Karlsruhe Institute of Technology

Vergleich verschiedener Sprachmodelle für den Einsatz in automatisierter Rückverfolgbarkeitsanalyse

Bachelorarbeit
von

Tim Noah Lachenicht

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr.-Ing. Anne Koziolek
Betreuender Mitarbeiter:	M.Sc. Tobias Hey

Bearbeitungszeit: 28.06.2022 – 28.10.2022

Kurzfassung

Informationen über logische Verbindungen zwischen Anforderungen und ihrer Umsetzung in Quelltext sind nützlich für viele Aufgabenstellungen der Softwareentwicklung. Sie können beispielsweise die Wartung von Software bei Anforderungs-Änderungen erleichtern. Diese Rückverfolgbarkeitsverbindungen können im Zuge einer Rückverfolgbarkeitsanalyse ermittelt werden. Verfahren wie *FTLR* führen eine automatisierte Rückverfolgbarkeitsanalyse durch. FTLR erkennt Rückverfolgbarkeitsverbindungen mithilfe eines Vergleichs von Repräsentationen von Anforderungen und Quelltext. Bislang setzt FTLR das Sprachmodell *fastText* zur Repräsentation von Anforderungen und Quelltext ein. Der Ansatz fastText besitzt jedoch Schwachstellen. Das Sprachmodell ist nicht in der Lage verschiedene Bedeutungen eines Wortes zu repräsentieren. Außerdem wurde es nicht auf Quelltext vortrainiert. In dieser Arbeit wurde untersucht, ob sich alternative Sprachmodelle ohne diese Schwachstellen besser zum Einsatz in FTLR eigenen als fastText. In einem Experiment auf fünf Vergleichsdatensätzen für die Rückverfolgbarkeitsanalyse wurden die Ergebnisse der beiden alternativen Sprachmodelle UniXcoder und Wikipedia2Vec mit fastText verglichen. Das Sprachmodell *UniXcoder* eignet sich auf den Vergleichsdatensätzen *iTrust* und *LibEST* besser als fastText. Das Sprachmodell *Wikipedia2Vec* eignet sich auf keinem der eingesetzten Vergleichsdatensatz besser als fastText. Im Durchschnitt über alle verwendeten Testdatensätze eignet sich fastText besser für den Einsatz in FTLR als UniXcoder und Wikipedia2Vec.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Software wird auf Basis von Anforderungen entwickelt. Diese spezifizieren meist in Form von natürlicher Sprache, welche Funktionen die erwartete Software anbieten soll. Anforderungen werden in der resultierenden Software durch Quelltextelemente umgesetzt. Das Wissen über Verbindungen zwischen Anforderungen und ihrer bestehenden Umsetzung ist für viele softwaretechnische Aufgabenstellungen sehr hilfreich [CHGZ12]. Zu wissen, welches Quelltextelement welche Anforderung implementiert, kann beispielsweise eine Wartung bei einer Anforderungsänderung unterstützen. Eine derartige Verbindung zwischen Anforderungen und Quelltextelementen bezeichnet man auch als Rückverfolgbarkeitsverbindung (engl.: trace link, TL). TLs werden im Zuge einer Rückverfolgbarkeitsanalyse (engl.: trace link analysis, TLA) ermittelt. Erfolgt die TLA im Anschluss an die Erstellung aller Anforderungen und Quelltextelemente spricht man auch von einer nachträglichen Rückverfolgbarkeitsanalyse (engl.: trace link recovery, TLR). Eine manuelle TLR ist mit sehr hohem Aufwand verbunden. Das automatisierte Ermitteln von TLs durch eine automatisierte nachträgliche Rückverfolgbarkeitsanalyse (engl.: automated trace link recovery, ATLR) eliminiert diesen erheblichen manuellen Zeitaufwand.

Das ATLR-Verfahren *Fine-grained Traceability Link Recovery* (FTLR) [HCWT21] ermittelt TLs zwischen Anforderungen und Quelltextklassen. Hierfür vergleicht das Verfahren die Semantik von Anforderungen und Quelltext. Ähnelt sich die Semantik, wird eine TL zwischen Anforderung und Quelltextklasse festgelegt. Der automatisierte semantische Vergleich erfolgt auf Basis einer Repräsentation der Semantik von Anforderungen und Quelltextklassen. Zur Repräsentation dieser Semantik nutzt FTLR das Sprachmodell (engl.: language model, LM) *fastText* [JGBM16]. LMs sind in der Lage, die Bedeutung von sprachlichen Einheiten zu repräsentieren und basieren auf maschinellem Lernen (engl.: machine learning, ML). fastText ist ein LM, welches auf der sprachlichen Ebene von Worten arbeitet und diese durch Vektoren repräsentiert. Der Ansatz von fastText basiert auf der Annahme, dass semantisch ähnliche Wörter häufig im selben Kontext und somit häufig in Kombination mit denselben anderen Wörtern auftreten [MCCD13]. Damit repräsentiert fastText zwei Worte genau dann ähnlich, wenn sie in Trainingstexten häufig gemeinsam mit denselben anderen Wörtern auftreten. Dieser Ansatz besitzt nicht unerhebliche Schwachstellen. Treten zwei semantisch ähnliche Wörter nicht häufig in Kombination mit denselben anderen Wörtern auf, werden sie von fastText nicht als ähnlich repräsentiert. Außerdem repräsentiert fastText jedes Wort statisch durch einen einzelnen Vektor. Natürlichsprachliche Worte können jedoch mehrere unterschiedliche, vom Kontext abhängende Bedeutungen besitzen. fastText ist nicht in der Lage zu repräsentieren, dass Worte mehrere Bedeutungen haben.

In die ermittelte Wortrepräsentation können stattdessen alle verschiedenen Wortbedeutungen einfließen, was zu einer ungenauen Repräsentation führen kann. fastText wird in FTLR zur Repräsentation von Quelltextklassen eingesetzt. Das LM wurde jedoch nicht auf Quelltext trainiert. Damit ist fraglich, ob sich fastText zur Repräsentation von Quelltextklassen eignet. fastText besitzt somit drei Schwachstellen. Das LM repräsentiert ähnliche Worte, die häufig in anderen Kontexten auftreten, nicht ähnlich. Außerdem ermittelt fastText statische Wortrepräsentationen und wurde nicht auf Quelltext trainiert. Diese drei Schwachstellen können zu Ungenauigkeiten in der von fastText ermittelten Repräsentation von Anforderungen und Quelltextklassen führen. Da FTLR diese zur Ermittlung von TLs nutzt, können sich Ungenauigkeiten des fastText-Verfahrens negativ auf die Güte der Ergebnisse von FTLR auswirken.

1.1 Zielsetzung

Diese Bachelorarbeit setzt sich zum Ziel, verschiedene alternative LMs auf ihre Eignung zum Einsatz in FTLR zu untersuchen. Diese zu untersuchenden LMs sollen in der Lage sein, die Bedeutung von Anforderungen und Quelltext geeignet für die Verwendung in FTLR zu repräsentieren. Außerdem sollen sie Schwachstellen von fastText eliminieren. Nach der Auswahl der zu betrachtenden LMs besteht das nächste Teilziel im Entwurf ihrer Integration in FTLR. Anschließend gilt es herauszufinden, ob sie sich besser für den Einsatz in FTLR eignen als fastText.

1.2 Aufbau der Arbeit

Kapitel 2 beschäftigt sich mit den theoretischen Grundlagen, die zum Verständnis dieser Arbeit nötig sind. Anschließend wird in Kapitel 3 auf Arbeiten eingegangen, deren Thematiken zum Gegenstand dieser Bachelorarbeit verwandt sind. Kapitel 4 stellt das Verfahren FTLR vor, in welches verschiedene LMs integriert werden. Welche LMs im Rahmen dieser Arbeit untersucht werden, wird in Kapitel 5 analysiert. Anschließend werden ebenfalls in Kapitel 5 Entwurfsentscheidungen in Bezug auf die Einbindung der zu untersuchenden LMs getroffen. In Kapitel 6 werden Implementierungsdetails der Integration der ausgewählten LMs in FTLR aufgezeigt. Kapitel 7 beschreibt die Evaluation der Ergebnisse von FTLR, die mit den untersuchten LMs im Vergleich zu fastText möglich sind. Daraus sollen Schlussfolgerungen gezogen werden, ob sich die alternativen LMs tatsächlich besser für den Einsatz in FTLR eignen als fastText. Abschließend fasst Kapitel 8 diese Bachelorarbeit zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

2 Grundlagen

In diesem Kapitel werden grundlegende Themen und Konzepte behandelt, auf die sich diese Arbeit stützt. Zunächst soll in diesem Zuge auf die Problemstellung der TLA eingegangen werden. Die TLR ist eine Variante der TLA. Sie kann auch als Problemstellung der Informationsrückgewinnung (engl.: information retrieval, IR) aufgefasst werden. IR wird daher im nächsten Schritt betrachtet. TLR und IR beziehen sich im Rahmen dieser Arbeit auf Artefakte bzw. Dokumente, die (zum Teil) in natürlicher Sprache verfasst sind. Natürliche Sprache und ihre Verarbeitung stellt aus diesem Grund den darauf folgenden Themenblock dar. Ein LM repräsentiert natürliche Sprache. Diese Repräsentation kann für die Weiterverarbeitung im Rahmen einer ATLR genutzt werden. LMs basieren auf ML. Viele LMs nutzen ein neuronales Netz (engl.: neural network, NN). Daher soll vor der Einführung von LMs zunächst auf ML und NNs eingegangen werden.

2.1 Rückverfolgbarkeitsanalyse

Softwareprodukte bestehen aus Artefakten und werden auf Basis von Artefakten entwickelt. Beispielsweise werden zu Beginn des Entwicklungsprozesses Anforderungsartefakte erstellt und das System auf ihrer Basis durch Entwurfsartefakte definiert. Artefakte beschreiben eine Einheit von Daten unbestimmter Granularität. Ein Artefakt besitzt einen Typ, der es klassifiziert. Beispiele hierfür stellen „class“ oder „requirement“ dar. Rückverfolgbarkeit ist eine Eigenschaft einer Menge von Artefakten. Sie liegt vor, wenn die Möglichkeit besteht, Artefakte miteinander in Verbindung zu setzen und diese Verbindungen zu beschreiben [GCHH+12]. Eine gerichtete Verbindung zwischen einem Quell-Artefakt und einem Ziel-Artefakt wird auch als TL bezeichnet. In praktischen Anwendungen existiert zu jeder TL in der Regel eine inverse TL. Denn häufig ist es möglich, in bidirektionaler Natur vom Quell-Artefakt auf das Ziel-Artefakt zu schließen und umgekehrt. Auch eine TL besitzt einen Typ, der sie klassifiziert. Beispiele hierfür stellen „implements“ oder „tests“ dar. Die Menge an Artefakten eines Softwaresystems besitzt in der Regel die Eigenschaft der Rückverfolgbarkeit. Denn die Artefakte eines Softwaresystems sind bereits von Natur aus logisch miteinander verknüpft. Beispielsweise werden die Anforderungen in Anforderungsartefakten durch Quelltextartefakte umgesetzt. Quelltextartefakte werden wiederum durch Dokumentationsartefakte beschrieben. Bei der TLA sollen TLs zwischen Artefakten explizit definiert werden, um sie für bestimmte Softwareentwicklungsaktivitäten zu nutzen. Ein Beispiel für eine solche Aktivität stellt die Qualitätssicherung dar. Ein fertiges Softwareprodukt muss alle an die Software gestellten Anforderungen erfüllen. Um

dieses Qualitätsmerkmal sicherzustellen, ist es unter anderem nötig, jede Anforderung zu ihrer Implementierung rückzuverfolgen. Die TLA soll in diesem Fall TLs zwischen Anforderungsartefakten und Quelltextartefakten ermitteln. Außerdem sollte jede Funktionalität des Systems dokumentiert sein. Die TLA sollte im Zuge der Qualitätssicherung also auch TLs zwischen Quelltextartefakten und Dokumentationsartefakten ermitteln. Eine TLA kann manuell durch die Aktivitäten eines Menschen, automatisiert durch die Aktivitäten eines Computers oder teilweise automatisiert durch deren Kombination erfolgen. Eine TLA kann parallel zur Erstellung der Artefakte oder nach der Erstellung der Artefakte erfolgen. Im letzteren Fall spricht man von einer TLR. Die ATLR zwischen Anforderungs- und Quelltextartefakten ist das zentrale Thema dieser Bacherlorarbeit. Es ist anzumerken, dass sich Anforderungs- und Quelltextartefakte sowohl in ihrer sprachlichen Abstraktion als auch in ihrer Granularität bei der Beschreibung von Prozessen stark unterscheiden. Die ATLR zwischen derart unterschiedlich abstrakten Artefakten wird auch als vertikale ATLR bezeichnet.

2.2 Informationsrückgewinnung

Eine im Rahmen des vorliegenden Anwendungsfalls treffende Definition beschreibt IR als die Wissenschaft des Rückgewinnens von Material (in der Regel Dokumente) unstrukturierter Natur aus einer Sammlung von Material. Das rückgewonnene Material soll einen Informationsbedarf befriedigen [MRS08]. Unstrukturierte Materialien sind Materialien, die keine semantisch offenkundige, für den Computer einfach zu handhabende Struktur besitzen. Sie stellen den Gegensatz zu strukturierten Materialien dar. Dies sind Materialien, die typischerweise in einer relationalen Datenbank gespeichert sind.

In früheren Zeiten beschäftigten sich nur wenig Menschen mit IR. Zu diesen zählten unter anderem Bibliothekare, Zeitungsverleger und Anwälte. Bibliothekare hatten beispielsweise die Aufgabe, in einer Sammlung von Büchern ein spezielles Buch zu finden, das den Informationsbedarf eines Kunden befriedigt. Nach dem Siegeszug des Internets führen Millionen Menschen täglich IR durch. Beispielsweise, wenn sie eine Suchmaschine nutzen oder ihre E-Mails durchsuchen. Der Erfolg des Internets hat das Bedürfnis nach effizienter IR stark erhöht und die Wissenschaft der IR populär gemacht. Denn die vielfache Nutzung des Internets hat zu einer enormen Menge an öffentlich frei verfügbaren Daten und Dokumenten geführt. Mit steigender Zahl an zugänglichen Dokumenten steigt der Bedarf nach Möglichkeiten der effizienten und zielgerichteten Suche nach in konkreten Situationen relevanten Dokumenten. Doch das Anwendungsgebiet der IR beschränkt sich nicht nur auf das Suchen von Informationen. Der Bereich IR umfasst u.a. auch das Filtern von Dokumentensammlungen oder die Weiterverarbeitung einer Reihe von rückgewonnenen Dokumenten. Ein weitere häufig eingesetzte IR-Aufgabe besteht im Gruppieren von Dokumenten auf Grundlage ihres Inhalts. Dies ist vergleichbar mit dem Ordnen von Büchern in einem Bücherregal nach ihrem Thema. Die IR-Aufgabe der Klassifizierung besteht darin, zu entscheiden, zu welcher Klasse bzw. Menge von Klassen ein Dokument gehört. Diese Klassen können sich beispielsweise auf die Eignung des Textes für verschiedene Altersgruppen beziehen. Außerdem findet IR nicht ausschließlich in Verbindung mit dem Internet statt. Vielmehr sollen Informationen auch aus beliebigen Textkorpora abgeleitet werden. Beispielsweise soll automatisiert ermittelt werden können, welche Bücher kinderfreundliche Inhalte haben. IR ist auf dem Weg, die dominierende Form des Zugriffs auf Informationen zu werden [MRS08].

IR soll ein Informationsbedürfnis befriedigen. Dieses Informationsbedürfnis wird einem IR-System in Form einer Anfrage übergeben. Im Fall der IR-Dokumentsuche sollen alle zur Anfrage relevanten Dokumente aus einer Dokumentsammlung extrahiert werden. Diese Dokumente sollen Informationen enthalten, die den durch die Anfrage repräsentierten

Informationswunsch befriedigen. Eine Anfrage an ein IR-System ist in der Regel in natürlicher Sprache formuliert. Zur Verarbeitung der Anfrage wird sie häufig in eine Anfragerepräsentation überführt. Analog dazu werden häufig auch natürlichsprachliche Dokumente in eine Dokumentrepräsentation überführt. Eine Ausnahme davon stellt beispielsweise die Suche nach Schlüsselbegriffen in Texten dar. Hier wird häufig auf eine zusätzliche Repräsentation der natürlichen Sprache verzichtet [Hen08]. Im Folgenden soll sich auf das in dieser Arbeit relevante Anwendungsgebiet der oben beschriebenen IR-Dokumentsuche fokussiert werden.

An dieser Stelle soll der Bezug zum in Kapitel 2.1 beschriebenen Konzept der TLR hergestellt werden. Bei der TLR werden TLs zwischen Artefakten ermittelt. Betrachtet man Artefakte als Dokumente, so kann man das Ziel der TLR wie folgt als Problemstellung der IR formulieren: „Für ein gegebenes Artefakt bzw. Dokument finde das relevanteste Artefakt bzw. Dokument aus der Zielmenge.“ Finde beispielsweise für ein gegebenes Anforderungsartefakt das relevanteste Quelltextartefakt. Zwischen diesen Artefakten kann dann eine TL definiert werden.

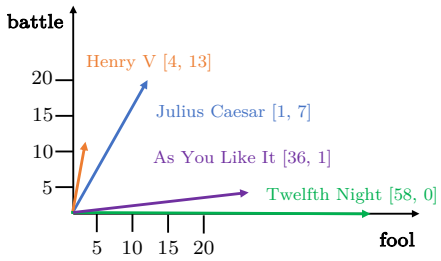
2.2.1 Vektorraummodell

Das Vektorraummodell (engl.: vector space model, VSM) ist ein konkretes Verfahren der IR. Der zentrale Grundgedanke des VSM besteht darin, die Bedeutung von Anfragen und Dokumenten durch Vektoren zu repräsentieren [JM22]. Der Vergleich der Relevanz von Dokumenten zur Anfrage kann dann auf Basis eines Ähnlichkeitsvergleichs von Vektoren erfolgen. Ein Ziel des VSM besteht darin, inhaltlich ähnliche Anfragen und Dokumente auf ähnliche Vektoren abzubilden. Damit soll erreicht werden, dass inhaltlich ähnliche Anfragen und Dokumente auch ähnlich repräsentiert werden. Anfragen und Dokumente bestehen meist aus mehreren natürlichsprachlichen Worten. Das VSM ermittelt in der Regel entweder einen einzigen Vektor zur Repräsentation aller Worte oder für jedes Wort eine einzelne Vektorrepräsentation. Man unterscheidet damit zwischen der Repräsentation von Dokumenten (Anfragen werden auch als Dokumente verstanden) durch Vektoren und von Worten durch Vektoren. Für einen geeigneten Ähnlichkeitsvergleich sollen Dokumente mit ähnlicher Bedeutung auf ähnliche Vektoren abgebildet werden. Auch sollen Worte mit ähnlicher Bedeutung auf ähnliche Vektoren abgebildet werden. Die Vektorrepräsentation basiert auf zwei Annahmen. Die erste Annahme sagt aus, dass Dokumente mit ähnlichem Inhalt meist dieselben Wörter enthalten. Die zweite Annahme sagt aus, dass Worte mit ähnlichen Bedeutungen häufig in Kombination mit denselben anderen Wörtern auftreten. Daher nutzt das VSM Koexistenzmatrizen. Diese repräsentieren, wie häufig bestimmte Wörter in Dokumenten bzw. in Kombination mit bestimmten anderen Wörtern auftreten.

2.2.1.1 Dokumente als Vektoren

Um Dokumente auf Vektoren abzubilden, nutzt das VSM eine Wort-Dokument-Matrix [JM22]. In einer Wort-Dokument-Matrix repräsentiert jede Zeile ein Wort aus dem Vokabular V der Sprache der Dokumente. Jede Spalte repräsentiert ein Dokument. Damit besitzt die Matrix Dimension $N \times M$, wobei N gleich der Anzahl Wörter des Vokabulars ($|V|$) und M gleich der Anzahl der Dokumente ist. Die Zellen der Matrix beschreiben, wie häufig ein Wort $x \in V$ des Vokabulars in einem Dokument y vorkommt. Abbildung 2.1 zeigt einen Ausschnitt aus einer exemplarischen Wort-Dokument-Matrix. In dieser Wort-Dokument-Matrix ist zu sehen, wie häufig die Worte „battle“, „good“, „fool“ und „wit“ in bestimmten Werken des Schriftstellers Shakespeare vorkommen. Beispielsweise kommt das Wort „battle“ einmal in *As You Like It* vor.

Das VSM bildet nun jedes Dokument auf seinen zugehörigen Spaltenvektor in der Wort-Dokument-Matrix ab. Dieser Vektor heißt Dokumentvektor zu Dokument y und hat Dimension N. Damit wird der Inhalt eines Dokuments im VSM effektiv darüber repräsentiert,



	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

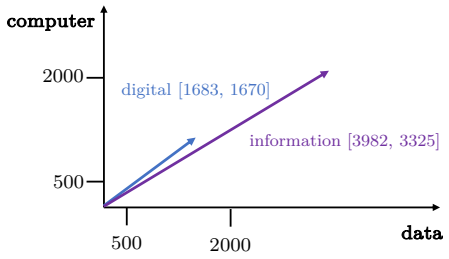
Abbildung 2.1: Wort-Dokument-Matrix und Dokumentvektoren

welche Wörter häufig und welche selten im Dokument vorkommen. Das VSM fasst ein Dokument demnach einfach als Menge von Worten (engl.: bag of words, BOW) auf. Damit ignoriert es die Anordnung oder Position der Worte, die für einen menschlichen Leser entscheidend zum Verständnis des Dokuments wären. Beispielsweise wird der Inhalt von *As You Like It* durch den Dokumentvektor (1 114 36 20) repräsentiert (s. Abbildung 2.1). Auch die Ähnlichkeit der Vektoren von inhaltsähnlichen Dokumenten zeigt sich im Beispiel von Abbildung 2.1. Der Vektor für die Komödie *As You Like It* (1 114 36 20) ähnelt stark dem Vektor der Komödie *Twelfth Night* (0 80 58 15). Er ähnelt jedoch nicht dem Vektor des kriegerischen Werkes *Henry V* (13 89 4 3). Eindeutige Metriken zur Bestimmung der Ähnlichkeit der Vektoren werden in Abschnitt 2.2.1.3 eingeführt.

2.2.1.2 Worte als Vektoren

Die Repräsentation von Worten mithilfe von 1-aus-n-Vektoren ist die einfachste Möglichkeit, Worte als Vektoren zu repräsentieren. Ein 1-aus-n Vektor hat Dimension $|V|$ und jede Vektorkomponente entspricht einem Wort im Vokabular V. Die Repräsentation eines Wortes ist dann durch den 1-aus-n Vektor gegeben, dessen Wert an genau der Komponente, die dem Wort entspricht, gleich 1 ist. Alle anderen Einträge sind gleich 0. Diese Repräsentation ist jedoch nicht geeignet für die meisten IR-Anwendungen, da sich Ähnlichkeitsmetriken nicht sinnvoll darauf anwenden lassen.

Um Worte auf Vektoren abzubilden, nutzt das VSM eine Wort-Wort-Matrix [JM22]. In einer Wort-Wort-Matrix repräsentiert jede Zeile und jede Spalte ein Wort aus dem Vokabular V der Sprache. Damit besitzt die Matrix Dimension $N \times N$. Die Zellen der Matrix beschreiben, wie häufig ein Wort x des Vokabulars im Kontext eines Wortes y in einer Dokumentsammlung auftritt. Der Kontext eines Wortes kann beispielsweise durch das Dokument an sich gegeben sein. In diesem Fall würde eine Zelle der Matrix beschreiben, wie häufig die Worte x und y im selben Dokument vorkommen. In der Regel wird der Kontext eines Wortes jedoch durch ein kleineres Wortfenster von beispielsweise vier Worten beschrieben. In diesem Fall würde eine Zelle der Matrix beschreiben, wie häufig das Wort x in einem Radius von vier Worten um y herum vorkommt. Abbildung 2.2 zeigt einen Ausschnitt aus einer exemplarischen Wort-Wort-Matrix für vier Worte aus dem Wikipedia-Datensatz. Beispielsweise kommt das Wort „digital“ 1670 Mal im Kontext von „computer“ vor. Das VSM bildet nun jedes Wort auf seinen zugehörigen Zeilenvektor in der Wort-Wort-Matrix ab. Dieser Vektor heißt Wortvektor zu Wort x und hat Dimension N. Damit wird die Bedeutung eines Wortes im VSM effektiv darüber repräsentiert, mit welchen Worten es häufig und mit welchen selten auftritt. Beispielsweise wird die Bedeutung von „information“ durch den Wortvektor (3325 3982 5 13) repräsentiert (s. Abbildung 2.2). Hier ist ebenfalls zu erkennen, dass dieser Vektor dem Wortvektor des bedeutungsähnlichen Wortes „digital“ ähnelt.



	computer	data	pie	sugar
cherry	2	8	442	25
strawberry	0	0	60	90
digital	1670	1683	5	4
information	3325	3982	5	13

Abbildung 2.2: Wort-Wort-Matrix und Wortvektoren

2.2.1.3 Ähnlichkeitsvergleich

Ein Ähnlichkeitsvergleich soll angeben, wie ähnlich sich zwei Worte bzw. Dokumente sind. Sein Ziel besteht darin, eine hohe Übereinstimmung zwischen genau denjenigen Worten und Dokumenten festzustellen, deren Bedeutung bzw. Inhalt ähnlich ist. Worte und Dokumente werden auf Vektoren abgebildet, die ihre Bedeutung bzw. ihren Inhalt in Abhängigkeit von Worten des Vokabulars repräsentieren. Es ist davon auszugehen, dass Worte und Dokumente mit hohen Werten in denselben Vektorkomponenten auch eine hohe bedeutungstechnische Übereinstimmung besitzen. So haben sowohl das Wort „digital“, als auch das Dokument „information“ hohe Werte in der Vektorkomponente, die mit „data“ assoziiert ist (s. Abbildung 2.2). Damit ist von einer hohen inhaltlichen Übereinstimmung auszugehen. Diese sollte das Ergebnis eines Ähnlichkeitsvergleichs widerspiegeln. Zum Vergleich der Ähnlichkeit zweier Vektoren, die jeweils ein Wort bzw. Dokument repräsentieren, bietet sich die Kosinusähnlichkeit an [JM22]. Diese beschreibt den Kosinus des Winkels zwischen zwei Vektoren. Die Kosinusähnlichkeit ist für zwei Vektoren v, w definiert als:

$$\cos(v, w) = \frac{v * w}{|v||w|} \tag{2.1}$$

Es wird deutlich, dass die Kosinusähnlichkeit von v, w ihrem normalisierten Skalarprodukt entspricht. Das Skalarprodukt von zwei Vektoren mit hohen Werten in derselben Dimension nimmt ebenfalls hohe Werte an. Damit betont die Kosinusähnlichkeit die obige Annahme, dass Worte und Dokumente mit hohen Werten in denselben Vektorkomponenten auch eine hohe bedeutungstechnische Übereinstimmung besitzen. Worte wie „the“, die von Natur aus häufig vorkommen, kommen auch häufig in Kombination mit vielen anderen Worten vor. Damit nimmt ihr Skalarprodukt mit anderen Worten überproportional hohe Werte an. Die Ähnlichkeit zweier Worte soll jedoch nicht von der allgemeinen Häufigkeit der Worte beeinflusst werden. Dies wird durch die Normalisierung der Vektoren sichergestellt. Die Kosinusähnlichkeit stellt damit eine Ähnlichkeitsmetrik dar, die für bedeutungstechnisch ähnliche Worte und Dokumente hohe Werte und ansonsten niedrige Werte liefert. Ein Dokument besteht aus mehren Worten und kann als BOW aufgefasst werden. Wird jedes Wort der BOW einzeln durch einen Vektor repräsentiert, so kann die BOW als Menge von Einbettungen (engl.: bag of embeddings, BOE) repräsentiert werden. Eine Einbettung meint hier eine Vektorrepräsentation eines Wortes. Zum Vergleich der Ähnlichkeit zweier BOEs bietet sich die Wortüberführungsdistanz (engl.: word mover's distance, WMD) an. Diese beschreibt die minimale Distanz zwischen den beiden Mengen. Die WMD ist für zwei BOEs BOE_1, BOE_2 definiert als:

$$\text{WMD}(BOE_1, BOE_2) = \sum_{v \in BOE_1} \min_{w \in BOE_2} \{d(v, w)\} \tag{2.2}$$

Hierbei ist $d(v, w)$ gleich der euklidischen Distanz der Vektoren v, w. Die WMD berechnet die Ähnlichkeit zweier BOWs als die minimale kumulierte euklidische Distanz zwischen zwei Vektoren, die jeweils ein Wort der BOWs repräsentieren. Diese soll die minimale semantische Distanz der Worte der BOWs repräsentieren. Je niedriger der Wert der WMD, desto ähnlicher sind zwei BOWs.

2.2.1.4 Gewichtung der Vektorkomponenten

Die zuvor beschriebene Wort-Dokument-Matrix (s. Abbildung 2.2.1.1) gibt an, wie häufig bestimmte Worte in Dokumenten vorkommen. Mithilfe dieser Häufigkeiten repräsentiert das VSM den Inhalt von Dokumenten. Hierbei gilt es jedoch zu beachten, dass nicht jedes Wort zur Bestimmung und Abgrenzung des Inhalts geeignet ist [JM22]. So kommen Worte wie „the“, „it“ und „they“ in nahezu jedem englischen Satz vor. Damit treten sie häufig in Kombination mit nahezu allen anderen Worten und in nahezu allen Dokumenten auf. Damit grenzen sie Worte und Dokumente semantisch nicht ab. Zum Beispiel ähneln sich die Dokumente *As You Like It* und *Henry V* (s. Abbildung 2.2.1.1) semantisch nicht, nur weil in ihnen häufig das Wort „the“ auftritt. Dies führt zu einer paradox anmutenden Erkenntnis. Ein Wort, das häufig in Kombination mit einem Dokument oder einem anderen Wort auftritt, sagt viel über seine Bedeutung aus. Ein Beispiel hierfür wäre das Wort „pie“, das häufig mit „cherry“ vorkommt. Andererseits sagen Worte, die zu häufig in Kombination mit einem Dokument oder anderen Wort auftreten, wenig über seine Bedeutung aus. Ein Beispiel hierfür wäre das Wort „the“, das sehr häufig mit „cherry“ vorkommt. Im Kontext von IR unterscheidet man daher zwischen Worthäufigkeit (engl.: term frequency, TF) und Dokumenthäufigkeit (engl.: document frequency, DF). TF beschreibt, wie häufig ein Wort x in einem Dokument A vorkommt. DF beschreibt, in wie vielen Dokumenten ein Wort x vorkommt. Worte mit einer hohen TF in einem Dokument A und einer niedrigen DF sagen mit hoher Wahrscheinlichkeit viel über die Bedeutung des Inhalts aus. Das VSM berechnet zu jedem Wort den Quotienten aus TF und DF. Dieser wird zur Gewichtung der Vektorkomponenten mit der reinen Häufigkeit des Wortes in der Wort-Dokument-Matrix multipliziert.

2.3 Natürliche Sprache und ihre Verarbeitung

Sprache gehört zu den zentralsten Konzepten und Werkzeugen des sozialen und beruflichen Lebens. Sie stellt die Basis der menschlichen Kommunikation dar und fungiert als Medium zur Übertragung von Wissen, Ideen, Informationen, Meinungen, Gefühlen und vielem mehr. Die Sprache, in der Menschen untereinander kommunizieren, wird als natürlich klassifiziert. Natürliche Sprache ist Sprache, die sich bei ihren menschlichen Anwendern durch Gebrauch und Wiederholung ohne bewusste Planung entwickelt hat. Natürliche Sprache besteht aus Worten. Worte tragen eine Bedeutung. Wortbedeutungen werden normalerweise als Einträge in einer Wissensdatenbank wie Wikipedia repräsentiert. Ein wichtiges Konzept der Wortbedeutungen besteht darin, dass Zusammenhänge zwischen den Bedeutungen verschiedener Worte bestehen können [JM22]. Ein solcher Zusammenhang ist beispielsweise der synonymische Zusammenhang. Zwei Worte werden als synonym bezeichnet, wenn ihre Bedeutungen nahezu identisch sind (z.B. „sofa“ und „couch“). Die Bedeutungsähnlichkeit stellt einen weiteren möglichen, sehr viel häufiger anzutreffenden Zusammenhang zwischen Wortbedeutungen dar. Die Worte „gift“ und „donation“ sind keine Synonyme, bezeichnen jedoch beide das Verschenken von Besitz, um anderen Menschen zu helfen oder eine Freude zu machen. Wortbedeutungen können einander auch entgegengesetzt sein (Antonyme) oder miteinander über ein semantisches Gebiet in Verbindung stehen. Als Beispiel lässt sich das semantische Gebiet des Krankenhauses anführen, zu dem Worte wie „surgeon“, „scalpel“ und „nurse“ gehören.

Verfahren zur Verarbeitung natürlicher Sprache (engl.: natural language processing, NLP) haben die Darstellung und Analyse von natürlicher Sprache zum Ziel. Die Darstellung natürlicher Sprache soll für ihre Analyse und für den Einsatz in einer Reihe von weiteren Anwendungsfällen geeignet sein. Zu diesen Anwendungsfällen zählen beispielsweise Textübersetzung, Textparaphrasierung oder Textrepräsentation [Lid01]. Eine weitere Aufgabenstellung der Analyse natürlicher Sprache ist die Bedeutungsauflösung (engl.: word sense disambiguation, WSD) in einem vorliegenden Kontext [Nav09]. Natürlichsprachliche Worte können mehrere Bedeutungen haben. Das Wort „mouse“ bezeichnet sowohl ein Tier aus der Gattung der Nagetiere als auch ein Gerät zur Bedienung eines Computers. Menschen sind in der Lage, sich die in einem Zusammenhang verwendete Bedeutung eines Wortes leicht aus dem Kontext zu erschließen. Für Maschinen ist die Problematik der Bestimmung der in einem Kontext verwendeten Wortbedeutung jedoch alles andere als trivial. Verfahren zur WSD basieren in der Regel auf einer Wissensdatenbank, in der für ein gegebenes Wort alle möglichen Wortbedeutungen gespeichert sind. WSD hat zum Ziel, Worten, die in einem Text vorkommen, ihre im vorliegenden Kontext korrekte Bedeutung aus der Wissensdatenbank zuzuweisen.

In den meisten Anwendungsfällen eignet sich eine Darstellung natürlicher Sprache, die die relevantesten Informationen eines Textes präzise betont. Darüber hinaus sollten die dargestellten Informationen möglichst eindeutig sein. Natürliche Sprache ist jedoch in der Regel weder präzise noch eindeutig. Worte wie „the“ oder „an“ haben wenig Informationsgehalt und machen natürliche Sprache unpräzise. Worte wie „plant“ oder „plants“ unterscheiden sich in ihrer Schreibweise, beziehen sich jedoch beide auf Pflanzen. Damit verringern sie die Eindeutigkeit natürlicher Sprache. Die Vorverarbeitung natürlicher Sprache hat zum Ziel, einen natürlichsprachlichen Text in eine Reihe von präzisen, eindeutigen und informationsreichen linguistischen Einheiten zu überführen [ID10]. Dies soll seine Erfassung und Verarbeitung durch den Computer verbessern. Zur Vorverarbeitung natürlicher Sprache existieren mehrere Techniken, die im Folgenden vorgestellt werden sollen.

2.3.1 Wortsegmentierung

Als Wortsegmentierung bezeichnet man den Prozess der Extraktion von Worten aus einem Textdokument. Die extrahierten Worte sollen möglichst relevant für die Bedeutung des Textes sein. In einem ersten Schritt wird zunächst jede Folge von Zeichen extrahiert, bei der ein Leerzeichen auf den letzten Buchstaben folgt [JM22]. Es folgen weitere miteinander kombinierbare Teilschritte.

Texte enthalten neben den Buchstaben in Worten auch weitere Sonderzeichen. Zu diesen zählen beispielsweise Satzzeichen wie Kommata, Ausrufezeichen und Fragezeichen. Aber auch Zeichen wie „&“, „-“ und „/“ gehören zur Menge der Sonderzeichen. Bei der Sonderzeichenbehandlung wird definiert, wie (bestimmte) Sonderzeichen verarbeitet werden sollen. Beispielsweise kann definiert werden, dass eine Folge von Zeichen, die ein Zeichen wie „/“ oder „.“ enthält, an dieser Stelle in zwei Worte aufgetrennt werden soll. Aus der Zeichenfolge „high/higher“ würden die Worte „high“ und „higher“ extrahiert werden. Aus der Zeichenfolge „Ph.D.“ die Worte „Ph“ und „D“. In der Regel werden alle Sonderzeichen aus dem Text herausgefiltert. Dies kann vorteilhaft sein, da Sonderzeichen in der Regel keine für den Text relevante Bedeutung tragen. Andererseits trennen beispielsweise Satzzeichen größere Sinneinheiten inhaltlich voneinander ab. Es existieren Anwendungsfälle und Verfahren, die auf die Kennzeichnung einer solchen Trennung angewiesen sind. In einem solchen Fall kann der Einsatz eines Sonderzeichenfilters nachteilig sein.

Die Binnenmajuskelschreibweise (engl. „camel case notation“) wird in der Praxis häufig bei der Benamung von Quelltexteinheiten eingesetzt. In Anwendungsfällen mit Quelltext

ist es vorteilhaft, Zeichenfolgen in Binnenversalschreibweise in ihre einzelnen Worte aufzutrennen. Auf diese Weise kann die Bedeutung der einzelnen Worte und damit der Quelltexteinheit in Folgeschritten bestimmt werden.

Stoppworte sind Worte, die in Textdokumenten häufig auftreten, jedoch wenig Bedeutung in sich tragen. Stoppworte wie „and“, „are“, „this“ werden beispielsweise zum Verbinden von Worten eines Satzes genutzt. Stoppworte leisten in der Regel keinen Beitrag zum Kontext oder Inhalt eines Satzes bzw. eines Textdokuments [LD]. Bei Techniken zur Stoppwortentfernung unterscheidet man zwischen statischer und dynamischer Stoppwortentfernung. Bei der statischen Stoppwortentfernung werden Stoppworte auf Basis einer zuvor erstellten Wortliste herausgefiltert. Außerdem können Worte, deren Zeichenlänge unter einem festgelegten Wert liegt, entfernt werden. Diese Variante der Stoppwortentfernung bezeichnet man als Wortlängenfilterung. Es werden folglich stets dieselben Worte unabhängig vom vorliegenden Text entfernt. Bei der dynamischen Stoppwortentfernung werden Stoppworte in Abhängigkeit des vorliegenden Textes entfernt. Beispielsweise werden stets die n häufigsten Worte des Textes entfernt [LD]. Aus einer Entfernung von Stoppworten können sich mehrere Vorteile ergeben. Stoppwortentfernung reduziert die Zahl der Worte in Textdokumenten um durchschnittlich 35-45%. Dadurch verringert sich auch die Trainings- und Test-Dauer eines Modells auf dem transformierten Text merklich. Außerdem werden Worte ohne Bedeutung aus dem Kontext von bedeutungsstarken Worten entfernt. Hierdurch können die für den Inhalt eines Textdokuments relevanten Worte bedeutungstechnisch genauer erfasst und abgegrenzt werden. Damit kann auch die Gesamtbedeutung des Textdokuments genauer erfasst werden. Nachteile können sich ergeben, wenn Stoppworte wie „not“ entfernt werden, die die Aussage eines Satzes entscheidend beeinflussen.

2.3.2 Wortnormalisierung

Worte sind die kleinsten selbstständigen Einheiten der natürlichen Sprache. Sie werden durch Folgen von Zeichen des zugrunde liegenden Alphabets der Sprache dargestellt. Wortnormalisierung ist der Prozess der Standardisierung von Worten. Dieser Prozess umfasst mehrere Teilschritte, die beliebig miteinander kombiniert werden können [JM22].

Im Zuge der Kleinbuchstaben-Transformation werden alle Buchstaben des Wortes in ihre Kleinschreibung überführt. Beispielsweise wird das am Satzanfang großgeschriebene Wort „Partner“ in „partner“ transformiert. Hierdurch gehen Informationen über die Schreibweise der Worte im jeweiligen Kontext verloren. Das Wort „US“ wird in seiner Bedeutung als Staat großgeschrieben. Nach der Kleinbuchstaben-Transformation kann jedoch nicht mehr zwischen dieser Bedeutung und der Bedeutung des Pronomens „us“ unterschieden werden. Auch können Worte, die am Anfang eines Satzes stehen und daher großgeschrieben werden, nicht mehr als solche erkannt werden. Doch in vielen Anwendungsfällen und insbesondere dem der IR überwiegt der Vorteil der Standardisierung. Denn häufig haben die Groß- und Kleinschreibweise eines Wortes dieselbe Bedeutung. In diesem Fall ist es sinnvoll, die beiden Schreibweisen desselben Wortes als gleich zu betrachten und sie auch gleich zu repräsentieren. Die Kleinbuchstaben-Transformation ermöglicht es dann häufiger das vereinheitlichte Wort zu erfassen und es dadurch bedeutungstechnisch genauer erfassen und abgrenzen zu können.

Die Lemmatisierung ist ein zentraler Teil des Prozesses der Wortnormalisierung [JM22]. Zu jedem Wort existiert eine Wortgrundform, auch Lemma genannt. Ein Lemma bildet den Ausgangspunkt, aus dem sich ein Wort ableiten lässt. Beispielsweise lässt sich das Wort „smaller“ vom Lemma „small“ ableiten. Bei der Lemmatisierung wird ein Wort auf sein Lemma abgebildet. Beispielsweise werden die Worte „compute“, „computing“ und „computed“ in ihr Lemma „(to) compute“ überführt. In vielen Anwendungsfällen der NLP ist es nicht sinnvoll, Worte mit demselben Lemma als einzelne Worte zu betrachten und ihre

Bedeutung unabhängig voneinander zu repräsentieren. Ein Informations-Nachfragender, der nach „compute cosine“ sucht, ist höchstwahrscheinlich auch an Texten mit „computing cosine“ interessiert. Häufig liegen Worte mit demselben Lemma in unterschiedlichen grammatikalischen Formen vor. Sie haben jedoch trotzdem dieselbe Bedeutung. Daher sollte ihre Bedeutung einheitlich repräsentiert werden. Um dies zu gewährleisten, werden sie durch die Lemmatisierung auch syntaktisch vereinheitlicht. Algorithmen zur Lemmatisierung basieren auf vollständiger morphologischer Rückverfolgung eines Wortes. Morphologie ist die Wissenschaft des Aufbaus von Worten auf Basis kleinerer bedeutungsenthaltender Einheiten, auch Morpheme genannt. Hierbei unterscheidet man zwischen Morphemen, die die zentrale Bedeutung des Wortes vorgeben, und Morphemen, die den zentralen Morphemen eine zusätzliche Bedeutung hinzufügen. Beispielsweise besteht das Wort „computers“ aus den zwei Morphemen „computer“ (zentrales Morphem) und „s“ (zusätzliches Morphem, indiziert den Plural). Bei der morphologischen Rückverfolgung wird ein Wort in seine Morpheme aufgeteilt und auf diese Weise in sein Lemma überführt. Verfahren zur morphologischen Rückverfolgung eines Wortes sind sehr komplex und technisch aufwendig. Daher wird häufig auf ein einfacheres, aber auch ungenaueres Verfahren zur Lemmatisierung zurückgegriffen: Das Stemming. Dieses einfachere Verfahren basiert auf der Annahme, dass die zentralen Morpheme, auf die reduziert werden soll, am Beginn des Wortes stehen. Die zusätzlichen, weniger bedeutungstragenden Morpheme stehen der Annahme nach am Ende. Auf Basis dessen wird beim Stemming zur Reduzierung der Komplexität nur die Wortendung eines jeden Wortes entfernt.

2.4 Maschinelles Lernen

ML ist ein Teilbereich der angewandten Informatik. Dieser erforscht und entwickelt Algorithmen, die automatisiert Lösungsverfahren für komplexe Probleme generieren. Ein Algorithmus lässt sich definieren als eine Funktion $f : X \to Y$ [Kap12]. Diese Funktion bildet Daten des Eingabe-Raums X auf Daten des Ausgabe-Raums Y ab. Beispielsweise bildet ein Sortierungsalgorithmus eine Menge von Elementen auf eine geordnete Liste dieser Elemente ab. ML-Algorithmen werden häufig zur Lösung von Problemen eingesetzt, die mit herkömmlichen Algorithmen und Programmiermethodiken nur sehr aufwendig gelöst werden können. Als Beispiel hierfür lässt sich das Erkennen von handschriftlichen Zeichen in einem Bild anführen. Für ein gegebenes Problem erhält ein ML-Algorithmus Trainingsdaten als Eingabe. Er lernt auf Basis der Eingabe einen Algorithmus $f : X \to Y$ zur Lösung des Problems. Dieser wird auch häufig als „Modell“ bezeichnet. ML-Algorithmen lernen also auf Basis der Analyse von empirischen Trainingsdaten Muster und Regeln zur Lösung eines Problems [Mit97]. Diese Muster und Regeln können dann auf neue Daten angewendet werden. ML-Verfahren lassen sich auf Basis der Beschaffenheit der Trainingsdaten in Kategorien unterteilen. Die für diese Arbeit relevantesten Kategorien sind überwachtes, unüberwachtes und teilüberwachtes ML.

