

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
PROGRAMMING LANGUAGES AND ARTIFICIAL INTELLIGENCE



Titel der Arbeit

Ruben Triwari

Bachelorarbeit
im Studiengang 'Informatik plus Mathematik'

Betreuer: Prof. Dr. Johannes Kinder

Mentor: Moritz Dannehl, M.Sc.

Ablieferungstermin: 28. Oktober 2024

Inhaltsverzeichnis

1	Einführung	1
1.1	Stand der Technik	2
1.2	Leistungen der Arbeit	2
1.3	Aufbau der Arbeit	3
2	Grundlagen und Termini	4
2.1	Maschinelles Lernen	4
2.1.1	Definition	4
2.1.2	Deep-Learning	5
2.1.3	Überwachtes Lernen	7
2.1.4	Unüberwachtes Lernen	7
2.1.5	Reinforcement Learning	7
2.2	Semantische Vektorräume	8
2.2.1	Bag of Words	8
2.2.2	Word2Vec	9
2.2.3	Transformer	10
2.2.4	BERT	11
2.2.5	Sentence Transformer	13
2.3	Code-Llama	14
2.4	Code2Vec	15
2.5	t-SNE	16
3	Methodik	17
3.1	Datensatz	17
3.2	Datenpipeline	17
3.3	Stabilität von SentenceTransformer	18
4	Funktionskommentare	19
4.1	Motivation	19
4.2	Methodik	19
5	Code2Vec	19
5.1	Motivation	19
5.2	Adaption auf C	20
5.3	Training	21
6	Funktionsnamen	21
7	Coddelama-Erklärungen	22
7.1	Motivation	22
7.2	Methodik	22

8	Ergebnisse	23
8.1	Evaluierung durch Experten	23
8.1.1	Methodik	23
8.1.2	Auswertung und Ergebnisse	23
8.2	Qualitative Evaluierung	23
8.3	Quantitative Evaluierung	23
9	Limitation	23
10	Diskussion	23
11	Fazit	23
12	Results: Comparing natural language supervised methods for creating Rich Binary Labels	23
13	Conclusion	24
14	General Addenda	25
14.1	Detailed Addition	25
15	Figures	25
15.1	Example 1	25
15.2	Example 2	25
	Literatur	28

Abstract

1 Einführung

In den letzten Jahren wurden wichtige Fortschritte in der natürlichen Sprachverarbeitung erzielt, ein wichtiger Faktor war die Codierung von natürlicher Sprache in reellwertigen Vektoren (engl. embeddings). Die hochdimensionalen Vektoren codieren die Semantik der ursprünglichen Wörter oder Sätze. Die daraus resultierenden Vektoren erlauben es, neuronale Netzwerke in diversen Anwendungsbereichen zu trainieren, wie beispielsweise. *spam detection* [1]. Zu den bekanntesten Modellen, die natürliche Sprache in Vektoren abbilden, gehören *Word2Vec* [2], *WordPiece* [3] und *SentenceTransformer* [4].

Diese Fortschritte dienen als Motivation, auch Assembler Quellcode auf einen Vektor abzubilden, der die Semantik des Quellcodes codiert. Die resultierenden Vektoren können dann wieder benutzt werden, um neuronale Netzwerke auf diverse Anwendungen zu trainieren, wie beispielsweise *binary code similarity detection* [5], *function boundary detection* [6], *function type inference* [7], *binary code search* [8], Reverse Engineering [9], oder das Klassifizieren von Malware in der Maschinensprache [10].

Um ein Modell mittels überwachtes Lernen zu trainieren, das Assemblercode in semantische Vektoren abbildet, ist für jede Assembler-Funktion ein Vektor erforderlich, der die Semantik der Funktion codiert. Der Datensatz kann durch Kompilieren des Quellcodes von höheren Programmiersprachen generiert werden.

Ziel: Das Ziel dieser Arbeit ist es, verschiedene Methoden zu vergleichen aus C Quellcode Vektoren zu generieren, die die Semantik von dem ursprünglichen Quellcode codieren, mithilfe von Werkzeugen aus der natürlichen Sprachverarbeitung.

Motivation: Der Prozess des Kompilierens reduziert Funktionen auf die für den Computer wesentlichsten Bausteine, dabei gehen viele Informationen verloren, wie Funktionsnamen, Variablennamen und Kommentare. Diese Informationen sind von großer Bedeutung und geben Aufschluss über die Funktionsweise und den Anwendungszweck der Funktion. Durch die Verwendung dieser Informationen könnte die semantische Codierung des Quellcodes verbessert werden. Aufgrund der Tatsache, dass diese Informationen in der natürlichen Sprache sind, ist es sinnvoll, bewährte Werkzeuge aus der natürlichen Sprachverarbeitung zu verwenden, um diese in semantische Vektoren zu codieren. Aus den resultierenden Vektoren kann schließlich ein Datensatz generiert werden, der ein neuronales Netzwerk darauf trainiert, Assemblercode auf hochdimensionale, reellwertige Vektoren abzubilden, die die Semantik des Quellcodes codieren.

1.1 Stand der Technik

Das **CLAP** (Contrastive Language Assembly Pre-training) Paper [11] setzte Anfang 2024 einen neuen Stand der Technik, mithilfe von natürlicher Sprachverarbeitung, in *binary code similarity detection*. Die Aufgabe besteht dabei darin, die semantische Ähnlichkeit zwischen zwei gegebenen Assemblercodes zu bestimmen.

Das Clap-Modell ist aus zwei Teilen zusammengesetzt: einem Assembler-Encoder und einem Text-Encoder. Der Assembler-Encoder, der aus Assemblercode reellwertige Vektoren erzeugt, knüpft mit kleinen Änderungen an den vorherigen Stand der Technik von JTrans an. Der Text-Encoder ist eine völlig neue Idee, die darauf abzielt, den Text wieder in einen reellwertigen Vektor zu konvertieren. Wang et. al starten dabei mit einem Modell aus der natürlichen Sprachverarbeitung namens *SentenceTransformer* und trainieren dieses darauf, Assemblercode die passende Quellcodeerklärung zuzuordnen. Dabei erhalten sie die Quellcodeerklärungen durch ein Large Language Model, wie beispielsweise Chat-GPT.

Das **JTrans**-Modell [5] baut auf einer Modellarchitektur aus der natürlichen Sprachverarbeitung namens Transformer [12] auf. Wang et. al behandeln die einzelnen Assembler-Instruktionen als Wörter, um so die Transformer Modellarchitektur anwenden zu können. Außerdem codieren sie Kontrollflussinformationen des Assemblercodes in die Eingabe des Transformer-Modells. Mit diesem Ansatz erzielten sie im Jahre 2022 einen neuen Stand der Technik in *binary code similarity detection*.

1.2 Leistungen der Arbeit

Die hauptsächlichen Leistungen dieser Arbeit sind:

- Ein Tool, das einen Datensatz mit Assemblercode und semantischen Vektoren aus C-Quellcode und dem dazugehörigen Assemblercode generiert.
- Eine qualitative Analyse durch *t-SNE* [13], die die Verwendung von Funktionskommentaren, Funktionsnamen, *Code Llama* [14] Erklärungen und *Code2Vec* [15] untersucht.
- Eine quantitative Auswertung, die durch Befragung von Experten erfolgt, vergleicht die Verwendung von Funktionsnamen, *Code Llama* Erklärungen und *Code2Vec* miteinander.
- Eine Formel, die die Verwendung von Funktionsnamen, *Code Llama* Erklärungen und *Code2Vec* vergleicht.

1.3 Aufbau der Arbeit

Kapitel 2 führt anfangs grundlegende Konzepte und Begriffe des maschinellen Lernens ein. Anschließend werden semantische Vektorräume in Bezug auf natürliche Sprache erläutert und die größten Fortschritte der letzten Jahre beschrieben. Am Ende werden noch *Code Llama*, *Code2Vec* und *t-SNE* vorgestellt, welche eine wichtige Rolle in dieser Arbeit spielen.

Die allgemeine Architektur des Tools und dessen Designentscheidungen werden im Kapitel 3 beschrieben. Es werden die Auswahl des C-Quellcodes, die Datenpipeline und die Stabilität des *SentenceTransformers* erläutert.

Kapitel 4, 5, 6 und 7 befassen sich mit den verschiedenen Methoden, Embeddings zu erzeugen. Also wie aus den Quellcodeinformationen über Funktionsnamen und -kommentare ein Vektor erstellt werden kann. Außerdem wird erläutert, wie mit *Code Llama* und *Code2Vec* Vektoren produziert werden können.

Die Ergebnisse werden in einer qualitativen und quantitativen Auswertung in Kapitel 8 vorgestellt. Zuerst wird durch eine Experten-Evaluierung die beste Methode identifiziert. Anhand dieser Einordnung wird überprüft, ob die Formel und Analyse durch *t-SNE* ein ähnliches Ergebnis aufweisen.

Die Ergebnisse werden im Kapitel 9 reflektiert, eingeordnet und diskutiert. Es werden die Stärken und Schwächen jeder Methode dargestellt und diskutiert.

Im letzten Kapitel wird die gesamte Arbeit reflektiert und anschließend ein Ausblick gegeben, um weitere Anregungen für die weitere Arbeit in diesem Thema zu geben.

2 Grundlagen und Termini

2.1 Maschinelles Lernen

Heutzutage ist maschinelles Lernen weitverbreitet und wird in nahezu jeden Bereich der Informatik verwendet. In diesem Abschnitt wird zunächst maschinelles Lernen definiert und dann darauf aufbauend grundlegende Trainingsarten vorgestellt. Maschinelles Lernen wird überall dort eingesetzt, wo eine analytische Lösung eines Problems zu aufwendig oder gar überhaupt nicht existiert. Diese Lösung durch maschinelles Lernen versucht aus den Daten ein Muster abzuleiten. Bei einer endlichen Menge an Daten ist meist, das resultierende Modell nur eine Approximation der gesuchten Lösung.

