

Gegenstand dieser Arbeit werden daher Konfigurationsoptionen von ML Algorithmen in Python Source Code, die mittels statischer Code-Analyse erkannt werden. Der Ansatz identifiziert automatisch die Stellen im Source Code, an denen die Optionen der zu untersuchenden Bibliotheken gelesen werden und ermittelt für jede dieser Stellen den Namen der Option. Darauf aufbauend er-

folgt eine Datenflussanalyse, um auch für Optionen, die in Form von Variablen als Parameter übergeben werden, die möglichen Werte zu ermitteln. Der Fokus liegt auf drei der populärsten Machine Learning Bibliotheken in Python (siehe Abbildung 4.1):

- TensorFlow
- PyTorch
- scikit-learn

Zudem wird die Verwendung von Konfigurationsoptionen der Machine Learning Lifecylce Plattform *MLflow* untersucht.

1.3 Aufbau dieser Arbeit und methodisches Vorgehen

Kapitel 2

Hintergrund

2.1 Statische Code-Analyse

Das wichtigste Werkzeug dieser Arbeit ist die statische Code-Analyse, mit der Software-Projekte unabhängig von der Ausführungsumgebung untersucht werden können. Die statische Code-Analyse ist ein Werkzeug, um Fehler in einer Softwareanwendung zu reduzieren. So ermöglichen sie den Anwendenden, Fehler in einem Programm zu finden, die für den Compiler nicht sichtbar sind [Bardas et al., 2010].

Im Gegensatz zur dynamischen Analyse wird die statische Analyse zur Übersetzungszeit durchgeführt und setzt damit bereits vor der tatsächlichen Ausführung des Source Codes an. Die erzeugten Ergebnisse der statischen Analyse lassen sich besser Verallgemeinern, da sie nicht abhängig von den Eingaben sind, mit denen das Programm während der dynamischen Analyse ausgeführt wurde [Gomes et al., 2009].

Für das Aufspüren von Konfigurationsoptionen bietet sich eine statische Code-Analyse daher aus mehreren Gründen an. So kann es viele Optionen geben, die nur in bestimmten Modulen oder als Folge bestimmter Eingaben verwendet werden. Mit der statischen Code-Analyse kann eine hohe Abdeckung hingegen deutlich leichter erreicht werden. Gleichzeitig verbergen sich hinter den Methoden und Klassen, der zu untersuchenden Machine Learning Bibliotheken, teils sehr komplexe und rechenintensive Berechnungen, die bis zu mehreren Tagen laufen könnten. Aus Kosten-Nutzen-Gründen ist hier eine dynamische Analyse nur bedingt sinnvoll.

2.2 Datenflussanalyse

Die Datenflussanalyse ist ein Werkzeug, um Informationen über die möglichen Werte, die an verschiedenen Stellen in einem Codesegment berechnet werden, zu erfassen [Pollock and Soffa, 1989]. Es handelt sich um eine statische Analysetechnik mit dem Ziel, das Programmverhalten schon zur Übersetzungszeit, also bevor es ausgeführt wurde, zu bestimmen. Der Datenfluss kann mittels eines Kontrollflussgraphens dargestellt werden und dem Betrachtenden Rückschlüsse über das Verhalten des Programms geben. Der Graph zeigt an, an welchen Stellen eine Variable verwendet wird und welche Werte sie annehmen kann.

Es gibt verschiedene Techniken, die innerhalb der Datenflussanalyse eingesetzt werden, um den Wert von Variablen zu bestimmen. Eine von ihnen ist *Constant Propagation*. Das Ziel von Constant Propagation ist zu bestimmen, an welchen Stellen im Programm, eine Variable einen konstanten Wert besitzt. So kann zum Beispiel toter Code, also redundanter Code, der im weiteren Programmverlauf nicht weiterverarbeitet wird, gefunden werden.

Eine weitere Technik ist *Static Single Assignment*. Bei dieser Methodik werden die Variablen im Verlauf des Übersetzungsprozesses in eine Zwischenform überführt, in der jede Variable genau einmal zugewiesen wird. Die Variablen werden in Versionen aufgeteilt und in der Regel mit einem aufsteigendem Index versehen, sodass jede Definition ihre eigene Version erhält.

Die beide Techniken Static Single Assignment und Constant Propagation werden in dem statischen Analyse-Framework *Scalpel* kombiniert. Aus dem folgenden Codebeispiel geht hervor, dass die Variable a zwei unterschiedliche Werte annehmen kann.

```
c = 10
a = -1
if c > 0:
    a = a + 1
else:
    a = 0
total = c + a
```

Der daraus resultierende Kontrollflussgraph besteht charakteristisch aus Knoten, die die jeweilige Code-Objekte beinhalten und gerichtete Kanten als Übergang zwischen den Knoten, die den Programmablauf darstellen. Mittels

Constant Propagation werden die tatsächlichen Werte der Variablen an den jeweiligen Verwendungszeitpunkten erkannt und über die Φ-Funktion kann durch Static Single Assignment abgeleitet werden, dass es zwei mögliche Rückgabewerte gibt.

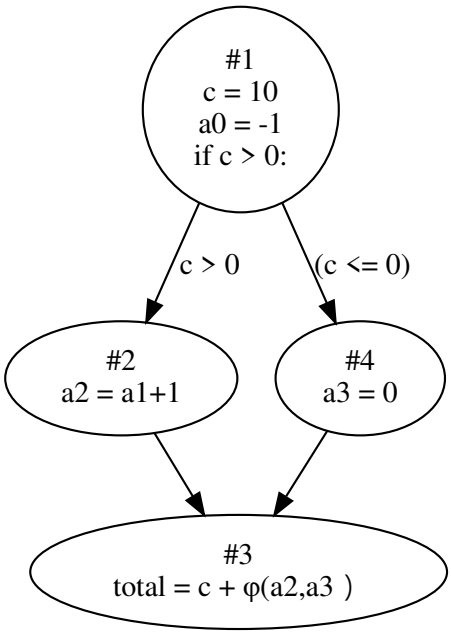

Abbildung 2.1: Kontrollflussgraph [Li et al., 2022]

2.3 Maschinelles Lernen

Die künstliche Intelligenz (KI) ist ein Teilgebiet der Informatik und befasst sich mit der Entwicklung von Computerprogrammen und Maschinen, die in der Lage sind, Aufgaben auszuführen, die Menschen von Natur aus gut beherrschen. Dazu gehören zum Beispiel die Verarbeitung von natürlicher Sprache (Natural Language Processing) oder Bilderkennung (Computer Vision). In der Mitte des 20. Jahrhunderts entstand das maschinelle Lernen (ML) als Teilbereich der KI und schlug eine neue Richtung für die Entwicklung von künstlicher Intelligenz ein, inspiriert vom konzeptionellen Verständnis der Funktionsweise des menschlichen Gehirns [Raschka et al., 2020].

Pionier Arthur Samuel definiert maschinelles Lernen als ein Fachgebiet, das Computern die Fähigkeit verleiht, zu lernen, ohne ausdrücklich programmiert zu werden. Es befasst sich mit der wissenschaftlichen Untersuchung von Algorithmen und statistischen Modellen, die Computersysteme verwenden, um

eine Aufgabe zu lösen [Mahesh, 2020]. Dabei werden statistische Methoden verwendet, um aus Daten zu lernen und Muster zu erkennen.

2.3.1 Python-Bibliotheken

Fürs maschinelle Lernen gilt Python schon seit langem als erste Wahl für Entwickler und Entwicklerinnen. Eine im Mai 2019 veröffentlichte Umfrage vom Portal KDnuggets ergab, dass Python in der Kategorie „Top Analytics, Data Science, Machine Learning Tools“ von rund 66% der Teilnehmenden verwendet wird und damit die populärste Programmiersprache in diesem Bereich ist [KDnuggets, 2019].

Python ist eine interpretierte Programmiersprache. Demnach wird während der Ausführung der Python-Code zur Laufzeit interpretiert, wodurch sie im Vergleich mit kompilierten Programmiersprachen wie C / C++ hinsichtlich Leistung und Geschwindigkeit schlechter abschneidet. Ein wichtiger Vorteil von Python ist jedoch die Möglichkeit, Code aus anderen Programmiersprachen relativ einfach einzubinden. Viele Lösungen im maschinellen Lernen basieren auf numerischen und vektorisierten Berechnungen mit Bibliotheken wie *NumPy* oder *SciPy*. Um diese Berechnungen schnell und effizient auszuführen, werden sogenannte *Wrapper* verwendet, die Algorithmen von kompilierten Programmiersprachen implementieren. Die wahrscheinlich am häufigsten verwendete Bibliothek für diesen Zweck ist Cython, die zwar auf Python basiert, aber auch den Aufruf von Funktionen, sowie die Verwendung von Variablen und Klassen aus der Programmiersprache C unterstützt [Stančin and Jović, 2019]. Dadurch können kritische Teile des Codes um ein Vielfaches beschleunigt werden.

