Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Automatische Extraktion von Funktionen aus klassifizierten E-Mails

Alexander Tiessen

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Verena Ebert, M.Sc. Gernot Haug, Dipl.-Inform. (FH), IT.TEM GmbH
Beginn am:	12. Dezember 2018
Beendet am:	12. Juni 2019

Kurzfassung

In vielen kleinen und großen Unternehmen hat die Zahl der eintreffenden E-Mails ein hohes Niveau erreicht. Die Bearbeitung dieser E-Mails ist mit einem hohen Zeit- und Kostenaufwand verbunden. Um diesen Aufwand zu reduzieren, wurde ein System entwickelt, das in zwei Teilschritten arbeitet. Zunächst findet eine Klassifikation statt, in der die E-Mails nach der Absicht des Senders klassifiziert werden. Hierfür wurden mehrere Klassifikatoren des maschinellen Lernens miteinander verglichen, um ein bestmögliches Ergebnis zu erreichen. Im zweiten Schritt wurden für geeignete Klassen Parameter definiert, die die Absicht des Senders genauer beschreiben. Die Werte der Parameter werden dann aus den E-Mails extrahiert. Dies wurde mit Hilfe von Natural-Language-Processing-Methoden in Verbindung mit regelbasierten Ansätzen gelöst. In dieser Arbeit wurde ein selbst erstellter E-Mail-Korpus aus ca. 1.500 E-Mails verwendet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API application programming interface. 15

IDE integrated development environment. 31

JSON JavaScript object notation. 15

KNN Künstliches Neuronales Netz. 7

REST representational state transfer. 15

SVM support vector machine. 19

TSV tab-separated values. 37

YAML YAML Ain't Markup Language. 34

1 Einleitung

In vielen Unternehmen treffen täglich zahlreiche E-Mails ein, die zeitnah verarbeitet werden müssen. Diese stammen von internen Mitarbeitern, als auch von externen Absendern. Um diese manuell zu sortieren und zu bearbeiten, ist ein hoher personeller und wirtschaftlicher Aufwand vonnöten. Deswegen ist die Bestrebung dieser Arbeit, geeignete Teile des Prozesses zu automatisieren. Dabei wird der Fokus auf die Erkennung der Absicht des Senders einer E-Mail gelegt. Um jenes zu erreichen, kann die Aufgabe in zwei Teilschritte zerlegt werden:

Zunächst sollen die E-Mails nach der Funktion (der Absicht des Senders) klassifiziert werden. Hierfür werden an den vorliegenden Datensatz angepasste Funktionsklassen definiert. Diese sind unter anderem eine Termineinladung, eine Abwesenheitsnotiz oder eine projektbezogene Information. Nach diesen Klassen werden dann die E-Mails durch Algorithmen des maschinellen Lernens klassifiziert. Mehrere Ansätze sollen dafür getestet werden. Zum einen sollen mehrere Modelle der Bibliothek Scikit-learn verwendet werden. Zum anderen soll das von Google-Mitarbeitern entwickelte Netz BERT verwendet werden.

Nach der Klassifizierung der E-Mails müssen noch deren Parameter extrahiert werden. Diese sollen die Absicht des Senders genauer repräsentieren. Es wird also für jede Funktionsklasse ein (eigenes) Tupel von Parametern definiert, das dann aus den E-Mails extrahiert werden soll. Dies soll mit Hilfe einer Kombination aus regelbasierten und neuronalen Ansätzen gelöst werden. Schließlich wird die Klasse der E-Mails, inklusive ihrer Parametern, in einer Datei im JavaScript object notation (JSON)-Format gespeichert. Daraus könnte in Zukunft ein vollwertige representational state transfer (REST) application programming interface (API) entwickelt werden.

Die Arbeit ist in folgender Weise gegliedert: In Kapitel 3 werden die wichtigsten Konzepte des maschinellen Lernens, insbesondere der künstlichen neuronalen Netze erklärt. In Kapitel 4 wird der verwendete Datensatz beschrieben und es wird erläutert, wie dieser verarbeitet wird. Danach wird in Kapitel 5 die Implementierung aller relevanten Module erläutert. In Kapitel 6 wird die Performance der trainierten Modelle dargelegt. Schließlich fasst Kapitel 7 die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Verwandte Arbeiten

Für die Aufgabe der Textklassifikation gibt es seit einigen Jahren bereits genügend Publikationen und Erkenntnisse. Viele davon nutzten ebenfalls maschinelles Lernen aus Beispielen (supervised learning). Allerdings beziehen sich nur einige davon auf E-Mailtexte.

Cui et al. ([CMS+05], 2005) haben sich damit beschäftigt, mithilfe von KNNs E-Mailtexte in interessant und uninteressant einzustufen. Dies ist Vergleichbar mit dem Versuch, entsprechende E-Mails als Spam zu identifizieren. In dieser Arbeit wird ein klassisches Feedforward-Netz mit verschiedenen Hyperparametern in Verbindung mit dem Backpropagation-Algorithmus verwendet. Dafür wurde ein Datenbestand von ca. 1.500 E-Mails von privaten E-Mail-Accounts zusammengetragen.

Zum einen wurde Untersucht, welchen Einfluss die Anzahl der Neuronen der Hidden Layer auf die Performance des Netzes hat. Dafür wurde Precision und Recall (Definitionen in Abschnitt 3.3 auf Seite 24) als Maß der Performance genommen. Es wurde deutlich, dass viele neun getesteten Kombinationen aus Layergrößen gute Ergebnisse erzielen. Es gibt aber auch Kombinationen, die sehr schlecht performen.

Ein weiterer Aspekt, der getestet wurde, ist die Selektion der Features auf die Performance des Modells. Diese stammen aus zwei Quellen: Zum einen wurden sieben strukturelle Features verwendet. Dies sind Features, die aus dem Header einer E-Mail abgeleitet werden können und Meta-Informationen enthalten (Sie definieren, ob Anhänge enthalten sind und ob die E-Mail weitergeleitet wurde etc.). Zum anderen handelt es sich um Text-Features, welche direkt aus dem Text abgeleitet werden. Die Anzahl der letzteren wurde bei Tests analysiert, um deren Einfluss auf die Performance des Modells zu überprüfen. Es ergab sich, dass die Performance bei einer Zahl von unter 200 Text-Features sehr instabil ist. Dieses Problem bestand bei der Verwendung von einer größeren Zahl nicht mehr. Die besten Ergebnisse wurden mit 1.313 insgesamt verwendeten Features, die größte hier getestete Menge, erzielt. Allerdings waren die Ergebnisse mit 508 Features nur geringfügig schlechter. Hierbei wurde jedoch über die Hälfte an Speicherbedarf und Trainingsdauer (im Vergleich zu 1.313 verwendeten Features) eingespart.

Ein letzter Hauptaspekt, der in dieser Arbeit untersucht wurde, ist der Einfluss einer Hauptkomponentenanalyse (Principal Component Analysis (PSA)) der Features auf die Performance des Modells. Mit Hilfe dieser mathematischen Analyse der Features können relevantere Features aus den bestehenden berechnet werden, welche eine geringere Dimension aufweisen. Eine Reduktion der Dimensionen der Features ist bei richtiger Durchführung sinnvoll, da ein neuronales Netz mit weniger komplexeren Daten trainiert werden kann. Die Tests ergaben eine Verbesserung der Performance bei einer Verwendung der Hauptkomponentenanalyse. Im Durchschnitt hat sich Precision und Recall von 85% auf 93% verbessert. Aus den Daten kann man ableiten, dass die Nutzung von der Hauptkomponentenanalyse, gerade mit wenigen Dimensionen, sinnvoll ist und stabilere Ergebnisse liefert, als ohne Verwendung dieser. Mit 37 Dimensionen wurde das beste Ergebnis erzielt. Durch diese Reduktion der Dimensionen ist der Speicherbedarf und der Berechnungsaufwand 90% kleiner.

3 Relevante Konzepte des maschinellen Lernens

In diesem Kapitel wird auf die für diese Arbeit relevanten Konzepte des maschinellen Lernens eingegangen. Dies sind neben der *Support Vektor Machine* hauptsächlich die *künstlichen neuronalen Netze (KNN)* und wie diese trainiert und verwendet werden.

3.1 Support Vector Machine

Die *support vector machine (SVM)* ist ein Algorithmus aus dem Bereich des maschinellen Lernens, der für die Klassifikation sowie die Regression verwendet werden kann. Hier wird der Feature-Raum als n-dimensionaler Vektorraum betrachtet, bei dem n die Anzahl der Vektoren entspricht. Der Algorithmus versucht bei einer Klassifikationsaufgabe mit einer Hyperebene d die Klassen der Datenpunkte aufzuteilen. Die zugehörige Klasse der Datenpunkte muss natürlich vorher bekannt sein. Diese Ebene teilt den Raum in diese Klassen auf. Bei der Bestimmung der Ebene wird darauf geachtet, dass sie einen maximalen Abstand zu den bestehenden Datenpunkten aus beiden Klassen besitzt. Durch diesen maximalen Abstand wird die Chance erhöht, dass ein neuer Datenpunkt ebenfalls richtig klassifiziert wird, d. h., dieser auf der richtigen Seite der Ebene liegt. Die Datenpunkte, die der Ebene am nächsten sind, werden Support-Vektoren genannt. Die Lage der Hyperebene im Raum wird ausschließlich von diesen Punkten beeinflusst. Beim Hinzufügen eines neuen Datenpunktes ist dieses also nur für die Lage der Ebene relevant, wenn er näher zur Ebene als die bisherigen Support-Vektoren ist. Oft ist es nicht möglich die Datenpunkte linear mithilfe einer Ebene zu teilen. Hier kann sich eines Tricks beholfen werden: Der ganze Vektorraum wird in eine höhere Dimension transformiert. In dieser sind die Datenpunkte nun korrekt linear teilbar. Somit kann hier eine teilende Ebene bestimmt werden, welche nach der Rücktransformation zu einer Hyperfläche wird. Sie ist also keine lineare Ebene mehr. In dem Beispiel von [Ray17] wird diese Ebene beispielsweise von einer Geraden zu einem Kreis. Diese Generation einer Hyperebene in einer höheren Dimension wird als *Kernel-Trick* bezeichnet [Ray17] [Gan18] [Ope14]. Eine SVM ist ein beliebter Klassifikator, der mit vergleichsweise wenig Trainingsdaten gute Ergebnisse erzielen kann und bereits in vielen Bibliotheken implementiert wurde. In dieser Arbeit wird die Implementierung der Python-Bibliothek Scikit-learn verwendet.

3.2 Künstliches neuronales Netz

Künstliche neuronale Netze (KNN) werden verwendet, um komplexe Funktionen zu erlernen. Das betrifft vor allem Funktionen, die unbekannt oder zu komplex sind, um sie algorithmisch zu definieren. KNNs sind gut geeignet, um aus Beispielen zu lernen, Muster zu erkennen und zu verallgemeinern [SHG90]. Die Idee eines KNNs entstammt dem menschlichen Gehirn. Wie der Begriff bereits erahnen lässt, sind künstliche Neuronen der zentrale Bestandteil eines solchen

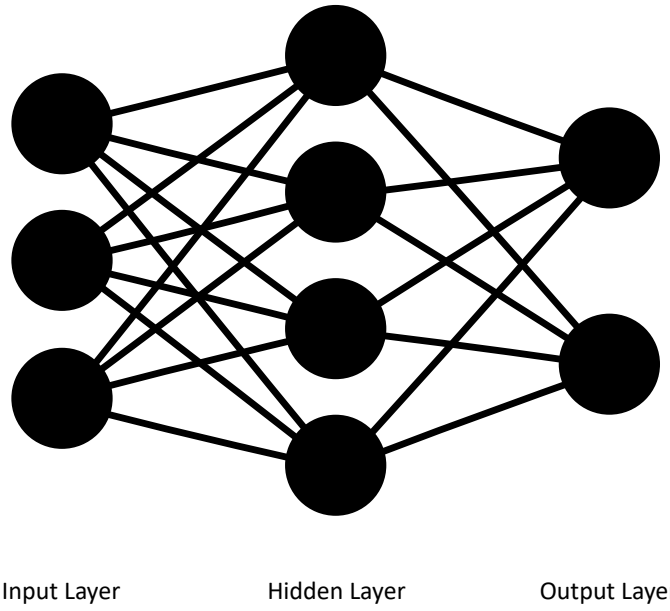

Abbildung 3.1: Beispielstruktur eines künstlichen neuronalen Netzes

Netzes. Diese sind schichtweise angeordnet, welche untereinander Verbunden sind. Durch die Verarbeitung der Schichten, kann eine Ausgabe erzeugt werden [Moe17] [Sch15] [Tie18]. Dies wird in Abbildung 3.2 [Neg11] illustriert.

Input Layer Die erste Schicht wird *Eingabeschicht* oder auch *Input Layer* genannt. Sie bekommt ihre Werte aus der realen Welt. Die Anzahl der Eingabewerte entspricht in der Regel der Anzahl der Neuronen der Eingabeschicht. Die Werte können aus allen möglichen Quellen stammen: z.B. aus Helligkeitswerten von Pixeln eines Bildes, Messdaten aus der Umgebung oder aus einem Text [Tie18].

Hidden Layer Die inneren Schichten interagieren nicht mit der Außenwelt und werden deshalb *verdeckte Schicht* oder auch *Hidden Layer* genannt [Neg11].

Output Layer Das resultierende Ergebnis wird durch die *Ausgabeschicht* (eng. Output Layer) bereitgestellt. Für die Kodierung des Outputs gibt es verschiedene Möglichkeiten. Oft wird das so genannte *One-Hot-Encoding* verwendet. Hierbei wird jedem Ausgabeneon ein Attribut bzw. eine Klasse zugewiesen. Bei einer Netzausgabe wird das Neuron mit dem größten Wert gewertet. Idealerweise gibt es nur ein Neuron, mit einem Wert ungleich Null [Lug03].

Künstliches Neuron Jedes KNN besteht aus untereinander verknüpften, künstlichen Neuronen. Sie bilden die zentrale Recheneinheit und bestehen im Wesentlichen aus drei Teilen:
Die *Eingänge (Inputs)* sind Grundlage der Berechnungen. Sie sind mit dem Ausgang der vorherigen

Schicht verbunden (außer bei den Neuronen der Input Schicht, hier gibt es keine vorherige Schicht). Jeder dieser Eingänge eines Neurons hat ein sogenanntes Gewicht, mit dem der Grad des Einflusses des Eingangs auf die sog. Netzeingabe des Neurons (gesammelte Eingabe eines Neurons) bestimmt werden kann.
Die *Übertragungsfunktion* (auch Propagierungsfunktion) errechnet unter Berücksichtigung der jeweiligen Gewichte aus den Eingängen die Netzeingabe des Neurons. Diese wird meist durch die gewichtete Summe der Eingabewerte definiert.
Die *Aktivierungsfunktion* errechnet aus der Netzeingabe die Ausgabe des Neurons. Hier kommen verschiedene Funktionen zum Einsatz. Für binäre Neuronen wird meist eine Schwellenwertfunktion (Hard-limit function) verwendet, diese wird aktiviert, sobald der Wert einen vorher definierten Schwellwert überschreitet. Diese Funktionen nehmen in den meisten Anwendungen die Werte 0 und 1 (Step function) oder -1 und 1 (Sign function) an (Englische Begriffe aus [Neg11]). Werte ungleich 0 sind oft für das Verhalten beim Trainieren förderlich [LC12]. Für kontinuierliche Werte wird meist eine sigmoide Funktion verwendet. Dies ist meist die logische Sigmoid-Funktion selbst. Sie bildet alle Eingabewerte auf das geschlossene Intervall $[0, 1]$ ab und wird in Gleichung (3.1) [Lug03] definiert. Hier ist λ ein Parameter, mit dem die Krümmung der Funktion bestimmt werden kann.

$$\varphi_{\lambda}^{sig}(v) = \frac{1}{1 + e^{-\lambda v}} \tag{3.1}$$

Eine weitere sigmoide Funktion, die häufig verwendet wird, ist der Tangens-Hyperbolicus. Der Hauptunterschied hierbei ist der Wertebereich, der im Intervall $[-1, 1]$ liegt. Ein Vorteil für die Nutzung einer sigmoiden gegenüber einer nicht stetigen Funktion ergibt sich bei der Mustererkennung eine KNNs mit Neuronen, die eine solche Funktion nutzen. Eine kleine Änderung der Eingänge hat auch nur eine kleine Änderung des Ausgangs zur Folge. Dies ist für einen Lernalgorithmus wichtig, der den Fehler des Netzes schrittweise verringert (z. B. Gradientenverfahren). Außerdem muss die Aktivierungsfunktion differenzierbar sein, da sie für die Fehlerbestimmung beim Backpropagation (siehe Abschnitt 3.2.2) differenziert werden muss. *(Andere Formulierung finden)* Dies ist bei einer sigmoiden Funktion gegeben [Lie02] [Dör18].