2.1.1 Definition

Trotz des Bekanntheitsgrades, gibt es den Irrglauben, dass maschinelles Lernen nur was mit neuronalen Netzwerken zu tun hat, diese Annahme ist im allgemeinen falsch. Generell kann ein Problem, das mit maschinellem Lernen gelöst wird, wie folgt formuliert werden:

Definition 1 Sei X eine beliebige Input Menge, Y eine beliebige Output Menge, $f \in \{X \rightarrow Y\}$ die gesuchte Lösung des Problems, \mathbb{D} eine beliebige Menge aus gegebenen Datenpunkten, $H_1 \subset \{X \rightarrow Y\}$ ein Hypothesenraum, und $A_1 : \mathcal{P}(\{X \rightarrow Y\}) \times \mathcal{P}(\mathbb{D}) \rightarrow \{X \rightarrow Y\}$ ein Lernalgorithmus. Dann ist das Ziel, bei gegebenen Daten, den Hypothesenraum H_1 und den Lernalgorithmus A_1 so zu wählen, sodass

$$A_1(H_1, \mathbb{D}) \approx f.$$

Maschinelles Lernen ist also die Suche nach einem Lernalgorithmus und Hypothesenraum, die dann in Kombination mit gegebenen Daten, die optimale Lösung approximieren. Dabei ist hervorzuheben, dass der Datensatz das Herzstück jeder Problemstellung im Bereich des maschinellen Lernens ist. Ist der Datensatz zu klein oder überhaupt nicht repräsentativ für das gegebene Problem, wird der Lernalgorithmus die falschen Muster erkennen und dadurch eine fehlerhafte Approximation produzieren.

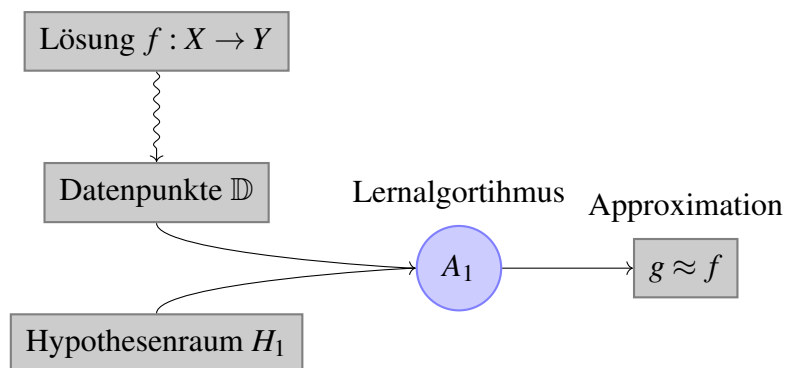


Abbildung 2.1: Grundlegendes maschinelles Lernen Problem

In der Figur 2.1 ist die Problembeschreibung nochmal bildlich dargestellt, erwähnenswert ist,

dass die Datenpunkte nicht immer in abhängigkeit mit $f : X \rightarrow Y$ stehen. Beispielsweise können die Datenpunkte einfach nur aus den Eingabewerten bestehen: $\mathbb{D} = \{x_1, x_2, x_2, \dots, x_n\} \subset X$. Die Struktur des Datensatzes kann sehr unterschiedlich sein, dass hängt auch mit unterschiedlichen Lernmethoden zusammen.

2.1.2 Deep-Learning

Bevor die Lernmethoden genauer betrachtet werden, wird kurz Deep-Learning vorgestellt. Deep-Learning ist ein Neuronales Netzwerk mit mehreren Layern zwischen Input und Output Layer. Zunächst müssen wir jedoch Neuronale Netzwerke definieren.

Definition 2 Ein Neuronales Netzwerk (NN) ist eine Funktion $N : \mathbb{R}^q \rightarrow \mathbb{R}^p$, wobei $q \in \mathbb{N}$ die Anzahl der Inputs und $p \in \mathbb{N}$ die Anzahl der Outputs ist. Sei $(L_i)_{i \in \{1, \dots, n\}}$ die Layer, $(K_i^l)_{l=1, \dots, n, i=1, \dots, r_l}$ die Knoten im jeweiligen Layer $l \in \{1, \dots, n\}$ und $r_l \in \mathbb{N}$ die Anzahl der Knoten im Layer L_l . Jeder Knoten im Layer L_l ist mit jedem Knoten im Layer L_{l+1} verbunden, mit $l \in \{1, \dots, n-1\}$. Jede Verbindung besitzt ein Gewicht $W_{i,j}^l$, wobei $l \in \{1, \dots, n-1\}$ und das Gewicht der Verbindung $K_i^l \rightarrow K_j^{l+1}$ zugeordnet ist. Daraus ergibt sich eine Familie von Matrizen $(W_l)_{l=1, \dots, n-1}$, wobei $W_l \in \mathbb{R}^{r_l \times r_{l+1}}$. Nun hat jeder Layer noch einen sogenannten Bias $(B_l)_{l=2, \dots, n}$, dieser ist ein Zeilenvektor $B_l \in \mathbb{R}^{r_l}$. Als letztes braucht jeder Knoten eine Aktivierungsfunktion, dass heißt für jeden Layer gibt es r_l Funktionen: $(F_l)_{l=1, \dots, n}$, mit $F_l \in \{\mathbb{R} \rightarrow \mathbb{R}\}^{r_l}$. Es ist hilfreich die Funktionsanwendung auch für den Vektor F_l zu definieren: Sei $x \in \mathbb{R}^{r_l}$, dann setze

$$F_l(x) := \begin{pmatrix} f_1(x_1) \\ \vdots \\ f_{r_l}(x_{r_l}) \end{pmatrix}.$$

Dann ist die Funktion $N : \mathbb{R}^q \rightarrow \mathbb{R}^p$ wie folgt definiert:

$$N(x) = h_1(x)$$

,wobei

$$h_l : \mathbb{R}^{r_l} \rightarrow \mathbb{R}^{r_{l+1}}$$

$$h_l(x) = \begin{cases} h_{l+1}(F_{l+1}(W_l x + B_{l+1})) & , \text{if } l < n \\ x & , \text{sonst} \end{cases}.$$

Wir bezeichnen L_1 als Input-Layer, L_n als Output-Layer und L_i , mit $i \in \{2, \dots, n-1\}$, als Hidden-Layer.

Also ist Deep-Learning ein bestimmter Hypothesenraum, denn alle Neuronale Netzwerke sind höher dimensionale reelwertige Funktionen. Schließlich gilt für den Hypothesenraum:

$$H = \{\mathbb{R}^n \rightarrow \mathbb{R}^k\}, \text{ wobei } n, k \in \mathbb{N}$$

Die Definition von einem Neuronalen Netzwerk erscheint zunächst länglich und unintuitiv, diese wird aber anschaulich anhand eines Beispiels.

Beispiel 2.1 Sei $N : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, $(L_i)_{i=1,2,3}$ Layer. Der Input-Layer besitzt zwei Knoten $r_1 = 2$, der erste Hidden-Layer besitzt $r_2 = 3$, der zweite Hidden-Layer besitzt $r_3 = 3$ Knoten und der Output-Layer besitzt $r_4 = 2$ Knoten. Mit den Knoten $(K_i^l)_{l=1,2,3,i=1,\dots,r_l}$ und zufälligen Gewichten:

$$W_1 = \begin{pmatrix} 0.2 & 0.7 & 0.4 \\ 0 & 0.7 & 0.8 \end{pmatrix} \in \mathbb{R}^{2 \times 3}, W_2 = \begin{pmatrix} 0.6 & 0 & 0.4 \\ 0.1 & 0.7 & 0.8 \\ 1 & 0.33 & 0.2 \end{pmatrix} \in \mathbb{R}^{3 \times 3}, W_3 = \begin{pmatrix} 0.2 & 0.45 \\ 0.1 & 0.23 \\ 1 & 0.33 \end{pmatrix} \in \mathbb{R}^{3 \times 2}.$$

Für den Bias setzen wir:

$$B_2 = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix} \in \mathbb{R}^3, B_3 = \begin{pmatrix} 0.4 \\ 0.5 \\ 0.6 \end{pmatrix} \in \mathbb{R}^3, B_4 = \begin{pmatrix} 0.7 \\ 0.8 \end{pmatrix} \in \mathbb{R}^2,$$

Außerdem setzen wir alle Aktivierungsfunktionen:

$$(F_l)_i = \tanh, \text{ wobei } l \in \{1, 2, 3\}, i \in \{1, \dots, r_l\}$$

Dann gilt für das Neuronale Netzwerk $N : \mathbb{R}^2 \rightarrow \mathbb{R}^2$:

$$N(x) = \tanh \left(\begin{pmatrix} 0.2 & 0.45 \\ 0.1 & 0.23 \\ 1 & 0.33 \end{pmatrix} \tanh \left(\begin{pmatrix} 0.6 & 0 & 0.4 \\ 0.1 & 0.7 & 0.8 \\ 1 & 0.33 & 0.2 \end{pmatrix} \tanh \left(\begin{pmatrix} 0.2 & 0.7 & 0.4 \\ 0 & 0.7 & 0.8 \end{pmatrix} x + \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix} \right) + \begin{pmatrix} 0.4 \\ 0.5 \\ 0.6 \end{pmatrix} \right) + \begin{pmatrix} 0.7 \\ 0.8 \end{pmatrix} \right)$$

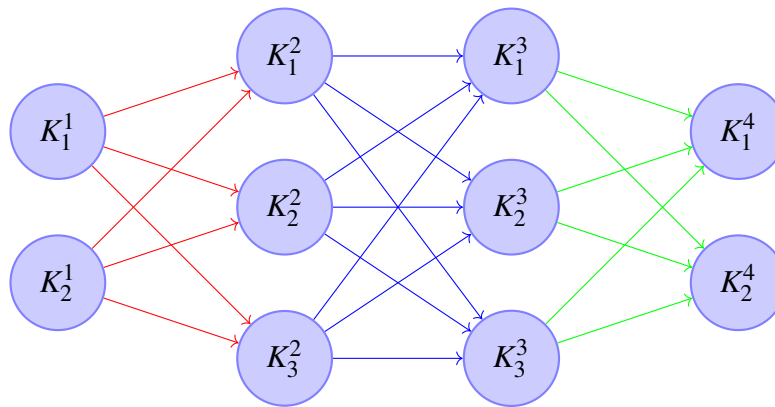


Abbildung 2.2: Neuronales Netzwerk bildlich als Graph dargestellt

Ein Gewicht zwischen zwei Knoten $K_1^1 \rightarrow K_1^2$ kann nun einfach nachgeschaut werden:

$$(W_1)_{1,1} = 0.2$$

Bei dem Beispiel oben handelt es sich um Deep-Learning, da das Neuronale Netzwerk zwei Hidden-Layer besitzt. Heutzutage haben Deep-Learning Models zwei bis drei stellige Anzahl an Hidden-Layer. Diese Dimensionen sind aber für ein Beispiel eher ungeeignet.