Einer der Hauptgründe für die Popularität von Python ist das riesige Ökosystem, das aus einer Vielzahl von umfangreichen und leistungsfähigen Bibliotheken besteht, über die ML-Algorithmen aufgerufen werden können, wodurch die Benutzendenfreundlichkeit bei gleichzeitiger Effizienz in der Performanz gewahrt bleiben kann [Raschka et al., 2020]. So sind nach einer jährlich von der Data Science-Plattform Kaggle durchgeführten Umfrage unter rund 25.000 ML-Engineers und Data Scientists, die Python Bibliotheken mit großem Abstand die am meisten genutzten Frameworks für maschinelles Lernen [Kaggle, 2021].

Eine Python-Bibliothek entspricht einer Sammlung zusammenhöriger Module, die aus vorkompiliertem Code bestehen. Nach erfolgreicher Installati-

on werden die Funktionalitäten der jeweiligen Bibliothek über die *import*-Anweisung für ein Programm zugänglich gemacht. Entwickler und Entwicklerinnen können nun die vorprogrammierten Klassen und Methoden aufrufen und beliebig verwenden, ohne die hinterlegten Algorithmen kennen zu müssen.

2.3.2 ML Konfigurationsoptionen

In objektorientierten Programmiersprachen wie Python sind Klassen einer der grundlegenden Bausteine, die bei der Entwicklung und Anwendung von maschinellem Lernen eingesetzt werden. Sie bieten die Möglichkeit, Daten und Funktionen zu kombinieren und dem Nutzenden von Machine Learning Bibliotheken, bereits vorformulierte Algorithmen aufzurufen. Dabei werden die Konfigurationsoptionen beim Aufruf der jeweiligen Klasse als Parameter übergeben, sodass die Algorithmen an die Bedürfnisse des Anwendenden angepasst werden können. So kann die zielgerichtete Nutzung effizienter und komplexer Algorithmen bei gleichzeitiger Konfigurierbarkeit gewährleistet werden.

Bei den Konfigurationsparametern von Lernalgorithmen handelt es sich um sogenannte *Hyperparameter*. Lernalgorithmen ermöglichen einem Computerprogramm, den menschlichen Lernprozess mittels statistischer Methoden zu imitieren. Sie werden zur Mustererkennung, Klassifizierung und Vorhersage verwendet, indem sie aus einem vorhandenen Trainingsdatensatz lernen. Hyperparameter werden festgelegt bevor der Lernprozess beginnt und um den Lernprozess zu steuern [Andonie, 2019]. Da sich die Algorithmen oft sehr unterschiedlich verhalten, wenn sie mit verschiedenen Hyperparameter instanziiert werden, wurden in den letzten Jahren eine Vielzahl an Hyperparameter-Optimierungsverfahren entwickelt [Hutter et al., 2014]. Optimierte Hyperparameter können die Leistungsfähigkeit eines Modells oder die Geschwindigkeit und Qualität Lernprozesses stark beeinflussen [Andonie, 2019].

Im folgenden Codebeispiel wird der *Gradient Boosting Classifier* aus der scikit-learn Bibliothek betrachtet.

```
from sklearn.ensemble import GradientBoostingClassifier

gbc = GradientBoostingClassifier(n_estimators=20,
        learning_rate=0.05, max_features=2, max_depth=2,
        random_state=0)
```

Es wird eine Instanz dieser Klasse erzeugt und der Variable *gbc* zugewie-

sen. Die Klasse verfügt über 20 Parameter (Version 1.0.2), die, sofern bei der Instanziierung für den jeweiligen Parameter kein neuer Wert zugewiesen wird, mit einem eigenen Default-Wert initialisiert werden. So werden im Beispiel, den Hyperparametern wie *n_ estimators*, *learning_ rate* oder *max_ depth* Zahlenwerte übergeben, die von den Default-Werten abweichen.

2.4 Web Scraping

Web Scraping ist eine Technik, um Daten aus dem World Wide Web zu extrahieren, um sie später abrufen oder analysieren zu können. Dafür werden die Webdaten über das Hypertext Transfer Protocol (HTTP) oder über einen Webbrowser ausgelesen. Dies kann entweder manuell durch den jeweiligen Benutzenden oder automatisch durch einen Bot oder Webcrawler erfolgen [Zhao, 2017]. Ein Web Scraper simuliert das menschliche Browsing-Verhalten im Web, um aus verschiedenen Websites detaillierte Informationen in einer vorgegebenen Struktur zu sammeln. Durch die Möglichkeit, für eine bestimmte Website-Struktur systematisch ausgerichtet und programmiert zu werden, liegt der Vorteil eines Web Scrapers in seiner Automatisierungsfähigkeit und Geschwindigkeit [Diouf et al., 2019]. Mögliche Anwendungsfälle sind zum Beispiel das Überwachen von Preis-Änderungen in Online-Shops oder das Auslesen und Kopieren von Kontaktinformationen.

Ein Web Scraping-Prozess gliedert sich üblicherweise in zwei Schritten [Zhao, 2017]:

1. Erfassen der Webressourcen
2. Extrahieren der gewünschten Informationen aus den erfassten Daten

Zunächst wird die Kommunikation zur Ziel-Website über das HTTP-Protokoll hergestellt [Glez-Peña et al., 2013]. Über die HTTP-Anfrage ist der Scraper in der Lage, die Ressourcen der jeweiligen Website zu erfassen [Zhao, 2017]. Dies erfolgt entweder als URL mit einer GET-Abfrage für Ressourcenanfragen oder als HTTP-Nachricht mit einer POST-Abfrage für die Übermittlung von Formularen [Glez-Peña et al., 2013]. Nachdem die Anfrage erfolgreich empfangen und von der Ziel-Website verarbeitet wurde, wird die angeforderte Ressource von der Website aufgerufen und an das Web Scraping-Programm zurückgesendet. Die Ressource kann in verschiedenen Formaten vorliegen: In Auszeichnungssprachen wie HTML (Hypertext Markup Language) oder XML (Extensible Markup Language), JSON-Format (JavaScript Object Notation) oder in Form

von Multimedia-Daten wie Bilder-, Audio oder Videodateien [Zhao, 2017].

Im zweiten Schritt folgt der Extraktionsprozess. Die heruntergeladenen Daten können nun geparst werden, um die benötigten Informationen zu filtern und in ein geeignetes Format umzuwandeln [Glez-Peña et al., 2013]. Die Daten können nun weiterverarbeitet werden, indem sie beispielsweise analysiert oder in eine gewünschte Struktur organisiert werden.

2.5 Abstract Syntax Trees

Um die Bedeutung von Abstract Syntax Trees (AST) zu verstehen, ist es hilfreich sich zu vergegenwärtigen, welche Prozesse bei der Ausführung eines Python-Skripts im Hintergrund ablaufen. Als eine interpretierte High-Level-Programmiersprache wird der Source Code in Python nicht kompiliert, sondern vom Python Interpreter nach einer bestimmten Abfolge von Schritten in Anweisungen übersetzt. Dabei wird der Python Code in Bytecode umgewandelt, sodass dieser von der Python Virtual Machine übersetzt und ausgeführt werden kann.

Zunächst wird der Code geparst und in sogenannte Tokens unterteilt. Diese Tokens unterliegen einer Reihe von Regeln, damit die verschiedenen Programmierkonstrukte unterschiedlich erkannt und behandelt werden können. Die Liste an Tokens werden dann in eine Baumstruktur, den sogenannten abstrakten Syntaxbaum (Abstract Syntax Tree), umgewandelt. Ein AST ist eine baumartige Darstellung des Codes, bestehend aus einer Sammlung von Knoten, die, basierend auf der Grammatik in Python, miteinander verbunden sind. Der Baum wird in maschinenlesbaren Binärcode umgewandelt und Anweisungen in Bytecode an den Python Interpreter übermittelt. Der Python Interpreter kann den Code nun ausführen und Systemaufrufe an den Kernel starten, um das Programm zu starten.