Bei überwachtem ML erhalten ML-Algorithmen markierte Trainingsdaten als Eingabe [Kap12]. Markierte Trainingsdaten sind eine Menge von Tupeln $(x, y) \in X \times Y$. Hierbei stellt x eine mögliche Eingabe in den gewünschten Algorithmus dar. y stellt die korrekte Ausgabe dar, die der Algorithmus für x erzeugen soll. Das Ziel des ML-Algorithmus besteht darin, eine Abbildung $f : X \to Y$ mit $\forall x : f(x) \approx y$ zu lernen. Diese Funktion f soll anschließend einen unbekannten Wert u möglichst exakt auf die gewünschte Ausgabe abbilden. Überwachte ML-Algorithmen basieren auf vier Schritten [CHP21]. Im ersten Schritt wird bestimmt, welche Trainingsdaten benötigt werden. Außerdem wird eine Menge dieser Daten gesammelt. Im zweiten Schritt erfolgt die Entwicklung einer Vektor-Repräsentation der Trainingsdaten für ihre Verarbeitung im ML-Algorithmus. Im dritten Schritt wird ein für die vorliegende Situation passender Lernalgorithmus ausgewählt. Die Wahl hängt von Faktoren wie dem Umfang oder der Heterogenität der Trainingsdaten ab.

Naive Bayes-Klassifikatoren, transparente Markov-Modelle und Entscheidungsbäume stellen häufig verwendete Beispiele für überwachte Lernalgorithmen dar. Im vierten Schritt trainiert der im dritten Schritt gewählte Lernalgorithmus auf den im ersten Schritt gesammelten Trainingsdaten. In diesem Schritt lernt der Algorithmus die Funktion f, die eine für die Trainingsdaten optimale Abbildung darstellt. In einem optionalen fünften Schritt kann die Güte des ML-Algorithmus untersucht werden. Hierzu erhält der ML-Algorithmus Daten eines ihm unbekannten Testdatensatzes als Eingabe. Die Güte seiner Ergebnisse werden zur Evaluation der Güte des Algorithmus selbst herangezogen.

Das Sammeln markierter Trainingsdaten ist ressourcen- und zeitaufwändig. Bei unüberwachtem ML erhalten ML-Algorithmen unmarkierte Trainingsdaten als Eingabe [CHP21]. Unmarkierte Trainingsdaten sind eine Menge von möglichen Eingaben $x \in X$. Die korrekte Ausgabe, die der gewünschte Algorithmus für x liefern soll, ist jedoch nicht gegeben. Das Ziel des unüberwachten MLs besteht dann erneut im Erkennen von Mustern in den Trainingsdaten. Diese Muster sollen dann analog zum überwachten ML zum Lernen einer Funktion $f : X \to Y$ führen. Als Beispiele für unüberwachte Lernalgorithmen lassen sich Algorithmen zur Gruppenerstellung anführen.

Ein typisches Szenario bei der Anwendung von ML-Algorithmen besteht darin, dass eine kleine Menge markierter und eine große Menge unmarkierter Daten vorliegt [CHP21]. Beispielsweise haben Entwickler Zugang zu vielen Bildern, auf denen handschriftliche Zeichen abgebildet sind. Nur bei den wenigsten ist jedoch bereits beschrieben, welche Zeichen abgebildet sind. Überwachte ML-Algorithmen könnten mit dieser geringen Menge von markierten Trainingsdaten nur ungenaue Ergebnisse erzielen. Unüberwachte ML-Algorithmen würden die vorhandene Menge von markierten Trainingsdaten nicht verwenden. Teilüberwachte ML-Algorithmen erhalten sowohl markierte als auch unmarkierte Trainingsdaten als Eingabe. Es soll also mithilfe der vielen unmarkierten Daten ein besseres Modell ermittelt werden als nur auf der Grundlage der markierten Daten. Gleichzeitig soll mithilfe der vorhandenen markierten Daten ein besseres Modell ermittelt werden als nur auf der Grundlage der unmarkierten Daten. Als Beispiele für teilüberwachte Lernalgorithmen lassen sich Selbst-Training-, Co-Training- und Erwartungsmaximierungs-Algorithmen anführen.

Klassifizierung ist ein Anwendungsgebiet des überwachten ML. Ein klassifizierender ML-Algorithmus erhält eine Eingabe $x = (x_1, \ldots, x_n) \in \mathbb{K}^n$ [JM22]. Das Ziel des ML-Algorithmus besteht darin, die Eingabe x in eine von $m \in \mathbb{N}$ verschiedenen Klassen y_j einzuteilen. Hierzu lernt er eine Funktion $f : \mathbb{K}^n \to \mathbb{K}^m$ mit $f(x) = f((x_1, \ldots, x_n)) = (P(y_1), \ldots, P(y_m))$. $P(y_j)$ gibt die Wahrscheinlichkeit an, dass x zu Klasse y_j gehört. Logistische Regression ist eines der am häufigsten verwendeten ML-Klassifizierungsverfahren. Bei diesem Verfahren hängt die zu lernende Funktion f von einem Gewichtsvektor $w \in \mathbb{K}^n$ und einer Wertung $b \in \mathbb{K}$ ab. Die Vektorkomponente w_i gewichtet hierbei den Datenpunkt x_i der Eingabe. Sie repräsentiert damit seine Wichtigkeit für die Klassifizierung. Die Wertung b wird auf jeden gewichteten Datenpunkt x_i addiert. Damit ergibt sich für jede Eingabe x der gewichtete und gewertete Klassifikationsvektor $z = (w_1 \cdot x_1 + b, \ldots, w_n \cdot x_n + b)$. Dieser Vektor wird mithilfe einer Funktion σ auf den Wahrscheinlichkeitsvektor $p = (P(y_1), \ldots, P(y_m))$ abgebildet. Der Wert $P(y_j)$ gibt an, mit welcher Wahrscheinlichkeit x laut dem ML-Algorithmus zu Klasse y_j gehört. Demnach ergibt sich für die zu lernende Funktion $f_{w,b}$: $f(x) = \sigma(w_1 \cdot x_1 + b, \ldots, w_n \cdot x_n + b))$. Im Zuge des Trainings sollen die Parameter w und b auf Basis zweier Optimierungsprinzipien gelernt werden. Zum einen soll für jede Eingabe x eines Trainingsdatensatzes gelten, dass $f(x) = P(y_j) \approx 1$ für die korrekte Klasse y_j von x. Zum anderen soll $f(x) = P(y_k) \approx 0$ für die nicht korrekten Klassen y_k von x gelten. Als Beispiel für eine Problemstellung der Klassifizierung kann die Erkennung von Tieren auf einem Bild angeführt werden. In diesem Fall erhält ein klassifizierender ML-Algorithmus ein Eingabebild x. Das Ziel des ML-Algorithmus besteht darin, auszugeben, welches von m verschiedenen Tieren y_j auf dem Bild zu sehen ist. Dies entspricht

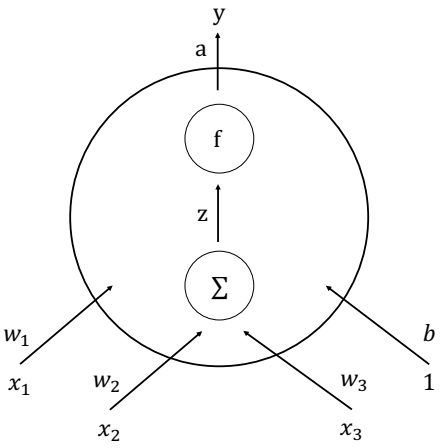


Abbildung 2.3: Eingabe, Berechnung und Ausgabe einer Recheneinheit eines neuronalen Netzes

einer Klassifizierung von x in eine der m verschiedenen Tierklassen y_j.

2.5 Neuronale Netze

Ein NN ist die Grundlage vieler ML-Algorithmen [JM22]. Die Grundlage in der Architektur eines NN bilden Recheneinheiten, die in mehreren Schichten angeordnet sind. In Anlehnung an Kapitel 2.4 kann ein NN auch als Reihe aufeinander aufbauender Klassifikationsschritte gesehen werden. NNs haben ihren Ursprung im Versuch der Nachahmung des menschlichen Gehirns. Analog zu den Neuronen des Gehirns besteht auch ein NN aus einzelnen Recheneinheiten, daher seine Bezeichnung als „neuronal“. Jede dieser Recheneinheiten erhält einen Vektor $x \in \mathbb{K}^n$, einen Gewichtsvektor $w \in \mathbb{K}^n$ und ein Gewicht $b \in \mathbb{K}$ als Eingabe. Damit ergibt sich für jede Recheneinheit eine gewichtete Summe $z = b + x \cdot w$. Neuronale Recheneinheiten wenden anschließend eine nicht-linearen Funktion f auf z an. Für diese Funktion f gilt $f(z) = a$. Abbildung 2.3 stellt die Eingabe, Berechnung und Ausgabe einer Recheneinheit eines NN dar. In der Praxis wird für f häufig die Sigmoid-, Tanh- oder Gleichgerichtet-Lineare-Funktion verwendet. Die Sigmoid-Funktion wird definiert durch $\sigma(z) := \frac{1}{1+e^{-z}}$. Sie bildet Werte auf das Intervall $[0, 1]$ ab und ist differenzierbar. Die Tanh-Funktion wird definiert durch $f(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Sie ist eine einfache Variante der Sigmoid-Funktion. Sie bildet Werte auf das Intervall $[-1, 1]$ ab. Die Gleichgerichtet-Lineare-Funktion wird definiert durch $f(z) = \max(0, z)$. Sie bildet positive Werte auf ihre Identität und negative Werte auf 0 ab.

2.5.1 Vorwärtsgerichtete neuronale Netze

Ein vorwärtsgerichtetes neuronales Netz (engl.: feed-forward neural network, FFNN) ist ein einfaches NN [JM22]. Es besteht aus mehreren hierarchisch angeordneten Schichten (s. Abbildung 2.4). Jede Schicht enthält eine Menge von Recheneinheiten. Die Recheneinheiten einer Schicht erhalten Eingaben von Recheneinheiten der nächst-niedrigeren Schicht. Sie geben ihre Ausgaben an Recheneinheiten der nächsthöheren Schicht weiter. FFNNs bestehen aus drei verschiedenen Arten von Schichten. Es gibt die Eingabeschicht, transparente Schichten und die Ausgabeschicht. Die Eingabeschicht ist die unterste Schicht eines FFNN. Die Recheneinheiten der Eingabeschicht kapseln die Werte $x_i \in \mathbb{K}$ der Eingabe $x \in \mathbb{K}^n$ des FFNN (s. Abbildung 2.4). Sie leiten diese Eingabe an die unterste der folgenden transparenten Schichten weiter. Jede der m Recheneinheiten einer transparenten Schicht erhält einen Vektor v als Eingabe. Im Fall der untersten transparenten Schicht

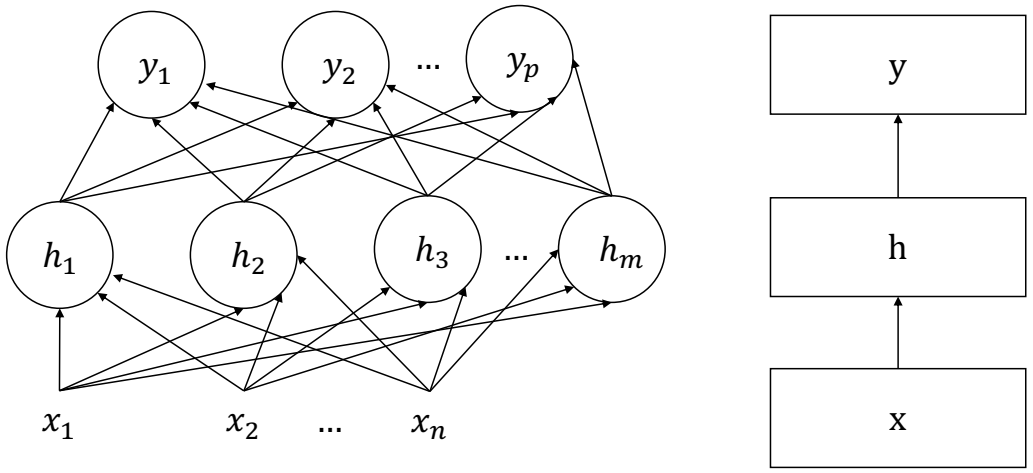


Abbildung 2.4: Aufbau eines vorwärtsgerichteten neuronalen Netzes

ist dies der Eingabevektor x. Jede Recheneinheit gewichtet und wertet die Eingabe v (s. Abbildung 2.3). Damit ergibt sich für jede Recheneinheit H_i die Ausgabe h_i. Die Ausgabe einer transparenten Schicht ist dann der Vektor $h = (h_1, \ldots, h_m) \in \mathbb{K}^m$ (s. Abbildung 2.4). Diese Ausgabe der transparenten Schicht gleicht einem Klassifizierungsvektor (s. Abschnitt 2.4). Bei mehreren transparenten Schichten gleicht die Funktionsweise eines FFNN demnach dem Ausführen mehrerer Klassifizierungsschritte. Eine transparente Schicht leitet ihre Ausgabe an ihre nachfolgende transparente Schicht weiter. Die oberste transparente Schicht leitet ihre Ausgabe an die Ausgabeschicht weiter. Analog zur transparenten Schicht erhalten auch alle Recheneinheiten der Ausgabeschicht den Ausgabevektor h der vorherigen transparenten Schicht als Eingabe. Auch diese Eingaben werden gewichtet und gewertet. Der Ausgabevektor y der Ausgabeschicht ist gleichzeitig auch die Ausgabe des FFNNs. Es ist zu sehen, dass sich die Ausgabe y eines FFNNs durch mehrere aufeinander folgende Gewichtungen der Eingabe x unter Einbeziehung von Wertungen der Recheneinheiten berechnet. Ein FFNN lernt mittels des sog. Rückwärtspropagations-Algorithmus geeignete Gewichtungen und Wertungen. Gewichtungen und Werte sind geeignet, wenn sie für möglichst viele Eingaben x zur gewünschten Ausgabe y führen.

2.5.2 Transformer

Ein Transformer ist ein NN zur Verarbeitung von Eingabesequenzen [JM22]. Transformer werden häufig in Anwendungsfällen eingesetzt, bei denen Daten in einer festen Reihenfolge vorliegen und ein Datenpunkt seine nachfolgenden Datenpunkte beeinflusst. Ein Beispiel hierfür stellt Sprachübersetzung dar. Ein Wort in einem Satz beeinflusst die Bedeutung und damit die Übersetzung aller nachfolgenden Worte des Satzes. Ein Transformer bildet eine Sequenz von Eingabepunkten $(x^{(1)}, \ldots, x^{(n)})$ auf eine Sequenz von Ausgabepunkten $(y^{(1)}, \ldots, y^{(n)})$ derselben Länge n ab. Beispielsweise bildet er eine Sequenz von deutschen Eingabeworten auf die Sequenz ihrer englischen Übersetzungen ab. Transformer erhalten alle Eingabepunkte gleichzeitig. Zur Repräsentation ihrer sequentiellen Natur wird jeder Eingabepunkt mit seiner Position in der Sequenz annotiert. Transformer bestehen analog zu FFNNs (s. Abschnitt 2.5.1) aus mehreren hierarchisch angeordneten Schichten von Recheneinheiten (s. Abbildung 2.3). Sie enthalten zusätzlich eine Selbstbeobachtungsschicht. Die Selbstbeobachtungsschicht ermöglicht es dem NN, den Einfluss von Eingabepunkten auf folgende Eingabepunkte zu bestimmen und zu nutzen. Während der Verarbeitung eines Eingabepunkts $x^{(i)}$ hat die Selbstbeobachtungsschicht Zugriff auf alle anderen Eingabepunkte $x^{(j)}$ $(j \leq i)$ zugleich. So kann sie beispielsweise einen Eingabepunkt mit der Menge der vorherigen Eingabepunkte vergleichen. Damit kann sie ermitteln, welche Ein-

gabepunkte den aktuell zu verarbeitenden Eingabepunkt beeinflussen. Auf Basis dessen wird der Ausgabepunkt $y^{(i)}$ eines Eingabepunkts $x^{(i)}$ unter Berücksichtigung der vorherigen, $x^{(i)}$ beeinflussenden Eingabepunkten berechnet. Transformer-Modelle beziehen bei der Verarbeitung eines Eingabepunkts also alle relevanten vorherigen Eingabepunkte ein. Bidirektionale Transformer-Modelle beziehen bei der Verarbeitung eines Eingabepunkts alle relevanten vorherigen und nachfolgenden Eingabepunkte ein.

2.6 Sprachmodelle

Ein LM ist ein ML-Modell (s. Abschnitt 2.4), welches natürliche Sprache repräsentiert [JM22]. Diese Repräsentation kann genutzt werden, um Sprache zu verarbeiten. Beispielsweise kann auf Basis eines natürlichsprachlichen Textes und seiner Repräsentation eine Zusammenfassung seines Inhalts generiert werden. LMs sind auch für die ATLR (s. Abschnitt 2.1) interessant. Ein Ansatz zur ATLR besteht darin, die semantische Ähnlichkeit von textuellen Artefakten zu vergleichen und zwischen ähnlichen Artefakten TLs festzulegen. Dieser Ähnlichkeitsvergleich kann auf Basis der durch ein LM erzeugten Repräsentation der Artefakte erfolgen. LMs repräsentieren Sprache in der Regel durch Vektoren. Die Granularität der repräsentierten Sprache kann variieren. Es können Worte, Sätze oder ganze Dokumente mithilfe von LMs repräsentiert werden.

LMs repräsentieren Worte analog zum VSM (s. Abschnitt 2.2.1) durch Betrachtung der Worte, mit denen sie häufig in Kombination auftreten. Die Form der Vektor-Repräsentationen unterscheiden sich jedoch deutlich. Die mithilfe des VSM ermittelten Vektoren werden in Abhängigkeit aller anderen Worte des Vokabulars V einer natürlichen Sprache ermittelt. Daher haben diese Vektoren Dimension $|V|$. Die Zahl der Worte eines Vokabulars V ist in der Regel sehr groß. Beispielsweise wird für $|V_{Englisch}|$ der Wert 50 000 angenommen [JM22]. Ein Wort x kommt in der Regel in Kombination mit nur einem Bruchteil der anderen Worte des Vokabulars vor. Daher sind viele Komponenten der im VSM erzeugten Vektoren gleich 0. Das VSM liefert folglich sehr große und nur dünn besetzte Vektorrepräsentationen für die Bedeutung von Worten. Die Weiterverarbeitung derartiger Vektoren ist sehr laufzeitintensiv.

LMs führen eine geeignetere Form der Vektorrepräsentation der Bedeutung eines Wortes ein: Die Worteinbettung (engl.: word embedding, WE). Eine WE ist ein niedrigdimensionaler (50 - 1000 Dimensionen) Vektor, dessen Komponenten nahezu alle ungleich 0 sind. LMs berechnen hierzu nicht wie das VSM, wie häufig ein Wort x in Kombination mit einem anderen Wort y vorkommt. LMs wenden stattdessen ML an, um eine Repräsentation natürlicher Sprache auf Basis von Trainingstexten zu lernen. Hierfür nutzen sie auf NNs (s. Abschnitt 2.5) basierende ML-Klassifikatoren. Dieser klassifiziert für jedes $x \in V$, ob Wort x wahrscheinlich in Kombination mit einem anderen Wort $y \in V$ auftritt. Die gewünschte Antwort wird dadurch bestimmt, ob x in Trainingstexten häufig nahe bei y steht. Diese korrekte Antwort lässt sich direkt aus dem Text ablesen. Daher wird diese Methode auch als selbstüberwachtes ML bezeichnet. LMs nutzen in der Regel sogenannte Kontext-Fenster, welche m Worte umfassen und sich über Trainingstexte bewegen. Die in diesem Fenster enthaltenen Worte bilden die Grundlage für die Eingabe des ML-Klassifizierers. Die Vektorrepräsentation von x ist dann häufig durch die Gewichtungen der Ausgabeschicht des NN gegeben [MCCD13]. Eine Vorgehensweise, die im Zusammenhang von LMs häufig eingesetzt wird, ist das Vortraining. Hierbei werden LMs auf großen Textkorpora wie dem Wikipedia-Datensatz [MGB+18] trainiert. Während dieses Trainings lernt das LM neben der Klassifizierung auch bereits WEs der in den Trainingstexten vorkommenden Worte. Dies bezeichnet man als Vortraining. Statt das LM für jeden Anwendungsfall einzeln zu trainieren, können Entwickler auf diese vortrainierten WEs zurückgreifen. Dies spart die Zeit des eigenen Trainings ein. Außerdem liegen in vielen Anwendungsfällen nicht

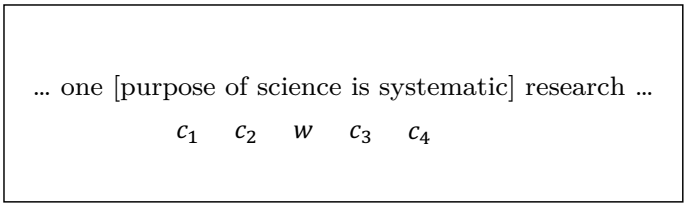


Abbildung 2.5: Wort und Kontext-Fenster

genug Trainingsdaten für ein eigenes effektives Training vor. Besonders in diesem Fall lohnt sich das Zurückgreifen auf vortrainierte WEs. Man unterscheidet zwischen statischen und kontextsensitiven LMs.

2.6.1 Statische Sprachmodelle

Ein statisches LM bildet jedes Wort einer natürlichen Sprache auf einen einzigen statischen Vektor ab. Zwei populäre statische LMs sind *Skip-Gram* und *CBOW*. Sie finden u.a. Anwendung im *Word2Vec*-Verfahren [MCCD13] zur Vektorrepräsentation von Worten.

Skip-Gram trainiert einen überwachten ML-Klassifizierer auf Basis eines FFNN (s. Abschnitt 2.5). Dieser erhält als Eingabe ein Wort w und soll die Menge $c = \{c_1, \ldots, c_n\}$ von Worten im Kontext-Fenster von w vorhersagen. Der Klassifizierer soll demnach die Wahrscheinlichkeit $P(+|w, c_i)$ ermitteln, dass ein Wort $c_i \in c$ tatsächliches ein Kontext-Wort von w darstellt. Ein Beispiel für ein Wort und sein Kontext-Fenster ist in Abbildung 2.5 veranschaulicht. Skip-Gram erhält im Beispiel die Eingabe $w = science$. Die gewünschte korrekte Ausgabe ist $c = \{purpose,\ of,\ is,\ systematic\}$. Für das Wort *systematic* und bedeutungsähnliche Worte soll eine Wahrscheinlichkeit $P(+|science, systematic) \approx 1$ ermittelt werden. Für ein bedeutungstechnisch entferntes Wort wie *eagle* soll eine Wahrscheinlichkeit $P(+|science, eagle) \approx 0$ ermittelt werden. Skip-Gram berechnet diese Wahrscheinlichkeit $P(+|w, c_i)$ auf Basis der Ähnlichkeit der WEs von w und c_i. Als Ähnlichkeitsmetrik wird das Vektorprodukt verwendet. Um eine Wahrscheinlichkeit zu erhalten, wird die Sigmoid-Funktion auf das Ergebnis des Vektorprodukts angewendet (s. Abschnitt 2.5). Skip-Gram weißt zu Beginn jedem Wort w eine zufällig generierte WE zu. Die im Trainingsprozess gelernten Gewichte der Ausgabeschicht ergeben die WEs der Worte.

CBOW trainiert analog zu Skip-Gram einen überwachten ML-Klassifizierer auf Basis eines FFNN. Dieser erhält als Eingabe eine Menge c der Worte im Kontext-Fenster eines Wortes w. Der Klassifizierer soll w vorhersagen. Hierzu ermittelt er die Wahrscheinlichkeit $P(+|c, w)$, dass ein Wort w im Kontext-Fenster c vorkommt. Während Skip-Gram ein Wort erhält und seine Kontext-Worte vorhersagt, erhält CBOW also Kontext-Worte und sagt das in diesen Kontext passende Wort vorher. Im Beispiel von Abbildung 2.5 erhält CBOW die Eingabe $c = \{purpose,\ of,\ is,\ systematic\}$. Für das Wort *science* und bedeutungsähnliche Worte soll eine Wahrscheinlichkeit $P(+|c, science) \approx 1$ ermittelt werden. Für ein bedeutungstechnisch entferntes Wort wie $flower$ soll er eine Wahrscheinlichkeit $P(+|c, flower) \approx 0$ liefern. CBOW berechnet die Wahrscheinlichkeit $P(+|c, w)$ analog zu Skip-Gram auf Basis der Ähnlichkeit der WEs von w und c_i. Die im Trainingsprozess gelernten Gewichte ergeben analog zu Skip-Gram die WEs der Worte.

CBOW und Skip-Gram weisen Worten Vektoren zu. Je häufiger Worte in Trainingstexten auftauchen, desto genauer werden sie durch WE repräsentiert und bedeutungstechnisch abgegrenzt. Worte, die nicht in Trainingstexten auftreten, erhalten keine WEs. Das statische LM „fastText“ [**?**] erweitert das Skip-Gram-LM. fastText repräsentiert zunächst jedes Wort als Menge von n-Grammen. Ein n-Gram ist eine Folge von n Token, im Fall von fastText eine Folge von n Buchstaben. Für das Wort „where“ und $n = 3$ ist die Menge

von n-Grammen gegeben durch: $\{whe, her, ere\}$. fastText ermittelt nun für jedes n-Gram eines Wortes eine WE. Die WE des Wortes selbst ergibt sich aus der Summe der WEs seiner n-Gramme. fastText ist so in der Lage, genauere WEs als Skip-Gram für Worte zu ermitteln, die nicht in den Trainingstexten vorkommen. Denn für unbekannte Worte besteht die Möglichkeit, dass WEs für die n-Gramme des Wortes gelernt wurden. Somit können auch unbekannte Worte WEs erhalten.

2.6.2 Kontextsensitive Sprachmodelle

Statische LMs ermitteln eine einzige statische, globale WE für jedes Wort. Sie repräsentieren damit die Bedeutung eines Wortes durch einen einzigen Vektor. Worte haben jedoch in verschiedenen Kontexten unterschiedliche Bedeutungen. Das Wort „mouse“ bezeichnet im Kontext des Tierreichs ein Nagetier und im Kontext der Informatik ein technisches Hilfsmittel zur Interaktion mit dem Computer. Statische WEs repräsentieren diese verschiedenen Bedeutungen nicht. Dies führt zu Ungenauigkeiten bei statischen LMs. Kontextsensitive LMs basieren auf dynamischen WEs. Dynamische WEs liefern eine Repräsentation der Bedeutung eines Wortes in Abhängigkeit seines gegenwärtigen Kontexts. Damit repräsentieren sie die Bedeutung eines Wortes durch verschiedene Vektoren, abhängig vom in einer Situation vorliegenden Gesamtkontext.

Ein klassisches Beispiel für ein kontextsensitives LM ist *BERT* [DCLT19]. BERT steht für „Bidirectional Encoder Representations from Transformers“. Es basiert auf einem bidirektionalen Transformer-Modell (s. Abschnitt 2.5.2). Beim Training und bei der Verwendung erhält BERT Wortsequenzen als Eingabe. Es basiert auf zwei Trainingsdirektiven: maskierte Sprachmodellierung (engl.: masked language modeling, MLM) und Satzvorhersage. Beim MLM maskiert BERT zunächst etwa 15% der vorkommenden Worte. Anschließend lernt es die Vorhersage der maskierten Worte auf Basis der Kontextworte. Aufgrund der Bidirektionalität des Transformer-Modells kann BERT die Vorhersage des maskierten Wortes auf Basis aller Kontextworte links und rechts vom Wort ermitteln. Zu Beginn des Lernprozesses werden die Eingabeworte auf 1-aus-n Vektoren abgebildet (s. Abschnitt 2.2.1). Es folgen mehrere aufeinanderfolgende Wahrscheinlichkeitsberechnungen durch Selbstbeobachtungsschichten. Hier werden die für den Kontext von benachbarten Worten relevanten Worte mit hoher Gewichtung einbezogen. Die WEs der Worte resultieren aus den Gewichten der letzten Selbstbeobachtungsschicht. Bei der Satzvorhersage erhält BERT zwei Sätze als Eingabe und soll ermitteln, ob die Sätze aufeinander folgen. Auf diese Art und Weise soll BERT semantische Zusammenhänge zwischen Sätzen lernen. BERT liefert für eine gegebene Wortfolge kontextsensitive WEs der Worte. Außerdem gibt BERT eine Einbettung der Bedeutung des Satzes zurück. BERT wurde auf dem *BooksCorpus*-Datensatz [ZKZ+15] und dem englischen Wikipedia-Datensatz vortrainiert. Für den Einsatz von BERT in spezifischen Aufgabenbereichen der NLP lässt sich BERT mithilfe weiterer Trainingsdaten auf die Aufgabensituation anpassen. Hierbei muss das Vortraining nicht erneut durchgeführt werden. Die Gewichte des NN von BERT werden lediglich an die Aufgabensituation angepasst. Dies bezeichnet man auch als Feinkalibrierung (engl. fine-tuning).

2.6.3 Bedeutungseinbettende Sprachmodelle

Bedeutungseinbettende LMs erfassen analog zu kontextsensitiven LMs die Bedeutung von Worten in Abhängigkeit des vorliegenden Kontexts [SPN20]. Sie basieren auf einer Wissensdatenbank. Diese Wissensdatenbank enthält für jedes Wort eine Menge seiner verschiedenen Bedeutungen. Zu jeder dieser Wortbedeutungen ist zudem eine Beschreibung gespeichert. Ein Beispiel für eine solche Wissensdatenbank ist Wikipedia. Wikipedia speichert beispielsweise, dass das Wort „View“ unterschiedliche Bedeutungen im Kontext von

„SQL“ bzw. „MVC“ hat. Zu jeder dieser Wortbedeutungen existiert ein beschreibender Artikel. Bedeutungseinbettende LMs lernen auf Basis einer Wissensdatenbank Vektorrepräsentationen für jede gespeicherte Wortbedeutung der enthaltenen Worte. Damit existiert zu jedem Wort w eine Menge an Repräsentationen seiner verschiedenen, in der Wissensdatenbank enthaltenen Wortbedeutungen. Zur Repräsentation eines Wortes wird einem bedeutungseinbettenden LM die im Kontext vorliegende Wortbedeutung übergeben. Diese muss im Vorfeld der Eingabe für ein Wort ermittelt werden. Die NLP-Aufgabe der WSD (s. Abschnitt 2.3) beschäftigt sich mit der Bestimmung der Wortbedeutung eines Wortes für einen gegebenen Kontext. Die Rückgabe des bedeutungseinbettenden LMs ist dann die Repräsentation der Wortbedeutung. Diese wird auch als Bedeutungseinbettung (engl.: sense embedding, SE) bezeichnet. Da die Bestimmung der Wortbedeutung in Abhängigkeit des Kontexts erfolgt ist, wird auch das Wort an sich in Abhängigkeit des Kontexts repräsentiert. Damit sind die von bedeutungseinbettenden LMs erzeugten Repräsentationen kontextsensitiv. Beispiele für derartige LMs sind *SensEmBERT* [SPN20] und *Wikipedia2Vec* [YAS+20].

3 Verwandte Arbeiten

In diesem Kapitel werden wissenschaftliche Arbeiten behandelt, deren Themen zum Gegenstand dieser Arbeit verwandt sind. Diese Arbeit beschäftigt sich mit dem Einsatz von LMs in Verfahren zur ATLR. Es lassen sich drei Kategorien von verwandten Arbeiten bilden, die in diesem Kapitel betrachtet werden sollen. Zunächst soll auf Arbeiten eingegangen werden, die dasselbe Problem wie diese Arbeit adressieren, aber andere Techniken zur Problemlösung einsetzen. In diesem Fall sind dies Arbeiten, die Verfahren zur ATLR thematisieren. Im nächsten Schritt sollen Arbeiten vorgestellt werden, die dieselbe Technik einsetzen, aber andere Probleme damit adressieren. In diesem Fall sind dies Arbeiten, die LMs und ihre Einsatzgebiete zum Thema haben. Schlussendlich werden Arbeiten beschrieben, die dieselbe Technik einsetzen, um dasselbe Problem wie in dieser Arbeit zu adressieren. In diesem Fall sind dies Arbeiten, die den Einsatz von LMs in Verfahren zur ATLR thematisieren.

3.1 Verfahren zur automatisierten nachträglichen Rückverfolgbarkeitsanalyse

Diese Arbeit beschäftigt sich mit der Rückverfolgbarkeitsanalyse zwischen Anforderungs- und Quelltext-Artefakten. Eine manuelle TLR (s. Abschnitt 2.1) ist mit einem hohen zeitlichen Aufwand verbunden. Eine ATLR soll diesen zeitlichen Aufwand eliminieren. ATLR-Ansätze vergleichen die Ähnlichkeit textueller Artefakte. Je ähnlicher zwei Artefakte, desto wahrscheinlicher liegt eine TL zwischen ihnen vor. Verfahren zur ATLR basieren u.a. auf Techniken der IR (s. Abschnitt 2.2) und der Ontologie.

3.1.1 Verfahren auf Basis von Informationsrückgewinnung

Eine grundlegende IR-Technik ist das VSM (s. Abschnitt 2.2.1). Dieses repräsentiert Worte und textuelle Dokumente als Vektoren. Ein Einsatz zur ATLR liegt damit nahe. Antoniol et al. präsentieren in ihrer Arbeit „Recovering traceability links between code and documentation“ [ACC+02] ein Verfahren zur ATLR auf Basis des VSM. Wie der Titel bereits aussagt, nutzen sie das VSM zur ATLR zwischen Dokumentations- und Quelltextartefakten. Ihr Verfahren repräsentiert zunächst mithilfe des VSM-Verfahrens alle Dokumentations- und Quelltextartefakte als Vektoren. Anschließend wird jeder Quelltextartefakt-Vektor als Anfrage interpretiert. Zu jeder dieser Anfragen werden die zur Anfrage ähnlichsten Dokumentationsartefakte ermittelt. Zur Berechnung der Ähnlichkeit zwischen Anfrage

und Dokumentationsartefakt-Vektor wird die Metrik der Kosinusähnlichkeit (s. Abschnitt 2.2.1.3) verwendet. Bei hoher Ähnlichkeit wird eine TL ermittelt. Der Repräsentation der Artefakte durch Vektoren geht eine Vorverarbeitungsphase (s. Abschnitt 2.3) voraus. Antoniol et al. setzen in ihrem Verfahren die Vorverarbeitungsschritte Kleinbuchstaben-Transformation, Stoppwortentfernung und Stemming ein. Bei den Quelltextartefakten werden nur natürlichsprachliche Identifikatoren einer Klasse wie Methoden-, Variablen- und Klassennamen extrahiert. Antoniol et al. nehmen an, dass Softwareentwickler bedeutungsvolle und aussagekräftige Bezeichnungen für diese Identifikatoren verwenden. Nach der Vorverarbeitung wird jedes Artefakt durch einen Vektor mit Dimension $|V|$ repräsentiert. Dieser besitzt für genau jedes vorkommende Wort $w_i \in V$ einen Wert $x \neq 0$ in der Vektorkomponenten i (s. Abschnitt 2.2.1.1). Bei der Berechnung von x nutzt das Verfahren auch die TF-IDF-Formel (s. Abschnitt 2.2.1.4). Das Verfahren erstellt anschließend für jedes Quelltextartefakt eine Anfrage in Form des Quelltextartefakt-Vektors. Unter Einsatz von IR wird daraufhin für jedes Quelltextartefakt das ähnlichste Dokumentationsartefakt ermittelt. Zwischen diesen Artefakten wird eine TL ermittelt. Antoniol et a. evaluieren ihr Verfahren auf dem *LEDA*-Datensatz [NM90]. Bei LEDA handelt es sich um ein in C++ entwickeltes Softwaresystem, bestehend aus 208 Quelltextklassen und 88 Dokumentationsseiten. Das Verfahren von Antonio et al. erkennt 80% der TLs (Ausbeute) zwischen Dokumentationsseiten und Quelltextklassen mit einer Präzision von 12,7%.

Eine Präzision von 12,7% ist für eine Anwendung in der Praxis nicht ausreichend. Daher ist es sinnvoll, andere IR-Techniken auf ihre Eignung zum Einsatz in ATLR zu untersuchen. Marcus und Maletic setzen die IR-Technik latente semantische Analyse (engl.: latent semantic analysis, LSA) zur ATLR ein. LSA hat zum Ziel, die Semantik von Worten und Dokumenten in ihre Vektorrepräsentation einzubeziehen. Die Grundidee besteht darin, die Informationen über den Wortkontext zur Bestimmung der Semantik eines Wortes zu nutzen. Anhand der Worte im Wortkontext wird das Thema eines Wortes abgeleitet. Worte werden in Abhängigkeit von Themen repräsentiert. Ihre Ergebnisse stellen Marcus und Maletic in ihrer Arbeit „Recovering documentation-to-source-code traceability links using latent semantic indexing“ [MM03] vor. Analog zu Antoniol et al. setzen sie ihr Verfahren zur ATLR zwischen Dokumentations- und Quelltextartefakten ein. Das Verfahren von Marcus und Maletic bereitet Dokumentations- und Quelltextartefakte in einer Vorverarbeitungsphase auf ihre spätere Weiterverarbeitung vor. Sie setzen hierfür die Vorverarbeitungsschritte Sonderzeichenbehandlung und Binnenversal-Auftrennung ein. Eine Lemmatisierung wird nicht durchgeführt. Mithilfe einer LSA werden die vorverarbeiteten Anforderungen und Quelltextklassen durch Vektoren repräsentiert. Anschließend wird für jede Vektorrepräsentation einer Quelltextklasse die Ähnlichkeit zu den Vektorrepräsentationen aller Dokumentationsartefakte ermittelt. Hierzu wird die Ähnlichkeitsmetrik der Kosinusähnlichkeit eingesetzt. Zwischen Quelltextklassen und Dokumentationsartefakten mit einer Ähnlichkeit von über 0,6 wird eine TL ermittelt. Marcus und Maletic evaluieren ihr Verfahren analog zu Antoniol et al. auf dem LEDA-Datensatz. Ihr Verfahren erkennt 71% der TLs (Ausbeute) zwischen Dokumentationsseiten und Quelltextklassen mit einer Präzision von 43%. Damit ist ihr Verfahren in der Metrik der Präzision bei vergleichbarer Ausbeute um etwa 30 Prozentpunkte besser als das Verfahren von Antoniol et al.

3.1.2 Verfahren auf Basis von Ontologien

Neben dem Anwenden von IR-Techniken existierten noch weitere vielversprechende Ansätze zur ATLR. Eine zweite Herangehensweise zum Ähnlichkeitsvergleich textueller Artefakte basiert auf einer ontologischen Analyse. Diese verwenden Li und Cleland-Huang in ihrer Arbeit „Ontology-based trace retrieva“ [LCH13]. Eine Ontologie ist eine Darstellung einer Menge von Konzepten und der zwischen ihnen bestehenden semantischen Beziehungen. Eine Ontologie repräsentiert Wissen mithilfe dieser expliziten Beziehungen. Eine

Ontologie wird häufig als Graph visualisiert, in dem jeder Knoten für ein Konzept und jede Kante für eine Beziehung zwischen zwei Konzepten steht. Der Ansatz von Li und Cleland-Huang vergleicht die Ähnlichkeit zweier Artefakte mithilfe von allgemeinen und domänenspezifischen Ontologien. Hierbei wird die domänenspezifische Ontologie zu einem Vergleich auf Satzebene und die allgemeine Ontologie zu einem Vergleich auf Wortebene genutzt. Zur Vorverarbeitung der Artefakte kommt zunächst das Standford Analyse-Werkzeug [SBMN13] zum Einsatz. Dieses teilt Artefakte in ihre einzelnen Sätze und Worte auf. Es folgen die Vorverarbeitungsschritte Stoppwortentfernung und Wortnormalisierung. Im ersten Schritt der Weiterverarbeitung verwenden Li und Cleland-Huang die TF-IDF-Formel, um eine Gewichtung der Sätze eines Artefakts nach Relevanz für seine Semantik vorzunehmen. Daraufhin wird für jeden Satz eines Artefakts das darin beschriebene Konzept in der domänenspezifischen Ontologie ermittelt. Sollte kein entsprechendes Konzept vorliegen, werden die Konzepte für die Nomen und Verben des Satzes in der allgemeinen Ontologie ermittelt. Beim Ähnlichkeitsvergleich wird die Nähe der Konzepte der Sätze bzw. Worte zweier Artefakte in den Ontologie-Graphen ermittelt. Bei ausreichend hoher Artefaktähnlichkeit wird eine TL ermittelt. Li und Cleland-Huang evaluieren ihr Verfahren auf dem *Unmanned Vehicle Highway System*-Datensatz [BTT02]. Bei diesem Testdatensatz handelt es sich um ein Softwaresystem zur Unterstützung von autonomem Fahren. Li und Cleland-Huang setzen ihr Verfahren zur ATLR zwischen Anforderungs- und Entwurfsartefakten ein. Im Testdatensatz existieren 226 Anforderungen und 1591 Entwurfsartefakte. Li verwendete drei Stunden zur Konstruktion einer allgemeinen Ontologie und zwei Tage zur Konstruktion einer domänenspezifischen Ontologie. Das Verfahren von Li und Cleland-Huang erkennt 100% der TLs (Ausbeute) zwischen Anforderungen und Entwurfsartefakten mit einer Präzision von 36,86%. Ein ATLR-Verfahren auf Basis von VSM erkennt auf demselben Testdatensatz 100% der TLs (Ausbeute) mit einer Präzision von 11,6%.

3.2 Sprachmodelle und ihre Einsatzgebiete

Der erste Abschnitt dieses Kapitels hat gezeigt, wie Verfahren zur ATLR ablaufen können. Es ist deutlich geworden, dass ein Ansatz darin besteht, Artefakte zu repräsentieren und Ähnlichkeitsmetriken auf die Repräsentation anzuwenden. LMs sind in der Lage Sprache und somit auch textuelle Artefakte semantisch zu repräsentieren. Es ist nun interessant, eine Auswahl an konkreten LMs zu betrachten. In diesem Zuge wird deutlich, ob LMs bereits erfolgreich eingesetzt werden. Im Folgenden sollen verschiedene LMs und ihre Einsatzgebiete vorgestellt werden.