Die Anordnung der Schichten eines KNNs kann beliebig variiert werden. Je nach Aufgabe des Netzes sind unterschiedliche Anordnungen der Schichten sinnvoll. Um zu wissen, welche Anordnung gut funktioniert, ist zum einen Erfahrung wichtig, zum anderen sollte aber auch experimentiert werden. Die Anordnungen können in verschieden Topologien kategorisiert werden. Die wichtigste wird im folgenden vorgestellt.

3.2.1 Feedforward

In einem *vorwärts verkettetem Netz* (*Feedforward Netz*) sind Neuronen einer Schicht ausschließlich mit Neuronen einer (nicht zwingend direkt) folgenden Schicht verbunden. So entsteht ein Informationsfluss in eine Richtung: Vorwärts [Moe17]. Meistens ist hierbei ein Neuron einer Schicht mit allen Neuronen der Folgeschicht verbunden. Abbildung 3.2 auf der nächsten Seite stellt ein solches Netz dar. Diese Topologie ist dann sinnvoll, wenn kein Zusammenhang zwischen den zu lernenden Beispielen besteht. Ein Zusammenhang zwischen den Mustern besteht, wenn zum Beispiel Bilder eines ungeschnittenen Videos analysiert werden. Hier ist jedes Bild von dem vorherigen Bild abhängig. Die Größe und Anzahl einzelner Schichten ist frei wählbar. Je komplexer ein zu lernendes

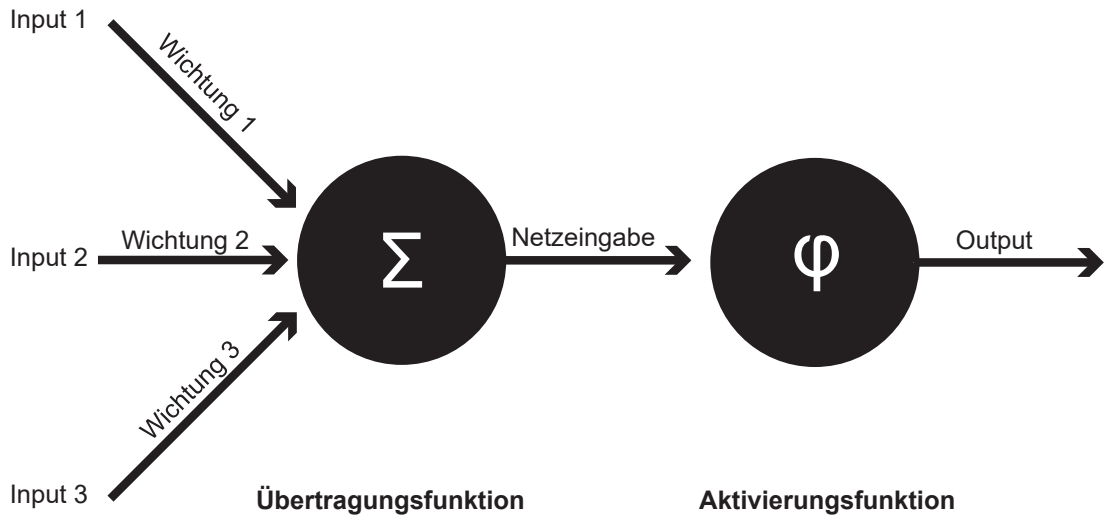

Abbildung 3.2: Schema eines Neurons

Muster ist, desto Größer sollten diese Parameter gewählt sein. Ein begrenzender Faktor ist hier die Leistung des Servers, der das Training durchführt. Für große Modelle wird viel Arbeitsspeicher verbraucht.

3.2.2 Training

Um ein in Abschnitt 3.2.1 erläutertes, mehrschichtiges Feedforward-Netz zu trainieren, wird der so genannte Backpropagation-Algorithmus (zu deutsch etwa Fehlerrückführung [Kin94]) angewandt. Beim Trainieren eines KNNs werden die Gewichte der Neuronenverbindungen in einer Weise verändert, sodass das Netz auf einer gegebenen Eingabe das gewünschte Ergebnis liefert [LC12]. Wie bereits in Abschnitt 3.2 erwähnt, muss dafür die Aktivierungsfunktion der künstlichen Neuronen differenzierbar sein.

Als Basis des Trainings benötigen wir neben den Trainingsdaten die zugehörigen, gewünschten Ausgaben des Netzes. Dies wird auch als überwachtes Lernen bezeichnet. Hierbei werden zunächst die Trainingsdaten auf die Eingabeschicht gegeben und die resultierende Ausgabe des Netzes berechnet. Wenn man diese tatsächliche Ausgabe mit der gewünschten Ausgabe (auch *Label* genannt) vergleicht, kann hieraus der Fehler des Netzes errechnet werden [Moe17]. Hierbei ist t_j der gewünschte und o_j der tatsächliche Ausgabewert des Knoten j.

$$E = \frac{1}{2} * \sum_{j} (t_j - o_j)^2 \tag{3.2}$$

Der berechnete Fehler kann durch eine Anpassung der Gewichte der Neuronen verringert werden. Dieses Training wird nun wiederholt, um unter der Berücksichtigung aller Trainingsdaten ein Minimum der Fehlerfunktion zu finden. Dies resultiert in einer besseren Erkennung des gestellten Problems durch das Netz [LC12].

3.2.3 Hyperparameter

Brownlee [Bro17b] beschreibt *Hyperparameter* im Kontext des maschinellen Lernens als Parameter, die extern, also vor dem Trainieren eines Modells, festgelegt werden und nicht aus Trainingsdaten abgeleitet werden können. Diese Parameter haben Einfluss auf die Performance und die Dauer des Trainings [Tra19]. Um geeignete Parameter zu finden, gibt es mehrere Ansätze und Algorithmen. Zunächst müssen die Mengen der Parameter bestimmt werden, die bei den Testdurchläufen variiert werden sollen. Oft wird dann jede Kombination, d. h., alle Elemente des kartesischen Produkts der genannten Mengen, durchprobiert. Dies wird auch Rastersuche (eng. *grid search* [Bro17b]) genannt und wurde auch in dieser Arbeit verwendet. Ein Hyperparameter ist meistens ein diskreter oder kontinuierlicher Zahlenwert

Welche konkreten Hyperparameter verfügbar sind, hängt von dem zu trainierenden Modell ab. In dieser Arbeit sind hauptsächlich die *learning rate*, *batch size* und die Anzahl der *training epochs* relevant.

learning rate Die *learning rate* (Lernrate) beeinflusst, wie stark die Gewichte der Neuronen eines KNNs beim lernen angepasst werden [LC12]. Eine gute Wahl der learning rate ist wichtig. Ist diese zu klein gewählt, könnte der Lernfortschritt sehr gering sein. Ist die learning rate dagegen zu groß gewählt, kann es passieren, dass ein Minimum der Fehlerfunktion nicht gefunden wird, da durch die hohe learning rate beim Training während des Annäherns an das Minimum so zu sagen über dieses hinaus geschossen wird [Zul18].

batch size Die *batch size* (zu deutsch etwa Batch-Größe) ist bei dem so genannten Batch-Verfahren relevant. Sie gibt an, wie viele Trainingsbeispiele ein zu trainierendes Netz durchläuft, bevor die Gewichte der Neuronen angepasst werden [LC12]. Je größer die batch size ist, desto mehr Hauptspeicher (RAM) wird beim Training verwendet, da alle Trainingsbeispiele eines Batches im Hauptspeicher gehalten werden müssen. Dies kann somit den maximalen Wert der batch size limitieren. Somit läuft man mit einer kleinen batch size nicht in die Gefahr, zu wenig Hauptspeicher zur Verfügung zu haben. Bei einer großen batch size liegt ein Vorteil in der schnelleren Ausführung des Trainings, weil die Gewichte der Neuronen viel seltener geändert werden müssen. Aus diesen selteneren Änderungen ergibt sich eine stabilere Annäherung an ein Minimum. Dadurch wiederum kann es passieren, dass nicht das kleinste Minimum gefunden wird, was zu einem schlechteren Trainingsergebnis führt [Bro17a].

training epoch Eine *training epoch* (deutsch: Trainingsepoche) ist im Bereich des maschinellen Lernens das einmalige Durchlaufen aller Trainingsdaten durch das zu trainierende Modell beim Training. Da beim Training eines KNNs nur sehr kleine Änderungen der Gewichte der Neuronen vorgenommen werden, ist es in der Regel sinnvoll, mehrere Epochen zu durchlaufen. Dadurch erhöht sich natürlich die benötigte Trainingszeit.

3.2.4 Vortrainierte Netze

Um eine komplexe Funktion mit einem Netz zu trainieren, benötigt man viele Trainingsdaten und Ressourcen. Hierfür und zur Lösung weiterer Probleme wurde das so genannte Pre-Training entwickelt. Dazu wird ein KNN mit einem großen Datensatz vortrainiert. Das bedeutet, ein Netz lernt bestimmte Zusammenhänge des Datensatzes, ohne bereits die eigentlich gewünschte Aufgabe zu trainieren. Hierfür kann ein nicht gelabelter Datensatz verwendet werden. Deswegen fällt das Pre-Training in den Bereich des *Unsupervised Learnings* (unüberwachtes Lernen). Dies hat offensichtlich den Vorteil, dass keine Ressourcen investiert werden müssen, um diese großen Datensätze mit Labeles zu versehen. Im Bereich des Textverstehens können dafür, je nach Anwendungsgebiet, große Textkorpora verwendet werden (z. B. Wikipedia-Artikel).

Wenn ein KNN vortrainiert wurde, kann man mit dem Fine-Tuning beginnen. Beim Fine-Tuning lernt das KNN die eigentliche Aufgabe, indem es zu jedem Trainingsbeispiel das gewünschte Resultat erhält. Deswegen fällt dieser Schritt in das *überwachte Lernen*. Es werden bei einem vortrainierten Netz viel weniger Trainingsiterationen und gelabelte Trainingsdaten benötigt. Das spart natürlich Zeit und Kosten [RNSS18] [Rad18].

Vortrainierte Netze sind beim Fine-Tuning im Vergleich zu Netzen mit neu initialisierten Kantengewichten nicht nur mit weniger Aufwand zu trainieren, sie erreichen oft auch bessere Ergebnisse. Dies kann laut Erhan et al. [EBC+10] an mehreren Faktoren liegen. Logisch erscheinen dürfte allerdings, dass das unsupervised Pre-Training eine gute Ausgangslage für das Fine-Tuning schafft, in dem es geeignete initiale Kantengewichte erzeugt. Diese sind dadurch in einem Bereich eingeschlossen, die für den Kontext der Trainingsdaten relevant sind.

3.3 Metrik der Evaluation

Die Evaluation eines Modells sind ein wichtiger Schritt in Projekten die maschinelles Lernen verwenden. Sie gibt Aufschluss darüber, wie geeignet ein Modell für eine gegebene Aufgabe ist. Im Folgenden werden die in dieser Arbeit verwendeten Metriken zur Evaluation eines trainierten Modells des maschinellen Lernens erläutert. Dabei werden diese Metriken hier im Bezug auf ein Klassifizierungsproblem betrachtet. Es gibt hier vier zu unterscheidende Fälle:

- ein Beispiel, das korrekt als Elemente einer Klasse erkannt wurde, nennt man *true positive*.
- ein Beispiel, das korrekt als als nicht der Klasse zugehörig erkannt wurde: *true negative* (TN).
- ein Beispiel, das Element einer Klasse ist, dies aber nicht erkannt wurde: *false negative* (FN).
- ein Beispiel, das nicht einer Klasse angehört, dies aber der Klasse zugeordnet wurde: *false positive* (FP).

Accuracy Die Accuracy, die man ins deutsche mit Genauigkeit übersetzen könnte, ist ein Maß dafür, welcher Anteil der getesteten Musterbeispiele richtig erkannt wurde [Met78]. Es gilt also folgende Gleichung:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.3}$$

Precision *Precision* gibt an, wie viele der als zu einer Klasse zugeordneten Elemente wirklich zu dieser Klasse zuzuordnen sind. Dies wird als Anteil angegeben, wie folgende Gleichung zeigt [OD08].

$$precession = \frac{TP}{TP + FP} \tag{3.4}$$

Recall *Recall* gibt den Anteil der Elemente einer Klasse an, die als solche erkannt werden [OD08].

$$recall = \frac{TP}{TP + FN} \tag{3.5}$$

F_1-Score Der harmonische Mittelwert von Precision und Recall wird durch den so genannten F_1-Score gebildet. Hiermit bekommt man mit nur einem Maß einen relativ guten Überblick über die Performance eines Klassifikators [OD08]. In dieser Arbeit wird zur Beurteilung der Performance eines Modell der Fokus auf dieses Maß gelegt.

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{3.6}$$

Confusion matrix Eine *Confusion matrix* ist eine quadratische Matrix, welche Informationen darüber gibt, wie ein Klassifikator (z. B. ein KNN) die verschiedenen Elemente klassifiziert. In dieser Arbeit stehen die einzelnen Zeilen der Matrix für die wahren (gelabelten) Klassen des Datensatzes und die Spalten für die von dem Modell vorhergesagten Klassen. Somit steht in einer Zelle a_{ij} (i-te Zeile und j-te Spalte) die Anzahl der Beispiele der Testdaten, die vom Klassifikator zur i-ten Klasse zugeordnet wurden und in Wahrheit der j-ten Klasse angehören. Daher ergibt es sich, dass bei einem perfekten Klassifikator, also ein Klassifikator, der alle Testdaten korrekt Klassifiziert, die Confusion matrix eine Diagonalmatrix darstellt.

3.4 Klassifizierung von E-Mail Acts

In der Arbeit von Tramountani [Tra19] wurde eine Klassifizierung von so genannten *speech acts* durchgeführt. Dies kann man als die Art einer Äußerung beschreiben. In diesem Fall wurden einzelne Sätze klassifiziert. Eine Frage, eine Anerkennung und eine Zustimmung sind Beispielklassen einer solchen Klassifizierung.

Aufbauend auf dieser Speech-Acts-Klassifizierung in [Tra19] mit BERT und anderen neuronalen Netzen, wurde von der selben Autorin die Klassifizierung von so genannten *E-Mail Acts* durchgeführt. Diese sind recht ähnlich zu den bisherigen speech acts. Allerdings wurden diese noch etwas genauer für das Auftreten in E-Mails spezifiziert. Eine Übersicht der Klassen bietet Tabelle 3.1 auf der nächsten Seite. Die dort genannten Abkürzungen werden auch in der Implementierung dieser Arbeit verwendet. Hierbei wird als *Floskel* (pl) hauptsächlich die Anrede am Anfang und der Gruß am Ende einer E-Mail klassifiziert. Eine *automatische E-Mail Aussage* wird von einem E-Mail Programm generiert, wenn man zum Beispiel auf eine E-Mail antwortet. Hier werden dann die wichtigsten Metadaten der ersten E-Mail hinzugefügt.

Abkürzung	**Englisch**	**Deutsch**
st	Statement	Aussage
qu	Question	Frage
re	Request	Anfrage/Bitte
de	Delivery	Übermittlung
pl	Pleasantry	Floskel
un	Incomplete sentence	Unvollständiger Satz
ae	Automatic E-Mail statement	Automatische E-Mail-Aussage

Tabelle 3.1: In diesem Projekt verwendete E-Mail Act Labels

4 Daten

4.1 Rohdaten

Die Rohdaten für die Arbeit bestehen aus einem Datenbestand von einzelnen E-Mails. Dieser Datenbestand beinhaltet etwas über 1.500 E-Mails und stammt aus Projekt- und Gemeinschaftspostfächern von einem IT- und Beratungsunternehmen. Die E-Mails beinhalten sowohl interne Konversationen, als auch solche mit Kunden (anonymisiert). Jede E-Mail wird durch eine .msg Datei repräsentiert. Dies ist ein binär kodiertes Format von Microsoft Outlook um E-Mailnachrichten und dessen Metadaten zu speichern.

4.2 Import

Um diese Dateien in Python verarbeiten zu können, wurde das Paket „pywin32“ verwendet. Mit diesem sind viele von Windows bereitgestellte application programming interfaces (APIs) in Python verwendbar. Durch die *Office Outlook Primary Interop Assembly* wird beim importieren der .msg Datei ein Objekt zurückgegeben, welches das Interface `Microsoft.Office.Interop.Outlook._MailItem` [Mic19] implementiert. Daraus können in den meisten Fällen alle relevanten Informationen einer E-Mail extrahiert werden. Manchmal konnten jedoch nicht alle Adressenten und/oder der Absender einer Mail extrahiert werden. In diesen Fällen wurde versucht, den E-Mail-Header auszulesen, um daraus die fehlenden Informationen zu beschaffen.