Während Bytecode eher für Maschinen gemacht ist, sind Abstract Syntax Trees strukturiert aufgebaut und auch für den Menschen lesbar. Im Anhang befindet sich das Codebeispiel aus **??**, nachdem es als AST geparst wurde. Aus der AST-Syntax lassen sich Typ und Struktur einzelner Python Komponenten direkt ablesen. So besteht das Modul aus zwei Codeobjekten vom Typ *ImportFrom* und *Assign*, die sich jeweils in kleinere Objekte verschiedenen Typs unterteilen lassen. Diese Datenstruktur stellt alle relevanten Informationen für die Ausführung des Codes bereit. Jeder Knoten des Baumes kann nun besucht

werden, um seine Daten zu verarbeiten und entsprechende Aktionen einzuleiten. Über das *ast*-Modul in Python kann der Code als AST verarbeitet und visualisiert werden, sodass er von Entwicklern und Entwicklerinnen entsprechend seiner AST-Syntax analysiert und manipuliert werden kann.

Kapitel 3

Verwandte Arbeiten

Zwar wurden bereits einige Ansätze zum Auffinden von Konfigurationsoptionen publiziert, jedoch ist diese Thematik im Rahmen von maschinellem Lernen kaum beleuchtet. Die meisten Ansätze, wie der von Rabkin and Katz aus dem Jahre 2011, behandeln Anwendungen in der Programmiersprache Java und ermitteln Konfigurationsoptionen mittels statischer Code-Analyse. Der von Rabkin and Katz entwickelte *Confalyzer* ist vermutlich einer der ersten bekannteren Tools, die sich mit der Thematik befassen. Unter der Annahme, das Konfigurationsoptionen als Key-Value Pair vorliegen, betrachtet der Confalyzer Methoden, die mit dem für Java typischen Schlüsselwort *get* in der Konfigurationsklasse beginnen. Mittels eines Aufrufgraphens (Call Graph) identifiziert der Confalyzer, wo die Methoden im Source Code aufgerufen werden. An den jeweiligen Aufrufstellen kann er dann die Konfigurationsoptionen aus den String-Parametern ablesen. Der Ansatz beruht auf der Annahme, dass viele Konfigurationsoptionen auf ähnliche Weise verwendet werden und bestimmte Muster ableitbar sind [Rabkin and Katz, 2011].

Der von Dong et al. entwickelte *ORPLocator* orientiert sich an dem Confalyzer und vergleicht sich mit diesem. Der ORPLocator untersuchte dieselben Java-Frameworks wie beispielsweise Hadoop und konnte mehr dokumentierte Optionen und dementsprechend auch mehr Verwendungen im Source Code finden. Dies liegt unter anderem daran, dass der gesamte Source Code nach Aufrufstellen von Konfigurationsschnittstellen durchsucht wird, während der Confalyzer einen Aufrufgraphen erstellt, der manche Optionen nicht erfasst [Dong et al., 2016].

Einen anderen Ansatz verfolgten Lillack et al. bei der Entwicklung von *Lotrack*. Lotrack ist ein Tool, das mittels einer erweiterten Taint-Analyse, einer Datenflussanalyse, die externe Daten (Eingabedaten) über den gesamten

Kontrollfluss verfolgt, eine Konfigurations-Map erstellt. Die Konfigurations-Map beschreibt, welche Codefragmente von Konfigurationsoptionen abhängen und hilft dabei die Beziehungen zwischen dem konkreten Programmverhalten und der Konfiguration zu finden. Der Fokus liegt auf Anwendungen, die auf Android-Systemen laufen [Lillack et al., 2018].

Weitere Ansätze wie der *PrefFinder* von Jin et al. verwenden zusätzlich auch dynamische Analysetechniken, um Konfigurationsoptionen nicht nur zu extrahieren, sondern auch in einer Datenbank zur Abfrage und Verwendung zu speichern. Der *Software Configuration Inconsistency Checker* (SCIC) hingegen von Behrang et al. erweitert die Extraktion von Konfigurationsoptionen im Key-Value-Modell des Confalyzer um ein baumstrukturiertes Modell. Im Gegensatz zu anderen Tools, ist dieses auch in der Lage, mehrere Programmiersprachen zu verarbeiten. Ziel dieses Tools ist die Identifikation von Konfigurationsfehlern [Behrang et al., 2015].

Kapitel 4

Methodik

4.1 Auswahl der Python-Bibliotheken

Bevor auf das Vorgehen zur Extraktion von Konfigurationsoptionen genauer eingegangen wird, werden an dieser Stelle die zu untersuchenden Python-Bibliotheken beschrieben. Im folgenden Abschnitt werden Repositorys beleuchtet, die diese Bibliotheken importieren und verwenden.

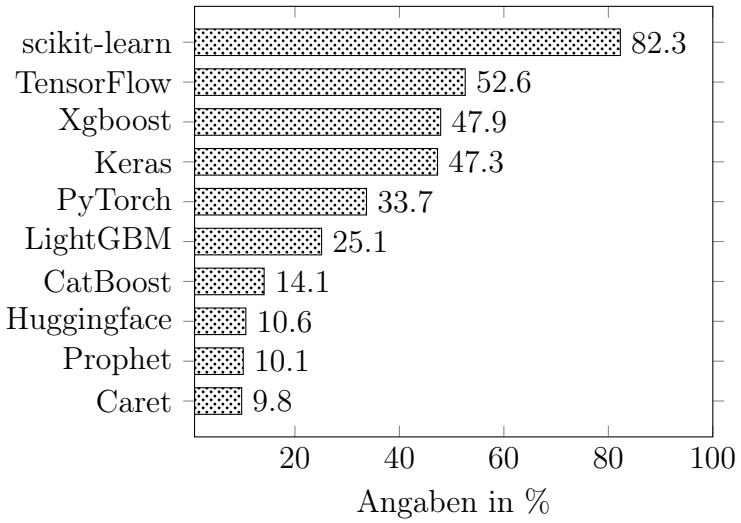

Abbildung 4.1: Nutzung von ML-Frameworks [Kaggle, 2021]

Aus der Grafik lassen sich drei der zu untersuchenden Python Bibliotheken entnehmen. Zum einen *scikit-learn*, das aufgrund der Vielzahl an implementierten Algorithmen wie ein „Schweizer Taschenmesser“ für die meisten Projekte

eingesetzt werden kann und von über 80% der Befragten verwendet wird [Kaggle, 2021]. Die Vorteile von scikit-learn ist die benutzerfreundliche Struktur und Dokumentation mit der maschinelles Lernen auch für unerfahrene oder fachfremde Menschen zugänglich gemacht wird [Varoquaux et al., 2015].

Bei der zweiten zu untersuchenden Bibliothek handelt es sich um *TensorFlow*. Nachdem es 2015 von Forschern bei Google ursprünglich für interne Zwecke entwickelt wurde, hat sich TensorFlow vor allem im Bereich Deep Learning als populäres Werkzeug bewährt. So bietet TensorFlow Projekte, die viel Customizing erfordern, eine leistungsfähige und flexible Umgebung, die das Trainieren künstlicher neuronaler Netze vereinfacht und beschleunigt [Pang et al., 2020].

Die dritte zu untersuchende Bibliothek *PyTorch* wird zwar insgesamt weniger häufig verwendet, verzeichnet jedoch Jahr für Jahr ein starkes Wachstum in der Nutzung [Kaggle, 2021]. PyTorch ist vor allem nützlich beim Umgang mit künstlichen neuronalen Netzen, weshalb es 2019 auch die meistgenutzte Deep Learning-Bibliothek auf allen großen Deep Learning-Konferenzen war [Raschka et al., 2020].

MLflow ist wie die eben aufgeführten Bibliotheken Open Source und wurde vom Silicon Valley Startup databricks 2018 ins Leben gerufen. Ziel der Plattform ist dabei, den Lebenszyklus von ML-Projekten ganzheitlich zu strukturieren. Dabei werden über die ML-Algorithmen hinaus, den Anwendenden Werkzeuge mit in die Hand zu geben, um den Herausforderungen von ML-Projekten gegenüber herkömmlicher Softwareentwicklungsprozesse zu begegnen [Zaharia et al., 2018].