3.2.1 CodeBERT

Wie bereits in Kapitel 2.6.2 beschrieben, lässt sich das LM BERT auf bestimmte Anwendungsgebiete feinanpassen. *CodeBERT* ist ein auf BERT basierendes LM, welches für Anwendungsfälle mit natürlicher Sprache in Verbindung mit Quelltext feinangepasst wurde. Feng et al. präsentieren es in ihrer Arbeit „CodeBERT: A Pre-Trained Model for Programming and Natural Language“ [FGT+20]. CodeBERT ermittelt kontextsensitive Repräsentationen, die sowohl für Elemente der natürlichen Sprache als auch Elemente einer Programmiersprache geeignet sind. Damit ist CodeBERT in der Lage die semantisch, logische Verbindung zwischen natürlicher Sprache und Programmiersprache zu erfassen. Ein mögliches Einsatzgebiet besteht in der automatisierten Generierung von Dokumentation für Quelltext. CodeBERT wurde auf dem *CodeSearchNet*-Datensatz [HWG+19] trainiert. Dieser enthält zwei Millionen bimodale und sechseinhalb Millionen unimodale Artefakte. Bimodale Artefakte enthalten sowohl Quelltext als auch natürliche Sprache. Unimodale Artefakte enthalten entweder Quelltext oder natürliche Sprache. Die Trainingsdirektive

auf bimodalen Daten besteht analog zu BERT in MLM. Hierbei erhält CodeBERT Tupel, bestehend aus einer natürlichsprachlichen Wortfolge und einer Folge von Quelltext-Tokens als Eingabe. CodeBERT maskiert daraufhin zufällig Worte bzw. Tokens und sagt sie anschließend mithilfe der gesamten Eingabe vorher. Die Trainingsdirektive auf unimodalen Daten besteht darin zu entscheiden, ob ein Wort bzw. Token tatsächlich an dieser Stelle im Text vorkommt. Beim Erstellen von WEs erwartet CodeBERT die Konkatenation eines natürlichsprachlichen Segments und eines Quelltext-Segments als Eingabe. Diese Segmente werden durch einen Trenn-Token separiert. CodeBERT ermittelt dann als Ausgabe für jedes Wort der natürlichen Sprache und jeden Token des Quelltexts eine kontextsensitive Vektorrepräsentation. CodeBERT kann auf bestimmte natürliche Sprachen und Programmiersprachen feinangepasst werden. Feng et al. evaluierten die Performanz von CodeBERT mithilfe der NLP-Aufgabe „Quelltextsuche“ auf Basis eines CodeSearchNet-Testdatensatz. Hierbei erhält CodeBERT eine Folge von natürlichsprachlichen Worten als Eingabe. Seine Aufgabe besteht darin, den für die Eingabe relevantesten Quelltextabschnitt aus einer Menge von Quelltextabschnitten zu finden. Als Evaluationsmetrik nutzen sie die Metrik durschnittlicher reziproker Wert (engl.: mean reciprocal rank, MRR). Diese Metrik gibt an, wie viele der von CodeBERT ermittelten Quelltextabschnitte auch tatsächlich relevant sind. Für einen Quelltextdatensatz in der Programmiersprache Java erzielen Feng et al. einen MRR von 0,75. Für Python ergibt sich sogar einen MRR von 0,87. Im Durchschnitt liefert CodeBERT Ergebnisse mit einem MRR von 0,69.

3.2.2 UniXcoder

UniXcoder ist ein LM, welches speziell für Anwendungsfälle mit natürlicher Sprache und Quelltext geeignet ist. Es wurde in der Arbeit „UniXcoder: Unified Cross-Modal Pretraining for Code Representation“ [GLD+22] von Guo et al. vorgestellt. UniXcoder basiert analog zu BERT und CodeBERT auf einem Transformer-Modell. Es ist in der Lage, kontextsensitive Repräsentationen für Quelltextabschnitte zu ermitteln. Als Kontext bezieht es neben Quelltext-Tokens auch natürlichsprachliche Kommentare und den abstrakten Syntaxbaum (engl.: abstract syntax tree, AST) eines Quelltextabschnitts ein. UniXcoder erwartet stets eine Folge von Tokens als Eingabe, bestehend aus Quelltext-Tokens und Kommentar-Worten des Quelltextabschnitts. UniXcoder wurde analog zu CodeBERT auf dem CodeSearchNet-Datensatz [HWG+19] trainiert. UniXcoder wird mithilfe von fünf Trainingsdirektiven trainiert. Dies ist zum einen MLM, analog zu BERT (s. Abschnitt 2.6.2). Darüber hinaus soll UniXcoder für eine gegebene Tokenfolge den nächsten Token vorhersagen. Drittens maskiert UniXcoder während des Trainings größere Tokenabschnitte und generiert sie neu. Außerdem führt UniXcoder während des Trainings eine AST-Kanten-Vorhersage durch. Hierfür erstellt es zunächst auf Basis der Tokenfolge den AST zum Quelltextabschnitt. Anschließend maskiert UniXcoder Kanten im AST und sagt diese vorher. Zuletzt wird UniXcoder mithilfe von multimodalem kontrastiven Lernen trainiert. Hierbei gruppiert es Ausschnitte der natürlichsprachlichen Kommentare mit Codeausschnitten der Tokenfolge. Daraufhin bestimmt es, wie ähnlich sich die Paare sind. Zur Evaluation nutzen Guo et al. analog zu CodeBERT die NLP-Aufgabe Quelltextsuche auf einem CodeSearchNet-Testdatensatz. UniXcoder erzielte Ergebnisse mit einem MRR von durchschnittlich 0,74. Damit liefert UniXcoder durchschnittlich um 5 Prozentpunkte bessere Ergebnisse als vergleichbare LMs wie CodeBERT.

3.2.3 XLNet

Die beiden zuvor vorgestellten LMs basieren ebenso wie BERT beide auf Transformer-Modellen (s. Abschnitt 2.5.2). Das LM *XLNet* verfolgt einen anderen Ansatz zur Ermittlung kontextsensitiver WEs. Präsentiert wurde XLNet von Yang et al. in der Arbeit „XLNet: Generalized Autoregressive Pretraining for Language Understanding“ [YDY+19].

Für eine gegebene Wortfolge $w_1 \dots w_n$ ermittelt XLNet die Repräsentation von Wort w_i unter Einbeziehung aller Kontextworte. Hierzu ermittelt XLNet zunächst alle möglichen Permutationen der Wortfolge. Für eine Wortfolge der Länge n ergeben sich dann $n!$ Permutationen. XLNet weist als Trainingsaufgabe für jede Permutation einem Wort w_i eine Wahrscheinlichkeit zu, auf seine vorhergehenden Worte zu folgen. XLNet sagt w_i damit stets auf Basis der vorherigen Worte als Kontext voraus. Dies bezeichnet man als autoregressiv. Für jede Kombination aus Worten der Wortfolge existiert eine Permutation, in der diese Kombination vor w_i vorkommt. Damit wird die Einbettung von w_i auf Basis des vollständigen Kontexts gelernt. Die Funktionalität von XLNet lässt sich an folgendem Beispiel veranschaulichen. Gegeben sei eine Wortfolge $w_1w_2w_3w_4$. XLNet soll nun w_3 repräsentieren. Die Wortfolge $w_1w_2w_3w_4$ hat die Länge vier und damit ergeben sich $4! = 24$ Permutationen. Hierbei steht w_3 jeweils sechs mal an Position eins, zwei, drei und vier. Die in den Permutationen vor w_3 positionierten Worte bilden seinen Kontext, der bei der Vorhersage von w_3 mit einbezogen wird. Damit wird w_3 auf Basis von allen möglichen Kontexten der Wortfolge vorhergesagt. Ein zu lernender Parameter θ wird zwischen allen Permutationen geteilt. Durch die Permutation kann XLNet auf das Maskieren der Trainingsdaten wie bei BERT (s. Abschnitt 2.6.2) verzichten. Somit entstehen keine Diskrepanzen zwischen Vortraining und Feinkalibrierung (s. Abschnitt 2.6). Zudem eliminiert XLNet die nicht stets korrekte Annahme von BERT, die maskierten Worte seien untereinander unabhängig. XLNet wird analog zu BERT auf dem BooksCorpus-Datensatz [ZKZ+15] sowie Teilen des englischsprachigen Wikipedias vortrainiert. XLNet erzielt bessere Ergebnisse im Vergleich zu BERT in zwanzig NLP-Aufgaben. Zu diesen zählen „Erstellen einer Dokument-Rangfolge“ und „Leseverstehen“.

3.2.4 Wikipedia2Vec

Bedeutungseinbettende LMs (s. Abschnitt 2.6.3) stellen einen vielversprechenden Ansatz in vielen Bereichen der NLP dar. Sie basieren auf Wissensdatenbanken. Eine der bekanntesten und größten Wissensdatenbanken ist Wikipedia. Nicht verwunderlich ist folglich die Existenz eines LMs, welches SEs auf Basis des Wikipedia-Datensatzes lernt. Dieses LM heißt Wikipedia2Vec und wurde von Yamada et al. in „Wikipedia2Vec: An Efficient Toolkit for Learning and Visualizing the Embeddings of Words and Entities from Wikipedia“ [YAS+20] vorgestellt. Wikipedia speichert für jedes Wort, welche verschiedenen Wortbedeutungen es besitzt. Zu jeder Wortbedeutung existiert dann ein beschreibender Wikipedia-Artikel. Beispielsweise besitzt das Wort „bank“ im Kontext von Geografie bzw. Meeren unterschiedliche Bedeutungen. Daher enthält Wikipedia Artikel zu „bank (geography)“ und „bank (sea floor)“. Wikipedia2Vec bezeichnet Wortbedeutungen auch als Entitäten. Wikipedia2Vec lernt die Repräsentation von Entitäten mithilfe des Anker-Kontext-Modells und des Graph-Kanten-Modells. Die resultierenden Repräsentationen sind dann SEs der korrespondierenden Wortbedeutungen. Das Anker-Kontext-Modell ähnelt dem Modell Skip-Gram (s. Abschnitt 2.6.1). Bei jedem Vorkommen einer Entität in einem Wikipedia-Artikel wird mit einem Hyperlink auf den Artikel der Entität verwiesen. Das Anker-Kontext-Modell nutzt die umliegenden Worte eines Hyperlinks zur Repräsentation der Entität, auf das der Verweis zeigt. Das Graph-Kanten-Modell lernt Repräsentationen von Entitäten, also SEs, durch das Vorhersagen von Nachbarentitäten in Wikipedias ungerichteten Entitäten-Graphen. Die Knoten des Entitäten-Graphen sind Entitäten. Die Kanten des Entitäten-Graphen entsprechen Verweisen in Artikeln von Entitäten auf andere Entitäten. Wikipedia2Vec setzt außerdem das Skip-Gram-LM ein, um zusätzlich zu SEs auch WEs der Worte in den Artikeln zu lernen. Yamada et al. evaluieren die Qualität ihrer WEs und SEs auf Basis der NLP-Aufgabe „Ähnlichkeiten finden“. Hierbei sollen Ähnlichkeiten von Artefakten ermittelt werden. Auf dem *SimLex-999*-Datensatz [HRK15] erzielt Wikipedia2Vec eine Genauigkeit von 40%. Dies übertrifft die Genauigkeit beim Einsatz von fastText auf diesem Testdatensatz um 3 Prozentpunkte.

3.2.5 Doc2Vec

Die bislang vorgestellten LMs repräsentieren einzelne Worte als Vektoren. Artefakte bestehen jedoch aus einer Folge von Worten. Ein Ansatz besteht darin, die Artefakte durch die Menge der WE ihrer Worte zu repräsentieren. Es bestehen jedoch auch LMs, die nicht nur Worte, sondern auch ganze Dokumente durch einen einzelnen Vektor repräsentieren. *Doc2Vec* ist ein solches LM. Es wurde von Le und Mikolov entwickeltet und in ihrer Arbeit „Distributed Representations of Sentences and Documents“ [LM] präsentiert. Doc2Vec ist in der Lage, Vektorrepräsentationen für Text(-abschnitte) variabler Länge zu ermitteln. Derartige Repräsentationen bezeichnet man auch als Paragrapheneinbettungen. Das Trainingsverfahren von Doc2Vec ähnelt dem von Word2Vec stark (s. Abschnitt 2.6.1). Für einen gegeben Textabschnitt $t = w_1 \dots w_n$ führt Doc2Vec zwei Trainingsaufgaben analog zu CBOW und Skip-Gram aus. Der Unterschied bei Doc2Vec besteht darin, dass zusätzlich zu den Vektoren für $w_1, \dots, w_n$ auch ein Vektor für t gelernt wird. Hierfür wird dem Klassifizierer neben $w_1, \dots, w_n$ auch ein Token t übergeben, für den ebenfalls Gewichtungen gelernt werden. Bei der ersten Trainingsaufgabe sagt Doc2Vec für einen gegebenen Teil des Textabschnitts $w_1 \dots w_i$ und Token t das nächste Wort w_{i+1} vorher. Somit wird Token t bei jeder Vorhersage einbezogen und seine Gewichtungen jedes Mal weitergelernt. Der Textabschnitt-Token stellt damit ein Gedächtnis dar, welches die Bedeutung des Textabschnitts repräsentiert. Bei der zweiten Trainingsaufgabe werden Worte des Textabschnitts mithilfe des Tokens t und seiner Gewichtungen vorhergesagt. Doc2Vec ist durch Repräsentation des Textabschnitts durch einen einzigen Vektor in der Lage, die Wortreihenfolge und die darin enthaltene Semantik einzubeziehen. Le und Marklov evaluierten Doc2Vec mithilfe einer IR-Aufgabe. Hierzu stellten sie einen Testdatensatz auf Basis der ersten zehn Resultate einer Internet-Suchmaschine für die 1 000 000 beliebtesten Suchanfragen zusammen. Für jede Suchanfrage gruppierten sie drei Textabschnitte, von denen zwei tatsächliche Resultate für diese Anfrage waren und einer das Resultat einer anderen Anfrage war. Ihr Ziel bestand in der Identifikation der beiden tatsächlichen Resultate der Anfrage. Ihr Werkzeug auf Basis von Doc2Vec lieferte Ergebnisse mit einer Fehlerrate von 3,82%. Ein Werkzeug auf Basis eines LMs zur WE statt Paragrapheneinbettung lieferte Ergebnisse mit einer Fehlerrate von 8,10%. Damit konnte Doc2Vec die Ergebnisse in Bezug auf die Fehlerrate um 32 Prozentpunkte verbessern.

3.3 Rückverfolgbarkeitsanalyse unter Einsatz von Sprachmodellen

Im vorherigen Abschnitt ist deutlich geworden, wie verschiedene LMs Sprache repräsentieren. Außerdem wurde aufgezeigt, dass LMs in vielen Bereichen der NLP bereits erfolgreich eingesetzt werden. Daher liegt die Überlegung nahe, LMs auch für ATLR einzusetzen. Im Folgenden sollen Arbeiten vorgestellt werden, die diesen Ansatz verfolgen.

3.3.1 WELR

Ein erstes Verfahren, welches LMs für die ATLR nutzt, ist *WELR*. Zhao und Cao präsentieren dieses Verfahren in ihrer Arbeit „An Improved Approach to Traceability Recovery Based on Word Embeddings“ [ZCS17]. WELR nutzt WE zur Berechnung der semantischen Ähnlichkeit zwischen Anforderungs- und Quelltextartefakten. Auf Basis dieser Ähnlichkeiten ermittelt WELR mögliche TLs. In einer Vorverarbeitungsphase werden zunächst Stoppworte entfernt. Anschließend wird die Binnenversalschreibweise von Quelltexteinheiten aufgetrennt und alle Worte in Kleinschreibweise transformiert. Das LM CBOW (s. Abschnitt 2.6.1) lernt daraufhin WEs für die Worte der Softwareartefakte. Anschließend werden mittels der Metrik IDF die für die Bedeutung eines Artefakts wesentlichsten Worte ermittelt. Die n wesentlichsten Worte eines Artefakts x bilden die Menge M_x. M_x soll die wesentliche Bedeutung des Artefakts x erfassen. Ein Ähnlichkeitsvergleich zwischen zwei

gegebenen Artefakten x und y erfolgt dann mithilfe von M_x und M_y. Die Ähnlichkeit eines Wortes $w_x \in M_x$ mit M_y berechnet sich über die maximale Ähnlichkeit zwischen w_x und einem Wort $w_y \in M_y$. Die Ähnlichkeit von M_x mit M_y berechnet sich auf Basis dieser Ähnlichkeiten aller $w_x \in M_x$ mit M_y. Zur Evaluation von WELR nutzen Zhao und Cao die Testdatensätze *eTour*, *iTrust* und *EasyClinic* des Forschungszentrums für Software- und System-Rückverfolgbarkeit (engl.: Center of Excellence for Software and Systems Traceability, CoEST) [SSTC]. Diese Testdatensätze enthalten natürlichsprachliche Anforderungs- und Java-Quelltext-Artefakte. Für den Testdatensatz eTour ermittelt WELR TLs mit einer Güte von 0,14 in F_1 bei einer Präzision von 0,28. Für den Testdatensatz EasyClinic ermittelt WELR TLs mit einer Güte von 0,43 in F_1 bei einer Präzision von 0,78.

3.3.2 S2Trace

Während WELR das LM CBOW verwendet, setzen Chen et al. bei ihrem Verfahren zur ATLR auf Doc2Vec (s. Abschnitt 3.2.5). Ihr Verfahren heißt *S2Trace* und wird in der Arbeit „Enhancing Unsupervised Requirements Traceability with Sequential Semantics“ [CWWW19] vorgestellt. S2Trace wird analog zu WELR zur ATLR zwischen Anforderungs- und Quelltextartefakten eingesetzt. S2Trace lernt Dokumenteinbettungen und generiert auf ihrer Basis TLs. Im ersten Schritt des Verfahrens erfolgt eine Textvorverarbeitung. Chen et al. setzen hierbei die Vorverarbeitungsschritte Stoppwortentfernung, Auftrennung der Binnenversalschreibweise und Stemming ein. Anschließend extrahiert S2Trace sequenzielle Muster aus den Artefakten. Sequenzielle Muster bezeichnen alle häufig auftretenden Wortsequenzen eines Textes. Diese sollen die grundlegende Themen-Semantik eines Artefakts repräsentieren. Mithilfe dieser sequenziellen Muster und den enthaltenen Worten lernt Doc2Vec Dokumenteinbettungen für die einzelnen Artefakte. Im letzten Schritt werden TLs auf Basis der Kosinus-Ähnlichkeit (s. Abschnitt 2.2.1.3) der Dokument-Vektoren ermittelt. Zwischen Artefakten mit einer Kosinus-Ähnlichkeit, die zu den höchsten 70% zählt, wird eine TL festgelegt. Für den Testdatensatz eTour ermittelt S2Trace TLs mit einer Güte von 0,16 in F_1 bei einer Präzision von 0,36. Für den Testdatensatz EasyClinic ermittelt S2Trace TLs mit einer Güte von 0,36 in F_1 bei einer Präzision von 0,7.

3.3.3 Ein Verfahren unter Einsatz eines Transformer-Sprachmodells

Die vorherigen Ansätze setzen statische LMs zur ATLR ein. Zhang et al. verwenden ein kontextsensitives LM (s. Abschnitt 2.6.2) auf Basis eines Transformer-NNs (s. Abschnitt 2.5.2). Ihr Verfahren präsentieren sie in ihrer Arbeit „Recovering Semantic Traceability between Requirements and Source Code Using Feature Representation Techniques“ [ZTGH21]. Sie setzen ihr Verfahren ebenfalls zur ATLR zwischen Anforderungs- und Quelltextartefakten ein. Hierbei erfolgt die Vorverarbeitung von Anforderungs- und Quelltextartefakten unterschiedlich. Bei der Vorverarbeitung von Anforderungen werden zunächst benannte Entitäten wie „New York“ extrahiert. Anschließend werden Stoppworte entfernt. Bei der Vorverarbeitung des Quelltexts wird die Binnenversalschreibweise von Quelltexteinheiten aufgetrennt und Worte mit nur einem Buchstaben entfernt. Kommentare im Quelltext werden analog zu Anforderungen vorverarbeitet. Nach dieser Phase enthalten Artefakte nur noch genau die Worte, die für ihre Semantik relevant sind. Für diese relevanten Worte werden anschließend kontextsensitive WEs durch ein Transformer-LM ermittelt. Dieses ermittelt gleichzeitig eine Vektorrepräsentation des Artefakts. Diese Artefakt-Vektoren werden mithilfe der Kosinusähnlichkeit verglichen. Zwischen Anforderungs- und Quelltextartefakten mit einer Kosinus-Ähnlichkeit größer n, wird eine TL festgelegt. Zhang et al. evaluieren ihr Verfahren auf Basis von iTrust und eTour. Für den Testdatensatz eTour ermittelt ihr Verfahren TLs mit einer Güte von 0,48 in F_1 für $n = 0,8$. Für den Testdatensatz iTrust ermittelt ihr Verfahren TLs mit einer Güte von 0,45 in F_1 für $n = 0,3$.

4 FTLR

In diesem Kapitel soll das Verfahren zur ATLR vorgestellt werden, welches dieser Arbeit zugrunde liegt. Ein Verständnis der grundlegenden Funktionsweise des Verfahrens ist essenziell, um die im nächsten Kapitel vorgestellten Anpassungen beim Einsatz alternativer LMs nachvollziehen zu können. Das Verfahren heißt *FTLR*, was für „fine-grained traceability link recovery“ steht. Hey et al. stellen es in ihrer Arbeit „Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations“ [HCWT21] vor. FTLR wird zum Ermitteln von TLs zwischen Anforderungs- und Quelltextartefakten eingesetzt. FTLR betrachtet jedes Artefakt als Menge von feingranularen Artefakt-Elementen. Ein Anforderungssatz ist ein Artefakt-Element eines Anforderungsartefakts. Ein Anforderungsartefakt kann optional auch einen Anwendungsfall (engl.: use case template, UCT) enthalten. Dieser UCT umfasst typischerweise eine Bezeichnung des Anwendungsfalls, eine kurze Beschreibung, die teilnehmenden Akteure, Eingangs- und Ausgangsbedingungen sowie den eigentlichen Ablauf. Die Bestandteile eines UCTs stellen dann ebenfalls Anforderungsartefakt-Elemente dar. Eine öffentliche Methode ist ein Artefakt-Element eines Quelltextartefakts. In die Repräsentation einer öffentlichen Methode fließen alle natürlichsprachlichen Elemente der Methode ein. Bei diesen handelt es sich um die Methodensignatur, den Namen der Klasse und optional Methodenkommentare (engl.: method comments, MC). FTLR nutzt das LM fastText (s. Abschnitt 2.6.1) zur Repräsentation von Artefakt-Elementen. Im Anschluss an die Repräsentation wird ein Ähnlichkeitsvergleich mittels der Metrik WMD (s. Abschnitt 2.2.1.3) zwischen Artefakt-Elementen durchgeführt. Bei einer hohen Ähnlichkeit wird eine feingranulare TL zwischen den Artefakt-Elementen definiert. TLs zwischen Artefakten werden dann auf Basis eines Mehrheitsvotums bestimmt. Bestehen zwischen vielen Elementen zweier Artefakte feingranulare TL, so wird eine TL zwischen diesen Artefakten ermittelt. Somit werden TLs auf Basis von feingranularen semantischen Ähnlichkeiten zwischen Artefakten erzeugt. FTLR lässt sich in zwei Phasen einteilen: Eine Vorkalkulationsphase und eine TL-Prozessierungsphase.

Abbildung 4.1 visualisiert die Vorkalkulationsphase. Hier werden die Artefakte zunächst vorverarbeitet. Hierzu setzt FTLR für Anforderungs- und Quelltextartefakte gleichermaßen die folgenden Vorverarbeitungsschritte ein: Nichtbuchstabenfilterung, Querverweisfilterung, Auflösung der Binnenmajuskelschreibweise, Kleinbuchstaben-Transformation, Lemmatisierung, Stoppwortentfernung und Wortlängenfilterung (s. Abschnitt 2.3). Anschließend werden die Artefakte in Artefakt-Elemente aufgeteilt. Im nächsten Schritt werden die Worte der Artefakt-Elemente mithilfe eines vortrainierten fastText-LMs durch WEs repräsentiert. Das verwendete fastText-Modell wurde auf dem Wikipedia- und dem *Com-*

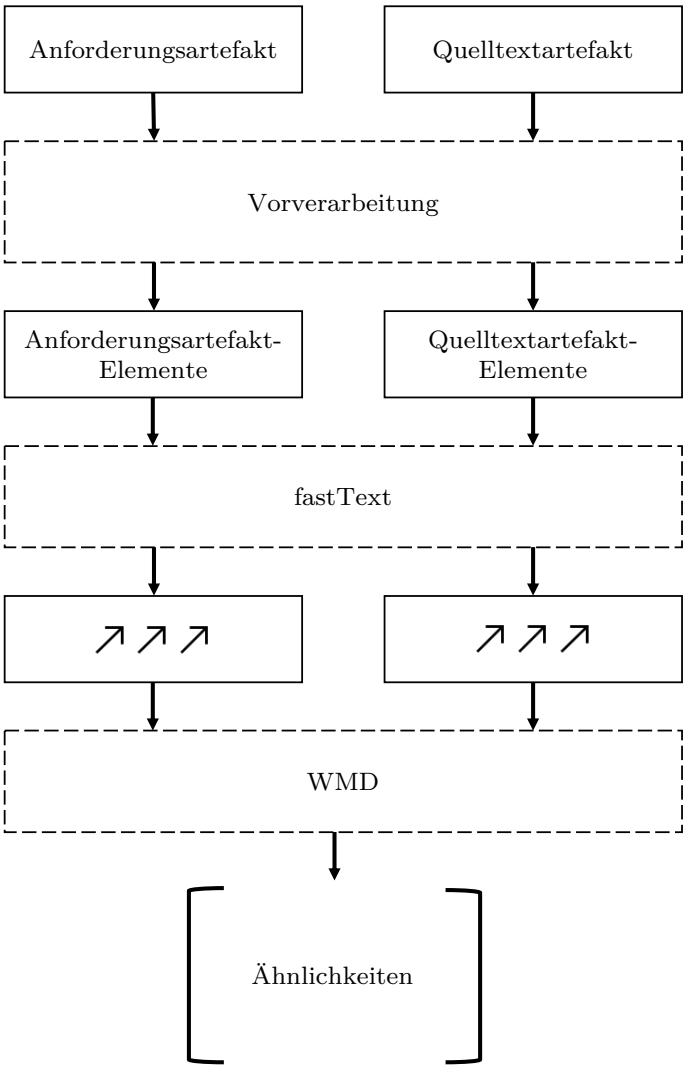


Abbildung 4.1: Vorkalkulationsphase von FTLR

monCrawl-Datensatz [GBG+18] vortrainiert. Der CommonCrawl-Datensatz besteht aus Texten des Internets unterschiedlichen Typs. Ein Artefakt-Element wird als BOE (s. Abschnitt 2.2.1.3) repräsentiert. Daraufhin führt FTLR einen Ähnlichkeitsvergleich der Artefakt-Elemente durch. Als Metrik der Ähnlichkeit verwendet FTLR WMD (s. Abschnitt 2.2.1.3).

In der TL-Prozessierungsphase aggregiert FTLR diese feingranularen Ähnlichkeiten zu TLs wie in Abbildung 4.2 dargestellt. Hierzu ermittelt FTLR zunächst für jedes Element q eines Quelltextartefakts Q das ähnlichste Element a eines Anforderungsartefakts A. FTLR ermittelt eine feingranulare TL zwischen q und a, wenn der WMD-Ähnlichkeitswert unter einem Schwellenwert von 0,59 liegt. Es folgt ein Mehrheitsvotum zur Aggregation der feingranularen Verbindungen. Dieses soll sicherstellen, dass die dominierende Aufgabe und somit Semantik einer Klasse die Bildung einer TL bestimmt. Besteht eine feingranulare TL zwischen $q \in Q$ und $a \in A$, votiert q für eine TL zwischen Q und A. Zwischen der Anforderung A, für die die meisten $q \in Q$ votiert haben, und Q wird eine Kandidaten-TL ermittelt. Besteht zwischen A und Q eine feingranulare Verbindung, deren WMD-Wert geringer als ein Schwellenwert von 0,44 ist, wird die Kandidaten-TL zur TL. FTLR bestimmt durch den Einsatz von zwei Schwellenwerten die bestmöglichen Kandidaten-TLs und stellt dennoch ein Minimum an Ähnlichkeit zwischen verbundenen Artefakten sicher. Zur Evaluierung der Ergebnisse von FTLR nutzen Hey et al. u.a. die Testdatensätze eTour

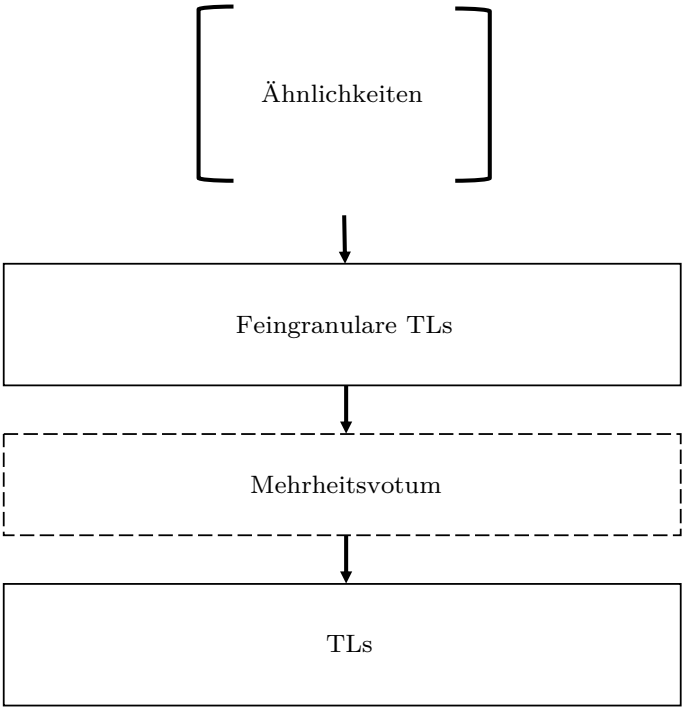


Abbildung 4.2: RV-Prozessierungsphase von FTLR

und iTrust des CoEST [SSTC]. Auf dem Testdatensatz eTour ermittelt FTLR TLs mit einer Güte von 0,472 in F_1 und einer durchschnittlichen Präzision (engl.: mean average precision, MAP) von 0,471. Auf dem Testdatensatz iTrust ermittelt FTLR TLs mit einer Güte von 0,231 in F_1 und einer MAP von 0,258. Im Durchschnitt über alle verwendeten Testdatensätze ermittelte FTLR TLs mit einer Güte von 0,327 in F_1. FTLR erzielt damit auf den Testdatensätzen des CoEST bessere Ergebnisse als vergleichbare Verfahren wie das in Abschnitt 3.3.2 vorgestellte Verfahren S2Trace .

5 Analyse und Entwurf

FTLR wird zur ATLR zwischen Anforderungen und Quelltextklassen eingesetzt (s. Kapitel 4). Anforderungen sind Kriterien an Softwareprodukte und spezifizieren das erwartete Softwaresystem. Quelltextklassen bilden die Grundlage des Softwareprodukts und sollen im Zusammenspiel alle Anforderungen erfüllen. Eine Quelltextklasse sollte über eine TL mit einer Anforderung verbunden sein, wenn sie die in der Anforderung beschriebene Funktionalität implementiert. Anforderungen und Quelltextklassen unterscheiden sich sprachlich stark. Anforderungen sind in natürlicher Sprache verfasst, Quelltextklassen in einer Programmiersprache. Daraus ergeben sich grundlegende syntaktische Unterschiede. Der gravierendste Unterschied besteht jedoch in ihrem sprachlichen Abstraktionsniveau. Anforderungen beschreiben die Funktionalitäten eines Systems sehr abstrakt. Quelltextklassen enthalten konkrete Implementierungen dieser Funktionalitäten. Implementierungsdetails sind wenig abstrakt. Korrespondierende Anforderungen und Quelltextklassen haben somit zwar ähnliche Bedeutungen, doch können syntaktisch und abstraktionstechnisch stark unterschiedlich formuliert sein. Häufig enthalten sie zwar bedeutungsähnliche aber nicht identische Worte. Verfahren auf Basis von IR-Techniken (s. Abschnitt 3.1.1), die Artefakte allein auf Basis von Worthäufigkeiten repräsentieren, sind damit für diesen Anwendungsfall ungeeignet. Verfahren zur ATLR zwischen Anforderungen und Quelltextklassen müssen die bestehende Abstraktionskluft überbrücken [HCWT21]. Ein vielversprechender Ansatz besteht in der Nutzung von LMs (s. Abschnitt 2.6). Diese sind in der Lage, die grundlegende Semantik von Artefakten zu repräsentieren. Dies begründet die Vermutung, dass sie zum Erfassen semantischer Ähnlichkeiten zwischen unterschiedlich abstrakten Artefakten geeignet sind. Auf Basis dieser Vermutung setzt FTLR das LM fastText (s. Abschnitt 2.6.1) zur ATLR ein. Dieses erweist sich im Anwendungsfall als geeignet. Es besitzt jedoch drei Schwachstellen.

Erstens ermittelt fastText die Semantik von Worten durch Betrachten von benachbarten Worten. Zwei Worte sind sich demnach genau dann semantisch ähnlich, wenn sie beide häufig in Kombination mit denselben anderen Wörtern auftreten. Durch dieses Vorgehen erkennt fastText keine semantische Ähnlichkeit zwischen sich eigentlich ähnlichen Worten, die häufig in unterschiedlicher Wortgesellschaft vorkommen. Dies soll an Beispiel 5.1 veranschaulicht werden. In diesem Beispiel sind die zwei semantisch ähnlichen Begriffe „building“ und „house“ gegeben. Darüber hinaus liegt eine Beschreibung bzw. Definition dieser Begriffe auf Basis der korrespondierenden Wikipedia-Artikel vor. Es wird deutlich, dass sich die Beschreibung von „building“ auf physische, strukturelle Eigenschaften eines Gebäudes beschränkt. Worte wie „structure“, „roof“, „wall“ und „function“ dominieren. Andererseits

wird der Begriff „house“ über seine Funktion als Heimat einer sozialen Einheit definiert. Worte wie „social“, „household“, „family“ und „group“ dominieren. Die Begriffe „building“ und „house“ kommen damit in diesen Texten trotz ihrer semantischen Ähnlichkeit in unterschiedlicher Wortgesellschaft vor. Lernt fastText ihre Bedeutung auf Basis dieser Texte, repräsentiert das LM ihre semantische Ähnlichkeit nicht genau genug.

Beispiel 5.1: Unterschiedliche Wortgesellschaft ähnlicher Worte

Building: A building is a structure with a roof and walls standing more or less permanently in one place. Buildings come in a variety of sizes, shapes, and functions, and have been adapted throughout history for a wide number of factors.

House: A house is the home of a social unit. This social unit is known as a household. Most commonly, a household is a family unit of some kind, although households may also be other social groups, such as roommates or, in a rooming house, unconnected individuals.

Zweitens ist fastText ein statisches LM (s. Abschnitt 2.6.1). Das bedeutet, es repräsentiert jedes Wort durch einen einzelnen festen Vektor. Worte können mehrere unterschiedliche, vom Kontext abhängende Bedeutungen besitzen (s. Abschnitt 2.3). Dies wird bei Betrachtung von Beispiel 5.2 deutlich. Das Wort „bank“ bezeichnet im Kontext von Finanzen eine Institution, die Geldgeschäfte betreibt. Diese Bedeutung wird im ersten Satz des Beispiels verwendet. Im Kontext von Gewässer beschreibt „bank“ das Land am Ufer eines Flusses. Diese Bedeutung wird im zweiten Satz des Beispiels verwendet. fastText ist nicht in der Lage, diese verschiedenen Bedeutungen eines Wortes in Abhängigkeit des Kontexts geeignet zu repräsentieren. Stattdessen kombiniert fastText die Bedeutungen in den unterschiedlichen Kontexten in eine einzige WE. Damit ist die resultierende WE von „bank“ ungenau. fastText repräsentiert weder die Tatsache, dass „bank“ mehrere kontextabhängige Bedeutungen hat, noch repräsentiert es eine der beiden Bedeutungen korrekt. Alle Kontextworte von „bank“ (z.B. „loan“ und „boat“) beeinflussen die WE unabhängig von der vorliegenden Wortbedeutung. In FTLR wird ein auf dem CommonCrawl-Vortrainingsdatensatz [GBG+18] vortrainiertes fastText-LM eingesetzt (s. Kapitel 4). Es ist davon auszugehen, dass für jedes Wort in der Regel eine Wortbedeutung auf diesem Datensatz dominiert. Das bedeutet, die meisten Worte werden häufig in einem bestimmten Kontext und damit mit einer bestimmten Bedeutung verwendet. Die WE von fastText wird dann stark durch diese dominierende Wortbedeutung beeinflusst. Wird ein Wort in einem Softwareprojekt, auf dem FTLR ausgeführt wird, mit einer anderen Bedeutung verwendet, repräsentiert fastText das Wort und damit das Artefakt-Element ungenau.

Beispiel 5.2: Kontextabhängige Bedeutungen

I had to take out a bank loan to start my own business.

We pushed the boat off from the bank.

fastText repräsentiert in FTLR neben natürlichsprachlichen Anforderungen auch Methoden durch Vektoren (s. Kapitel 4). Das in FTLR eingesetzte fastText-LM ist jedoch nicht auf quelltextnahen Datensätzen vortrainiert worden. Daher ist das fastText-LM nicht in der Lage, geeignete Repräsentationen für reinen Quelltext als Eingabe zu ermitteln [Che20]. Aus diesem Grund erhält fastText ausschließlich den natürlichsprachlichen Methodennamen, die Methodenparameter, den Klassennamen und optional MCs als Eingabe. Der Methodenrumpf (engl.: method body, MB) trägt jedoch ebenfalls zur Semantik der Me-

thode bei. Dieser wird vom fastText-LM nicht zur Repräsentation der Methode genutzt. Dies kann zu Ungenauigkeiten in der Repräsentation führen. Es wird deutlich, dass alle drei Schwachstellen des in FTLR eingesetzten fastText-LMs zu ungenauen Repräsentationen von Anforderungen und Quelltextklassen führen können. FTLR ermittelt TLs auf Basis der von fastText erzeugten Repräsentationen. Ungenauigkeiten der Ergebnisse von fastText können damit zu Ungenauigkeiten der Ergebnisse von FTLR führen. Damit stellen die Schwachstellen von fastText eine mögliche Problematik für FTLR dar. LMs wie BERT (s. Abschnitt 2.6.2) und XLNet (s. Abschnitt 3.2.3) repräsentieren Worte kontextsensitiv. LMs wie CodeBERT (s. Abschnitt 3.2.1) und UniXcoder (s. Abschnitt 3.2.2) sind auf quelltextnahen Trainingsdaten vortrainiert. Es existieren demnach LMs, die zumindest einen Teil der aufgeführten Schwachstellen von fastText nicht aufweisen. Diese LMs stellen folglich eine potentielle Lösung für die durch fastText möglicherweise verursachten Ungenauigkeiten von FTLR dar.

Diese Arbeit setzt sich zum Ziel, verschiedene alternative LMs auf ihre Eignung zum Einsatz in FTLR zu untersuchen. LMs ohne fastText‘s Schwachstellen könnten in der Lage sein, genauere Repräsentationen von Anforderungen und Quelltextklassen zu erzeugen. Dies könnte zu genaueren Ergebnissen von FTLR führen. Zur Zielerreichung soll im ersten Schritt eine Menge an aussichtsreichen Kandidaten an zu untersuchende LMs gebildet werden. Hierzu werden zuerst Eigenschaften definiert, die ein LM zu einem aussichtsreichen Kandidaten machen. Im Wesentlichen ist ein LM ein aussichtsreicher Kandidat, wenn erwartet werden kann, dass es die vorliegenden Artefakte möglichst geeignet repräsentiert. Zudem sollte ein untersuchenswertes LM mindestens eine von fastText‘s Schwachstellen nicht besitzen. Außerdem sollte es keine neue, bislang nicht aufgeführte Schwachstelle aufweisen. Daraufhin werden bestehende LMs in Bezug darauf untersucht, ob sie die definierten Eigenschaften besitzen. Es werden insbesondere LMs analysiert, die bereits erfolgreich in einem ähnlichen Anwendungsfall eingesetzt wurden. Auf Basis der Menge an aussichtsreichen Kandidaten sollen in einer ersten Entwurfsentscheidung die vielversprechendsten ausgewählt werden. Diese werden in dieser Arbeit auf ihre Eignung zum Einsatz in FTLR untersucht. Im zweiten Schritt sollen Möglichkeiten der Vorverarbeitung für die Eingabe der zu untersuchenden LMs analysiert und ausgewählt werden. Beispielsweise können je nach LM andere Vorverarbeitungsschritte vorteilhaft sein als die bislang von FTLR genutzten. Im dritten Schritt wird analysiert und entschieden, wie der Ähnlichkeitsvergleich der von den LMs erzeugten Repräsentationen erfolgt.

5.1 Analyse bestehender Sprachmodelle

Das Ziel dieses Abschnitts besteht darin zu analysieren, welche LMs aussichtsreiche Kandidaten für einen Einsatz in FTLR im Rahmen dieser Arbeit darstellen. Hierzu werden zuerst gewünschte Eigenschaften der zu untersuchenden LMs definiert. Anschließend wird analysiert, welche LMs möglichst viele dieser Eigenschaften besitzen. In dieser Arbeit sollen LMs untersucht werden, die sich für den vorliegenden Anwendungsfall vermutlich besser eignen als fastText. fastText ist aufgrund seiner Schwachstellen möglicherweise ungeeignet für den Einsatz in FTLR im Vergleich zu anderen LMs Ein LM ohne fastText’s Schwachstellen und ohne neue gravierende Schwachstellen könnte besser für den Einsatz in FTLR geeignet sein. Erwünschte Eigenschaften von LMs sind damit genau solche Eigenschaften, die Schwachstellen von fastText auflösen und keine neuen Schwachstellen hinzufügen.

fastText repräsentiert ähnliche Worte mit unterschiedlicher Wortgesellschaft als unterschiedlich. Dies ist eine allgemeine Problematik von LMs. LMs repräsentieren Worte darüber, mit welchen Worten sie häufig in Kombination auftreten. Treten ähnliche Worte nie gemeinsam mit ähnlichen Worten auf, können sie nicht als ähnlich repräsentiert werden. Dieser Fall tritt jedoch empirisch gesehen nur sehr selten auf [JM22]. In der Regel kommen

ähnliche Wörter zumindest in ähnlichen Themenbereichen vor. Damit sind die Worte, mit denen sie gemeinsam vorkommen, mit hoher Wahrscheinlichkeit ähnlich zueinander. Mit ihrer Hilfe kann dann auch eine Ähnlichkeit der Ausgangsworte repräsentiert werden. Die erste Schwachstelle ist damit eine allgemeine Schwachstelle von LMs und besitzt empirisch nur wenig Relevanz.

fastText repräsentiert Worte statisch. Daraus können ungenaue, ungeeignete Repräsentationen von Wortbedeutungen resultieren. Beispielsweise kann der Fall eintreten, dass in Trainingstexten von fastText eine bestimmte Bedeutung eines Wortes dominiert. Dementsprechend repräsentiert die WE des Wortes vor allem diese Bedeutung. Wird in einem Datensatz, auf dem FTLR ausgeführt wird, jedoch eine andere Bedeutung desselben Wortes verwendet, ist die WE für die Repräsentation des Wortes ungeeignet. Somit kann dies dazu führen, dass fastText Worte und damit Artefakt-Elemente ungenau repräsentiert. Ein kontextsensitives LM ermittelt eine WE in Abhängigkeit des vorliegenden Kontexts (s. Abschnitt 2.6.2). Trotz Dominanz einer anderen Wortbedeutung in Trainingstexten des LMs, könnte ein kontextsensitives LM damit in der Lage sein, geeignetere WEs als fastText zu liefern. Durch stets an die vorliegende Wortbedeutung angepasste WEs, ist ein kontextsensitives LM somit möglicherweise in der Lage, genauere Repräsentationen von Artefakten zu liefern als fastText. Ein zu untersuchendes LM sollte damit die Eigenschaft besitzen kontextsensitiv zu sein.

fastText wurde auf natürlichsprachlichen Texten vortrainiert. Damit ist fastText mit der Struktur und Beschaffenheit natürlicher Sprache vertraut. Aus diesem Grund ist fastText vermutlich für die Repräsentation von natürlichsprachlichen Anforderungen geeignet. Da fastText nicht auf Quelltext vortrainiert wurde, eignet sich das LM nicht für die Einbeziehung des MBs in die Repräsentation eines Quelltextartefakt-Elements. Dadurch fließen möglicherweise wichtige semantische Informationen der Methode nicht in ihre Repräsentation ein. Ein Vortraining auf natürlicher Sprache und Quelltext löst diese Schwachstelle. Ein auf Quelltext vortrainiertes LM kann einen größeren Teil der semantisch relevanten Elemente einer Methode in ihre Repräsentation einfließen zu lassen. Hiermit könnte ein auf Quelltext vortrainiertes LM Methoden möglicherweise geeigneter für FTLR repräsentieren als fastText. Daher besteht eine gewünschte Eigenschaft darin, dass ein zu untersuchendes LM auf Quelltext vortrainiert wurde. Ein Vortraining auf ausschließlich Quelltext würde jedoch zu der neuen Schwachstelle führen, dass natürlichsprachliche Anforderungen nicht mehr genau genug repräsentiert werden könnten. Daher sollte ein geeignetes LM auch auf natürlicher Sprache vortrainiert sein.

Nun sollen bestehende LMs analysiert werden. Der Gegenstand der Analyse besteht darin zu eruieren, ob sie die beiden eingeführten gewünschten Eigenschaften besitzen. Ein untersuchenswertes LM soll kontextsensitive WE ermitteln. Außerdem soll es auf Quelltext und natürlicher Sprache vortrainiert sein.