4.3 Vorverarbeitung

Die richtige Vorverarbeitung ist essenziell, um gute Ergebnisse beim Lernen eines KNNs zu erreichen. In dieser Arbeit wurden mehrere Schritte vorgenommen, um geeignete Daten in einer sinnvollen Form für das Trainieren des Netzes bereitzustellen.
Zunächst wird beim Import der .msg-Daten jeder E-Mail eine ID (inkrementell fortlaufend) zugewiesen. Danach werden die Anzeichen einer Satztrennung in dem Text einer E-Mail erkannt und die Sätze werden einzeln gespeichert. Für eine spätere Zuordnung bekommt jeder Satz ebenfalls eine eigene ID. Als erste Filterung werden von Tramountani [Tra19] mit Hilfe von der Pythonbibliothek *langdetect* [Dan17], welche eine Portierung von Googles *language-detection* Bibliothek [Nak10] darstellt, alle nicht deutschen Sätze herausgefiltert. Dies ist wichtig, weil sich die Arbeit auf deutsche Sätze beschränkt. Die nun getrennten, deutschen Sätze werden einzeln mit Hilfe eines KNNs nach dem *Mail Act* klassifiziert und mit einem *Mail Act Label* versehen (dazu mehr in Abschnitt 3.4 auf Seite 25). Mit Hilfe dieser Klassifizierung kann jetzt der gewünschte Mailtext vom Rest der Mail extrahiert werden. Dies ist nötig, da Mails oft eine ganze Serie von einzelnen E-Mails aufweisen.

Diese beinhaltet alle vorhergegangenen Konversationen, auf denen die betrachtete E-Mail eine Antwort ist. Diese sind für die Klassifizierung nicht erwünscht. Hier wird lediglich die eigentliche E-Mail ohne vorangegangene Nachrichten verwendet. Um diese zu extrahieren, können die Mail-Act Labels der einzelnen Sätze einer Mail herangezogen werden. Eine Antwort einer E-Mail beginnt immer mit automatisch generierten Informationen. Dies repräsentiert das Mail-Act Label 'ae'. Auf diese Weise wurde der eigentliche E-Mailtext vom Rest der Nachricht extrahiert. Das Verfahren setzt natürlich voraus, dass die Sätze einer Mail korrekt nach dem Mail-Act klassifiziert wurden, was für diese Arbeit auch zutrifft.

4.4 Labeling

Nachdem die Daten importiert und vorverarbeitet wurden, können diese gelabelt werden. Man weist hierbei jeder E-Mail aus dem Datensatz ein Label zu. So bestimmt man die Zuordnung zu den von mir festgelegten Funktion, die das Netz später bei der Klassifizierung in Betracht ziehen soll. Der Datensatz umfasste etwas mehr als 1.500 E-Mails und wurde nach 18 Funktionsklassen klassifiziert. Eine Auflistung dieser Klassen bietet Tabelle 4.1 auf der nächsten Seite. Für das Training des Klassifikators wurden allerdings nur Funktionsklassen verwendet, in denen über 20 Beispiele verfügbar sind. Die anderen Klassen werden somit nicht berücksichtigt. Dies ist sinnvoll, da mit zu wenig Trainingsbeispielen kein vernünftiges Training stattfinden kann.

4.4.1 Analyse

Nachdem allen E-Mails des initialen Datensatzes ein Label zugewiesen wurde, konnte die Verteilung der Labels analysiert werden. Wie bereits abzusehen war, ist diese sehr ungleichmäßig. Die meisten E-Mails stammen aus im Abschnitt 4.1 bereits erwähnten Projektpostfächern. Hier werden hauptsächlich Informationen und Fragen mit dem Kunden bzw. Auftraggeber eines Projekts ausgetauscht. Daher ist es verständlich, dass vielen E-Mails das Label *pi* (*Projektbezogene Information geben*) oder *pf* (*Projektbezogene Frage/Anfrage*) zugewiesen wurde. Idealerweise müssten diese Funktionsklassen weiter aufgeteilt werden, um eine bessere Gleichverteilung der Häufigkeit der Klassen herzustellen. Dies würde allerdings den Rahmen dieser Arbeit sprengen und wurde deshalb unterlassen. Eine vollständige Übersicht über die Verteilung der Labels über die Daten bietet Abbildung 4.1.

4.5 Splittung

Der Mail-Korpus wurde in einen Trainingsdatensatz und einen Testdatensatz eingeteilt. Hierbei wurden 90% der E-Mails für ersteren verwendet. Eine solche Einteilung ist wichtig, da für eine korrekte Evaluation ungesehene, das heißt für das Training nicht verwendete, Daten verwendet werden müssen. Die Ergebnisse der Trainings befinden sich in Kapitel 6 auf Seite 47.

Abkürzung	Bezeichnung	Erklärung
aw	Abwesenheitsnotiz	Information Über die Abwesenheit des Senders (z. B. Urlaub)
hp	Homepage freigeben	Die Freigabe, Inhalt auf einer Homepage zu veröffentlichen
bt	Bestellung tätigen	Aufgelistete Dinge bestellen (z. B. in einem Onlineshop)
ip	Inhalt prüfen	Den Inhalt einer Vorhergehenden E-Mail oder eines Dokuments prüfen und Rückmeldung geben
se	Server einrichten	Server mit gewünschten Spezifikationen einrichten und zugänglich machen (sowohl Hardwareserver als auch Softwareserver (Service))
sa	Server abschalten	Nicht mehr benötigten Server abschalten
nh	Netzwerk-Laufwerk Berechtigung hinzufügen	Berechtigung(en) einer/mehrerer Person(en) für ein Netzwerklaufwerk hinzufügen
na	Netzwerk-Laufwerk Berechtigung ändern	Berechtigung(en) einer/mehrerer Person(en) für ein Netzwerklaufwerk ändern
ne	Netzwerk-Laufwerk Berechtigung entfernen	Berechtigung(en) einer/mehrerer Person(en) für ein Netzwerklaufwerk entfernen
mh	E-Mailverteiler: Person hinzufügen	Person(en) zu einem E-Mailverteiler hinzufügen
me	E-Mailverteiler: Person entfernen	Person(en) von einem E-Mailverteiler entfernen
sy	Systemfehler	Beschreibung oder Nennung eines Fehlers in einem System, z. B. Bug in einem Softwaresystem
pf	Projektbezogene Frage /Anfrage	Frage, die sich auf ein Projekt bezieht
pi	Projektbezogene Information	Information, die sich auf ein Projekt bezieht
te	Terminvereinbarung	Vereinbarung eines Termins, z. B. Meeting oder Telefonkonferenz
pc	PC-Problem lösen	Problem einer Person / eines Mitarbeiters mit einem PC oder anderes technisches System lösen
ki	Keine Information	E-Mail enthält keine oder zu wenige Informationen um sie zu klassifizieren
ue	Überspringen	E-Mail wurde beim Labeln übersprungen

Tabelle 4.1: Verwendete Labels der Funktionsklassen und deren Bedeutung

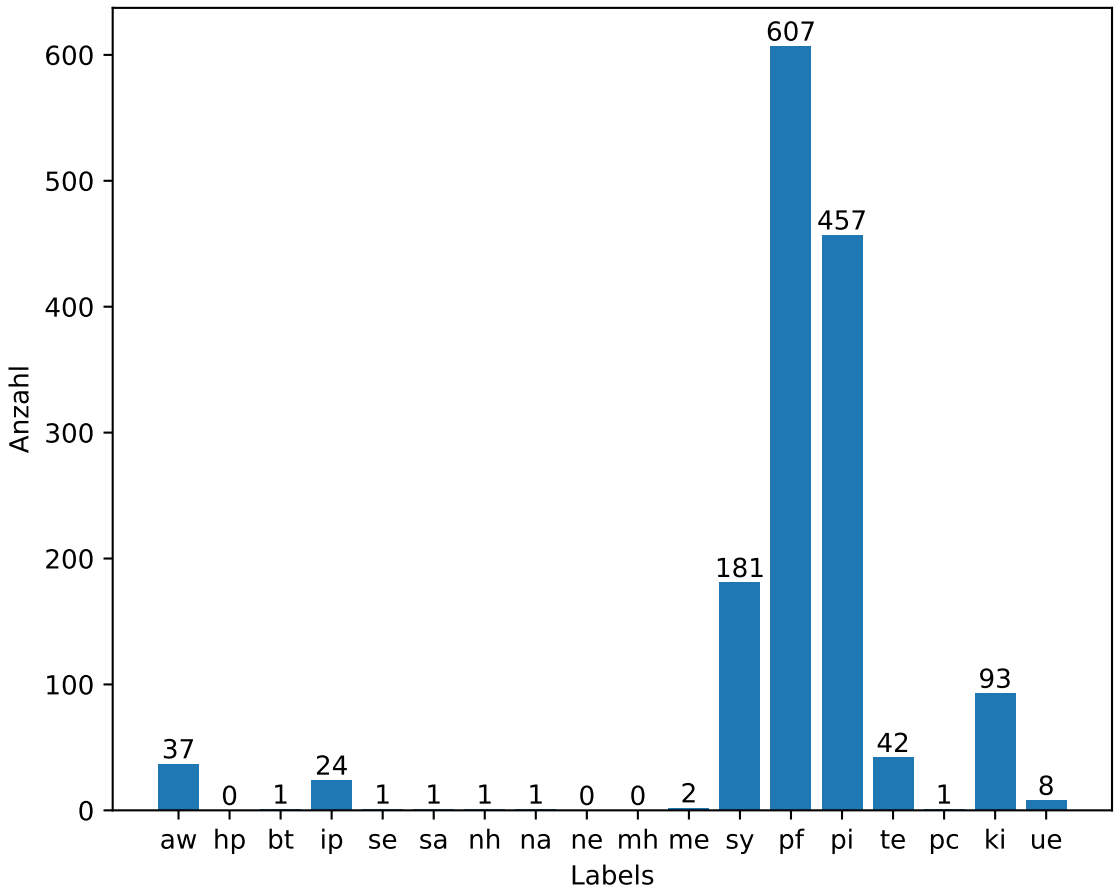

Abbildung 4.1: Häufigkeitsverteilung der Labels im gesamten Datenbestand

5 Implementierung

5.1 Ansatz

Wie bereits in Kapitel 1 beschrieben, kann die Bestimmung der Absicht des Senders einer E-Mail in zwei Teilschritten vollzogen werden: Zunächst werden die E-Mails nach ihrer Funktion klassifiziert. Hierfür wurden an den vorliegenden Datensatz angepasste Funktionsklassen definiert. Eine Verteilung der Daten in diesen Klassen zeigt Abbildung 4.1 auf der vorherigen Seite.

5.2 Architektur

In dieser Arbeit wird die Programmiersprache *Python* in der Version 3.6 verwendet. Sie ist eine Sprache mit einfacher Syntax, in der es sehr viele Bibliotheken im Bereich des maschinellen Lernens inklusive künstlicher Intelligenz gibt. Dies macht Python zu einer hervorragenden Wahl für Anwendungen mit neuronalen Netzen [Pat18]. Implementiert wurde diese Arbeit mit der Hilfe von der Community Version der integrated development environment (IDE) *PyCharm* von Jetbrains [Jet19].

5.2.1 Softwarearchitektur

Die Implementierung der Arbeit ist in drei Pakete (eng. packages) unterteilt: `mail`, `classification` und `parameter_extraction`. Diese repräsentieren die drei wesentlichen Arbeitsschritte, um eine eingehende Mail zu verarbeiten. Hierbei befinden sich im Paket `mail` die Rohdaten, als auch die vorverarbeiteten Varianten der Mails und die Logik, um diese zu generieren. Eine Übersicht der Paketstruktur bietet Abbildung 5.1 auf der nächsten Seite.

mail Das Paket `mail` gliedert sich in drei Unterpakete: Zum einen ist hier ein Paket namens `mail_parser`, welches die meiste Logik der Mail Vorverarbeitung enthält. Hier werden zunächst im Modul `importer` die Rohdaten, welche aus einzelnen .msg Datein bestehen, eingelesen. Die Texte und Metadaten einer Mail werden dann im Modul `reader` eingelesen und an den `importer` zurückgegeben, welcher dann die Informationen aller Mails in der JSON abspeichert. Des Weiteren gibt es im Paket `mail` das Modul `separator`. Dieses ist für die weitere Vorverarbeitung der Mailtexte verantwortlich. Hauptsächlich werden hier Mailtexte voneinander separiert, bestimmte Mails aussortiert und getrennt gespeichert. Außerdem existiert ein Ordner mit der Bezeichnung `charts`, in dem sich generierte Grafiken befinden, die Informationen über die Verteilung der Häufigkeit der vergebenen Labels geben. Als letztes befindet sich ein Python-Modul `mail_helper`

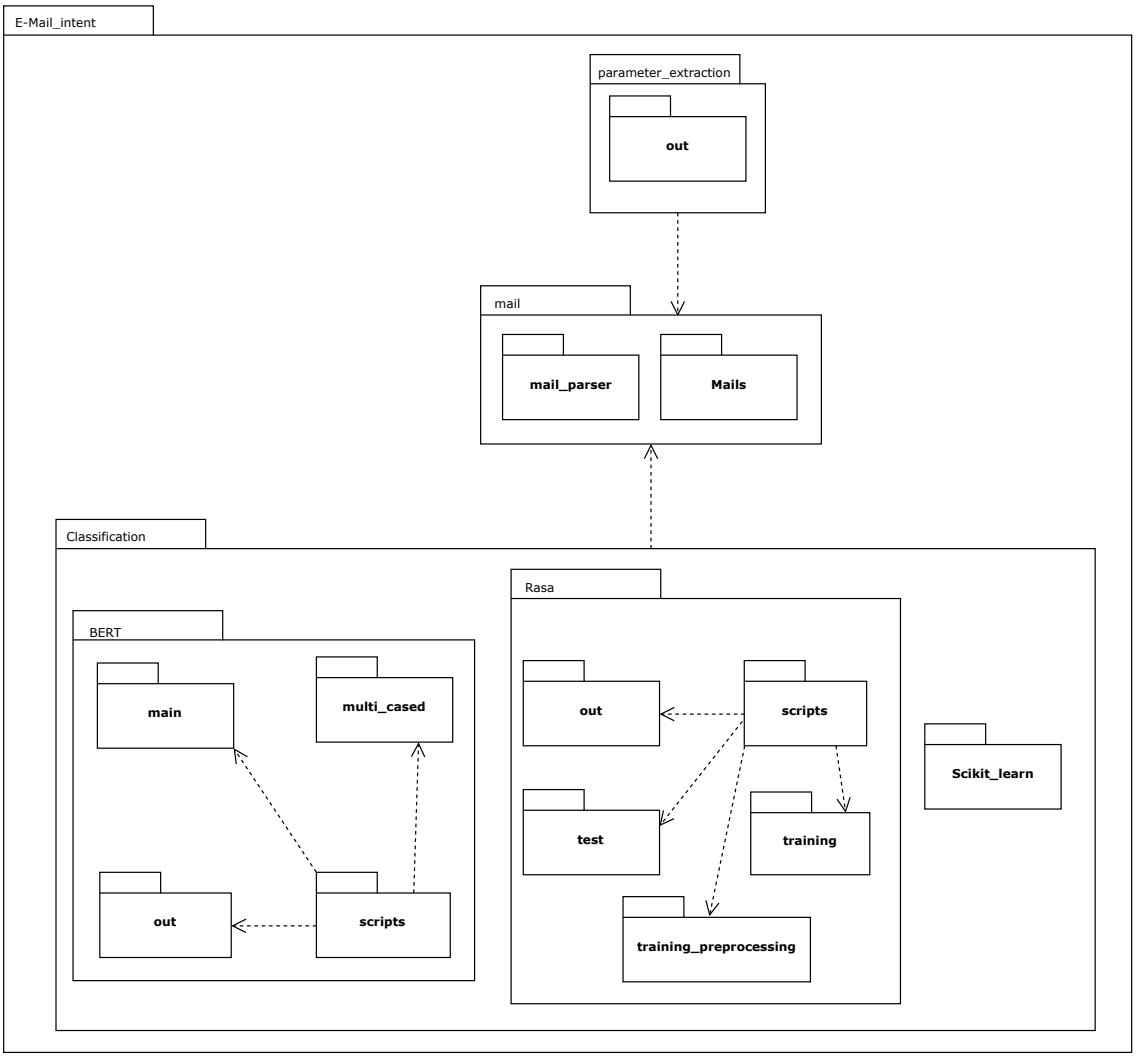

Abbildung 5.1: UML-Diagramm: Übersicht über die Paketstruktur der Arbeit

im Hauptverzeichnis, welches verscheide Hilfsfunktionen bereit stellt. Diese Funktionen sind für die verschiedenen Aufgaben, darunter das Lesen und Schreiben von JSON-Dateien und das Extrahieren von bestimmten E-Mail-Sätzen und -Adressen zuständig.