4.2 Web Scraping der ML-Klassen

Bevor im Source Code nach den Konfigurationsoptionen gesucht werden kann, ist es erforderlich, sich einen Überblick über die zu untersuchenden Bibliotheken zu verschaffen. Über die Dokumentation der Websites der Bibliotheken erhält man einen Einblick in die jeweilige Modul- und Klassenstruktur. Da die gesuchten Optionen als Parameter von ML-Klassen übergeben werden, zielt dieser Ansatz darauf ab, alle Klassen mitsamt der möglichen Parameter mithilfe von Webscraping zu extrahieren. Aufgrund des systematischen Aufbaus der verschiedenen Websites, kann man an dieser Stelle mit technischer Unterstützung die gesuchten Daten automatisiert erfassen. Die Python-Bibliothek

Beautiful Soup stellt für diesen Zweck, das notwendige Werkzeug bereit.

Der Scraping-Prozess lässt sich für die vier verschiedenen ML-Bibliotheken in folgende Schritte einteilen, die über die Klasse *ClassScraper* aufgerufen werden können.

Je nach Komplexität der Dokumentation, werden diese Schritte teilweise in weitere kleinere Schritte unterteilt. Zudem werden die Informationen in einem jeweils unterschiedlichen HTML-Format abgebildet, sodass kein generischer Algorithmus über alle vier Dokumentationen laufen kann. Deshalb wird für jede Bibliothek eine eigene Klasse implementiert, die vom ClassScraper zielführende Methoden erbt und bei einer abweichenden Dokumentationsstruktur Methoden überschreibt.

Im folgendem, exemplarischem Codesnippet werden die Modul-URLs aus der mlflow-Dokumentation *gescraped*. Um mit dem Scrapen der Daten zu beginnen, wird über über das *request*-Modul der Python-Bibliothek *urllib* die URL geöffnet. Die HTML-Datei der URL wird in Beautiful Soup geparst, um auf die Daten der Seite zugreifen zu können. In einem HTML-Dokument werden die Inhalte baumartig gespeichert. Die Knoten dieses Baums werden mit sogenannten *Tags* versehen, die dem Inhalt Form und Struktur verleihen. In einem sogenannten *div*-Container befindet sich eine Liste mit den einzelnen Modul-Links, dessen Elemente mit einem *li*-Tag (list item) versehen sind. Um auf den Bereich der Seite zuzugreifen, in dem die mlflow-Module verlinkt sind, wird der Bereich über die Beautiful Soup-Methoden *find* und *findAll* weiter eingegrenzt. Diese filtern die HTML-Datei nach den jeweiligen Tags, sodass schließlich der Variable *li* ein Set an HTML-Elementen zugewiesen wird. Durch dieses Set wird iteriert, um für jedes Modul die Hyperlink-Referenz zu speichern.

```
def scrape_module_urls(self):
    link = "mlflow.org/docs/latest/python_api/index.html"
    html = urllib.urlopen(link)
    soup = bs4.BeautifulSoup(html, "html.parser")
```

```
    div = soup.find("div", {"class": "section"})
    li = div.findAll("li", {"class": "toctree-l1"})

    for element in li:
        url = element.find("a").attrs["href"]
        self.module_urls.append(url)
```

Im Anschluss kann nun nach einem ähnlichen Prinzip durch die Liste der Modul-URLs iteriert werden. Die URL der Module werden ebenfalls mit urllib geöffnet und die HTML-Datei in Beautiful Soup geparst, sodass schlussendlich alle Klassen inklusive der Parameter gefunden und in einer JSON-Datei gespeichert werden. In der JSON-Datei wird für jede Klasse der gesamte Pfad als eindeutiger Schlüssel angegeben. Zusätzlich zum Klassen- und zu den Parameternamen, werden die Default-Werte der einzelnen Parameter aufgeführt. Die scikit-learn-Klasse *StratifiedGroupKFold* wird demnach wie folgt gespeichert:

```
'sklearn.model_selection.StratifiedGroupKFold': {
    'short name': 'StratifiedGroupKFold',
    'parameters': {
        'n_splits": '5',
        'shuffle": 'False',
        'random_state': 'None'
    }
}
```

Im Gegensatz zu den anderen Modulen ist der Web Scraping Vorgang nur einmal je Bibliothek notwendig und muss nicht für jedes Projekt von neuem gestartet werden. Die am Ende erzeugte JSON-Datei bildet die Basis für das Extrahieren von Konfigurationsoptionen und kann für beliebige viele Projekte verwendet werden. Da bei diesem Vorgang eine Vielzahl an Websites angesteuert werden, handelt es sich hierbei um das Modul mit der längsten Laufzeit.

Das Python-Skript kann sowohl über eine Entwicklungsumgebung, als auch über die Kommandozeile ausgeführt werden. Um Python-Dateien über die Kommandozeile auszuführen, muss man lediglich den Befehl *python3* oder bei älteren Versionen *python*, gefolgt vom Namen der Python-Datei eingeben. Zusätzlich ist es möglich neben diesem Befehl noch weitere Werte hinzugefügt werden. Diese sogenannten Argumente können im Source Code verarbeitet werden, um den Anwendenden eine Gestaltungsspielraum für die Ausführung zu überlassen. In diesem konkreten Fall wird über die Eingabe des Namens der

Bibliothek entschieden, von welcher Bibliothek die Klassen *gescraped* werden sollen. Damit auch bei unterschiedlichen Schreibweisen einer Bibliothek das Programm ausgeführt wird, sind im Source Code für jede Bibliothek mehrere Möglichkeiten hinterlegt. Das Scraping von mlflow-Klassen erfolgt zum Beispiel über folgenden Befehl:

```
python3 scraping.py mlflow
```

4.3 Parsen der Repositorys

Der erste Schritt, um die Konfigurationsoptionen zu extrahieren, ist eine Umgebung zu schaffen, in der ein zu untersuchendes Git-Repository vorliegt. Über die *GitPython*-Bibliothek wird das Repository geklont, sodass es lokal für die Analyse verfügbar ist. Da sich die Analyse auf Source Code in Python bezieht, werden im nächsten Schritt alle Ordner sukzessiv nach Dateien mit einer *.py*-Endung durchsucht. Diese Python-Dateien werden in einer Liste gespeichert, durch die in der Folge iteriert wird, um die Konfigurationsoptionen auszulesen.

Analog zum Web Scraping kann der gesamte Parse- und Extraktionsprozess ebenfalls über die Kommandozeile gestartet werden. Die zu übergebenen Argumente sind zum Einen der Link des zu untersuchenden Git-Repositorys und zum Anderen die ML-Bibliothek, dessen Konfigurationselemente gefunden werden sollen. Der entsprechende Befehl für ein Beispielprojekt mit mlflow sieht demnach wie folgt aus:

```
python3 main.py https://github.com/user/project scikit-learn
```

Im Gegensatz zum Web Scraping erfolgt der gesamte Parse- und Extraktionsprozess für die zu untersuchenden Bibliotheken generisch. Die Klasse *ConfigOptions* enthält alle Methoden, um aus den Repositorys die Konfigurationsoptionen auszulesen. Für jede der vier Bibliotheken gibt es eine Klasse, die von der *ConfigOptions* abgeleitet ist und ihre Funktionen erbt. Dadurch lässt sich dieses Tool beliebig auf andere Bibliotheken erweitern. Voraussetzung ist jedoch, dass eine JSON-Datei mit den gescrapten Klassen der jeweiligen Bibliothek im entsprechendem Format vorliegt. Möchte man beispielsweise die Funktionalitäten dieses Tools auf die ML-Bibliothek *Keras* anwenden, muss man nach erfolgreichem Scraping, lediglich folgende Codezeilen zur *main.py*-

Datei hinzufügen.

```
class KerasOptions(ConfigOptions):
    def __init__(self, repo):
        ConfigOptions.__init__(self, repo)
        self.library = "keras"
```

4.4 Extraktion der ML-Klassen

Als nächstes werden die Klassen der jeweiligen ML-Bibliothek aus dem Source Code extrahiert. Dafür wird durch die Liste mit Python-Dateien iteriert und entlang der AST-Struktur nach den entsprechenden Klassen gesucht. Der Extraktion-Prozess der ML-Klassen lässt sich in vier Schritte einteilen, die über die Klasse *ASTClasses* aufgerufen werden können.