2.1.3 Überwachtes Lernen

Das überwachte Lernen ist die meist eingesetzte Trainingsmethode, deswegen auch die wichtigste. Bei diesem Ansatz liegt immer die korrekte Lösung für jeden Input bei, dementsprechend ist der Datensatz ein Tuple aus Input und korrektem Output:

$$\mathbb{D} = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n)\} \subset X \times Y.$$

Ein klassisches Beispiel für diesen Ansatz ist die Bilderkennung, hierbei wäre ein möglicher Input ein Vektor mit Grauwerten und der Output ein Label bzw. die Bezeichnung für das Bild.

Beispiel 2.2 Sei $X = \mathbb{R}^{4096}$ und $Y = \{\text{Katze}, \text{Hund}, \text{Auto}\}$, dann könnte der Datensatz wie folgt aussehen:

$$\mathbb{D} = \left\{ \left(\begin{pmatrix} 0.2 \\ 0.9 \\ 0.5 \\ \vdots \end{pmatrix}, \text{Hund} \right), \left(\begin{pmatrix} 0.1 \\ 0.1 \\ 0.6 \\ \vdots \end{pmatrix}, \text{Hund} \right), \left(\begin{pmatrix} 0.6 \\ 0.7 \\ 0 \\ \vdots \end{pmatrix}, \text{Katze} \right), \dots, \left(\begin{pmatrix} 0.4 \\ 0.3 \\ 0.9 \\ \vdots \end{pmatrix}, \text{Auto} \right) \right\}$$

Der entscheidende Punkt ist also hier, dass wir das richtige Verhalten unseres Modells kennen und deswegen direkt wissen, wenn es Fehler macht. Bei vielen anderen Arten ist dieser Aspekt, der sehr natürlich erscheint, nicht so selbstverständlich. Beim **selbst-überwachten Lernen** generiert der Lernalgorithmus die richtigen Lösungen und jeden Input aus gegebenen Daten selber.

2.1.4 Unüberwachtes Lernen

Unüberwachtes Lernen ist der Extremfall, denn hier bekommt der Lernalgorithmus ausschließlich die Inputwerte: $\mathbb{D} = (x_1, x_2, x_3, \dots, x_n) \subset X$. Der Lernalgorithmus erhält also keine Hinweise darauf, was ein richtiger oder ein falscher Output ist. Unüberwachtes Lernen wird meist eingesetzt, um in Daten, Strukturen und Mustern zu identifizieren. Ein Beispiel ist die Cluster-Analyse, hier bekommt der Lernalgorithmus eine Menge von Daten und gruppiert diese in Teilmengen. In der Abbildung 2.2 ist ein Beispiel für das Resultat einer möglichen Cluster-Analyse dargestellt.



Abbildung 2.3: Cluster Analyse angewendet auf 2-dimensionale Daten

2.1.5 Reinforcement Learning

Das Reinforcement Learning ist nicht so extrem wie das unüberwachte Lernen, bei diesem Paradigma liegen dem Lernalgorithmus, zwar auch nicht die korrekten Outputwerte vor, aber der Lernalgorithmus kriegt für jeden vorhergesagten Wert eine Rückmeldung, wie erwünscht dieser Wert ist. Hier ist ein Beispiel, ein Modell, mittels eines Lernalgorithmus, darauf zu trainieren, ein Videospiel zu gewinnen. Der Lernalgorithmus bekommt ein Abbild von der Umgebung und gibt dem Spiel ein Input, welcher eine Aktion zufolge hat. Falls nun die Aktion dazu beiträgt, den Spieler in eine gute Position zu bringen oder gar das Spiel zu gewinnen, bekommt die Aktion eine positive

Bewertung, andernfalls eine negative. Intuitiv könnte man dieses Paradigma auch als „learning by doing“ bezeichnen, da der Lernalgorithmus nach ausreichendem Ausprobieren das gewünschte Verhalten erlernt.

Die Problemstellung im maschinellen Lernen ist allgemein gehalten und abstrakt. Aber genau aus diesem Grund kann maschinelles Lernen in so vielen unterschiedlichen Bereichen eingesetzt werden. In dieser Arbeit werden wir uns hauptsächlich mit dem Bereich der linguistischen Datenverarbeitung (engl. Natural Language Processing) befassen, was uns zum nächsten Begriff führt. Linguistische Datenverarbeitung wird im folgenden mit NLP abgekürzt.

2.2 Semantische Vektorräume

Das semantische kodieren von natürlicher Sprache in einen Vektorraum, ist ein bedeutsames Problem in NLP(Referenz). Der resultierende Vektorraum kann, für verschieden Problemstellungen (engl. downstream task) verwendet werden. Ein Beispiel ist das klassifizieren von Fake-News oder das zusammenfassen von längeren Texten(Quelle?). Semantischer Vektorraum heißt hier das ähnliche Wörter, also Wörter mit ähnlicher Bedeutung, im projizierten Vektorraum einen geringen Abstand zu einander haben. Diese Vektoren werden wir im folgenden Embeddings nennen. In einem optimalen Vektorraum würde im Figur 2.3 gezeigte Beziehung gelten.

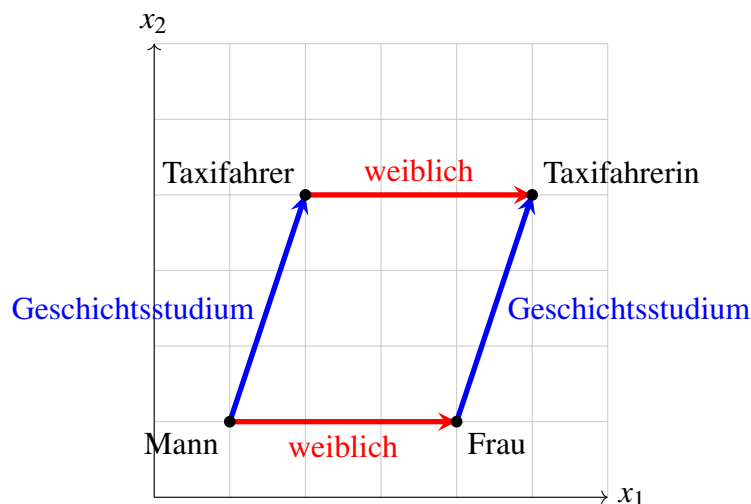


Abbildung 2.4: Optimaler fiktiver semantischer Vektorraum

2.2.1 Bag of Words

Der naivste Ansatz ist es jedem Wort im vorliegenden Text eine Zahl zuzuordnen. Daraus kann zum einen die Häufigkeit eines Wortes abgeleitet werden, aber auch die Sätze, in denen ein bestimmtes Wort vorkommt, effizient gefunden werden. Die Bedeutung eines Wortes ist hier komplett unabhängig von der Wahl der zugewiesenen Zahl. Das führt dazu, dass sogar Synonyme einen hohen Abstand haben können.

Beispiel 2.3 Sei $T = \{\text{Input}, \text{Mann}, \text{Frau}, \text{Eingabe}\}$ eine Menge von Wörtern, dann ist die Zuordnung zu dem Vektorraum \mathbb{N}^1 wie folgt:

$$\text{Input} \rightarrow 1, \text{Mann} \rightarrow 2, \text{Frau} \rightarrow 3, \text{Eingabe} \rightarrow 4.$$

Obwohl Eingabe und Input semantisch sehr ähnlich sind, haben sie hier einen sehr unterschiedlichen Wert.

2.2.2 Word2Vec

Im Jahr 2013 veröffentlichte Google ein Paper, indem sie zwei Modellarchitekturen vorstellen, mit denen ein neuronales Netzwerk effizient lernen kann, semantische Embeddings zu produzieren. Das Neuronale Netzwerk hat in beiden Architekturen ein Hidden-Layer, die Gewichte des Hidden-Layers beinhalten nach dem Training die reellwertigen Vektorrepräsentationen. Der Aufbau bei beiden Modellen ist in Figur 2.4 dargestellt. Zu beachten ist, dass der Hidden-Layer keine Aktivierungsfunktion besitzt, da er nur als lineare Transformation von einer One-Hot-Kodierung zu einem reellwertigen Vektor dient. One-Hot-Kodierung produziert für jedes aus n Wörtern einen Vektor $v \in \{0, 1\}^n$. Der Vektor besitzt immer nur eine eins und sonst überall eine null. Jede Position, wo eine eins ist, gibt es jeweils nur einmal und kodiert so, mittels der Position, ein Wort. Der Output-Layer hat als Aktivierungsfunktion einen Softmax, der Softmax gibt Werte zwischen 0 und 1 zurück. Der Output ist also ein Vektor $v \in (0, 1)^n$, daraus kann dann entweder je nach Anwendung ein Wort oder mehrere abgeleitet werden.

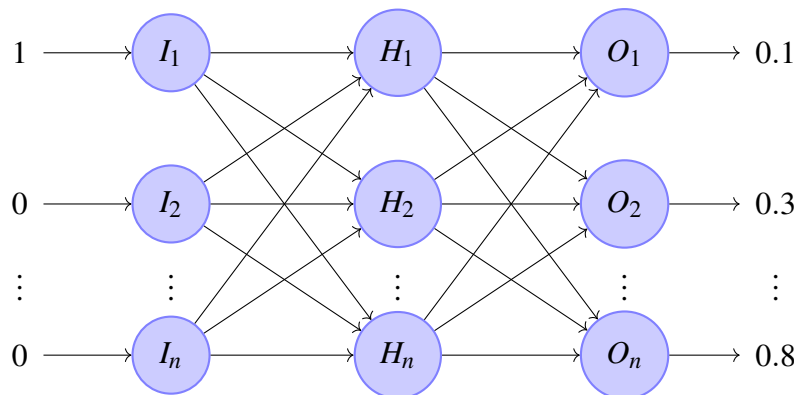


Abbildung 2.5: Neuronales Netzwerk von Word2Vec

Es gibt nun zwei unterschiedliche Herangehensweisen, dieses Neuronale Netzwerk zu trainieren. Die erste ist Skip-gram, gegeben ein Wort muss das Modell die naheliegenden Wörter vorhersagen. Das heißt es muss aus einem Wort einen richtigen Kontext zuordnen. Das Modell wird so lernen, Wörtern die ähnliche Kontexte haben ähnliche Embeddings zuzuordnen. Die zweite Methode ist Continuous-Bag-Of-Words (CBOW), bei der es genau umgekehrt ist, bei dieser sind Wörter nah an einem bestimmten gegeben und die Aufgabe des Modells ist es, dieses bestimmte Wort vorherzusagen. Hier ist also der Kontext gegeben und das Wort, welches in den gegebenen Kontext benutzt wird, muss hervorgesagt werden. Da ähnliche Wörter ähnliche Kontexte haben, neigt das

Modell ähnliche Outputs für ähnliche Kontexte zu erlernen. Ähnliche Outputs können dadurch erzeugt werden, dass die Embeddings für die Kontextwörter ähnlich sind.