5.2.2 Verwendete Bibliotheken

In der Umsetzung dieser Arbeit wurden einige fremde Python Pakete verwendet. Diese werden in der Tabelle 5.1 auf der nächsten Seite mit ihrem Namen, Lizenz und Version gelistet.

matplotlib Matplotlib ist eine Bibliothek, um Diagramme in einer hohen Qualität zu erstellen. Sie besitzt eine einfache Schnittstelle, um eine Vielzahl von mathematischen Plots zu generieren. Dabei bietet sie auch ein Framework an, das ähnlich wie Matlab zu verwenden ist [Hun07]. In dieser Arbeit wird die Bibliothek für die Erstellung zweier Grafiktypen verwendet. Dies ist zum

Name	Version	Lizenz
matplotlib	2.2.4	BSD
numpy	1.16.2	3-Clause BSD
pywin32	224	PSF
rasa-nlu	0.14.6	Apache 2.0
scikit-learn	0.20.3	3-Clause BSD
seaborn	0.9.0	3-Clause BSD
spacy	2.1.3	MIT

Tabelle 5.1: Verwendete externe Bibliotheken der Arbeit

einen eine Häufigkeitsverteilung der Labels in dem vorliegenden Maildatensatz. Diese wird als klassisches Balkendiagramm dargestellt. Abbildung 4.1 auf Seite 30 zeigt die Verteilung des gesamten Datensatzes. Zum anderen ist das eine Heatmap, die eine Wahrheitsmatrix (eng. confusion matrix) visualisiert. Diese Heatmap ist wie die zugehörige Wahrheitsmatrix angeordnet.

numpy Numpy ist eine leistungsstarke Python-Bibliothek, die effiziente, arithmetische Operationen auf Arrays (auch mehrdimensional) ermöglicht. Dafür verwendet sie eine eigene Datenstruktur *ndarray* [Num19]. Für die Klassifizierung der E-Mails mit Hilfe der Scikit-learn Bibliothek wird numpy in der Vorverarbeitung verwendet. Hier müssen die in numerischen Werten konvertierten E-Mail Daten in einen normalisierten Bereich gebracht werden, um von einigen Modellen des maschinellen Lernens verwendet werden zu können.

pywin32 Pywin23 ermöglicht es, auf ein .net API zuzugreifen. Hiermit kann man auf Funktionen von dem installierten Outlook zugreifen [pywin32]. In der Arbeit wird Pywin23 dazu verwendet, die Rohdaten zu importieren und die Metadaten zu extrahieren.

rasa-nlu Rasa-nlu ist eine Bibliothek die zum Rasa Framework gehört. Mit diesem kann man intelligente Assistenten und Chatbots entwickeln. Dabei ist dieses Framwork in zwei Bibliotheken aufgeteilt. 1. Die eine ist die bereits erwähnte rasa-nlu. Diese ist dafür verantwortlich, die Eingaben des Benutzers zu interpretieren und zu verstehen. Sie kann hauptsächlich dazu verwendet werden, die Absicht (*intent classification*) und die Eigennamen (*entity extraction*) einer Nachricht erkennen. 2. Die andere ist rasa-core, welche für das Steuerung der Konversation und Bereitstellen von Antworten zuständig ist [Ras19]. In der Arbeit wird ausschließlich erstere für die Klassifizierung der E-Mails verwendet und mit den anderen Lösungsansätzen verglichen.

scikit-learn Scikit-learn (oder kürzer sklearn) ist eine Bibliothek, die eine große Anzahl von Aufgaben in vielen verschiedenen Bereich des maschinellen Lernens lösen kann. Dies geht über die eigentliche Klassifizierung, Regression und Clustering hinaus. Sie bietet z. B. auch Möglichkeiten der Datenvorverarbeitung und Datenanalyse an [PVG+11]. In der Arbeit werden geeignete Modelle der Klassifizierung für Texte dieser Bibliothek verwendet und untereinander, sowie mit den anderen Modellen verglichen.

seaborn Seaborn ist eine Bibliothek, die auf Matplotlib aufbaut. Sie ist ebenfalls geeignet um mathematische Grafiken zu erstellen. Hierbei wurden die Dataframes von Pandas, ebenfalls eine verbreitete Python-Bibliothek, für Verwendung als Datenquelle der Plots integriert. Außerdem bietet sie ein graphisches Interface, um die generierten Grafiken nach den Wünschen anzupassen [Was18]. Um eine Heatmap von der Wahrheitsmatrix generieren zu können, wird Seaborn in der Arbeit verwendet.

spacy Spacy ist eine Bibliothek für die Textverarbeitung. Sie dient als als Werkzeugkasten, um mit Texten umzugehen, sie zu analysieren und umzuwandeln. Spacy ist für die Zusammenarbeit mit neuronalen Netzen sehr geeignet. So kann sie z. B. Texte tokenisieren (Segmentierung von Sätzen auf Wortebene) und grammatikalische und andere linguistische Zusammenhänge erkennen. Dafür stellt Spacy Sprachmodelle in acht verschiedenen Sprachen, sowie ein multilinguales Modell zur Verfügung. In dieser Arbeit wird Spacy zum einen für die Konvertierung von E-Mailtexten in feauture vectors verwendet, zum anderen für die linguistische Analysefähigkeiten (insbesondere das so genannte *Part-of-speech tagging* und *Syntactic Dependency Parsing*[1]) [HM19].

Des Weiteren wurde *BERT* [DCLT18] verwendet. Dies wurde allerdings noch nicht als Python-Paket hochgeladen. Für die Verwendung lädt man hier stattdessen das vortrainierte Sprachmodell und die Quelltextdateien herunter. Mehr Informationen zu BERT sind in Abschnitt 5.4 auf Seite 36.

5.3 Rasa-NLU

Rasa ist ein Framework für die Erstellung eines Chatbots. Hierbei ist Rasa-NLU für das Verstehen, die Klassifizierung und die Eigennamenserkennung zuständig. Diese Arbeit nutzt die ersten beiden genannten Fähigkeiten der Bibliothek. Die Verwendung von Rasa-NLU ist gut durchdacht und wurde mit guten Beispielen auf einer eigenen Webseite dokumentiert. Dies macht die Nutzung und Implementierung sehr angenehm [Ras19].

5.3.1 Pipeline

Rasa verwendet für den Ablauf der einzelnen Arbeitsschritte eine konfigurierbare Pipeline. Hierüber kann definiert werden, welche Module verwendet werden und in welcher Reihenfolge dies geschieht. Dafür gibt es eine Konfigurationsdatei im YAML Ain't Markup Language (YAML) (.yml) Format. Des Weiteren bietet Rasa-NLU bereits vorgefertigte Beispiele einer Pipeline an. Für die Nutzung dieser Beispiele muss lediglich dessen Name in der eben genannten Konfigurationsdatei angegeben werden. Diese vorgefertigten Pipelines sind gerade für das Kennenlernen der und das Experimentieren mit der Bibliothek von Vorteil. Die Konfiguration, die in dieser Arbeit verwendet wird, ist in Listing 5.1 auf der nächsten Seite dargestellt. Dies ist eine Abwandelung der vorgefertigten *pretrained_embeddings_spacy* Pipeline[2].

[1] Eine Dokumentation hierfür ist unter `https://spacy.io/api/annotation` zu finden.

[2] Eine Dokumentation dieser ist auf `https://rasa.com/docs/rasa/nlu/choosing-a-pipeline/#pretrained-embeddings-spacy` zu finden.

Listing 5.1 Konfiguration der Rasa-NLU Bibliothek

```
language: "en"

language: "de"

pipeline:
- name: "SpacyNLP"
- name: "SpacyTokenizer"
- name: "SpacyFeaturizer"
- name: "RegexFeaturizer"
- name: "SklearnIntentClassifier"
```

Rasa verwendet einige Module der Spacy Bibliothek. Damit spacy korrekt in Rasa eingebunden wird, muss es zunächst initialisiert werden. Hiernach wird mit dem *SpacyTokenizer* der Text in einzelne Wortelemente aufgeteilt und wichtige Features mit dem *SpacyFeaturizer* extrahiert. Der *RegexFeaturizer* erstellt beim Training regular expressions. Dann wird untersucht, welche regular expressions in welchen Trainingsbeispielen vorkommen. Dies soll das Klassifizieren vereinfachen. Am Ende der Pipeline steht der *SklearnIntentClassifier*, welcher mittels der extrahierten Features die Klassifizierung der E-Mails durchführt. Wie der Name des Pipelinemoduls schon vermuten lässt, wird dabei die Scikit-learn (sklearn) Bibliothek verwendet. Genauer gesagt, wird eine SVM trainiert, die durch das *Grid Search*[3]-Verfahren optimiert wird [Ras19].

5.3.2 Vorverarbeitung der Trainingsdaten

Damit Rasa die E-Mail Daten einlesen kann, müssen diese in eine von Rasa akzeptierte Form gebracht werden. Hier werden zwei verschiedene Formate angeboten. Dies ist zum einen *Markdown*, eine Auszeichnungssprache, die leicht von Menschen gelesen und geschrieben werden kann. Zum anderen wird mit der JSON eine Alternative angeboten. Diese kann sehr leicht von Computern geparst werden. Da die JSON bereits für die interne Speicherung der E-Mail Daten in der Arbeit verwendet wird, ist der Aufwand etwas geringer, dieses Format auch für den Export für Rasa zu verwenden. Ausschlaggebend ist allerdings die einfache Konvertierung von geladenen Daten in das JSON Format innerhalb eines Python Skriptes, was eine Fehlersuche auch vereinfacht. Ein Ausschnitt dieser Datei könnte so aussehen wie es Listing 5.2 auf der nächsten Seite zeigt.

Die Implementierung dieses Scripts befindet sich in dem Modul `classification/Rasa/training_preprocessing/training_data_preprocessing.py`. Für die Konvertierung der Daten wird das Script einmal für die Trainingsdaten und einmal für die Testdaten aufgerufen. Es besitzt zwei Parameter, einen für den Eingabe- und einen für den Ausgabepfad der Daten. Die Logik ist recht einfach gehalten. Es wird die Eingabedatei geöffnet und die Mailtexte inklusive der zugehörigen Labels eingelesen. Anschließend werden diese Elemente im benötigter Struktur in eine neue JSON gespeichert. Diese befindet sich in folgenden Unterordnern des Softwareprojekts: `classification/Rasa/training/BERT_import_train.json`, sowie `classification/Rasa/test/BERT_import_test.json`.

[3] Eine Erklärung dieses Begriffs ist in Abschnitt 3.2.3 auf Seite 23 zu finden.

Listing 5.2 Beispielausschnitt aus den Trainingsdaten für Rasa

```
{
   "rasa_nlu_data": {
      "common_examples": [
         {
            "intent": "pi",
            "text": "Hallo Herr Mustermann, Wir haben entsprechend ihren Anmerkungen die Rechtematrix
            ↪ modifiziert, das Update finden Sie im Anhang. [...]"
         },
         {
            "intent": "aw",
            "text": "Vielen Dank für Ihre Nachricht. Ich bin bis einschließlich 1. Januar 2019 nicht im
            ↪ Hause. Ich werde Ihre Nachricht nach meiner Rückkehr beantworten. In dringenden Fällen
            ↪ können Sie sich auch an max.mustermann@example.com wenden."
         }
      ]
   }
}
```

5.4 BERT

Bidirectional Encoder Representations from Transformers (kurz BERT) ist eine von Google entwickelte Architektur für ein KNN. Dieses wurde mit einer großen Anzahl von nicht gelabelten Webtexten (Wikipediaartikel) vortrainiert. Daraus entsteht ein Sprachmodell, für das man sehr wenige gelabelte Trainingsdaten für das Fine-Tuning benötigt. Man kann also mit einem kleinen Datensatz bereits sehr gute Ergebnisse erzielen, die auch im Vergleich zu anderen Lösungen auf einem hohen Niveau sind. Die Besonderheit bei BERT ist die Art und Weise, wie der Kontext bei der Repräsentation einzelner Wörter ermittelt und verwendet wird.

Damit ein KNN Wörter verarbeiten kann, müssen diese in Zahlenwerte konvertiert werden. Dies sollte auf eine semantisch geeignete Weise vollzogen werden. Hierfür kann der Kontext der Wörter, also die anderen Wörter eines Satzes oder Absatzes, berücksichtigt werden (contextual). Allerdings wurde hierfür bisher, von dem zu analysierenden Wort aus gesehen, nur Wörter einer Richtung berücksichtigt (unidirectional) bzw. beide Richtungen einzeln analysiert (shallowly bidirectional) und die Erbgebisse der beiden Teilanalysen zusammengefasst. Hier geht BERT einen neuen Weg. Es werden erstmals die vorherigen und nächsten Wörter des zu analysierenden Wortes für den Kontext des Wortes berücksichtigt. Dies wird von den Entwicklern als *deep bidirectional* bezeichnet [DCLT18].

5.4.1 Pre-Training

Für das Pre-Training des Sprachmodells von BERT wurden Wikipediatexte verwendet. Bei diesen wurden 15% der Wörter maskiert (durch [MASK] ersetzt). Diese mussten dann von dem Netz vorhergesagt werden. Mit dieser Methode kann dem Netz ein Sprachverständnis antrainiert werden. Um auch Zusammenhänge zwischen Sätzen zu trainieren wurde ein weiteres Trainingsverfahren verwendet. Hierbei werden dem Netz zwei Sätze übergeben. Das zu trainierende Netz soll nun vorhersagen, ob der zweite Satz auf den ersten folgt oder nicht.

Zum Zeitpunkt dieser Arbeit wurden bereits mehrere Sprachmodelle durch die Entwickler veröffentlicht. Diese sind englische Sprachmodelle, zwei mehrsprachige Modelle, sowie Modell für die chinesische Sprache. In dieser Arbeit wird eines der zwei mehrsprachigen Modelle verwendet. Diese zwei Modell haben einen Unterschied: Es gibt ein „uncased“ Modell. Hier wurde der gesamte Text vor der weiteren Verwendung in Kleinschrift konvertiert. Darüber hinaus wurden alle Akzentzeichen entfernt. Bei der „cased“ Version des Modells wurde diese Vorverarbeitung nicht durchgeführt [DCLT18].

5.4.2 Performance Überblick

BERT erreicht bei mehreren Aufgaben im Vergleich mit anderen neuronalen Netzen sehr gute Ergebnisse. So erreicht BERT bei SQuAD 1.1, ein Datensatz, bei dem durch Textverstehen Fragen beantwortet werden müssen, den zweiten Platz (Stand Mai 2019). Hieraus ergibt sich ein besserer F_1-Score (93.16) als die menschliche Leistung (91.22). In dem neuen Datensatz SQuAD 2.0, bei dem auch unbeantwortbare Fragen vorkommen, liegt BERT derzeit sogar auf dem ersten Platz. Diese Leistung ist keine Ausnahme. BERT erreicht in vielen Aufgaben des neuronalen Sprachverstehens (NLP) die besten Ergebnisse. Daher ist BERT, auch wegen der Tatsache, dass wenige Trainingsdaten benötigt werden, für diese Arbeit gut geeignet.

5.4.3 Vorverarbeitung der Trainingsdaten

Damit BERT Trainingsdaten verwenden kann, müssen diese natürlich eingelesen und geladen werden. Anders als bei anderen Bibliotheken, die schon länger veröffentlicht sind und daher schon einige Funktionen um die Hauptfunktionalität haben, gibt es bei BERT kein festgelegtes Format, in dem die Daten eingelesen werden können. Um das Einlesen der Daten zu ermöglichen, muss für jeden Datensatz eine eigene, kleine Klasse implementiert werden. Diese wird im Code mit dem Namen *Processor* versehen, um diese als solchen zu kennzeichnen. Von BERT wird eine Basisklasse bereitgestellt, von der die zu implementierende Klasse erben soll. Somit wird eine Kompatibilität sichergestellt. Um diesen Vorgang zu vereinfachen und um BERT testen zu können, wurden bereits einige Fine-Tuning Beispiele implementiert und (direkt in der Quelldatei) veröffentlicht. In diesen Beispielen liegen die verwendeten Trainingsdaten allerdings im tab-separated values (TSV) Format vor. Da das bei den Trainingsdaten dieser Arbeit nicht der Fall ist (diese sind im JSON Format gespeichert), musste eine eigene Lösung gefunden werden.

Es wurde somit die Klasse `MailProcessor(DataProcessor)` erstellt. Vererbt durch die Basisklasse, besitzt die Klasse drei `get`-Methoden, jeweils eine für das Trainings-, Development- und Test Menge, sowie eine `get`-Methode für die Menge der Labels. Des Weiteren ist eine Klassenmethode für das Parsen von TSV-Dateien vorhanden, diese ist für die Arbeit allerdings nicht relevant. Die drei `Get`-Methoden für die Maildaten-Mengen erwarten eine Python-Liste mit `InputExample`-Objekten als Elemente dieser Liste. Jedes Element entspricht hier einem Trainingsbeispiel. Diese können mit folgendem Konstruktor erstellt werden: `InputExample(guid=guid, text_a=text_a, text_b=text_b, label=label)`. Hier ist der Parameter `guid` eine id des Trainingsbeispiels, die einzigartig sein muss. Dies ist in diesem Fall ein zusammengesetzter `String`. Der Parameter `text_a` ist hier der eigentliche Mailtext. `text_b` ist für Aufgaben bei

denen zwei Texte vergleichen oder im Bezug zueinander Verarbeitet werden sollen und hat hier keine Relevanz. Daher kann dieser hier auf `None` gesetzt werden. Das Label ist die zu trainierende Ausgabe des Netzes.

