

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: 23. November 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	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	8
2.2	Semantische Vektorräume	8
2.2.1	Bag-of-Words	9
2.2.2	Word2Vec	9
2.2.3	Transformer	10
2.2.4	BERT	12
2.2.5	Sentence Transformer	13
2.3	Code Llama	15
2.4	Code2Vec	15
2.5	T-Distributed Stochastic Neighbor Embedding	17
3	Methodik	18
3.1	Datensatz	18
3.2	Datenpipeline	19
3.3	Stabilität von SentenceTransformer	20
3.4	Funktionskommentare	21
3.5	Code2Vec	22
3.5.1	Adaption auf C	22
3.5.2	Training	23
3.6	Funktionsnamen	24
3.7	Code-Llama-Erklärungen	25
4	Ergebnisse	25
4.1	Befragung von Experten	25
4.2	Quantitative Evaluierung	25
4.3	Qualitative Evaluierung	25
5	Diskussion	25
6	Fazit	25
	Literatur	27

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 Maleware 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.

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.

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

CLAP (Contrastive Language Assembly Pre-training) [11] setzte Anfang 2024 einen neuen Stand der Technik, in *Binary Code-Similarity-Detection* mithilfe von natürlicher Sprachverarbeitung. Die Aufgabe bei *Binary Code-Similarity-Detection* besteht 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.

JTrans [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 Assembler-Codes 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

2.1 Maschinelles Lernen

Heutzutage ist maschinelles Lernen weitverbreitet und wird in nahezu jedem Bereich der Informatik verwendet. Maschinelles Lernen wird überall dort eingesetzt, wo eine analytische Lösung eines Problems zu aufwendig oder gar überhaupt nicht existiert. Maschinelles Lernen sucht nach einer Lösung, indem es aus den Daten ein Muster ableitet. Sind die Daten endlich, ist meist das resultierende Modell nur eine Approximation der gesuchten Lösung. In diesem Abschnitt wird zunächst maschinelles Lernen definiert und dann darauf aufbauend grundlegende Trainingsarten vorgestellt.

2.1.1 Definition

Die weit verbreitete Ansicht, dass maschinelles Lernen nur etwas mit neuronalen Netzwerken zu tun hat, ist im Allgemeinen falsch. Generell kann ein Problem, das mit maschinellem Lernen gelöst wird, wie folgt definiert 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, der dann in Kombination mit gegebenen Daten die bestmögliche Lösung approximiert. Dabei ist hervorzuheben, dass der Datensatz das Herzstück jeder Problemstellung im Bereich des maschinellen Lernens ist. Wenn der Datensatz zu klein oder überhaupt nicht repräsentativ für das gegebene Problem ist, wird der Lernalgorithmus die falschen Muster erkennen und dadurch eine fehlerhafte Approximation erzeugen.

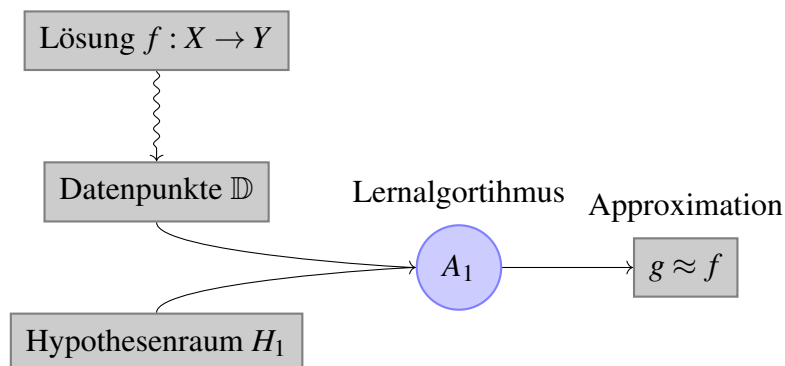


Abbildung 2.1: Herangehensweise um mit maschinellen Lernen ein Problem zu lösen

In der Figur 2.1 ist die Problembeschreibung noch einmal bildlich dargestellt. Beachtenswert 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, das hängt auch mit unterschiedlichen Lernmethoden zusammen.

2.1.2 Deep Learning

Vor der Betrachtung der Lernmethoden wird kurz auf Deep Learning eingegangen. Deep Learning ist ein neuronales Netzwerk, das über mehrere Layer zwischen Input und Output Layer verfügt. Zunächst einmal müssen wir neuronale Netzwerke definieren, die folgende Definition ist inspiriert durch [16].

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, das 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.

Deep Learning ist ein Hypothesenraum, da alle neuronalen Netzwerke höherdimensionale reellwertige Funktionen sind. 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 nicht intuitiv, 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 $N : \mathbb{R}^2 \rightarrow \mathbb{R}^2$:

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

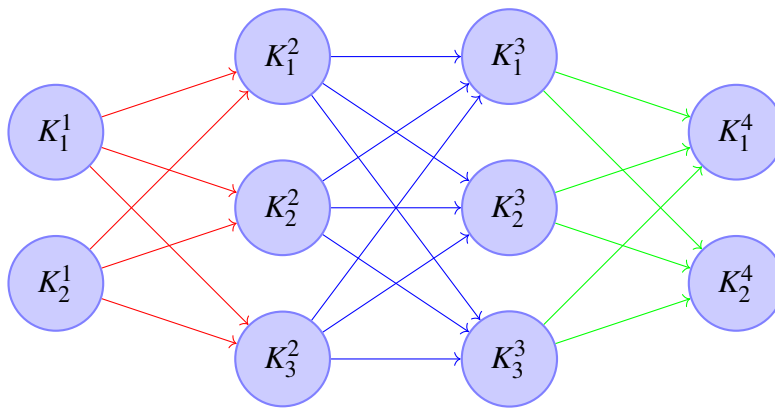


Abbildung 2.2: Neuronales Netzwerk bildlich als Graph dargestellt

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

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

In diesem Beispiel handelt es sich um Deep Learning, da das neuronale Netzwerk zwei Hidden Layer besitzt. Deep Learning Models verfügen heutzutage über eine zwei- bis dreistellige Anzahl an Hidden Layer, diese Dimensionen sind aber für ein Beispiel ungeeignet.

2.1.3 Überwachtes Lernen

Das überwachte Lernen ist die am häufigsten verwendete Trainingsmethode und ist daher auch die wichtigste. Dieser Ansatz basiert immer auf der korrekten Lösung für jeden Input und ist daher ein Tupel aus Input und korrekten Output-Werten.

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

Ein Beispiel für diesen Ansatz ist die Bilderkennung, bei der ein Vektor mit Grauwerten und ein Label bzw. eine Bezeichnung für das Bild verwendet werden.