CBOW

Das ATLR-Verfahren WELR setzt Word2Vec auf Basis des LMs CBOW (s. Abschnitt 2.6.1) zur Repräsentation von Artefakten ein (s. Kapitel 3.3.1). CBOW ist ein statisches LM (s. Abschnitt 2.6.1). Das in WELR eingesetzte CBOW-Modell wurde auf Trainingsdatensätzen vortrainiert, die ausschließlich natürlichsprachliche Texte enthalten. Quelltext ist nicht enthalten. CBOW besitzt damit keine der gewünschten Eigenschaften. Des Weiteren erzielt WELR Ergebnisse, deren Güte nicht zufriedenstellend ist. Auf dem Testdatensatz eTour liegt die Güte der TLs bei nur 0,140 in F_1 (s. Abschnitt 3.3.1). FTLR ermittelt auf eTour Ergebnisse mit einer Güte von 0,472 in F_1, wie in Kapitel 4 beschrieben wurde. Die um 0,332 schlechteren F_1-Werte von WELR legen die Vermutung nahe, dass ein CBOW-LM im vorliegenden Anwendungsfall semantisch ähnliche Artefakte nicht ähnlich genug für einen geeigneten Einsatz in FTLR repräsentiert.

Doc2Vec

Das LM Doc2Vec, vorgestellt in Abschnitt 3.2.5, wird im ATLR-Verfahren S2Trace (s. Abschnitt 3.3.2) eingesetzt. S2Trace wurde für den denselben Anwendungsfall wie FTLR entwickelt. Doc2Vec ermittelt Repräsentationen für große sprachliche Einheiten wie Sätze oder Paragrafen. Informationen über die Wortreihenfolge, die entscheidenden Einfluss auf die Bedeutung einer Wortfolge haben kann, werden in die Repräsentation einbezogen. Doc2Vec lernt zusätzlich Repräsentationen für Worte. Diese WEs sind statisch. Es existieren keine vortrainierten Doc2Vec-LMs. Damit besitzt Doc2Vec keine der gewünschten Eigenschaften. Es lässt sich jedoch auch Folgendes heranführen: Anforderungen bestehen aus natürlichsprachlichen Sätzen. Quelltexteinheiten enthalten stichwortartige Bezeichner und Methodenkommentare, die künstlich in eine natürlichsprachliche, satzähnliche Form gebracht werden können. Daher eigenen sich Anforderungs- und Quelltexteinheiten als Eingabe für Doc2Vec. Doc2Vec ist in der Lage, in IR-Problemstellungen deutlich bessere Ergebnisse in der Metrik Fehlerrate zu erzielen als LMs, die ausschließlich WE ermitteln können (s. Abschnitt 3.2.5). Daher ist davon auszugehen, dass das LM in diesem verwandten Problem der ATLR geeignete Repräsentation von Anforderungs- und Quelltextartefakt-Elementen ermitteln kann. Andererseits erzielte S2Trace mit Doc2Vec kaum bessere Ergebnisse als WELR mit CBOW. Auf dem Testdatensatz eTour sind die von S2Trace ermittelten TLs um nur 2 Prozentpunkte genauer in F_1 als die von WELR. Damit sind die Ergebnisse von S2Trace ebenfalls wenig zufriedenstellend (s. Abschnitt 3.3.2). Die statischen Repräsentationen, der Mangel an vortrainierten Modellen und die wenig zufriedenstellenden Ergebnisse in vergleichbaren Arbeiten lassen darauf schließen, dass Doc2Vec vermutlich nicht für den Einsatz in FTLR geeignet ist.

CodeBERT

CodeBERT ist ein auf dem BERT-Modell basierendes LM (s. Abschnitt 3.2.1). Es wurde bereits erfolgreich in IR-Anwendungsfällen mit natürlicher Sprache in Kombination mit Quelltext eingesetzt. CodeBERT ermittelt analog zu BERT Worteinbettungen unter Einbeziehung aller Worte eines Satzes als Kontext. CodeBERT repräsentiert Worte somit nicht isoliert, sondern in Abhängigkeit ihrer Kontextworte und ihrer Position im Satz. CodeBERT ermittelt demnach kontextsensitive WEs. CodeBERT wurde auf einem Datensatz vortrainiert, der sowohl natürliche Sprache als auch Quelltext enthält [HWG+19]. Außerdem besteht ein Viertel des Datensatzes aus Artefakten, in denen natürliche Sprache in Kombination mit Quelltext auftritt. Diese bimodalen Artefakte enthalten beispielsweise Quelltext und beschreibender Dokumentation. Bei der Trainingsdirektive des MLM (s. Abschnitt 2.6.2) wurden zur Vorhersage der maskierten Worte natürlich-sprachliche Worte und Quelltext-Tokens genutzt. Damit lernte CodeBERT im Vortraining auch bereits, wie Quelltext und natürliche Sprache zusammenhängen können. CodeBERT besitzt damit beide gewünschten Eigenschaften. Dies lässt bereits vermuten, dass es sich um einen vielversprechenden Kandidaten handelt. Darüber hinaus liefert ein Verfahren auf Basis von CodeBERT bei der NLP-Aufgabe Quelltextsuche auf Java-Quelltext Ergebnisse mit einem MRR von 0,75 (s. Abschnitt 3.2.1). Bei der Quelltextsuche wird für eine gegebene natürlichsprachliche Beschreibung der relevanteste Quelltextabschnitt gesucht. Dies spricht für eine hohe Eignung von CodeBERT in Anwendungsfälle mit natürlicher Sprache und Quelltext. CodeBERT wird auch explizit als geeignet für derartige Anwendungsfälle deklariert.

UniXcoder

UniXcoder basiert auf einem Transformer-Modell (s. Abschnitt 3.2.2). Es ist für Anwendungsfälle mit Quelltext und natürlicher Sprache geeignet. Dies resultiert aus einem Vortraining auf demselben Datensatz wie CodeBERT [HWG+19]. Dieser enthält neben natürlichsprachlichen Texten auch Quelltext. UniXcoder ist ein kontextsensitives LM. Es

lernt WEs stets in Abhängigkeit des Kontexts einer sprachlichen Einheit neu. Für die Repräsentation von Quelltexteinheiten bezieht es benachbarte Worte bzw. Quelltext-Tokens, natürlichsprachliche Quelltextkommentare und die Struktur des Quelltexts ein. Durch die Trainingsdirektive des MLM wird natürliche Sprache zur Vorhersage von Quelltext und umgekehrt genutzt. Damit hat UniXcoder bereits im Vortraining gelernt, Zusammenhänge zwischen natürlicher Sprache und Quelltext zu erkennen. Das LM erzielt insbesondere sehr zufriedenstellende Ergebnisse in NLP-Aufgaben, in denen das Verständnis der Funktion eines Quelltextabschnitts entscheidend ist. Eine solche NLP-Aufgabe ist beispielsweise Quelltextsuche. UniXcoder erzielt bei der Quelltextsuche auf dem CodeSearchNet-Datensatz [HWG+19] Ergebnisse mit einem MRR von durchschnittlich 0,74. Damit übertreffen seine Ergebnisse die von CodeBERT um 5 Prozentpunkte. UniXcoder besitzt somit beide gewünschten Eigenschaften und erzielt hervorragende Ergebnisse beim Erkennen von Ähnlichkeiten zwischen Quelltext und natürlicher Sprache. Damit stellt es einen sehr vielversprechenden zu untersuchenden Kandidaten dar.

XLNet

XLNet ist ein autoregressives LM (s. Abschnitt 3.2.3). Es erzielt bessere Ergebnisse als BERT in zwanzig NLP-Aufgaben. XLNet erhält analog zu CodeBERT und UniXcoder einen Satz als Eingabe. Ein Wort wird in Abhängigkeit seiner Kontextworte im Satz repräsentiert. Durch Permutation der Reihenfolge der Worte im Satz kann auf eine Maskierung von Worten wie bei BERT verzichtet werden. XLNet ermittelt damit analog zu CodeBERT und UniXcoder kontextsensitive WEs. Zwei Worte werden ähnlich repräsentiert, wenn sie häufig in Sätzen mit ähnlichen Worten vorkommen. Die Position eines Wortes im Satz wird nicht einbezogen. Damit gehen im Vergleich zu CodeBERT und UniXcoder semantische Informationen der Wortreihenfolge verloren. XLNet wurde auf dem BooksCorpus [ZKZ+15] vortrainiert. Dieser Datensatz enthält Texte natürlicher Sprache, aber keinen Quelltext. Damit besitzt XLNet nur eine der beiden gewünschten Eigenschaften. Es ist davon auszugehen, dass XLNet für Anforderungen in natürlicher Sprache sehr geeignete Einbettungen generiert. Hierfür spricht zum einen das Vortraining auf natürlichsprachlichen Datensätzen. Zum anderen erzielt XLNet hervorragende Ergebnisse bei der NLP-Aufgabe Leseverstehen auf natürlichsprachlichen Texten, mit einer MAP von 85,4% (s. Abschnitt 3.2.3). Diese Ergebnisse übertreffen die Ergebnisse von BERT um 13 Prozentpunkte. Es ist durchaus untersuchenswert, inwiefern XLNet auch geeignete Repräsentationen für Quelltexteinheiten ermittelt. Dagegen spräche, dass XLNet nicht auf Quelltext-basierten Daten vortrainiert wurde und keine entsprechend vortrainierten Versionen existieren. Dafür spräche, dass XLNet durch Permutation der Wortreihenfolge nicht auf eine Satzähnlichkeit der Eingabe angewiesen sein. Daher eignet sich XLNet vermutlich zur Repräsentation von Methodennamen, -parametern und -kommentaren, die bestenfalls satzähnlich sind. Insgesamt eignet sich XLNet vermutlich sehr für die Repräsentation von Anforderungen.

Wikipedia2Vec

Wikipedia2Vec ist ein bedeutungseinbettendes LM und wurde in Abschnitt 3.2.4 vorgestellt. Es basiert auf der Wikipedia-Wissensdatenbank. Diese Wissensdatenbank enthält Worte, z.B. „power“ und speichert ihre Wortbedeutungen in verschiedenen Kontexten, beispielsweise „power (physics)“ und „power (social and political)“. Zu jeder Wortbedeutung hat Wikipedia2Vec auf Basis der korrespondierenden Artikel eine SE gelernt. Für ein zu repräsentierendes Wort muss im Vorfeld eine Bestimmung seiner Wortbedeutung erfolgen. Dies ist die Aufgabe eines Verfahrens zur WSD (s. Abschnitt 2.6.3). Die Bedeutung wird in Abhängigkeit des Kontexts ermittelt. Wikipedia2Vec erhält die Wortbedeutung als Eingabe und liefert eine SE. Das einzubettende Wort wird dann durch diese SE repräsentiert.

LM	Kontextsensitiv	Geeignetes Vortraining
fastText	X	X
CBOW	X	X
Doc2Vec	X	X
CodeBERT	√	√
UniXcoder	√	√
XLNet	√	X
Wikipedia2Vec	√	X

Tabelle 5.1: Sprachmodelle und gewünschte Eigenschaften

Die Wortbedeutung wurde in Abhängigkeit des Kontexts ermittelt, damit hängt ihre Repräsentation und damit die des Wortes vom vorliegenden Kontext ab. Daher sind die von Wikipedia2Vec ermittelten Repräsentationen kontextsensitiv. Wikipedia2Vec wurde auf dem Wikipedia-Datensatz vortrainiert. Wikipedia-Artikel enthalten überwiegend natürlichsprachliche Texte, jedoch keinen Quelltext. Damit ist Wikipedia2Vec kontextsensitiv, aber nicht auf geeigneten Daten vortrainiert. Damit besitzt es eine der beiden gewünschten Eigenschaften. Wikipedia2Vec verfolgt als bedeutungseinbettendes LM einen zu den anderen LMs einzigartigen Ansatz. Es stellt auch vor allem deshalb einen sehr interessanten Ansatz dar, da es das Konzept von statischen WEs um das Konzept von kontextsensitiven SEs erweitert. Erhält Wikipedia2Vec ein Wort als Eingabe, zu welchem kein Wikipedia-Artikel existiert, so liefert es eine statische, vortrainierte WE. Existieren jedoch Wikipedia-Artikel zu einem gegebenen Wort, so kann eine kontextsensitive SE ermittelt werden. Aufgrund der Fähigkeit, auch statische WE zu erzeugen und eines ähnlichen Vortrainingsdatensatzes, ist Wikipedia2Vec vermutlich nicht weniger gut geeignet für den Einsatz in FTLR als fastText. SEs können Worte genauer repräsentieren als WEs, falls die im Vortraining dominierende Wortbedeutung von der verwendeten Bedeutung abweicht. Dies kann sich positiv auf die Genauigkeit der Repräsentation der Artefakt-Elemente auswirken. Damit stellt Wikipedia2Vec einen vielversprechenden Ansatz dar. Darüber hinaus übertrifft Wikipedia2Vec fastText in der NLP-Aufgabe des Analogien-Findens in der Metrik MAP um 16 Prozentpunkte (s. Abschnitt 3.2.4).

5.2 Auswahl bestehender Sprachmodelle

Auf Basis der Analyseergebnisse aus dem vorherigen Abschnitt 5.1 sollen nun die in dieser Arbeit zu untersuchenden LMs ausgewählt werden. Tabelle 5.1 visualisiert, welche der im letzten Abschnitt analysierten LMs welche gewünschten Eigenschaften besitzen. Es ist zu sehen, dass CBOW und Doc2Vec keine der beiden gewünschten Eigenschaften besitzen. Darüber hinaus wurde deutlich, dass Verfahren zur ATLR, die eines dieser beiden LMs einsetzen, wenig zufriedenstellende Ergebnisse liefern. Damit sind CBOW und Doc2Vec mit hoher Wahrscheinlichkeit nicht für den Einsatz im ATLR-Verfahren FTLR. Daher werden sie in dieser Arbeit nicht untersucht. XLNet ist analog zu UniXcoder und CodeBERT ein kontextsensitives LM. Im Gegensatz zu UniXcoder und CodeBERT, die auf einem Transformer-Modell basieren, setzt XLNet auf autoregressives Training. Verfahren zur ATLR basierend auf Transformer-LMs erzielen sehr zufriedenstellende Ergebnisse (s. Abschnitt 3.3.3). Dies spricht dafür, dass sich UniXcoder und CodeBERT besonders für den Einsatz in FTLR eignen. Außerdem ist XLNet im Gegensatz zu UniXcoder und CodeBERT nicht auf quelltextnahen Daten vortrainiert. Daher ist davon auszugehen, dass UniXcoder und CodeBERT besser für den vorliegenden Anwendungsfall geeignet sind als XLNet. Daher wird XLNet in dieser Arbeit nicht untersucht. Tabelle 5.1 veranschaulicht, dass CodeBERT und UniXcoder als einzige vorgestellte LMs beide gewünschten Eigen-

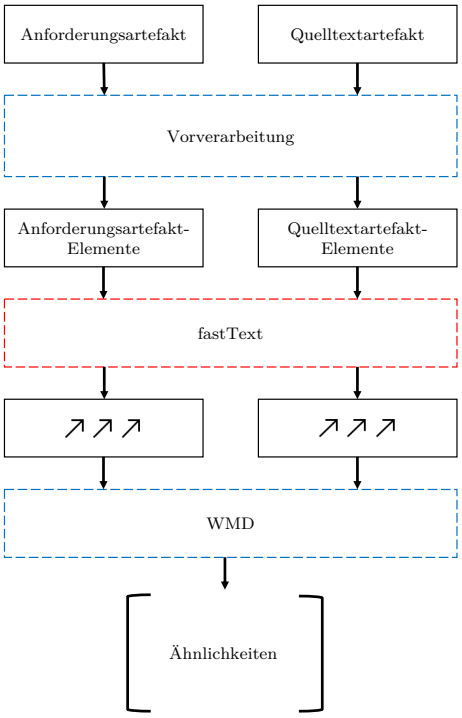


Abbildung 5.1: Anpassungen des FTLR-Verfahrens

schaften erfüllen. Darüber hinaus wurde analysiert, dass sie vermutlich in der Lage sind, Anforderungen und Quelltexteinheiten geeignet für die Weiterverarbeitung in FTLR zu repräsentieren. Die Ansätze beider LMs sind sich sehr ähnlich. Beide LMs basieren auf Transformer-Modellen und wurden auf demselben Datensatz mit ähnlichen Trainingsdirektiven vortrainiert. Es ist folglich sinnvoll nur eines der beiden LMs in dieser Arbeit zu untersuchen. UniXcoder erzielt in Aufgabenstellungen, in denen Ähnlichkeiten zwischen natürlicher Sprache und Quelltext gefunden werden müssen, bessere Ergebnisse als CodeBERT. Da diese Aufgabenstellung zu ihrem Einsatzgebiet in FTLR verwandt ist, ist davon auszugehen, dass sich UniXcoder besser für den Einsatz in FTLR eignet als CodeBERT. Daraus resultiert Hypothese 5.1 in Anlehnung an die Zielsetzung.

> **Hypothese 5.1: UniXcoder eignet sich besser für den Einsatz in FTLR als fastText**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR für ein Softwareprojekt beim Einsatz von UniXcoder als beim Einsatz von fastText.

Wikipedia2Vec erfüllt die gewünschte Eigenschaft, kontextsensitiv zu sein. Trotz eines mangelnden Vortrainings auf Quelltext, liegt dem LM ein im vorliegenden Anwendungsfall sehr interessanter Ansatz zugrunde. Als bedeutungseinbettendes LM verfolgt es einen zu UniXcoder stark verschiedenen und somit geeignet ergänzenden Ansatz. Es erweitert die Funktionalität von fastText um das Bereitstellen von SEs. Damit können Worte und somit Anforderungen und natürlichsprachliche Elemente der Quelltextartefakt-Elemente möglicherweise genauer repräsentiert werden. In einer NLP-Aufgabenstellung zum Finden von Ähnlichkeiten hat es vielversprechende Resultate erzielt (s. Abschnitt 3.2.4). Auf Basis dessen ist davon auszugehen, dass sich Wikipedia2Vec für den Einsatz in FTLR eignet und die Untersuchung von UniXcoder im Rahmen dieser Arbeit geeignet ergänzt. Daraus resultiert Hypothese 5.2.

Hypothese 5.2: Wikipedia2Vec eignet sich besser für den Einsatz in FTLR als fastText

FTLR erzielt bessere Ergebnisse bei der ATLR für ein Softwareprojekt beim Einsatz von Wikipedia2Vec als beim Einsatz von fastText.

Abbildung 5.1 veranschaulicht den modularen Ablauf der Vorkalkulationsphase des FTLR-Verfahrens analog zu Kapitel 4. In dieser werden Artefakt-Elemente durch fastText als BOEs repräsentiert. Zum Einsatz neuer, alternativer LMs anstelle von fastText müssen am Baustein „fastText“ Änderungen vorgenommen werden. Die rote Umrandung dieses Bausteins veranschaulicht, dass die Entwurfsentscheidung, welche LMs zu verwenden sind, bereits getroffen wurde. In Abbildung 5.1 ist ebenfalls zu erkennen, dass vor der Repräsentation der Artefakt-Elemente eine Vorverarbeitung erfolgt. Im Anschluss an die Repräsentation werden die ermittelten BOEs mithilfe der Ähnlichkeitsmetrik WMD verglichen. Welche Vorverarbeitungsschritte bzw. Ähnlichkeitsmetriken sinnvoll sind, hängt vom verwendeten LM ab. Daher müssen beim Einsatz neuer LMs auch diese Bausteine gegebenenfalls verändert werden. Dies wird durch die blaue Umrandung symbolisiert. Mögliche Anpassungen sollen in den folgenden Kapiteln analysiert werden. Weitere Module des FTLR-Verfahrens müssen nicht angepasst werden. Die Bestimmung der feingranularen TLs und die Durchführung des Mehrheitsvotums in der TL-Prozessierungsphase erfolgen unabhängig vom verwendeten LM (s. Kapitel 4).

5.3 Vorverarbeitung

Bevor Anforderungs- und Quelltext-Elemente durch ein LM repräsentiert werden, werden sie zunächst vorverarbeitet. Dies ist ein für NLP-Verfahren typischer Schritt, der beispielsweise auch bei Verfahren wie WELR und S2Trace eingesetzt wird (s. Abschnitte 3.3.1 und 3.3.2). Das Ziel der Vorverarbeitung besteht darin, Artefakte in eine Form zu bringen, die sich für die Weiterverarbeitung durch ein spezifisches LM eignet. Eine Reihe der wichtigsten Vorverarbeitungsschritte wurde bereits in Abschnitt 2.3 vorgestellt. Hierzu zählen im Wesentlichen die folgenden Schritte:

1. Nichtbuchstabenfilterung
2. Querverweisfilterung
3. Binnenmajuskelauftrennung
4. Kleinbuchstaben-Transformation
5. Lemmatisierung
6. Stoppwortentfernung
7. Wortlängenfilterung

FTLR verwendet alle diese sechs Schritte für die Vorverarbeitung von Anforderungs- und Quelltextartefakten beim Einsatz von fastText. Diese Schritte haben sich als sinnvoll für die Vorverarbeitung von Artefakten vor der Repräsentation ihrer Artefakt-Elemente durch fastText erwiesen. In diesem Abschnitt soll analysiert werden, welchen Effekt diese Vorverarbeitungsschritte auf die Weiterverarbeitung mithilfe von LMs besitzen. Auf Basis dieser Analyse soll abgeleitet werden, welche Vorverarbeitungsschritte auch für die zu untersuchenden LMs UniXcoder und Wikipedia2Vec sinnvoll sind. Außerdem soll analysiert werden, welche weiteren Vorverarbeitungsschritte eventuell benötigt werden. Beide zu untersuchenden LMs wurden vortrainiert. Es ist sinnvoll, die Artefakte so vorzuverarbeiten,

dass sie den im Training verwendeten Artefakten ähneln. Denn im Vortraining wird Wissen über Eigenschaften und Strukturen der Sprache der Artefakte antrainiert. Ähneln die Artefakte den Vortrainingsdaten kann dieses Wissen bei der Repräsentation einbezogen werden. Dies kann zu genaueren Repräsentationen führen.

Nichtbuchstaben (z.B. Sonderzeichen und Zahlen) und Querverweise tragen wenig zur Semantik einer Anforderung bzw. einer Methode bei. Die Entfernung derartiger sprachlicher Elemente im Zuge der Nichtbuchstaben- und Querverweisfilterung verringert somit die Zahl einzubettender sprachlicher Einheiten. Die erhöht die zeitliche Performanz von FTLR. Gleichzeitig gehen auch keine für die Semantik eines Artefakt-Elements relevanten sprachlichen Einheiten verloren. Damit wird die semantische Repräsentation des Artefakt-Elements durch den Einsatz dieses Vorverarbeitungsschritts vermutlich nicht ungenauer. Damit ist dieser Schritt sowohl für den Einsatz von fastText als auch für UniXcoder und Wikipedia2Vec sinnvoll.

Bei der Binnenmajuskelauftrennung werden Bezeichnungen in Binnenmajuskelschreibweise in ihre einzelnen Worte aufgeteilt. Beispielsweise wird die Bezeichnung „resultOfCalculation“ in die Worte „result“, „of“, „calculation“ aufgeteilt. In Quelltext wird die Binnenmajuskelschreibweise häufig zur Benennung von Bezeichnern wie Methodennamen oder Attributen eingesetzt. Daher tritt die Binnenmajuskelschreibweise häufig in den zu repräsentierenden Quelltextartefakt-Elementen auf. Natürliche Sprache verwendet in der Regel keine Binnenmajuskelschreibweise. Im Rahmen von Softwareprojekten kann Binnenmajuskelschreibweise jedoch auch in natürlichsprachlichen Artefakten auftreten. Beispielsweise wird in Dokumentationsartefakten häufig auf Quelltext-Methoden oder -Klassen verwiesen, die in Binnenmajuskelschreibweise verfasst sind. Anforderungsartefakte beschreiben das Softwaresystem sehr abstrakt und implementierungsfern. Nur selten wird tatsächlich auf konkrete Implementierungen verwiesen. In diesen Fällen wird jedoch auch in zu repräsentierenden Anforderungen Binnenmajuskelschreibweise verwendet. Damit kann Binnenmajuskelschreibweise sowohl in Anforderungs- als auch in Quelltextartefakt-Elementen auftreten. LMs, die ausschließlich auf natürlichsprachlichen Texten vortrainiert wurden, sind nicht mit der Binnenmajuskelschreibweise von Bezeichnungen vertraut. Für sie ist der Einsatz dieses Vorverarbeitungsschrittes notwendig, um eine Bezeichnung geeignet repräsentieren zu können. Wikipedia2Vec wurde ausschließlich auf natürlicher Sprache vortrainiert. Daher ist es beim Einsatz von Wikipedia2Vec wichtig, den Vorverarbeitungsschritt der Binnenmajuskelauftrennung bei Anforderungen und Quelltext durchzuführen. LMs, die auf Quelltext vortrainiert wurden, erwarten bei der Repräsentation von Quelltext Binnenmajuskelschreibweise. Erfolgt keine Binnenmajuskelauftrennung ähneln die Artefakte stärker den Artefakten im Vortraining. Da UniXcoder auf Quelltext vortrainiert wurde, sollte dieser Vorverarbeitungsschritt bei der Vorverarbeitung von Quelltext nicht durchgeführt werden. Bei Anforderungen sollte die Binnenmajuskelschreibweise ebenfalls nicht aufgetrennt werden. Eine Bezeichnung in Binnenmajuskelschreibweise in einer Anforderung verweist in der Regel auf eine Quelltexteinheit. Dies deutet häufig auf eine logische Verbindung und damit auf eine TL zwischen Anforderungs- und Quelltextartefakt hin. Wird die Bezeichnung in beiden Artefakten nicht aufgetrennt, ähnelt sich die Form der enthaltenen Worte stärker als bei uneinheitlicher Vorgehensweise. Dies kann eine ähnliche Repräsentation der Elemente der Artefakte begünstigen.

Im Zuge der Kleinbuchstaben-Transformation werden alle Buchstaben eines Wortes in ihre Kleinschreibung überführt. Beispielsweise wird das Wort „Calculation“ in „calculation“ transformiert. Anforderungen sind im Anwendungsfall auf Englisch formuliert. In der englischen Sprache werden Worte in der Regel kleingeschrieben. Nur Worte am Satzanfang oder spezielle Bezeichnungen wie „New York“ werden großgeschrieben. Auch Bezeichner im Quelltext sind, wenn nicht in Binnenmajuskelschreibweise, klein geschrieben. Die Bedeutung eines Wortes ist nicht davon abhängig, ob das Wort am Satzanfang positioniert

ist. Stattdessen hängt sie vom vorliegenden Kontext ab. Gleiches gilt auch für spezielle Bezeichnungen wie Eigennamen. Die Bedeutung erschließt sich über den Kontext und Kombinationsworte und nicht über die Groß- oder Kleinschreibung. Kontextsensitive LMs sollten gleiche Worte daher nicht als unterschiedlich betrachten, nur weil sie an manchen Stellen im Text groß- oder kleingeschrieben sind. Daher ist dieser Vorverarbeitungsschritt für UniXcoder und Wikipedia2Vec für Anforderungen und für Quelltext sinnvoll.

Der Vorverarbeitungsschritt der Lemmatisierung überführt jedes Wort in sein zugrunde liegende Lemma. Beispielsweise wird „drove“ in sein Lemma „drive“ überführt. Lemmatisierung eignet sich, wenn unterschiedliche Wortformen eines Wortes unterschiedlich weiterverarbeitet werden sollen. Dies ist vor allem dann der Fall, wenn davon ausgegangen werden kann, dass die Wortform entscheidend zur Bedeutung des Wortes im Kontext beiträgt. UniXcoder nimmt eine eigene, modellspezifische Lemmatisierung vor. Hierbei wird ein Wort in Subworte aufgeteilt. Ein Subwort ist beispielsweise der Wortstamm oder die Wortendung. Eine zusätzliche vorherige Lemmatisierung ist daher nicht nötig. Außerdem ist UniXcoder durch sein Vortraining an unlemmatisierte Eingaben angepasst. Daher wird für UniXcoder der Vorverarbeitungsschritt Lemmatisierung nicht durchgeführt. Für Wikipedia2Vec eignet sich eine Lemmatisierung. Denn Wikipedia2Vec wurde auf lemmatisierten Daten vortrainiert und hat WEs und SEs für die Grundformen von Worten gelernt. Daher wird für Wikipedia2Vec eine Lemmatisierung durchgeführt.

Bei der Stoppwortentfernung werden Worte aus einem Artefakt entfernt, die in der zugrunde liegenden Sprache allgemein sehr häufig auftreten und wenig Informationsgehalt besitzen. Beispiele für Stoppworte sind „the“ und „of“. Es existieren neben allgemeinen Stoppworten auch quelltextspezifische Stoppworte. Zu diesen zählt beispielsweise „get“, welches häufig in Bezeichnern für Methodennamen auftritt. Damit hält „get“ selbst wenig Informationen, die benachbarten Worte dagegen viel. Diese quelltextspezifischen Stoppworte werden bei der Stoppwortentfernung bei Quelltext zusätzlich zu den allgemeinen Stoppworten entfernt. Dies ist sinnvoll beim Einsatz von LMs wie fastText, die Worte einzeln repräsentieren. Stoppworte werden aufgrund ihres häufigen Auftretens ähnlich zu sehr vielen Worten repräsentiert. Hierzu zählen auch Worte, die ihnen semantisch nicht ähneln. Nutzt fastText Stoppworte zur Repräsentation von Artefaktelementen, können daraus Ungenauigkeiten im Ähnlichkeitsvergleich resultieren. Analog lässt sich bei dem bedeutungseinbettenden LM Wikipedia2Vec argumentieren. Wikipedia2Vec sollte ebenfalls nur genau diejenigen Worte zur Repräsentation eines Artefakts nutzen, die seine Bedeutung beeinflussen. Daher wird für Wikipedia2Vec eine Stoppwortentfernung durchgeführt. UniXcoder nutzt zur Repräsentation von sprachlichen Einheiten den gesamten vorliegenden Kontext. Hierzu zählen auch Stoppworte. Durch den Einsatz von Selbstbeobachtungsschichten lernt UniXcoder selbst, welche Worte für den Kontext eines Wortes besonders relevant sind und welche nicht. UniXcoder wurde auf Artefakten trainiert, die Stoppworte enthalten. Daher erwartet UniXcoder Eingabedaten, bei denen keine Stoppwortentfernung durchgeführt wurde. Daher wird für UniXcoder keine Stoppwortentfernung durchgeführt.

Das Herausfiltern von Worten mit einer sehr niedrigen Anzahl an Zeichen wird häufig ergänzend zur Stoppwortentfernung durchgeführt. Worte mit wenig Buchstaben sind in der Regel Abkürzungen oder häufig auftretende Worte mit wenig Semantik wie „if“ oder „to“. Analog zur Stoppwortentfernung kann argumentiert werden, dass dieser Schritt vermutlich für Wikipedia2Vec geeignet und für UniXcoder ungeeignet ist. Bei Wikipedia2Vec entfernt die Wortlängenfilterung semantisch unwesentliche sprachliche Einheiten, deren Einbeziehung die Genauigkeit der Repräsentation der Artefakt-Elemente verschlechtern könnte. UniXcoder hingegen erwartet auch derartige kurze Worte und evaluiert ihre Relevanz für den Kontext durch sein NN.

Zusammenfassend wurde analysiert, dass der Einsatz der Vorverarbeitungsschritte Nicht-

buchstabenfilterung, Querverweisfilterung und Kleinbuchstaben-Transformation vermutlich geeignet für UniXcoder ist. Binnenmajuskelauftrennung, Lemmatisierung, Stoppwortentfernung und Wortlängenfilterung hingegen sollten nicht durchgeführt werden. Daraus resultiert Hypothese 5.3. Für Wikipedia2Vec wird der Einsatz aller zuletzt vorgestellter Vorverarbeitungsschritte benötigt.

> **Hypothese 5.3: UniXcoder Vorverarbeitung Schritte**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR unter Einsatz von UniXcoder bei alleiniger Durchführung der Vorverarbeitungsschritte Nichtbuchstabenfilterung, Querverweisfilterung und Kleinbuchstaben-Transformation als bei zusätzlicher Durchführung von Binnenmajuskelauftrennung, Lemmatisierung, Stoppwortentfernung und Wortlängenfilterung.

Nun soll betrachtet werden, welche zusätzlichen Vorverarbeitungen für UniXcoder bzw. Wikipedia2Vec infrage kommen.

FTLR ermittelt TLs zwischen Anforderungs- und Quelltextartefakten auf Basis von Ähnlichkeiten zwischen feingranularen Artefakt-Elementen (siehe Kapitel 4). Das Herausfiltern der Artefakt-Elemente aus einem Artefakt kann ebenfalls als Teil der Vorverarbeitung betrachtet werden. Eine öffentliche Methode ist ein Artefakt-Element eines Quelltextartefakts. fastText ist aufgrund seines Vortrainings nur für die Repräsentation von natürlicher Sprache geeignet. Daher können beim Einsatz von fastText nur natürlichsprachliche Elemente einer Methode zu ihrer Repräsentation genutzt werden. Da Wikipedia2Vec ebenfalls nur auf natürlichsprachlichen Texten vortrainiert wurde, gilt Selbiges beim Einsatz von Wikipedia2Vec. Eine Methode umfasst neben der Methodensignatur und optional MCs auch einen MB. Dieser ist in einer Programmiersprache und nicht in natürlicher Sprache verfasst. Daher kann er von fastText und vermutlich auch Wikipedia2Vec nicht geeignet in die Repräsentation der Semantik der Methode einbezogen werden. Er enthält jedoch semantische Informationen über die konkrete Implementierung der Methode. Damit kann sich seine Einbeziehung positiv auf die Genauigkeit der Repräsentation der Methode auswirken. UniXcoder ist auf Quelltext vortrainiert und ist daher in der Lage, auch den MB in die Repräsentation einfließen zu lassen. Außerdem wurde UniXcoder auf vollständigen Methoden in den Trainings-Artefakten vortrainiert. Eine Eingabe bestehend aus der vollständigen Methode einschließlich des MBs ähnelt seinen Vortrainingsdaten. Es resultiert Hypothese 5.4:

> **Hypothese 5.4: UniXcoder Vorverarbeitung Umfang der Eingabe**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR unter Einsatz von UniXcoder, wenn auch der MB in die Repräsentation eines Quelltextartefakt-Elements einbezogen wird, als wenn er nicht bezogen wird.

Um Worte durch kontextsensitive SEs zu repräsentieren, benötigt Wikipedia2Vec ihre Wortbedeutung als Eingabe. Die Bestimmung der Wortbedeutung muss demnach im Vorfeld des eigentlichen LM-Einsatzes geschehen. Daher ist die Durchführung einer WSD (s. Abschnitt 2.6.3) für jedes Wort ein notwendiger Teil der Vorverarbeitung für den Einsatz von Wikipedia2Vec. Diese WSD kann manuell oder automatisiert erfolgen. Bei manueller Vorgehensweise entsteht ein hoher zeitlicher Aufwand für die Entwickler, die eine ATLR mittels FTLR und Wikipedia2Vec durchführen möchten. Außerdem wäre die Rückverfolgbarkeitsanalyse durch den manuellen Aufwand streng genommen nicht mehr vollautomatisiert. Eine automatisierte Kontextbestimmung reduziert den zeitlichen Aufwand

deutlich. Es ist jedoch davon auszugehen, dass mit einer Automatisierung auch eine Verschlechterung der Genauigkeit der ermittelten Wortbedeutung im Vergleich zur manuellen Durchführung einhergeht. Ungenauigkeiten bei der WSD führen dazu, dass Worte durch eine ungenaue SE repräsentiert werden könnten. Dies kann zu Ungenauigkeiten bei der Repräsentation von Artefakt-Elementen führen. Aus dieser Annahme resultiert Hypothese 5.5.

> **Hypothese 5.5: Wikipedia2Vec Vorverarbeitung Kontextbestimmung**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR unter Einsatz von Wikipedia2Vec, wenn die WSD der Artefaktworte manuell erfolgt als wenn sie automatisiert erfolgt.

5.4 Abbildungsverfahren und Ähnlichkeitsvergleich

LMs repräsentieren sprachliche Einheiten. Dies kann als Abbildung $f : X \rightarrow Y$ von einer sprachlichen Einheit X auf ihre Repräsentation Y aufgefasst werden. Diese Abbildung soll im Folgenden für die zu untersuchenden LMs UniXcoder und Wikipedia2Vec analysiert werden. Darauf aufbauend sollen Möglichkeiten ermittelt werden, die von den LMs erzeugten Repräsentationen Y auf Ähnlichkeit zu vergleichen.

UniXcoder erhält eine Folge von Token als Eingabe (s. Abschnitt 3.2.2). Bei diesen Token kann es sich um natürlichsprachliche Worte oder Quelltext-Bezeichner handeln. Diese Wortfolge wird durch einen *CLS*-Token ergänzt, der der Tokenfolge vorangestellt wird. Dieser symbolisiert die Semantik der gesamten Tokenfolge. UniXcoder lernt anschließend eine kontextsensitive Repräsentation für jedes Wort. In diesem Zuge wird auch eine Repräsentation des CLS-Token mitgelernt. Im Zuge des Lernprozesses aktualisiert UniXcoder die vortrainierten Gewichtungen der Recheneinheiten seines NNs. UniXcoder liefert für jeden Token die gelernten Gewichte der letzten transparenten Schicht seines NN als Ausgabe. Diese Gewichte lassen sich als Vektor auffassen und bilden die kontextsensitiven Repräsentationen der Tokens. Die Repräsentation des CLS-Tokens entspricht der Repräsentation der gesamten Tokenfolge. Zusammengefasst handelt es sich bei der sprachlichen Einheit X, die UniXcoder auf ein Y abbildet, um eine Tokenfolge. Bei der Repräsentation Y handelt es sich um eine Menge von Vektoren, von denen jeder jeweils einen Token der Tokenfolge X repräsentiert (BOE). Darüber hinaus liefert UniXcoder einen Vektor, der die Semantik der gesamten Tokenfolge X repräsentiert.

Wikipedia2Vec erhält entweder ein Wort als Eingabe, falls zu diesem kein Wikipedia-Artikel existiert, oder eine Wortbedeutung. Für eine gegebene Eingabe, gibt Wikipedia2Vec einen Vektor zurück, der die Eingabe repräsentiert. Bei der Eingabe eines Wortes ist dies eine statische WE. Bei der Eingabe einer Wortbedeutung wird eine kontextsensitive SE zurückgegeben. Zusammengefasst handelt es sich bei der sprachlichen Einheit X, die Wikipedia2Vec auf ein Y abbildet, um ein Wort oder eine Wortbedeutung. Bei der Ausgabe Y handelt es sich um einen Vektor, der das Wort bzw. die Wortbedeutung repräsentiert.

Bei einem zu repräsentierenden Artefakt-Element handelt es sich um eine Folge von Worten. UniXcoder und Wikipedia2Vec können für diese Folge von Worten eine BOE liefern. Diese enthält in beiden Fällen Vektoren, die die einzelnen Worte kontextsensitiv repräsentieren Durch die separate Repräsentation der einzelnen Worte wird die Semantik eines jeden Wortes einzeln erfasst. Es bestehen mehrere grundlegende Möglichkeiten, zwei BOEs auf Ähnlichkeit zu vergleichen. Zum einen können die Vektoren der BOEs aggregiert werden. Beispielsweise können zuerst die Durchschnittsvektoren der BOEs berechnet werden. Auf diese resultierenden Vektoren kann dann ein Ähnlichkeitsmaß des VSM (s. Abschnitt

2.2.1) angewendet werden. Die euklidische Distanz oder die Kosinusähnlichkeit stellen derartige Ähnlichkeitsmaße dar (s. Abschnitt 2.2.1.3). Die euklidische Distanz ermittelt Werte, deren Wertebereich unbegrenzt ist. Zur Anwendung eines Schwellenwerts der Ähnlichkeit (s. Kapitel 4) müssen die resultierenden Werte jedoch in einem festgelegten Wertebereich liegen. Die Kosinusähnlichkeit ist daher ein geeigneteres Maß der Ähnlichkeit. Hier wird der Winkel zwischen zwei Vektoren betrachtet. Daher ist der Wertebereich der Ausgabe begrenzt. Eine weitere Möglichkeit besteht in der Berechnung der WMD zwischen den Vektoren der BOEs (s. Abschnitt 2.2.1.3). Hierbei wird die kleinste Distanz berechnet, die nötig ist, um alle Vektoren einer BOE auf einen beliebigen Vektor der anderen BOE abzubilden. Aufgrund einer vorherigen Normierung der Vektoren, ermittelt die WMD Werte in einem festen Wertebereich. Die WMD ist für den Einsatz von Wikipedia2Vec vermutlich geeigneter als die erste Möglichkeit der Vektor-Aggregation und anschließender Berechnung der Kosinusähnlichkeit. Denn durch die Aggregation der Vektoren gehen Informationen über die Vektoren verloren. Damit gehen Informationen über die Semantik der Worte verloren. Bei der WMD wird jedes Paar an Vektoren der BOE einzeln betrachtet. Damit werden alle semantischen Informationen über die repräsentierten Worte in den Ähnlichkeitsvergleich einbezogen. Außerdem wird die WMD auch beim Einsatz von fastText in FTLR dem Vergleich einer Vektor-Aggregation vorgezogen.

> **Hypothese 5.6: Wikipedia2Vec Ähnlichkeitsvergleich**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR beim Einsatz von Wikipedia2Vec unter Verwendung der WMD für den Ähnlichkeitsvergleich als unter Verwendung der Kosinusähnlichkeit nach Vektordurchschnittsbildung.

Da auch UniXcoder eine BOE liefert, kann die WMD-Metrik auch im Fall von UniXcoder zum Ähnlichkeitsvergleich eingesetzt werden. UniXcoder ermittelt jedoch zusätzlich zu einer BOE auch einen einzelnen Vektor zur Repräsentation eines gesamten Artefakt-Elements. Dieser bezieht zusätzlich zu den semantischen Informationen der Worte auch die Semantik der Wortreihenfolge in die Repräsentation eines Artefakt-Elements ein. Damit ist zu erwarten, dass dieser Vektor die Semantik von Artefakt-Elementen genauer beschreibt als die BOE. Damit ist ein Vergleich basierend auf diesen Vektoren vermutlich geeigneter als ein Vergleich basierend auf der BOE. Zum Vergleich zweier derartiger Vektoren kann die zuvor beschriebene Kosinusähnlichkeit angewendet werden.

> **Hypothese 5.7: UniXcoder Ähnlichkeitsvergleich**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR beim Einsatz von UniXcoder unter Verwendung der Kosinusähnlichkeit für den Ähnlichkeitsvergleich als unter Verwendung der WMD.

5.5 Zusammenfassung des Entwurfs

Im Rahmen dieser Arbeit sollen LMs in FTLR integriert werden. Diese LMs besitzen Eigenschaften, die annehmen lassen, dass die LMsfastText's Schwachstellen nicht besitzen. fastText's Schwachstellen bestehen darin, dass das LM keine kontextsensitiven WEs ermittelt und nicht auf Quelltext vortrainiert wurde. Es wird angenommen, dass alternative LMs ohne diese Schwachstellen Anforderungs- und Quelltextartefakt-Elemente genauer repräsentieren können als fastText. Aus diesem Grund sind derartige LMs vermutlich geeigneter für den Einsatz in FTLR als fastText In Abschnitt 5.2 wurde die Entwurfsentscheidung getroffen, die LMs UniXcoder und Wikipedia2Vec zu untersuchen (s. Abschnitt 5.2). Bei-

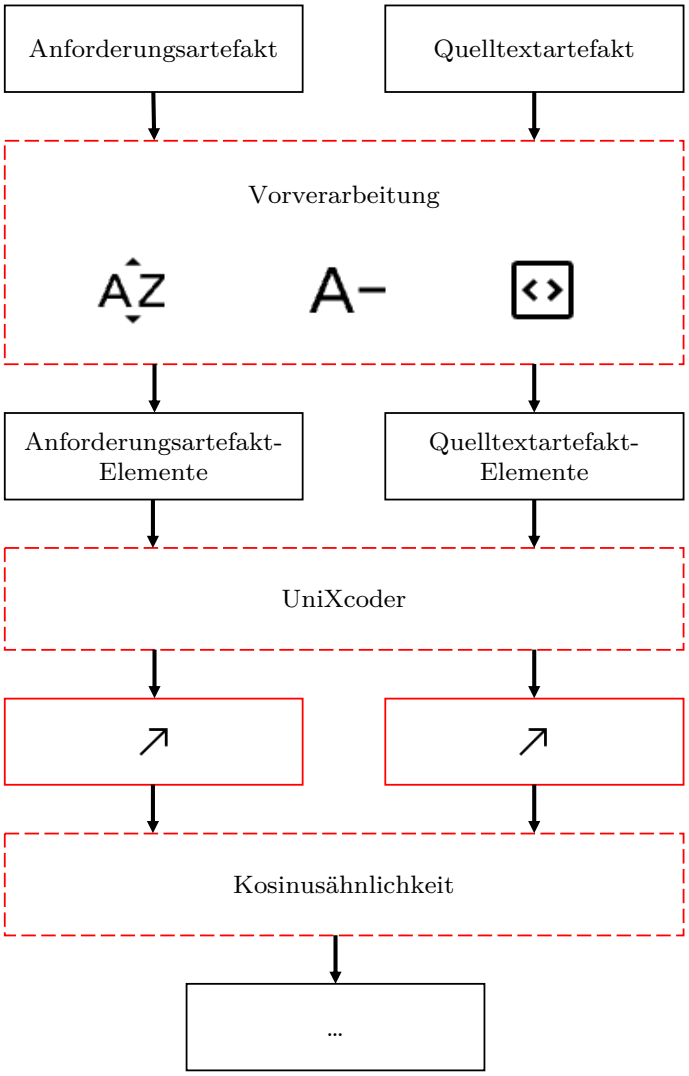


Abbildung 5.2: Entwurf zum Einsatz von UniXcoder in FTLR

de LMs besitzen bestimmte Schwachstellen von fastText nicht und fügen vermutlich keine weiteren hinzu. Daher wird erwartet, dass ihr Einsatz zu besseren Ergebnissen von FTLR führt als der Einsatz von fastText. Zur Integration von UniXcoder und Wikipedia2Vec sind Anpassungen an verschiedenen Stellen der Funktionsweise von FTLR notwendig. Neben des alleinigen Austauschs des LMs soll auch die verwendete Vorverarbeitung und der eingesetzte Ähnlichkeitsvergleich auf das jeweilige LM abgestimmt werden.