Die Word2Vec Embeddings sind also dann ähnlich bzw. im Vektorraum nah aneinander, wenn ihre Kontexte in denen sie verwendet werden ähnlich sind. Das Problem ist, dass die Kontexte limitiert sind auf eine feste Größe, die am Anfang gewählt wird. Sei nun die Kontextgröße $k \in \mathbb{N}$ ungerade, das bedeutet, dass der Kontext eines Wortes alle Wörter sind die $\lfloor \frac{k}{2} \rfloor$ stellen vor dem Wort positioniert sind und $\lfloor \frac{k}{2} \rfloor$ nach dem Wort positioniert sind. Falls das $k \in \mathbb{N}$ zu klein gewählt wird könnte es passieren, dass nur inhaltslose Wörter wie z.B. Artikel oder Präpositionen, im Kontext enthalten sind und somit das Wort mit Inhaltslosen Wörtern assoziiert wird. Wenn aber das $k \in \mathbb{N}$ zu breit gewählt ist, könnten verschieden Kontexte verschwimmen und so ungenau Ergebnisse entstehen. Die Kontextgröße ist also ein entscheidender Parameter, für den Erfolg des Modells. Google selber löst das Kontextproblem in ihrem Paper „Attention Is All You Need“, mit ihrer Modellarchitektur namens Transformer.

2.2.3 Transformer

Das Paper „Attention Is All You Need“ ist ein Meilenstein im NLP Bereich. Die vorgestellte Deep Learning-Architektur ist Grundlage für BERT, Sentence Transformer und Large Language Models (wie z.B. Chat-GPT). Das Modell wurde Ursprünglich entwickelt um bei gegebenem Satz, einen sinnvollen neuen Satz zu generieren. Die verwendete downstream task im Paper ist das Übersetzen von englisch zu deutsch und englisch zu französisch. Der Transformer kann aber wieder dazu verwendet werden semantische Embeddings zu produzieren, wie wir im Abschnitt zu dem Sentence Transformer sehen werden. Die Architektur hat zwei große Blöcke, die in der Abbildung 2.6 zusehen sind. Der erste Block ist der Encoder, dieser erhält die Eingabe. Beim Übersetzen wäre das der zu übersetzende Text. Der Encoder kodiert den Input in semantische embeddings und gibt den wichtigen Kontext für den Input aus, das heißt er gibt aus wie wichtig jedes Wort in dem Satz ist für ein gegebenes Wort. Es gibt also keine Kontextgröße, sondern der gesamte Input ist der Kontext, aber der Encoder gibt vor, welche Wörter wichtig sind und welche eher unwichtig sind, für ein gegebenes Wort. Der Decoder kriegt den von ihm selber produzierten Output und das Ergebnis des Encoders, um das nächste Wort zu generieren. Am Anfang bekommt der Decoder ein konstantes Start Embedding und um die Generierung zu stoppen gibt er selber ein konstantes Stop Embedding aus. Zu beachten ist, dass man mehrere Encoder und Decoder hintereinander schalten kann.

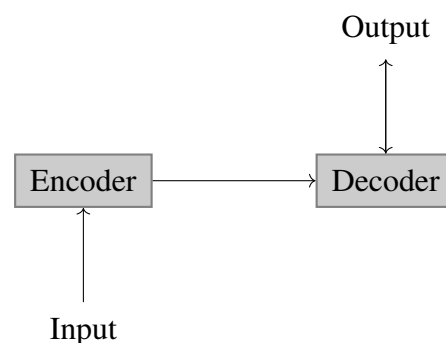


Abbildung 2.6: Transformer Architektur stark vereinfacht

In Abbildung 2.7 ist die Architektur genauer dargestellt. Im folgenden werden alle Bestandteile die in Abbildung vorkommen kurz erklärt. **Input Embedding** gibt dem Input der in Textform vorliegt eine Vektor repräsentation. Danach wird auf dem Vektor ein **Positional Encoding** darauf addiert. Der Transformer verarbeitet alle Wörter Parallel, deswegen geht die Positionsinformation verloren. Um diese Information trotzdem im vektor zu kodieren wird für jede position ein einzigartigen Vektor darauf addiert. Im Block **Add & Norm** werden zwei Inputs addiert und dann normalisiert, damit die Werte in dem Modell nicht zu groß werden. **Feed Forward** ist einfach ein Neuronales Netzwerk mit zwei Layern. Alle Wortembeddings werden nacheinander und identische in das Netzwerk ein gegeben. Der erste Input-Layer hat als Aktivierungsfunktion ein Softmax und der zweite hat die Identitätsfunktion. Dann gilt für das Netzwerk:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2.$$

Der **Linear** Block ist wieder ein Neuronales Netzwerk, mit allen Aktivierungsfunktionen: $f_{i,j}(x) = x$. Nun fehlt noch der wohl wichtigste Bestandteil der Architektur das **Multi-Head Attention** Modul. Dieses gibt die Korrelation zwischen ein Wort und allen restlichen im Input aus. Hier wird also der ganze Input als Kontext genommen, aber das Modell lernt auf welche Wörter es im Kontext viel oder wenig aufmerksamkeit schenken sollte. Das **Masked Multi-Head Attention** Modul ist nur beim trainieren anders, als das Multi-Head Attention Modul. Beim trainieren ist der Input des Decoders schon der Satz der von dem Modell vorhergesagt werden soll, deswegen müssen alle wörter die es noch nicht vorhergesagt hat verdeckt (engl. Masked) werden.

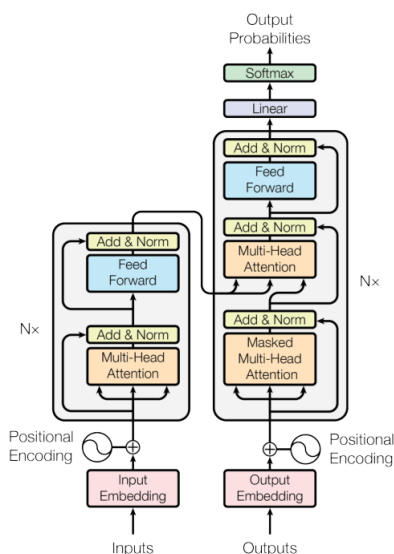


Abbildung 2.7: entommen aus „Attention Is All You Need“

2.2.4 BERT

Das BERT Modell ist wieder ein großer Meilenstein im NLP Bereich und kann als einer der ersten Large Language Models betrachtet werden. Das Modell verbessert die Transformer Architektur indem es die Embeddings der Token verbessert, zwei neue Trainingsaufgaben einführt und

ausschließlich Encoder verwendet.

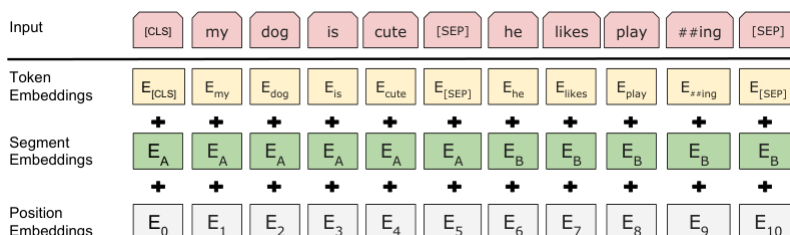


Abbildung 2.8: Token Embedding Prozess entnommen aus dem BERT Paper.

In Abbildung 2.8 ist der Token Embedding Prozess abgebildet, der im folgenden näher beschrieben wird. Zunächst muss erläutert werden was ein Token ist. Ein **Token** ist in unsere Fall ein Wort, aber generell ist ein Token, eine kleinere Zeichenkette die aus einen Text gewonnen wird und einer Bedeutung zugewiesen wird oder in eine Zahl umgewandelt wird um die weiterverarbeitung zu vereinfachen.

Das BERT Modell addiert drei unterschiedlich informationen in Vektorform auf, um noch bessere Token Embeddings zu generieren. BERT benutzt WordPiece um die Token Embeddings zu bekommen, auf diesen wird nun noch ein Segment Embedding und schließlich Position Embedding darauf addiert. **Segment Embedding** kodiert die Information welche Tokens strukturell zusammen gehören (Bspw. ein Satz). Das **Position Embedding** kodiert die sequentielle Position im Input.

Das wichtigste Puzzle teil sind die zwei unterschiedlichen Trainingsaufgaben. Während der Transformer strikt die Token von links nach rechts vorhersagt, wird BERT darauf trainiert Wörter die auf beliebiger Position fehlen vorherzusagen, deswegen hat das Modell zugriff auf den Kontext der sich links und recht von dem zu verher sagenden Token befindet. Die Autoren benennen diese Trainingsform **Masked Language Modeling**, es werden 15% der Input Tokens durch, entweder das [Mask] Token oder mit einen zufälligen anderen Token ersetzt. Dabei bleiben 10% der Tokens die maskiert werden sollten doch gleich, weitere 10% davon werden durch ein zufälligen anderen Token ersetzt und die restlichen 80% davon werden durch das [Mask] Token ersetzt. Die zweite Traininsaufgabe ist **Next Sentence Prediction** (NSP), bei dieser Aufgabe muss das BERT Modell bei der Eingabe von zwei Sätzen entscheiden, ob diese sequentiell nacheinander kommen oder nicht. Dabei sind 50% der Daten, zwei Sätze die nacheinander kommen und bei den anderen 50% wird zu einen gegebenen Satz aus den gesamten Daten ein zufälliger nicht sequentieller Satz gewählt. Um eine vorhersage zu treffen, ob die Sätze nacheinander kommen oder nicht wird das konstante [cls] Token Embedding verwendet, welches immer an erster stelle steht. Der erster Outputvektor korrespondiert dann mit dem [cls] Token und wird verwendet um den Output isNext oder notNext zu produzieren.