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 Aspekt ist, dass wir das richtige Verhalten unseres Modells kennen und deshalb direkt wissen, wenn es Fehler macht. Beim selbst-überwachten Lernen werden die richtigen Lösungen aus gegebenen Daten generiert. Bei vielen anderen Arten ist dieser Aspekt, der sehr natürlich erscheint, nicht selbstverständlich.

2.1.4 Unüberwachtes Lernen

Das unüberwachte Lernen ist der Extremfall, da der Lernalgorithmus ausschließlich die Input-Werte erhält.

$$\mathbb{D} = (x_1, x_2, x_3, \dots, x_n) \subset X$$

Der Lernalgorithmus erhält keine Hinweise darauf, was richtig oder falsch ist. Unüberwachtes Lernen ist eine Methode, um in Daten Strukturen und Muster zu identifizieren. Ein Beispiel ist die Cluster-Analyse, hier bekommt der Lernalgorithmus eine Menge von Daten und gruppiert diese in Teilmengen. In der Abbildung 2.3 ist ein Beispiel für das Resultat einer möglichen Cluster-Analyse dargestellt.



Abbildung 2.3: Cluster Analyse angewendet auf 2-dimensionale Daten aus [17].

2.1.5 Reinforcement Learning

Das Reinforcement Learning ist nicht so extrem wie das unüberwachte Lernen. Bei diesem Paradigma liegen dem Lernalgorithmus zwar nicht die korrekten Output-Werte vor, aber der Lernalgorithmus erhält für jeden vorhergesagten Wert eine Rückmeldung, wie erwünscht dieser Wert ist. Ein Beispiel ist ein Modell, das mittels eines Lernalgorithmus darauf trainiert wird, ein Videospiel zu gewinnen. Der Lernalgorithmus bekommt ein Abbild von der Umgebung und gibt dem Spiel einen 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.

Die Problemstellung im maschinellen Lernen ist allgemein formuliert und abstrakt. Dies ist der Grund, warum maschinelles Lernen in vielen unterschiedlichen Bereichen eingesetzt werden kann. Einer der Bereiche ist die linguistische Datenverarbeitung (engl. natural language processing), was uns zum nächsten Begriff führt. Linguistische Datenverarbeitung wird im Folgenden mit NLP abgekürzt.

2.2 Semantische Vektorräume

Das semantische Codieren von natürlicher Sprache in einem Vektorraum ist ein bedeutsames Problem in NLP. Der resultierende Vektorraum kann für verschiedene Problemstellungen (engl. downstream tasks) verwendet werden. Ein Beispiel ist die Klassifizierung von Spam Nachrichten [1]. Semantischer Vektorraum heißt hier, dass ähnliche Wörter, also Wörter mit ähnlicher Bedeutung, im projizierten Vektorraum einen geringen Abstand zueinander haben. Wir werden die Vektoren im Folgenden als Embeddings bezeichnen. In einem idealen semantischen Vektorraum würde die Beziehung gelten, die in Figur 2.4 dargestellt wird.

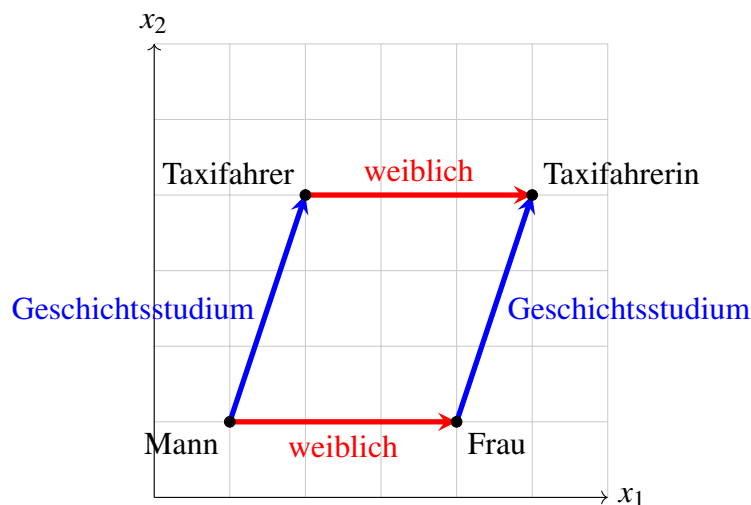


Abbildung 2.4: Optimaler fiktiver semantischer Vektorraum

2.2.1 Bag-of-Words

Der naivste Ansatz ist es, jedem Wort im vorliegenden Text eine Zahl zuzuordnen. Dadurch können sowohl die Häufigkeit eines Wortes als 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 veröffentlichten Mikolov et al. [2], zwei Modellarchitekturen, mit der ein neuronales Netzwerk effizient lernen kann, semantische Embeddings zu produzieren. Mikolov et al. haben eine Implementierung veröffentlicht die sie *Word2Vec*¹ genannt haben. In beiden Architekturen besteht das neuronale Netzwerk aus einem Hidden Layer, der nach dem Training die reellwertigen Vektorrepräsentationen enthält. Der Aufbau bei beiden Modellen ist in Figur 2.5 dargestellt. Es ist zu beachten, dass der Hidden Layer keine Aktivierungsfunktion besitzt, da er nur als lineare Transformation von einer One-Hot-Codierung zu einem reellwertigen Vektor dient.

One-Hot-Codierung produziert für jedes aus n Wörtern einen Vektor $v \in \{0, 1\}^n$. Dieser Vektor ist an einer Stelle mit einer Eins und sonst überall mit einer Null versehen. Jede Position, an der eine Eins ist, gibt es nur einmal und codiert so durch die 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 durch One-Hot-Codierung ein oder mehrere Wörter abgeleitet werden.

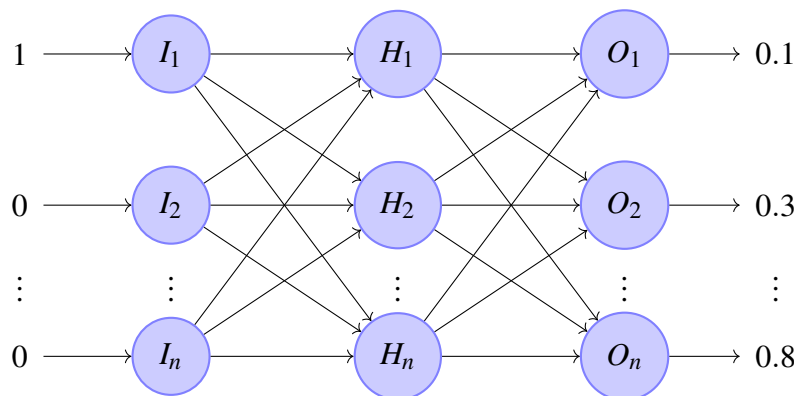


Abbildung 2.5: Architektur des neuronalen Netzwerks von Word2Vec

¹<https://code.google.com/archive/p/word2vec/>

Es gibt nun zwei unterschiedliche Strategien, das neuronale Netzwerk zu trainieren. Die erste ist Skip-Gram, gegeben ein Wort, muss das Modell die naheliegenden Wörter vorhersagen. Das heißt, es muss einem Wort einen richtigen Kontext zuordnen. Das Modell wird die Fähigkeit entwickeln, Wörter mit ähnlichen Kontexten ähnlichen Embeddings zuzuordnen.