1. JSON-Datei mit gescrapten Klassen einlesen
2. Code aus Python-Datei als AST parsen
3. Traversierung des AST
   (a) AST-Knoten nach Import-Anweisungen der ML-Bibliothek durchsuchen
   (b) AST-Knoten, die Import-Bezeichnungen enthalten filtern
   (c) Prüfen, ob es sich um ML-Klassen handelt
   (d) ML-Klassen als Dictionary speichern
4. Liste mit Dictionaries zurückgeben

Zunächst wird die JSON-Datei mit den gescrapten Klassen eingelesen und der Python-Code über die *ast*-Bibliothek in das AST-Format umgewandelt. Die Traversierung des Syntax-Baumes erfolgt nach dem Preoder-Verfahren, sodass jeder Knoten eines Teilbaums vollständig durchlaufen wird, bevor der nächste Knoten auf der gleichen Stufe betrachtet wird. Im ersten Schritt werden alle Import-Objekte rausgefiltert, um festzustellen, ob und wie die ML-Bibliothek verwendet wird. Import-Objekte, die die ML-Bibliothek betreffen, werden in einer Liste gespeichert und bilden die Basis für die Suche nach den ML-Klassen. Es gibt verschiedene Möglichkeiten wie Elemente einer ML-Bibliothek zugänglich gemacht werden können. Das folgende Codebeispiel zeigt

fünf Möglichkeiten, die die Verwendung derselben scikit-learn-Klasse *KNeighborsRegressor* über einen jeweils anderen Pfad im Code ermöglichen.

```
import sklearn                                                  # A
import sklearn as skl                                           # B
from sklearn import neighbors                                   # C
from sklearn.neighbors import KNeighborsRegressor               # D
from klearn.neighbors import KNeighborsRegressor as KNN         # E

sklearn.neighbors.KNeighborsRegressor()                         # A
skl.neighbors.KNeighborsRegressor()                             # B
neighbors.KNeighborsRegressor()                                 # C
KNeighborsRegressor()                                           # D
KNN()                                                           # E
```

Die Eindeutigkeit des Pfades ist wichtig, um sicherzustellen, dass es sich bei dem jeweiligen Code-Objekt, um die Klasse einer ML-Bibliothek handelt. Einige Klassen haben geläufige Bezeichnungen, wie zum Beispiel *Pipeline* aus scikit-learn oder *Model* aus Mlflow. Um zu vermeiden, dass irrtümlicherweise falsche Objekte gefunden werden, werden die Code-Objekte aus dem AST auf Basis der Import-Anweisungen der jeweiligen ML-Bibliothek rausgefiltert.

Angenommen das folgende Codebeispiel wird auf Klassen der aus der scikit-learn-Bibliothek untersucht.

```
from sklearn.neighbors import KNeighborsRegressor

def knn_func():
    n_neighbors = 6
    knn = KNeighborsRegressor(n_neighbors, leaf_size=25)
    return knn
```

Für das obige Codebeispiel wird der Syntaxbaum nach Knoten durchsucht, deren Name identisch mit *KNeighborsRegressor* ist und vom AST-Typ *Call* ist. Der AST-Typ versichert, dass ein (Klassen-)Aufruf vorliegt und nicht beispielsweise ein gleichnamiger String ohne Klassenbezug. Ist dies zutreffend, wird geprüft, ob es sich bei diesem Knoten, um eine ML-Klasse handelt oder um einen Pfad, an dessen Ende eine ML-Klasse steht. Ist auch dies gegeben, wird der gesamte Pfad des Objektes beleuchtet, um die Klasse eindeutig zuordnen zu können. Dieser Schritt ist wichtig, da Klassenamen nicht immer

eindeutig sind. So verfügt beispielsweise PyTorch über zwei unterschiedliche *profiler*-Klassen. Die eine Klasse ist Teil des *autograd*-Moduls, während die andere dem *profiler*-Modul zugeordnet ist. Beide verfügen über unterschiedliche Konfigurationsoptionen. An dieser Stelle sollte man sich nicht irritieren lassen, dass die beiden Klassen kleingeschrieben sind. Die Namenskonvention, dass Klassen mit Großbuchstaben beginnen, wird von manchen Bibliotheken nicht immer eingehalten.

Die gefundenen Klasse aus dem Codebeispiel wird als Listenelement in Form eines Dictionaries gespeichert. Das Dictionary enthält nun alle benötigten Informationen, um im nächsten Schritt die Konfigurationsoptionen auszulesen. Dazu gehört der vollständige Pfad mit dem Namen der Klasse, der verwendete Pfad inklusive Alias, sowie die Datei, indem die Klasse gefunden wurde. Zudem wird das AST-Objekt gespeichert, aus dem sich weitere Informationen wie Parameter und Zeilennummer auslesen lassen. Für das Codebeispiel ergibt sich folgender Listeneintrag.

```
{'file': 'path/file.py',
 'class': 'sklearn.neighbors.KNeighborsRegressor',
 'class alias': 'KNeighborsRegressor',
 'object': <ast.Assign object at 0x106178370>}
```

4.5 Extraktion und Verarbeitung der Konfigurationsoptionen

Der nächste Vorgang gliedert sich in zwei Teilschritte, bei dem die Ergebnisse aus 4.2 und 4.4 miteinander verbunden werden.

1. Extraktion der Parameterwerte aus AST-Objekt des Dictionaries
2. Zuordnung der Werte zu den gescrapten Parametern aus JSON-Datei

Zunächst wird durch die Liste mit Dictionaries aus 4.4 iteriert. Jedes Dictionary wird zusätzlich mit einem Eintrag zu den gescrapten Parametern der jeweiligen Klasse ergänzt und an die Methode *get_parameters()* der Klasse *ASTParameters* übergeben. In dieser Klasse wird der Syntaxbaum des Code-Objekts traversiert, um die übergebenen Parameterwerte in der entsprechenden Reihenfolge zu extrahieren. Die Klasse verfügt über eine Vielzahl an Methoden, die die einzelnen Knoten des Syntaxbaums entsprechend ihres Typs

verarbeiten können. Da die gesuchten Werte Parameter von Klassenaufrufen sind, ist die Behandlung von Knoten des AST-Typs *Call* von besonderer Bedeutung. Hinter dem AST-Objekt des Dictionaries aus 4.4 verbirgt sich folgender Syntaxbaum:

```
Assign(
    targets=[
        Name(
            id='knn',
            ctx=Store())],
    value=Call(
        func=Name(
            id='KNeighborsRegressor',
            ctx=Load()),
        args=[
            Name(
                id='n_neighbors',
                ctx=Load())],
        keywords=[
            keyword(
                arg='leaf_size',
                value=Constant(
                    value=25))]))
```

Ein Code-Objekt vom AST-Typ *Call* besteht aus 3 Teilknoten. Der Knoten *func* gibt den Namen Aufrufs an. Ist an dieser Stelle die *id* des Knoten identisch zur ML-Klasse, werden im nächsten Schritt die Parameter extrahiert. Diese liegen zweigeteilt vor: Im Knoten *args* befinden sich Argumente, die nach ihrer Position übergeben und zugeordnet werden. Im Gegenzug werden im Knoten *keywords* Argumente zusammen mit der Parameterbezeichnung übergeben, sodass direkt kenntlich ist, um welchen Parameter es sich handelt.

Im gegebenen Beispiel wird die Variable *n_neighbors* als Positionsargument vom AST-Typ *Name* klassifiziert. Wenn Parameter vom AST-Typ *Name* oder *Attribute* vorliegen, werden diese zusätzlich vermerkt, um an späterer Stelle mittels Datenflussanalyse mögliche Werte für die Variablen zu ermitteln. Das Keyword-Argument für *leaf_size* kann hingegen direkt zugeordnet werden und ist auch konstant.

Zusätzlich zur Ermittlung der Parameter wird in der *AST-Parameters*-Klasse auch geprüft, ob die Instanziierung der Klasse einer Variable zugewie-

sen wird. Im gegebenen Beispiel wird die Instanz der Klasse (*value*) von der Variablen *knn* (*target*) referenziert. Darüber hinaus kann eine Zuweisung auch über andere Wege als Keyword-Argument erfolgen: bei der Definition einer Klasse oder Funktion und beim Aufruf eines Objektes.

Im zweiten Teil werden die gefundenen Parameter im Code mit den gescrapten aus der JSON-Datei zusammengeführt. Zunächst wird durch die Liste an Positionsargumente iteriert, sodass die Positionsargumente der Reihe nach, den gescrapten Parameter zugeordnet werden können. Für die Keyword-Argumente wird geprüft, ob das Keyword als Parameter vorhanden ist. Bei falschen Keyword-Bezeichnungen, also Keywords, die es als Parameter in der Spezifikation gar nicht gibt, kann keine Zuordnung zu den gescrapten Parametern erfolgen.