Nachdem Training erhält man aus BERT die Outputvektoren welche genau gleich viele sind wie der Inputvektoren, diese kann man dann mit wenig aufwand weiter verwenden, um das Modell für bestimmte Probleme in NLP anzupassen (engl. fine-tuning).

2.2.5 Sentence Transformer

Das Sentence-BERT (Reimers et al. 2019) Modell ist Stand der Technik, im semantischen kodieren von natürlicher Sprache in einem Vektorraum. Die Implementierung der Autoren ist unter den Namen Sentence-Transformer ([LINK??](#)) bekannt und wird in dieser Arbeit verwendet. Die Motivation hinter Sentence-BERT (SBERT) ist das effiziente semantische vergleichen von zwei Sätzen. Wenn man mit BERT herausfinden will, wie semantisch ähnlich zwei gegebene Sätze sind, dann muss man beide als Input in das Modell eingeben. Dass heißt wenn man die Ähnlichkeit von allen n Sätzen jeweils zueinander haben will, gibt es $\frac{n(n-1)}{2}$ eingaben in das BERT Modell, die man tätigen müsste. Bei SBERT allerdings kann nun jeder Satz einzeln eingegeben werden, der resultierende Output ist dann ein semantischer Vektorraum, indem jeder Satz ein semantischen Vektor besitzt. Um jetzt die semantische Ähnlichkeit zwischen zwei Vektoren zu erhalten, müssen beide Vektoren nur in eine zu wählende Metrik eingesetzt werden. In den Paper wird die Kosinus-Ähnlichkeit verwendet: Sei $u, v \in \mathbb{R}^n$, dann setze

$$Kos(u, v) := \cos(\phi) = \frac{u \cdot v}{\|u\| \|v\|} = \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2} \sqrt{\sum_{i=1}^n v_i^2}}.$$

Damit kann dann effizient die semantische Ähnlichkeit zwischen zwei Vektoren berechnet werden. Die Modellarchitektur von SBERT wird als **Siamese Neural Network** bezeichnet, da scheinbar zwei unterschiedliche BERT Modelle verwendet werden um beide Sätze in einen Vektor umzuwandeln, aber beide Modell teilen sich die Gewichte.

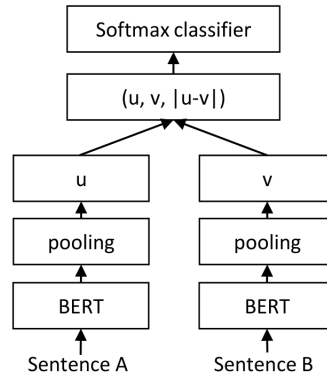


Abbildung 2.9: SBERT Architektur entnommen aus dem SBERT Paper.

In Abbildung 2.9 ist die SBERT Siamese Neural Network Architektur abgebildet, das heißt dass sich beide BERT Modelle die selben Gewichte teilen. Danach geht der Output von BERT in ein pooling Modul. **Pooling** ist das transformieren von Output mit vielen Vektoren in eine geringere Anzahl von Vektoren oder Dimension. In unseren Fall werden die n Output Vektoren von BERT in einen einzigen Output Vektor transformiert. Das in dem Paper verwendete default pooling ist der Mittelwert, d.h. auf alle Outputvektoren von BERT, wird das arithmetische Mittel angewendet, daraus erhält man einen Vektor mit den Mittelwerten der jeweiligen BERT Output Vektoren. Desweiteren wird das BERT Modell noch einwenig, auf die spezielle aufgabe, zwei Vektoren semantisch zu

vergleichen, angepasst. Dafür wird der SNLI Datensatz verwendet, dieser beinhaltet immer zwei sätze und eins von drei möglichen Label: contradiction, neutral, und entailment. Also muss SBERT für zwei sätze vorhersagen ob sie sich inhaltlich widersprechen, neutral zueinander sind, oder der eine Satz eine Fortsetzung des anderen ist. Daraus lernt das Modell ein semantisches verständnis für Sätze zu erlangen. Beim Training wird um die label Vorhersage zubekommen, wie in Abbildung 2.9 dargestellt, die jeweiligen Vektoren und ihre Differenz konkatinert. Anschließend werden diese mit einen lernbaren Gewicht multipliziert, sodass ein Vektor mit drei Dimensionen herauskommt, dann wird der Softmax angewendet. Die Position in dem Vektor mit dem höchsten Wert wird schließlich dem zugehörigen Label zugeordnet. Mathematisch:

Sei $W \in \mathbb{R}^{n \times 3}$ das lernbare Gewicht und $v \in \mathbb{R}^n$ Outputvektor von pooling, dann

$$o = \text{softmax}(Wv).$$

Nachdem Training, bei der Inferenz wird der Vektor nachdem pooling als outputvektor ausgegeben. SBERT liefert uns also ein Tool, um Sätze semantisch in einen Vektorraum abzubilden, welches sich in späteren Kapiteln als sehr hilfreich herausstellt.

2.3 Code-Llama

Die Code-Llama familie an Large Language Models, wurde von Meta AI im Jahre 2024 entwickelt. **Large Language Models** sind Modelle, die auf einer großen Menge von Daten trainiert wurden, welche sich darin auszeichnen, natürliche Sprache verstehen sowie generieren zu können und deswegen in der Lage sind, eine Vielzahl von Aufgaben im NLP-Bereich zu lösen. Meta AI optimiert das vorangegangene Llama2 Modell, auf programmiersprachen spezifischen Aufgaben. Das Llama2 Modell wird um zum resultierenden Code-Llama zu kommen, erneut auf einen neuen Datensatz trainiert. Dieser besteht aus drei verschiedenen kategorien von Daten. Der größte Teil im Datensatz mit 85%, ist der mit Programmiersprachen spezifischen aufgaben, indem wird das Modell darauf trainiert fehlende Programmzeilen in einer vorgebenen Lücke zu füllen. Dabei kann es sich um Befehle, aber auch um Kommentare handeln. Der zweit größte Teil im Datensatz mit 8%, besteht aus natürlicher Sprache in der es um programmcode geht. Dieser Teil beinhaltet diskussion über Programmcode ,sowie Fragen und Antworten welche sich auf Programmcode beziehen. Der kleinste Teil im Datensatz mit 7% besteht aus beliebiger natürlicher Sprache, damit das Modell seine alten fähigkeiten erhält.

Das Llama2 Modell ist eine Verbesserung des Llama1 Modells. Dieses wurde auf neuen Daten Trainiert und verwendet einen 40% größeren Datensatz. Weiteren wurde die **context length** verdoppelt, welche die maximale Anzahl der gleichzeitig zu verarbeitende Token beschreibt. Außerdem gab es eine veränderung in der Llama1 Transformer Architektur. In den Attention Modulen werden zwei Werte die normalerweise jedesmal wieder berechnet werden geteilt, d.h. jedes Attention Modul hat Zugriff auf die gleichen Werte. Das führt zu etwas schlechteren Ergebnissen, aber einer deutlichen Leistungs steigerung.

Das Llama1 Modell wiederum besteht aus einer Decoder-Only Transformer Architektur. Der **Decoder-Only Transformer** besteht wie der Name es schon vermuten lässt, nur aus Decoder Transformer Blöcken. Bei dieser Architektur ist der anfängliche Input des Decoders, die Eingabe des Nutzers. Auf diese Eingabe wird dann immer wieder das neu generierte Token konkatinert. Auf diese weise wird die Eingabe nicht explizit von dem bereits generierten getrennt sondern beides

wird als bereits generierter Kontext verwendet. Das besondere an Llama1 ist das es nur auf frei verfügbaren Daten trainiert wurde und dass das Modell Open Source ist. Der Datensatz besteht aus **English Common Crawl** [67%], **C4** [15%], **Github** [4.5%], **Wikipedia** [4.5%], **Gutenberg and Books3** [4.5%], **ArXiv** [2.5%], und **Stack Exchange** [2%].

Code Llama ist Open-Source und schneidet unter allen öffentlich verfügbaren Modellen am besten in multilingual Benchmarks ab. Mit multilingual ist hier die verwendung von mehreren Programmiersprachen gemeint. Diese Eigenschaften machen es sehr geeignet für diese Arbeit.

2.4 Code2Vec

Cod2Vec ist eine im Jahre 2018 entwickelte Modellarchitektur, die Quellcode in einen semantischen Vektor kodiert. Nach den vielen erfolgen in NLP, natürliche Sprache in semantische Vektoren zu kodieren, ergab sich der natürliche Wunsch auch Quellcode in semantische Vektoren abzubilden. Mit diesen Vektoren kann dann wieder viele verschiedene Aufgaben gelöst werden. Das motivierende Beispiel im Code2Vec Paper ist dem Quellcode einer Funktion einen sinnvollen Namen vorherzusagen. Im folgende wird die Architektur die in Abbildung 2.10 zusehen ist erklärt.

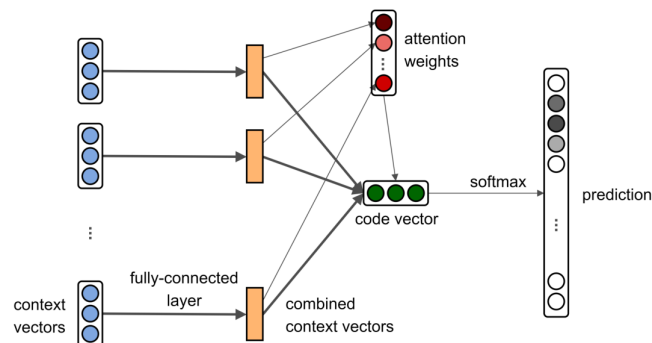


Abbildung 2.10: Code2Vec Architektur entnommen aus dem Code2Vec Paper

Die Autoren fanden heraus das eine geeignete Darstellung von Quellcode als ein mathematisches Objekt ein abstrakter Syntaxbaum ist. Dieser erhält die strukturellen Zusammenhänge zwischen den Tokens und kann gut in einen Vektor kodiert werden. Die Inputvektoren (context vectors) bestehen jeweils aus einem Pfad im abstrakten Syntaxbaum, mit dem jeweiligen Starttoken und Endtoken des Pfades. Danach folgt ein Hidden-Layer, mit tanh als Aktivierungsfunktion. Der finale Vektor (code vector) berechnet sich einfach als lineare Kombination aus den Outputvektoren und den Attention weights. Sei $h_1, \dots, h_n \in \mathbb{R}^d$ die Outputvektoren von dem Hidden-Layer und $\alpha \in \mathbb{R}^n$ der attention weights Vektor.

$$\text{code vector } v = \sum_{i=1}^n \alpha_i \cdot h_i$$

Mit dem code vector kann dann das gewünschte Label vorhergesagt werden. Das Modell kann demnach darauf trainiert werden ein bestimmtes Label vorherzusagen, welches zu einem Quellcode, also einer Reihe an context vector zugeordnet wird. Nachdem Training kann es dann auch Label für Quellcode vorhersagen die es noch nie gesehen hat.