Die zweite Methode ist Continuous Bag-of-Words (CBOW), bei der das Modell Wörter nahe an einem bestimmten Wort als Input erhält und versucht, dieses Wort vorherzusagen. Folglich muss das Wort, das in dem gegebenen Kontext verwendet wird, prädiziert werden. Da ähnliche Wörter ähnliche Kontexte haben, neigt das Modell dazu, ähnliche Outputs für ähnliche Kontexte zu lernen.

Die Word2Vec-Embeddings sind dann ähnlich, wenn ihre Kontexte, in denen sie verwendet werden, ähnlich sind. Die Kontexte haben eine feste Größe, die am Anfang ausgewählt wird. Wenn die Kontextgröße zu klein gewählt wird, kann es passieren, dass das Wort mit inhaltslosen Wörtern assoziiert wird, wie z.B. Artikel oder Präpositionen. Wenn die Kontextgröße zu groß ist, können unterschiedliche Kontexte verschwimmen und ungenaue Ergebnisse entstehen. Folglich hat die Kontextgröße einen entscheidenden Einfluss auf den Erfolg des Modells. Vaswani et al. lösen das Kontextproblem durch die Transformer-Modellarchitektur [12].

2.2.3 Transformer

Die Transformer-Architektur [12] ist ein Meilenstein im NLP Bereich. Die vorgestellte Deep Learning Architektur bildet die Basis für BERT, *SentenceTransformer* und Large Language-Models (wie z.B. Chat-GPT). Das Modell wurde ursprünglich entwickelt, um bei einem gegebenen Satz einen sinnvollen neuen Satz zu generieren. Die Downstream Task auf die eine Implementierung der Architektur trainiert wurde, ist das Übersetzen von Englisch zu Deutsch oder Französisch. Der Transformer kann jedoch wieder zur Erzeugung semantischer Embeddings verwendet werden, wie wir im Abschnitt zum *SentenceTransformer* sehen werden.

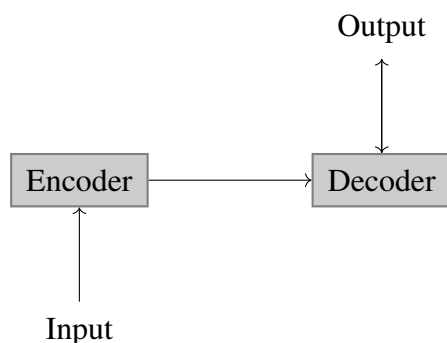


Abbildung 2.6: Transformer-Architektur stark vereinfacht

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 codiert den Input in semantische Embeddings und gibt an, wie wichtig jedes Wort in dem Satz für ein gegebenes Wort ist. Es gibt keine Kontextgröße, sondern der Kontext ist die gesamte Eingabe. Der Encoder bestimmt, welche Wörter wichtig sind und welche eher unwichtig sind in Bezug auf ein gegebenes Wort. Der Decoder erhält den von ihm selbst produzierten Output

sowie das Ergebnis des Encoders, um das nächste Wort zu generieren. Außerdem startet dieser mit einem konstanten Start-Embedding und um die Generierung zu stoppen, gibt er selbst ein konstantes Stopp-Embedding aus. Es ist wichtig, dass mehrere Encoder und Decoder gleichzeitig hintereinander geschaltet werden können.

Die Architektur ist in Abbildung 2.7 dargestellt. Im Folgenden werden die wichtigsten Bestandteile der Abbildung erläutert. **Input-Embedding** gibt dem Input, der in Textform vorliegt, eine Vektorrepräsentation. 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 codieren, wird für jede Position ein einzigartiger 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 Wort-Embeddings werden nacheinander und identisch in das Netzwerk eingegeben. 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: $f_{i,j}(x) = x$. Das **Multi Head-Attention** Modul ist der wohl wichtigste Bestandteil der Architektur. Dieses gibt die Korrelation zwischen einem Wort und allen Restlichen im Input aus. Der gesamte Input wird als Kontext betrachtet, 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 bereits der Satz, den das Modell vorhersagen soll. Daher müssen alle Wörter, die es noch nicht vorhergesagt hat, verdeckt (engl. masked) werden.

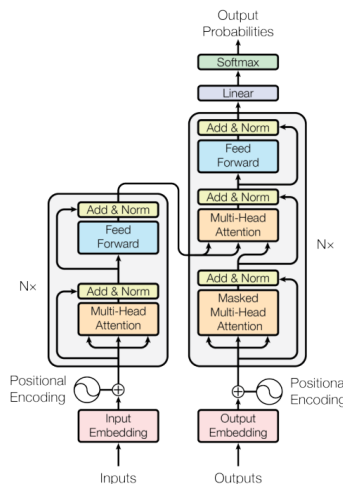


Abbildung 2.7: Transformer-Architektur entnommen aus [12]. Der linke Block ist der Encoder und der rechte Block der Decoder.

2.2.4 BERT

Das BERT-Modell [18] ist ein wichtiger Meilenstein im NLP-Bereich und kann als eines der ersten Large Language-Models betrachtet werden. Das Modell verbessert die Transformer-Architektur, indem es die Token-Embeddings verbessert, zwei neue Trainingsaufgaben einführt und ausschließlich Encoder verwendet.

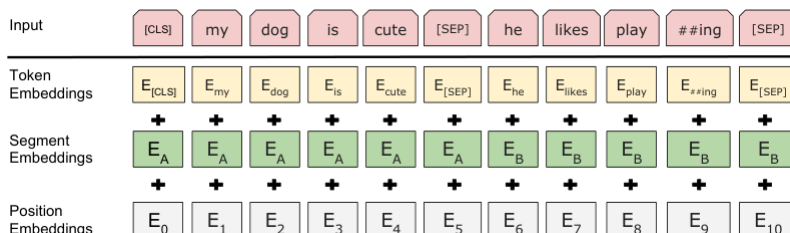


Abbildung 2.8: Token-Embedding-Prozess entnommen aus [18].