Darüber hinaus gibt es drei Besonderheiten in der Spezifikation, die vom Algorithmus bei der Zuordnung der Parameter beachtet werden müssen: *, **args* und ***kwargs*. Befindet sich in der Spezifikation der Parameter ein Sternchen, dürfen danach keine Positionsargumente mehr übergeben werden. Bei der Spezifikation der Parameter der *KNeighborsRegressor*-Klasse befindet sich dieses Sternchen an zweiter Stelle. Dementsprechend müssen alle Parameter ab dem zweiten Parameter *leaf_ size* als Keyword-Argumente übergeben werden. **args* werden verwendet, wenn die Anzahl der Übergabeparameter variabel bleiben soll oder die Parameter in Form eines Tupels oder einer Liste vorliegen. Im ersten Fall kann die Länge der übergebenen Parameter, die Länge der Parameter in der Spezifikation also überschreiten. Die *überzähligen* Parameter werden zusammen mit dem letzten übergebenen Parameter als Tupel zusammengefasst. Die Übergabe als Tupel oder als Liste erfolgt mit einem anführenden Sternchen. ***kwargs* hingegen sind eine Möglichkeit, um Dictionaries zu übergeben, die als Schlüssel Parameter der Klasse enthalten können.

Nachdem die Parameter erfolgreich zugeordnet wurden, gibt die *ASTParameters*-Klasse ein Dictionary zurück. Es enthält wie das Dictionary aus 4.4 den Namen der Python-Datei, den vollständigen Pfad der Klasse und das AST-Objekt. Ergänzt wird das Dictionary mit Informationen zu den Parametern. Variable Parameter werden zusätzlich gesondert in einem Dictionary aufgeführt. Da die möglichen Werte der variablen Parameter noch unbekannt sind, werden diese mit *None* gekennzeichnet. Sofern vorhanden, wird die Variable, die die Klasse referenziert, ebenfalls aufgeführt, sodass sich folgendes Dictionary ergibt:

```
{'file': 'path/file.py',
```

```
'class': 'sklearn.neighbors.KNeighborsRegressor',
'object': <ast.Assign object at 0x106178370>},
'parameter': [(None, 'n_neighbors'),
              ('leaf_size', '25')],
'variable parameters': {'n_neighbors': None},
'variable': 'knn'}
```

4.6 Ermittlung variabler Konfigurationswerte

Zunächst wird durch die Liste mit Dictionaries aus 4.5 iteriert. Die Ermittlung der variablen Parameterwerte erfolgt über die Methode *get_parameter_value()* der *DataFlowAnalysis*-Klasse. Beim Aufruf der Methode wird für dieses Beispiel das Dictionary zusammen mit der Parametervariable *n_neighbors* übergeben. Die anschließende Datenflussanalyse lässt sich in zwei Bereich einteilen:

1. Sammeln von Informationen: Kontrollflüsse, Aufrufpfade und Funktionsaufrufe
2. Analyse der Informationen: SSA, Constant Propagation, Klassen- und Funktionsparameter

Im ersten Schritt wird über die *CFGBuilder*-Klasse des statischen-Analyse-Framework *Scalpel* der Kontrollflussgraph für die Python Datei erstellt und in einer Liste gespeichert. Der Kontrollflussgraph stellt alle globalen Pfade dar, die während der Ausführung des Programms durchlaufen werden können. Da Klassen- und Funktionskörper sich jedoch nicht auf dieser Ebene befinden, wird zusätzlich jede Klasse und Funktion der globalen Ebene traversiert und die Kontrollflussgraphen dieser Objekte ebenfalls in der Liste gespeichert. Dieser Vorgang wiederholt sich für jede weitere Ebene bis schließlich alle Kontrollflussgraphen der Klassen und Funktionen der Liste hinzugefügt wurden.

Im nächsten Schritt werden die möglichen Pfade, mit denen man die einzelnen Klassen und Funktionen erreichen kann, ermittelt. Das folgende Beispiel zeigt verschiedene Möglichkeiten, um ein Objekt aufzurufen. Die globale Funktion *func_c* kann von überall direkt mit ihrem Namen aufgerufen werden (*call 1*). Die Funktionen *func_a* und *func_b* der Klasse *A* werden innerhalb der Klasse mit einem vorgesetzten *self.* angesprochen (*call 2*). Außerhalb der Klasse werden sie nur über die Angabe des jeweiligen Namens der Klasse erreicht (*call 3*). Übergibt man einer Klasse beim Aufruf einen Wert, wird dieser Wert über die Python-Methode *__init__()* verarbeitet. Um Klassen- und

Funktionsaufrufe richtig zuzuordnen und die übergebenen Parameter richtig zu verarbeiten, ist es daher wichtig, den Typ und die Objektzugehörigkeit zu kennen. Für jede Klasse und jede Funktion wird ein Dictionary erstellt, dessen Werte die verschiedenen Aufrufpfade sind.

```
class A:
    def __init__(self, variable):
        self.cls_variable = variable

    def func_a(self):
        pass

    def func_b(self):
        self.func_a()                    # call 2

def func_c():
    pass

func()                                   # call 1
A(1).func_a()                            # call 3
```

Im Anschluss wird durch die Liste der Kontrollflussgraphen iteriert und alle enthaltenen Anweisungen als AST geparst. Der abstrakte Syntaxbaum wird traversiert, um Funktionsaufrufe rauszufiltern und in einer Liste zu speichern. Nachdem alle Kontrollflussgraphen durchlaufen wurden, enthält die Liste sämtliche Funktionsaufrufe der Datei im AST-Format. Falls nun die gesuchte Variable als Parameter in der Definition einer Funktion steht, können auf Basis der möglichen Pfade der Funktion, die Funktionsaufrufe gefunden werden. Dadurch kann der Wert einer Variable bzw. des Parameters über den gesamten Kontrollfluss der Datei verfolgt werden.

Nachdem im ersten Teil die notwendigen Informationen gesammelt wurden, werden diese im zweiten analysiert. Für jeden Kontrollflussgraphen der Datei werden über die *compute_SSA()*-Methode der Scalpel-Klasse *SSA* Zuweisungen und Werte der enthaltenen Variablen ermittelt. Dafür macht die Methode von den Datenflussanalyse-Techniken *Static Single Assignment* und *Constant Propagation* Gebrauch. Als Ergebnis erhält man eine Liste an Dictionaries. Die Schlüssel sind Variablen, die Ziel einer Zuweisung sind, während die Werte des Dictionaries, den jeweiligen Wert der Zuweisung entsprechen. Wird der gesuchten Variable ein Wert zugewiesen, wird dieser in der Liste möglicher Werte gespeichert. Handelt es sich zudem um einen variablen Wert

vom AST-Typ *Name* oder *Attribute*, wird im Anschluss auch für diesen Wert der gesamte zweite Teil der Datenflussanalyse ausgeführt.

In diesem Zusammenhang stößt man an mehrere Grenzen des Scalpel-Frameworks, die im folgenden Codebeispiel dargestellt werden:

```
class A:
    def __init__(self):
        a, b = 1, 2                          # issue 1
        c = 3                                # issue 2
        self.cls_variable = c                # issue 3
        c = 4
```

Die aktuelle Version von Scalpel (Version ???) kann keine Zuweisungen an Tupel verarbeiten, sodass für die Variablen *a* und *b* von Scalpel keine Wertzuweisungen ausgegeben werden (*issue 1*). Zudem speichert Scalpel nur die letzte Zuweisung. Für die Variable *c* beträgt der Wert bei der Ausgabe demnach *4*, obwohl der Variable vorher ein anderer Wert zugewiesen wird (*issue 2*). Das größte Problem ist jedoch, dass Scalpel Zuweisungen an Objekte des AST-Typs *Attribute* nicht verarbeitet und den Wert für *self.cls_variable* nicht ermitteln kann (*issue 3*). Da die Lösung dieses Problems essenziell für die Datenflussanalyse ist, wurde der Source Code der *compute_SSA()*-Methode untersucht und angepasst, sodass nun auch Zuweisungen an Instanzvariablen wie *self.cls_variables* verarbeitet werden können. Leider beträgt der Wert für *c* wie in *issue 2* beschrieben zum Zeitpunkt der Zuweisung der SSA-Analyse von Scalpel *4*, obwohl *3* richtig ist. Für diese Probleme wurden jeweils ein Issue, teilweise mit Lösungsansatz, im Git-Repository von Scalpel erstellt. Zum Zeitpunkt der Abgabe der Bachelorarbeit wurden die Probleme zwar zur Kenntnis genommen, jedoch noch nicht behoben.