Die Trainingsart ist demnach Überwachteslernen, was eine aufbereitung der Daten benötigt. Das Modell kann deswegen auch nur Label vorhersagen in der Inferenz, die es vorher im Training gesehen hat.

2.5 t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) ist ein Algorithmus, welcher zum visualisieren von hochdimensionalen Daten eingesetzt wird. Um das zu ermöglichen reduziert t-SNE die Dimension von $n \in \mathbb{N}$ zu einer niedrigeren Dimension wie zwei oder drei, in der der Mensch die Datenpunkte leicht interpretieren kann, ohne die Nachbarschaftsverhältnisse der Datenpunkte in mitleidenschaft zu ziehen.

Im folgende wird der Algorithmus skizziert und danach wird aufgezeigt was bei der effektiven verwendung von t-SNE zu beachten ist. Die hochdimensionalen Datenpunkte werden mit \mathbf{H} und die niedrigdimensionale Datenpunkte mit \mathbf{N} bezeichnet. Der erste schritt des Algorithmuses ist es jedem Datenpunktpaar im Datensatz \mathbf{H} ein Ähnlichkeitsscore zu zuweisen. Dieser wird berechnet indem man zuerst die euklidische Distanz von jeden Datenpaar berechnet und dann das Ergebnis in eine Wahrscheinlichkeitsverteilung eingibt, dadurch wird der Wert unter anderen Normalisiert. Das Ergebnis ist dann eine Tabelle mit einen Ähnlichkeitsscore für jedes Datenpaar in \mathbf{H} . Als nächstes werden die Datenpunkte zufällig in der niedrigen Dimension \mathbf{N} angeordnet. Die nachfolgenden zwei schritte werden $T \in \mathbb{N}$ mal wiederholt, wobei T ein Parameter ist der wählbar ist.

1. Berechne Ähnlichkeitsscore von \mathbf{N} , diesmal wird aber die studentische t-Verteilung als wahrscheinlichkeitsverteilung genommen.
2. Verschiebe die Datenpunkte von \mathbf{N} um ein kleinen Wert in die Richtung, die den Unterschied der Ähnlichkeitsscores von \mathbf{H} und \mathbf{N} minimiert.

Nach T wiederholung ist der Ähnlichkeitsscore von \mathbf{N} und \mathbf{H} nahe bei einander, d.h. die Nachbarschaftsverhältniss von \mathbf{N} und \mathbf{H} sind nun ähnlich.

Mitarbeiter von Google haben untersucht, wie man t-SNE sinnvoll anwendet und welche schlüsse man aus der visualisierung ziehen kann. Sie fanden heraus, dass die wahl der Parameter für das Ergebnis eine wichtige Rolle spielen. Die wichtigsten Parameter sind die Iterationen $T \in \mathbb{N}$ und die Perplexity $P \in \mathbb{N}$. Die **Perplexity** kann intuitiv als schätzung für die Anzahl an nahen Nachbarn die jeder Datenpunct hat gesehen werden. Eine geeignete Iteration T kann relative einfach durch ausprobieren herausgefunden werden: Falls sich die Datenwolke bei erhöhung von T nicht mehr wirklich verändert, ist die Anzahl der Iteration T gefunden worden. Eine geeignete Perplexity zu finden ist schwieriger, da wir die hochdimensionalen Nachbarschaftsbeziehungen meistens nicht kennen. Die Autoren des t-SNE Paper empfehlen eine Perplexity $P \in \{5, 6, \dots, 50\}$. Außerhalb dieses Bereiches können verschieden ungewollte Phänomene auftreten. Bei $P = 2$ haben die Google Mitarbeiter herausgefunden das t-SNE bei einer zufällig generierten Datenwolke, fälschlicher weise kleine Gruppierungen (Cluster) bildet. Falls P größer ist als die Anzahl der Datenpunkte, ist das Ergebnis überhaupt nicht interpretierbar. Es ist also immer Sinnvoll, mehrere Werte für P aus zuprobieren, um sicher zu gehen das t-SNE keine falschen Nachbarschaftsbeziehungen darstellt. Die Mitarbeiter von Google fanden ausßerdem heraus, dass sowie die Information der Breite eines Clusters, als auch der Abstände von einen Cluster zu einen anderen durch t-SNE komplett verloren gehen. Es kann also nach betrachten der t-SNE Ausgabe keine Aussage über den Durchmesser eines Clusters, die

Position des Cluster und die Lagebeziehungen zwischen Clustern getroffen werden.

Der t-SNE Algorithmus ist ein wichtiges Tool um qualitative Aussagen über Daten zu treffen. Allerdings sollten immer mehrere Parameter ausprobiert werden. Es kann nur eine Aussage über die Existenz von Cluster getroffen werden und nicht über ihre geometrischen Gegebenheiten. Wenn diese Rahmenbedingungen beachtet werden ist t-SNE ein sehr mächtiges Visualisierung Tool um eine Intuition von der Anordnung der Datenpunkte zu erhalten.

3 Methodik

3.1 Datensatz

Im Maschinellen Lernen hat der Datensatz bzw. die Trainingsdaten den größten Einfluss auf die Güte des Modells. Wir haben die Open Source Standard C Bibliothek von GNU ausgewählt. Die Sprache C wurde gewählt, da sie die weit verbreitetste zu maschinen Code kompilierbare Sprache ist. Die GNU Bibliothek wurde zum einen ausgewählt, da die Qualität des Quellcodes hoch ist, das liegt daran, dass das Projekt seit 1987 existiert und die große Community jederzeit auf Fehler im Quellcode aufmerksam machen kann. Zum anderen wird die Bibliothek weitgehend in vielen Applikationen eingesetzt. Der wohl wichtigste Punkt ist, dass man mit diesen Datensatz die Ergebnisse gut vergleichen kann, denn die C Bibliothek ist im POSIX Standard festgelegt und wurde mehrmals unterschiedlich implementiert, dadurch erhält man mehrere hoch qualitative Quellcode Projekten, die die selbe Semantik aufweisen. Damit kann dann bspw. die Binary Code Similarity Detection Aufgabe durchgeführt werden.

Die **Binary Code Similarity Detection** (BCSD) gibt an, wie unterschiedlich zwei in Assembler vorliegende Funktionen sind. Da unterschiedliche Implementierungen zu unterschiedlichen Assembler Code kompiliert werden, kann mit unterschiedlichen Standard-C-Bibliotheken, die den POSIX Standard implementieren, getestet werden, ob das finale Modelle beide Implementierungen als sehr ähnlich klassifiziert.

3.2 Datenpipeline

Die große praktische Arbeit der Bachelorarbeit war es, eine große Menge von Daten in verschiedenen Darstellungen immer wieder umzuwandeln. Dabei wurde jeder Zwischenschritt gespeichert, damit nicht jede Darstellung immer wieder generiert werden muss. Im folgenden wird die generelle Architektur, wie Daten verarbeitet werden, vorgestellt (engl. Pipeline), um einen Überblick über den praktischen Teil dieser Arbeit zu geben.

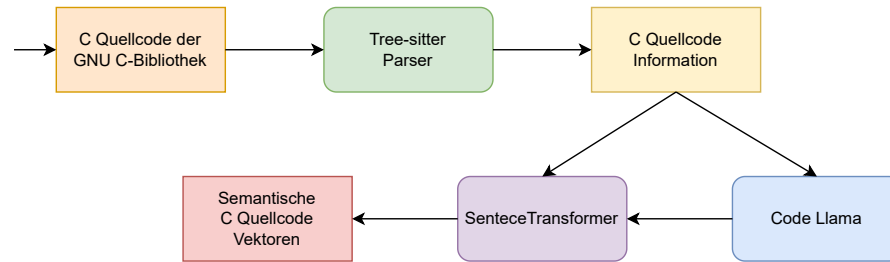


Abbildung 3.1: Datenpipeline

In der Abbildung 3.1 ist die Datenpipeline dargestellt, dabei sind Vierecke mit runden Ecken Tools die Daten in eine andere Darstellung umwandeln und Vierecke mit spitzen Ecken repräsentieren Darstellungen von Daten. Die erste Darstellung sind die Rohdaten, das entspricht den unverarbeiteten C Quellcode aus der GNU Standard C Bibliothek. Nun brauchen wir aber nicht alle Teile des Quellcode sondern nur bestimmte Teile, zum Beispiel arbeiten wir nur mit Funktionen d.h. alle Datenstrukturen die in dem Quellcode definiert werden, wollen wir aus dem Quellcode entfernen. Die Zerlegung und Umwandlung des Inputs in sinnvolle Teile wird parsing genannt. In dieser Arbeit wurde **Tree-sitter** als Parser verwendet, welcher alle populären Programmiersprachen in Syntaxbäume umwandeln kann. Ursprünglich wurde Tree-sitter für den Texteditor Atom entwickelt und wird heute noch in vielen Texteditoren für bspw. Syntaxhighlighting verwendet. Generell kann Tree-sitter jedoch für alles das Quellcode verarbeiten will verwendet werden. Aus dem Syntaxbaum kann dann jegliche gewünschte Information entnommen werden. Diese Quellinformationen werden dann erstmal zwischen gespeichert, damit von nun an, die Verarbeitung hier angesetzt werden kann. Die Quellinformationen beziehen sich immer auf eine Funktion, also ist das Format eine Tabelle die für jede Funktion in den Rohdaten die gewünschten Quellinformationen enthält. Danach gibt es eine Abzweigung, entweder werden die Quellinformationen Code Llama nochmal erklärt und dann in den SentenceTransformer gegeben oder die Quellinformation wird direkt in den SentenceTransformer gegeben. Schließlich erhält man wieder eine Tabelle die gespeichert wird, die für jede Funktion das zugewiesene Semantische Embedding enthält. Bei dieser Architektur kann mühelos jedes Tool ausgetauscht werden solange die Ausgabeformate eingehalten werden.