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 diesem Fall ein Wort, aber generell ist ein Token eine kleinere Zeichenkette, die aus einem Text gewonnen wird und einer Bedeutung zugewiesen wird oder in eine Zahl umgewandelt wird, um die Weiterverarbeitung zu vereinfachen.

Das BERT Modell kombiniert drei unterschiedliche Informationen in Vektorform, um noch bessere Token-Embeddings zu generieren. BERT verwendet *WordPiece* [3], um die Token-Embeddings zu erhalten. Auf diese werden nun ein Segment-Embedding und schließlich die Position-Embeddings addiert. **Segment-Embedding** codiert die Information, welche Tokens strukturell zusammen gehören (Bspw. ein Satz). Das **Position-Embedding** codiert die sequenzielle Position im Input.

Die beiden unterschiedlichen Trainingsaufgaben stellen das wichtigste Puzzleteil dar. Während der Transformer die Tokens von links nach rechts vorhersagt, wird BERT darauf trainiert, Wörter vorherzusagen, die auf beliebiger Position fehlen. Dadurch erhält das Modell Zugang zu dem Kontext, der sich links und rechts von dem zu prädizierenden Token befindet. Wu et al. benennen diese Trainingsform **Masked Language-Modeling**. Bei diesem Verfahren werden 15% der Input-Token entweder durch das [Mask] Token oder durch einen zufälligen anderen Token ersetzt. Dabei werden 10% der Token, die maskiert werden sollten, unverändert bleiben. Weitere 10% werden durch ein zufälliges anderes Token ersetzt, während die restlichen 80 % durch das [Mask] Token ersetzt werden.

Die zweite Trainingsaufgabe ist **Next Sentence-Prediction** (NSP), bei dieser Aufgabe muss das BERT Modell bei der Eingabe von zwei Sätzen entscheiden, ob diese sequenziell nacheinander kommen oder nicht. Bei der einen Hälfte der Daten handelt es sich um zwei Sätze, die nacheinander auftreten. Bei der anderen Hälfte der Daten handelt es sich um zufällige, nicht sequenzielle Sätze. Zur Vorhersage, ob die Sätze nacheinander kommen oder nicht, wird das konstante [cls] Token-Embedding verwendet, welches beim Input immer an erster Stelle steht. Der erste Output-Vektor ist dann das Ergebnis des [cls] Tokens und wird verwendet, um den Output isNext oder notNext zu produzieren.

Nach dem Training erhält man aus BERT für jeden Input-Vektor genau einen Output-Vektor, diesen 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 [4] Modell ist der aktuelle Stand der Technik im semantischen Codieren von natürlicher Sprache in einem Vektorraum. Die Implementierung von Reimers et al. ist unter dem Namen *SentenceTransformer*² bekannt. Die Motivation hinter Sentence-BERT (SBERT) ist das effiziente semantische Vergleichen von zwei Sätzen. Wenn man mit BERT herausfinden möchte, wie semantisch ähnlich zwei gegebene Sätze sind, dann muss man beide Sätze als Input in das Modell eingeben. Das 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 kann dagegen jeder Satz einzeln eingegeben werden und der resultierende Output ist dann ein semantischer Vektorraum, indem jeder Satz einen semantischen Vektor besitzt. Um die semantische Ähnlichkeit zwischen zwei Vektoren zu erhalten, müssen beide Vektoren lediglich in eine zu wählende Metrik eingesetzt werden. Dies ermöglicht es, die semantische Ähnlichkeit zwischen zwei Vektoren effizient zu berechnen.

Die Modellarchitektur von SBERT wird als **Siamese Neural Network** bezeichnet, da zwei unterschiedliche BERT-Modelle verwendet werden, um beide Sätze in einen Vektor umzuwandeln, aber beide Modelle teilen sich die Gewichte.

²<https://sbert.net>

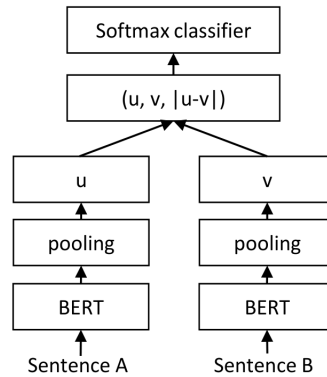


Abbildung 2.9: SBERT-Architektur entnommen aus [4]. Beide BERT-Modelle teilen sich dieselben Gewichte.

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 Dimensionen. In diesem Fall werden die n Output-Vektoren von BERT in einen einzigen Output-Vektor transformiert. Das Default Pooling ist der Mittelwert, d.h. auf alle Output-Vektoren von BERT wird das arithmetische Mittel angewendet. Daraus erhält man einen Vektor mit den jeweiligen gemittelten BERT-Output-Vektoren.

Das BERT Modell wird an die spezielle Aufgabe angepasst, zwei Vektoren semantisch zu vergleichen. Dafür wird der SNLI Datensatz verwendet, dieser beinhaltet immer zwei Sätze und eins von drei möglichen Labels: *contradiction*, *neutral*, und *entailment*. SBERT muss für zwei Sätze vorhersagen, ob sie sich inhaltlich widersprechen, neutral zueinander sind oder ob der eine Satz eine Fortsetzung des anderen ist. Daraus lernt das Modell, ein semantisches Verständnis für Sätze zu erlangen. Zur Ermittlung der Label-Vorhersage werden die jeweiligen Vektoren und ihre Differenz konkateniert. Anschließend werden diese mit einem, durch Training erlernten, Gewicht multipliziert, sodass ein Vektor mit drei Dimensionen entsteht. Danach wird der Softmax auf das Ergebnis 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 Gewicht und $v \in \mathbb{R}^n$ Output-Vektor von dem Pooling, dann

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

Nachdem Training bei der Inferenz wird der Vektor, nachdem Pooling als Output-Vektor ausgegeben. SBERT liefert uns also ein Tool, um Sätze semantisch in einem 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 Rozière et al. bei Meta AI im Jahre 2024 entwickelt [14]. **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 Llama-2-Modell auf Programmiersprachen spezifische Aufgaben. Das LLama-2-Modell wird, um zum resultierenden Code Llama zu kommen, erneut auf einem neuen Datensatz trainiert. Dieser besteht aus drei verschiedenen Kategorien von Daten. Der größte Teil im Datensatz, mit 85%, ist mit Programmiersprachen spezifischen Aufgaben verbunden, indem das Modell darauf trainiert wird, fehlende Programmzeilen in einer vergebenen Lücke zu füllen. Dabei kann es sich um Programmcode, aber auch um Kommentare handeln. Der zweitgrößte Teil im Datensatz, mit 8%, besteht aus natürlicher Sprache, in der es um Programmcode geht. Dieser Teil beinhaltet Diskussion über Quellcode sowie Fragen und Antworten, welche sich auf Quellcode beziehen. Der kleinste Teil im Datensatz mit 7% besteht aus beliebiger natürlicher Sprache, damit das Modell seine alten Fähigkeiten erhält.