5.4.4 Fine-Tuning

Das Trainieren eines vortrainierten Modells wird Fine-Tuning genannt. Hierbei wird das vortrainierte Netz auf die eigentlich zu lösende Aufgabe trainiert.

Parameter

Das starten des Fine-Tunings (für eine Erklärung siehe Abschnitt 3.2.4 auf Seite 24) eines BERT-Netzes geschieht mittels eines Konsolenbefehls. Dabei werden alle nötigen Einstellungen des Netzes (inkl. Hyperparameter) als Optionen mit dessen Parametern übergeben. Der eigentliche Befehl ist `python run_classifier.py`. Die Optionen dieses Befehls, die beim Training und bei der Evaluation verwendet wurden, sind nachfolgend aufgelistet. Hierbei sind die bei den Trainingsdurchläufen veränderlichen Parameter kursiv dargestellt.

– – task_name = mail Dieser Parameter teilt BERT mit, was für ein Textkorpus zu erwarten ist. Mit `mail` kann hier also die Zuordnung zu dem `MailProcessor` vollzogen werden. Dies geschieht mittels einem zentralen Python-Dictionarys.

– – do_train = True Hiermit weißt man BERT an, ein Training durchzuführen.

– – do_eval = True Hiermit weißt man BERT an, das trainierte Modell mit dem Development-Set zu evaluieren. Dabei ermittelt BERT die Werte *accuracy* und *loss*.

– – do_predict = True Hiermit weißt man BERT an, das trainierte Modell auf das Test-Set anzuwenden. Es wird also für jedes Beispiel aus dieser Menge eine Ausgabe des Netzes generiert und abgespeichert. Wie diese Ausgabe interpretiert und evaluiert wird, kann nachfolgend unter *Automatisiertes Trainieren* nachgelesen werden.

– – do_lower_case = False Diese Flag teilt BERT mit, dass die Wörter nicht in Kleinbuchstaben konvertiert werden sollen, sondern die Groß- und Kleinschreibung mit berücksichtigt werden soll. Dies ist bei der Verwendung eines *Cased* Sprachmodells, also ein Modell bei dem das Pre-Training ebenfalls unter der Berücksichtigung von Groß -und Kleinschreibung vollzogen wurde, sinnvoll.

– – data_dir = mail\Mails Hierüber wird der Pfad zu dem Ordner festgelegt, in dem sich die Trainings- und Testdaten befinden.

– – vocab_file = BERT\BERT-Base_Multilingual_Cased\vocab.txt Dies ist eine Textdatei, die das Vokabular des Modells widergespiegelt. Es wird von den Entwicklern von BERT in dem Sprachmodell mitgeliefert und wird zur Indexierung dieses Vokabulars verwendet [DCLT18]. Diese Datei befindet sich, wie das gesamte Sprachmodell, nicht im Projektverzeichnis.

– – bert_config_file = BERT\BERT-Base_Multilingual_Cased\bert_config.json Diese Konfigurationsdatei legt die beim Fine-Tuning unveränderlichen Hyperparameter fest. Sie wird ebenfalls mit dem Sprachmodell mitgeliefert und befindet sich nicht im Projektverzeichnis.

– – init_checkpoint = BERT\BERT-Base_Multilingual_Cased\bert_model.ckpt Hierbei handelt es sich um eine Checkpoint-Datei von TensorFlow, welche die von den Entwicklern vortrainierten Gewichte des KNNs enthält.

– – max_seq_length = 128 Mit diesem Parameter wird die maximale Länge des Eingabestrings festgelegt. Diese wurde bei den Tests in diesem Projekt nicht variiert und hatte stets den Wert 128.

– – train_batch_size = *train_batch_size* Dieser Parameter legt die Batch Size des Fine-Tunings fest. Er wurde zwischen 1 und 32 variiert. Die diskreten Schritte dabei waren 1, 2, 4 ,8 16 und 32.

– – learning_rate = *learning_rate* Die Lernrate des Fine-Tunings wird durch diesen Parameter festgelegt.

– – num_train_epochs = *num_train_epochs* Mit diesem Parameter kann die Anzahl der durchgeführten Trainingsepochen des Fine-Tunings festgelegt werden

– – output_dir = *out_root* + "model_"+ *model_nr* + "–" + *model_iteration* Dieser Parameter legt den Speicherort des Modells fest. Er wird aus einem vorher definierten Hauptverzeichnis (*out_root*), sowie aus der Modell- und Trainingsiterationsnummer generiert.

Automatisiertes Trainieren

Für das Fine-Tuning von BERT wurde ein Script zur dessen Automatisierung geschrieben. Hiermit können vorher definierte Parameterkombinationen getestet werden, ohne erneut ein Training manuell starten zu müssen. Um dieses Script zu verwenden, müssen zunächst die gewünschten Parameter für das Training festgelegt werden. Dabei sind diese bereits initialisiert und es müssen nur die nötigen Änderungen zur Initialisierung definiert werden. Hiernach kann bereits das Training mit einem Funktionsaufruf ausgeführt werden. Dabei wird 1. der Befehl inklusive der im letzten Abschnitt erklärten Parameter generiert, um BERT auszuführen, 2. dieser generierte Befehl ausgeführt, 3. die Zeit für die Dauer der Ausführung ermittelt, 4. die Evaluation der Performance des aktuellen Modells durchgeführt (siehe nächster Abschnitt). Ein beispielhafter Aufruf des Scripts wird in Listing 5.3 auf der nächsten Seite dargestellt.

Das Fine-Tuning eines Modells kann für eine gegebene Parameterkombination mehrmals durchgeführt werden. Dies ist sinnvoll, da beim Trainieren der Zufall einen Einfluss auf die Performance des Modells haben kann. Der Parameter *training_executions* legt diese Anzahl der Durchläufe fest. In dieser Arbeit werden stets fünf Durchläufe durchgeführt.

5.4.5 Evaluation der trainierten Modelle

Nachdem ein Modell trainiert wurde, kann es evaluiert werden. Es wird hier getestet, wie gut es auf ungesehene Eingaben, dies sind Eingaben, die nicht bereits beim Training des Modells genutzt wurde, anwendbar ist. Durch die Evaluation des BERT-KNNs, welches durch die Option – – do_predict = True (vergleiche Absatz *Parameter* aus vorherigem Abschnitt) gestartet wird, wurde die Datei *test_results.tsv* erstellt. Diese enthält durch Tabs separierte Gleitkommazahlen. Jede Zeile dieser Datei steht für das Ergebnis eines Testbeispiels, jeder Wert in dieser Zeile steht für

Listing 5.3 Beispielausschnitt für den Aufruf des automatisierten Testscripts für BERT

```
num_train_epochs = 1

model_nr = "01"
train_batch_size = 1
run_training_config()

model_nr = "02"
train_batch_size = 2
run_training_config()

model_nr = "03"
train_batch_size = 4
run_training_config()

model_nr = "04"
train_batch_size = 8
run_training_config()
```

die vorhergesagte Wahrscheinlichkeit, die gesuchte Klasse zu sein. Mit diesem Wissen kann aus einer solchen Zeile das Maximum gefunden werden [DCLT18]. Mit dem Index des Eintrages kann die Zuordnung zu der vom Netz vorhergesagten Klasse hergestellt werden. Danach wird diese Klasse mit der gelabelten Klasse des Beispiels verglichen. Somit kann für jedes Testbeispiel ermittelt werden, ob sie durch das Netz korrekt vorhergesagt wurde. Hieraus wird eine Liste mit allen vorhergesagten Klassen, eine Liste mit allen gelabelten Klassen und eine Liste mit den Namen aller verwendeten Labels erstellt. Mit Hilfe dieser Listen kann das Modell evaluiert werden. Das kann für die Ergebnisse aller in dieser Arbeit getesteten Modelle zur Klassifikation verallgemeinert werden. Dieser Zusammenhang wird für alle Klassifizierungsansätze im Abschnitt 5.6 auf Seite 43 erläutert. Die Ergebnisse dieser Evaluation wird für jedes Modell gespeichert. Am Ende werden die Parameter des Modells zusammen mit den Ergebnissen der Evaluation in einer JSON gespeichert. Dieses hat folgende Parameter:

avg_duration Gibt die durchschnittliche Dauer des Trainings des Modells an. Hierbei wird der Durchschnitt aus den fünf einzelnen Durchläufen des Trainings eines Modells errechnet.

best_f1 Gibt den besten F_1-Score der fünf durchgeführten Durchläufe an.

best_version Der Index der Version mit dem besten F_1-Score wird durch diesen Parameter angegeben.

model_nr Jeder Parameterkombination wird eine eigenen Modellnummer zugewiesen. Diese wird bei eins angefangen iteriert. Für eine geeignete Darstellung in einer Liste, wird dieser Parameter zweistellig als String gespeichert. Somit bekommt das erste Modell die Modellnummer `01`.

model_parameter Enthält eine Liste der in dem zugehörigem Modell verwendeten Hyperparameter. Diese sind *learning_rate*, *max_seq_length*, *train_batch_size*, *train_epochs* und *training_executions*.

out_root Gibt den Pfad an, in dem die trainierten Modelle gespeichert wurden.

Die Bedeutung der weiteren Parameter wurde bereits in Abschnitt 5.4.4 auf Seite 38 erläutert.

models Enthält eine Liste der Messgrößen der Evaluation eines Trainingsdurchlaufs. Diese Messgrößen sind accuracy, F_1, precision und recall. Sie wurden in Abschnitt 3.3 auf Seite 24 erklärt.

model_iteration Gibt den Index des aktuellen Durchlaufs des Trainings mit der selben Parameterkombination an.

Ein Beispielausschnitt zeigt Listing 5.4 auf der nächsten Seite.

5.5 Scikit-learn

Scikit-learn ist eine Python-Bibliothek, die eine Reihe von Lösungen im Feld des maschinellen Lernens anbietet [PVG+11]. In dieser Arbeit werden mehrere Modelle getestet und miteinander verglichen.

5.5.1 Vorverarbeitung der Trainingsdaten

Für die Verwendung der Trainingsdaten müssen diese zunächst in eine von Scikit-learn lesbare Form konvertiert werden. Dafür werden in einem ersten Schritt die Mailtextdaten in Vektoren konvertiert. Dies geschieht mit Hilfe der Bibliothek *Spacy*. Diese hat für mehrere Sprachen vortrainierte Sprachmodelle. In dieser Arbeit wird hierfür das deutsche Modell *de_core_news_md* verwendet. Spacy generiert also aus den in dem Sprachmodell gespeicherten Wortvektoren einen Vektor für einen E-Mail Text. Diese werden in einer Liste gespeichert. Des Weiteren müssen die Labels der Texte ebenfalls konvertiert werden. Dies geschieht mit einem Modul von Scikit-learn namens *LabelEncoder*. Dieser kodiert die Labelklassen in Integerwerten mit 0 beginnend. Hiermit wird jedes Label einer Mail codiert und in einer Liste gespeichert. Somit ergeben sich für die Trainings- und für die Testdaten jeweils zwei Listen (je eine für codierte Texte und codierte Labels). Diese Listen können dann bereits an ein zu trainierendes Modell übergeben werden. Die hier beschriebene Vorverarbeitung ist für fast alle Modelle ausreichend. Lediglich für das *Complement Naive Bayes* musste für erfolgreiches Training zusätzlich die Werte der Vektoren der E-Mailtexte in den nicht negativen Bereich verschoben werden.

5.5.2 Training

Für das Training eines Scikit-learn Modells, muss dieses zunächst erstellt werden, was mit einem Konstruktoraufruf zu bewerkstelligen ist. Hiernach kann bereits trainiert werden, indem die `fit()` Funktion des Modells aufgerufen wird. In dieser Arbeit wurden drei verschiedene Modelltypen der Scikit-learn Bibliothek trainiert. Diese sind die folgenden (in Klammern ist jeweils der Name des Moduls): *Linear Support Vector Classification* (LinearSVC), *Complement Naive Bayes* (ComplementNB) und *Decision Tree Classifier* (DecisionTreeClassifier). Für das Testen verschiedener Hyperparameter werden die Modelle nacheinander ausgeführt.

Listing 5.4 Beispielausschnitt der Evaluationsergebnisse von BERT. Diese werden von dem automatisierten Testscripts für BERT erstellt.


```
{
   "bert_model_evaluation_data": [
      {
         "avg_duration": 5,
         "best_f1": 0.6520746073498367,
         "best_version": 0,
         "model_nr": "01",
         "model_parameter": {
            "learning_rate": "2e-5",
            "max_seq_length": 128,
            "out_root": "nlu\\BERT\\out\\mail_output",
            "train_batch_size": 1,
            "train_epochs": 1,
            "training_executions": 5
         },
         "models": [
            {
               "accuracy": 0.7152777777777778,
               "conf_mat": "[[ 0  0  0  4  0  0  0]\n [ 0  0  0  3  0  0  0]\n [ 0  0  0  7  4  0  1]\n [
               ↪  0  0  0 57  6  0  0]\n [ 0  0  0  8 42  0  0]\n [ 0  0  0  1  2  0  0]\n [ 0  0  0  0
               ↪  5  0  4]]",
               "f1": 0.6520746073498367,
               "model_iteration": 0,
               "precision": 0.6088938912429379,
               "recall": 0.7152777777777778
            },
            {
               "accuracy": 0.4375,
               "conf_mat": "[[ 0  0  0  4  0  0  0]\n [ 0  0  0  3  0  0  0]\n [ 0  0  0 12  0  0  0]\n [
               ↪  0  0  0 63  0  0  0]\n [ 0  0  0 50  0  0  0]\n [ 0  0  0  3  0  0  0]\n [ 0  0  0  9
               ↪  0  0  0]]",
               "f1": 0.266304347826087,
               "model_iteration": 1,
               "precision": 0.19140625,
               "recall": 0.4375
            },
            {
               "accuracy": 0.4375,
               "conf_mat": "[[ 0  0  0  4  0  0  0]\n [ 0  0  0  3  0  0  0]\n [ 0  0  0 12  0  0  0]\n [
               ↪  0  0  0 63  0  0  0]\n [ 0  0  0 50  0  0  0]\n [ 0  0  0  3  0  0  0]\n [ 0  0  0  9
               ↪  0  0  0]]",
               "f1": 0.266304347826087,
               "model_iteration": 2,
               "precision": 0.19140625,
               "recall": 0.4375
            },
            [...]
         ]
      },
      [...]
   ]
}
```

5.5.3 Evaluation

Bei der Evaluation wird das im Training erstellte Modell getestet. Dazu wird zunächst die Ausgabe des trainierten Modells auf den Testdaten generiert. Das Modell soll also die Labels der E-Mail vorhersagen (mittels *predict()*). Die Vorhersagen sind hier allerdings noch in Zahlenform codiert. Diese können mithilfe der Funktion `inverse_transform(predictions)` des zugehörenden LabelEncoders dekodiert werden. Die nun zu den Abkürzungen der Bezeichnung konvertierten Labels können in letzten Schritt mit den wahren Labels der Testdaten verglichen werden. Dazu wird, ähnlich wie bei der Evaluation der BERT Modelle, das *metrics* Modul von Scikit-learn verwendet. Dieses berechnet aus diesem Vergleich der Labels die Metriken zur Evaluation der Performance der betrachtenden Modelle. Des Weiteren wird für die bessere Visualisierung mit *seaborn* [Was18] eine Heatmap der Wahrheitsmatrix erstellt. Die Ergebnisse der Evaluation werden der Konsolenausgabe entnommen.

5.6 Evaluation der Klassifizierung

Nachdem für ein Modell eines Klassifizierungsnetzes jeweils eine Liste für die vorhergesagten und für die gelabelten Klassen erstellt wurde, kann durch den Vergleich dieser beiden Listen die Performanz des Modells bestimmt werden. Dafür wird das *metrics* Paket der Python-Bibliothek *scikit-learn* [PVG+11] verwendet. Dieses berechnet mit der Angabe der beiden obengenannten Listen, sowie einer Liste mit der Bezeichnung aller verwendeten Labels, F1-Score, precision, accuracy, accuracy, sowie die confusion matrix. Hier wurde die gewichtete Summe[4] zur Durchschnittsberechnung der Klassen verwendet. Eine Erklärung zu diesen Messgrößen wurde in Abschnitt 3.3 auf Seite 24 gegeben. Die Ergebnisse der Evaluation befinden sich in Abschnitt 6.1 auf Seite 47.

5.7 Parameterextraktion

Diese Sektion erklärt nach der Klassifizierung der E-Mails den zweiten, wesentlichen Schritt des Ablaufs des in dieser Arbeit vorgestelltem Programms: Die Extraktion von Parametern. Ein Parameter ist in diesem Zusammenhang eine Information, die die Absicht des Senders in Erweiterung zu der Funktionsklasse genauer beschreibt. Dies könnte z. B. bei der Funktion Terminvereinbarung das Datum oder der Typ des Termins sein.