Abbildung 5.2 visualisiert den Entwurf zum Einsatz von UniXcoder in FTLR. Zunächst wird eine Vorverarbeitung der Anforderungs- und Quelltextartefakte durchgeführt (s. Abschnitt 5.3). Hierbei werden als erstes alle Nichtbuchstaben und Querverweise entfernt (AẐ). Anschließend werden alle Worte der Artefakte in ihre Kleinschreibweise transformiert (A–). Auf diese Weise ähnelt die Form der vorverarbeiteten Artefakte der Form der Vortrainingsdaten von UniXcoder. Es wird erwartet, dass UniXcoder die Artefakte dann besonders geeignet weiterverarbeiten kann. Im letzten Schritt der Vorverarbeitung werden alle Artefakte in Artefakt-Elemente aufgeteilt. Hierbei werden Quelltextartefakt-Elemente um den MB ergänzt (‹›). UniXcoder ist in der Lage, diesen in die Repräsentation eines Quelltextartefakt-Elements einzubeziehen. Ein Einbeziehen könnte die Genauigkeit der Repräsentation der Quelltextartefakt-Elemente verbessern. Nach der Vorverarbeitung werden die resultieren Artefakt-Elemente im nächsten Schritt durch UniXcoder repräsentiert

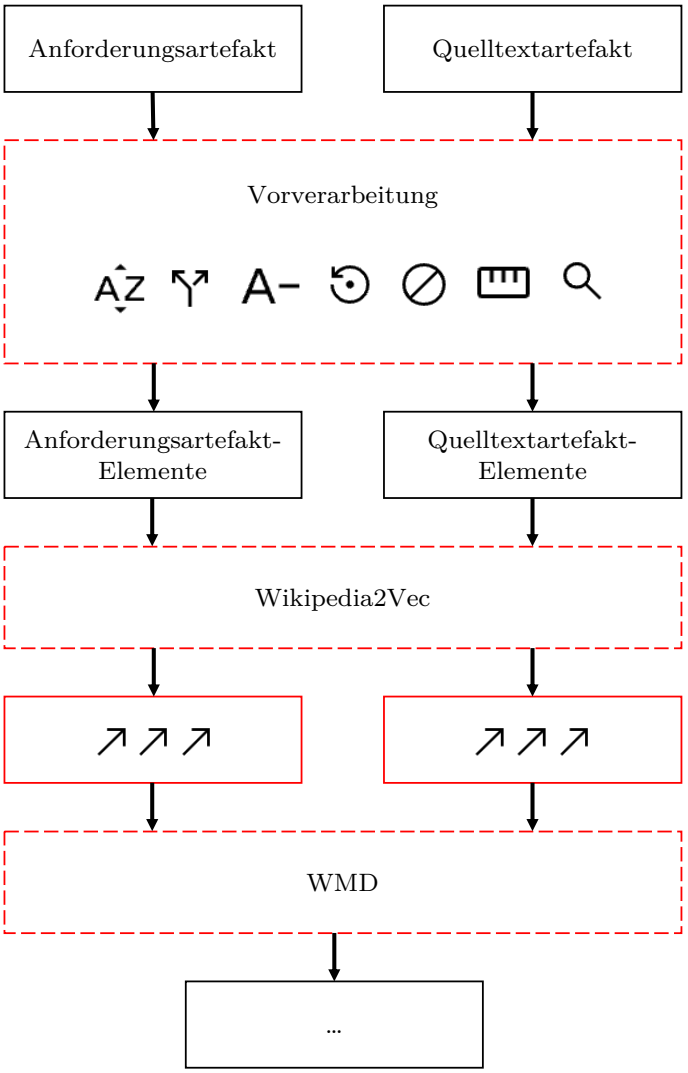


Abbildung 5.3: Entwurf zum Einsatz von Wikipedia2Vec in FTLR

(s. Abbildung 5.2) UniXcoder repräsentiert die Semantik eines jeden Artefakt-Elements durch einen einzelnen Vektor. UniXcoder lernt diesen Vektor während der Repräsentation der Semantik der einzelnen Worte eines Artefakt-Elements mit. Dadurch wird der Vektor in Abhängigkeit der Semantik aller Worte bestimmt. UniXcoder repräsentiert die einzelnen Worte nacheinander in der vorliegenden Wortreihenfolge. Ihre Semantik fließt somit auch nacheinander in die Vektor-Repräsentation der gesamten Wortfolge ein. Dadurch wird die Vektor-Repräsentation auch durch die Wortreihenfolge beeinflusst. Die Vektorrepräsentation repräsentiert somit neben der Semantik der Worte auch die Semantik, die durch die Wortreihenfolge bestimmt wird. Es wird angenommen, dass sich diese Repräsentation durch UniXcoder daher für den vorliegenden Anwendungsfall besonders eignet (s. Abschnitt 5.4). Im Anschluss an die Repräsentation der Artefakt-Elemente werden die ermittelten Repräsentationen auf Ähnlichkeit verglichen. Im Fall der Integration von UniXcoder erfolgt dieser Ähnlichkeitsvergleich mittels der Kosinusähnlichkeit (s. Abschnitt 5.4). Hierbei wird der Winkel zwischen den beiden Vektorrepräsentation als Ähnlichkeit der repräsentierten Artefakt-Elemente berechnet. Damit liegen die resultierenden Werte in einem festen Wertebereich, was sich für das spätere Ermitteln von TLs besonders gut eignet.

Abbildung 5.3 veranschaulicht den Entwurf zum Einsatz von Wikipedia2Vec in FTLR.

Im Zuge der Vorverarbeitung der Anforderungs- und Quelltextartefakte werden analog zu UniXcoder zunächst alle Nichtbuchstaben und Querverweise entfernt (s. Abschnitt 5.3). Anschließend wird die Binnenmajuskelschreibweise von Quelltext-Bezeichnern aufgetrennt (). Es folgt eine Transformation aller Worte in Kleinbuchstaben, sowie eine Lemmatisierung (), Stoppwortentfernung () und Wortlängenfilterung (). Das Ziel dieser Vorverarbeitung besteht darin, dass Artefakt-Elemente im Anschluss nur noch Semantik-tragende Worte enthalten. Außerdem sollen diese relevanten Worte in einer für Wikipedia2Vec geeigneten Form vorliegen. Beispielsweise ist eine Auftrennung der Binnenmajuskelschreibweise nötig, damit Wikipedia2Vec die einzelnen in der Schreibweise zusammengeführten Worte repräsentieren kann. Im letzten Schritt der Vorverarbeitung wird für alle relevanten Worte der Artefakte eine WSD durchgeführt () Hierbei wird für jedes Wort automatisiert bzw. manuell seine Wortbedeutung im vorliegenden Kontext ermittelt. Für diese kann Wikipedia2Vec dann eine SE ermitteln. Worte werden dann kontextsensitiv durch die SE ihrer Wortbedeutung repräsentiert. Durch die Einbeziehung des Kontexts in die Wortrepräsentation wird erwartet, dass die resultieren Repräsentation genauer sind als statische WEs. Im letzten Teil der Vorverarbeitung werden die Artefakte in ihre Artefakt-Elemente aufgeteilt. Im Anschluss an die Vorverarbeitung repräsentiert Wikipedia2Vec diese resultierenden Artefakt-Elemente. Hierbei repräsentiert Wikipedia2Vec zunächst jedes Wort der Artefakt-Elemente einzeln. Konnte im Zuge der Vorverarbeitung eine Wortbedeutung aufgelöst wird, wird ein Wort durch seine SE repräsentiert. Konnte keine Wortbedeutung ermittelt werden, wird auf eine zusätzlich vortrainierte statische WE zurückgegriffen (s. Abschnitt 3.2.4). Ein Artefakt-Element, welches aus einer Menge von Worten besteht, wird dann durch eine Menge von Repräsentation bzw. Vektoren (BOE) dargestellt. Im nächsten Schritt werden diese BOEs paarweise auf Ähnlichkeit verglichen. Hierfür wird im Fall von Wikipedia2Vec die Metrik WMD verwendet (s. Kapitel 4). Diese ermittelt für jeden Vektor einer BOE die kürzeste Distanz zu einem Vektor einer zweiten BOE. Die aggregierte Distanz aller Vektoren einer BOE zu den Vektoren einer anderen BOE ist dann die WMD-Ähnlichkeit der BOEs. Hierbei wird jeder Vektor und somit auch die durch ihn repräsentierte Wortsemantik einzeln unaggregiert in den Ähnlichkeitsvergleich einbezogen. Dies ist laut Abschnitt 5.4 sehr geeignet für den Einsatz von Wikipedia2Vec.

Auf Basis der ermittelten Ähnlichkeiten zwischen Artefakt-Elementen bestimmt FTLR feingranulare TLs zwischen Artefakt-Elementen (s. Kapitel 4). Diese werden anschließend zu TLs auf Artefakt-Ebene aggregiert. Diese Elemente der TL-Prozessierungsphase ist unabhängig vom verwendeten LM. Daher muss sie im Rahmen dieser Arbeit nicht angepasst werden.

6 Implementierung

Dieses Kapitel beschreibt die Implementierung des in Abschnitt 5.5 beschriebenen Entwurfs. Bei FTLR handelt es sich um ein bereits vollumfänglich implementiertes Verfahren zur ATLR. Zur Verwendung alternativer LMs anstelle von fastText sind somit nur punktuelle Erweiterungen bzw. Anpassungen nötig. Die Beschreibung der Implementierung dieser Anpassungen erfolgt in drei Schritten. Zunächst wird veranschaulicht, wie die Vorverarbeitung für den jeweiligen Einsatz von UniXcoder und Wikipedia2Vec umgesetzt wurde. Anschließend wird auf die konkrete Einbindung von UniXcoder und Wikipedia2Vec eingegangen. Es folgt die Beschreibung der Implementierung des jeweils verwendeten Ähnlichkeitsvergleichs. Für jeden dieser drei Teilschritte wird zunächst die existierende Rahmenarchitektur betrachtet, die FTLR zur bisherigen Umsetzung mit fastText anbietet. Daraufhin wird aufgezeigt, an welchen Stellen der Implementierung Änderungen nötig sind, um den Entwurf für die neuen LMs umzusetzen. Anschließend werden diese Änderungen näher beschrieben.

6.1 Vorverarbeitung

Im ersten Schritt der ATLR führt FTLR eine Vorverarbeitung der gegebenen Anforderungs- und Quelltextartefakte durch. Abbildung 6.1 stellt die Rahmenarchitektur der bisherigen Umsetzung der Vorverarbeitung für fastText dar.

Die Vorverarbeitung beginnt damit, dass der textuelle Inhalt der Artefakte im Zuge einer Wortsegmentierung in einzelne Worte aufgetrennt wird (s. Abschnitt 2.3.1). Anschließend werden die Worte eines Artefakts zu Artefakt-Elementen zusammengefasst. Die Unterteilung von natürlichsprachlichen Anforderungsartefakten in Worte und Artefakt-Elemente wird in der Klasse `NaturalSpeechTokenizer` durchgeführt. Diese Klasse verwendet Funktionalitäten der `nltk`-Bibliothek [BKL09]. Nach der Auftrennung in Worte fasst `NaturalSpeechTokenizer` alle Worte eines Anforderungssatzes in einem Artefakt-Element zusammen. Ein Artefakt-Element ist damit eine Folge an Worten eines Anforderungssatzes. Anforderungsartefakte können optional auch UCTs enthalten (s. Kapitel 4). UCTs umfassen u.a. eine Bezeichnung und Beschreibung des Anwendungsfalls, Vorbedingungen und Nachbedingungen sowie den eigentlichen Ablauf des Anwendungsfalls. Ein `NaturalSpeechTokenizer` trennt alle Worte eines UCTs auf. Alle Worte eines UCT-Elements (z.B. der Bezeichnung) werden daraufhin in einem Artefakt-Element zusammengefasst. Der Ablauf eines Anwendungsfalls stellt hierbei einen Sonderfall dar. Dieser ist in der

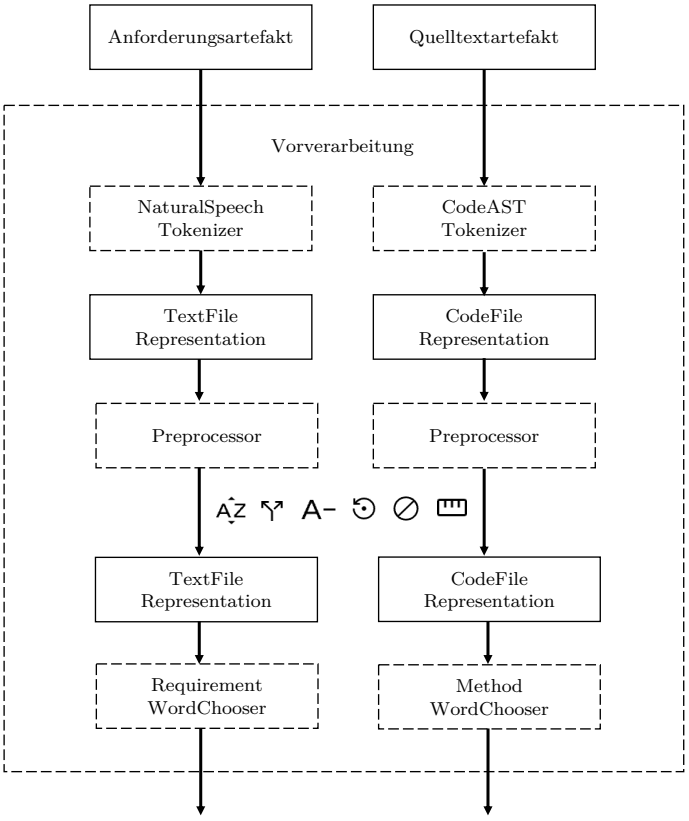


Abbildung 6.1: Rahmenarchitektur Vorverarbeitung fastText

Regel in mehreren Sätzen beschrieben. Analog zu Anforderungssätzen werden alle Worte eines UCT-Ablauf-Satzes in einem eigenständigen Artefakt-Element zusammengefasst. Die Menge der resultierenden Artefakt-Elemente eines Anforderungsartefakts wird in einem Objekt der Klasse `TextFileRepresentation` gespeichert. Für die Unterteilung von Quelltextartefakten ist die Klasse `CodeASTTokenizer` zuständig. Diese teilt unter Einsatz der Bibliotheken `javalang` [c2n22] und `pycparser` [Eli22] Quelltextartefakte in ihre enthaltenen Token (z.B. Variablen) und Strukturen (z.B. Verzweigungen) auf. Diese Token und Strukturen werden anschließend in einen abstrakten Syntaxbaum überführt. Aus diesem Syntaxbaum werden daraufhin alle öffentlichen Methoden, einschließlich ihrer MCs und des MBs als Quelltextartefakt-Element extrahiert. Ein Quelltextartefakt-Element ist damit eine Folge der in einer öffentlichen Methode enthaltenen Token. Ein Objekt der Klasse `CodeFileRepresentation` speichert alle Artefakt-Elemente eines Quelltextartefakts.

Im nächsten Teil der Vorverarbeitung werden die Artefakt-Elemente in mehreren Schritten auf die Weiterverarbeitung durch ein LM vorbereitet. Um fastText geeignet einsetzen zu können, führt FTLR bislang alle in Abschnitt 5.3 beschriebenen Vorverarbeitungsschritte durch. Zuerst werden alle Nichtbuchstaben und Querverweise in den Artefakt-Elementen entfernt. Dies wird mittels der Klassen `NonLetterFilter` und `UrlRemover` umgesetzt. Es folgt eine Auftrennung der Binnenmajuskelschreibweise (`CamelCaseSplitter`) sowie eine Transformation aller Worte in ihre Kleinschreibweise (`LowerCaseTransformer`). Für die anschließende Lemmatisierung (`Lemmatizer`) und Stoppwortentfernung (`StopWordRemover` und `JavaCodeStopWordRemover`) werden erneut Funktionalitäten der `nltk`-Bibliothek verwendet. Zuletzt werden mittels der Wortlängenfilterung in Klasse `WordLengthFilter` alle Worte mit einer Länge von unter drei Buchstaben entfernt. Jede der Klassen, die einen Vorverarbeitungsschritt umsetzt, erbt von `PreprocessingStep`. Zur Ausführung der Schritte wird eine Instanz einer jeden einzusetzenden Vorverarbeitungsschritt-Klasse

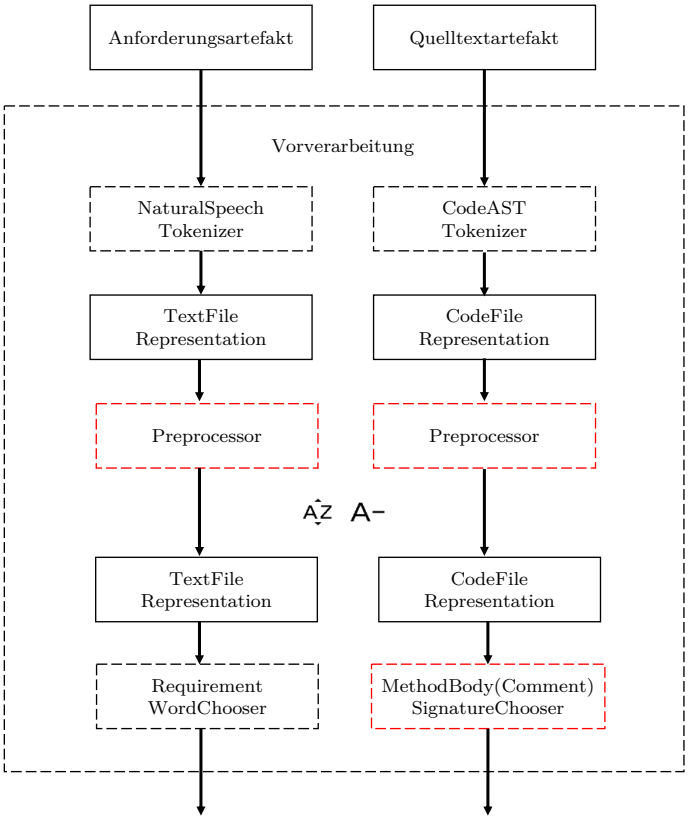


Abbildung 6.2: Architektur Vorverarbeitung UniXcoder

erzeugt. Im Fall des Einsatzes von fastText sind dies alle eben genannten Klassen. Diese Instanzen werden dann als Konfiguration der Klasse `Preprocessor` übergeben. Abbildung 6.1 veranschaulicht, dass der `Preprocessor` für Anforderungsartefakt-Elemente und Quelltextartefakt-Elemente separat konfiguriert werden kann. Die Klasse `Preprocessor` führt für alle Artefakt-Elemente in einer `TextFileRepresentation` bzw. `CodeFileRepresentation` alle konfigurierten Vorverarbeitungsschritte nacheinander durch. Im letzten Schritt bevor Artefakt-Elemente durch ein LM repräsentiert werden, erfolgt die Auswahl der Worte eines Artefakt-Elements, die zur Repräsentation verwendet werden sollen. Diese Aufgabe erfüllen Unterklassen der Klasse `RequirementWordChooser` für Anforderungsartefakt-Elemente und Unterklassen der Klasse `MethodWordChooser` für Quelltextartefakt-Elemente. Für Anforderungsartefakte werden im Fall von fastText alle Worte eines Artefakt-Elements einbezogen. Für Quelltextartefakte werden bislang nur die Worte der Methodensignatur, des Klassennamens und optional der MCs ausgewählt.

Nach der vorangegangenen Beschreibung der existierenden Rahmenarchitektur von FTLR zur Vorverarbeitung werden nun die für den Einsatz von UniXcoder nötigen Anpassungen erläutert. Diese werden in Abbildung 6.2 veranschaulicht. Die bisherige Funktionalität zur Aufteilung eines Artefakts in Worte und Artefakt-Elemente wurde nicht angepasst. Für den Einsatz von UniXcoder sollen ausschließlich die Vorverarbeitungsschritte Nichtbuchstabenfilterung, Querverweisfilterung und Kleinbuchstaben-Transformation durchgeführt werden (s. Abschnitt 5.3). Daher wird der jeweilige `Preprocessor` für Anforderungs- und Quelltextartefakte nur mit Instanzen der Klassen `NonLetterFilter`, `UrlRemover` und`LowerCaseTransformer` konfiguriert. Außerdem sollen beim Einsatz von UniXcoder die Worte des MBs in die Quelltextartefakt-Element-Repräsentation einbezogen werden. Daher wurden der bisherigen Implementierung die Klassen `MethodBodySignatureChooser` und `MethodBodyCommentSignatureChooser` hinzugefügt. Diese wählen zusätzlich zu den

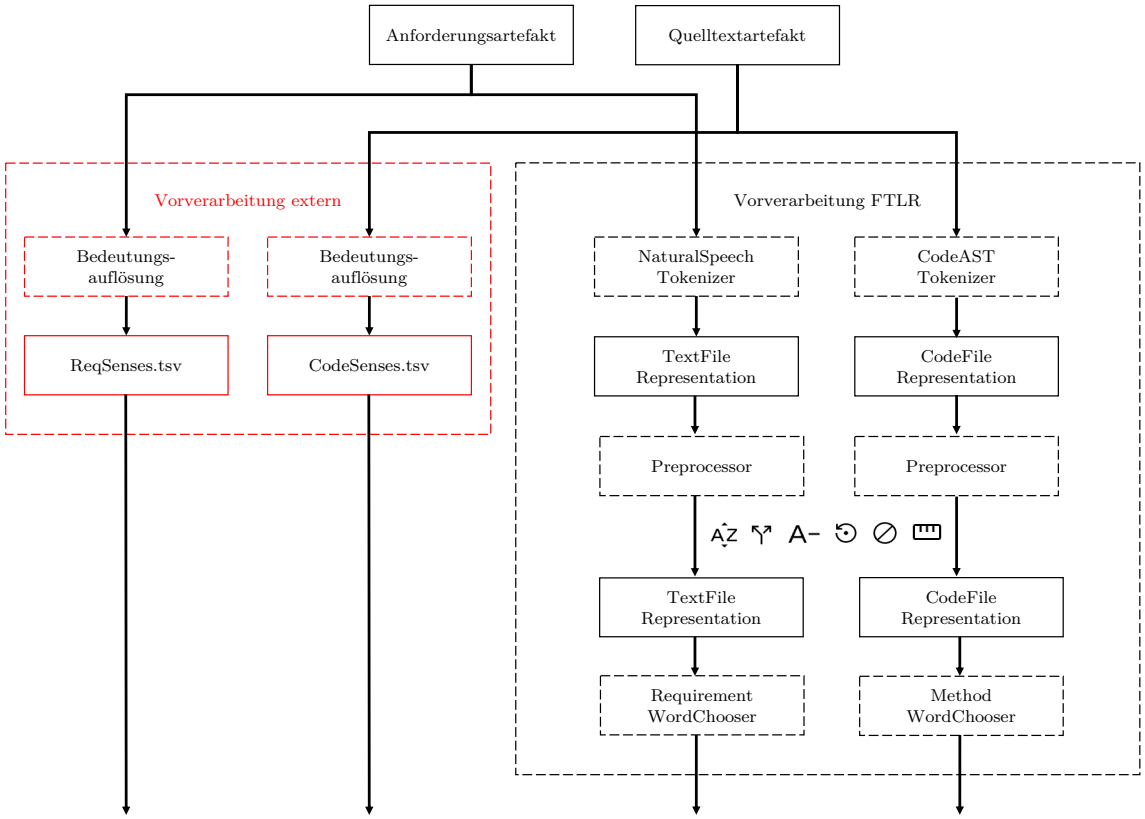


Abbildung 6.3: Architektur Vorverarbeitung Wikipedia2Vec

Worten der Methodensignatur, des Klassennamens und optional der MCs auch die Worte des MBs für die Artefakt-Element-Repräsentation aus.

Abbildung 6.3 visualisiert die für den Einsatz von Wikipedia2Vec nötigen Anpassungen der Vorverarbeitung. Auch hier wurde die Artefakt-Aufteilung analog zu UniXcoder nicht angepasst. In Abschnitt 5.3 wurde analysiert, dass für Wikipedia2Vec dieselben Vorverarbeitungsschritte wie für fastText sinnvoll sind. Außerdem eignet sich für Wikipedia2Vec vermutlich derselbe Wortumfang zur Repräsentation eines Artefakt-Elements wie für fastText. Dies bezieht sich insbesondere darauf, dass das Einbeziehen von MBs beim Einsatz von Wikipedia2Vec vermutlich nicht geeignet ist. Aus diesem Grund wurde für Wikipedia2Vec weder die Konfiguration der `Preprocessor`-Instanzen noch die Auswahl der Worte für die anschließende Artefakt-Element-Repräsentation angepasst. Wikipedia2Vec repräsentiert Artefakt-Worte, deren Bedeutungen in Wikipedia gespeichert sind, durch SEs. Hierfür benötigt Wikipedia2Vec die aufgelösten Wortbedeutungen der Worte als Eingabe (s. Abschnitt 5.3). Daher wird die bisherige Vorverarbeitung von FTLR für Wikipedia2Vec durch einen Schritt ergänzt, in dem eine WSD der Artefakt-Worte durchgeführt wird. Diese WSD erfolgt manuell bzw. durch ein Verfahren, das von `INDIRECT` [Hey19] bereitgestellt wird. Die WSD der Artefakt-Worte wird im Vorfeld der Ausführung von FTLR durchgeführt. Bevor die Artefakte in FTLR eingegeben werden, werden also bereits alle Wortbedeutungen der enthaltenen Worte ermittelt. Die Worte der Artefakte und ihre Wortbedeutungen im jeweiligen Kontext werden im Anschluss an die WSD in einer `.tsv`-Datei gespeichert. Die WSD erfolgt separat für Anforderungs- und Quelltextartefakte. Damit resultieren aus diesem Vorverarbeitungsteil zwei `.tsv`-Dateien. Diese werden FTLR übergeben. Sie haben keinen Einfluss auf die weitere Vorverarbeitung. Im Zuge der Repräsentation der Artefakt-Elemente werden die Wortbedeutungen der Artefakt-Element-Worte eingelesen und durch Wikipedia2Vec repräsentiert.

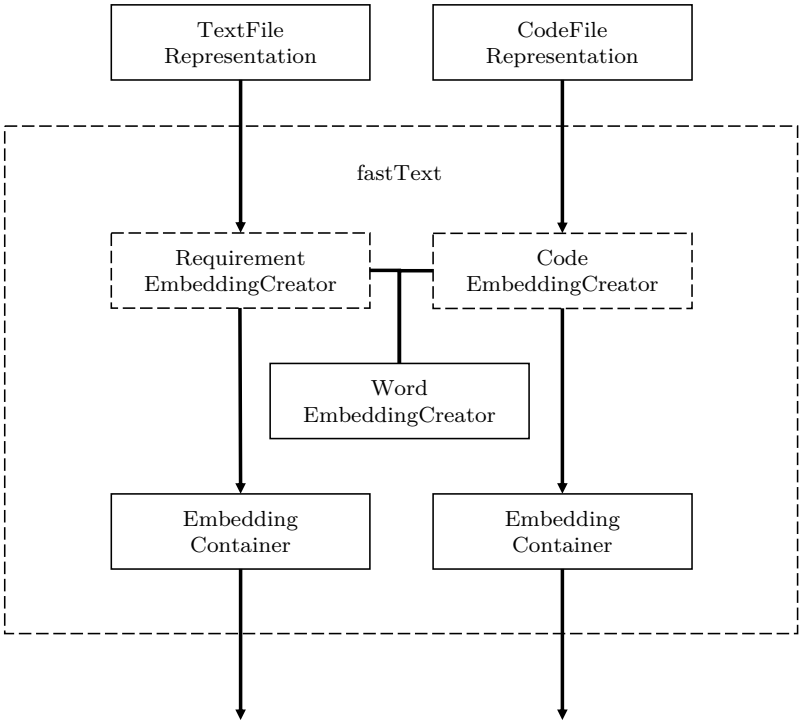


Abbildung 6.4: Rahmenarchitektur Sprachmodell fastText

6.2 Sprachmodell

Im Anschluss an die Vorverarbeitung werden Artefakt-Elemente durch ein LM repräsentiert. Dies erfolgt in der bisherigen Rahmenarchitektur durch Unterklassen von `EmbeddingCreator` (s. Abbildung 6.4). Ein `EmbeddingCreator` definiert eine Schablonenmethode `create_embeddings`. In Unterklassen von `EmbeddingCreator` wird in dieser Methode die Repräsentation von Artefakt-Elementen implementiert.

Die Klasse `RequirementEmbeddingCreator` ist eine solche Unterklasse von `EmbeddingCreator`. Sie definiert eine `create_embeddings`-Methode, die ein im Zuge der Vorverarbeitung ermitteltes Objekt der Klasse `TextFileRepresentation` als Eingabe (s. Abschnitt 6.1) entgegen nimmt. Dieses Objekt enthält alle vorverarbeiteten Artefakt-Elemente eines Anforderungsartefakts. Die `create_embeddings`-Methode der Klasse `CodeEmbeddingCreator`, ebenfalls eine Unterklasse von `EmbeddingCreator`, erhält eine `CodeFileRepresentation` als Eingabe. Diese umfasst alle vorverarbeiteten Artefakt-Elemente eines Quelltextartefakts. Die `create_embeddings`-Methode beider `EmbeddingCreator`-Instanzen repräsentiert jedes übergebene Artefakt-Element. In die Repräsentation fließen alle relevanten Worte eines Artefakt-Elements ein. Im Rahmen der Vorverarbeitung wurde für jedes Artefakt-Element definiert, welche Worte relevant sind und in die Repräsentation einfließen (s. Abschnitt 6.1). Beispielsweise wählt der `MethodBodySignatureChooser` alle Worte der Methodensignatur, des Klassennamens und des MBs zur Repräsentation eines Quelltextartefakt-Elements aus. Zur Repräsentation eines Artefakt-Elements bzw. seiner relevanten Wortfolge nutzt `create_embeddings` Funktionalitäten, die durch eine Unterklasse der Klasse `WordEmbeddingCreator` bereitgestellt werden. In der bisherigen Umsetzung wird in der verwendeten Unterklasse `FastTextEmbeddingCreator` die Methode `create_word_embedding` definiert. Diese liefert für ein übergebenes Wort eine mithilfe eines fastText-LMs ermittelte Vektorrepräsentation dieses Worts. Damit kann die Methode `create_embeddings` jedes einzelne relevante Wort durch einen Vektor repräsentieren. Die Repräsentation eines Artefakt-Elements ist dann eine BOE, also die Menge aller Vektor-

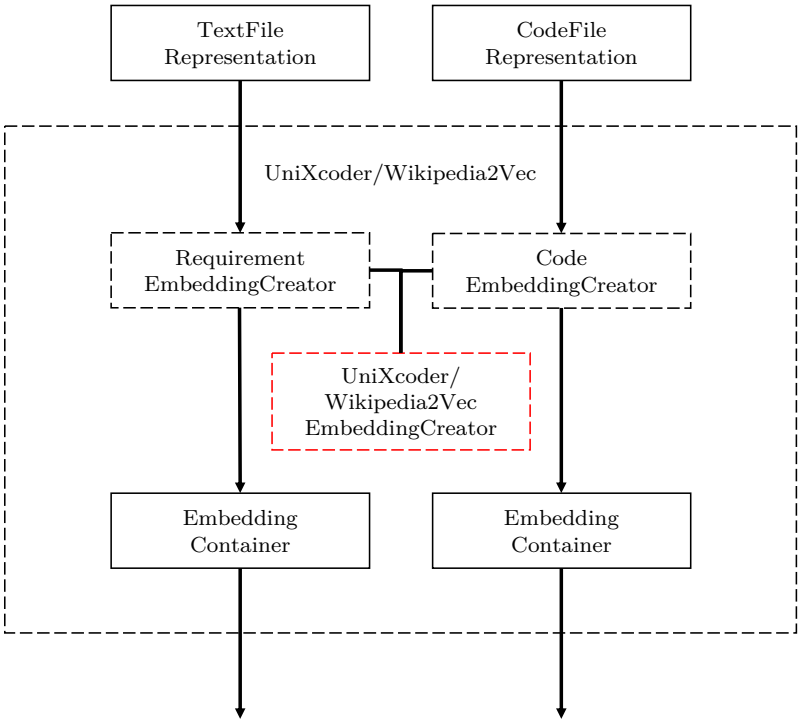


Abbildung 6.5: Architektur Sprachmodell UniXcoder und Wikipedia2Vec

repräsentationen der relevanten Worte. Diese Repräsentation wird in einem Objekt der Klasse `EmbeddingContainer` gekapselt.

Zur Integration des LMs UniXcoder wurde die von `WordEmbeddingCreator` erbende Klasse `UniXcoderEmbeddingCreator` erstellt. Dies ist in Abbildung 6.5 visualisiert. In `UniXcoderEmbeddingCreator` wird das UniXcoder-LM `unixcoder-base` [Mic22] initialisiert. Dieses wurde bimodal auf dem CodeSearchNet-Datensatz vortrainiert. Die Einbindung erfolgt mithilfe der Bibliothek `pytorch` [PGM+19]. Diese stellt NNs und auf NNs basierende LMs für eine performante Nutzung bereit. UniXcoder ist in der Lage, eine Wortfolge durch einen einzelnen Vektor zu repräsentieren. Diese Repräsentation eignet sich laut Abschnitt 5.5. Daher definiert `UniXcoderEmbeddingCreator` eine Methode `create_word_list_embedding`. Diese nimmt eine Wortfolge entgegen. Diese Wortfolge wird durch einen `CLS`-Token ergänzt, der der Wortfolge vorangestellt wird. Dieser symbolisiert die Semantik der gesamten Wortfolge (s. Abschnitt 5.4). Die Vektorrepräsentationen der einzelnen Worte und des `CLS`-Tokens ergeben sich dann aus den Gewichtungen der Recheneinheiten der letzten transparenten Schicht des NNs. Da die letzte transparente Schicht aus 768 Recheneinheiten besteht, haben die resultierenden Vektoren die Dimension 768. Die Methode `create_word_list_embedding` gibt die Vektorrepräsentation des `CLS`-Tokens und damit eine Repräsentation der gesamten Wortfolge zurück. Wie zuvor beschrieben, ist Methode `create_embeddings` in `RequirementEmbeddingCreator` und `CodeEmbeddingCreator` für die Repräsentation von Artefakt-Elementen zuständig. Diese ruft zum Einsatz von UniXcoder nun die Methode `create_word_list_embedding` in `UniXcoderEmbeddingCreator` auf. Dieser übergibt sie die gesamte relevante Wortfolge eines Artefakt-Elements und speichert die resultierende Vektorrepräsentation in einem `EmbeddingContainer`-Objekt.

Zur Integration von Wikipedia2Vec wurde die Klasse `Wikipedia2VecEmbeddingCreator` erstellt (s. Abbildung 6.5). Diese erbt analog zu `UniXcoderEmbeddingCreator` von `WordEmbeddingCreator`. In dieser Klasse wird das Wikipedia2Vec-LM `enwiki_20180-420` [YSTT16] verwendet. Es liegt im `.pkl`-Format vor. Es enthält auf englisch-sprachlichen

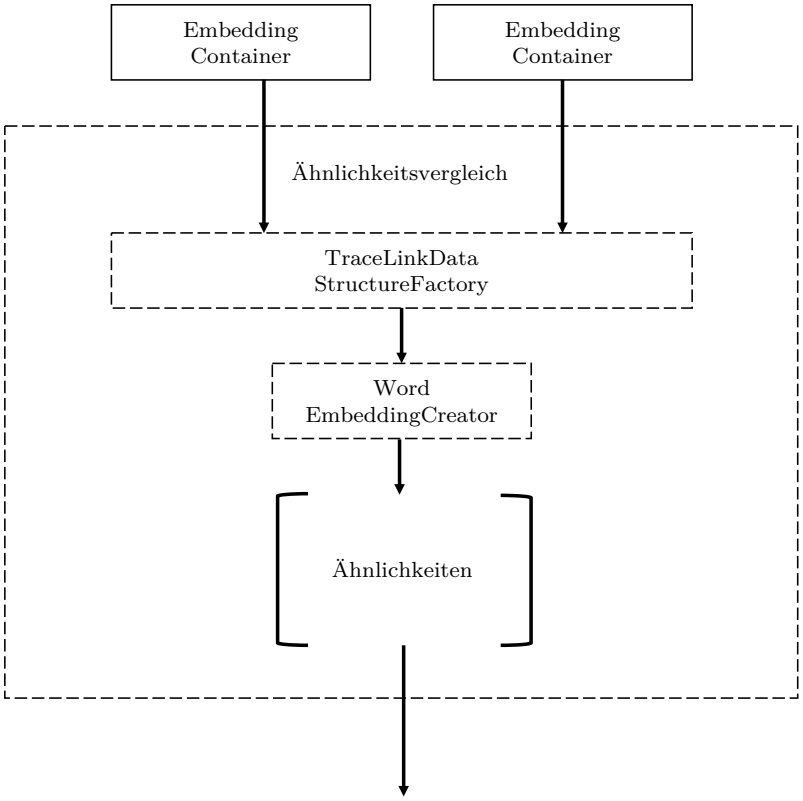


Abbildung 6.6: Rahmenarchitektur Ähnlichkeitsvergleich fastText

Wikipedia-Artikeln vortrainierte WEs und SEs mit einer Dimension von 100. Das Laden des Modells erfolgt durch eine interne Schnittstelle der Wikipedia2Vec-Bibliothek [YAS+20]. Außerdem werden in dieser Klasse die Dateien eingelesen, die die Artefaktworte und ihre im Zuge der Vorverarbeitung aufgelösten Wortbedeutungen enthalten. Zum Einlesen der `.tsv`-Dateien werden Funktionalitäten der `pandas`-Bibliothek [M+10] verwendet. Die eingelesenen Daten werden zur Weiterverarbeitung in einer Liste gespeichert. Jeder Eintrag der Liste umfasst ein Wort, seine aufgelöste Wortbedeutung sowie die Bezeichnung des Artefakt-Elements, in dem das Wort auftritt. Wikipedia2Vec repräsentiert Worte analog zu fastText einzeln. Wortfolgen werden durch BOEs repräsentiert. `Wikipedia2VecEmbeddingCreator` definiert daher analog zu `FastTextEmbeddingCreator` eine Methode `create_word_embedding`. Diese erhält ein Wort als Eingabe und liefert eine statische WE. Außerdem definiert `Wikipedia2VecEmbeddingCreator` die Methode `create_entity_embedding`. Diese erhält eine aufgelöste Wortbedeutung als Eingabe und gibt die von `enwiki_20180-420` gelernte SE zurück. Die Methode `create_embeddings` in `RequirementEmbeddingCreator` und `CodeEmbeddingCreator` kann mithilfe dieser Schnittstellen jedes relevante Artefakt-Element-Wort repräsentieren. Im ersten Schritt wird hierzu überprüft, ob für jedes Wort und die Artefakt-Element-Bezeichnung ein Eintrag in der zuvor eingelesenen Bedeutungs-Liste besteht. Ist dies der Fall, so wird das Wort durch Aufruf von `create_entity_embedding` durch eine SE repräsentiert. Konnte zu dem Wort des Artefakt-Elements keine Bedeutung aufgelöst werden, wird eine statische WE mittels `create_word_embedding` ermittelt. Die gesamte Wortfolge wird anschließend analog zur bisherigen Umsetzung durch eine BOE repräsentiert und diese in einem `EmbeddingContainer`-Objekt gekapselt.

6.3 Ähnlichkeitsvergleich

Im Anschluss an die Repräsentation von Artefakt-Elementen durch ein LM sollen Artefakt-Elemente auf Basis ihrer Repräsentation auf Ähnlichkeit verglichen werden. FTLR vergleicht im Zuge des Ähnlichkeitsvergleichs jedes Paar bestehend aus einem Anforderungsartefakt-Element und einem Quelltextartefakt-Element. In der bisherigen Rahmenarchitektur ist hierfür die Klasse `TraceLinkDataStructureFactory` zuständig (s. Abbildung 6.6). Diese Klasse wird durch eine Methode konfiguriert, die zwei Artefakt-Element-Repräsentationen auf Ähnlichkeit vergleicht. `TraceLinkDataStructureFactory` werden zudem zwei Mengen von `EmbeddingContainer`-Objekten übergeben. Eine Menge hält die gekapselten Repräsentationen aller Anforderungsartefakt-Elemente, die andere umfasst die Repräsentationen aller Quelltextartefakt-Elemente. Für jedes mögliche Tupel an `EmbeddingContainer`-Objekten, wobei die Tupelelemente nicht in derselben Menge liegen, führt `TraceLinkDataStructureFactory` die konfigurierte Ähnlichkeitsvergleichmethode durch. Die resultierenden Ähnlichkeitswerte werden in einer Ähnlichkeitsmatrix gespeichert. Diese wird im weiteren Verlauf zur Bestimmung von feingranularen und aggregierten TLs verwendet. In der bisherigen Umsetzung mit fastText wird die Ähnlichkeitsvergleichmethode `word_movers_distance` verwendet, die in `WordEmbeddingCreator` definiert ist (s. Abbildung 6.6). Sie berechnet die WMD-Ähnlichkeit zweier Artefakt-Elemente mithilfe einer entsprechenden Schnittstelle des fastText-LMs. Diese Schnittstelle nimmt zwei Wortfolgen entgegen, repräsentiert die Worte intern mittels fastText und liefert die WMD der Wortfolgen zurück. Daher ist im Fall des Einsatzes der WMD mit fastText keine Vektor-Repräsentation der einzelnen Worte eines Artefakt-Elements nötig, wie sie in Abschnitt 6.2 beschrieben wurde. In diesem Fall enthält das `EmbeddingContainer`-Objekt eines Artefakt-Elements keine BOE, sondern einfach die Menge der relevanten Worte. Um das fastText-LM nicht noch ein zweites Mal in einer anderen Klasse zu initialisieren, ist `word_movers_distance` in derselben Klasse definiert, die auch die Repräsentation von Worten über `create_word_embedding` anbietet. Die Methode `create_word_embedding` kann in der bisherigen Umsetzung beispielsweise beim Einsatz anderer Ähnlichkeitsmetriken zum Einsatz kommen.

Das verwendete UniXcoder-LM bietet keine Schnittstelle an, die die Repräsentation und den Vergleich zweier Wortfolgen kombiniert. Daher werden Artefakt-Elemente im Fall des Einsatzes von UniXcoder wie in Abschnitt 6.2 beschrieben, zunächst in `UniXcoderEmbeddingCreator` durch einen einzigen Vektor repräsentiert. Anschließend werden die resultierenden Vektoren auf Ähnlichkeit verglichen. Die Analyse mehrerer möglicher Ähnlichkeitsmetriken ergab, dass ein Vergleich dieser Vektoren mithilfe der Metrik der Kosinusähnlichkeit vermutlich geeignet ist (s. Abschnitt 5.4). Für die Durchführung dieses Ähnlichkeitsvergleichs wurde der Rahmenarchitektur von FTLR die Klasse `SimilarityComparator` hinzugefügt (s. Abbildung 6.7). Diese definiert die Ähnlichkeitsvergleichmethode `cosine_similarity`. Die Methode `cosine_similarity` erhält zwei Vektoren als Eingabe und berechnet deren Kosinusähnlichkeit mithilfe einer in `Util.py` bereitgestellten Funktionalität. Diese wiederum setzt eine Methode der `sklearn`-Bibliothek [BLB+13] ein. Die Ausgabe der Ähnlichkeitsvergleichmethode ist ein Wert im Intervall $[0, 1]$. Je höher der resultierende Wert, desto ähnlicher werden die Artefakt-Elemente im weiteren Verlauf gesehen.