Das LLama-2-Modell ist eine Verbesserung des Llama-Modells. Dieses wurde auf neueren Daten trainiert und verwendet einen 40% größeren Datensatz. Zudem wurde die maximale Anzahl der gleichzeitig zu verarbeitenden Token verdoppelt. Außerdem gab es eine Veränderung in der Llama-Transformer-Architektur. In den Attention-Modulen werden zwei Werte, die normalerweise immer wieder berechnet werden, geteilt, was bedeutet, dass jedes Attention-Modul Zugriff auf die gleichen Werte hat. Dies führt zu geringfügig schlechteren Ergebnissen, jedoch zu einer deutlichen Leistungssteigerung.

Das Llama-Modell wiederum besteht aus einer Decoder-only-Transformer-Architektur. Der **Decoder-only-Transformer** besteht lediglich aus Decoder-Transformer-Blöcken, wie der Name bereits vermuten lässt. Bei dieser Architektur ist der anfängliche Input des Decoders die Eingabe des Nutzers. Auf diese Eingabe wird dann immer wieder das neu generierte Token drauf konkateniert. Auf diese Weise wird die Eingabe nicht explizit von dem bereits generierten getrennt, sondern beides wird als gleicher Kontext verwendet.

Das Besondere an Llama ist, dass es nur auf frei verfügbaren Daten trainiert wurde und dass das Modell Open Source ist. Der Datensatz besteht aus English Common Crawl [67%], f C4 [15%], Github [4.5%], Wikipedia [4.5%], Gutenberg and Books3 [4.5%], ArXiv [2.5%], und Stack Exchange [2%].

Code Llama ist eine Open Source-Software, die unter allen verfügbaren Modellen am besten in multilingualen Benchmarks abschneidet. Multilingual bedeutet die Verwendung mehrerer Programmiersprachen. Diese Eigenschaften machen es sehr geeignet für diese Arbeit.

2.4 Code2Vec

Code2Vec ist eine im Jahre 2018 von Alon et al. entwickelte Modellarchitektur, die Quellcode in einen semantischen Vektor kodiert [15]. Nach den Erfolgen in NLP, natürliche Sprache in semantische Vektoren zu kodieren, entstand der Wunsch, auch Quellcode in semantischen Vektoren abzubilden. Mit diesen Vektoren können dann wieder viele verschiedene Aufgaben gelöst werden. Das motivierende Beispiel bei *Code2Vec* ist die Vorhersage eines sinnvollen Namens für eine

Funktion. Die Architektur, welche in Abbildung 2.10 dargestellt ist, wird im Folgenden erläutert.

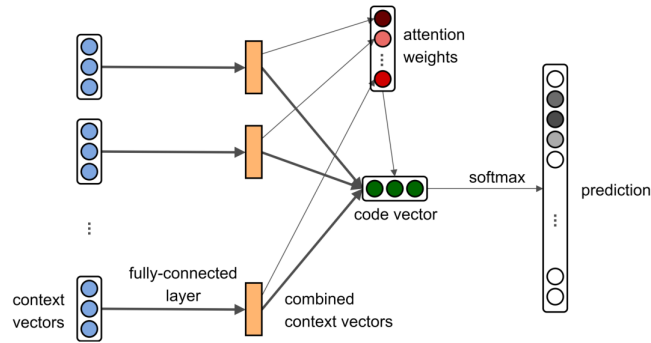


Abbildung 2.10: *Code2Vec*-Architektur entnommen aus [15]

Alon et al. fanden heraus, dass 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 Input-Vektoren (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 endgültige Vektor wird als lineare Kombination aus den Output-Vektoren und den Attention-Weights berechnet. Sei $h_1, \dots, h_n \in \mathbb{R}^d$ die Output-Vektoren 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, bei gegebenem Code-Vector bzw. Quellcodeauszug ein bestimmtes Label in natürlicher Sprache vorherzusagen. Nachdem das Training abgeschlossen ist, kann es ein Label für einen Quellcode vorhersagen, welches nicht im Trainingssatz enthalten ist. Die Trainingsart ist demnach überwachtes Lernen, was eine Aufbereitung der Daten benötigt. Das Modell kann lediglich die Labels vorhersagen, die es vorher im Training gesehen hat.

2.5 T-Distributed Stochastic Neighbor Embedding

T-Distributed Stochastic Neighbor Embedding (t-SNE) ist ein Algorithmus, welcher zum Visualisieren von hochdimensionalen Daten eingesetzt wird [13]. 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. Dabei versucht der Algorithmus, die Nachbarschaftsverhältnisse der Datenpunkte nicht in Mitleidenschaft zu ziehen.

Im Folgenden wird der Algorithmus skizziert und danach wird aufgezeigt, was bei der effektiven Verwendung von *t-SNE* zu beachten ist. Die hochdimensionalen Datenpunkte werden mit **H** und die niedrig dimensional Datenpunkte mit **N** bezeichnet. Der erste Schritt des Algorithmus ist es, jedem Datenpunktpaar im Datensatz **H** einen Ähnlichkeitsscore zuzuweisen. Dieser wird berechnet, indem man zuerst die euklidische Distanz von jedem Datenpaar berechnet und dann das Ergebnis in eine Wahrscheinlichkeitsverteilung eingibt. Dies führt zu einer Normalisierung des Wertes. Das Ergebnis ist dann eine Tabelle mit einem Ähnlichkeitsscore für jedes Datenpaar in **H**.

Als Nächstes werden die Datenpunkte zufällig in der niedrigen Dimension **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 **N**, diesmal wird aber die studentische t-Verteilung als Wahrscheinlichkeitsverteilung genommen.
2. Verschiebe die Datenpunkte von **N** um einen kleinen Wert in die Richtung, die den Unterschied der Ähnlichkeitsscores von **H** und **N** minimiert.

Nach T Wiederholung ist der Ähnlichkeitsscore von **N** und **H** nahe bei einander, d.h. die Nachbarschaftsverhältnisse von **N** und **H** sind nun ähnlich.

Wattenberg et al. haben untersucht, wie man *t-SNE* sinnvoll anwendet und welche Schlüsse man aus der Visualisierung ziehen kann [17]. Sie fanden heraus, dass die Wahl der Parameter für das Ergebnis eine wichtige Rolle spielt. Die wichtigsten Parameter sind die Iterationen $T \in \mathbb{N}$ und die Perplexity $P \in \mathbb{N}$.