3.3 Stabilität von SentenceTransformer

In dem vorherigen Unterkapitel haben wir gesehen dass der finale Schritt für alle Daten der SentenceTransformer ist, deswegen ist er das Herzstück der Datenpipeline. In dieser Arbeit sollen verschiedene semantische Beschreibungen des Quellcodes in natürlicher Sprache verglichen werden. Um hier sinnvoll zu messen, müssen, wie bei einem physikalischen Experiment, alle anderen Elemente in der Datenpipeline konstante Ergebnisse liefern und nicht schwanken. Aus diesen Gründen wird im folgenden untersucht, wie stabil sich der SentenceTransformer bei selber Eingabe verhält. Im Idealfall sollte der SentenceTransformer bei selben Input selbes Ergebnis liefern.

Um das zu überprüfen wurden $n = ?$ Code-Llama Quellcode Erklärungen $m = ?$ mal in den SentenceTransformer eingegeben, dabei beträgt der höchste Abstand von zwei Vektoren die aus der gleichen Erklärung resultiert sind $d = ?$. Dieser Abstand ist hinreichend gering um ihn in der Evaluation zu

vernachlässigen. Der Sentence Transformer ist also für die Anwendung in dieser Arbeit hinreichend stabil.

4 Funktionskommentare

4.1 Motivation

Bevor ein Programm kompiliert wird und nur noch die nötigsten Informationen für den Computer bestehen bleiben, gibt es eine Menge an Informationen die die Semantik der Funktion in natürlicher Sprache beschreiben. Eine offensichtliche Quellcodeinformation die im optimalfall die Semantik der Funktion in natürlicher Sprache beschreibt ist der Kommentar. Ein gelungener Kommentar für eine Funktion beschreibt präzise die kernfunktion der Prozedur, d.h. der Input, den Output und wie diese umwandlung erfolgt. Dieser könnte man dann mit dem SentenceTransformer in einen Semantischen Vektoraum abbilden, was in einen Semantischen Quellcode Vektor resultiert.

4.2 Methodik

Das parsen der Kommentare wurde wie in dem Unterkapitel Datenpipeline erwähnt mit Tree-sitter realisiert. Dabei gab es zwei große Designentscheidungen zu treffen. Zum einen welche Kommentare in einer Funktion berücksichtigt werden sollen und zweitens was macht man mit Funktionen die eine leicht Variationen von einer anderen Funktion sind und deswegen keine Kommentare besitzen. Ein gutes Beispiel für das zweite Problem ist `exit` und `__run_exit_handlers`, wenn `exit` aufgerufen wird, ruft diese Funktion einfach `__run_exit_handlers` mit speziellen Parametern auf. Dabei ist `exit` nicht kommentiert, aber `__run_exit_handlers` ist kommentiert.

Bei der ersten Designentscheidung welche Kommentare ich berücksichtige, habe ich mich ausschließlich für den Kommentar direkt über der Funktion entschieden, da die einzeligen Kommentare in der Funktion meistens keinen großen Semantischen Wert haben, sondern auf gefahren oder Designentscheidungen hinweisen. (Beleg maybe: A survey on Reasearh of code comment) Für das zweite Problem habe ich mich für folgende Lösung entschieden. Falls eine Funktion keinen Kommentar besitzt, dann werden die Kommentare von allen Funktionen die in den Funktionskörper aufgerufen werden konkatinert und als eigenen Kommentar übernommen. Dardurch hat `exit` dann einen Kommentar und zwar exakt den selben wie `__run_exit_handlers`.

Hierbei muss man aufpassen das der Prozess des parsens nicht zu speziell an den vorliegenden Daten angepasst wird, sonst verliert er seine Allgemeingültigkeit. Deswegen habe ich mich nicht auf weitere Optimierungen die ein wenig mehr Kommentare erbrignen könnten eingelassen, sondern es bei den oben beschriebenen belassen.

5 Code2Vec

5.1 Motivation

Das besondere an Code2Vec ist, dass es den Quellcode in eine abstrakten Syntaxbaum kodiert. Damit nutzt Code2Vec alle Quellcodeinformationen die in dem Quellcode vorhanden sind. Danach

wird das Modell darauf trainiert eine Eigenschaft in natürlicher Sprache über den Quellcode vorherzusagen. Bei dieser herangehensweise wird also kein SentenceTransformer verwendet sondern der Semantische Vektorraum wird durch Training des Code2Vec Modells als Nebenprodukt erzeugt. Wie in den Paper wird auch hier Code2Vec darauf trainiert Funktionsnamen vorherzusagen. Die Implementierung des Code2Vec Modells wurde von den Autorn nur auf Java trainiert, deswegen gab es einige anpassungen nötig, um Code2Vec auf C Quellcode trainieren zu können. Da das Code2Vec Modell auf überwachtes Lernen basiert, muss der Datensatz vor dem Training erstmal aus den ursprünglichen Quellcode aufbereitet werden.

5.2 Adaption auf C

Um das Cod2Vec Modell Trainieren zu können muss aus den rohen Quellcode einer Funktion jeweils ein abstrakter Syntaxbaum und den Funktionsnamen extrahiert werden. Die Autoren des Papers stellen ein Tool für Java zu Verfügung, welches die Extrahierung von abstrakten Syntaxbäumen und Funktionsnamen aus Java Quellcode ermöglicht.

Der *astminer* von JetBrains stellt ermöglicht es aus C-Quellcode abstrakte Syntaxbäume und Funktionsnamen zu extrahieren. Das Tool wurde von dem JetBrains Research Team entwickelt, um Quellcode in abstrakte Syntaxbäume (AST) zu kodieren, welche sich als Inputformat für maschinelle Lernen Modelle eignen.

Das Cod2Vec Modell braucht ein bestimmtes Format indem der Datensatz vorliegen muss. Ein Trainingsbeispiel besteht jeweils aus einen Funktionsnamen, dem Label und einer Liste von Kontexten, welche den AST repräsentieren sollen. Ein Kontext besteht aus einen Token gefolgt von einer Pfadbeschreibung zu einen anderen Token, welches danach folgt. Dieses Code2Vec Format kann durch eine zusätzlich Option in der Konfiguration des *astminers* generiert werden. Trotz dieser Option die mit „code2vec“ betitelt ist, ist der Output von dem *astminer* nicht direkt von Code2vec verwendbar. Das Tool weist nämlich jeden Token eine einzigartige Zahl zu und verwendet dann in Datensatz nur noch die Zahl. Dieses Format reduziert zwar den Speicheraufwand stark, aber es wird von Code2vec nicht als valide Eingabe akzeptiert. Demnach mussten noch die Nummern wieder zu der passenden Token umgewandelt werden.

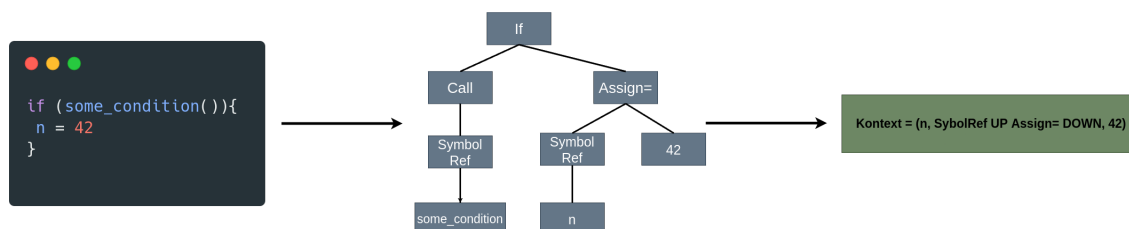


Abbildung 5.1: Extrahierung eines Kontexts

Da es unterschiedliche Konventionen für Funktionsnamen gibt, wie Bspw. Camelcase und Snakecase, normalisiert der *astminer* die Funktionsnamen. Dadurch sind die Trainingsdaten unabhängig von den spezifischen Quellcodekonventionen.

Beispiel 5.1

Snakecase: funktions_name \rightarrow **funktions|name**

Camelcase: funktionsName \rightarrow **funktions|name**

Code2Vec kann nach dem Training für jede Funktion einen Hochdimensionalen Vektor ausgeben, der die Semantik der Funktion beschreiben soll. Damit Code2Vec diesen Vektor generiert, muss der abstrakte Syntaxbaum eingegeben werden. Die abstrakten Syntaxbäume können nur durch ihren normalisierten Namen identifiziert werden, da sie als Tupel in dieser Form im Datensatz vorliegen. Die normalisierten Namen können aber nicht mehr eindeutig dem initialen Namen zugeordnet werden, da bei der Normalisierung Informationen verloren gehen. Um jedoch Code2Vec mit anderen Ansätzen wie Funktionskommentaren vergleichen zu können, müssen die normalisierten Namen wieder zu den ursprünglichen Namen zurückgeführt werden. Aufgrund dessen mussten im Quellcode von astminer Änderungen vorgenommen werden, so dass zu jeder Position eines Trainingsbeispiels im Datensatz den ursprünglichen Funktionsnamen zugeordnet werden konnte.

Mit diesen Anpassungen die noch zusammengefügt werden mussten konnte nun trainiert werden und nach Abschluss des Trainings konnten die semantischen Vektoren, durch diese Anpassungen verglichen und ausgewertet werden.