Das Paket ist in vier Untermodule aufgeteilt. Diese sind 1. *extractor*, 2. *parameter_helper*, 3. *aw* und 4. *te*. Ersteres ist das Hauptmodul der Parameterextraktion. Von hier aus wird alles Weitere aufgerufen. Hierin befindet sich die Funktion `extract_all()`, welche die extract-Funktionen der einzelnen Funktionsklassen aufruft und die Ausgabe dieser strukturiert. Dies schafft eine Modularität und ist leicht erweiterbar, sollten neue Funktionsklassen hinzukommen. Zusätzlich existiert der Ordner *out*, der die Ausgabe der Parameterextraktion enthält. Dies ist eine JSON-Datei, deren Aufbau weiter unten erläutert wird.

[4] Mehr Details dazu in der Dokumentation der jeweiligen Funktion von scikit-learn, hier z.B für den F_1-Score: `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html`

Die Ausgabe einer Parameterextraktion ist immer gleich aufgebaut: Die Daten einer Mail werden mittels eines Python-Dictionarys übergeben. Dies ist zum einen sehr flexibel und zum anderen lässt sich dieses leicht in einer JSON-Datei speichern. In diesem Dictionary befindet sich die `mail_id`. Hiermit kann die Ausgabe einer bestimmten Mail zugeordnet werden. Des Weiteren ist das `function_label` angegeben. Hiermit wird klar, welcher Funktionsklasse der Mail zugeordnet wird und welche extrahierten Parameter zu erwarten sind. Diese werden in einer Liste von Dictionary-Einträgen gespeichert. Im Falle einer Abwesenheitsnotiz zum Beispiel wird diese Liste `aw_parameter` bezeichnet. Die extrahierten Parameter einer Funktionsklasse werden nachfolgend beschrieben.

5.7.1 Abwesenheitsnotiz

`aw` ist das Modul, welches die Parameter der Mails einer Abwesenheitsnotiz extrahiert. Die Hauptfunktion dieses Moduls, `detect_parameters`, wird für jede zu bearbeitende Mail einmal aufgerufen. Hierbei werden die mail ID und einer Liste von Sätzen einer Mail an diese übergeben. Die einzelnen Parameter können unabhängig voneinander extrahiert werden.

Dies ist zum einen *date*, der in der Funktion `detect_par_date(sentences)` extrahiert wird. In dieser wird der Mailtext regelbasiert nach Datumsangaben durchsucht. Da diese Funktion auch in der Funktionsklasse *Terminvereinbarung* benötigt wird, wurde die Funktion in die Hilfsklasse der Parameterextraktion ausgelagert. Es werden alle Namen von Wochentagen und Monaten, sowie deren gängigen Abkürzungen erkannt. In folgender Liste wird beispielhaft gezeigt, welche Datumsangaben erkannt werden:

- Dienstag 01.01.2019
- Dienstag, 01.01.2019
- Dienstag, 1. Januar 2019
- Dienstag 1. Januar 2019
- Dienstag, 1. Januar
- Dienstag 1. Januar
- 1. Januar 2019
- 1. Januar

Ein weiterer Parameter der Abwesenheitsnotiz ist die Adresse des Senders einer E-Mail. Diese kann aus den Metadaten der E-Mail bestimmt werden. Generell werden Metadaten nicht in die Liste der extrahierten Parameter mit aufgenommen. Allerdings erscheint mir die Adresse der Abwesenden Person als sehr wichtig. Deswegen habe ich diese hier hinzugefügt.

Viele E-Mails dieser Funktionsklasse enthalten eine Information zu Personen, die in dringenden Fällen erreicht werden sollten. Dies wird in der Regel als Bitte formuliert (z. B. „in dringenden Fällen kontaktieren sie bitte X unter Max.Mustermann@example.de“). Daher kann dieser Satz mittels der E-Mail Act Klassifizierung von anderen Sätzen unterschieden werden. Eine Liste aller Sätze, die als Bitte klassifiziert wurden, sind in Parameter `urgent_text` gespeichert.

Aus diesen gefundenen Sätzen wird mittels einer regular expression die Nennung von E-Mail Adressen extrahiert. Diese werden in einer weiteren Liste im Parameter `urgent_address` gespeichert.

5.7.2 Terminvereinbarung

Das Modul `te` ist für die Extraktion von Parametern von E-Mails, die eine Terminvereinbarung darstellen, zuständig. Die Definition geeigneter Parameter ist hier etwas schwieriger.

Um mögliche, von dem Sender einer Terminvereinbarung gewünschte Aktionen zu erlangen, werden Sätze, die als Frage oder Bitte klassifiziert wurden, extrahiert und als Liste in dem Parameter *qu_re_sentences* gespeichert. Hierbei wird der eigentliche Satz mit dem zugehörigen E-Mail Act Label (hier `qu` oder `re`) gespeichert.

Darüber hinaus wird nach einer Terminaktion gesucht. Dies kann eine Terminvereinbarung sein, aber auch eine Verschiebung oder Absage. Realisiert wird diese Suche über das Verb eines Satzes. Dabei wird zunächst das Lemma eines Wortes gebildet. Dieses wird verglichen mit einer festgelegten Liste an Verben, die für eine Aktion stehen. Die verwendeten Listen sind: stattfinden, treffen, austauschen, reden, bereden und besprechen für Vereinbarungen; absagen, ausfallen und entfallen für Absagen; und schieben und verschieben für Terminverschiebungen. Dabei ist eine Erkennung auch möglich, wenn ein Verbpräfix abgespalten ist (z. B. bei „Ich sage den Termin ab.“), oder ein Hilfsverb im Satz vorhanden ist (z. B. bei „Der Termin kann abgesagt werden.“). Dieses Verb wird im Parameter `action_name` ausgegeben. Der Integer `action_type` beschreibt zusätzlich, in welcher Liste das Verb gefunden wurde, d. h., ob es sich um eine Vereinbarung, Verschiebung oder Absage handelt. Hierbei steht eine Eins für eine Vereinbarung, eine Zwei für Absage und eine Drei für eine Verschiebung. Nachdem ein solches Verb erkannt wurde, wird von diesem ausgehend, die Terminbezeichnung gesucht. Dies wird, wie auch schon die Lemmatisierung, mit Hilfe dem *Part-of-speech tagging* (POS) und *Syntactic Dependency Parsing* (DEP) von Spacy[5] umgesetzt. Es werden also Wortphrasen gesucht, die in Relation zu dem gefundenen Verb stehen. Hiermit entstehen somit Kombinationen, die im Idealfall bereits die Terminaktionen der Terminvereinbarung offenbaren. Beispiele sind hierfür: „müssen schieben -> den Termin“ und „können absagen -> die Abstimmung“. Dabei werden alle Kandidaten für eine Terminbezeichnung in der Liste `relevant_names` und das selektierte Element, welches am wahrscheinlichsten das Relevante ist, in der Variable `appointment_name` ausgegeben. Darüber hinaus wir noch geprüft, ob mit dem Verb eine Negation zu verknüpfen ist. Ob dies der Fall ist, wird mit der Variable `negation` ausgegeben. Zu guter Letzt wird der ganze Satz, indem diese Elemente gefunden wurden, in dem Parameter `sentence` ausgegeben. Alle in diesem Absatz genannten Parameter (`action_name`, `action_type`, `appointment_name`, `negation`, `relevant_names` und `sentence`) gehören zu einem Treffer innerhalb eines Satzes und werden im Parameter `te_action` in Form einer Liste zusammengefasst.

Somit können auch mehrere Treffer dargestellt werden. Ein ganzes Beispiel einer Parameterextraktion einer E-Mail kann in Listing 5.5 auf der nächsten Seite nachvollzogen werden.

Hier wird zunächst ein Satz als Bitte erkannt. Deshalb taucht in der Liste `qu_re_sentences` dieser Eintrag auf. Bei der Detektion der Terminaktionen gibt es zwei Einträge. Hieraus kann bereits abgeleitet werden, dass die Abstimmung abgesagt werden kann und Themen per Telko besprochen werden können.

[5] Die Dokumentation dazu ist unter `https://spacy.io/api/annotation` zu finden.

Listing 5.5 Beispielausschnitt für die Ausgabe der Parameterextraktion


```
{
   "mail_parameter_data": [
      [
         {
            "function_label": "te",
            "mail_id": 515,
            "te_parameter": {
               "qu_re_sentences": [
                  {
                     "mail_act_label": "re",
                     "sentence": "Lassen sie mich wissen ob es Bedarf zur Abstimmung gibt."
                  }
               ],
               "te_action": [
                  {
                     "action_name": "k\u00f6nnen absagen",
                     "action_type": 2,
                     "appointment_name": [
                        "die Abstimmung"
                     ],
                     "negation": false,
                     "relevant_names": [
                        "die Abstimmung"
                     ],
                    "sentence": "Aus meiner Sicht können wir die Abstimmung heute absagen, sollte es aus
                     ↪  ihrer Sicht Herr Mustermann ergänzend zu gestern noch Themen zur Diskussion
                     ↪  geben, können wir diese gerne per Telko besprechen."
                  },
                  {
                     "action_name": "k\u00f6nnen besprechen",
                     "action_type": 1,
                     "appointment_name": [
                        "per Telko"
                     ],
                     "negation": false,
                     "relevant_names": [
                        "per Telko"
                     ],
                    "sentence": "Aus meiner Sicht können wir die Abstimmung heute absagen, sollte es aus
                     ↪  ihrer Sicht Herr Mustermann ergänzend zu gestern noch Themen zur Diskussion
                     ↪  geben, können wir diese gerne per Telko besprechen."
                  }
               ]
            }
         }
      ]
   ]
}
```

6 Evaluation

6.1 Klassifizierung

6.1.1 Rasa

Die Rasa-NLU-Bibliothek ist sehr entwicklerfreundlich gestaltet und ist auf den Einsatz in einer Chatbot-Anwendung spezialisiert. Sie kann allerdings auch, wie in dieser Arbeit geschehen, getrennt von den Chatbot-Logiken verwendet werden. Für die Klassifizierung ist in der Rasa-Pipeline das Scikit-learn-Modul definiert, welches eine SVM mit linearem Kern verwendet. Diese wird intern bereits mit dem Grid-Search-Algorithmus optimiert [Ras19]. Dies alles führt dazu, dass der Anwender wenige Parameter verändern muss, um die Performance des Modells zu optimieren. Es genügt, die Trainings- und Testdaten (inkl. der Labels) in einem der zwei vorgegebenen Formaten bereitzustellen.

Es wurden insgesamt fünf Modelle mit den gleichen Parametern trainiert. Der F_1-Score ist bei allen Modellen ziemlich Konstant und betrug zwischen 66% und 68% (bestes Ergebnis mit Rasa). Eine Heatmap der Klassifikation des besten Modells ist in Abbildung 6.1 auf der nächsten Seite zu sehen. Eine Normalisierung war hier leider nicht möglich. Zu sehen sind hier also die absoluten Werte.

Die Dauer des Trainings war bei allen Trainingsdurchläufen ähnlich und lag bei ca 3 Minuten.

6.1.2 BERT

BERT wurde mittels eines automatisierenden Scripts trainiert und evaluiert. Das Skript übernimmt also das manuelle Eintragen der Hyperparameter, das Starten des Trainings, das Evaluieren des Modells und das Dokumentieren der Evaluation. Es können also die Hyperparameter für jeden Trainingsdurchlauf festgelegt werden. Hier wurden Mittels des grid search-Verfahren viele Kombinationen von Hyperparametern getestet. Die verwendeten Hyperparameter, die beim Training der BERT Modelle verwendet wurden, können der Tabelle 6.1 auf der nächsten Seite entnommen werden. Verständlicher Weise wurden nicht alle möglichen Kombinationen überprüft, sondern nur die relevanten davon getestet. So wurden in dieser Arbeit 76 verschiedene Kombinationen von Hyperparametern mit Hilfe von BERT trainiert. Um den Einfluss des Zufalls beim Training zu minimieren, wurde das Training jeder Kombination von Hyperparametern fünf mal durchgeführt und das beste Ergebnis hervorgehoben (vergleiche `best_f1` in Abschnitt 5.4.5 auf Seite 39).

Abbildung 6.2 auf Seite 49 zeigt die Trainingsergebnisse dieser besten Modelle. Die Lernrate beträgt bei allen Ergebnissen dieser Grafik $2 * 10^{-5}$. Je dunkler hier eine Zelle gefärbt ist, desto besser ist das Trainingsergebnis. Aus der Grafik wird deutlich, dass die Anzahl der Trainingsepochen wesentlich zur Performance (dem F_1-Score) beiträgt. Die verwendete batch size hingegen hat eher

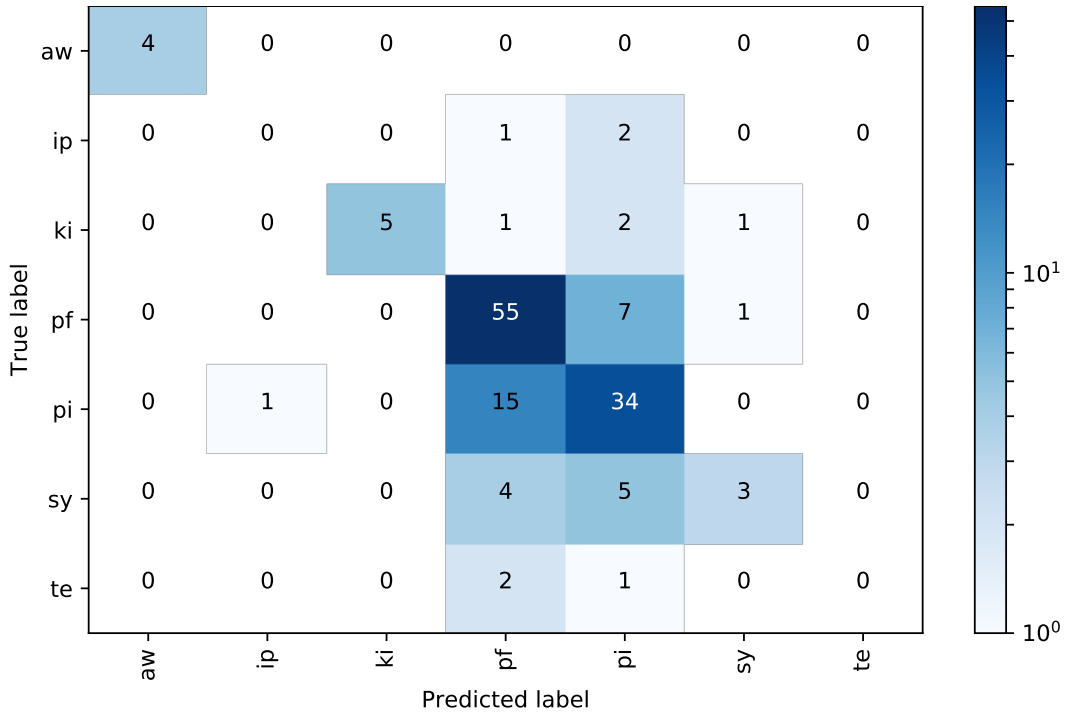

Abbildung 6.1: Nicht normalisierte Heatmap der Klassifikation des RASA-Modells mit dem besten F_1-Score

Bezeichnung	**Werte**
batch size	1, 2, 4, 5, 8, 16, 24
training epochs	1, 2, 3, 4, 5, 8, 12, 16, 24, 32
learning rate	$2 * 10^{-6}$, $5 * 10^{-6}$, $1 * 10^{-5}$, $2 * 10^{-5}$, $4 * 10^{-5}$, $8 * 10^{-5}$, $1 * 10^{-4}$, $2 * 10^{-4}$

Tabelle 6.1: Hyperparameter beim BERT Fine-Tuning

einen geringen Einfluss auf den Lernerfolg des Modells. Es ist zwar eine Auswirkung sichtbar, diese ist allerdings über die verschiedene Anzahl von Trainingsepochen unregelmäßig und wird mit zunehmender Anzahl der Trainingsepochen geringer.

Wie Tabelle 6.1 zeigt, wurden auch viele verschiedene Lernraten verwendet.

Bei einer Analyse des Einflusses der Lernrate auf die Performance des Modells, wurden diese alle mit einer batch size von 2 und 24 Trainingsepochen trainiert. Die erreichte Performance dieser Modelle wird in Abbildung 6.3 auf der nächsten Seite dargestellt. Hierbei wird deutlich, dass das beste Ergebnis bei einem knappen Vorsprung mit einer Lernrate von $2 * 10^{-5}$ erzielt wird. Darüber hinaus fällt auf, dass der Lernerfolg stark von der Wahl der Lernrate abhängen kann. Bei einer zu großen Lernrate fällt die Performance des Modells stark ab. Dies wurde beim Training beachtet und deshalb die Lernrate für weitere Trainingsdurchläufe auf $2 * 10^{-5}$ gesetzt.