Im nächsten Schritt werden die Funktionsdefinitionen der Datei untersucht, um zu überprüfen, ob die gesuchte Variable als Parameter übergeben wird. Ist die gesuchte Variable in der Definition der Funktion vorhanden, müssen einige Besonderheiten bei der Behandlung der Argumente berücksichtigt werden. Die Parameter einer Funktion können fünf unterschiedliche AST-Typen besitzen: *arg*, *vararg*, *kwonlyargs*, *posonlyarg* oder *kwarg*. Ähnlich wie die Argumente bei *ast.Call* aus 4.4 müssen die AST-Typen unterschiedlich behandelt werden, damit die Parametervariable beim Aufruf der Funktion eindeutig zugeordnet werden kann. Im folgenden Codebeispiel werden die verschiedenen Typen mit ihrem Namen kenntlich gemacht:

```
def func(posonlyarg1, posonlyarg2, /, arg1, arg2,
         arg3 = 1, *vararg, kwonlyarg1, kwonlyarg2, **kwarg):
    return

func("posonlyarg1", "posonlyarg2", arg2=2, *["arg1"],
     kwonlyarg1=1, kwonlyarg2=2, **{"arg3": 3, "new_arg": 0})
```

Positional-only arguments sind Argumente, die vor der /-Markierung stehen und dürfen nicht mit einem Keyword-Argument übergeben werden. *Keyword-only arguments* sind das Gegenstück. Sie müssen mit einem Keyword übergeben werden und befinden sich in der Definition immer nach *varargs*. *Args* sind Argumente bei denen nicht festgelegt ist, ob sie mit einem Keyword übergeben werden. Sie werden ansonsten ihrer Position nach zugeordnet. Die Typen *vararg* und *kwarg* referenzieren die übergebenen **args*- und ***kwargs*-Parameter, die in 4.5 besprochen wurden. Ist die gesuchte Variable in der Definition der Signatur vorhanden, besteht die Möglichkeit, dass ihr wie bei *arg3* ein Default-Wert zugewiesen wird, der dementsprechend zur Liste möglicher Werte der Parametervariablen hinzugefügt werden muss.

Im letzten Schritt der Datenflussanalyse wird, sofern die gesuchte Variable ein Funktionsparameter ist, durch die Liste mit Funktionsaufrufen iteriert. Jeder Funktionsaufruf wird mit den möglichen Werten aus der Liste an Dictionaries mit möglichen Aufrufspfaden verglichen und einer Funktion zugeordnet. Für jeden Aufruf der Funktion wird je nach Argumenttyp und Position der gesuchten Parametervariable, der übergebene Wert ausgelesen. Handelt es sich bei einem übergebenen Wert erneut um eine Variable, wird im Anschluss auch für diesen Wert der gesamte zweite Teil der Datenflussanalyse wiederholt.

Nach erfolgreichem Durchlauf gibt die *get_parameter_value()*-Methode der *DataFlowAnalysis*-Klasse eine Liste mit möglichen Werten zurück. Die Liste wird im Anschluss in ein Dictionary umgewandelt, dessen Schlüssel aufsteigende Zahlen sind. Schlussendlich wird eine JSON-Datei erstellt, die alle gesuchten Konfigurationsoptionen des Git-Repositories enthält. Für das *KNeighborsRegressor*-Beispiel sieht der Eintrag wie folgt aus:

```
{'file': 'path/file.py',
 'class': 'sklearn.neighbors.KNeighborsRegressor',
 'parameter': {'n_neighbors': 'n_neighbors',
               'leaf_size': '25'}
```

```
'variable parameters': {'n_neighbors': {'0': '6'}},
'variable': 'knn',
'line no': '4'}
```

Kapitel 5

Evaluation

Im folgenden Kapitel wird das soeben beschriebene Tool evaluiert. In der Tabelle 5.1 wird für jede ML-Bibliothek die Anzahl an Klassen und Parameter dargestellt.

ML-Bibliothek	Version	Klassen	Parameter
TensorFlow	2.9.1	844	3711
PyTorch	1.11.0	408	1340
scikit-learn	1.1.1	262	2081
Mlflow	1.26.1	29	96

Tabelle 5.1: Scraping Ergebnisse der ML-Dokumentation

Die Anzahl der Klassen steht im Verhältnis zum Dauer des Scraping-Prozesses. Während das Scraping der Klassen aus Mlflow-Dokumentation die geringste Laufzeit hatte, dauerte der Prozess bei TensorFlow deutlich am längsten an. Zu beachten ist, dass der Erfolg und Reproduzierbarkeit des Scraping-Moduls stark mit der Website-Struktur der Dokumentation zusammenhängen, da es nicht Robust gegenüber Änderungen ist. Ändert sich das Layout indem beispielsweise HTML-Elemente anders strukturiert werden, so kann es sein, dass der Scraping-Algorithmus nicht mehr die gewünschten Ergebnisse liefert. Gleichzeitig mussten insgesamt acht Klassen von TensorFlow, PyTorch und Mlflow manuell zum Scraping-Ergebnis hinzugefügt werden, da sie nicht wie die anderen Klassen in die Dokumentation eingebettet und mit den gleichen HTML-Tags versehen wurden.

Die Verwendung von Konfigurationsoptionen von ML-Bibliotheken im Source Code ist nicht umfassend erforscht, weshalb es keine Datensätze oder Vergleichswerte gibt, um die Genauigkeit des Algorithmus bewerten zu können. Im

???-Ordner des Git-Repositorys befinden sich verschiedene Codebeispiele mit Konfigurationsoptionen, die vom Algorithmus erkannt werden müssen. Mit Hilfe des *pytest*-Frameworks werden die gefundenen Konfigurationsoptionen aus der JSON-Datei, die der Algorithmus als Ausgabe erzeugt, mit dem *richtigen* Ergebnis verglichen. Falls sich die Ergebnisse gibt pytest eine Fehlermeldung aus. Um darüber hinaus eine aussagekräftige Bewertung abzugeben, wird sich am Evaluationsansatz von Rabkin and Katz orientiert. Dafür werden verschiedene Softwareprojekte mit dem entwickelten Algorithmus analysiert und das Ergebnis mit einer manuellen Analyse verglichen. Für die manuelle Analyse wird der Source Code auf die Verwendung von Konfigurationsoptionen der zu untersuchenden ML-Bibliothek händisch untersucht.

Die Auswahl der Softwareprojekte erfolgt nach folgenden Kriterien:

- Open Source Git Repository mit mindestens 200 Sternen
- Verwendung von mindestens 25-100 Klassen einer ML-Bibliothek
- Verwendung von variablen Parametern beim Aufruf der Klassen
- Code ist syntaktisch fehlerfrei und kann als AST geparst werden

Insgesamt werden sieben Softwareprojekte untersucht:

- Real-Time Voice Cloning
- AutoGluon
- TensorForce
- MLServer

Projekt	TensorForce	RTVC	AutoGluon	MLServer
ML-Bibliothek	TensorFlow	PyTorch	scikit-learn	Mlflow
ML-Klassen	55	86	65	30
gefunden	55	86	65	30
Anteil	100%	100%	100%	100%
Parameter	105	165	98	57
gefunden	105	165	98	57
Anteil	100%	100%	100%	100%
davon variabel	58	78	32	11
Wertzuweisungen	96	83	19	-
gefunden	76	80	19	-
Anteil	79%	96%	100%	-

Tabelle 5.2: Gefundene Konfigurationsoptionen

Kapitel 6

Fazit

Anhang A

Abstract Syntax Tree

```
Module(
    body=[
        ImportFrom(
            module='sklearn.ensemble',
            names=[
                alias(
                    name='GradientBoostingClassifier')],
            level=0),
        Assign(
            targets=[
                Name(
                    id='gbc',
                    ctx=Store())],
            value=Call(
                func=Name(
                    id='GradientBoostingClassifier',
                    ctx=Load()),
                args=[],
                keywords=[
                    keyword(
                        arg='n_estimators',
                        value=Constant(
                            value=20)),
                    keyword(
                        arg='learning_rate',
                        value=Constant(
                            value=0.05)),
                    keyword(
                        arg='max_features',
                        value=Constant(
```

```
                value=2)),
        keyword(
            arg='max_depth',
            value=Constant(
                value=2)),
        keyword(
            arg='random_state',
            value=Constant(
                value=0))]))])
```