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.

Die Perplexity kann intuitiv als Schätzung für die Anzahl an nahen Nachbarn, die jeder Datenpunkt hat, gesehen werden. Die Suche nach einer geeigneten Perplexity ist schwieriger, da wir die hochdimensionalen Nachbarschaftsbeziehungen häufig nicht kennen. Van der Maaten et al., welche *t-SNE* entwickelt haben, empfehlen eine Perplexity $P \in \{5, 6, \dots, 50\}$ [13]. Außerhalb dieses Bereichs kann es zu unerwünschten Ereignissen kommen. Bei $P = 2$ haben Wattenberg et al. herausgefunden, dass *t-SNE* bei einer zufällig generierten Datenwolke fälschlicherweise kleine Gruppierungen (engl. cluster) bildet. Falls P größer ist als die Anzahl der Datenpunkte, ist das Ergebnis überhaupt nicht interpretierbar. Es ist daher ratsam, mehrere Werte für P zu verwenden, um sicherzustellen, dass *t-SNE* keine falschen Nachbarschaftsbeziehungen darstellt.

Wattenberg und Kollegen stellten außerdem fest, dass sowohl die Information des Durchmessers eines Clusters als auch die Abstände zwischen einem Cluster und einem anderen durch *t-SNE* vollständig verloren gehen. Nach Betrachtung der *t-SNE* Ausgabe kann daher keine Aussage über den Durchmesser eines Clusters, die Position des Clusters und die Lagebeziehungen zwischen Clustern getroffen werden.

Der *t-SNE* Algorithmus ist ein wichtiges Tool, um qualitative Aussagen über Daten zu treffen. Es ist jedoch wichtig, mehrere Parameter auszuprobieren. Es kann nur eine Aussage über die Existenz von Clustern gemacht werden, nicht über ihre geometrischen Gegebenheiten. Wenn diese Bedingungen beachtet werden, kann *t-SNE* ein mächtiges Visualisierungs-Tool sein, um eine Intuition für die Anordnung der Datenpunkte zu erlangen.

3 Methodik

3.1 Datensatz

Im maschinellen Lernen hat der Datensatz bzw. die Trainingsdaten den größten Einfluss auf die Güte des Modells. Wir haben uns für die Open Source-Standard-C-Bibliothek von GNU entschieden. Die Sprache C wurde gewählt, da sie die zweitpopulärste Sprache nach C++ ist, die zu Maschinencode kompilierbar ist. Die GNU-Bibliothek wurde aufgrund der hohen Qualität des Quellcodes ausgewählt, da das Projekt seit 1987 existiert und die Community jederzeit auf Fehler im Quellcode hinweisen kann. Zum anderen wird die Bibliothek weitgehend in vielen Applikationen eingesetzt.

Der wichtigste Aspekt ist, dass man mit diesem Datensatz die Ergebnisse des trainierten Modells vergleichen kann, da die C-Bibliothek im POSIX Standard festgelegt ist und mehrfach implementiert wurde. Dadurch erhält man mehrere hochqualitative Quellcodeprojekte, die die gleiche Semantik aufweisen. Die unterschiedlichen Implementierungen werden zu unterschiedlichem Assemblercode kompiliert. Deswegen kann mit unterschiedlichen Standard-C-Bibliotheken, die den POSIX Standard implementieren, getestet werden, ob das finale Modell beide Implementierungen als sehr ähnlich klassifiziert.

3.2 Datenpipeline

Der große praktische Teil der Arbeit bestand darin, eine große Menge an Daten in unterschiedliche Darstellungen umzuwandeln. Jeder Zwischenschritt wurde gespeichert, um nicht jede Darstellung erneut generieren zu müssen. Im Folgenden wird die generelle Architektur, wie Daten verarbeitet werden (engl. pipeline), vorgestellt, um einen Überblick über den praktischen Teil dieser Arbeit zu geben.

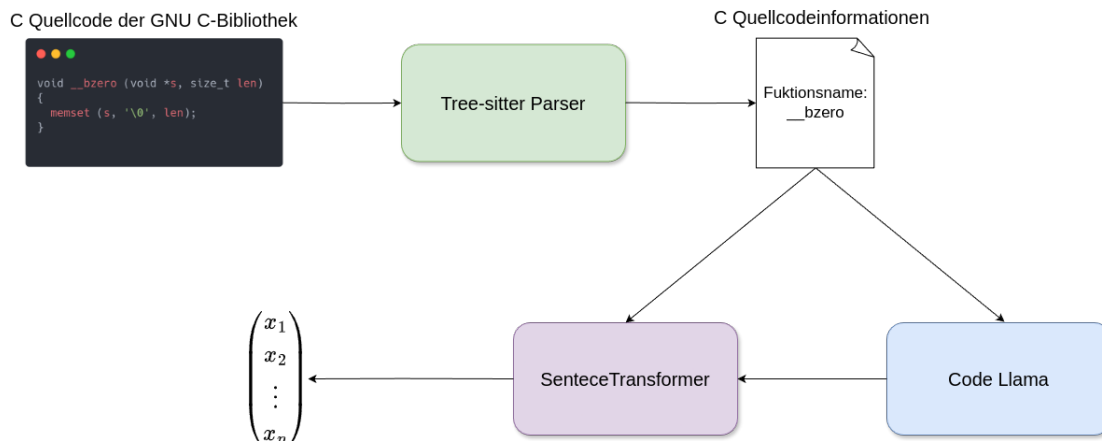


Abbildung 3.1: Die Datenpipeline dargestellt, dabei sind Vierecke mit abgerundeten Ecken Tools, die Daten in eine andere Darstellung umwandeln.

Die Rohdaten entsprechen dem unverarbeiteten C-Quellcode aus der GNU-Standard-C-Bibliothek. Wir benötigen nicht alle Teile des Quellcodes, sondern nur bestimmte. Wir betrachten ausschließlich nur Funktionen, d.h. wir wollen bspw. alle Datenstrukturen, die in dem Quellcode definiert werden, aus dem Quellcode entfernen. Die Zerlegung und Umwandlung des Inputs in sinnvolle Teile wird Parsing genannt.

In dieser Arbeit wurde **Tree-sitter**³ 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 bspw. für die Hervorhebung von Syntax verwendet. Generell kann *Tree-sitter* jedoch für alle Anwendungen benutzt werden, die Informationen aus Quellcode verarbeiten wollen. Aus dem von *Tree-sitter* generierten Syntaxbaum kann dann jegliche gewünschte Information entnommen werden.