5.3 Training

Das Ziel beim Training war es die selbe Qualität wie im Paper zu erhalten. Also die selben Ergebnisse für Java auch für C-Quellcode zu replizieren. Für den Anwendungszweck einen Datensatz zu erstellen der für eine Funktion in Assemblersprache einen semantischen Vektor als Label zuordnet, können bestimmte Problemstellungen beim generellen trainieren von Modellen ignoriert werden. Eine Problemstellung bei Modellen ist es inwiefern das Modell generalisiert, also wie leistungsfähig das Modell auf neuen Daten im Gegensatz zu den Trainingsdaten ist. Diese Problemstellung muss nicht beachtet werden, da das Ergebnis kein fähiges Modell ist, sondern semantische Vektoren. Auch der kleine Datensatz mit $n = 5155$ Datenpunkten, ist zwar für die Güte des Modells problematisch, jedoch nicht für die resultierenden semantischen Vektoren. Selbst wenn das Modell nur die passenden Funktionsnamen auswendig lernt, entstehen dabei Vektoren die gewisse Informationen des Namens widerspiegeln. Nachdem das Trainingsszenario genau das selbe wie im Paper ist, wurden alle Trainingsparameter gleich gelassen. Die Ergebnisse nach der 84 Epoche sind: **Precision:** 65.6, **F1:** 65.1, **Recall:** 64.7. Diese Werte sind etwas über den Werten von den Code2Vec Autoren, sie kamen beim Full Test Set auf : **Precision:** 63.1, **F1:** 58.4, **Recall:** 54.4. Dabei ist hervorzuheben das unser Test und Trainings Datensatz der selbe ist, im Gegensatz zu dem Test Datensatz von Code2Vec. Die vorgehensweise ist normalerweise ein grober Fehler, da wir nicht die Generalisierung des Modells messen. In diesem speziellen Anwendungszweck ist jedoch die Güte des Modells nicht von Bedeutung, sondern nur die Güte der erzeugten Vektoren. Diese werden von der Verwendung des gleichen Datensatzes beim Testen nicht in Mitleidenschaft gezogen.

Damit haben wir ähnliche Ergebnisse wie aus dem Paper und können diese mit den anderen vorgestellten Ansätzen vergleichen.

6 Funktionsnamen

Eine andere Quellinformation, die in jedem Quellcode enthalten ist, sind die Funktionsnamen. Funktionsnamen sollen in wenigen Wörtern den Kerninhalt der Funktion widerspiegeln. Dadurch

eignen sich Funktionsnamen um die Semantik einer Funktion zu beschreiben. Außerdem ist die Extrahierung der Funktionsnamen keine schwierige Aufgabe. Hierfür wäre Tree-sitter nicht unbedingt nötig, aber falls ein Datensatz für eine andere Sprache wie Rust erstellt werden sollte, müsste der Parser jedes mal angepasst werden. Deswegen wurde hier für die einfache Erweiterung des Programms, Tree-sitter verwendet. Tree-sitter bietet nämlich Parser für eine große Anzahl an Sprachen an.

7 Codelama-Erklärungen

7.1 Motivation

Large Language Models werden immer besser Quellcode selbst zuschreiben, zu verstehen und zusammen zu fassen. Eine Zusammenfassung von Quellcode in natürlicher Sprache spiegelt die Semantik des Quellcodes wieder. So ist auch Code-Llama fähig Quellcode zu erklären und somit einen Text zu generieren der die Semantik der Funktion enthält. Dieser Ansatz verwendet wie Code2Vec jede Quellcodeinformation, da der gesamte Quellcode als Eingabe genutzt wird. Da wir einen Datensatz erstellen sollte der Labelgenerierungsprozess deterministisch sein, das ist Code-llama anfänglich nicht. Außerdem ist der Eingabetext in Code-llama, abgesehen von dem Quellcode, entscheidend für die Erklärungen.

7.2 Methodik

Die Quellcode erklärungen werden von Code-Llama erzeugt, indem Code-Llama eine präzise Fragestellung und den gesamten Quellcode einer Funktion enthält. Die Fragestellung wurde aus dem Clap Projekt übernommen, diese haben sich auch damit beschäftigt, wie die Fragestellung formuliert werden muss um Semantik reiche und präzise Quellcodeerklärungen zu erhalten.

Damit der Code-Llama Output deterministisch ist, muss der Temperatur Parameter auf Null gesetzt werden. Der Parameter ist Teil einer Zufalls komponente bei der Auswahl des nächsten Tokens, ist dieser auf Null wird der Token mit der höchsten Score gewählt, dieser bleibt bei gleicher Eingabe immer der selbe. Leider ist der Determinismus trotzdem nicht garantiert, wie die studio von ... heraus fand. Der Grund ist laut ... das bei denen vielen fließkomma operationen rundungsfehler entstehen. Diese Erkenntnisse stammen aber aus einen anderen Anwendungszweck.

Die Stabilität mit Temperatur Null sollte deswegen für den Anwendungszweck dieser Arbeit getestet werden. Um das zu überprüfen wurden von $n = ?$ Funktionen der Quellcode mit Fragestellung $m = ?$ mal in Code-Llama eingegeben, bei den resultierenden Vektoren beträgt der höchste Abstand von zwei Vektoren die aus der gleichen Quellcode resultiert sind $d = ?$. Dieser Abstand ist wieder hinreichend gering um ihn in der Evaluation zu vernachlässigen. Das Code-llama LLM ist also für die Anwendung in dieser Arbeit hinreichend Stabil.

8 Ergebnisse

8.1 Evaluierung durch Experten

8.1.1 Methodik

8.1.2 Auswertung und Ergebnisse

8.2 Qualitative Evaluierung

8.3 Quantitative Evaluierung

9 Limitation

10 Diskussion

11 Fazit

12 Results: Comparing natural language supervised methods for creating Rich Binary Labels

- Stabilität von Sentence Transformer
- Kommentare von Funktionen um Embeddings zu generieren
- Funktionsnamen von Funktionen um Embeddings zu generieren
- Code2Vec um Embeddings zu generieren
- CodeLlama Erklärungen von Funktionen um Embeddings zu generieren
- Evaluierung durch tSNE-Plots
- Evaluierung durch Experten
- Evaluierung durch Formel

$$I_k : \mathbf{N} \times \mathbf{N} \times \mathbf{N}^k \rightarrow [0, 1]$$

$$I_k(x, i, v) = \begin{cases} 1 & , \exists j \in \mathbf{N} : x = v_j \wedge i = j \\ \frac{1}{2} & , \exists j \in \mathbf{N} : x = v_j \wedge i \neq j \\ 0 & , \text{otherwise} \end{cases}$$

$$E_k : \mathbf{N}^k \times \mathbf{N}^k \rightarrow [0, 1]$$

$$E_k(u, v) = \frac{1}{G_k} \sum_{i=1}^k \frac{I_k(u_i, i, v_i)}{\log_2(i+1)}$$

$$\text{wo } G_k := \sum_{i=1}^k \frac{1}{\log_2(i+1)}.$$

$$CMP_k : \mathbf{R}^{N \times l} \times \mathbf{R}^{N \times l} \times \{\mathbf{R}^l \times \mathbf{R}^{N \times l} \rightarrow \mathbf{N}^k\} \times \{\mathcal{P}([0, 1]) \rightarrow [0, 1]\} \rightarrow [0, 1]$$

$$CMP_k(X, Y, f_k, agg) = agg(\{E_k(f_k(X_{i,j}, X), f_k(Y_{i,j}, Y)) | j \in \{1, 2, 3, \dots, N\}\})$$

13 Conclusion

14 General Addenda

If there are several additions you want to add, but they do not fit into the thesis itself, they belong here.

14.1 Detailed Addition

Even sections are possible, but usually only used for several elements in, e.g. tables, images, etc.

15 Figures

15.1 Example 1

15.2 Example 2

Abbildungsverzeichnis

2.1	Grundlegendes maschinelles Lernen Problem	4
2.2	Neuronales Netzwerk bildlich als Graph dargestellt	6
2.3	Cluster Analyse angewendet auf 2-dimensionale Daten	7
2.4	Optimaler fiktiver semantischer Vektorraum	8
2.5	Neuronales Netzwerk von Word2Vec	9
2.6	Transformer Architektur stark vereinfacht	10
2.7	entommen aus „Attention Is All You Need“	11
2.8	Token Embedding Prozess entnommen aus dem BERT Paper.	12
2.9	SBERT Architektur entnommen aus dem SBERT Paper.	13
2.10	Code2Vec Architektur entnommen aus dem Code2Vec Paper	15
3.1	Datenpipeline	18
5.1	Extrahierung eines Kontexts	20

Tabellenverzeichnis

Literatur

- [1] S. Ballı and O. Karasoy, “Development of content-based sms classification application by using word2vec-based feature extraction,” *IET Software*, vol. 13, no. 4, p. 295–304, Aug. 2019. [Online]. Available: <http://dx.doi.org/10.1049/iet-sen.2018.5046>
- [2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” 2013. [Online]. Available: <https://arxiv.org/abs/1301.3781>
- [3] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” 2016. [Online]. Available: <https://arxiv.org/abs/1609.08144>
- [4] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” 2019. [Online]. Available: <https://arxiv.org/abs/1908.10084>
- [5] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, “jtrans: Jump-aware transformer for binary code similarity,” 2022. [Online]. Available: <https://arxiv.org/abs/2205.12713>
- [6] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 611–626. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin>
- [7] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, “Neural nets can learn function type signatures from binaries,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 99–116. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua>
- [8] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, “Codee: A tensor embedding scheme for binary code search,” *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2224–2244, 2022.
- [9] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig, and B. Vasilescu, “Dire: A neural approach to decompiled identifier naming,” 2019. [Online]. Available: <https://arxiv.org/abs/1909.09029>
- [10] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, “Malware detection by eating a whole exe,” 2017. [Online]. Available: <https://arxiv.org/abs/1710.09435>
- [11] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, “Clap: Learning transferable binary code representations with natural language supervision,” 2024. [Online]. Available: <https://arxiv.org/abs/2402.16928>

- [12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023. [Online]. Available: <https://arxiv.org/abs/1706.03762>
- [13] L. van der Maaten and G. Hinton, “Visualizing data using t-sne,” *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: <http://jmlr.org/papers/v9/vandermaaten08a.html>
- [14] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, “Code llama: Open foundation models for code,” 2024. [Online]. Available: <https://arxiv.org/abs/2308.12950>
- [15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” 2018. [Online]. Available: <https://arxiv.org/abs/1803.09473>
- [16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019. [Online]. Available: <https://arxiv.org/abs/1810.04805>
- [17] C. Charitsis, C. Piech, and J. C. Mitchell, “Function names: Quantifying the relationship between identifiers and their functionality to improve them,” in *Proceedings of the Ninth ACM Conference on Learning @ Scale*, ser. L@S ’22, vol. 28. ACM, Jun. 2022, p. 93–101. [Online]. Available: <http://dx.doi.org/10.1145/3491140.3528269>
- [18] M. Astekin, M. Hort, and L. Moonen, “An exploratory study on how non-determinism in large language models affects log parsing,” in *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*, ser. InteNSE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 13–18. [Online]. Available: <https://doi.org/10.1145/3643661.3643952>
- [19] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2302.13971>
- [20] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, “Llama 2: Open foundation and fine-tuned chat models,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.09288>