Auch das verwendete Wikipedia2Vec-LM `enwiki_20180-420` bietet keine Schnittstelle zum direkten Vergleich zweier Artefakt-Element-Wortfolgen an. Artefakt-Elemente werden zunächst mithilfe von `Wikipedia2VecEmbeddingCreator` durch eine BOE repräsentiert. Anschließend werden zwei BOEs mittels der Ähnlichkeitsmetrik WMD verglichen. Die verwendete Ähnlichkeitsvergleichmethode `word_movers_distance` wird ebenfalls in der Klasse `SimilarityComparator` implementiert (s. Abbildung 6.7). Zur Berechnung

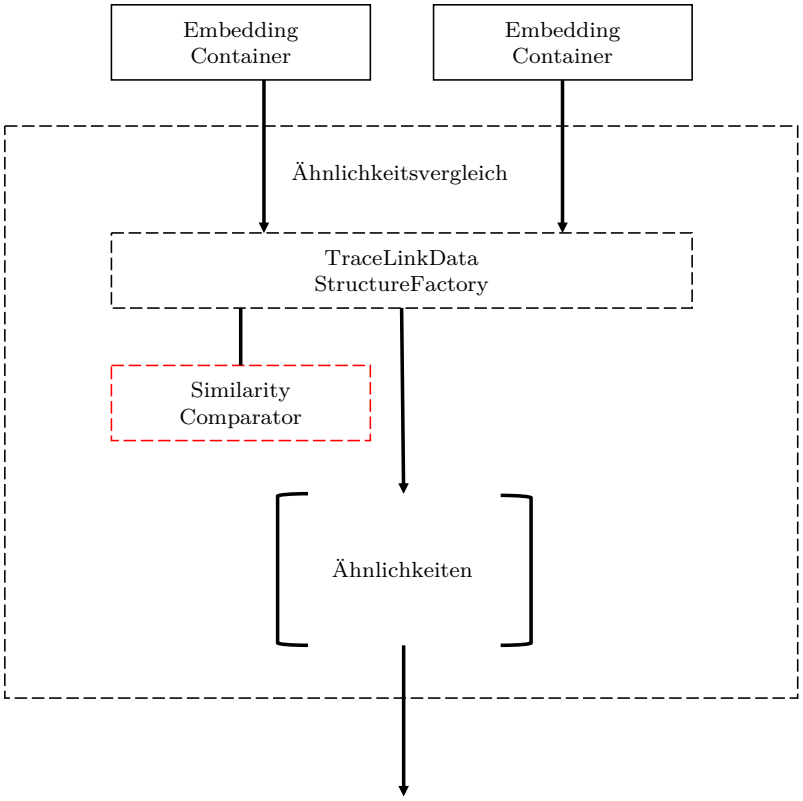


Abbildung 6.7: Architektur Ähnlichkeitsvergleich UniXcoder und Wikipedia2Vec

der WMD wird eine Funktion der `pyemd`-Bibliothek [PW09] eingesetzt. Der resultierende WMD-Wert ist ein Wert im Intervall $[0, 2]$ und wird vor der Ausgabe ins Intervall $[0, 1]$ abgebildet. Je niedriger der von der Vergleichsfunktion berechnete Wert, als desto ähnlicher werden die Artefakt-Elemente im weiteren Verlauf betrachtet.

7 Evaluation

Diese Arbeit hat zum Ziel, verschiedene LMs auf ihre Eignung zum Einsatz in FTLR zu untersuchen. Denn das bisher eingesetzte LM fastText besitzt Schwachstellen, die sich negativ auf die Güte der Ergebnisse von FTLR auswirken können (s. Kapitel 5). Unter diesem Hintergrund wurden zwei zu untersuchende LMs ausgewählt: UniXcoder und Wikipedia2Vec. Beide verfolgen vielversprechende Ansätze zur Sprachrepräsentation und wurden in ähnlichen Anwendungsfällen erfolgreich eingesetzt (s. Abschnitt 5.2). Daher wird erwartet, dass sie sich auch für den Einsatz in ATLR eignen. Außerdem eliminieren die Ansätze der beiden LMs mindestens eine von fastText's Schwachstellen. Daher wird erwartet, dass sich UniXcoder und Wikipedia2Vec besser für den Einsatz in FTLR eignen als fastText. Die in Kapitel 5 beschriebenen Hypothesen resultieren aus dieser Erwartungshaltung.

> **Zusammenfassung 7.1: Zentrale Hypothesen**
>
> FTLR erzielt bessere Ergebnisse bei der ATLR für ein Softwareprojekt beim Einsatz von UniXcoder als beim Einsatz von fastText.
>
> FTLR erzielt bessere Ergebnisse bei der ATLR für ein Softwareprojekt beim Einsatz von Wikipedia2Vec als beim Einsatz von fastText.

Sie sollen nun mithilfe von Experimenten auf Gültigkeit überprüft werden. Hierbei wird FTLR in Kombination mit UniXcoder und Wikipedia2Vec zur ATLR auf mehreren Vergleichsdatensätzen eingesetzt. Die Vergleichsdatensätze umfassen Anforderungs- und Quelltextartefakte von Softwareprojekten. Die von FTLR im Zuge der ATLR ermittelten Ergebnisse werden mithilfe von Evaluationsmetriken auf ihre Güte überprüft. Der Einsatz dieser Metriken macht die Qualität der Ergebnisse von FTLR und damit auch die Eignung der LMs vergleichbar. In den Tabellen der folgenden Abschnitte werden Nachkommastellen durch die Voranstellung eines Punktes deutlich gemacht. Außerdem wird eine 0 an erster Vorkommastelle nicht ausgeschrieben. Dies dient der Lesbarkeit der Daten in den Tabellen.

7.1 Vergleichsdatensätze

Im Rahmen der Evaluation soll die Güte der ATLR-Ergebnisse verglichen werden, die FTLR beim Einsatz verschiedener LMs erzeugt. Im Zuge dessen wird FTLR unter Einsatz

		Sprache		Anzahl Artefakte			Musterlösung		Grundlage	
Projekt	Domäne	NS	PS	Anf.	Quell.	Verb.	Anf.	Quell.	Präzision	F_1
eTour	Tourismus	EN	Java	58	116	308	.983	.767	.046	.088
iTrust	Gesundheit	EN	Java	131	226	286	.802	.385	.010	.020
LibEST	Industrie	EN	C	52	14	204	.786	.900	.280	.438
SMOS	Bildung	IT	Java	67	100	1044	1.000	.684	.159	.274
eAnci	Verwaltung	IT	Java	139	55	567	.281	1.000	.074	.137

Tabelle 7.1: Vergleichsdatensätze zur Evaluation von FTLR

der zu untersuchenden LMs auf mehreren Vergleichsdatensätzen ausgeführt. Daraus resultieren für jedes LM Ergebnisse, die FTLR auf den Vergleichsdatensätzen ermittelt. Die Güte dieser Ergebnisse kann im Anschluss vergleichend gegenübergestellt werden. Daraus kann abgeleitet werden, welches LM sich am besten für den Einsatz in FTLR auf den Vergleichsdatensätzen eignet. Auf Basis dessen können Schlussfolgerungen darüber gezogen werden, welches LM sich vermutlich allgemein am besten für den Einsatz in FTLR eignet. FTLR führt eine ATLR zwischen Anforderungs- und Quelltextartefakten durch. Daher müssen geeignete Vergleichsdatensätze derartige Artefakte enthalten. Auf Basis dieses Kriteriums wurden fünf Softwareprojekte ausgewählt, die im Rahmen der Evaluation als Vergleichsdatensätze fungieren: eTour, iTrust, LibEST, SMOS und eAnci. Alle Projekte enthalten Anforderungen und ein auf Basis dieser Anforderungen entwickeltes Softwaresystem bestehend aus Quelltextklassen. Damit ist es FTLR möglich, eine ATLR auf den Projekten durchzuführen. Bereitgestellt werden die Datensätze durch das CoEST [SSTC], einer Vereinigung von Wissenschaftlern, die an der Automatisierung von TLA forscht. Neben den Anforderungs- und Quelltextartefakten enthält jeder Datensatz eine Musterlösung an TLs zwischen den Artefakten [HCWT21]. Die von FTLR erzeugten TLs können zur Bemessung der Güte von FTLR mit den TLs in der Musterlösung verglichen werden. Die Softwareprojekte und einige ihrer Eigenschaften sind in Tabelle 7.1 veranschaulicht.

Zu diesen Eigenschaften gehören die Domäne des Projekts, die natürliche Sprache (NS) der Anforderungen und Quelltextbezeichner und die zur Umsetzung verwendete Programmiersprache (PS). Es wird deutlich, dass sich die Domänen der einzelnen Vergleichsdatensätze stark unterscheiden. Während eTour im Bereich Tourismus eingesetzt wird, ist SMOS ein Softwaresystem, welches im Bildungssektor verwendet wird. Somit decken die Projekte insgesamt ein breites Domänen-Spektrum ab. Außerdem unterscheiden sich die Projekte hinsichtlich der Sprache der Anforderungen und der Programmiersprache. Drei der vorliegenden Softwareprojekte enthalten Artefakte, die auf Englisch formuliert sind. Bei diesen handelt es sich um eTour, iTrust und LibEST. Die Artefakte der Vergleichsdatensätze SMOS und eAnci sind in italienischer Sprache verfasst. Es existiert ein UniXcoder-LM, welches auf englisch-sprachlichen Daten vortrainiert wurde (s. Abschnitt 6.2). Dieses ist daher in der Lage, die englisch-sprachlichen Artefakte von eTour, iTrust und LibEST zu repräsentieren. Das englisch-sprachliche UniXcoder-LM eignet sich nicht für Repräsentation der italienisch-sprachlichen Artefakte von SMOS und eAnci. Ein auf Trainingsdaten in italienischer Sprache vortrainiertes UniXcoder-LM steht nicht zur Verfügung. Aus diesem Grund wird FTLR beim Einsatz von UniXcoder auf Versionen von SMOS und eAnci ausgeführt, die ins Englische übersetzt wurden (SMOSTrans und eAnciTrans). Hierfür kann das bestehende englisch-sprachliche UniXcoder-LM verwendet werden. Für Wikipedia2Vec existiert ein auf englisch-sprachlichen Daten und ein auf italienisch-sprachlichen Daten vortrainiertes LM. Die WSD, die für den Einsatz des Wikipedia2Vec-LMs benötigt wird, ist jedoch bislang nur für englisch-sprachliche und nicht für italienisch-sprachliche Artefakte möglich. Daher wird FTLR auch beim Einsatz von Wikipedia2Vec auf den übersetzten

Versionen von SMOS und eAnci ausgeführt. Die Anforderungen von eTour, iTrust, SMOS und eAnci wurden mithilfe der Programmiersprache Java umgesetzt. Nur LibEST wurde in C entwickelt. Die Programmiersprache C ist nicht objektorientiert und kennt keine Klassen. Einzelne Dateien enthalten jedoch trotzdem Methoden, die repräsentiert und für das Bestimmen von TLs genutzt werden können. Tabelle 7.1 veranschaulicht darüber hinaus die Anzahl der Anforderungs- und Quelltextartefakte sowie die Anzahl der TLs der Musterlösung. Es ist anzumerken, dass die Anforderungsartefakte von eTour, iTrust, SMOS und eAnci ausschließlich UCTs enthalten. Im Fall von eTour, SMOS und eAnci umfassen diese UCTs eine Bezeichnung des Anwendungsfalls, eine kurze Beschreibung, die teilnehmenden Akteure, Eingangs- und Ausgangsbedingungen sowie den eigentlichen Ablauf. Im Fall von iTrust umfassen die UCTs lediglich die Beschreibung des Anwendungsfalls. Die UCT-Beschreibung wird im Fall dieser Softwareprojekte als die eigentliche Anforderung betrachtet. Die Anforderungsartefakte von LibEST enthalten ausschließlich Anforderungssätze. Im Tabellen-Abschnitt zur Musterlösung wird veranschaulicht, wie viele der Artefakte in mindestens einer TL der Musterlösung auftreten. Beispielsweise enthält die Musterlösung im Fall von eTour TLs, in denen 98,3% und damit 52 der 53 Anforderungen enthalten sind. Im Fall von iTrust und eAnci enthält die Musterlösung nur einen niedrigen Anteil der Quelltext- bzw. Anforderungsartefakte. Dies lässt darauf schließen, dass hier nicht alle Anforderungen in Quelltext umgesetzt wurden oder nicht alle Funktionalitäten des Systems durch Anforderungen beschrieben wurden. Damit bestehen nur zwischen wenigen Anforderungs- und Quelltextartefakten TLs. FTLR ermittelt TL-Kandidaten durch das Bestimmen des Anforderungsartefakts mit der höchsten semantischen Übereinstimmung für ein gegebenes Quelltextartefakt. Damit erwartet FTLR implizit, dass jedes Artefakt Teil einer TL ist. Bestehen in einem Vergleichsdatensatz aber nur sehr wenige tatsächliche TLs, kann dies zu vielen Fällen führen, in denen FTLR fälschlicherweise eine TL ermittelt. Abschließend stellt die Tabellen-Sektion „Grundlage“ dar, wie hoch die Güte eines Verfahrens zur ATLR wäre, das zwischen allen Artefakten eine TL ermittelt. Auf den Datensätzen eTour, iTrust und eAnci ist die Güte eines solchen einfachen Verfahrens niedrig. LibEST und SMOS dagegen enthalten nur wenige Artefakte, von denen jedoch viele mit einer hohen Zahl anderer Artefakte verbunden sind. Daraus ergibt sich beispielsweise der hohe F_1-Wert (0,438) der Ergebnisse des Grundlagen-Verfahrens im Fall von LibEST. Damit ein Verfahren zur ATLR als geeignet betrachtet werden kann, muss es Ergebnisse ermitteln, die mindestens der Güte der Ergebnisse des Grundlagen-Verfahrens entsprechen.

7.2 Evaluationsmetriken

Evaluationsmetriken bemessen die Güte der Ergebnisse von Verfahren. Damit kann die Qualität von Verfahren auf Basis der Werte von Evaluationsmetriken miteinander verglichen werden. Im Fall von FTLR ergibt sich als Verfahrensergebnis eine Liste von TLs. Die TLs dieser Liste können gemäß ihrer Korrektheit klassifiziert werden. Die Klassifizierung erfolgt in Abhängigkeit davon, ob eine TL von FTLR ermittelt wird und ob sie auch in der Musterlösung enthalten ist. Dies ist auch in Tabelle 7.2 veranschaulicht. Ist eine von FTLR ermittelte TL auch Teil der Musterlösung, so wird die TL als zurecht positives Ergebnis (engl.: true positive, TP) bezeichnet. Ermittelt FTLR eine TL fälschlicherweise und ist sie nicht in der Musterlösung zu finden, so ist diese TL ein fälschlich positives Ergebnis (engl.: false positive, FP). Tritt der Fall auf, dass FTLR eine in der Musterlösung enthaltene TL nicht erkennt, so spricht man von einem fälschlich negativen Ergebnis (engl.: false negative, FN). Wird eine TL von FTLR korrekterweise nicht ermittelt, so bezeichnet man diese TL als zurecht negatives Ergebnis (engl.: true negative, TN).

Diese Klassifizierung der Ergebnisse von FTLR bildet die Grundlage der Evaluationsmetriken. Als erste Metrik lässt sich die Ausbeute ausführen [JM22]. Im Kontext von ATLR

		FTLR ermittelt RV	
		ja	nein
RV in Musterlösung	ja	zurecht positiv	fälschlich negativ
	nein	fälschlich negativ	zurecht negativ

Tabelle 7.2: Klassifizierung der Ergebnisse von FTLR

beschreibt die Ausbeute, wie viele der TLs der Musterlösung durch FTLR ermittelt werden. Sie ist definiert als der Anteil der Menge von TPs an der Menge der gesamten TLs der Musterlösung. Die Ausbeute gibt damit die Fähigkeit von FTLR an, bestehende TLs zu erkennen. Darüber hinaus besteht die Metrik der Präzision. Die Präzision gibt an, wie viele der von FTLR ermittelten TLs korrekt sind. Sie ist damit definiert als der Menge von TPs an der Menge der gesamten von FTLR ermittelten TLs. Die Präzision veranschaulicht damit die Fähigkeit von FTLR, keine TLs zwischen unzusammenhängenden Artefakten zu ermitteln. Es wird deutlich, dass hohe Werte in beiden Metriken gleichermaßen erstrebenswert sind. Zum einen soll FTLR möglichst viele der bestehenden TLs erkennen. Ohne diese Eigenschaft wäre FTLR kaum nützlich, um logische Verbindungen zwischen Artefakten aufzudecken. Handelt es sich bei den erkannten TLs jedoch größtenteils um FPs, so liefert FTLR zu ungenaue Ergebnisse und somit keinen echten Mehrwert. Daher soll FTLR TLs also zusätzlich möglichst korrekt erkennen. Das F_1-Maß kombiniert die Metriken Ausbeute und Präzision. Es berechnet sich als harmonischer Mittelwert zwischen Präzision und Ausbeute:

$$F_1 = 2 \cdot \frac{Ausbeute * Präzision}{Ausbeute + Präzision} \tag{7.1}$$

Der Wertebereich des F_1-Maß ist $[0, 1]$. Damit liefert F_1 eine Aussage über die gleichzeitige Erfüllung beider gewünschten Eigenschaften der Ergebnisse von FTLR. Hohe F_1-Werte verdeutlichen, dass FTLR in der Lage ist, viele bestehende TLs zu erkennen, ohne viele FP zu produzieren. Es ist eine häufig verwendete Metrik zur Evaluation von Klassifikations- und ATLR-Verfahren. Daher wird das F_1-Maß auch in dieser Arbeit zur Evaluation der Güte von FTLR verwendet.

Viele derzeitige Verfahren zur ATLR erzielen keine zufriedenstellenden Ergebnisse im F_1-Maß. Sie sind zwar in der Lage einen Großteil der bestehenden TLs zu erkennen, ermitteln jedoch deutlich zu viele TLs und haben somit niedrige Werte in der Metrik Präzision. Aus diesem Grund liefern viele dieser Verfahren für ein gegebenes Artefakt eine Liste von Kandidaten-TLs, die von diesem Artefakt ausgehen. Diese Liste ist danach sortiert, wie wahrscheinlich es ist, dass es sich bei einer Kandidaten-TL tatsächlich um eine TL handelt. Das Erkennen, welche dieser Kandidaten-TLs tatsächliche TLs sind, wird dann manuell durchgeführt. Die Metrik der MAP beschreibt mit welcher Präzision tatsächliche TLs auch an vorderen Stellen in der sortierten Kandidaten-TL Liste vorkommen. Sie sagt damit aus, wie hoch der manuelle Aufwand beim Erkennen der tatsächlichen TLs noch ist. Damit eignet sie sich für die Evaluation dieser Verfahren. FTLR soll zur Einordnung seiner Güte auch mit derartigen Verfahren verglichen werden. Aus diesem Grund wird FTLR, auch in dieser Arbeit, ebenfalls mittels der Metrik MAP evaluiert. FTLR ermittelt im Verlauf des Verfahrens Kandidaten-TLs zwischen Anforderungs- und Quelltextartefakten. Liegt die semantische Ähnlichkeit der Artefakte einer Kandidaten-TL über einem gegebenen Schwellenwert, wird die Kandidaten-TL zu einer tatsächlichen TL. Zur Berechnung der MAP wird kein Schwellenwert angewandt und die Kandidaten-TLs ungefiltert und nach semantischer Ähnlichkeit sortiert als TLs übernommen. Auf diese Kandidaten-TL-Liste

	eTour		iTrust		LibEST		SMOSTrans		eAnciTrans		
Ansatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	$\varnothing F_1$
NQK	.402	.393	.217	.258	.550	.642	.303	.343	.256	.152	.346
NQK + UCT	.447	.489	.217	.258	.552	.629	.320	.354	**.296**	.173	.366
NQK + MC	.444	.408	**.282**	.275	**.568**	.653	.328	.384	.248	.148	.374
NQK + MB	.392	.360	.160	.169	.559	.622	.310	.339	.252	.148	.335
NQK + UCT & MC	.437	.474	**.282**	.275	.566	.649	.346	.397	.267	.158	.380
NQK + UCT & MB	.402	.402	.160	.169	.567	.605	.317	.354	.277	.162	.345
NQK + MC & MB	.410	.380	.176	.177	.550	.610	.322	.362	.242	.144	.340
NQK + UCT & MC & MB	.384	.395	.176	.177	.551	.619	.342	.382	.248	.154	.340
ALL	.412	.372	.218	.217	.540	.603	.320	.344	.238	.126	.346
ALL + UCT	**.515**	.504	.218	.217	.533	.588	.355	.350	.280	.143	.380
ALL + MC	.404	.343	.230	.243	.554	.589	.332	.359	.215	.132	.347
ALL + MB	.347	.348	.140	.118	.559	.619	.320	.327	.192	.123	.312
ALL + UCT & MC	.496	.475	.230	.243	.550	.564	**.368**	.367	.269	.145	**.383**
ALL + UCT & MB	.453	.446	.140	.118	.547	.568	.344	.339	.231	.137	.343
ALL + MC & MB	.359	.319	.140	.121	.547	.591	.330	.344	.179	.127	.311
ALL + UCT & MC & MB	.448	.415	.140	.121	.533	.563	.351	.355	.198	.136	.334

Tabelle 7.3: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder und WMD

wird dann die Metrik der MAP angewandt. Der Wertebereich der MAP ist analog zum F_1-Maß $[0, 1]$.

7.3 Evaluation des Sprachmodell-Einsatzes in FTLR

In diesem Abschnitt soll evaluiert werden, ob sich UniXcoder und Wikipedia2Vec tatsächlich besser für den Einsatz in FTLR eignen als fastText. Hierfür wird FTLR in Kombination mit den drei LMs auf den in Abschnitt 7.1 vorgestellten Datensätzen ausgeführt. Je besser die Werte der Ergebnisse von FTLR in den vorgestellten Evaluationsmetriken (siehe Abschnitt 7.2) sind, desto geeigneter ist das eingesetzte LM. In Kapitel 5 wurde deutlich, dass die Art der Vorverarbeitung und der verwendete Ähnlichkeitsvergleich in FTLR auf das LM abgestimmt sein müssen. Beispielsweise wird vermutet, dass ein Einbeziehen des MBs in die Repräsentation von Quelltextartefakt-Elementen durch UniXcoder positive Einflüsse auf die Ergebnisse von FTLR hat. Eine konkrete Wahl der Vorverarbeitung und des Ähnlichkeitsvergleichs wird im Folgenden als Konfiguration bezeichnet. Im ersten Schritt der Evaluation wird für die LMs UniXcoder und Wikipedia2Vec ermittelt, mit welcher Konfiguration sie zu den besten FTLR-Ergebnissen führen. Hierzu wurden in Kapitel 5 bereits Hypothesen aufgestellt, die im Zuge dessen evaluiert werden. In einem finalen zweiten Schritt werden die bestmöglichen Ergebnisse der LMs mit den Werten unter Einsatz von fastText verglichen.

7.3.1 Evaluation des Einsatzes von UniXcoder in FTLR

In diesem Abschnitt soll evaluiert werden, wie geeignet UniXcoder für den Einsatz in FTLR ist. In Kapitel 5 wurden bereits Hypothesen in Bezug auf sinnvolle Konfigurationen für den Einsatz von UniXcoder aufgestellt. Zunächst wird angenommen, dass die alleinige Durchführung der Vorverarbeitungsschritte Nichtbuchstaben- und Querverweisfilterung und Kleinbuchstaben-Transformation (NQK) dem Einsatz aller in Abschnitt 5.3 beschriebenen Vorverarbeitungsschritte (ALL) vorzuziehen ist (s. Hypothese 5.3). Außerdem wurde in Hypothese 5.4 angenommen, dass das Einbeziehen des MBs in die Eingabe von FTLR sinnvoll ist. Hypothese 5.7 besagt, dass sich die Metrik der Kosinusähnlichkeit besser zum Ähnlichkeitsvergleich eignet als die WMD. Es gilt nun zu evaluieren, ob diese Hypothesen auf Basis der Ergebnisse von FTLR auf Vergleichsdatensätzen akzeptiert werden können.

	eTour		iTrust		LibEST		SMOSTrans		eAnciTrans		
Ansatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	$\varnothing F_1$
NQK	.440	.477	.242	.239	.581	.635	.311	.349	.216	.146	.358
NQK + UCT	.404	.435	.242	.239	.574	.631	.311	.305	.264	.170	.359
NQK + MC	.453	.450	**.307**	.292	**.594**	.666	.340	.400	.230	.148	.385
NQK + MB	.395	.403	.169	.161	.585	.636	.318	.340	.228	.147	.339
NQK + UCT & MC	.346	.372	**.307**	.292	.580	.652	.318	.345	.242	.152	.359
NQK + UCT & MB	.323	.350	.169	.161	.575	.634	.304	.294	.253	.160	.325
NQK + MC & MB	.386	.376	.171	.179	.581	.625	.338	.385	.224	.145	.340
NQK + UCT & MC & MB	.313	.291	.171	.179	.560	.620	.313	.325	.248	.152	.321
ALL	.428	.409	.216	.234	.567	.600	.331	.354	.243	.152	.353
ALL + UCT	**.524**	.501	.216	.234	.567	.583	.357	.358	**.272**	.147	**.387**
ALL + MC	.411	.373	.233	.239	.583	.561	.346	.369	.211	.141	.357
ALL + MB	.364	.348	.121	.114	.580	.636	.329	.336	.243	.148	.327
ALL + UCT & MC	.482	.461	.233	.239	.566	.531	**.367**	.376	.234	.143	.376
ALL + UCT & MB	.434	.411	.121	.114	.582	.600	.337	.345	.240	.145	.343
ALL + MC & MB	.361	.323	.124	.109	.556	.550	.344	.356	.212	.143	.312
ALL + UCT & MC & MB	.408	.389	.124	.109	.535	.524	.350	.363	.214	.139	.326

Tabelle 7.4: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder und Kosinusähnlichkeit

In Abhängigkeit der angenommenen Gültigkeit bzw. Ungültigkeit der Hypothesen können Rückschlüsse auf die geeignetste Konfiguration für UniXcoder gezogen werden.

Die Tabellen 7.3 und 7.4 veranschaulichen die F_1-Güte und die MAP der Ergebnisse von FTLR unter Einsatz von UniXcoder. Im Anhang finden sich Tabellen, die zusätzlich die Werte in den Metriken Präzision und Ausbeute darstellen (s. Abschnitt A.1). Es gilt anzumerken. dass diese Ergebnisse unter Einsatz von idealen Schwellenwerten ermittelt wurden. FTLR verwendet zwei Schwellenwerte zur Bestimmung von TLs auf Basis von ermittelten Ähnlichkeiten zwischen Artefakt-Elementen (s. Kapitel 4). Diese können für jede Konfiguration und jeden Datensatz individuell so gesetzt werden, dass FTLR die für gegebene Artefakt-Element-Ähnlichkeiten bestmöglichen Ergebnisse erzielt. In diesem Fall wird von idealen Schwellenwerten gesprochen. FTLR wurde im Rahmen der Untersuchung mit unterschiedlichsten Konfigurationen auf den fünf in Abschnitt 7.1 vorgestellten Datensätzen ausgeführt. Für die Vorverarbeitung wurden entweder nur die Schritte NQK oder ALL durchgeführt. Anforderungsartefakte können optional durch UCTs erweitert werden (s. Kapitel 4). Die Eingabe für Quelltextartefakt-Elemente kann optional durch MCs und den MB erweitert werden. Für diese möglichen Erweiterungen wurden alle möglichen Kombinationen ausgeführt. Als Metrik des Ähnlichkeitsvergleichs kamen Kosinusähnlichkeit und WMD zum Einsatz. In die Repräsentation von Quelltextartefakt-Elementen können optional die Bezeichner der Datentypen von Methodenparametern einfließen. Diese wurden bei der Durchführung der Experimente nicht einbezogen. Im Anhang findet sich eine Tabelle, die die Güte der Ergebnisse von FTLR bei Einsatz von UniXcoder in den besten drei Konfigurationen unter Einbeziehung der Datentypen zeigt (s. Abschnitt A.1). Es wird deutlich, dass die zusätzliche Verwendung der Datentypen zur Repräsentation keine wesentlichen Verbesserungen für den Einsatz von UniXcoder bringt.

Die beste Konfiguration für den Datensatz eTour ist durch Kosinusähnlichkeit und ALL + UCT gegeben. Hier erzielt FTLR unter Einsatz von UniXcoder Ergebnisse mit einer F_1-Güte von 0,524 und einer MAP von 0,501 (s. Tabelle 7.4). Die Konfiguration WMD und ALL + UCT ist mit Ergebnissen von 0,515 in F_1 und einer MAP von 0,504 die zweitbeste Konfiguration für eTour. Auf iTrust und LibEST eignet sich die Konfiguration Kosinusähnlichkeit und NQK + MC am besten für den Einsatz von UniXcoder in FTLR. Die von FTLR ermittelten Ergebnisse haben für diese Konfiguration eine Güte von 0,307

	eTour	iTrust	LibEST	SMOSTrans	eAnciTrans	∅
NQK	.453	.307	.594	.346	.296	.399
ALL	.524	.233	.583	.368	.280	.398

Tabelle 7.5: Bestmögliche F_1-Güte der Ergebnisse von FTLR für Konfigurationen mit NQK bzw. ALL als Vorverarbeitung

in F_1 für iTrust und 0,594 in F_1 für LibEST. Für das Softwareprojekt SMOS in seiner übersetzten Form SMOSTrans erreicht FTLR unter Einbeziehung von UCTs und MCs die besten Ergebnisse. Für WMD und UCT & MC erzielt FTLR unter Einsatz von UniXcoder Ergebnisse mit einer Güte von 0,368 in F_1 und 0,367 in MAP. Wird die Kosinusähnlichkeit anstelle der WMD verwendet, ergibt sich ein F_1-Wert von 0,367 und ein MAP-Wert von 0,376. FTLR erzielt mit UniXcoder auf eAnciTrans die besten Ergebnisse mit der Konfiguration WMD und NQK + UCT (s. Tabelle 7.3). Die erzielten Ergebnisse haben eine F_1-Güte von 0,296 und eine MAP von 0,173. Es wird deutlich, dass die beste Konfiguration für den Einsatz von UniXcoder von Vergleichsdatensatz zu Vergleichsdatensatz variiert. Im Durchschnitt über alle Vergleichsdatensätze lassen sich drei Konfigurationen identifizieren, die für den Einsatz von UniXcoder besonders geeignet erscheinen. Für die Konfiguration Kosinusähnlichkeit und ALL + UCT ergibt sich über alle fünf Vergleichsdatensätze ein durchschnittlicher F_1-Wert von $0,387$. Damit handelt es sich bei dieser Konfiguration um die durchschnittlich beste in F_1. Die Konfiguration Kosinusähnlichkeit und NQK + MC ist die durchschnittlich zweitbeste Konfiguration mit einer F_1-Güte von durchschnittlich 0,385. Bei WMD und UCT + MC handelt es sich um die drittbeste Konfiguration. FTLR erzielt unter Einsatz von UniXcoder mit dieser Konfiguration Ergebnisse mit einer Güte von durchschnittlich 0,383 in F_1 auf den fünf Vergleichsdatensätzen. Die in Kapitel 5 beschriebenen Hypothesen implizieren, dass die beste Konfiguration durch eine Konfiguration mit Kosinusähnlichkeit, NQK und MB gegeben ist. Die im Durchschnitt beste Konfiguration mit Kosinusähnlichkeit, NQK und MB ist die Konfiguration Kosinusähnlichkeit und NQK + MC + MB. Für sie ergeben sich FTLR-Ergebnisse mit einer durchschnittlichen Güte von 0,340 in F_1. Die tatsächlich besten Konfigurationen sind somit mit einem Unterschied von mindestens 0,043 in F_1 besser als die erwartete beste Konfiguration. Die Frage liegt nahe, welche Einflussfaktoren in der Konfiguration maßgeblich für diese Abweichung sind und welche Hypothesen nun auf Basis der Vergleichsdaten Gültigkeit besitzen.

Hypothese 5.3 besagt, dass der Einsatz von ausschließlich NQK zur Vorverarbeitung geeigneter ist als der von ALL. Daher soll nun in einem ersten Schritt für jedes Projekt betrachtet werden, mit welcher Vorverarbeitung die besten FTLR-Ergebnisse möglich sind. Die Konfiguration, mit der FTLR die besten Ergebnissen in F_1 ermittelt und die NQK zur Vorverarbeitung einsetzt, ist für eTour, iTrust und LibEST gegeben durch Kosinusähnlichkeit und NQK + MC. Bei dieser Konfiguration handelt es sich auch um die durchschnittlich zweitbeste Konfiguration. Auf eTour ermittelt FTLR mit dieser Konfiguration Ergebnisse mit einer F_1-Güte von 0,453, auf iTrust mit einer Güte von 0,307 in F_1 und auf LibEST mit 0,594 in F_1. Auf iTrust und LibEST ist auch die MAP dieser Ergebnisse die beste MAP aller Ergebnisse, die mit einer Konfiguration mit NQK erzielt wurden. Auf SMOSTrans eignet sich bei Verwendung von NQK für die Vorverarbeitung der Einsatz von WMD und das Einbeziehen von UCT & MC am besten. Für diese Konfiguration erzielt FTLR Ergebnisse mit einer Güte von 0,346 in F_1 und einer MAP von 0,397. Für eAnciTrans ist WMD und NQK + UCT mit einer F_1-Güte der Ergebnisse von 0,296 die beste Konfiguration mit NQK. Wird ALL zur Vorverarbeitung eingesetzt, so erzielt FTLR auf eTour die besten Ergebnisse in F_1 (0,524) und die zweitbesten Ergebnisse in MAP (0,501) mit der Konfiguration Kosinusähnlichkeit und ALL + UCT. Diese Konfiguration ist gleichzeitig auch die durchschnittlich beste Konfiguration. Auf iTrust und LibEST

ist die Konfiguration Kosinusähnlichkeit und ALL + MC unter allen Konfigurationen mit ALL am besten geeignet. Mit ihr erzielt FTLR auf iTrust Ergebnisse mit einer F_1-Güte von 0,233 und auf LibEST Ergebnisse mit einer Güte von 0,583 in F_1. Die beste Konfiguration in F_1 für SMOSTrans unter Verwendung von ALL ist die bereits beschriebene, insgesamt beste Konfiguration für SMOS (WMD und ALL + UCT & MC). Für eAnciTrans führt die Konfiguration WMD und ALL + UCT zu den besten Ergebnissen in F_1 (0,280) von FTLR mit einer Konfiguration mit ALL. Tabelle 7.5 veranschaulicht für jeden Datensatz die F_1 Güte der bestmöglichen Ergebnisse, die mit einer Konfiguration mit NQK bzw. ALL erzielt werden können. Es wird deutlich, dass unter Verwendung von NQK zur Vorverarbeitung auf iTrust, LibEST und eAnciTrans bessere Ergebnisse von FTLR möglich sind als unter Verwendung von ALL. Auf LibEST führt der Einsatz von NQK zu 0,011 besseren Ergebnissen in F_1 als bei Einsatz von ALL und auf eAnci zu 0,016 besseren. Für iTrust ist der Unterschied der F_1-Güte der Ergebnisse von FTLR, die mit NQK bzw. ALL möglich sind, mit 0,074 signifikant hoch. Diese Ergebnisse indizieren, dass Hypothese 5.3 auf den Vergleichsdatensätzen iTrust, LibEST und eAnciTrans Gültigkeit besitzt. Um die Annahme der Gültigkeit von Hypothese 5.3 auf iTrust weiter zu begründen, wurde mithilfe der ermittelten Daten ein Wilcoxon-Vorzeichen-Rang-Test durchgeführt. Hierfür wurde die F_1-Güte der FTLR-Ergebnisse auf iTrust für jedes Konfigurationspaar, das sich nur in der Art der Vorverarbeitung unterscheidet, verglichen. Hiermit sollte untersucht werden, ob der Einsatz von NQK auf iTrust zu statistisch signifikant besseren Ergebnissen von FTLR führt als der Einsatz von ALL. Dies würde Hypothese 5.3 auf iTrust untermauern. Die Wahrscheinlichkeit, die Nullhypothese zu Hypothese 5.3 zu Unrecht abzulehnen, wurde als 0,042% ermittelt. Bei einem Signifikanzniveau von 5% ist dies ausreichend, um Hypothese 5.3 auf Basis des Tests als gültig auf iTrust zu betrachten. Hypothese 5.3 wurde mit der Begründung angenommen, dass UniXcoder mithilfe der NQK-Vorverarbeitung sprachliche Einheiten als Eingabe erhält, die seinen Trainingsdaten ähneln (s. Abschnitt 5.3). Bei NQK wird u.a. im Gegensatz zu ALL keine Wortlemmatisierung oder Stoppwortentfernung durchgeführt. UniXcoder wurde auf unlemmatisierten Daten mit Stoppworten vortrainiert. Daher wurde davon ausgegangen, dass der Einsatz von UniXcoder mit NQK zu genaueren Ergebnissen von FTLR führt als der Einsatz mit ALL. Auf den Vergleichsdatensätzen iTrust, LibEST und eAnciTrans lässt sich der positive Effekt von NQK im Vergleich zu ALL bestätigen. Auf den Projekten eTour und SMOSTrans sind jedoch unter Verwendung von ALL zur Vorverarbeitung bessere Ergebnisse von FTLR möglich als unter Verwendung von NQK (s. Tabelle 7.5). Während die beste Konfiguration mit NQK auf eTour zu Ergebnissen mit lediglich 0,453 F_1-Güte führt, ist mit ALL eine Güte von 0,524 in F_1 möglich. Daraus ergibt sich eine Verbesserung von 0,071 in F_1 beim Einsatz von ALL im Vergleich zu NQK. Auf SMOSTrans erzielt FTLR mit der besten Konfiguration mit ALL um 0,026 in F_1 bessere Ergebnisse als mit der besten Konfiguration mit NQK. Die Ergebnisse des Experiments sprechen somit dafür, dass NQK auf eTour und SMOSTrans nicht besser geeignet ist als ALL. Damit ist Hypothese 5.3 auf eTour und SMOSTrans vermutlich ungültig. Eine mögliche Erklärung hierfür ist, dass die Artefakte von eTour und SMOSTrans viele Stoppworte und Worte mit wenig Semantik enthalten. Diese werden im Zuge der NQK-Vorverarbeitung nicht herausgefiltert. Ihre hohe Zahl führt dazu, dass UniXcoder sie dennoch stark in die Repräsentation der Artefakt-Elemente einbezieht. Damit werden Artefakt-Elemente zu ungenau repräsentiert. Eine ALL-Vorverarbeitung ist in der Lage, die semantisch nicht relevanten Worte vor der Repräsentation zu entfernen. Damit werden Artefakt-Elemente unter Einsatz von ALL genauer repräsentiert als unter Einsatz von NQK. Dies würde die höhere Güte der Ergebnisse von FTLR bei Verwendung von ALL auf eTour und SMOSTrans erklären. Zusammenfassend wurde evaluiert, dass Hypothese 5.3 auf iTrust, LibEST und eAnciTrans als gültig angenommen werden kann. Auf eTour und SMOSTrans ist sie vermutlich ungültig. Die Gültigkeit der Hypothese hängt somit von der Beschaffenheit des Datensatzes ab, auf dem FTLR mit UniXcoder durchgeführt

	eTour	iTrust	LibEST	SMOSTrans	eAnciTrans	∅
MB	.410	.176	.581	.338	.277	.356
Kein MB	.524	.307	.594	.368	.296	.418

Tabelle 7.6: Bestmögliche F_1-Güte der Ergebnisse von FTLR für Konfigurationen mit MB bzw. ohne MB

wird. eTour und SMOSTrans enthalten möglicherweise eine hohe Zahl an Stoppworten und Worten ohne Semantik, was NQK auch beim Einsatz von UniXcoder ungeeignet macht. Allgemein könnte Hypothese 5.3 somit für Datensätze mit dieser Eigenschaft ungültig sein und für Datensätze ohne diese Eigenschaft wie iTrust, LibEST und eAnci gültig. Tabelle 7.5 zeigt auch die durchschnittlich bestmögliche F_1-Güte beim Einsatz von NQK bzw. ALL über alle Vergleichsdatensätze hinweg. Bei Konfigurationen mit NQK sind FTLR-Ergebnisse mit einer Güte von durchschnittlich 0,399 in F_1 möglich. Konfigurationen mit ALL führen zu einer bestmöglichen Ergebnis-Güte von durchschnittlich 0,398 in F_1. Die durchschnittliche Güte der besten Ergebnisse beim Einsatz von NQK bzw. ALL unterscheidet sich damit marginal. Dies deutet darauf hin, dass das Durchführen von NQK und von ALL im Durchschnitt ähnlich gut für UniXcoder geeignet ist. Dies spricht dafür, dass Hypothese 5.3 gemessen am Durchschnitt über alle Vergleichsdatensätze ungültig ist. Auch die Tatsache, dass der höchste durchschnittliche F_1-Wert von 0,387 mit einer Konfiguration unter Einsatz von ALL und nicht unter Einsatz von NQK erzielt wurde, spricht für die Ungültigkeit von Hypothese 5.3 im Durchschnitt über alle Vergleichsdatensätze.