Insgesamt gesprochen, wurden gute Ergebnisse beim Fine-Tuning erreicht. So erzielt das Modell mit der besten Performance einen F_1-Score von 91,5%. Eine normalisierte Heatmap der Klassifikation des besten Modells stellt Abbildung 6.4 auf Seite 50 dar. Hier wird sichtbar, dass das Modell durchweg

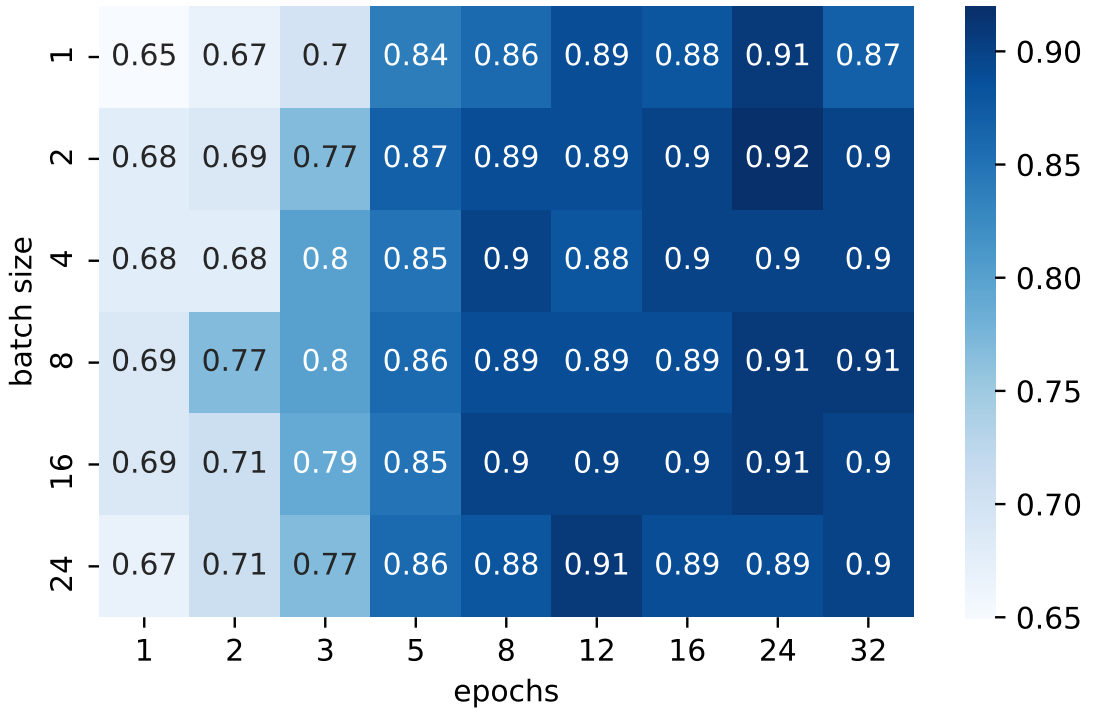

Abbildung 6.2: Trainingsergebnisse von BERT. Die gezeigten Werte spiegeln den F_1-Score der Modellauswertung wieder. Die Lernrate beträgt hier in allen Durchläufen $2 * 10^{-5}$.

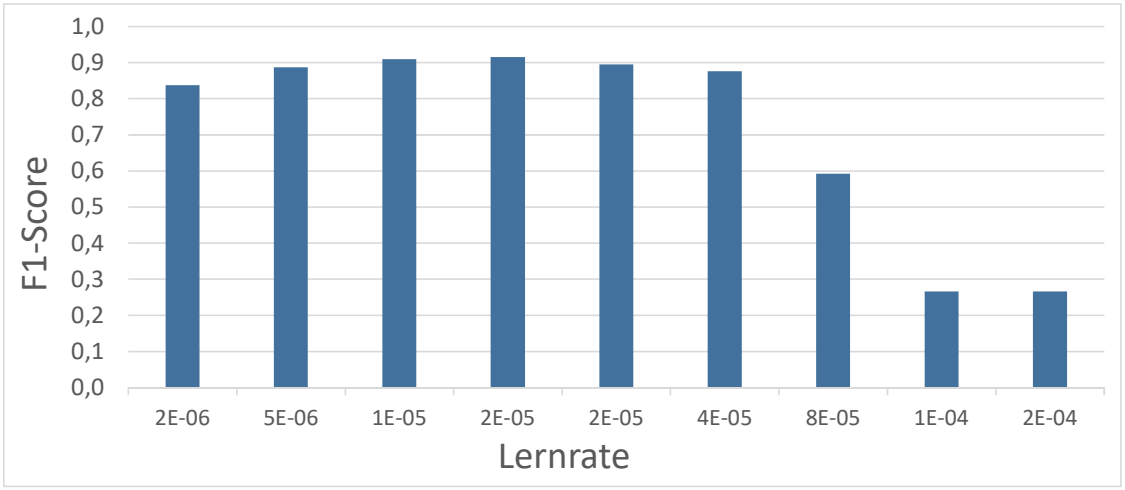

Abbildung 6.3: Trainingsergebnisse von BERT nach Lernrate sortiert. Die gezeigten Werte spiegeln den F_1-Score der Modellauswertung wieder. Die Batch-Size beträgt hier immer 2 und die Anzahl der Trainingsepochen 24.

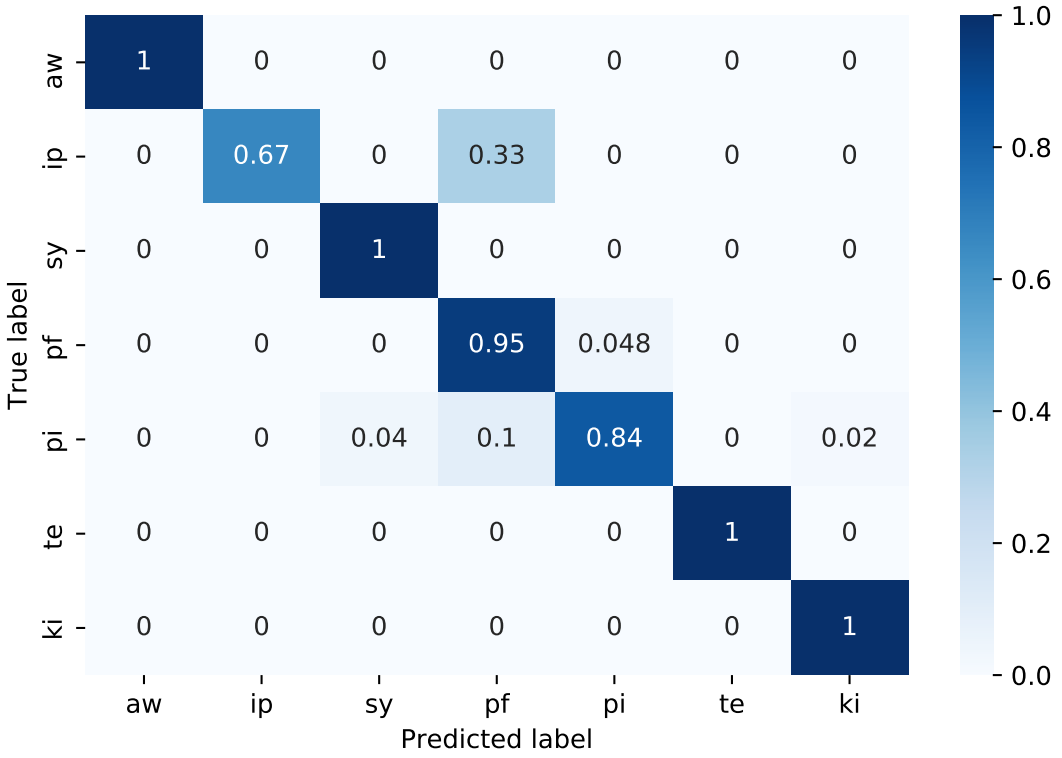

Abbildung 6.4: Normalisierte Heatmap der Klassifikation des BERT-Modells mit dem besten F_1-Score

gute Ergebnisse erzielt. Lediglich wird ein drittel der Klasse *Inhalt Prüfen* als *projektbezogene Frage* erkannt. Die Repräsentativität dieser Verteilung ist allerdings anzuzweifeln, da die Anzahl der Testdaten in dieser Klasse sehr gering ist. Es wurde hier eine von insgesamt drei Beispielen falsch erkannt.

Ein weiterer Aspekt ist die Trainingsdauer. Diese variiert stark durch die Wahl der Hyperparameter. So ist eine große batch size in Verbindung mit einer geringen Anzahl von Trainingsepochen ideal für eine kurze Trainingsdauer. In Bezug auf eine späteren Anwendung muss ein KNN gar nicht oder nur sehr selten neu trainiert werden. Deshalb sollte die Trainingsdauer in den meisten Fällen kein großes Gewicht bei der Wahl der Hyperparameter haben. Es ist allerdings interessant zu sehen, wie der theoretische Einfluss der Hyperparameter auch tatsächlich die Trainingsdauer beeinflusst, sodass dadurch in der Abbildung 6.5 auf der nächsten Seite ein kontinuierlicher Farbverlauf von unten links (große batch size und wenig Epochen) nach oben rechts (kleine batch size und viele Epochen) zu erkennen ist.

6.1.3 Scikit-learn

Von der Scikit-learn Bibliothek wurden mehrere Modelle des maschinellen Lernens auf die Eignung der E-Mail Klassifikation geprüft. Die Modelle wurden mit den gleichen Trainings- und Testdatensatz trainiert, wie auch schon die Modelle der BERT- und RASA-Bibliotheken. Es wurden

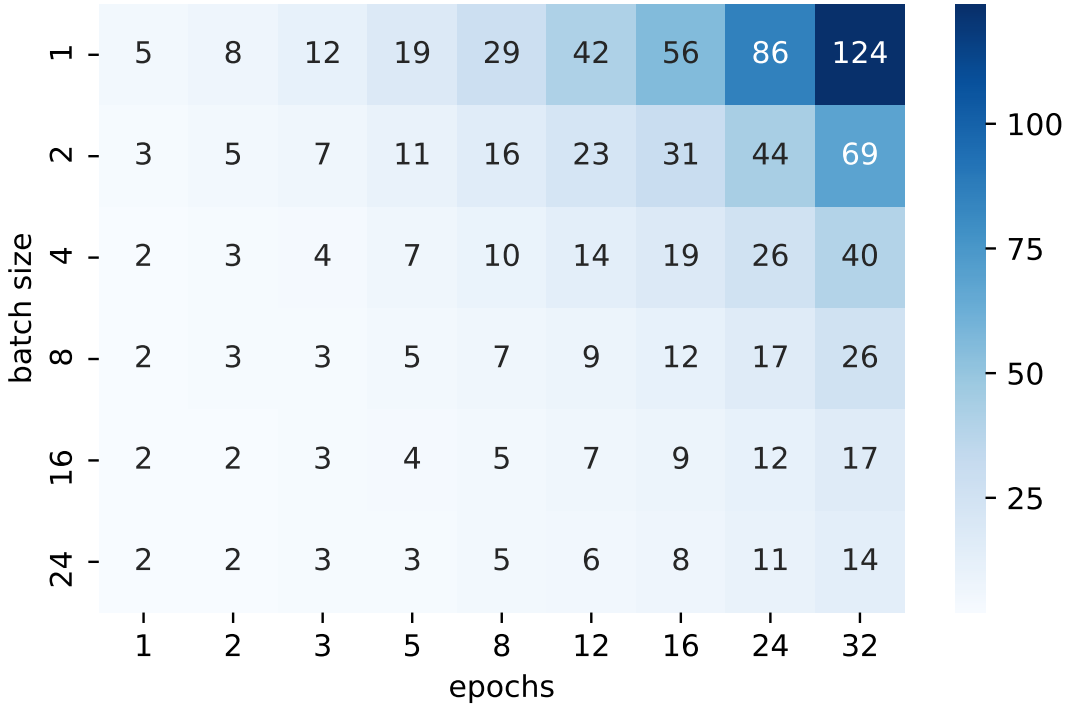

Abbildung 6.5: Dauer des Trainings der BERT-Modelle. Die gezeigten Werte spiegeln die durchschnittliche Trainingsdauer der jeweils fünf trainierten Modelle in Minuten wieder. Die Lernrate Beträgt hier in allen Durchläufen $2 * 10^{-5}$.

drei LinearSVC Modelle mit verschiedenen Hyperparameterkombinationen getestet. Des Weiteren wurde wurden Modelle für ein CompNB, MultiNP GauNB und DTree trainiert. Dies sind die Modellbezeichnungen, wie sie auch von der Bibliothek selbst verwendet werden.

Wie Abbildung 6.6 auf der nächsten Seite zeigt, werden die besten Ergebnisse von den SVM-Modellen erzielt. Der höchste F_1-Score lag hier bei 77,4%.

Es gibt allerdings auch Modelle, die gar nicht funktionieren. Gerade die Naive Bayes Modelle schneiden vergleichsweise schlecht ab. Der F_1-Score rückt diese Modell sogar noch in ein besseres Licht. Das trainierte *MultinomialNB*-Modell klassifiziert z. B. einen Großteil der Mails als projektbezogene Frage. Da diese Klasse die meisten Beispiele enthält, ist der gewichtete F_1-Score, gemessen an der schlechten Performance des Modells, relativ hoch. Die Verteilung der Klassifizierung des *MultinomialNB*-Modells wird in der Abbildung 6.7 auf der nächsten Seite dargestellt.

Die Unterschiede in der benötigten Zeit des Trainings zwischen den Modellen und zwischen den verschiedenen Kombinationen von Hyperparametern ist sehr gering. Allgemein benötigen diese hier betrachten Modelle vergleichsweise wenig Zeit, um ein Training abzuschließen. Hier dauert bereits das Laden dem von Spacy vortrainierten Sprachmodell vergleichsweise etwas länger.

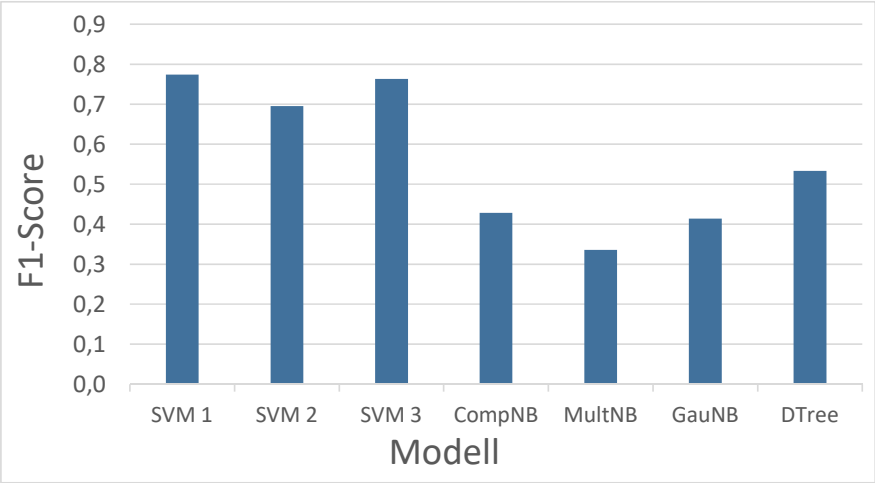

Abbildung 6.6: Trainingsergebnisse der getesteten Scikit-learn Modelle. Die gezeigten Werte spiegeln den F_1-Score der Modellauswertung wieder.

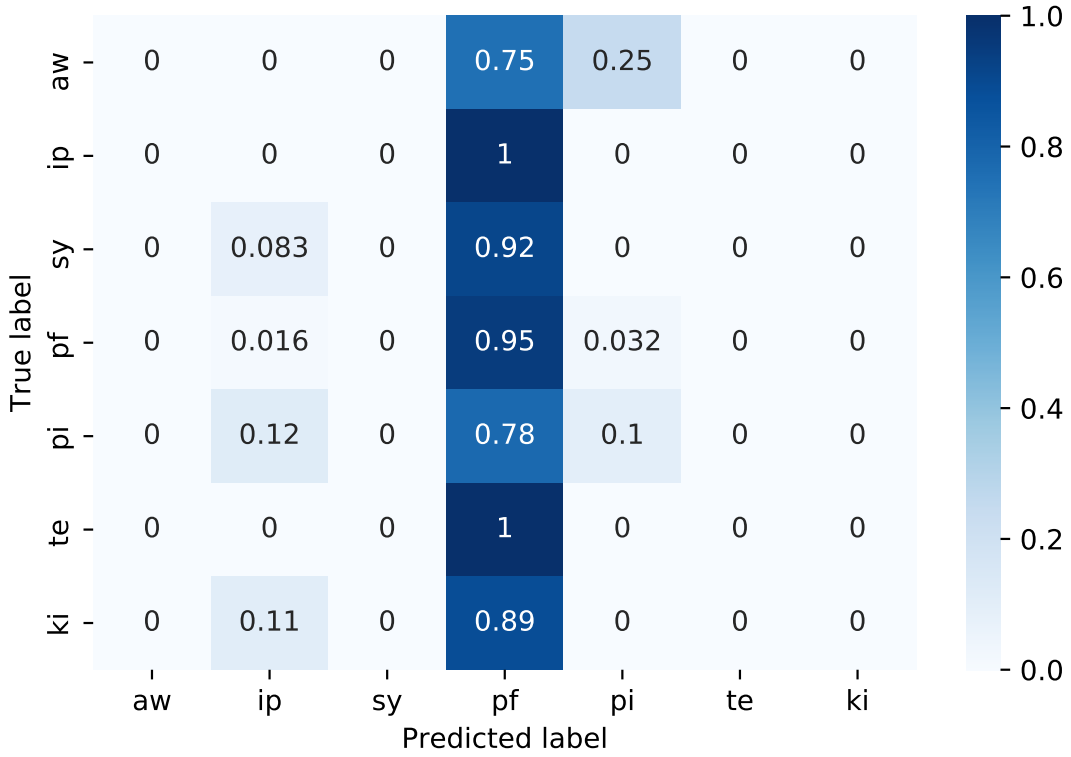

Abbildung 6.7: Normalisierte Heatmap der Klassifikation des MultiNB-Modells

6.2 Parameterextraktion

Die Extraktion der Parameter ist regelbasiert. Hier wurden für jede Funktionsklasse Parameter definiert, die dann aus dem E-Mailtext extrahiert werden sollen. In dieser Arbeit wurde beispielhaft die Parameter zweier Funktionsklassen extrahiert: Abwesenheitsnotiz und Terminvereinbarung. Bei den restlichen Funktionsklassen war es nicht möglich geeignete Parameter zu definieren, da diese E-Mails sehr individuell sind und oft ein eigenes Anliegen, das strukturell wenige Gemeinsamkeiten mit anderen Anliegen teilen, haben. Die Regelerstellung basierte auf den Trainingsdaten. Es wurde also anhand dieser Daten versucht die Parameter zu extrahieren. Anschließend wurde die Extraktion mit Hilfe der Textdaten verifiziert.