Die Quellinformationen werden immer zunächst gespeichert, damit die Verarbeitung beim nächsten Mal an diesem Punkt beginnen kann. Die Quellinformationen beziehen sich immer auf eine Funktion, deswegen bestehen die Daten aus einem Tupel, der aus je einer Funktion und den gewünschten Quellinformationen besteht.

Danach gibt es eine Abzweigung in der Datenpipeline, entweder werden die Quellinformationen von Code Llama nochmal in natürlicher Sprache beschrieben und dann in den *SentenceTransformer* eingegeben oder die Quellinformation wird direkt in den *SentenceTransformer* eingegeben. Schließlich erhält man wieder eine Datenmenge, die aus Tupeln besteht. Ein Tupel besteht aus dem

³<https://tree-sitter.github.io/tree-sitter/>

Funktionsnamen und dem zugewiesenen semantischen Embedding. Bei dieser Architektur kann mühelos jedes Tool ausgewechselt werden, solange die Ausgabeformate eingehalten werden.

3.3 Stabilität von SentenceTransformer

Im vorherigen Abschnitt haben wir festgestellt, dass der endgültige Schritt für alle Daten der *SentenceTransformer* ist, deshalb bildet er das Herzstück der Datenpipeline. In dieser Arbeit sollen verschiedene semantische Beschreibungen des Quellcodes in natürlicher Sprache verglichen werden. Es ist wichtig, dass alle anderen Elemente in der Datenpipeline konstante Ergebnisse liefern und nicht schwanken. Aus diesem Grund wird im Folgenden untersucht, wie sich der *SentenceTransformer* bei selber Eingaben verhält. Im Optimalfall sollte der SentenceTransformer bei selbem Input dasselbe Ergebnis liefern.

Um dies zu überprüfen, wurden $n = 100$ Code Llama Quellcode-Erklärungen zufällig ausgewählt und anschließend in den SentenceTransformer $m = 100$ Mal eingegeben. Dabei beträgt der höchste Abstand von zwei Vektoren, die aus der gleichen Erklärung resultiert sind $d = 6.661338 \cdot 10^{-16}$. Der Abstand ist ausreichend klein, um ihn in der Evaluation zu vernachlässigen. Der *SentenceTransformer* ist daher für die Anwendung in dieser Arbeit ausreichend stabil.

3.4 Funktionskommentare

Bevor ein Programm kompiliert wurde, 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 zu einer Funktion beschreibt die Kernfunktion der Prozedur, d.h. den Input, den Output und die Umwandlung. Dieser Kommentar kann man dann in den SentenceTransformer eingeben und so in einen semantischen Vektorraum abbilden. Das Parsen der Kommentare wurden mit *Tree-sitter* realisiert, wie im Abschnitt zur Datenpipeline erwähnt. Dabei gab es eine große Designentscheidung zu treffen, welche Kommentare in einer Funktion berücksichtigt werden sollen. Im Rahmen dieser Arbeit werden ausschließlich Kommentare verwendet, die direkt über der Funktion in stehen. Da die einzeiligen Kommentare innerhalb der Funktion meistens keinen großen semantischen Wert haben, sondern auf Gefahren oder Designentscheidungen hinweisen. Hierbei muss man aufpassen, dass der Prozess des Parsen nicht zu speziell an die vorliegenden Datensatz angepasst wird, sonst verliert er seine Allgemeingültigkeit.

3.5 Code2Vec

Das Besondere an *Code2Vec* ist, dass der Quellcode in einem abstrakten Syntaxbaum kodiert wird. Dies ermöglicht die Verwendung aller Quellcodeinformationen, die im Quellcode enthalten sind. Anschließend wird das Modell darauf trainiert, eine Eigenschaft in natürlicher Sprache über den Quellcode vorherzusagen.

Bei dieser Herangehensweise, semantische Vektoren zu erzeugen, wird also kein *Sentence-Transformer* verwendet, sondern die Vektorraum werden durch Training des *Code2Vec* Modells als Nebenprodukt erzeugt. Wie in dem [15] wird das Modell auch hier darauf trainiert, Funktionsnamen vorherzusagen. Die Implementierung des *Code2Vec* Modells nutzte Java als Datensatz, weshalb einige Anpassungen notwendig waren, um das Modell auf C-Code zu trainieren.

Da das *Code2Vec* Modell auf überwachtem Lernen basiert, muss der Datensatz vor dem Training zunächst aus dem rohen Quellcode aufbereitet werden.

3.5.1 Adaption auf C

Um das *Code2Vec* Modell zu trainieren, ist es notwendig, aus dem rohen Quellcode einer Funktion jeweils einen abstrakten Syntaxbaum und den Funktionsnamen zu extrahieren. Alon et al. stellen ein Tool für Java zur Verfügung, welches das Extrahieren von abstrakten Syntaxbäumen und Funktionsnamen aus Java-Quellcode ermöglicht.

Mit dem *ASTminer* von Kovalenko et al. können abstrakte Syntaxbäume und Funktionsnamen extrahiert werden [19]. Das Tool wurde vom JetBrains-Research-Team entwickelt, um Quellcode in abstrakte Syntaxbäume (AST) zu kodieren, die als Input-Format für maschinelles Lernen dienen.

Die Implementierung des *Code2Vec*-Modells braucht ein bestimmtes Format, den der Datensatz aufweisen muss. Ein Trainingsbeispiel besteht jeweils aus einem Funktionsnamen, dem Label und einer Liste von Kontexten, welche den AST repräsentieren. Ein Kontext besteht aus einem Starttoken, einem Endtoken und einer Pfadbeschreibung vom Starttoken bis zum Endtoken.

Dieses *Code2Vec*-Format kann durch eine zusätzliche Option in der Konfiguration des *ASTminers* generiert werden. Obwohl diese Option mit „code2vec“ betitelt ist, ist der Output des *ASTminers* nicht direkt von *Code2Vec* verwendbar. Das Tool weist nämlich jedem Token eine einzigartige Zahl zu und verwendet dann im Datensatz nur noch die Zahl. Dieses Format reduziert zwar den Speicheraufwand, aber es wird von *Code2Vec* nicht als valide Eingabe akzeptiert. Demnach mussten noch die Nummern wieder zu den passenden Token umgewandelt werden.