Hypothese 5.3 nimmt an, dass das Einbeziehen von MBs in die Eingabe von UniXcoder einen positiven Effekt auf die Güte der Ergebnisse von FTLR hat. Tabelle 7.6 veranschaulicht die bestmögliche F_1-Güte der Ergebnisse, die FTLR für die einzelnen Vergleichsdatensätze mit einer Konfiguration mit bzw. ohne MBs ermittelt. Die für eTour geeignetste Konfiguration mit MB ist durch die Konfiguration WMD und NQK + MC & MB gegeben. Die Ergebnisse von FTLR haben bei dieser Konfiguration eine Güte von 0,410 in F_1. Dieselbe Konfiguration führt auch auf iTrust zu den besten FTLR-Ergebnissen. Kosinusähnlichkeit und NQK + MC & MB führt auf LibEST (0,581 in F_1) und SMOSTrans (0,338 in F_1) zu den besten FTLR-Ergebnissen, die unter Einbeziehen von MBs ermittelt wurden. Für eAnciTrans erzielt FTLR unter Einsatz von WMD + NQK + UCT & MB bestmögliche Ergebnisse in F_1 für eine Konfiguration mit MBs. Die bestmöglichen Ergebnisse von FTLR mit einer Konfiguration ohne MBs stimmen für alle Vergleichsdatensätze mit den für das Projekt insgesamt besten Ergebnissen überein. Bei näherer Betrachtung der Daten in Tabelle 7.6 fällt auf, dass das Einbeziehen von MBs die F_1-Güte der Ergebnisse von FTLR auf allen Vergleichsdatensätzen verschlechtert. Beispielsweise erzielt FTLR in der Konfiguration Kosinusähnlichkeit und ALL + UCT auf eTour Ergebnisse mit 0,524 in F_1. Beim Einbeziehen von MBs in die Eingabe ist nur noch eine Güte von bestenfalls 0,410 in F_1 auf eTour möglich. Im Durchschnitt verschlechtern sich die bestmöglichen Ergebnisse, die FTLR auf einem der Vergleichsdatensätze erzeugt, um 0,062 in F_1 bei Hinzunahme von MBs. Dies deutet stark darauf hin, dass das Einbeziehen von MBs in die Eingabe von UniXcoder die Ergebnisse von FTLR nicht verbessert. Das Gegenteil scheint der Fall zu sein. Dies spricht dafür, Hypothese 5.3 auf allen Vergleichsdatensätzen abzulehnen. Zur Bestätigung der Ungültigkeit von Hypothese 5.3 wurden Wilcoxon-Vorzeichen-Rang-Tests für jeden einzelnen Vergleichsdatensatz durchgeführt. Hierfür wurde die F_1-Güte der FTLR-Ergebnisse für jedes Konfigurationspaar, das sich nur in der Einbeziehung von MBs unterscheidet, auf jedem Vergleichsdatensatz verglichen. Das Ziel bestand darin, zu evaluieren, ob das Einbeziehen von MBs statistisch gesehen zu besseren Ergebnissen von FTLR führt als das Nicht-Einbeziehen von MBs. Der Test führte auf jedem Vergleichsdatensatz zu dem Ergebnis, dass die Nullhypothese zu Hypothese 5.4 mit einer Wahrscheinlichkeit

von über 99% als gültig betrachtet werden kann. Daher lässt sich Hypothese 5.4 auf jedem Vergleichsdatensatz mit hoher Wahrscheinlichkeit als ungültig annehmen. Hypothese 5.4 wurde mit der Begründung aufgestellt, dass das Einbeziehen von mehr semantischen Informationen zu einer genaueren Repräsentation einer Methode führt. Es wurde angenommen, dass dies einen positiven Einfluss auf das Ermitteln von semantischen Ähnlichkeiten zwischen Methoden und Anforderungssätzen hat. Andererseits lässt sich Folgendes anführen. Die Signatur einer Methode beschreibt die Funktionalität einer Methode auf einem sprachlich abstrakten Niveau. Damit ähneln die verwendeten Bezeichner mit hoher Wahrscheinlichkeit den Worten in der zugehörigen Anforderung. Dies führt dazu, dass Methoden und die Anforderungen, auf denen sie basieren, ähnlich repräsentiert werden. Dies ermöglicht ein Ermitteln von TLs durch den Vergleich semantischer Ähnlichkeiten. Im Gegensatz zur Methodensignatur weist der Rumpf einer Quelltext-Methode maximale Implementierungsnähe und damit eine niedrige semantische Abstraktion auf. Im MB erfolgt die konkrete, kleinschrittige Implementierung einer Funktionalität, die durch eine Anforderung beschrieben wird. Die verwendeten Bezeichner und Strukturen im MB beschreiben niedrigabstrakte Abläufe zur Umsetzung eines hochabstrakten Konzepts. Aufgrund des hohen sprachlichen Abstraktionsunterschieds unterscheiden sich die Bezeichner im MB und der zugehörigen Anforderung vermutlich stark. Das Einbeziehen von Bezeichnern aus dem MB kann damit dazu führen, dass Methoden und zugehörige Anforderungen nicht ähnlich genug repräsentiert werden. Dies begründet sich in einer Methodenrepräsentation, die zu stark durch die konkrete, niedrigabstrakte Implementierungs-Semantik beeinflusst ist. Damit ähnelt sie dann der Repräsentation der hochabstrakten Anforderung nicht mehr. Dies kann negative Effekte auf das Erkennen von TLs im Fall von FTLR haben und würde die vorliegende Verschlechterung der Güte der Ergebnisse von FTLR erklären. UniXcoder wurde im Rahmen bisheriger Experimente auf seine Eignung für die NLP-Aufgabe Quelltextsuche untersucht (s. Abschnitt 3.2.2). Auch in diesem Anwendungsfall wird UniXcoder zum Erkennen von semantischen Ähnlichkeiten zwischen natürlicher Sprache und Quelltext eingesetzt. Für eine gegebene natürlichsprachliche Methoden-Beschreibung soll UniXcoder hier die zugehörige Quelltext-Methode auswählen. UniXcoder bezieht auch den MB in die Repräsentation von Methoden ein. Die Tatsache, dass UniXcoder bei dieser Aufgabe vielversprechende Ergebnisse liefert, war ein weiterer Grund für das Aufstellen von Hypothese 5.4. Im Unterschied zum vorliegenden Anwendungsfall haben die natürlichsprachlichen Beschreibungen bei der Quelltextsuche ein niedriges Abstraktionsniveau. Vermutlich enthalten viele sogar konkrete Implementierungsdetails. Damit ähnelt sich das Abstraktionsniveau der natürlichen Sprache und des MBs stark und UniXcoder ermittelt für die Quelltextsuche geeignete Repräsentationen unter Einbeziehung des MBs. Dies lässt sich vermutlich aufgrund des zuvor beschriebenen hohen Abstraktionsunterschieds von Anforderungen und Methoden nicht auf den vorliegenden Anwendungsfall übertragen.

Eine Eingabe, der zusätzlich UCTs und/oder MCs hinzugefügt werden, scheint für UniXcoder hingegen sehr geeignet zu sein. Beispielsweise ermittelt FTLR mit der Konfiguration KOS + ALL + UCT Ergebnisse mit einer Güte von durchschnittlich 0,387 in F_1. Für KOS + NQK + MC ergibt sich ein F_1-Wert von 0,385. Bei Verwendung von WMD, allen Vorverarbeitungsschritten und UCT & MC kommen Ergebnisse mit 0,383 in F_1 zustande. Mithilfe dieser drei Konfigurationen konnten die bestmöglichen Ergebnisse im Experiment erzielt werden. Dies deutet darauf hin, dass die semantischen Informationen aus UCTs und MCs tendenziell zu geeigneten Repräsentationen von Artefakten führen. Hierbei gilt es jedoch zu beachten, dass dieser Effekt nicht auf jedem Vergleichsdatensatz (gleichermaßen) auftritt. FTLR ermittelt auf eTour mithilfe der Kosinusähnlichkeit, allen Vorverarbeitungsschritten und UCTs Ergebnisse, deren Güte 0,524 in F_1 beträgt. Das zusätzliche Einbeziehen von MC lässt die Güte um 0,042 auf 0,482 in F_1 sinken. Ein ähnliches Muster ergibt sich auf eTour unter Verwendung von WMD, ALL und UCT bzw. UCT & MC. Das Erweitern der Eingabe um MCs scheint für eTour nicht immer geeignet zu sein. Andererseits erzielt

	eTour	iTrust	LibEST	SMOSTrans	eAnciTrans	∅
Kosinusähnlichkeit	.524	.307	.594	.367	.272	.413
WMD	.515	.282	.568	.368	.296	.406

Tabelle 7.7: Bestmögliche F_1-Güte der Ergebnisse von FTLR für Konfigurationen mit Kosinusähnlichkeit bzw. WMD

FTLR auf iTrust mit KOS + ALL Ergebnisse mit einer Güte von 0,216 in F_1. Mithilfe von Eingaben, die um MCs erweitert wurden, ergibt sich ein F_1-Wert von 0,233 und damit eine Steigerung um 0,017. Des Weiteren resultiert auf iTrust für KOS + NQK ein F_1-Wert von 0,217, für KOS + NQK + MC ein F_1-Wert von 0,282. Dies deutet darauf hin, dass das Einbeziehen von MCs auf iTrust sehr geeignet ist. Es erfolgt also beispielsweise bei eTour eine Verschlechterung der Güte der Ergebnisse von FTLR bei Einbeziehen von MCs, auf iTrust hingegen eine Verbesserung. Dies ließe sich beispielsweise dadurch erklären, dass die MCs von eTour weniger aussagekräftig sind als die von iTrust bzw. semantisch niedrigabstraktere Implementierungsdetails beschreiben.

In Bezug auf die zweite Dimension der Konfiguration stellt sich nun die Frage. ob sich die Metrik der Kosinusähnlichkeit oder WMD besser zum Ähnlichkeitsvergleich eignet. Hypothese 5.7 besagt, dass die Kosinusähnlichkeit geeigneter ist. Tabelle 7.7 veranschaulicht die Güte der bestmöglichen Ergebnisse von FTLR für Konfiguration mit Kosinusähnlichkeit und für WMD. Bei diesen Werten handelt es sich um die für jeden Datensatz markierten Werte in den Tabellen 7.3 und 7.4. Beispielsweise erzielt FTLR mit der für LibEST best-geeignetsten Konfiguration mit WMD (WMD + NQK + MC) Ergebnisse mit einer Güte von 0,568 in F_1. Es wird deutlich, dass unter Verwendung der Kosinusähnlichkeit auf eTour, iTrust und LibEST bessere Ergebnisse möglich sind als mit WMD. Auf eTour führt die beste Konfiguration mit WMD zu FTLR-Ergebnissen mit einer F_1-Güte von 0,515. Mit Kosinusähnlichkeit und ALL + UCT erzielt FTLR Ergebnisse mit einer um 0,009 in F_1 besseren Güte auf eTour. Auf iTrust verbessert sich die bestmögliche Güte um 0,025 in F_1 und auf LibEST sogar um 0,026 in F_1 bei der Verwendung von Kosinusähnlichkeit im Vergleich zur WMD. Dies deutet darauf hin, dass sich die Kosinusähnlichkeit beim Einsatz von UniXcoder in FTLR auf eTour, iTrust und LibEST besser für den Ähnlichkeitsvergleich eignet als die WMD. Dies lässt vermuten, dass Hypothese 5.7 auf eTour, iTrust und LibEST als gültig angenommen werden kann. Auf LibEST ist der Einsatz der Kosinusähnlichkeit sogar über alle Konfigurationen der Vorverarbeitung hinweg besser geeignet als die Verwendung der WMD. Dies legt die Vermutung nahe, dass sich die Kosinusähnlichkeit auf LibEST signifikant besser zum Ähnlichkeitsvergleich eignet als WMD. Es wurde ein Wilcoxon-Vorzeichen-Rang-Test zur Verifizierung dieser Vermutung und damit zur Bestätigung der Gültigkeit von Hypothese 5.7 auf LibEST durchgeführt. Hierfür wurde die F_1-Güte der FTLR-Ergebnisse auf LibEST für jedes Konfigurationspaar, das sich nur in der verwenden Ähnlichkeitsmetrik unterscheidet, verglichen. Die Nullhypothese zu 5.7 ist auf LibEST mit einer Wahrscheinlichkeit von 0,02% gültig. Damit kann Hypothese 5.7 bei einem Signifikanzniveau von 5% als gültig auf LibEST angenommen werden. Sie wurde in Kapitel 5 aufgestellt, da die Kosinusähnlichkeit im Gegensatz zu WMD auch die Wortfolge in die Artefaktelement-Repräsentation einbezieht. Es wurde angenommen, dass dies zu besonders genauen Artefakt-Element-Repräsentation durch UniXcoder führt. Die Ergebnisse bestätigen diese Hypothese auf eTour, iTrust und insbesondere auf LibEST und damit diese Annahme.

Tabelle 7.7 zeigt jedoch auch, dass auf SMOSTrans und eAnciTrans bessere FTLR-Ergebnisse mit Konfigurationen mit WMD als mit Kosinusähnlichkeit möglich sind. Beispielsweise erzielt FTLR in der auf eAnciTrans besten Konfiguration mit Kosinusähnlichkeit

	eTour		iTrust		LibEST		SMOSTrans		eAnciTrans		
Ansatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	$\varnothing F_1$
KOS + NQK + MC & MB	.386	.376	.171	.179	.581	.625	.338	.385	.224	.145	.340
KOS + ALL + UCT	**.524**	.501	.216	.234	.567	.583	.357	.358	**.272**	.147	**.387**
KOS + NQK + MC	.453	.450	**.307**	.292	**.594**	.666	.340	.400	.230	.148	.385
WMD + ALL + UCT & MC	.496	.475	.230	.243	.550	.564	**.368**	.367	.269	.145	.383

Tabelle 7.8: Güte der Ergebnisse von FTLR für die erwartet beste Konfiguration und die tatsächlich besten Konfigurationen

Ergebnisse mit einer Güte von 0,272 in F_1. Mit WMD und NQK + UCT sind auch eAnciTrans Ergebnisse mit einer F_1-Güte von 0,296 möglich. Die besten FTLR-Ergebnisse unter Verwendung der WMD sind auf SMOSTrans um 0,001 in F_1 besser als die bestmöglichen Ergebnisse mit Kosinusähnlichkeit. Damit scheint Hypothese 5.7 auf SMOSTrans und eAnciTrans keine Gültigkeit zu besitzen. Es fällt jedoch auf, dass es sich bei beiden Datensätzen um übersetze Versionen der ursprünglichen Datensätze handelt. Die Übersetzung wurde für beide Datensätze automatisiert generiert. Es ist davon auszugehen, dass die Datensätze Wort für Wort vom Italienischen ins Englische übersetzt wurden. Damit wurde die Wortreihenfolge der italienischen Worte auch für die englischen Worte übernommen. Es ist zu bezweifeln, dass der allgemeine Satzbau im Italienischen stets mit dem im Englischen übereinstimmt. Damit stehen die übersetzen Worte möglicherweise häufig in einer für die englische Sprache unsinnvollen Reihenfolge. Damit führt das Einbeziehen der Semantik der Wortreihenfolge zu einer ungenauen Repräsentation der Artefakt-Elemente der übersetzen Versionen eAnciTrans und SMOSTrans. Damit ließen sich die schlechteren Ergebnisse von FTLR beim Einsatz der Kosinusähnlichkeit als beim Einsatz der WMD durch die Auswirkungen einer ungenauen Übersetzung erklären. Die Ergebnisse des Experiments auf eTour, iTrust und LibEST hat gezeigt: Stimmt die Sprache der Vortrainingsdaten des UniXcoder-LMs mit der Sprache der originalen Vergleichsdatensätze überein, eignet sich die Kosinusähnlichkeit besser als die WMD. Es lässt sich dann annehmen, dass FTLR mit einem auf italienischen Daten vortrainiertem UniXcoder-LM bessere Ergebnisse auf SMOS und eAnci unter Verwendung der Kosinusähnlichkeit als mit WMD erzeugt. Ein solches ist jedoch bislang nicht öffentlich verfügbar. Im Durchschnitt über alle Vergleichsdatensätze zeigt sich, dass mit der Kosinusähnlichkeit beim Einsatz von UniXcoder bessere FTLR-Ergebnisse möglich sind als mit der WMD (s. Tabelle 7.7). Die Verwendung der Kosinusähnlichkeit führt im Durchschnitt über die bestmöglichen Ergebnisse auf allen Vergleichsdatensätzen zu einer Verbesserung von 0,007 in F_1. Dies spricht für eine Gültigkeit von Hypothese 5.7 über alle Vergleichsdatensätze hinweg. Das durchschnittliche Ergebnis spricht sogar noch stärker für Hypothese 5.7, falls man die Ergebnisse von FTLR auf SMOSTrans und eAnciTrans aufgrund der ungenauen Übersetzung nicht mit einbezieht. Es lässt sich noch ein weiteres Argument für die Gültigkeit von Hypothese 5.7 im Durchschnitt über alle Vergleichsdatensätze hinweg anführen. Die beiden durchschnittlich besten Konfigurationen für den Einsatz von UniXcoder in FTLR verwenden beide die Kosinusähnlichkeit zum Ähnlichkeitsvergleich (s. Tabelle 7.4).

Tabelle 7.8 veranschaulicht erneut die Ergebnisse von FTLR für die Konfiguration Kosinusähnlichkeit + NQK + MC & MB. Es wurde auf Basis der Hypothesen aus Kapitel 5 erwartet, dass diese die beste Konfiguration für den Einsatz von UniXcoder in FTLR darstellt. Die im Rahmen der Evaluation durchgeführten Experimente konnten diese Erwartungshaltung nicht bestätigen. Die tatsächlich besten drei Konfigurationen für UniXcoder sind gegeben durch: Kosinusähnlichkeit und ALL + UCT, Kosinusähnlichkeit und NQK + MC sowie WMD und ALL + UCT & MC (s. Tabelle 7.8). Ihre Verwendung führt zu durchschnittlich mindestens 0,043 in F_1 besseren FTLR-Ergebnisse als der Einsatz der

	eTour		iTrust		LibEST		SMOSTrans		eAnciTrans		
Ansatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	$\varnothing F_1$
WMD	.330	.237	.186	.167	.504	.520	.323	.303	.164	.114	.301
WMD + UCT	**.449**	.409	.186	.167	**.544**	.505	.323	.303	**.210**	.117	**.342**
WMD + MC	.347	.256	**.190**	.179	.526	.559	**.325**	.321	.167	.120	.311
WMD + UCT & MC	.432	.419	**.190**	.179	.535	.588	**.325**	.321	.184	.120	.333
KOS	.277	.269	**.153**	.165	.514	.494	.296	.248	.133	.099	**.275**
KOS + UCT	**.329**	.422	**.153**	.165	.438	.331	**.299**	.263	.132	.084	.270
KOS + MC	.284	.242	.131	.167	**.519**	.594	.296	.257	**.143**	.118	**.275**
KOS + UCT & MC	.287	.377	.131	.167	.438	.331	.298	.262	.132	.084	.257

Tabelle 7.9: Güte der Ergebnisse von FTLR unter Einsatz von Wikipedia2Vec

erwartet besten Konfiguration. In diesem Abschnitt wurde evaluiert, dass der Grund für die deutlich schlechteren Ergebnisse der erwartet besten Konfiguration im Einbeziehen des MBs besteht. Daher wurde Hypothese 5.4 in diesem Zuge auf den Vergleichsdatensätzen falsifiziert. Es konnte außerdem beobachtet werden, dass die Verwendung von NQK statt ALL zur Vorverarbeitung durchschnittlich nur marginale Auswirkungen auf die Güte der Ergebnisse von FTLR hat. Auf den Datensätzen iTrust und LibEST führt sie jedoch zu sichtbar besseren Ergebnissen in F_1 (s. auch Tabelle 7.8). Daher kann Hypothese 5.3 auf iTrust und LibEST als gültig angenommen werden. Der Einsatz der Kosinusähnlichkeit eignet sich bei den unübersetzten Vergleichsdatensätzen besser als die Verwendung der WMD zum Ähnlichkeitsvergleich. Hypothese 5.7 wird damit als gültig auf eTour, iTrust und LibEST angenommen.

7.3.2 Evaluation des Einsatzes von Wikipedia2Vec in FTLR

In diesem Abschnitt soll die Eignung des Einsatzes von Wikipedia2Vec in FTLR evaluiert werden. Analog zu UniXcoder wurden auch für Wikipedia2Vec Entwurfsentscheidungen in Bezug auf sinnvolle Konfigurationen getroffen (s. Kapitel 5). In Kapitel 5 wurde analysiert, dass sich für den Einsatz von Wikipedia2Vec vermutlich dieselben Vorverarbeitungsschritte wie für fastText eignen. Dies begründet sich in einer Ähnlichkeit des Ansatzes beider LMs, den Wikipedia2Vec um SEs erweitert. Daher wurden im Rahmen des Experiments für Wikipedia2Vec alle in Abschnitt 5.3 vorgestellten Schritte durchgeführt. Zusätzlich zu diesen Schritten ist die Durchführung einer WSD für die Worte von Artefakten notwendig. Hypothese 5.5 besagt, dass eine manuelle WSD zu genaueren Ergebnissen von FTLR führt als eine automatisiert durchgeführte. Für den Ähnlichkeitsvergleich kommt der Einsatz der Kosinusähnlichkeit oder der Metrik WMD infrage. Hierbei gilt es zu beachten, dass die Kosinusähnlichkeit im Fall von Wikipedia2Vec auf Vektoren angewendet wird, die durch Aggregation bzw. Durchschnittsbildung der BOE eines Artefakt-Elements ermittelt wurden. Im Fall von UniXcoder war das LM standardmäßig in der Lage, Wortfolgen durch einen einzigen Vektor zu repräsentieren. Laut Hypothese 5.6 ist die WMD besser geeignet als die aggregierte Kosinusähnlichkeit.

Tabelle 7.9 veranschaulicht die Ergebnisse von FTLR unter Einsatz von Wikipedia2Vec für verschiedenste Konfigurationen. Im Anhang finden sich eine Tabelle. die zusätzlich die Werte in den Metriken Präzision und Ausbeute darstellt (s. Abschnitt A.2). Zur Vorverarbeitung werden stets alle bereits bestehenden Vorverarbeitungsschritte eingesetzt (s. Abschnitt 5.3). Darüber hinaus wird stets eine WSD durchgeführt. Aufgrund des hohen zeitlichen Aufwands beim manuellen Erstellen von Bedeutungsauflösungen (s. Abschnitt 5.3) werden für dieses Experiment automatisiert generierte genutzt. Analog zur Evaluation von UniXcoder wurden auch hier MCs und UCTs in allen möglichen Kombinationen einbezogen. Da Wikipedia2Vec nicht auf Quelltext vortrainiert wurde, ist ein Einbeziehen von

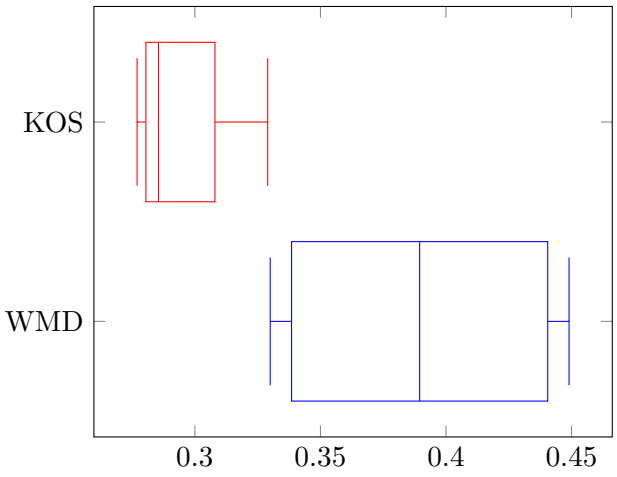


Abbildung 7.1: Güte der Ergebnisse von FTLR auf eTour unter Einsatz von Wikipedia2Vec mit WMD bzw. Kosinusähnlichkeit

MBs nicht sinnvoll. Auch im Rahmen dieses Experiments werden ideale Schwellenwerte verwendet (s. Abschnitt 7.3.1). FTLR ermittelt die besten Ergebnisse auf eTour, LibEST und eAnciTrans unter Verwendung von WMD als Ähnlichkeitsvergleich und unter Einbeziehung von UCTs zur Repräsentation von Anforderungsartefakt-Elementen. Für diese Konfiguration erzielt FTLR auf eTour Ergebnisse mit einer Güte von 0,449 in F_1 (s. Tabelle 7.9). Auf LibEST haben die FTLR-Ergebnisse mit dieser Konfiguration eine F_1-Güte von 0,544 und auf eAnciTrans eine Güte von 0,210. Für iTrust und SMOSTrans ergibt sich die geeignetste Konfiguration durch WMD und MC. FTLR ermittelt für diese Konfiguration Ergebnisse mit einer Güte von 0,190 auf iTrust und 0,325 auf SMOSTrans. Die beste Konfiguration für eTour, LibEST und eAnciTrans ist gleichzeitig auch die durchschnittlich Beste über alle Vergleichsdatensätze hinweg (F_1-Güte von 0,342).

Bei WMD und UCT handelt es sich um eine Konfiguration, die von Hypothese 5.6 impliziert wird. Dies deutet darauf hin, dass der Einsatz von WMD tatsächlich geeigneter ist als der Einsatz der aggregierten Kosinusähnlichkeit. Auch weitere im Rahmen dieses Experiments erhaltene Daten sprechen dafür. Abbildung 7.1 visualisiert die Verteilung der Ergebnisgüte auf eTour für alle Konfigurationen mit WMD bzw. aggregierter Kosinusähnlichkeit. Es wird deutlich, dass selbst die beste Konfiguration mit aggregierter Kosinusähnlichkeit nicht zu besseren FTLR-Ergebnissen führt als die Schlechteste mit WMD. Bei weiterer Betrachtung von Tabelle 7.9 fällt auf, dass sich die WMD bei jeder Konfiguration der Vorverarbeitung auf jedem Vergleichsdatensatz besser in F_1 eignet als die aggregierte Kosinusähnlichkeit. Beispielsweise ermittelt FTLR mit der Konfiguration WMD + MC Ergebnisse mit einer Güte von $0,190$ in F_1 auf iTrust. Beim Einsatz der aggregierten Kosinusähnlichkeit sinkt die Güte bei gleicher Vorverarbeitung (MC) um 0,059 auf 0,131 in F_1 (s. Tabelle 7.9). Die einzige Ausnahme stellt die Konfiguration mit Vorverarbeitung ohne Einbeziehen von MCs oder UCTs auf LibEST dar. Die Ergebnisse sprechen für eine Gültigkeit von Hypothese 5.6 auf allen Vergleichsdatensätzen. Ein Wilcoxon-Vorzeichen-Rang-Test soll dies bestätigen. Zur Durchführung des Tests wurde die F_1-Güte der FTLR-Ergebnisse auf allen Datensätzen einzeln für jedes Konfigurationspaar, das sich nur in der Art des Ähnlichkeitsvergleichs unterscheidet, verglichen. Hiermit sollte untersucht werden, ob der Einsatz von WMD auf jedem Vergleichsdatensatz zu statistisch signifikant besseren Ergebnissen von FTLR führt als der Einsatz der aggregierten Kosinusähnlichkeit. Die Wahrscheinlichkeit die Nullhypothese zu Hypothese 5.3 zurecht abzulehnen wurde für jeden Datensatz als über 99% ermittelt. Bei einem Signifikanzniveau von 5% ist dies ausreichend, um Hypothese 5.3 auf Basis des Tests auf allen Vergleichsdatensätzen als gültig

	WMD		WMD + UCT		WMD + MC		WMD + UCT & MC	
Datensatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP
eTour	.356	.301	**.462**	.489	.346	.275	.444	.462
	KOS		KOS + UCT		KOS + MC		KOS + UCT & MC	
Datensatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP
eTour	.260	.262	**.299**	.389	.251	.243	.285	.342

Tabelle 7.10: Güte der Ergebnisse von FTLR unter Einsatz von Wikipedia2Vec und manueller Bedeutungsauflösung

zu betrachten. Auf Basis dessen lässt sich Hypothese 5.6 mit hoher Wahrscheinlichkeit als wahr annehmen. Diese Hypothese wurde in Abschnitt 5.4 aufgrund der Annahme aufgestellt, dass bei der Aggregation der Vektoren einer BOE durch z.B. Durchschnittsbildung semantische Informationen verloren gehen. Laut der Annahme ist es daher sinnvoller, jede WE der BOE eines Artefakt-Elements im Ähnlichkeitsvergleich einzeln zu betrachten. Damit können alle semantischen Informationen jedes relevanten Wortes unverfälscht gleichermaßen in den Ähnlichkeitsvergleich einbezogen werden. Dies wurde als geeigneter im Fall des Einsatzes von Wikipedia2Vec in FTLR angenommen. Da mithilfe der WMD im Gegensatz zur Kosinusähnlichkeit ein solches einzelnes Einbeziehen der WEs einer BOE möglich ist, wurde die WMD als geeigneter für Wikipedia2Vec als die aggregierte Kosinusähnlichkeit angenommen (s. Hypothese 5.6). Da Hypothese 5.6 im Rahmen der Evaluation auf den Vergleichsdatensätzen bestätigt werden konnte, spricht dies stark für eine Gültigkeit der eben beschriebenen Annahme.

Für das Projekt eTour (s. Abschnitt 7.1) wurde neben einer automatisierten WSD, auch eine manuelle WSD für die Worte von Quelltextartefakten durchgeführt. Tabelle 7.10 veranschaulicht die Ergebnisse, die FTLR unter Einsatz von Wikipedia2Vec und manuell aufgelösten Wortbedeutungen auf eTour erzielt. Eine manuelle WSD scheint die Güte der Ergebnisse von FTLR unter Einsatz der Kosinusähnlichkeit im Vergleich zu einer Automatisierten nicht zu verbessern. Beispielsweise ermittelt FTLR in der Konfiguration automatische WSD und KOS + UCT & MC Ergebnisse mit einer Güte von 0,287 in F_1 auf eTour. Bei Durchführung einer manuellen WSD in derselben Konfiguration der Vorverarbeitungsschritte und des Ähnlichkeitsvergleichs ergibt sich eine Verschlechterung der Ergebnisse um 0,002 auf 0,285 in F_1. Dies lässt sich wie folgt erklären. Hypothese 5.5 nimmt die manuelle WSD als besonders geeignet an, da sie die Bedeutung von Worten im Kontext vermutlich sehr genau bestimmt. Daher wird davon ausgegangen, dass die Semantik der Worte eines Artefakt-Elements in diesem Fall sehr genau repräsentiert werden kann. Bei der aggregierten Kosinusähnlichkeit wird ein Artefakt-Element durch einen Vektor repräsentiert, der sich aus der Durchschnittsbildung der Vektorrepräsentationen aller Worte ergibt. Wie bereits zuvor beschrieben, kann dies zu einem Verlust relevanter semantischer Informationen der Artefakt-Elemente führen. Damit kann die durch die manuelle WSD bestimmte Wortsemantik möglicherweise nicht geeignet in den Ähnlichkeitsvergleich einbezogen werden. Dies erklärt das Ausbleibenden eines positiven Effekts der manuellen WSD durch die verwendete Ähnlichkeitsmetrik. Unter Verwendung der WMD führt eine manuelle WSD zu einer leichten Verbesserung der Ergebnisse von FTLR im Vergleich zu einer Automatisierten. Für beinahe alle Konfigurationen mit WMD ergibt sich eine Verbesserung der F_1-Güte der Ergebnisse von FTLR bei Einsatz einer manuellen WSD. Beispielsweise erzielt FTLR in der Konfiguration automatisierte WSD und WMD + MC Ergebnisse auf eTour mit einem F_1-Wert von $0,449$. Ersetzt man in dieser Konfiguration die automatisierte durch eine manuelle WSD, so erhöht sich die Güte der Ergebnisse um 0,013 auf 0,462 in F_1 (s.

Tabellen 7.9 und 7.10). Allein in der Konfiguration, die ausschließlich MCs in die Eingabe von Wikipedia2Vec einbezieht, verschlechtert die manuelle WSD die Ergebnisse von FTLR leicht im Vergleich zur Automatisierten. FTLR erzielt mit manueller WSD und WMD + MC Ergebnisse mit einer Güte von 0,346 in F_1 auf eTour. Mit automatisierter WSD und derselben Vorverarbeitung haben die Ergebnisse eine Güte von 0,347 in F_1. Dies ließe sich dadurch erklären, dass die MCs von eTour wenig aussagekräftig bzw. semantisch niedrigabstrakt beschreibend sind. Dies wurde bereits im Zuge der Evaluation von UniXcoder vermutet (s. Abschnitt 7.3.1). Mit einer manuellen WSD ist Wikipedia2Vec dann in der Lage, diese niedrigabstrakte Semantik genauer zu erfassen als mit einer Automatisierten. Somit fließt die niedrigabstrakte Semantik bei der Verwendung einer manuellen WSD stärker in die Repräsentation des Quelltextartefakt-Elements ein. Damit unterscheiden sich die ermittelten Repräsentationen von logisch verbundenen niedrigabstrakten Methoden und hochabstrakten Anforderungen stärker als beim Einsatz einer automatisierten WSD. Dies kann sich negativ auf die Güte der Ergebnisse von FTLR auswirken. Auffällig ist, dass eine manuelle WSD die Präzision der Ergebnisse von FTLR unter Einsatz von Wikipedia2Vec und WMD erhöht. Beispielsweise ermittelt FTLR Ergebnisse mit einer MAP von 0,409 mit automatisierter WSD und WMD + UCT. Mit manueller WSD und WMD + UCT ergibt sich eine MAP der Ergebnisse von 0,489 und damit eine Verbesserung von 0,08 in MAP. Dies lässt darauf schließen, dass eine manuelle WSD die Bedeutung von Worten tatsächlich genauer ermittelt. Denn dies würde zu einer genaueren Repräsentation der Semantik ganzer Artefakt-Elemente führen. Die Verwendung von WMD führt dazu, dass sich diese genauere Repräsentation positiv auf die ermittelte Ähnlichkeit tatsächlich verbundener Artefakten auswirkt. Dadurch werden mehr TPs (s. Abschnitt 7.2) ermittelt und die Präzision der Ergebnisse von FTLR steigt. Die Ergebnisse des Experiments deuten aus diesem Grund darauf hin, dass eine manuelle WSD in der Tat zu genaueren Ergebnissen von FTLR führt. Das Experiment für den Vergleich der Art der WSD wurde jedoch nur auf dem Datensatz eTour ausgeführt. Daher ist die Datengrundlage nicht ausreichend für eine Verallgemeinerung der Schlussfolgerung bzw. ein generelles Akzeptieren von Hypothese 5.5. Die Untersuchung liefert jedoch einen starken Indiz für die Gültigkeit von Hypothese 5.5.

7.3.3 Evaluation der Eignung von UniXcoder und Wikipedia2Vec im Vergleich zu fastText

In den beiden vorherigen Abschnitten 7.3.1 und 7.3.2 erfolgte eine separate Evaluation der Eignung von UniXcoder und Wikipedia2Vec für den Einsatz in FTLR. Hierbei lag der Fokus auf der Bestimmung der bestmöglichen Konfiguration für Vorverarbeitung und Ähnlichkeitsvergleich der beiden LMs. In diesem finalen Abschnitt der Evaluation sollen nun die bestmöglichen Ergebnisse von FTLR unter Einsatz von UniXcoder bzw. Wikipedia2Vec mit denen unter Verwendung des bisherigen LMs fastText verglichen werden. In Kapitel 5 wurde analysiert, dass fastText zwei wesentliche Schwachstellen besitzt. Zum einen ist fastText ein statisches LM und ermittelt somit keine kontextsensitiven WEs. Zum anderen ist fastText nicht auf Quelltext vortrainiert. UniXcoder und Wikipedia2Vec eliminieren fastText's Schwachstellen (zum Teil) und besitzen vermutlich keine neuen. Es wird angenommen, dass sie daher geeigneter für den vorliegenden Anwendungsfall als fastText sind. Daraus resultieren die Hypothesen 5.1 und 5.2. Ein Vergleich der Ergebnisse von FTLR unter Einsatz der drei LMs auf Vergleichsdatensätzen soll nun zeigen, ob sich UniXcoder und Wikipedia2Vec tatsächlich besser für den Einsatz in FTLR eignen als fastText. Daraus können Schlussfolgerungen darüber gezogen werden, ob die Schwachstellen von fastText seine Eignung für FTLR tatsächlich negativ beeinflussen.

Tabelle 7.11 und Abbildung 7.2 veranschaulichen die Güte der Ergebnisse von FTLR für UniXcoder, Wikipedia2Vec und fastText auf fünf Vergleichsdatensätzen (s. Abschnitt 7.1).

	eTour		iTrust		LibEST		SMOSTrans		eAnciTrans		
Ansatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	$\varnothing F_1$
UniXcoder Hyp.	.386	.376	.171	.179	.581	.625	.338	.385	.224	.145	.340
UniXcoder 1.	**.524**	.501	.216	.234	.567	.583	.357	.358	.272	.147	.387
UniXcoder 2.	.453	.450	**.307**	.292	**.594**	.666	.340	.400	.230	.148	.385
UniXcoder 3.	.496	.475	.230	.243	.550	.564	.368	.367	.269	.145	.383
Wikipedia2Vec	.449	.409	.186	.167	.544	.505	.323	.303	.210	.117	.342
fastText	.511	.490	.241	.271	.543	.562	.375	.378	.203	.135	.375
							SMOS		eAnci		
							F_1	MAP	F_1	MAP	
fastText	.511	.490	.241	.271	.543	.562	**.427**	.456	**.279**	.146	**.400**

Tabelle 7.11: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder, Wikipedia2Vec und fastText

Im Fall von UniXcoder wird zwischen der Güte der Ergebnisse mit den drei bestmöglichen Konfigurationen („1.“, „2.“, „3.“) und der durch Hypothesen implizierten Konfiguration („Hyp.“) unterschieden. Ein wesentlicher Unterschied zwischen diesen Konfigurationen besteht im fehlenden Einbeziehen von MBs in den tatsächlich besten Konfigurationen. Für Wikipedia2Vec entspricht die erwartete beste auch der tatsächlich besten Konfiguration. Für den Einsatz des fastText-LMs im Experiment werden alle in Abschnitt 5.3 beschriebenen Vorverarbeitungsschritte durchgeführt. Außerdem werden sowohl UCTs als auch MCs in die Eingabe von fastText einbezogen. Mit dieser Konfiguration erzielt fastText auf den Vergleichsdatensätzen die besten durchschnittlichen Ergebnisse [HCWT21]. Eine Besonderheit beim Einsatz von fastText besteht darin, dass ein fastText-LM existiert, welches auf Datensätzen in italienischer Sprache vortrainiert wurde. Dieses kann zur Repräsentation der originalen, unübersetzten Artefakt-Elemente der Datensätze SMOS und eAnci eingesetzt werden. Das auf englisch-sprachlichen Daten vortrainierte fastText-LM kann analog zu dem UniXcoder- und Wikipedia2Vec-LM auf SMOSTrans und eAnciTrans eingesetzt werden. Tabelle 7.11 zeigt die Güte der Ergebnisse, die FTLR mit dem englischsprachlichen fastText-LM auf eTour, iTrust, LibEST, SMOSTrans und eAnciTrans erreicht. Außerdem wird die Güte der Ergebnisse veranschaulicht, die aus dem Einsatz des englisch-sprachlichen LMs auf eTour, iTrust und LibEST und des italienisch-sprachlichen fastText-LM auf SMOS und eAnci resultiert.

Bei Betrachtung von Tabelle 7.11 fällt auf, dass FTLR mit UniXcoder in seiner zweitbesten Konfiguration bessere Ergebnisse auf iTrust und LibEST erzielt als mit fastText. Auf iTrust sind um 0,046 in F_1 bessere Ergebnisse beim Einsatz von UniXcoder möglich, auf LibEST um 0,051 in F_1 bessere Ergebnisse. Auch die MAP der Ergebnisse ist auf beiden Datensätzen unter Verwendung von UniXcoder höher als unter Einsatz von fastText. FTLR erzielt beispielsweise auf LibEST Ergebnisse mit einer MAP von 0,666 mit UniXcoder und Ergebnisse mit einer MAP von 0,562 mit fastText. Daraus lässt sich folgern, dass der Einsatz von UniXcoder in der zweitbesten Konfiguration auf iTrust und LibEST geeigneter für FTLR ist als der Einsatz von fastText. Dies spricht für eine Gültigkeit von Hypothese 5.1 auf iTrust und LibEST. Bei dieser zweitbesten UniXcoder-Konfiguration handelt es sich um Kosinusähnlichkeit und NQK + MC. Die höhere Eignung von UniXcoder führt zu dem Schluss, dass UniXcoder auf iTrust und LibEST geeignetere Repräsentationen von Artefakt-Elementen ermittelt als fastText. Auf beiden Vergleichsdatensätzen beziehen fastText und UniXcoder in dieser Konfiguration MCs in ihre Eingabe ein. Auf LibEST setzt fastText zusätzlich UCTs zur Anforderungsartefakt-Repräsentation ein. FTLR erzielt mit UniXcoder auch in der drittbesten Konfiguration bessere Ergebnisse auf LibEST als mit

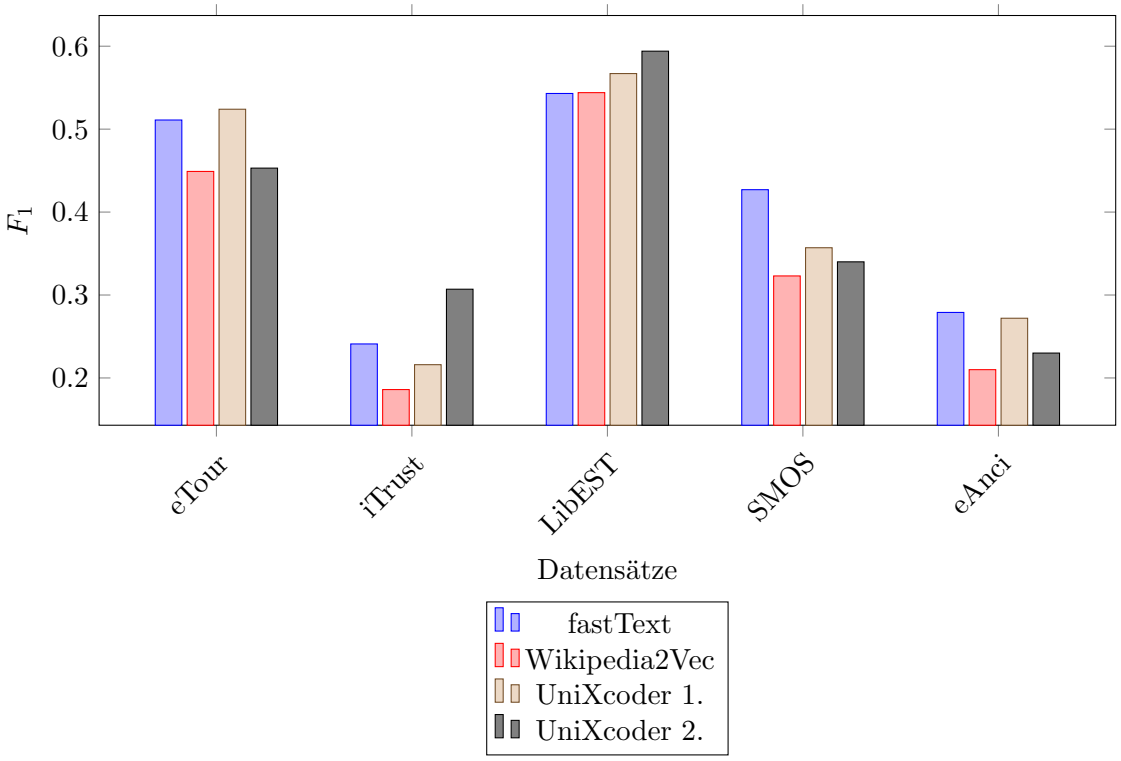


Abbildung 7.2: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder, Wikipedia2Vec und fastText

fastText (s. Tabelle 7.11). In dieser Konfiguration bezieht auch UniXcoder UCTs ein. Somit scheint sich die geeignetere Repräsentation der Artefakt-Elemente durch UniXcoder nicht durch einen unterschiedlichen Umfang der Eingabe in die LMs erklären zu lassen. UniXcoder und fastText setzen auf beiden Vergleichsdatensätzen für ihren Einsatz geeignete Vorverarbeitungsschritte und einen geeigneten Ähnlichkeitsvergleich ein. Eine ungeeignete Vorverarbeitung bzw. ein ungeeigneter Ähnlichkeitsvergleich sind folglich auch keine Erklärung für die schlechteren Ergebnisse von fastText auf iTrust und LibEST.

Damit verbleibt den Erklärungsansatz, dass UniXcoder aufgrund seiner Eigenschaft kontextsensitive WEs zu ermitteln in der Lage ist, auf iTrust und LibEST geeignetere Artefakt-Element-Repräsentationen als fastText zu ermitteln. Daraus folgt, dass die von fastText ermittelten statischen WEs die Bedeutung der Worte in iTrust und LibEST nicht geeignet repräsentieren. Die Ergebnisse des Experiments deuten dann darauf hin, dass die verwendeten Wortbedeutungen im Fall von iTrust und LibEST häufig nicht den dominanten in den Trainingsdaten von fastText verwendeten Wortbedeutungen entsprechen. Dies ließe sich beispielsweise durch stark domänenspezifische Bedeutungen der Worte in iTrust und LibEST erklären. Dies deutet darauf hin, dass fastText vortrainierte WEs ungeeignet sind, falls die Domäne der Softwareprojekte nicht mit der in den Trainingstexten von fastText dominanten Domäne übereinstimmt. Eine kontextsensitive WE durch UniXcoder bringt hier eine Verbesserung der Genauigkeit der Repräsentation, da diese an den vorliegenden Kontext und die dort vorliegenden Domänen-Worte angepasst ist.

FTLR erzielt mit UniXcoder in der zweitbesten Konfiguration keine besseren Ergebnisse auf eTour als fastText (s. Tabelle 7.11). Mit UniXcoder in der zweitbesten Konfiguration haben die FTLR-Ergebnisse auf eTour eine F_1-Güte von 0,453. Unter Einsatz von fastText beträgt die F_1-Güte 0,511. In der zweitbesten Konfiguration werden MCs in die Eingabe von UniXcoder einbezogen. Es wurde bereits in Abschnitt 7.3.1 deutlich, dass die MC von eTour vermutlich eine schlechte Qualität besitzen. Dies würde die schlechteren

FTLR-Ergebnisse von UniXcoder in der zweitbesten Konfiguration im Vergleich zu fastText erklären. Denn fastText bezieht zusätzlich zu MCs auch UCTs in die Eingabe ein. UCT führen auf eTour auch bei UniXcoder zu einer Verbesserung der FTLR-Ergebnisse. In der durchschnittlich besten Konfiguration bezieht UniXcoder ausschließlich UCTs in die Eingabe ein. Dies führt auf eTour zu besseren Ergebnissen als mit der durchschnittlich zweitbesten Konfiguration (s. Tabelle 7.11). Mit UniXcoder in der durchschnittlich besten Konfiguration erzielt FTLR auf eTour sogar bessere Ergebnisse als mit fastText. Da fastText jedoch zusätzlich zu UCTs auch MCs einbezieht, lässt sich dies erneut durch die schlechte Qualität von MCs auf eTour erklären. In der drittbesten Konfiguration bezieht UniXcoder analog zu fastText sowohl UCTs als auch MCs ein. Die FTLR-Ergebnisse bei Einsatz von UniXcoder in dieser Konfiguration übertreffen die Ergebnisse bei Einsatz von fastText nicht in F_1-Güte oder MAP. Damit scheinen die kontextsensitiven WEs von UniXcoder keine geeignetere Repräsentation der Worte von eTour darzustellen als die statischen WEs von fastText. Dies deutet darauf hin, dass die verwendete Wortbedeutung im Fall von eTour häufig auch der dominanten in den Trainingsdaten von fastText entspricht.