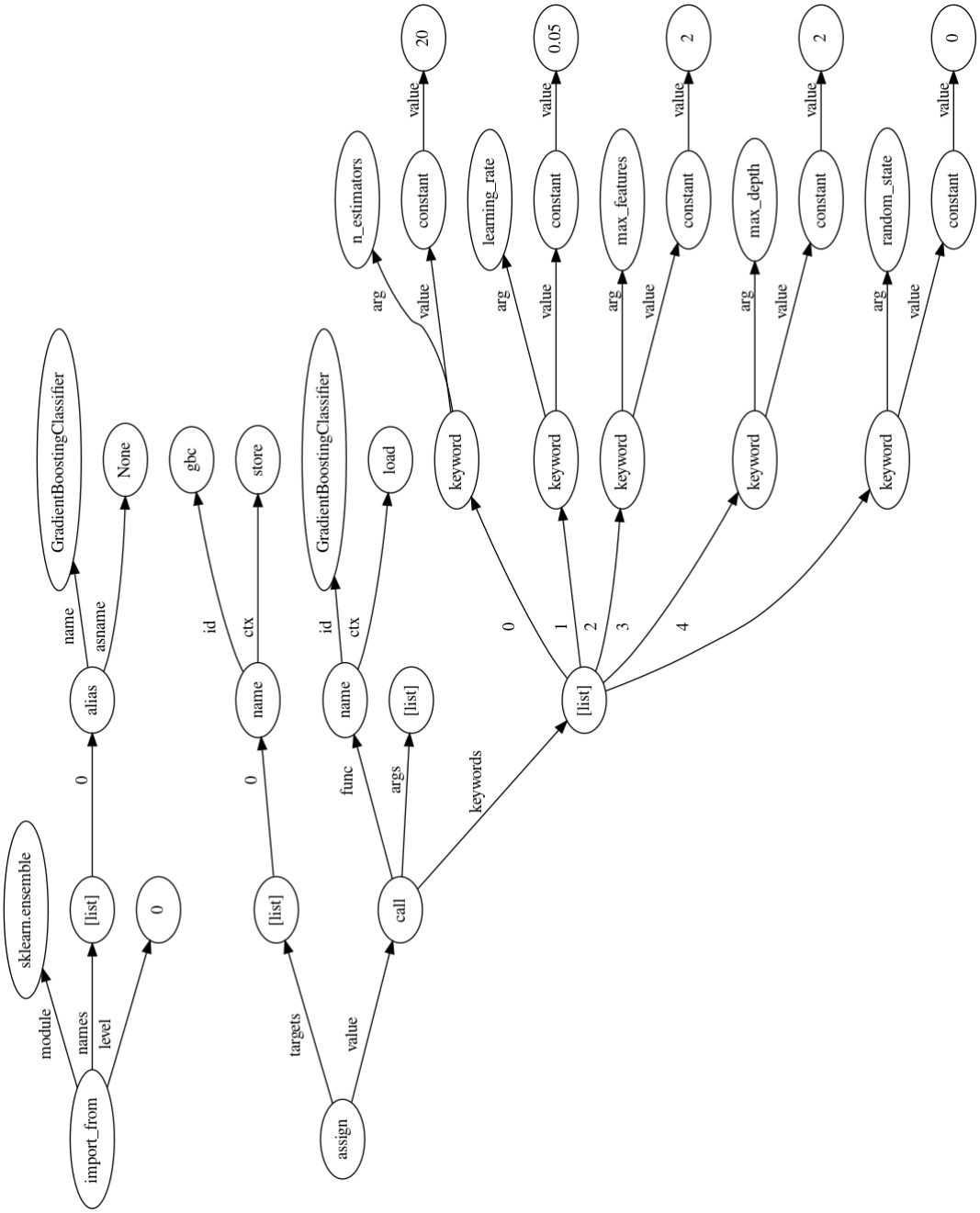

Abbildung A.1: Abstract Syntax Tree

Literaturverzeichnis

Răzvan Andonie. Hyperparameter optimization in learning systems. *Journal of Membrane Computing*, 1(4):279–291, 2019. doi: 10.1007/s41965-019-00023-0. URL `https://doi.org/10.1007/s41965-019-00023-0`. 2.3.2

Alexandru G Bardas et al. Static code analysis. *Journal of Information Systems & Operations Management*, 4(2):99–107, 2010. 2.1

Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 295–306, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786869. URL `https://doi.org/10.1145/2786805.2786869`. 3

TIOBE Software BV. Tiobe index for march 2022. `https://www.tiobe.com/tiobe-index/`, 2022. Online; accessed 22-March-2022. 1.2

Rabiyatou Diouf, Edouard Ngor Sarr, Ousmane Sall, Babiga Birregah, Mamadou Bousso, and Sény Ndiaye Mbaye. Web scraping: State-of-the-art and areas of application. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 6040–6042, 2019. doi: 10.1109/BigData47090.2019.9005594. 2.4

Zhen Dong, Artur Andrzejak, David Lo, and Diego Costa. Orplocator: Identifying read points of configuration options via static analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 185–195, 2016. doi: 10.1109/ISSRE.2016.37. 1.1, 1.2, 3

Daniel Glez-Peña, Anália Lourenço, Hugo López-Fernández, Miguel Reboiro-Jato, and Florentino Fdez-Riverola. Web scraping technologies in an API world. *Briefings in Bioinformatics*, 15(5):788–797, 04 2013. ISSN 1467-5463. doi: 10.1093/bib/bbt026. URL `https://doi.org/10.1093/bib/bbt026`. 2.4

Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009. 2.1

Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 754–762, Bejing, China, 22–24 Jun 2014. PMLR. URL `https://proceedings.mlr.press/v32/hutter14.html`. 2.3.2

Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. Preffinder: Getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 151–162, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. doi: 10.1145/2642937.2643009. URL `https://doi.org/10.1145/2642937.2643009`. 3

Kaggle. State of machine learning and data science 2021. `https://www.kaggle.com/kaggle-survey-2021`, 2021. Online; accessed 05-April-2022. (document), 2.3.1, 4.1

KDnuggets. Python leads the 11 top data science, machine learning platforms: Trends and analysis. `https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html`, 2019. Online; accessed 04-April-2022. 2.3.1

Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The python static analysis framework. *arXiv preprint arXiv:2202.11840*, 2022. (document), 2.1

Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. *IEEE Transactions on Software Engineering*, 44(12):1269–1291, 2018. doi: 10.1109/TSE.2017.2756048. 1.2, 3

Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386, 2020. 2.3

Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020. doi: 10.3102/1076998619872761. URL `https://doi.org/10.3102/1076998619872761`. 4.1

L.L. Pollock and M.L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, 1989. doi: 10.1109/32.58766. 2.2

Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 131–140, New York, NY, USA, 2011. Association for Computing Machinery. doi: 10.1145/1985793.1985812. URL `https://doi.org/10.1145/1985793.1985812`. 1.1, 1.2, 3, 5

Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, Apr 2020. ISSN 2078-2489. doi: 10.3390/info11040193. URL `http://dx.doi.org/10.3390/info11040193`. 1.2, 2.3, 2.3.1, 4.1

scikit-learn developers. scikit-learn api. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression`, 2022. Online; accessed 25-March-2022. 1.1

I. Stančin and A. Jović. An overview and comparison of free python libraries for data mining and big data analysis. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 977–982, 2019. doi: 10.23919/MIPRO.2019.8757088. 2.3.1

G. Varoquaux, L. Buitinck, G. Louppe, O. Grisel, F. Pedregosa, and A. Mueller. Scikit-learn: Machine learning without learning the machinery. *GetMobile: Mobile Comp. and Comm.*, 19(1):29–33, jun 2015. ISSN 2375-0529. doi: 10.1145/2786984.2786995. URL `https://doi.org/10.1145/2786984.2786995`. 4.1

Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 159–172, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043572. URL `https://doi.org/10.1145/2043556.2043572`. 1.1

Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018. 4.1

Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 152–163, New York, NY, USA, 2014. Association

for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568251. URL https://doi.org/10.1145/2568225.2568251. 1.1

Bo Zhao. Web scraping. *Encyclopedia of big data*, pages 1–3, 2017. 2.4, 2.4