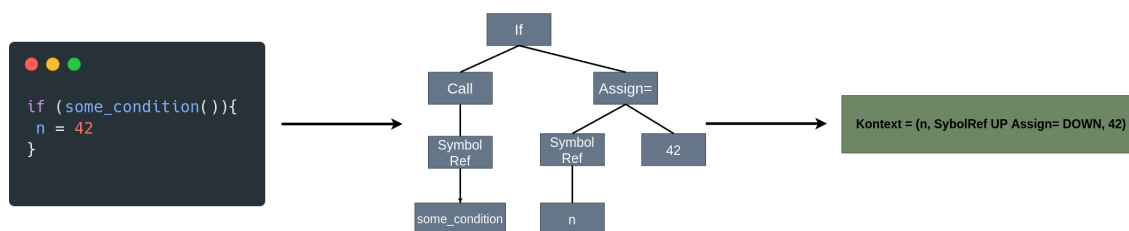


Abbildung 3.2: Extrahierung eines Kontexts aus einem C-Quellcode

Da es unterschiedliche Konventionen für Funktionsnamen gibt, wie z.B. Camel-Case und Snake-Case, normalisiert der *ASTminer* die Funktionsnamen. Dadurch sind die Trainingsdaten

unabhängig von den jeweiligen Quellcodekonventionen.

Beispiel 3.1

Snake-Case: `funktions_name` → **funktions|name**

Camel-Case: `funktionsName` → **funktions|name**

Code2Vec kann nach dem Training für jede Funktion einen hochdimensionalen Vektor ausgeben, der die Semantik der Funktion beschreiben soll. Um mit *Code2Vec* diesen Vektor generieren zu können, 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 nicht mehr eindeutig dem initialen Funktionsnamen zugeordnet werden, da durch die Normalisierung Informationen verloren gehen. Zur Vergleichbarkeit von *Cod2Vec* mit anderen Ansätzen wie z.b. Funktionskommentaren ist es notwendig, die normalisierten Namen wieder zu den ursprünglichen Namen zurückzuführen. Aufgrund dessen mussten im Quellcode von *ASTminer* Änderungen vorgenommen werden, sodass zu jeder Position eines Trainingsbeispiels im Datensatz der ursprüngliche Funktionsname zugeordnet werden kann.

3.5.2 Training

Das Ziel beim Training war es, dieselbe Qualität wie Alon et al. zu erhalten. Das heißt, dieselben Ergebnisse für Java auch für C-Quellcode zu replizieren. Bei der Erstellung eines Datensatzes, der aus dem Modell resultiert, können bestimmte Gütekriterien beim Trainieren von Modellen vernachlässigt werden. Eine Problemstellung beim Trainieren von 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 berücksichtigt werden, da das Ergebnis kein fähiges Modell ist, sondern semantische Vektoren für den Datensatz. 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 dasselbe wie bei Alon et al. ist, wurden alle Trainingsparameter gleich gelassen. Die Ergebnisse nach der 84 Epoche sind: **Precision:** 65.6, F_1 : 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, F_1 : 58.4, **Recall:** 54.4. Dabei ist hervorzuheben, dass unser Test- und Trainings-Datensatz derselbe ist und der Test-Datensatz von *Code2Vec* verschieden ist von dem Trainings-Datensatz. Die Vorgehensweise ist normalerweise grob fahrlässig, 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 in [15] und können diese mit den anderen vorgestellten Ansätzen vergleichen.

3.6 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. Die Extrahierung von Funktionsnamen ist 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 immer wieder angepasst werden. Deswegen wurde hier für die einfache Erweiterung des Programms, *Tree-sitter* verwendet. *Tree-sitter* bietet Parser für eine Vielzahl von Sprachen an.

3.7 Code-Llama-Erklärungen

Large Language-Models werden immer besser, Quellcode selbst zuschreiben, zu verstehen und zusammenzufassen. Eine Zusammenfassung des Quellcodes in natürlicher Sprache spiegelt die Semantik des Quellcodes wider. 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. Code Llama generiert Quellcodeerklärungen, indem Code Llama eine präzise Fragestellung und den gesamten Quellcode einer Funktion erhält. Die Fragestellung wurde aus der Implementierung von CLAP [11] übernommen, diese haben sich auch damit beschäftigt, wie die Fragestellung formuliert werden muss, um semantikleiche und präzise Quellcodeerklärungen zu erhalten.

Damit der Code-Llama-Output deterministisch ist, muss der Temperaturparameter auf null gesetzt werden. Der Parameter ist Teil einer Zufallskomponente bei der Auswahl des nächsten Tokens, ist dieser auf null, wird der Token mit dem höchsten Score gewählt, dieser bleibt bei gleicher Eingabe immer derselbe. Leider ist der Determinismus trotzdem nicht garantiert, wie die Studie von Astekin et al. heraus fand [20]. Laut Astekin et al. könnte ein Grund dafür sein, dass bei den vielen Fließkommaoperationen, die für die Berechnung eines Code-Llama-Outputs erforderlich sind, Rundungsfehler entstehen. Diese Erkenntnisse stammen aber aus einem anderen Anwendungszweck, nämlich für Log-Parsing.

Die Stabilität mit Temperatur Null sollte daher 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 dem gleichen Quellcode resultiert sind, $d = ?$. Dieser Abstand ist ausreichend gering, um ihn in der Evaluation zu vernachlässigen. Code Llama ist also für die Anwendung in dieser Arbeit ausreichend stabil.

4 Ergebnisse

Um die Frage zu beantworten welche der vorgestellten Strategien (Funktionsnamen, Funktionskommentare, *Code2Vec* und Code-Llama-Erklärungen) zu Erzeugung von Quellcode-Embeddings, die besten semantischen Embeddings liefern, wurden drei verschiedene Evaluierungsmethoden verwendet.

4.1 Befragung von Experten

4.2 Quantitative Evaluierung

4.3 Qualitative Evaluierung

5 Diskussion

6 Fazit

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

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

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

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

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

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

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

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

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.
- [2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.
- [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,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.
- [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 detection,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA ’22. ACM, Jul. 2022, p. 1–13.
- [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.
- [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.
- [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. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, “DIRE: A Neural Approach to Decompiled Identifier Naming,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2019, pp. 628–639.
- [10] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, “Malware detection by eating a whole EXE,” in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, ser. AAAI Technical Report, vol. WS-18. AAAI Press, 2018, pp. 268–276.

- [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,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA ’24. ACM, Sep. 2024, p. 503–515.
- [12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.
- [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.
- [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,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 1–29, Jan. 2019.
- [16] R. Kruse, S. Mostaghim, C. Borgelt, C. Braune, and M. Steinbrecher, *Computational Intelligence: A Methodological Introduction*. Springer International Publishing, 2022.
- [17] M. Wattenberg, F. Viégas, and I. Johnson, “How to use t-sne effectively,” *Distill*, 2016. [Online]. Available: <http://distill.pub/2016/misread-tsne>
- [18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL-HLT (1)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186.
- [19] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, “Pathminer: a library for mining of path-based representations of code,” in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 13–17.
- [20] 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.
- [21] 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.

- [22] 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>
- [23] 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. Kardaş, 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>