Bei SMOS und eAnci handelt es sich um Datensätze, deren Artefakte in italienischer Sprache verfasst sind (s. Abschnitt 7.1). FTLR wird für die ATLR auf SMOS und eAnci beim Einsatz von UniXcoder auf übersetzen Versionen der Vergleichsdatensätze ausgeführt. Beim Einsatz von fastText kann ein auf italienisch-sprachlichen Datensätzen zur ATLR auf den Original-Datensätzen verwendet werden. FTLR erzielt mit UniXcoder in seiner durchschnittlich besten Konfiguration Ergebnisse mit einer F_1-Güte von 0,357 auf SMOSTrans mit einer F_1-Güte von 0,272 auf eAnciTrans. Unter Einsatz des italienisch-sprachlichen fastText-LMs haben die FTLR-Ergebnisse auf SMOS eine Güte von 0,427 in F_1 und auf eAnci eine Güte von 0,279 in F_1. Auf beiden Trainingsdatensätzen haben die bestmöglichen Ergebnisse mit fastText eine höhere Güte als die Ergebnisse mit UniXcoder. Die höheren Ergebnisse mit fastText auf SMOS und eAnci führen dazu, dass FTLR mit fastText bessere Ergebnisse im Durchschnitt über alle Vergleichsdatensätze erreicht als mit UniXcoder. Wird fastText in FTLR eingesetzt und für die ATLR auf SMOS und eAnci das italienisch-sprachliche fastText-LM auf den Original-Datensätzen verwendet, ergibt sich eine durchschnittliche Güte für fastText von 0,400 in F_1. Damit sind diese für fastText bestmöglichen Ergebnisse von FTLR durchschnittlich um 0,013 in F_1 besser als die bestmöglichen Ergebnisse beim Einsatz von UniXcoder. Somit eignet sich UniXcoder im Durchschnitt über alle Vergleichsdatensätze nicht besser als fastText für den Einsatz von FTLR. Damit muss Hypothese 5.1 als ungültig im Hinblick auf den Durchschnitt über alle Vergleichsdatensätze angenommen werden. Wird FTLR unter ausschließlichem Einsatz des englisch-sprachlichen fastText-LMs auf denselben Datensätzen wie UniXcoder ausgeführt, ergibt sich eine Güte der Ergebnisse von durchschnittlich 0,375 in F_1. Damit erzielt UniXcoder auf eTour, iTrust, LibEST, SMOSTrans und eAnciTrans um durchschnittlich 0,012 in F_1 bessere Ergebnisse als fastText (s. Tabelle 7.11). Wird FTLR mit UniXcoder und fastText also auf denselben, zum Teil übersetzen Datensätzen ausgeführt, eignet sich UniXcoder im Durchschnitt besser für den Einsatz in FTLR. Dies legt die Vermutung nahe, dass UniXcoder bei Existenz eines entsprechend auf italienisch-sprachlichen Daten vortrainierten LMs auch im Durchschnitt über eTour, iTrust, LibEST, SMOS und eAnci bessere Ergebnisse erzielt als fastText. In diesem Fall könnte Hypothese 5.1 als gültig auf den Vergleichsdatensätzen betrachtet werden.

Neben der Kontextsensitivität der ermittelten WEs wurde UniXcoder als geeigneter als fastText für den vorliegenden Anwendungsfall angenommen, da das LM auch auf Quelltext vortrainiert wurde. Damit ist es in der Lage auch den MB einer Methode in die Repräsentation eines Quelltextartefakt-Elements einzubeziehen. Eine von fastText ermittelte Quelltextartefakt-Element-Repräsentation unter Einbeziehen des MB eignet sich für die Weiterverarbeitung in FTLR nicht [Che20]. Dies wurde als Schwachstelle von fastText an-

genommen, die zu einer möglicherweise ungenauen Repräsentation von Quelltextartefakt-Elementen führen kann. Tabelle 7.11 veranschaulicht auch die Güte der Ergebnisse von FTLR, die unter Einsatz von UniXcoder in der durch Hypothesen implizierten Konfiguration ermittelt wurden. Hier nutzt UniXcoder auch MBs zur Repräsentation von Quelltextartefakt-Elementen. Für eTour, iTrust, SMOS und eAnci ergeben sich durchweg schlechtere Ergebnisse von FTLR unter Einsatz von UniXcoder in dieser Konfiguration als beim Einsatz von fastText. Beispielsweise erzielt FTLR mit UniXcoder in der besten Konfiguration mit MBs Ergebnisse mit einer Güte von 0,386 in F_1 auf eTour. Mit fastText ohne Einbeziehen von MBs erzielt FTLR auf eTour einen um 0,125 besseren F_1-Wert von 0,511. Alleine für LibEST führt die Verwendung von UniXcoder unter Einbeziehung des MBs zu einer Verbesserung der Ergebnisse von FTLR in F_1 um 0,038. Es zeigte sich schon in der Evaluation von UniXcoder, dass das Einbeziehen des MBs auch hier zu ungeeigneten Repräsentationen führt (s. Abschnitt 7.3.1). Denn hierdurch wird die Repräsentation der Semantik einer Methode vermutlich zu stark durch Implementierungsdetails beeinflusst. Dies führt dazu, dass die Abstraktionskluft zwischen Anforderungs- und Quelltextartefakten nicht geeignet überbrückt werden kann. Die MBs der Quelltextartefakte von LibEST könnten im Gegensatz zu denen der anderen Datensätze ein ähnliches Abstraktionsniveau wie die Anforderungen besitzen. Ihr Einbeziehen würde dann zu einer genaueren und für den Ähnlichkeitsvergleich geeigneteren Repräsentation der Methode führen. Dies würde die besseren FTLR-Ergebnisse auf LibEST beim Einsatz von UniXcoder mit MBs im Vergleich zu fastText erklären. Im Durchschnitt über alle Vergleichsdatensätze scheint die Schwachstelle von fastText, nicht auf Quelltext vortrainiert zu sein, nicht relevant für den Einsatz in FTLR auf den Vergleichsdatensätzen zu sein. Eine Repräsentation einer Methode mithilfe ihrer natürlichsprachlichen Elemente erscheint auch intrinsisch bei UniXcoder geeigneter als eine Repräsentation, die zusätzlich den MB einbezieht.

Mit Wikipedia2Vec als zugrunde liegendem LM erzielt FTLR bestmögliche Ergebnisse mit einer Güte von durchschnittlich 0,342 in F_1 auf den verwendeten Datensätzen (s. Tabelle 7.11). Damit ist auch Wikipedia2Vec nicht in der Lage bessere Ergebnisse als fastText (durchschnittlich 0,400 in F_1) beim Einsatz in FTLR zu erzielen. Für keines der untersuchten Projekte liefert FTLR mit Wikipedia2Vec bessere Ergebnisse als mit fastText. Die einzige Ausnahme stellt der Vergleichsdatensatz LibEST dar, auf dem Wikipedia2Vec die Ergebnisse von FTLR um 0,001 in F_1 im Vergleich zu fastText verbessert. Im Durchschnitt verschlechtert der Einsatz von Wikipedia2Vec die Ergebnisse von FTLR (s. Abbildung 7.2). Beispielsweise ermittelt FTLR mit Wikipedia2Vec auf iTrust bestenfalls Ergebnisse mit einer Güte von 0,186 in F_1. Unter Verwendung von fastText sind die Ergebnisse mit einer Güte von 0,241 in F_1 besser. Diese Ergebnisse deuten damit darauf hin, dass auch Wikipedia2Vec nicht geeigneter für den Einsatz in FTLR ist als fastText. Damit ist Hypothese 5.2 auf den Vergleichsdatensätzen als falsch anzunehmen. Daraus lässt sich schlussfolgern, dass der bedeutungseinbettende Ansatz von Wikipedia2Vec auf den Vergleichsdatensätzen nicht besser als der statische Ansatz von fastText geeignet ist. Scheinbar ist eine automatisierte WSD auf den Projekten häufig nicht in der Lage, die Bedeutung von Worten korrekt zu erfassen. Dann werden Worte mithilfe einer Bedeutung repräsentiert, die sie in diesem Kontext nicht tragen. Dies führt anscheinend zu ungenaueren Wort-Repräsentationen als bei fastText, bei dem die Repräsentation eines Worts durch seine allgemein dominanteste Wortbedeutung beeinflusst wird. Ungenaue Repräsentationen wirken sich negativ auf die Ergebnisse von FTLR aus. Ein möglicher Erklärungsansatz für die unzureichende Güte der Ergebnisse der WSD besteht darin, dass im Fall von Quelltextartefakt-Elementen möglicherweise nicht genug Kontextworte für eine geeignete WSD vorliegen. Zur Repräsentation von Quelltextartefakt-Elementen durch Wikipedia2Vec wird die natürlichsprachliche Methodensignatur, der Klassenname und optional MCs verwendet. Diese sprachlichen Einheiten sind in der Regel stichwortartig verfasst. Somit existieren zu einem gegebenem Wort möglicherweise nicht ausreichend relevante Kontextworte. Verfahren zur WSD kön-

nen in diesem Fall den Kontext des Worts nur ungenau bestimmen, woraus eine ungenaue Wortrepräsentation resultiert. Die Durchführung einer manuellen WSD auf Quelltextseite führte bereits zu Verbesserungen der Güte von FTLR auf eTour (s. Abschnitt 7.3.2). Diese ermittelt Wortbedeutungen genauer als eine Automatisierte. Fortschritte bei Verfahren zur automatisierten WSD, die dazu führen, dass Wortbedeutungen auch bei wenig Kontext geeignet erfasst werden können, könnten somit positive Auswirkungen auf den Einsatz von Wikipedia2Vec in FTLR haben. Auch bei Einsatz der manuellen WSD auf Quelltextseite führt Wikipedia2Vec jedoch nicht zu besseren Ergebnissen als fastText. Dies ließe sich mit den Ungenauigkeiten der automatisierten WSD auf der Anforderungsseite erklären.

Zusammenfassend konnte im Rahmen der Evaluation festgestellt werden, dass sich UniXcoder auf iTrust und LibEST besser eignet als fastText. Dies lässt sich mit seinem Ansatz zur Ermittlung kontextsensitiver WEs begründen, der auf diesen Vergleichsdatensätzen zu geeigneteren Repräsentationen von Artefakt-Elementen führt. Auf iTrust und LibEST kommt fastText's Schwachstelle, statisch zu sein, damit tatsächlich negativ zum Tragen. Im Durchschnitt über alle Vergleichsdatensätze ist UniXcoder nicht besser geeignet als fastText. Dies begründet sich jedoch darin, dass kein auf italienisch-sprachlichen Daten vortrainiertes UniXcoder-LM besteht. Beim Vergleich von UniXcoder und fastText, bei dem beide LMs auf denselben, zum Teil übersetzten Datensätzen ausgeführt werden, eignet sich UniXcoder besser als fastText. Dies legt die Vermutung nahe, dass UniXcoder's kontextsensitiver Ansatz tatsächlich geeigneter ist als der Ansatz von fastText. Ein Einbeziehen des MB in die Eingabe von UniXcoder eignet sich nicht. Eine UniXcoder-Konfiguration mit MB führt im Vergleich zu fastText und zu den besten UniXcoder-Konfigurationen zu schlechteren Ergebnissen auf den Vergleichsdatensätzen. Damit scheint fastText's Schwachstelle, nicht auf Quelltext vortrainiert zu sein, nicht zu ungeeigneteren Repräsentationen von Artefakt-Elementen zu führen. Wikipedia2Vec's Ansatz, Worte durch SE zu repräsentieren, ist nicht geeigneter als fastText's Ansatz auf den Vergleichsdatensätzen. Dies begründet sich vermutlich auf einer ungenauen Ermittlung der Wortbedeutung durch die automatisierte WSD in der Vorverarbeitung.

8 Zusammenfassung und Ausblick

FTLR ist ein Verfahren zur automatisierten nachträglichen Rückverfolgbarkeitsanalyse zwischen Anforderungs- und Quelltextartefakten. Es repräsentiert Elemente von Anforderungen und Quelltext mithilfe des Sprachmodells fastText. Anschließend vergleicht es die Ähnlichkeit der Artefakt-Element-Repräsentationen und ermittelt auf dieser Basis Rückverfolgbarkeitsverbindungen. fastText besitzt Schwachstellen. Das Sprachmodell repräsentiert jedes Wort durch eine statische Worteinbettung. Diese wird häufig in Abhängigkeit der Wortbedeutung ermittelt, die auf den Vortrainingsdaten von fastText dominiert. Wird ein Wort in einem Softwareprojekt, auf dem FTLR ausgeführt wird, mit einer anderen Bedeutung verwendet, repräsentiert fastText das Wort ungenau. fastText wird in FTLR zur Repräsentation von Methoden eingesetzt. Der CommonCrawl-Datensatz, auf dem das verwendete fastText-Sprachmodell vortrainiert wurde, umfasst ausschließlich natürlichsprachliche Trainingsdaten. Daher repräsentiert fastText Methoden nur mithilfe ihrer natürlichsprachlichen Elemente und bezieht den Methodenrumpf in Programmiersprache nicht mit ein. Dieser bestimmt die Funktion und damit die Semantik einer Methode maßgeblich. Daher nutzt fastText bislang nicht alle sematiktragenden Informationen zur Repräsentation einer Methode. Damit kann fastText die Semantik einer Methode eventuell in manchen Fällen nicht geeignet erfassen. Insgesamt ermittelt fastText statische Worteinbettungen und kann den Methodenrumpf aufgrund eines fehlenden Vortrainings auf Quelltext nicht in die Repräsentation von Methoden einbeziehen. Dies kann zu ungenauen Repräsentationen von Anforderungen und Quelltext und damit zu ungenauen Ergebnissen von FTLR führen.

Das Ziel dieser Arbeit bestand daher darin zu untersuchen, ob sich alternative Sprachmodelle besser zum Einsatz in FTLR eignen als fastText. Zunächst wurde analysiert, welche Sprachmodelle sich aufgrund ihrer Eigenschaften eignen. In einem ersten Schritt wurde hierzu betrachtet, welche Sprachmodelle bereits in verwandten Arbeiten erfolgreich eingesetzt wurden. Als nächstes wurde untersucht welche dieser Sprachmodelle Eigenschaften besitzen, die mindestens eine Schwachstelle von fastText eliminieren ohne eine neue hinzuzufügen. Auf Basis dieser Analyse wurden zwei Sprachmodelle mit maximal einer von fastText's Schwachstellen ausgewählt. Das Sprachmodell UniXcoder ist in der Lage, kontextsensitive Worteinbettungen mithilfe seines neuronalen Netzes zu ermitteln. Außerdem wurde es auf dem CodeSearchNet-Datensatz und damit sowohl auf natürlicher Sprache als auch auf Quelltext vortrainiert. Es besitzt somit vermutlich keine der beiden Schwachstellen von fastText. Das Sprachmodell Wikipedia2Vec ist ein bedeutungseinbettendes Sprachmodell. Es basiert auf der Wikipedia-Wissensdatenbank, die für jedes Wort

alle vorhandenen Wortbedeutungen speichert. Nachdem die verwendete Bedeutung eines Wortes im Kontext ermittelt wurde, kann Wikipedia2Vec diese Bedeutung und das Wort durch eine Bedeutungseinbettung kontextsensitiv repräsentieren. Wikipedia2Vec besitzt damit fastText's Schwachstelle der statischen Worteinbettungen nicht. Von UniXcoder und Wikipedia2Vec wurde erwartet, dass sie sich zum Einsatz in FTLR besser eignen als fastText.

Im Anschluss an die Auswahl der zu untersuchenden Sprachmodelle wurden Möglichkeiten ihrer Integration in FTLR betrachtet. Die Analyse konzentrierte sich hierbei insbesondere darauf herauszufinden, welche Vorverarbeitung und welcher Ähnlichkeitsvergleich für die Sprachmodelle geeignet ist. Es wurde angenommen, dass sich für UniXcoder die alleinige Durchführung der Vorverarbeitungsschritte Nichtbuchstabenfilterung, Querverweisfilterung und Kleinbuchstaben-Transformation am besten eignet. Außerdem wurde vermutet, dass sich das Einbeziehen des Methodenrumpfs in die Eingabe von UniXcoder für die Repräsentation von Quelltextartefakt-Elementen eignet. Die Verwendung der Kosinusähnlichkeit für den Ähnlichkeitsvergleich erschien besonders vielversprechend. Denn sie ermöglicht den Vergleich zweier Vektorrepräsentationen von Artefakt-Elementen, in die die Semantik aller Worte und die Semantik der Wortreihenfolge eingeflossen ist. Des weiteren wurde analysiert, dass für Wikipedia2Vec die Durchführung derselben Vorverarbeitungsschritte, die auch fastText nutzt, nötig sind. Denn beiden liegt ein ähnliches Vortraining zu Grunde. Außerdem müssen Wortbedeutungen für die Artefaktworte vorhanden sein, welche entweder manuell oder automatisiert zur Verfügung gestellt werden können. Es wurde angenommen, dass sich die Wortüberführungsdistanz besonders zum Ähnlichkeitsvergleich für Wikipedia2Vec eignet. Denn sie ermöglicht einen wortweisen Vergleich der ermittelten Artefakt-Element-Repräsentationen.

Die Evaluation des Einsatzes von UniXcoder auf fünf Vergleichsdatensätzen ergab, dass sich die erwartete beste Vorverarbeitung im Durchschnitt über alle Testprojekte ähnlich gut wie die bei fastText eingesetzte Vorverarbeitung für UniXcoder eignet. Das Einbeziehen des Methodenrumpfs in die Sprachmodell-Eingabe eignet sich auch für den Einsatz von UniXcoder nicht. Die Annahme, dass sich die Kosinusähnlichkeit im Durchschnitt über die Testdatensätze besser zum Ähnlichkeitsvergleich als die Wortüberführungsdistanz, konnte bestätigt werden. Auf den Testdatensätzen iTrust und LibEST eignet sich der Einsatz von UniXcoder in FTLR tatsächlich besser als der von fastText. Dies lässt sich auf die kontextsensitive Eigenschaft von UniXcoder zurückführen. FTLR erzielt mit UniXcoder bestenfalls Ergebnisse mit einer Güte von 0,387 in F_1 im Durchschnitt über alle Testdatensätze. Mit fastText sind Ergebnisse mit durchschnittlich 0,400 F_1-Güte möglich. Damit eignet sich UniXcoder im Durchschnitt über alle Testdatensätze nicht besser als fastText für den Einsatz in FTLR. Bei der Evaluation des Einsatzes von Wikipedia2Vec wurde herausgefunden, dass sich eine manuelle Bedeutungsauflösung auf eTour besser eignet als eine automatisierte. Selbst mit einer manuelle Bedeutungsauflösung führt Wikipedia2Vec jedoch nicht zu besseren Ergebnissen als UniXcoder oder fastText auf eTour. Außerdem eignet sich die Wortüberführungsdistanz besser zum Ähnlichkeitsvergleich als die aggregierte Kosinusähnlichkeit. Allerdings erzielt FTLR mit UniXcoder bestenfalls Ergebnisse mit einer Güte von 0,342 in F_1 im Durchschnitt über alle Testdatensätze. Wikipedia2Vec eignet sich auf keinem der Testdatensätze und auch nicht im Durchschnitt über alle Testdatensätze besser als fastText für den Einsatz in FTLR. Insgesamt wurde geschlussfolgert, dass sich der Ansatz von UniXcoder zur Ermittlung kontextsensitiver Worteinbettungen auf zwei Testdatensätzen tatsächlich besser eignet als der statische Ansatz von fastText. Die Tatsache, dass sich fastText nicht für die Repräsentation des Methodenrumpfs eignet, stellt nur auf LibEST eine Schwachstelle dar. Hier ermittelt FTLR mit UniXcoder auch unter Einbeziehung des Methodenrumpfs bessere Ergebnisse als mit fastText. UniXcoder ermittelt jedoch auf jedem Vergleichsdatensatz bessere Ergebnisse ohne den Methoden-

rumpf als mit ihm. Damit ist das Einbeziehen des Methodenrumpfs auch beim Einsatz von UniXcoder nicht hilfreich.

Zwei der Testdatensätze (SMOS und eAnci) sind in italienischer Sprache verfasst. Im Rahmen dieser Arbeit stand kein auf italienisch-sprachlichen Daten vortrainiertes UniXcoder-LM zur Verfügung. Damit musste für die ATLR mit UniXcoder für diese zwei Testdatensätze auf einer übersetzten Version erfolgen. Für die ATLR mit fastText konnte ein auf italienisch-sprachlichen Daten vortrainiertes fastText-LM verwendet werden. Eine ungenaue Übersetzung könnte einen Grund dafür darstellen, dass UniXcoder auf SMOS und eAnci zu schlechteren Ergebnissen führte als fastText. Die schlechteren Ergebnisse auf SMOS und eAnci bedingten die durchschnittlich schlechteren Ergebnisse. Folgende Arbeiten könnten UniXcoder auf italienisch-sprachlichen Daten vortrainieren und das resultierende LM auf den originalen Versionen von SMOS und eAnci einsetzen. Dies könnte aufzeigen, dass sich UniXcoder auch auf nicht-englisch-sprachlichen Datensätzen besser eignet als fastText. Dies würde vermuten lassen, dass sich die kontextsensitiven WEs von UniXcoder auch allgemein besser für FTLR eigenen als die statischen WEs von fastText.

Literaturverzeichnis

[ACC+02] Antoniol, G. ; Canfora, G. ; Casazza, G. ; De Lucia, A. ; Merlo, E.: Recovering traceability links between code and documentation. In: *IEEE Transactions on Software Engineering* 28 (2002), Nr. 10, S. 970–983. http://dx.doi.org/10.1109/TSE.2002.1041053. – DOI 10.1109/TSE.2002.1041053 (zitiert auf Seite 19).

[BKL09] Bird, Steven ; Klein, Ewan ; Loper, Edward: *Natural language processing with Python: analyzing text with the natural language toolkit*. Ö'Reilly Media, Inc.", 2009 (zitiert auf Seite 49).

[BLB+13] Buitinck, Lars ; Louppe, Gilles ; Blondel, Mathieu ; Pedregosa, Fabian ; Mueller, Andreas ; Grisel, Olivier ; Niculae, Vlad ; Prettenhofer, Peter ; Gramfort, Alexandre ; Grobler, Jaques ; Layton, Robert ; VanderPlas, Jake ; Joly, Arnaud ; Holt, Brian ; Varoquaux, Gaël: API design for machine learning software: experiences from the scikit-learn project. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, S. 108–122 (zitiert auf Seite 56).

[BTT02] Bernabeu, E.J. ; Tornero, Josep ; Tomizuka, Masayoshi: A navigation system for unmanned vehicles in automated highway systems, 2002. – ISBN 0–7803–7398–7, S. 696 – 701 vol.1 (zitiert auf Seite 21).

[c2n22] c2nes: *javalang*. https://github.com/c2nes/javalang. Version: 2022 (zitiert auf Seite 50).

[Che20] Chen, Fei: *Anforderungen-zu-Quelltextrückverfolgbarkeit mittels Wort- und Quelltexteinbettungen*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Master's Thesis, 2020. https://code.ipd.kit.edu/hey/indirect/wikis/Theses/chen_ma (zitiert auf den Seiten 32 und 77).

[CHGZ12] Cleland-Huang, Jane (Hrsg.) ; Gotel, Orlena (Hrsg.) ; Zisman, Andrea (Hrsg.): *Software and Systems Traceability*. London : Springer London, 2012. http://dx.doi.org/10.1007/978-1-4471-2239-5. http://dx.doi.org/10.1007/978-1-4471-2239-5. – ISBN 978–1–4471–2238–8 978–1–4471–2239–5 (zitiert auf Seite 1).

[CHP21] Chatzilygeroudis, Konstantinos ; Hatzilygeroudis, Ioannis ; Perikos, Isidoros: Machine learning basics. In: *Intelligent Computing for Interactive System Design: Statistics, Digital Signal Processing, and Machine Learning in Practice*. 2021, S. 143–193 (zitiert auf den Seiten 11 und 12).

[CWWW19] Chen, Lei ; Wang, Dandan ; Wang, Junjie ; Wang, Qing: Enhancing Unsupervised Requirements Traceability with Sequential Semantics. In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, S. 23–30 (zitiert auf Seite 25).

[DCLT19] Devlin, Jacob ; Chang, Ming-Wei ; Lee, Kenton ; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *ArXiv* abs/1810.04805 (2019) (zitiert auf Seite 17).

[Eli22] Eliben: *pycparser*. `https://github.com/eliben/pycparser`. Version: 2022 (zitiert auf Seite 50).

[FGT+20] Feng, Zhangyin ; Guo, Daya ; Tang, Duyu ; Duan, Nan ; Feng, Xiaocheng ; Gong, Ming ; Shou, Linjun ; Qin, Bing ; Liu, Ting ; Jiang, Daxin ; Zhou, Ming: CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online : Association for Computational Linguistics, November 2020, 1536–1547 (zitiert auf Seite 21).

[GBG+18] Grave, Edouard ; Bojanowski, Piotr ; Gupta, Prakhar ; Joulin, Armand ; Mikolov, Tomas: Learning Word Vectors for 157 Languages. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. Miyazaki, Japan : European Language Resources Association (ELRA), Mai 2018 (zitiert auf den Seiten 28 und 32).

[GCHH+12] Gotel, Orlena ; Cleland-Huang, Jane ; Hayes, JaneHuffman ; Zisman, Andrea ; Egyed, Alexander ; Grünbacher, Paul ; Dekhtyar, Alex ; Antoniol, Giuliano ; Maletic, Jonathan ; Mäder, Patrick: *Traceability Fundamentals*. 2012. – 3–22 S. `http://dx.doi.org/10.1007/978-1-4471-2239-5_1`. `http://dx.doi.org/10.1007/978-1-4471-2239-5_1`. – ISBN 9781447122388 (zitiert auf Seite 3).

[GLD+22] Guo, Daya ; Lu, Shuai ; Duan, Nan ; Wang, Yanlin ; Zhou, Ming ; Yin, Jian: UniXcoder: Unified Cross-Modal Pre-training for Code Representation, 2022, S. 7212–7225 (zitiert auf Seite 22).

[HCWT21] Hey, Tobias ; Chen, Fei ; Weigelt, Sebastian ; Tichy, Walter F.: Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, S. 12–22 (zitiert auf den Seiten 1, 27, 31, 60 und 75).

[Hen08] Henrich, Andreas: Information Retrieval 1. (2008), S. 421 (zitiert auf Seite 5).

[Hey19] Hey, Tobias: INDIRECT: Intent-Driven Requirements-to-Code Traceability. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, S. 190–191 (zitiert auf Seite 52).

[HRK15] Hill, Felix ; Reichart, Roi ; Korhonen, Anna: SimLex-999: Evaluating Semantic Models With (Genuine) Similarity Estimation. In: *Computational Linguistics* 41 (2015), Dezember, Nr. 4, 665–695. `http://dx.doi.org/10.1162/COLI_a_00237`. – DOI 10.1162/COLI_a_00237 (zitiert auf Seite 23).

[HWG+19] Husain, Hamel ; Wu, Ho-Hsiang ; Gazit, Tiferet ; Allamanis, Miltiadis ; Brockschmidt, Marc: CodeSearchNet challenge: Evaluating the state of semantic code search. In: *arXiv preprint arXiv:1909.09436* (2019) (zitiert auf den Seiten 21, 22, 35 und 36).

[ID10] Indurkhya, Nitin ; Damerau, Fred J.: *Handbook of Natural Language Processing*. 2nd. Chapman & Hall/CRC, 2010. – ISBN 1420085921 (zitiert auf Seite 9).

[JGBM16] Joulin, Armand ; Grave, Edouard ; Bojanowski, Piotr ; Mikolov, Tomas: *Bag of Tricks for Efficient Text Classification*. http://dx.doi.org/10.48550/arXiv.1607.01759. Version: 2016. – type: article (zitiert auf Seite 1).

[JM22] Jurafsky, Daniel ; Martin, James H.: *Speech and Language Processing*. 3. ed. draft. 2022 https://web.stanford.edu/~jurafsky/slp3/ (zitiert auf den Seiten 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 33 und 61).

[Kap12] Kapitanova, S Krasimira {and} S. Krasimira {and} Son: Machine learning basics. (2012), S. 3–29 (zitiert auf Seite 11).

[LCH13] Li, Yonghua ; Cleland-Huang, Jane: Ontology-based trace retrieval. In: *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, 2013, S. 30–36. – ISSN: 2157-2194 (zitiert auf Seite 20).

[LD] Ladani, Dhara J. ; Desai, Nikita P.: Stopword Identification and Removal Techniques on TC and IR applications: A Survey. In: *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, S. 466–472. – ISSN: 2575-7288 (zitiert auf Seite 10).

[Lid01] Liddy, Elizabeth D.: Natural language processing. (2001) (zitiert auf Seite 9).

[LM] Le, Quoc V. ; Mikolov, Tomas: *Distributed Representations of Sentences and Documents*. http://dx.doi.org/10.48550/arXiv.1405.4053. – type: article (zitiert auf Seite 24).

[M+10] McKinney, Wes u. a.: Data structures for statistical computing in python. In: *Proceedings of the 9th Python in Science Conference* Bd. 445 Austin, TX, 2010, S. 51–56 (zitiert auf Seite 55).

[MCCD13] Mikolov, Tomas ; Chen, Kai ; Corrado, Greg ; Dean, Jeffrey: *Efficient Estimation of Word Representations in Vector Space*. http://dx.doi.org/10.48550/arXiv.1301.3781. Version: 2013. – type: article (zitiert auf den Seiten 1, 15 und 16).

[MGB+18] Mikolov, Tomas ; Grave, Edouard ; Bojanowski, Piotr ; Puhrsch, Christian ; Joulin, Armand: Advances in Pre-Training Distributed Word Representations. In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018 (zitiert auf Seite 15).

[Mic22] Microsoft: *unixcoder-base*. https://huggingface.co/microsoft/unixcoder-base. Version: 2022 (zitiert auf Seite 54).

[Mit97] Mitchell, T.M.: *Machine Learning*. McGraw-Hill, 1997 (McGraw-Hill International Editions). https://books.google.de/books?id=EoYBngEACAAJ. – ISBN 9780071154673 (zitiert auf Seite 11).

[MM03] Marcus, A. ; Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, S. 125–135. – ISSN: 0270-5257 (zitiert auf Seite 20).

[MRS08] Manning, Christopher D. ; Raghavan, Prabhakar ; Schutze, Hinrich: Introduction to Information Retrieval. (2008), S. 506 (zitiert auf Seite 4).

[Nav09] Navigli, Roberto: Word sense disambiguation: A survey. In: *ACM computing surveys (CSUR)* 41 (2009), Nr. 2, S. 1–69 (zitiert auf Seite 9).

[NM90] Näher, Stefan ; Mehlhorn, Kurt: LEDA: A Library of Efficient Data Types and Algorithms. In: *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. Berlin, Heidelberg : Springer-Verlag, 1990. – ISBN 0387528261, S. 1–5 (zitiert auf Seite 20).

[PGM+19] Paszke, Adam ; Gross, Sam ; Massa, Francisco ; Lerer, Adam ; Bradbury, James ; Chanan, Gregory ; Killeen, Trevor ; Lin, Zeming ; Gimelshein, Natalia ; Antiga, Luca ; Desmaison, Alban ; Kopf, Andreas ; Yang, Edward ; DeVito, Zachary ; Raison, Martin ; Tejani, Alykhan ; Chilamkurthy, Sasank ; Steiner, Benoit ; Fang, Lu ; Bai, Junjie ; Chintala, Soumith: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, S. 8024–8035 (zitiert auf Seite 54).

[PW09] Pele, Ofir ; Werman, Michael: Fast and robust earth mover's distances. In: *2009 IEEE 12th International Conference on Computer Vision* IEEE, 2009, S. 460–467 (zitiert auf Seite 57).

[SBMN13] Socher, Richard ; Bauer, John ; Manning, Christopher D. ; Ng, Andrew Y.: Parsing with Compositional Vector Grammars. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Sofia, Bulgaria : Association for Computational Linguistics, August 2013, 455–465 (zitiert auf Seite 21).

[SPN20] Scarlini, Bianca ; Pasini, Tommaso ; Navigli, Roberto: SensEmBERT: Context-Enhanced Sense Embeddings for Multilingual Word Sense Disambiguation. In: *Proceedings of the Thirty-Fourth Conference on Artificial Intelligence*, Association for the Advancement of Artificial Intelligence, 2020, S. 8758–8765 (zitiert auf den Seiten 17 und 18).

[SSTC] Software & Systems Traceability (CoEST), Center of Excellence f.: *Datasets*. `http://sarec.nd.edu/coest/datasets.html` (zitiert auf den Seiten 25, 29 und 60).

[YAS+20] Yamada, Ikuya ; Asai, Akari ; Sakuma, Jin ; Shindo, Hiroyuki ; Takeda, Hideaki ; Takefuji, Yoshiyasu ; Matsumoto, Yuji: Wikipedia2Vec: An Efficient Toolkit for Learning and Visualizing the Embeddings of Words and Entities from Wikipedia. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Association for Computational Linguistics, 2020, S. 23–30 (zitiert auf den Seiten 18, 23 und 55).

[YDY+19] Yang, Zhilin ; Dai, Zihang ; Yang, Yiming ; Carbonell, Jaime ; Salakhutdinov, Russ R. ; Le, Quoc V.: XLNet: Generalized Autoregressive Pretraining for Language Understanding. In: Wallach, H. (Hrsg.) ; Larochelle, H. (Hrsg.) ; Beygelzimer, A. (Hrsg.) ; Alché-Buc, F. d. (Hrsg.) ; Fox, E. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 32, Curran Associates, Inc., 2019 (zitiert auf Seite 22).

[YSTT16] Yamada, Ikuya ; Shindo, Hiroyuki ; Takeda, Hideaki ; Takefuji, Yoshiyasu: Joint Learning of the Embedding of Words and Entities for Named Entity Disambiguation. In: *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, Association for Computational Linguistics, 2016, S. 250–259 (zitiert auf Seite 54).

[ZCS17] Zhao, Teng ; Cao, Qinghua ; Sun, Qing: An Improved Approach to Traceability Recovery Based on Word Embeddings. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, S. 81–89 (zitiert auf Seite 24).

[ZKZ+15] Zhu, Y. ; Kiros, R. ; Zemel, R. ; Salakhutdinov, R. ; Urtasun, R. ; Torralba, A. ; Fidler, S.: Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA : IEEE Computer Society, dec 2015. – ISSN 2380–7504, 19-27 (zitiert auf den Seiten 17, 23 und 36).

[ZTGH21] Zhang, Meng ; Tao, Chuanqi ; Guo, Hongjing ; Huang, Zhiqiu: Recovering Semantic Traceability between Requirements and Source Code Using Feature Representation Techniques. In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, S. 873–882 (zitiert auf Seite 25).

Anhang

A Ergebnisse der Evaluation

A.1 UniXcoder

Die folgenden Tabellen A.1 und A.2 veranschaulichen Evaluationsergebnisse beim Einsatz von UniXcoder in FTLR auf Vergleichsdatensätzen. Sie zeigen zusätzlich zur F_1-Güte und MAP der Ergebnisse auch die Werte für Präzision und Ausbeute.

Tabelle A.3 zeigt die Güte der Ergebnisse von FTLR beim Einsatz von UniXcoder in den besten drei Konfigurationen unter Einbeziehung von Datentypen.

Ansatz	eTour				iTrust				LibEST				SMOS				eAnci				∅F_1
	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	
WMD + NQK	.455	.360	.402	.393	.243	.196	.217	.258	.393	.912	.550	.642	.238	.415	.303	.343	.227	.295	.256	.152	.346
WMD + NQK + UCT	.439	.455	.447	.489	.243	.196	.217	.258	.404	.873	.552	.629	.229	.528	.320	.354	.277	.317	**.296**	.173	.366
WMD + NQK + MC	.476	.416	.444	.408	.353	.234	**.282**	.275	.452	.765	**.568**	.653	.261	.441	.328	.384	.236	.261	.248	.148	.374
WMD + NQK + MB	.382	.403	.392	.360	.154	.168	.160	.169	.484	.662	.559	.622	.254	.397	.310	.339	.229	.280	.252	.148	.335
WMD + NQK + UCT & MC	.410	.468	.437	.474	.353	.234	**.282**	.275	.456	.745	.566	.649	.263	.506	.346	.397	.252	.284	.267	.158	.380
WMD + NQK + UCT & MB	.440	.370	.402	.402	.154	.168	.160	.169	.420	.873	.567	.605	.236	.480	.317	.354	.258	.298	.277	.162	.345
WMD + NQK + MC & MB	.438	.386	.410	.380	.165	.189	.176	.177	.405	.858	.550	.610	.256	.433	.322	.362	.248	.236	.242	.144	.340
WMD + NQK + UCT & MC & MB	.402	.367	.384	.395	.165	.189	.176	.177	.404	.868	.551	.619	.255	.518	.342	.382	.250	.247	.248	.154	.340
WMD + ALL	.402	.422	.412	.372	.176	.287	.218	.217	.425	.740	.540	.603	.226	.544	.320	.344	.232	.243	.238	.126	.346
WMD + ALL + UCT	.469	.571	**.515**	.504	.176	.287	.218	.217	.371	.946	.533	.588	.260	.559	.355	.350	.282	.277	.280	.143	.380
WMD + ALL + MC	.355	.471	.404	.343	.197	.276	.230	.243	.425	.794	.554	.589	.305	.365	.332	.359	.210	.220	.215	.132	.347
WMD + ALL + MB	.376	.321	.347	.348	.102	.224	.140	.118	.484	.662	.559	.619	.273	.387	.320	.327	.154	.257	.192	.123	.312
WMD + ALL + UCT & MC	.452	.549	.496	.475	.197	.276	.230	.243	.552	.549	.550	.564	.271	.573	**.368**	.367	.271	.266	.269	.145	**.383**
WMD + ALL + UCT & MB	.464	.442	.453	.446	.102	.224	.140	.118	.466	.662	.547	.568	.258	.517	.344	.339	.241	.222	.231	.137	.343
WMD + ALL + MC & MB	.360	.357	.359	.319	.154	.129	.140	.121	.418	.794	.547	.591	.279	.405	.330	.344	.120	.351	.179	.127	.311
WMD + ALL + UCT & MC & MB	.465	.432	.448	.415	.154	.129	.140	.121	.394	.824	.533	.563	.272	.495	.351	.355	.159	.261	.198	.136	.334

Tabelle A.1: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder und WMD

Ansatz	eTour				iTrust				LibEST				SMOSTrans				eAnciTrans				
	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	$\varnothing F_1$
NQK	.543	.370	.440	.477	.298	.206	.242	.239	.449	.824	.581	.635	.276	.355	.311	.349	.150	.390	.216	.146	.358
NQK + UCT	.355	.468	.404	.435	.298	.206	.242	.239	.445	.809	.574	.631	.194	.782	.311	.305	.253	.275	.264	.170	.359
NQK + MC	.528	.396	.453	.450	.421	.241	**.307**	.292	.483	.770	**.594**	.666	.292	.407	.340	.400	.182	.309	.230	.148	.385
NQK + MB	.359	.438	.395	.403	.123	.269	.169	.161	.463	.794	.585	.636	.235	.489	.318	.340	.169	.349	.228	.147	.339
NQK + UCT & MC	.311	.390	.346	.372	.421	.241	**.307**	.292	.485	.721	.580	.652	.204	.723	.318	.345	.202	.302	.242	.152	.359
NQK + UCT & MB	.348	.302	.323	.350	.123	.269	.169	.161	.456	.780	.575	.634	.188	.784	.304	.294	.249	.257	.253	.160	.325
NQK + MC & MB	.402	.370	.386	.376	.127	.262	.171	.179	.467	.770	.581	.625	.273	.443	.338	.385	.219	.229	.224	.145	.340
NQK + UCT & MC & MB	.324	.302	.313	.291	.127	.262	.171	.179	.406	.902	.560	.620	.205	.663	.313	.325	.218	.287	.248	.152	.321
ALL	.431	.425	.428	.409	.178	.276	.216	.234	.452	.760	.567	.600	.289	.387	.331	.354	.261	.228	.243	.152	.353
ALL + UCT	.505	.545	**.524**	.501	.178	.276	.216	.234	.442	.789	.567	.583	.263	.553	.357	.358	.258	.287	**.272**	.147	**.387**
ALL + MC	.369	.464	.411	.373	.220	.248	.233	.239	.456	.809	.583	.561	.316	.382	.346	.369	.152	.346	.211	.141	.357
ALL + MB	.400	.334	.364	.348	.083	.224	.121	.114	.459	.790	.580	.636	.265	.433	.329	.336	.211	.286	.243	.148	.327
ALL + UCT & MC	.478	.487	.482	.461	.220	.248	.233	.239	.423	.853	.566	.531	.278	.539	**.367**	.376	.210	.261	.234	.143	.376
ALL + UCT & MB	.500	.383	.434	.411	.083	.224	.121	.114	.490	.716	.582	.600	.259	.482	.337	.345	.208	.284	.240	.145	.343
ALL + MC & MB	.313	.425	.361	.323	.093	.185	.124	.109	.442	.750	.556	.550	.276	.458	.344	.356	.176	.265	.212	.143	.312
ALL + UCT & MC & MB	.392	.425	.408	.389	.093	.185	.124	.109	.393	.838	.535	.524	.269	.501	.350	.363	.227	.203	.214	.139	.326

Tabelle A.2: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder und Kosinusähnlichkeit

	eTour		iTrust		LibEST		SMOSTrans		eAnciTrans		
Ansatz	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	F_1	MAP	$\varnothing F_1$
UniXcoder 1.	.521	.493	.222	.238	.560	.581	.353	.355	.285	.147	.388
UniXcoder 2.	.443	.427	.297	.291	.584	.657	.338	.400	.233	.149	.379
UniXcoder 3.	.510	.500	.231	.248	.548	.551	.363	.364	.257	.151	.382

Tabelle A.3: Güte der Ergebnisse von FTLR unter Einsatz von UniXcoder unter Einbeziehung der Datentypen

A.2 Wikipedia2Vec

Die folgende Tabelle A.4 veranschaulicht Evaluationsergebnisse beim Einsatz von Wikipedia2Vec in FTLR auf Vergleichsdatensätzen. Sie zeigt zusätzlich zur F_1-Güte und MAP der Ergebnisse auch die Werte für Präzision und Ausbeute.

Ansatz	eTour				iTrust				LibEST				SMOSTrans				eAnciTrans				
	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	Prä	Aus	F_1	MAP	$\varnothing F_1$
WMD	.352	.312	.330	.237	.198	.175	.186	.167	.386	.725	.504	.520	.248	.464	.323	.303	.098	.487	.164	.114	.301
WMD + MC	.354	.341	.347	.256	.171	.213	**.190**	.179	.382	.843	.526	.559	.241	.499	**.325**	.321	.139	.210	.167	.120	.311
WMD + UCT	.492	.412	**.449**	.409	.198	.175	.186	.167	.421	.770	**.544**	.505	.248	.464	.323	.303	.149	.354	**.210**	.117	**.342**
WMD + UCT & MC	.486	.390	.432	.419	.171	.213	**.190**	.179	.399	.814	.535	.588	.241	.499	**.325**	.321	.135	.291	.184	.120	.333
KOS	.235	.338	.277	.269	.136	.175	**.153**	.165	.375	.814	.514	.494	.204	.538	.296	.248	.071	.942	.133	.099	**.275**
KOS + MC	.233	.364	.284	.242	.100	.210	.131	.167	.406	.721	**.519**	.594	.208	.516	.296	.257	.135	.153	**.143**	.118	**.275**
KOS + UCT	.263	.438	**.329**	.422	.136	.175	**.153**	.165	.280	1.000	.438	.331	.196	.623	**.299**	.263	.071	.942	.132	.084	.270
KOS + UCT & MC	.299	.276	.287	.377	.100	.210	.131	.167	.280	1.000	.438	.331	.194	.643	.298	.262	.071	.942	.132	.084	.257

Tabelle A.4: Güte der Ergebnisse von FTLR unter Einsatz von Wikipedia2Vec