Für die Funktionsklasse *Abwesenheitsnotiz* (aw) wurden vier Parameter bestimmt. Der erste Parameter ist hier *date*. Dieser repräsentiert alle Datumsangaben, die in der E-Mail vorkommen. Hauptsächlich ist das Datum bis wann der Absender der E-Mail abwesend ist relevant. Bei der Extraktion wurde gute Ergebnisse erzielt: Hier wurden 33 Beispiele im Trainings- und vier Beispiele im Testdatensatz analysiert. Auf den Trainingsdaten konnte eine Genauigkeit von 94% erzielt werden. Hier konnten also bei zwei Datensätzen kein Datum extrahiert werden, obwohl hier ein Datum angegeben wurde. Auf den Testdaten lag die Genauigkeit bei 100% (bei vier Datensätzen). Der Parameter *urgent_address* gibt die E-Mail-Adresse an, die der Empfänger der E-Mail in dringenden Fällen kontaktieren soll. Dieser wurde in nur einem Fall des Trainingsdatensatzes falsch erkannt. In 14 Fällen wurde dieser korrekt extrahiert und in 18 Fällen wurde hierfür keine E-Mail-Adresse angegeben. Somit ergibt sich eine Genauigkeit von 97%. In 8 Fällen wird für dringende Fälle eine Telefonnummer angegeben. Hierfür gibt es in dieser Arbeit kein Extraktionsparameter. In vier Fällen wurden zusätzlich E-Mailadressen für bestimmte Aufgabengebiete angegeben. Diese wurden nicht erkannt, da sie nicht in Verbindung mit dringenden Fällen erwähnt wurden. Der Parameter *urgent_text* verhält sich im Bezug auf die Genauigkeitsstatistik gleich zu *urgent_address*. Auf den Testdaten wurde lediglich die E-Mail-Adresse eines Beispiels korrekt erkannt. Bei zwei Fällen war keine Adresse angegeben. In einem Fall wurde eine angegebene Adresse nicht erkannt. Es ergibt sich somit eine Genauigkeit von 75%. In drei Beispielen des Testdatensatzes wurde außerdem eine Telefonnummer angegeben. *urgent_text* wurde in drei Fällen korrekt erkannt. In dem letzten Fall wurde der relevante Satz der E-Mail nur teilweise erkannt.

Für die Funktionsklasse *Terminvereinbarung* (te) wurden zwei Hauptparameter definiert: Zum einen *te_action* und zum anderen *qu_re_sentences*. Mehr Informationen über die Bedeutung und Implementation dieser Parameter befinden sich in Abschnitt 5.7.2 auf Seite 45. Für diesen Parameter wurden 39 Beispiele im Trainings- und vier Beispiele im Testdatensatz analysiert. Die Fragen und Bitten aus den Trainings- und Testdaten wurden, bis auf eine Ausnahme, alle richtig erkannt. Die Genauigkeit liegt in diesem Fall folglich bei 98%. Dies wurde allerdings von Tramountani [Tra19] basierend auf ihrer Arbeit mit Hilfe eines seperaten BERT-Netzes klassifiziert. In dieser Arbeit wurde hierfür lediglich die Sätze, die eine Frage oder eine Bitte enthalten aus der menge der Sätze gefiltert. Der Parameter *te_action* wird basierend auf dem Verb des Satzes extrahiert. Auch hierzu befinden sich mehr Informationen in dem Abschnitt 5.7.2. Im Trainingsdatensatz wurden 19 Aktionen richtig und 16 Aktionen falsch bzw. nicht erkannt. Des Weiteren wurde in vier E-Mails keine Aktion genannt, was richtig erkannt wurde. Daraus ergibt sich eine Genauigkeit von 59%. Dies ist für einen regelbasierten Ansatz bereits ein akzeptables Ergebnis, es kann jedoch durch die Hinzunahme der Berücksichtigung eines anderer Satzbaus noch verbessert werden. Von

den Testdaten wurde die Aktion einer E-Mail richtig erkannt, bei einer gab es keine Aktion, was richtig erkannt wurde, und die Aktionen von zwei E-Mails wurde nicht erkannt. Dies führt zu einer Genauigkeit von 50%.

Allgemein lässt sich sagen, dass mit Hilfe von Methoden des maschinellen Lernen die Performance der Parameterextraktion vermutlich noch verbessert werden kann. Hierfür standen allerdings nicht genügend Variationen an E-Mails zur Verfügung.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein System entwickelt und beschrieben, welches E-Mails nach der Absicht des Senders klassifiziert. Hierfür wurden verschiedene Modelle des maschinellen Lernens verwendet und teilweise auch kombiniert. Diese Modelle wurden dann miteinander verglichen, um ihre Performance zu evaluieren. Dabei hat sich das vortrainierte neuronale Netz von Devlin et al. [DCLT18] als beste Lösung erwiesen. Das Netz der Google-Entwickler erreichte einen F_1-Score von 91,5%. In der Vorverarbeitung der Daten wurde zur Filterung der E-Mailtexte die Netzausgaben des trainierten Netzes von Tramountani [Tra19] verwendet. Dieses klassifiziert jeden Satz einer E-Mail nach der Art des Satzes. Dabei wurden auf den Datenbestand angepasste Klassen definiert.

Neben der Klassifizierung der E-Mails wurde in dieser Arbeit ein weiterer Schritt vollzogen. Es wurden für zwei geeignete Klassen jeweils Parameter definiert, welche die Absicht des Sender näher beschreiben. Diese wurden dann regelbasiert aus den E-Mails bestimmt. Dabei wurden die Entscheidungen von Natural-Language-Processing-Bibliotheken unterstützt. Die Performance dieser Extrahierung der Parameter hängt stark von der Diversität und Komplexität der E-Mails ab. So wurden die Parameter bei Abwesenheitsnotizen in 97% der E-Mails korrekt erkannt. Demgegenüber wurden bei einer Terminvereinbarung lediglich 78% der Parameterwerte der E-Mails korrekt extrahiert. Dabei gibt es allerdings wenige false-positives, da meist kein Wert für den Parameter gefunden wird.

7.1 Ausblick

Die Klassifizierung der E-Mails nach der Absicht des Senders ergab bereits gute Ergebnisse. Die verwendeten Klassen wurden anhand des vorliegenden Datensatzes ausgesucht. Hier besteht die Möglichkeit, die Klassen weiter zu verfeinern, sodass zum Beispiel eine Terminverschiebung getrennt von einer Absage klassifiziert wird. Somit muss diese Unterscheidung nicht bei der Parameterextraktion vollzogen werden. Es wäre auch denkbar, eine Klassifizierung in mehreren Schritten durchzuführen. Hier würde zunächst eine grobe Unterteilung stattfinden, wie sie in dieser Arbeit bereits verwendet wird. Danach würde ein weiteres Modell für eine genauere Klassifizierung zuständig sein. Hier könnte evaluiert werden, ob dieser zweistufige oder der davor erwähnte, einstufige Ansatz bessere Ergebnisse erzielen würde. Es sind hierfür allerdings vermutlich mehr Trainingsdaten, als sie dieser Arbeit zur Verfügung standen, vonnöten.

Bezüglich der Parameterextraktion erreicht ein neuronales Netz bessere Ergebnisse und ist besser erweiterbar, als regelbasierte Ansätze, da ein neuronales Netz komplexe Strukturen erlernen kann. Diese sind für Menschen of nicht nachvollziehbar und können deswegen nicht in Regeln definiert werden. Auch hier ist vermutlich ein deutlich größerer Datensatz notwendig. Die Bibliothek Rasa bietet hier zum Beispiel bereits Möglichkeiten, Parameter zu extrahieren. Hierfür werden gelabelte Trainingsdaten verwendet, in denen die Positionen der Beispiele im Text genau definiert werden muss.

Es müsste hierbei evaluiert werden, in wie weit ein Diversifikation des Satzbaus die Performance der Parameterextraktion beeinflusst. Dieser Sachverhalt würde sich gut als Basis für weitere Arbeiten eignen.

Darüber hinaus ist es möglich, aus den bereitgestellten Ergebnissen der Klassifikation und Parameterextraktion eine vollwertige Schnittstelle zu entwickeln. Hier würde ich z. B. ein REST-API anbieten.

Literaturverzeichnis

[Bro17a] J. Brownlee. *A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size*. 21. Juli 2017. URL: https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/ (zitiert auf S. 23).

[Bro17b] J. Brownlee. *What is the Difference Between a Parameter and a Hyperparameter?* 26. Juli 2017. URL: https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/ (zitiert auf S. 23).

[CMS+05] B. Cui, A. Mondal, J. Shen, G. Cong, K.-L. Tan. „On Effective E-mail Classification via Neural Networks“. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, S. 85–94. DOI: 10.1007/11546924_9 (zitiert auf S. 17).

[Dan17] M. Danilák. *Langdetect. Port of Google's language-detection library to Python*. 2017. URL: https://github.com/Mimino666/langdetect (zitiert auf S. 27).

[DCLT18] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova. „BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding“. In: *arXiv preprint arXiv:1810.04805* (2018) (zitiert auf S. 34, 36–38, 40, 55).

[Dör18] S. Dörn. *Neuronale Netze*. 2018. URL: https://sebastiandoern.de/neuronale-netze/ (zitiert auf S. 21).

[EBC+10] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, S. Bengio. „Why Does Unsupervised Pre-training Help Deep Learning?“ In: *Journal of Machine Learning Research 11* (2010), S. 625–660 (zitiert auf S. 24).

[Gan18] R. Gandhi. *Support Vector Machine—Introduction to Machine Learning Algorithms*. Towards Data Science. 7. Juni 2018. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47 (zitiert auf S. 19).

[HM19] M. Honnibal, I. Montani. *Spacy. Industrial-Strength Natural Language Processing*. 19. März 2019. URL: https://spacy.io/ (zitiert auf S. 34).

[Hun07] J. D. Hunter. „Matplotlib: A 2D graphics environment“. In: *Computing in Science & Engineering* 9.3 (2007), S. 90–95. DOI: 10.1109/MCSE.2007.55 (zitiert auf S. 32).

[Jet19] Jetbrains. *PyCharm. The Python IDE for Professional Developers*. 2019. URL: https://www.jetbrains.com/pycharm/ (zitiert auf S. 31).

[Kin94] W. Kinnebrock. *Neuronale Netze*. De Gruyter Oldenbourg, 22. Juni 1994. 184 S. ISBN: 3486229478. URL: https://www.ebook.de/de/product/17778636/werner_kinnebrock_neuronale_netze.html (zitiert auf S. 22).

[LC12] U. Lämmel, J. Cleve. *Künstliche Intelligenz*. Hanser, 11. Mai 2012. 332 S. ISBN: 978-3-446-42758-7 (zitiert auf S. 21–23).

[Lie02] E. von Lienen. „Neuronale Netze in der Robotik“. Seminarvortrag. TU Clausthal, 18. Jan. 2002. URL: http://www2.in.tu-clausthal.de/~reuter/ausarbeitung/Elke_von_Lienen_-_Neuronale_Netze_in_der_Robotik.pdf (zitiert auf S. 21).

[Lug03] G. F. Luger. *Künstliche Intelligenz. Strategien zur Lösung komplexer Probleme*. Pearson Studium, 2003. ISBN: 3827370027 (zitiert auf S. 20, 21).

[Met78] C. E. Metz. „Basic Principles of ROC Analysis“. In: *Seminars in Nuclear Medicine* 8.4 (Okt. 1978), S. 283–298. DOI: 10.1016/s0001-2998(78)80014-2 (zitiert auf S. 24).

[Mic19] Microsoft. *MailItem Interface*. Microsoft Corporation. 2019. URL: https://docs.microsoft.com/de-de/dotnet/api/microsoft.office.interop.outlook._mailitem?view=outlook-pia (zitiert auf S. 27).

[Moe17] J. Moeser. *Künstliche neuronale Netze – Aufbau & Funktionsweise*. 2017. URL: https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/ (zitiert auf S. 20–22).

[Nak10] S. Nakatani. *Language Detection Library for Java*. 2010. URL: https://github.com/shuyo/language-detection (zitiert auf S. 27).

[Neg11] M. Negnevitsky. *Artificial Intelligence. A Guide to Intelligent Systems*. Addison Wesley, 2011. ISBN: 1408225743 (zitiert auf S. 20, 21).

[Num19] Numpy. *NumPy*. 31. Jan. 2019. URL: https://www.numpy.org (zitiert auf S. 33).

[OD08] D. L. Olson, D. Delen. „Performance Evaluation for Predictive Modeling“. In: *Advanced Data Mining Techniques*. Springer Berlin Heidelberg, 2008, S. 137–147. DOI: 10.1007/978-3-540-76917-0_9 (zitiert auf S. 25).

[Ope14] Opencv. *Introduction to Support Vector Machines*. 2014. URL: https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html (zitiert auf S. 19).

[Pat18] P. Patel. *Why Python is the most popular language used for Machine Learning*. Udacity India. 8. März 2018. URL: https://medium.com/@UdacityINDIA/why-use-python-for-machine-learning-e4b0b4457a77 (zitiert auf S. 31).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830 (zitiert auf S. 33, 41, 43).

[Rad18] A. Radford. *Improving Language Understanding with Unsupervised Learning*. 2018. URL: https://openai.com/blog/language-unsupervised/ (zitiert auf S. 24).

[Ras19] Rasa. *Rasa NLU: Language Understanding for Chatbots and AI assistants*. 2019. URL: http://rasa.com/docs/rasa/nlu/about/ (zitiert auf S. 33–35, 47).

[Ray17] S. Ray. *Understanding Support Vector Machine algorithm from examples (along with code)*. Analytics Vidhya. 13. Sep. 2017. URL: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/ (zitiert auf S. 19).

[RNSS18] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever. „Improving Language Understandingby Generative Pre-Training“. 11. Juni 2018 (zitiert auf S. 24).

[Sch15] J. Schmidhuber. „Deep Learning in Neural Networks: An Overview“. In: *Neural Networks* 61 (Jan. 2015), S. 85–117. DOI: 10.1016/j.neunet.2014.09.003. arXiv: 1404.7828 (zitiert auf S. 20).

[SHG90] E. Schöneburg, N. Hansen, A. Gawelczyk. *Neuronale Netze. Einführung, Überblick und Anwendungsmöglichkeiten*. Makt&Technik, 1990. ISBN: 3-89090-329-0 (zitiert auf S. 19).

[Tie18] M. Tiedemann. *KI, künstliche neuronale Netze, Machine Learning, Deep Learning: Wir bringen Licht in die Begriffe rund um das Thema „Künstliche Intelligenz“*. 2018. URL: https://www.alexanderthamm.com/de/artikel/ki_artificial-intelligence-ai-kuenstliche-neuronale-netze-machine-learning-deep-learning/ (zitiert auf S. 20).

[Tra19] E. Tramountani. „Intent Classification for the Automation of Email Processing Tasks“. Bachelor's Thesis. Hochschule der Medien Stuttgart, 1. Feb. 2019 (zitiert auf S. 23, 25, 27, 53, 55).

[Was18] M. Waskom. *seaborn: statistical data visualization*. 2018. DOI: 10.5281/zenodo.1313201. URL: https://seaborn.pydata.org/index.html (zitiert auf S. 34, 43).

[Zul18] H. Zulkifli. *Understanding Learning Rates and How It Improves Performance in Deep Learning*. Towards Data Science. 21. Jan. 2018. URL: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 (zitiert auf S. 23).

Alle URLs wurden zuletzt am 10. 06. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift