Named Entity Recognition (NER)

Mohammad Matar 219200060

07.11.2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Grundlage von NER-Konzepte

1.1 Intro

Derzeit ist eine große Menge an digitalen Daten (wie E-Mails, soziale Anwendungen, Zeitungen und Instagram) in verschiedenen Sprachen verfügbar. Diese Informationen werden sowohl in strukturierter als auch unstrukturierter Form gesammelt, um die Daten zu verarbeiten und nützliche Informationen zu extrahieren. Es ist jedoch eine der größten Herausforderungen, bedeutungsvolle Kenntnisse aus solchen riesigen Datenmengen zu extrahieren. Das Hauptziel der natürlichen Sprachverarbeitung (Natural Language Processing, NLP) besteht darin, nützliche Informationen aus den menschlichen natürlichen und Informationssprachen zu gewinnen, damit Maschinen die Informationen der menschlichen Sprache besser verstehen und entsprechend handeln können. Viele Informationssysteme zur Extraktion von Informationen in NLP wurden entwickelt, die automatisch Fragen beantworten und Texte zusammenfassen [8]. Dazu gehört das sogenannte Information Extraction (IE), eine der wichtigen Aufgaben in der Textanalyse und der Verarbeitung natürlicher Sprache (Natural Language Processing, NLP), die das Extrahieren von bedeutungsvollen Wissensfragmenten aus unstrukturierten Informationsquellen umfasst, da unstrukturierte Daten rechnerisch undurchsichtig sind. Das Ziel von IE ist es, eine Wissensbasis zu erstellen, d.h. die Informationen so zu organisieren, dass sie für Menschen nützlich sind und in einer semantischen Weise angeordnet werden, sodass Algorithmen nützliche Schlussfolgerungen daraus ziehen können [5]. Eine der wesentlichen Unteraufgaben von IE ist das Named Entity Recognition (NER), bei dem es darum geht, Namen/Entitäten zu finden und zu klassifizieren. Sobald diese Named Entities (NE) extrahiert sind, können sie indexiert und durchsuchbar gemacht werden, Beziehungen abgeleitet, Fragen beantwortet und vieles mehr. Die NER-Techniken unterscheiden sich je nach Domain, aufgrund der Einzigartigkeit, die in jeder Domäne existiert, obwohl der Prozess auf einer Reihe grundlegender Schritte der natürlichen Sprachverarbeitung (NLP) basiert, wie z.B. Tokenisierung, Part-of-Speech-Tagging, Parsing und Modellbildung [5].

1.2 Motivation und Zielsetzung

Die zentrale Herausforderung dieser Arbeit liegt im `"WossiDi"`-Archiv, das die digitale Version des Zettelkastensystems von Richard Wossidlo (1859–1939) zur Verfügung stellt. Wossidlo, ein wegweisender Feldforscher, reiste durch ganz Mecklenburg und schuf – unterstützt von zahlreichen Sammelhelfern –

eine einzigartige ethnografische und regionalsprachliche Sammlung. Das Archiv umfasst eine Zettelsammlung, die nach Volksüberlieferungen, Arbeits- und Lebensbereichen systematisch verschlagwortet ist, sowie die dazugehörige Beiträgerkorrespondenz. Zudem enthält es wertvolle Hinweise zu überregionalen Traditionen und den alphabetischen Zettelkatalog Wossidlos, ergänzt durch den abgeleiteten Katalog von Hermann Teuchert, der als Grundlage für das Mecklenburgische Wörterbuch dient. Weitere Bestandsgruppen aus dem Umfeld des Wossidlo-Archivs bereichern das Material. WossiDiA stellt etwa zwei Millionen überwiegend handschriftliche Dokumente im freien Zugang online zur Verfügung. Diese sind durch Schlagwörter, Personen, Orte und Zeitangaben durchsuchbar. Transkriptionen sowie Erklärungen, etwa zu Kürzeln, erleichtern die Nutzung und ermöglichen eine benutzerfreundliche und effiziente Recherche [9]. Die Zielsetzung dieser Arbeit ist es, die Prinzipien der Named Entity Recognition (NER) zu erläutern und deren praktischen Nutzen anhand der Anwendung auf die transkribierten Texte der Korrespondenzen von Richard Wossidlo zu demonstrieren. Dabei wird untersucht, wie NER-Tools eingesetzt werden können, um historische Dokumente, insbesondere handschriftliche Transkripte, zu analysieren und strukturierte Informationen wie Namen von Personen, Orten und Zeitangaben automatisch zu extrahieren. Die Arbeit soll auch aufzeigen, wie NER zur Identifikation und Kategorisierung von Entitäten in den Korrespondenzen von Wossidlo beitragen kann, um deren Analyse und Interpretation zu erleichtern. Ziel ist es, die Möglichkeiten und Herausforderungen der Anwendung von NER auf historische Texte zu verdeutlichen und zu zeigen, wie diese Technologien die Forschung zu historischen und geisteswissenschaftlichen Datenbereichen verbessern können.

1.3 Definition der NER

Named Entity Recognition (NER) ist eine zentrale Aufgabe in der Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) und Information Extraction (IE). Es handelt sich dabei um den Prozess, bei dem benannte Entitäten wie Personennamen, Organisationen, Orte, Zeitangaben und andere bedeutende Begriffe aus unstrukturierten Texten extrahiert und klassifiziert werden. Das Hauptziel von NER besteht darin, aus unstrukturierten Daten strukturierte Informationen zu gewinnen, die für die weitere Analyse und maschinelle Verarbeitung von Bedeutung sind. Diese Entitäten werden in vordefinierte Kategorien eingeteilt, um den Text in einer maschinenlesbaren Form darzustellen, die für Algorithmen und Wissensdatenbanken nutzbar ist. Die Identifikation von benannten Entitäten ermöglicht es, wichtige Informationen zu extrahieren, die für Anwendungen wie Fragebeant-

wortung, Wissensgraphen und Textzusammenfassungen verwendet werden können [8] [5] [2]. Ein Beispiel für die Anwendung von NER in der Praxis wäre der Satz:`"Jhon kaufte 500 Aktien von Acme Corp. im Jahr 2016."` In diesem Fall werden `"Jhon"` als Person, `Äcme Corp."` als Organisation und `"2016"` als Zeitangabe erkannt und entsprechend klassifiziert [8].

2 Einsatz von Tools

In diesem Kapitel wurden verschiedene Tools für die NER untersucht und auf die spezifischen Anforderungen der historischen Daten des Projekts angewendet. Ziel ist es, die Leistung und Funktionalität der Tools zu evaluieren und das beste Werkzeug für zukünftige Anwendungen auszuwählen.

2.1 Überblick

In der Abschnitt wird ein umfassender Einblick in die Zielsetzung und die zentralen Aspekte der NER für historische Texte gegeben. Zunächst werden die zu erkennenden Entitäten definiert und ihre Relevanz für die Analyse erläutert. Anschließend werden die verschiedenen Eingabe- und Ausgabeformate beschrieben, um eine strukturierte Verarbeitung der Daten zu ermöglichen. Dieser Überblick dient als Grundlage für die weitere Analyse und Bewertung der eingesetzten NER-Tools.

2.1.1 Zielsetzung

Das Ziel diesses Kapitals besteht darin, verschiedene Tools für die NER einzusetzen, um deren Funktionalität und Leistung auf den spezifischen historischen Daten unseres Projekts zu bewerten. Dabei liegt der Fokus auf:

- Der Evaluation der Genauigkeit und Effizienz der Tools bei der Erkennung und Klassifizierung benannter Entitäten in historischen Texten.
- Dem Vergleich mehrerer Tools, um deren Stärken und Schwächen im Hinblick auf die Verwendbarkeit für unsere Daten zu analysieren.
- Der Identifikation des besten Ansatzes für zukünftige Anwendungen auf ähnliche Datenquellen.

2.1.2 Zu erkennenden Entitäten

Im Rahmen des Projekts sind die folgenden Entitätstypen definiert, die aus den Texten extrahiert werden sollen. Jede Kategorie wird basierend auf ih-

rer Relevanz für die Analyse und den Kontext der historischen Dokumente beschrieben:

Orte (`LOC`)

- Beschreibung: Bezieht sich auf geografische Standorte wie Städte, Länder oder Regionen.
- Beispiele: `<LOC>Rostock</LOC>`, `<LOC>Mecklenburg</LOC>`.

Personen (`PER`)

- Beschreibung: Namen von Individuen, einschließlich historischer Persönlichkeiten oder Autoren.
- Beispiele: `<PER>E Geinitz</PER>`, `<PER>Herr Dr. Krause</PER>`.

Organisationen (`ORG`)

- Beschreibung: Institutionen oder Organisationen, die in den Texten erwähnt werden.
- Beispiele: `<ORG>Statistisches Amt</ORG>`.

Ereignisse (`EVENT`)

- Beschreibung: Zeitbezogene oder thematische Ereignisse, die in den Dokumenten erwähnt werden.
- Beispiele: `<EVENT>7.1.08</EVENT>`, `<EVENT>Kommissionssitzung</EVENT>`.

Tiere (`ANIMAL`)

- Beschreibung: Erwähnte Tiere oder Tierarten in den Texten.
- Beispiele: `<ANIMAL>Pferd</ANIMAL>`, `<ANIMAL>Schwan</ANIMAL>`.

Die Definition dieser Entitätenkategorien ermöglicht eine strukturierte Extraktion der wichtigsten Informationen aus den Texten und bildet die Grundlage für die nachfolgende Analyse.

2.1.3 Eingabedatenformat

Die Eingabedaten, die bei der Verarbeitung der historischen Texten bereitgestellt werden, liegen in zwei Formaten, wo jeder Text in den beiden Formaten verfügbar ist, vor:

- **Annotierte-Format:**Ein strukturiertes Textformat, bei dem benannte Entitäten bereits durch Tags markiert sind. Beispiel:

```
<LOC>Rostock</LOC>, <EVENT>7.1.08</EVENT>.
Sehr geehrter <PER>Herr Dr.</PER>
Möchten Sie die Güte haben, mir für den Druck der „Beispiel\-Seite
Ihre festen Vorschläge aufschreiben?
```

- **Freitext-Format:** Texte ohne vorgegebene Struktur, die roh verarbeitet werden müssen. Beispiel:

```
4.11.14 Rostock, 6. 11. 08
Die Kommissionssitzung ist nunmehr auf
Sonnabend, d. 8. Nachm. nach 3 Uhr in Güstrow
festgesetzt.
```

2.1.4 Ausgabedatenformat

Die Ergebnisse der NER-Tools werden in einem JSON-Format bereitgestellt. Dieses Format strukturiert die extrahierten Informationen und macht sie für die Weiterverarbeitung nutzbar. Jedes Dokument wird durch die folgenden Attribute beschrieben:

- **ID (id)**: Eine eindeutige Identifikationsnummer des Dokuments, die es ermöglicht, die Ergebnisse den entsprechenden Eingabedaten zuzuordnen.
- **Absender (sender)**: Der Absender des Dokuments basierend auf dem Dokumentnamen.
- **Text (text)**: Der Originaltext des Dokuments, der für die NER verwendet wurde.
- **Entitäten (entities)**: Eine Liste der erkannten benannten Entitäten, wobei jede Entität durch die folgenden Attribute beschrieben wird:

- **Textausschnitt**: Der exakte Text, der als benannte Entität erkannt wurde.
- **Kategorie**: Die Klassifikation der Entität, wie z. B. `LOC` (Ort), `PER` (Person) oder `EVENT` (Ereignis).
- **Startposition**: Der Index des ersten Zeichens der Entität im Originaltext.
- **Endposition**: Der Index des letzten Zeichens der Entität im Originaltext.

Beispielausgabe:


```
{
  "results": [
    {
      "id": "21776",
      "sender": "Geinitz",
      "text": "Rostock, 6. 11. 08...",
      "entities": [
        [
          "Rostock",
          "LOC",
          0,
          7
        ],
        [
          "Herrn Peltz",
          "PER",
          199,
          210
        ]
      ]
    }
  ]
}
```


Die JSON-Struktur erlaubt es, die Ergebnisse einfach zu speichern, zu durchsuchen oder in anderen Anwendungen weiterzuverarbeiten.

2.2 Auswahl der Tools

Für NER in diesem Projekt wurden verschiedene Tools analysiert, um diejenigen auszuwählen, die den Anforderungen am besten entsprechen.

2.2.1 Kriterien für die Auswahl

Die folgenden Kriterien wurden verwendet, um die am besten geeigneten Tools für die NER auszuwählen. Diese Kriterien gewährleisten, dass die Tools sowohl den technischen Anforderungen als auch den spezifischen Bedürfnissen des Projekts gerecht werden:

- **Genauigkeit**: Wie präzise und vollständig das Tool benannte Entitäten erkennt und klassifiziert.
- **Anpassungsfähigkeit**: Die Möglichkeit, das Tool an domänenspezifische oder historische Texte anzupassen.
- **Unterstützung der deutschen Sprache**: Das Tool muss vortrainierte Modelle oder eine einfache Anpassung für die Verarbeitung deutscher Texte bieten, da die Analyse historischer Dokumente in deutscher Sprache erfolgt.
- **Unterstützung neuer Entitäten**: Das Tool sollte es ermöglichen, zusätzliche oder benutzerdefinierte Entitätsklassen (z. B. `ANIMAL` oder `EVENT`) effizient zu integrieren und zu trainieren.
- **Benutzerfreundlichkeit**: Wie einfach die Installation, Konfiguration und Nutzung des Tools ist, insbesondere für Benutzer ohne tiefgehende technische Vorkenntnisse.
- **Performance**: Wie schnell und ressourcenschonend das Tool große Mengen an Textdaten verarbeiten kann, ohne die Genauigkeit der Analyse zu beeinträchtigen.

2.2.2 Ausgewählte einzusetzende Tools

Für die Erkennung von den Entitäten in den Texten werden gezielt folgende Tools basierend auf den oben vorgestellten Kriterien ausgewählt und eingesetzt:

- **Spacy**:

  - **Beschreibung**: Spacy ist ein Open-Source-Softwarepaket für fortgeschrittene natürliche Sprachverarbeitung in Python. Es bietet schnelle und effiziente vortrainierte Modelle für NER und andere natürliche Sprachverarbeitungsaufgaben [3].
  - **Stärken**:
    * Sehr schnelle Verarbeitungsgeschwindigkeit, auch für große Textmengen [3].
    * Benutzerfreundlich und gut dokumentiert mit einer aktiven Community [3].
  - **Schwächen**:
    * Begrenzte Flexibilität für domänenspezifische Anpassungen ohne zusätzliches Training [3].
    * Modelle sind primär auf moderne und nicht auf historische Texte optimiert [3].

- **Flair**:
  - **Beschreibung**: Flair ist ein einfach zu verwendendes Framework für modernste natürliche Sprachverarbeitung, entwickelt von der Humboldt-Universität zu Berlin. Es bietet vortrainierte Modelle für NER und ermöglicht benutzerdefinierte Anpassungen[1].
  - **Stärken**:
    * Hohe Genauigkeit durch kontextuelle Einbettungen wie *Flair Embeddings* [1].
    * Flexibilität für domänenspezifische Anwendungen durch Training eigener Modelle [1].
  - **Schwächen**:
    * Vergleichsweise langsame Verarbeitung bei großen Datenmengen [1].
    * Höhere Anforderungen an die Rechenleistung, insbesondere bei der Nutzung von GPU [1].

Beide Tools bieten spezifische Vorteile: Flair zeichnet sich durch hohe Genauigkeit aus und ist besonders für anspruchsvolle NER-Aufgaben geeignet, während Spacy für schnelle und effiziente Verarbeitung moderner Texte ideal ist. Der Vergleich und die Bewertung dieser Tools werden im weiteren Verlauf detailliert behandelt.

2.3 Einsatz des Spacy-Tools

In diesem Abschnitt wird untersucht, wie Spacy für die Extraktion benannter Entitäten in historischen Texten eingesetzt wurde. Dabei werden die verfügbaren Modelle vorgestellt und die konkrete Implementierung des gewählten Modells für die Analyse unserer Daten beschrieben. Ziel ist es, die Effizienz und Genauigkeit von Spacy in diesem speziellen Anwendungsfall zu bewerten.

2.3.1 Verfügbare Modelle vom Spacy

Spacy bietet vier vortrainierte Modelle für die deutsche Sprache, die auf Nachrichten- und geschriebene Texte optimiert sind und die Erkennung von vier Entitätsklassen ermöglichen: `PER` (Personen), `ORG` (Organisationen), `LOC` (geografische Orte) und `MISC` (sonstige Kategorien) [4].

- **de_core_news_sm**: Ein kleines Modell, das grundlegende Funktionen wie Syntax- und Entitätserkennung bietet.
- **de_core_news_md**: Ein mittleres Modell mit zusätzlichen Wortvektoren zur Verarbeitung komplexerer Daten.
- **de_core_news_lg**: Ein großes Modell mit erweitertem Vokabular und detaillierten Wortvektoren.
- **de_dep_news_trf**: Ein Transformermodell für Abhängigkeitsparsing und Syntaxanalyse, basierend auf Nachrichten- und geschriebener Sprache.

Die vortrainierten Modelle von Spacy ermöglichen eine Erkennung von der Entitäten, ohne dass zusätzliches Training erforderlich ist.

2.3.2 Einsatz des Spacy-Modells

In diesem Projekt kam das Modell *de_core_news_lg* zum Einsatz, um benannte Entitäten wie Personen (`PER`), Orte (`LOC`) und Organisationen (`ORG`) aus den Texten zu extrahieren. Der Einsatz des Spacy-Modells erfolgt durch die Methode *classify_entities_std_Spacy*, die als Python-Quellecode in Listing 3 dargestellt ist und wie folgt arbeitet:

- **Modell laden:** Das Modell wird mit der Funktion *Spacy.load("de_core_news_lg")* geladen. Dieses Modell ist speziell auf die Verarbeitung der deutschen Sprache optimiert und enthält vortrainierte Pipelines für NER.

- **Metadaten extrahieren:** Vor der Textanalyse werden aus dem Dateinamen Metadaten wie *sender* und *id* der Datei extrahiert. Dies wird durch die Funktion *extract_meta_data(file_path)* erreicht, die den Dateinamen analysiert und relevante Informationen extrahiert.
- **Textverarbeitung:** Der Text der Eingabedatei wird in ein Spacy-Dokument mit *doc = nlp(text)* umgewandelt, das dann durchlaufen wird, um alle erkannten Entitäten zu extrahieren.
- **Extraktion der Entitäten:** Für jede erkannte Entität im Text werden die folgenden Informationen gesammelt:
  - **Text der Entität** (*ent.text*): Der erkannte Begriff oder Name.
  - **Kategorie der Entität** (*ent.label_*): Die Klassifikation der Entität, z. B. `LOC` (Ort), `PER` (Person) oder `ORG` (Organisation).
  - **Startposition im Text** (*ent.start_char*): Die Position, an der die Entität im Text beginnt.
  - **Endposition im Text** (*ent.end_char*): Die Position, an der die Entität im Text endet.
- **Speicherung der Ergebnisse:** Die extrahierten Entitäten werden in einem strukturierten JSON-Format in der Datei *std_Spacy_model_output.json* gespeichert.

Das Spacy-Modell wird direkt auf die Eingabedaten angewendet, ohne dass zusätzliches Training erforderlich war. Es bietet eine einfache Möglichkeit, benannte Entitäten wie Personen, Orte und Organisationen zu identifizieren und die Ergebnisse in einem strukturierten Format bereitzustellen.

2.4 Einsatz des Flair-Tools

In diesem Abschnitt wird untersucht, wie das Standardmodell von Flair zur Erkennung benannter Entitäten in historischen Texten eingesetzt wurde. Ziel ist es, die Funktionalität, die Genauigkeit sowie die Anwendbarkeit des Tools auf unsere spezifischen Daten zu bewerten und mit anderen Modellen zu vergleichen.

2.4.1 Verfügbare Modelle von Flair

Flair bietet eine Vielzahl vortrainierter Modelle für die Erkennung benannter Entitäten in verschiedenen Sprachen an. Diese Modelle erkennen ebenfalss dieselben Entitätsklassen bei Spacy. Für die deutsche Sprache stehen unter anderem folgende Modelle zur Verfügung [7]:

- **de-ner**: Ein Standardmodell für die Erkennung deutscher Entitäten, das Personen (`PER`), Orte (`LOC`), Organisationen (`ORG`) und weitere Kategorien identifiziert. Dieses Modell bietet ein ausgewogenes Verhältnis zwischen Genauigkeit und Geschwindigkeit.
- **de-ner-large**: Ein erweitertes Modell mit höherer Genauigkeit, das jedoch mehr Rechenressourcen benötigt. Es ist besonders geeignet für Anwendungen, bei denen die Präzision im Vordergrund steht.

Die Modelle von Flair basieren auf kontextualisierten String-Embeddings und wurden auf umfangreichen deutschen Textkorpora trainiert, um eine hohe Erkennungsrate für verschiedene Entitätstypen zu gewährleisten. [7]

2.4.2 Einsatz des Flair-Modells

Das vortrainierte Modell *de-ner* von Flair wurde aufgrund seines ausgewogenen Verhältnisses von Genauigkeit und Geschwindigkeit ausgewählt und verwendet, um benannte Entitäten aus deutschen Texten zu extrahieren. Der Einsatz des Flair-Modells erfolgt im Code durch die Methode *classify_entities_std_flair*, die als Python-Quellecode in Listing 3 dargestellt ist und wie folgt arbeitet:

- **Modell laden:** Das Modell *de-ner* wird mit der Funktion *Classifier.load("de-ner")* geladen. Dieses Modell ist speziell für die Verarbeitung der deutschen Sprache optimiert und enthält vortrainierte Pipelines für NER.
- **Metadaten extrahieren:** Vor der Textanalyse werden aus dem Dateinamen Metadaten wie *sender* und *id* der Datei extrahiert. Dies wird durch die Funktion *extract_meta_data(file_path)* erreicht, die den Dateinamen analysiert und relevante Informationen extrahiert.
- **Textverarbeitung:** Der Text der Eingabedatei wird in ein Flair-Sentence-Objekt durch *sentence = Sentence(text)* umgewandelt, das dann von der Modellinstanz mit *tagger.predict(sentence)* vorhergesagt wird.
- **Extraktion der Entitäten:** Für jede erkannte Entität im Text werden die folgenden Informationen gesammelt:
  - **Text der Entität** (*entity.text*): Der erkannte Begriff oder Name.
  - **Kategorie der Entität** (*entity.tag*): Die Klassifikation der Entität, z. B. `LOC` (Ort), `PER` (Person) oder `ORG` (Organisation).
  - **Startposition im Text** (*entity.start_position*): Die Position, an der die Entität im Text beginnt.

  - **Endposition im Text** (*entity.end_position*): Die Position, an der die Entität im Text endet.

- **Speicherung der Ergebnisse:** Die extrahierten Entitäten werden in einem strukturierten JSON-Format in der Datei *std_flair_model_output.json* gespeichert.

Das Flair-Modell wird auch wie Spacy-Modell direkt auf die Eingabedaten angewendet, ohne dass zusätzliches Training erforderlich war. Es bietet eine Möglichkeit, benannte Entitäten wie Personen, Orte und Organisationen zu identifizieren und die Ergebnisse in einem strukturierten Format bereitzustellen.

2.5 Einsatz des weiter zu trainierenden Flair-Modells

Aufgrund der Ergebnisse der Standardmodelle von Flair und Spacy sind deutlich, dass diese für die spezifischen Anforderungen historischer Texte nicht optimal geeignet sind, wie in 2.6 vorgestellt wird. Daher wird im Rahmen dieses Projekts auch ein eigenes Flair-Modell mit spezifischen Daten und benutzerdefinierten Entitäten trainiert. Ziel ist es, die Erkennungsleistung für domänenspezifische Textdaten zu verbessern. Der Trainingsprozess umfasste mehrere Schritte, darunter die Aufbereitung der Trainingsdaten, das Modelltrainig sowie die Evaluation und Anwendung des neu trainierten Modells auf bisher unbekannte Textdaten [6].

2.5.1 Vorverarbeitung der Trainingsdaten

Die Rohdaten bestanden aus annotierten Texten, in denen Entitäten mit Tags wie `<LOC>`, `<PER>` oder `<ORG>` markiert waren. Diese Texte wurden in das BIO-Format (Beginning, Inside, Outside) konvertiert, um sie für das Training des Flair-Modells vorzubereiten. Die Umsetzung dieser Konvertierung erfolgt durch die Funktion *parse_annotated_text*, die als Python-Quellcode in Listing 1 dargestellt ist. Das BIO-Format enthält folgende Kennzeichnungen:

- `B-(Beginning)`: Beginn einer Entität (z. B. `B-LOC` für den Anfang einer Ortsangabe).
- `I-(Inside)`: Innerhalb einer Entität (z. B. `I-LOC` für die Fortsetzung der Ortsangabe).
- `O-(Outside)`: Kein Teil einer Entität.

Ein Beispiel für das BIO-Format:

```
31    B-EVENT
1     I-EVENT
1999  I-EVENT
ist   O
eine  O
Stadt O
.     O
```

Die in das BIO-Format konvertierten Daten werden anschließend mit der Funktion *split_train_file*, dargestellt in Listing 1, in drei Dateien unterteilt, die für das Modelltraining erforderlich sind:

- **train.txt**: Enthält die Trainingsdaten zur Modellanpassung.
- **dev.txt**: Dient der Validierung und Überwachung der Modellleistung während des Trainings.
- **test.txt**: Wird zur abschließenden Bewertung des trainierten Modells verwendet.

Diese drei Dateien bilden die Grundlage für das Training und die Evaluierung des Modells. Während *train.txt* zur Modelloptimierung genutzt wird, dient *dev.txt* zur laufenden Leistungsüberprüfung, und *test.txt* wird zur finalen Bewertung der Modellgenauigkeit herangezogen.

2.5.2 Training des Modells

Das Flair-Modell wird dann mit den vorbereiteten Daten trainiert. Hierbei werden *de* Word-Embeddings sowie Flair-Embeddings *(de-forward und de-backward)* kombiniert, um eine verbesserte Kontexterkennung zu ermöglichen. Der Trainingsprozess erstreckt sich über 50 Epochen und wird unter Verwendung der Klasse `ModelTrainer` durchgeführt. Der gesamte Trainingsablauf ist in Listing 2 dargestellt und umfasst die folgenden Schritte:

- **Dateninitialisierung**: Die Trainings-, Test- und Validierungsdaten werden aus dem definierten Verzeichnis eingelesen und im CoNLL-Format verarbeitet. Dies geschieht durch die Funktion:

```
1      corpus: Corpus = ColumnCorpus(
2          self.train_data_folder,
3          column_format={0: "text", 1: "ner"},
4          train_file="train.txt",
5          test_file="test.txt",
6          dev_file="dev.txt")
```

- **Erstellung eines Label-Wörterbuchs**: Mit der Funktion `make_label_dictionary()` wird ein Wörterbuch aller in den Trainingsdaten vorkommenden Entitätsklassen erstellt:

```
label_type = "ner"
label_dict = corpus.make_label_dictionary(label_type=
    label_type)
```

- **Definition der Embeddings**: Die Trainingsdaten werden mit drei Arten von vortrainierten Embeddings ausgestattet: Word-Embeddings für die deutsche Sprache sowie zwei Flair-Embeddings für die Vorwärts- und Rückwärtsrichtung. Diese werden mittels der Funktion `StackedEmbeddings` kombiniert:

```
embedding_types_de = [
                        WordEmbeddings("de"),
                        FlairEmbeddings("de-forward"),
                        FlairEmbeddings("de-backward")
                      ]
stacked_embeddings = StackedEmbeddings(
    embedding_types_de)
```

- **Erstellung des Sequence-Taggers**: Das NER-Modell wird unter Verwendung der `SequenceTagger`-Klasse mit einer versteckten Größe von 256, CRF-Schicht und den zuvor definierten Embeddings konfiguriert:

```
tagger = SequenceTagger(
                        hidden_size=256,
                        embeddings=stacked_embeddings,
                        tag_dictionary=label_dict,
                        tag_type=label_type,
                        use_crf=True
                        )
```

- **Training des Modells**: Der Trainingsprozess wird mit einer Lernrate von 0.1, einer Mini-Batch-Größe von 16 und einer GPU-gestützten Embedding-Speicherung über 50 Epochen durchgeführt:

```
trainer = ModelTrainer(tagger, corpus)
trainer.train(
                  self.models_folder,
                  learning_rate=0.1,
                  mini_batch_size=16,
                  max_epochs=self.epochs,
                  embeddings_storage_mode="gpu"
                )
```

- **Speichern des Modells** Nach Abschluss des Trainings werden zwei Versionen des trainierten Modells automatisch erzeugt und gespeichert, um eine flexible Nutzung je nach Anwendungsfall zu ermöglichen. Die gespeicherten Modellversionen sind:

  - **Final-Model**: Das Modell der letzten Trainingsepoche wird unter dem Namen *final-model.pt* gespeichert. Dieses Modell repräsentiert den Endzustand des Trainings und ermöglicht es, die zuletzt trainierte Version für zukünftige Anwendungen erneut zu laden und einzusetzen.
  - **Best-Model**: Zusätzlich wird das Modell mit der besten Leistung während des Trainingsprozesses unter dem Namen *best-model.pt* gespeichert. Dieses Modell wird anhand der besten validierten Metriken bestimmt und ist besonders für Anwendungen geeignet, bei denen maximale Genauigkeit erforderlich ist.

Die Speicherung beider Modelle stellt sicher, dass sowohl eine kontinuierliche Weiterentwicklung als auch eine optimale Modellnutzung je nach Anforderungen gewährleistet sind.

2.5.3 Anwendung des Modells

Die gespeicherten trainierten Modelle, *final-model.pt* und *best-model.pt*, werden verwendet, um benannte Entitäten aus neuen Texten zu extrahieren. Der gesamte Prozess der Modellanwendung ist im Python-Quellcode in Listing 3 durch die Methode *classify_entities* implementiert und umfasst folgende Schritte:

- **Laden des Modells:** Das gewünschte Modell wird mit der Methode *SequenceTagger.load(model_path)* geladen. Hier kann entweder das *final-model.pt* oder das leistungsstärkere *best-model.pt* gewählt werden.

```
1       self.model = SequenceTagger.load(self.model_path)
```

- **Lesen der Eingabedaten:** Alle Textdateien aus dem angegebenen Verzeichnis werden gelesen und die Zeichenkodierung wird mit der Funktion *detect()* automatisch erkannt.

```
1       txt_files = glob.glob(os.path.join(self.
           input_data_folder, "*.txt"))
2
3       with open(file_path, "rb") as file:
```

```
        raw_data = file.read()
        encoding_info = detect(raw_data)
        detected_encoding = encoding_info['encoding']

    content = raw_data.decode(detected_encoding)
```

- **Extraktion von Metadaten:** Der Absender und die Dokumenten-ID werden aus dem Dateinamen extrahiert, um die Ergebnisse besser zuordnen zu können.

```
    sender, file_id = self.extract_meta_data(file_path)
```

- **Entitätenerkennung:** Die gelesenen Texte werden mit der Methode *Sentence()* in Flair verarbeitet, und die erkannten Entitäten werden extrahiert.

```
    sentence = Sentence(text, use_tokenizer=True)
    self.model.predict(sentence)
    for entity in sentence.get_spans("ner"):
        entities.append([entity.text, entity.tag, entity.
            start_position, entity.end_position])
```

- **Speicherung der Ergebnisse** Die Ergebnisse werden in einem JSON-Format in der Datei *best-model.pt_output.json* oder *final-model.pt_output.json* je nach der geladenen Modellversion gespeichert, das die Entitäten mit Positionen und Kategorien darstellt.

```
    with open(self.output_file, "w", encoding="utf-8") as
        json_file:
        json.dump(all_results, json_file, indent=2,
            ensure_ascii=False)
```

2.5.4 Trainingsverlauf und Ergebnisse

Der Trainingsverlauf des benutzerdefinierten Flair-Modells wird mithilfe von unterschiedlichen Metriken überwacht. Diese folgenden Metriken dienen zur Bewertung der Modellleistung in Bezug auf die Erkennung und Klassifikation benannter Entitäten:

- **Präzision (Precision):** Der Anteil der korrekt erkannten Entitäten im Vergleich zu allen vom Modell vorhergesagten Entitäten.
- **Recall:** Der Anteil der korrekt erkannten Entitäten im Vergleich zu allen tatsächlich vorhandenen Entitäten.

- **F1-Score:** Das harmonische Mittel von Präzision und Recall, das eine ausgewogene Bewertung der Modellleistung ermöglicht.
- **Accuracy**: Der Anteil korrekt klassifizierter Wörter (Entität oder keine Entität) in Relation zu allen Wörtern im Text.

Während des Trainings wird die Leistung des Modells in jeder Epoche gemessen. Der folgende Verlauf zeigt die Entwicklung des F1-Score-Wertes über 50 Epochen:

```
Epoche 10: F1 = 40.78\%
Epoche 20: F1 = 53.33\%
Epoche 30: F1 = 57.14\%
Epoche 40: F1 = 59.77\%
Epoche 50: F1 = 63.53\%
```

Nach einem Trainingsverlauf sind beispielweise die folgenden Ergebnisse von (Precision, Recall, F1-Score und Accuracy) abhängi von den 5 Entitäten erzielt:

```
Results:
- F-score (micro) 0.5217
- F-score (macro) 0.4201
- Accuracy 0.3576

By class:
              precision    recall  f1-score

       EVENT     0.5745    0.7105    0.6353
       LOC       0.4545    0.5000    0.4762
       PER       0.5000    0.4783    0.4889
       ORG       0.0000    0.0000    0.0000
       ANIMAL    1.0000    0.3333    0.5000
```

Das trainierte Modell zeigt eine moderate Leistung mit einem finalen F1-Score von **63.53%**. Ereignisse (`EVENT`) wurden mit einem F1-Score von **63.53%** am besten erkannt, während Organisationen (`ORG`) überhaupt nicht identifiziert wurden (F1-Score: **0.00%**). Ortsangaben (`LOC`) und Personennamen (`PER`) erreichten mäßige Werte, was auf unzureichende Trainingsdaten hinweist. Tiere (`ANIMAL`) wurden mit hoher Präzision (**100.00%**), aber geringem Recall (**33.33%**) erkannt. Insgesamt zeigt das Modell Schwächen bei seltenen Entitäten und unausgewogene Trainingsdaten.

2.6 Evaluation der eingesetzten Modelle

In diesem Abschnitt werden die Leistungen der eingesetzten Modelle zur NER evaluiert. Ziel der Evaluation ist es, die Genauigkeit und Effizienz der Modelle bei der Erkennung und Klassifizierung benannter Entitäten in historischen Texten zu bewerten. Dazu wurden die 3 verschiedene Modelle auf identischen Evaluationsdaten getestet und anhand standardisierter Metriken wie Precision, Recall, F1-Score und Accuracy verglichen. Die gewonnenen Erkenntnisse helfen dabei, die Stärken und Schwächen der einzelnen Modelle zu identifizieren und Optimierungspotenziale für zukünftige Anwendungen aufzuzeigen.

2.6.1 Evaluationsmethodik und Vergleich der Modelle

Zur Evaluation der eingesetzten Modelle werden drei verschiedene Modelle auf denselben, nicht annotierten Texten anhand vordefinierter Metriken angewendet und bewertet. Der gesamte Evaluationsprozess ist als Python-Quellcode in Listing 4 implementiert und umfasst folgende Schritte:

- **Datenverarbeitung und Extraktion der Grundwahrheit (Ground Truth):** Annotierte Textdateien werden eingelesen, Metadaten extrahiert und Entitäten im JSON-Format gespeichert. Dies erfolgt durch die Methoden *extract_ground_truth*. Die Modelle werden auf dieselben Texte angewendet, und die extrahierten Entitäten werden mit der Grundwahrheit verglichen. Dabei wird die Methode *match_flexible* genutzt, um wahre und erkannte Entitäten zu vergleichen.
- **Berechnung der Metriken:** Die Leistung der Modelle wird anhand von Präzision, Recall, F1-Score und Accuracy evaluiert. Diese Berechnungen werden mit der Methode *calculate_metrics* durchgeführt.
- **Visualisierung der Ergebnisse:** Schließlich werden die Ergebnisse in Form von Diagrammen mit *plot_metrics* und *plot_individual_metrics* visualisiert.

Die Ergebnisse dieser Evaluation werden im JSON-Format gespeichert und erlauben eine direkte Gegenüberstellung der Modelle, wobei folgende Modelle evaluiert wurden:

- **Weiter trainiertes Flair-Modell** (`Final Model und Best Model`): Das Modell, das auf domänenspezifischen Texten trainiert ist, um die Erkennung von Entitäten zu optimieren.

- **Standardmodell von Flair** (`Standard Flair Model`): Ein vortrainiertes Modell, das ohne zusätzliche Anpassungen verwendet wird.
- **Standardmodell von Spacy** (`Standard Spacy Model`): Ein vortrainiertes Modell aus der Spacy-Bibliothek, das ebenfalls direkt eingesetzt wird.

Durch die Evaluierung dieser Modelle können wichtige Erkenntnisse über ihre Stärken und Schwächen bei der Erkennung und Klassifikation benannter Entitäten gewonnen werden.

2.6.2 Auswertung der Evaluation

Die Ergebnisse der Evaluation der drei Modelle sind in Abbildung 1, 2, 3 dargestellt. Die nachfolgenden Abbildungen zeigen eine detaillierte Beschreibung der Ergebnisse der Modelle hinsichtlich der Metriken *Precision*, *Recall*, *F1-Score* und *Accuracy*. Dabei werden verschiedene Kombinationen von Trainings- und Evaluationsdaten berücksichtigt, wie etwa Modelle, die mit Daten von Nachrichten eines einzelnen Absenders oder mit einer Mischung aus mehreren Absendern trainiert und evaluiert werden. Diese geben einen Überblick über die Leistungsunterschiede der Modelle unter den jeweiligen Bedingungen.

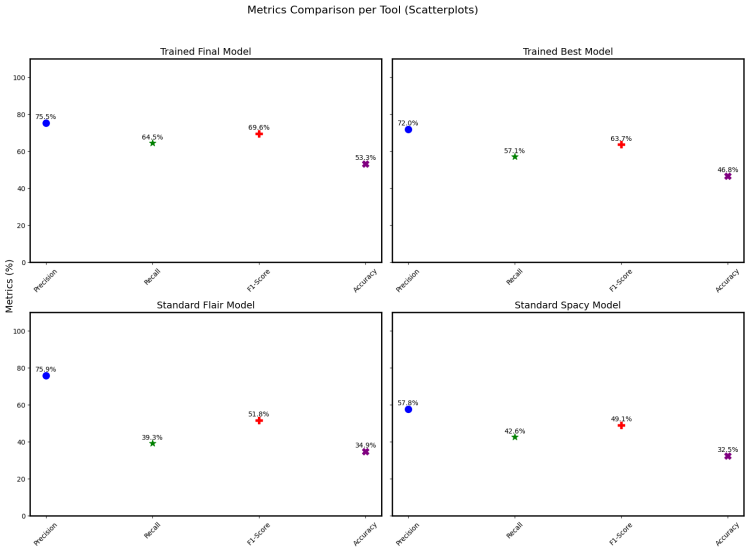

Abbildung 1: Die Trainingsdaten wurden aus Texten von einem bestimmten Absender erstellt, während die Evaluationsdaten von einem einzigen Absender stammen

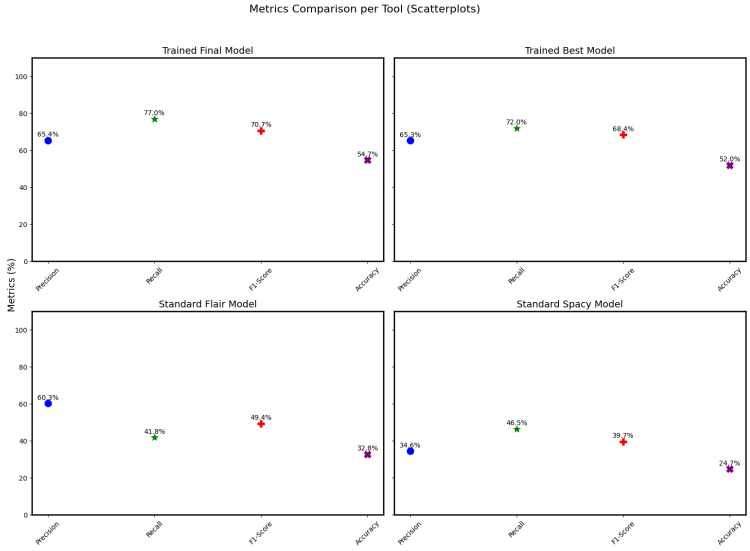

Abbildung 2: Die Trainingsdaten wurden aus einer Mischung von Daten mehrerer Absender erstellt, während die Evaluationsdaten von einem einzigen Absender stammen.

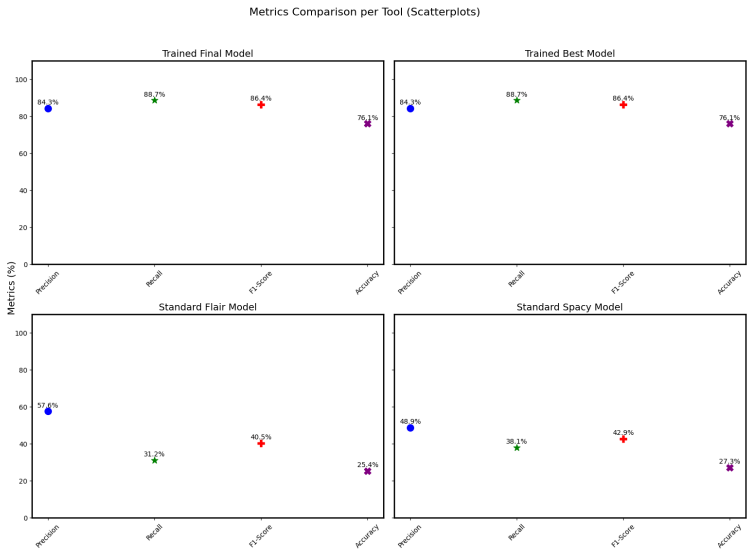

Abbildung 3: Sowohl die Trainingsdaten als auch die Evaluationsdaten wurden aus einer Mischung von Daten verschiedener Absender zusammengestellt.

2.6.3 Analyse der Ergebnisse und Schlussfolgerung

Die Analyse der Abbildungen zeigt, dass das weiter trainierte Flair-Modell (`Final Model und Best Model`) die beste Leistung unter den getesteten Modellen erzielt hat. Besonders der hohe Recall verdeutlicht, dass das Modell nahezu alle relevanten Entitäten erkennen konnte. Der F1-Score des trainierten Modells zeigt ein ausgewogenes Verhältnis zwischen Präzision und Recall, was auf die effektive Anpassung des Modells an die domänenspezifischen Trainingsdaten zurückzuführen ist. Die Abbildungen machen jedoch auch deutlich, dass die Art der Trainings- und Evaluationsdaten eine entscheidende Rolle für die Ergebnisse spielt. Flair-Modelle, die mit einer Mischung von Daten aus verschiedenen Absendern trainiert wurden, schnitten bei der Evaluation mit Daten eines einzelnen Absenders schlechter ab. Ebenso zeigen die Abbildungen, dass ein Modell, das mit Daten eines bestimmten Absenders trainiert und mit Daten eines anderen Absenders evaluiert wurde, in seiner Leistung limitiert ist. Diese Unterschiede verdeutlichen, dass eine Konsistenz zwischen Trainings- und Evaluationsdaten die Modellleistung stark beeinflusst. Im Vergleich dazu zeigen die Standardmodelle von Flair und Spacy in den Abbildungen geringere Leistungen. Das Standardmodell von Flair liefert zwar ähnliche Ergebnisse, erreicht jedoch weder die Präzision noch den Recall des weiter trainierten Modells. Das Standardmodell von Spacy zeigt eine insgesamt schwächere Leistung, was darauf hinweist, dass es weniger geeignet ist für die Anforderungen an domänenspezifische historische Texte. Zusammenfassend verdeutlichen die Abbildungen, dass die Anpassung durch eigenes Training auf spezifischen Daten essenziell ist, um eine hohe Erkennungsleistung zu erzielen. Gleichzeitig zeigt sich, dass die Art der Trainings- und Evaluationsdaten einen signifikanten Einfluss auf die Ergebnisse hat. Das weiter trainierte Flair-Modell bietet eine solide Grundlage für zukünftige Anwendungen und zeigt, dass benutzerdefinierte Trainingsansätze notwendig sind, um optimale Ergebnisse in spezialisierten Szenarien zu erreichen.

3 Zusammenfassung und Ausblick

In diesem Bericht wurden verschiedene Tools und Ansätze zur NER untersucht, evaluiert und miteinander verglichen. Ziel war es, die Erkennung benannter Entitäten in historischen, deutschsprachigen Texten zu optimieren. Die durchgeführten Analysen und Tests haben wertvolle Einblicke in die Leistungsfähigkeit der eingesetzten Modelle geliefert.

3.1 Zusammenfassung

Im Rahmen der Untersuchung wurden drei Ansätze eingestzt und evaluiert: ein Standardmodell von Spacy, ein Standardmodell von Flair und ein weiter trainiertes Flair-Modell, das speziell an domänenspezifische Textdaten angepasst wurde. Die Ergebnisse der Evaluation zeigen, dass das weiter trainierte Flair-Modell die beste Gesamtleistung erzielte, insbesondere in Bezug auf Recall und F1-Score. Dies verdeutlicht die Relevanz einer domänenspezifischen Anpassung für die Verbesserung der NER-Leistung. Die wichtigsten Erkenntnisse lassen sich wie folgt zusammenfassen:

- Das weiter trainierte Flair-Modell erreichte mit einem Recall bis zum 88.0% und einem F1-Score bis zum 75.0% abhängig von den Trainings -und Evaluationsdatenart die beste Erkennungsleistung. Es konnte nahezu alle relevanten Entitäten erkennen und war somit für die spezifischen Anforderungen der Daten bestens geeignet.

- Die Standardmodelle von Spacy und Flair lieferten geringere Ergebnisse. Insbesondere das Spacy-Modell zeigte Schwächen bei der Präzision und Genauigkeit, da es ohne zusätzliche Anpassungen verwendet wurde.

- Die Ergebnisse unterstreichen, dass eine Anpassung des Modells durch Training mit spezifischen Daten entscheidend für die Leistungssteigerung ist.

3.2 Ausblick

Trotz der erzielten Ergebnisse bestehen weitere Möglichkeiten zur Optimierung der NER-Leistung:

- **Erweiterung der Trainingsdaten:** Die Verwendung eines größeren, ausgewogeneren Datensatzes könnte die Modellleistung insbesondere bei unterrepräsentierten Klassen wie `ORG` oder `ANIMAL` verbessern.

- **Anwendung weiterer Modelle:** Die Evaluation zusätzlicher NER-Ansätze, wie beispielsweise (Hugginge Face), könnte weitere Erkenntnisse über die Leistungsfähigkeit moderner Ansätze liefern.

- **Integration in Workflows:** Die Ergebnisse könnten in einer praktischen Anwendung integriert werden, z. B. in der historischen Textanalyse oder bei der digitalen Archivierung.

Zusammenfassend lässt sich feststellen, dass die gezielte Anpassung von NER-Modellen an spezifische Anforderungen eine wesentliche Rolle für den Erfolg spielt. Die Erkenntnisse aus diesem Projekt bieten eine solide Grundlage für weiterführende Forschungen und die Entwicklung verbesserter NER-Methoden.

4 Python-Codes zum Einsatz der Tools

4.1 Trainingsdatenvorverarbeitung

```
import os
import random
from charset_normalizer import detect
from transformers import DataProcessor

class DataProcessor:

    # Initialize file paths for training and base data.
    def __init__(self):
        self.output_file = "train_data/train.txt"
        self.base_data_folder = "../base_data"
        self.train_data_folder = "train_data"

    # Read file content with automatic encoding detection.
    def read(self, file_path):
        try:
            with open(file_path, "rb") as file:
                raw_data = file.read()
                encoding_info = detect(raw_data)
                detected_encoding = encoding_info['encoding']
            content = raw_data.decode(detected_encoding)
            print("File content loaded successfully.")
            return content
        except FileNotFoundError:
            print(f"Error: The file '{file_path}' was not
                found.")
            return ""
        except Exception as e:
            print(f"Error: {e}")
            return ""

    # Convert annotated text to CoNLL format with BIO tagging
    def parse_annotated_text(self, input_text):
        sentences = []
        current_sentence = []
        current_tag = None
        inside_tag = False
        word_buffer = []
        i = 0
        while i < len(input_text):
            char = input_text[i]
            if char == "<":
                closing_index = input_text.find(">", i)
                tag_content = input_text[i + 1: closing_index
```

```
                    ]
                if tag_content.startswith("/"):
                    if word_buffer:
                        word = "".join(word_buffer).strip()
                        if word:
                            if inside_tag:
                                current_sentence.append((word
                                    , f"I-{current_tag}"))
                            else:
                                current_sentence.append((word
                                    , "O"))
                        word_buffer = []
                    inside_tag = False
                    current_tag = None
                else:
                    current_tag = tag_content
                    inside_tag = True
                i = closing_index
            elif char in {" ", "\n", ".", "!", "?"}:
                if word_buffer:
                    word = "".join(word_buffer).strip()
                    if word:
                        tag = f"B-{current_tag}" if not any(w
                            [1].startswith("B-") for w in
                            current_sentence) else f"I-{
                            current_tag}"
                        current_sentence.append((word, tag))
                    word_buffer = []
                if char in {".", "!", "?"}:
                    current_sentence.append((char, "O"))
                    sentences.append(current_sentence)
                    current_sentence = []
            else:
                word_buffer.append(char)
            i += 1
        if word_buffer:
            word = "".join(word_buffer).strip()
            if inside_tag:
                tag = f"B-{current_tag}" if not any(w[1].
                    startswith("B-") for w in current_sentence
                    ) else f"I-{current_tag}"
                current_sentence.append((word, tag))
            else:
                current_sentence.append((word, "O"))
        if current_sentence:
            sentences.append(current_sentence)
        return sentences

    # Write the processed sentences in CoNLL format to an
```

```
        output file.
    def write_to_conll(self, sentences, output_file):
        with open(output_file, "a", encoding="utf-8") as f:
            for sentence in sentences:
                for word, tag in sentence:
                    f.write(f"{word} {tag}\n")
                f.write("\n")

    # Read, parse, and save text data in CoNLL format.
    def process_text_file(self, file_path):
        text = self.read(file_path)
        if text:
            sentences = self.parse_annotated_text(text)
            self.write_to_conll(sentences, self.output_file)
            print(f"CoNLL-formatted data saved to '{self.
                output_file}'.")

    # Split the data into train, test, and dev sets.
    def split_train_file(self, test_ratio=0.20, dev_ratio
        =0.1, seed=42):
        with open(self.output_file, "r", encoding="utf-8") as
            f:
            sentences = f.read().strip().split("\n\n")
        random.seed(seed)
        random.shuffle(sentences)
        test_size = int(len(sentences) * test_ratio)
        dev_size = int(len(sentences) * dev_ratio)
        test_sentences = sentences[:test_size]
        dev_sentences = sentences[test_size:test_size +
            dev_size]
        train_sentences = sentences[test_size + dev_size:]
        with open(f"{self.train_data_folder}/train.txt", "w",
            encoding="utf-8") as f:
            f.write("\n\n".join(train_sentences))
        with open(f"{self.train_data_folder}/test.txt", "w",
            encoding="utf-8") as f:
            f.write("\n\n".join(test_sentences))
        with open(f"{self.train_data_folder}/dev.txt", "w",
            encoding="utf-8") as f:
            f.write("\n\n".join(dev_sentences))
        print("Data file was split")

    # Delete the existing training data file.
    def delete_train_data(self):
        with open(self.output_file, "w", encoding="utf-8") as
            file:
            pass

    # Process all text files from the base data folder.
```

```
125     def run(self):
126         for file_name in os.listdir(self.base_data_folder):
127             file_path = os.path.join(self.base_data_folder,
                    file_name)
128             if os.path.isfile(file_path):
129                 self.process_text_file(file_path)
130
131 # Initialize the DataProcessor class
132 data_processor = DataProcessor()
133
134 # Deletes the content of the training data file if it exists
135 data_processor.delete_train_data()
136
137 # Runs the entire data processing pipeline (reads and
      processes input files)
138 data_processor.run()
139
140 # Splits the processed data into training, test, and
      validation datasets
141 data_processor.split_train_file()
142
143 #------------------------------------------------------------
```

Listing 1: Datenverarbeitungsskript für NER-Modelle

4.2 NER-Modelltraining

```
from flair.data import Sentence, Corpus
from flair.datasets import ColumnCorpus
from flair.embeddings import WordEmbeddings, FlairEmbeddings,
    StackedEmbeddings
from flair.models import SequenceTagger
from flair.trainers import ModelTrainer

class Model:

    # Initialize the model parameters such as data folders
       and number of epochs
    def __init__(self, train_data_folder="train_data",
       test_data_folder="",
                 models_folder="models/flair_model",
                 model_path="models/flair_model/final-model.
                    pt", epochs=50):
        self.train_data_folder = train_data_folder
        self.test_data_folder = test_data_folder
        self.models_folder = models_folder
        self.model_path = model_path
        self.epochs = epochs

    # Train the NER model using Flair embeddings and save it
    def train(self):
        columns = {0: "text", 1: "ner"}
        corpus: Corpus = ColumnCorpus(
            self.train_data_folder, column_format=columns,
            train_file="train.txt", test_file="test.txt",
               dev_file="dev.txt",
        )
        label_type = "ner"
        label_dict = corpus.make_label_dictionary(label_type=
           label_type)

        embedding_types_de = [
            WordEmbeddings("de"),
            FlairEmbeddings("de-forward"),
            FlairEmbeddings("de-backward"),
        ]
        stacked_embeddings = StackedEmbeddings(
           embedding_types_de)

        tagger = SequenceTagger(
            hidden_size=256, embeddings=stacked_embeddings,
            tag_dictionary=label_dict, tag_type=label_type,
               use_crf=True,
        )
```

```
40
41          trainer = ModelTrainer(tagger, corpus)
42          trainer.train(
43              self.models_folder, learning_rate=0.1,
44              mini_batch_size=16, max_epochs=self.epochs,
45              embeddings_storage_mode="gpu",
46          )
47
48      # Evaluate the trained model using test data and display
            performance metrics
49      def evaluate(self):
50          columns = {0: "text", 1: "ner"}
51          corpus = ColumnCorpus(self.train_data_folder,
                column_format=columns, test_file="test.txt")
52          tagger = SequenceTagger.load(self.model_path)
53          result = tagger.evaluate(corpus.test, gold_label_type
                ="ner")
54          print(result.detailed_results)
55
56      # Test the trained model with a given input text and
            print detected entities
57      def test(self, input_text):
58          tagger = SequenceTagger.load(self.model_path)
59          sentence = Sentence(input_text, use_tokenizer=True)
60          tagger.predict(sentence)
61          print(sentence.get_spans("ner"))
62          for entity in sentence.get_spans("ner"):
63              print(f"Text: {entity.text}, Type: {entity.tag},
                    Confidence: {entity.score:.2f}")
64
65  # Model usage
66  model = Model()
67
68  # Train the model
69  model.train()
70
71  #-------------------------------------------------------------
```

Listing 2: NER-Modelltraining mit Flair

4.3 NER-Klassifikation

```
import glob
import json
import os
import re
import spacy
from flair.data import Sentence
from flair.models import SequenceTagger
from charset_normalizer import detect
from flair.nn import Classifier

class NERClassifier:

    # Initialize the classifier with model paths and
       configuration options
    def __init__(self,
                 std_spacy=False,
                 std_flair=False,
                 json_data_folder="json_data",
                 input_data_folder="../input_data",
                 model_path="models/flair_model/",
                 model_type="final-model.pt"):

        self.json_data_folder = json_data_folder
        self.input_data_folder = input_data_folder
        self.std_spacy = False
        self.std_flair = False

        if std_spacy:
            self.output_file = os.path.join(self.
               json_data_folder, "std_spacy_model_output.json
               ")
            self.std_spacy = True

        elif std_flair:
            self.output_file = os.path.join(self.
               json_data_folder, "std_flair_model_output.json
               ")
            self.std_flair = True

        else:
            self.output_file = os.path.join(self.
               json_data_folder, model_type + "_output.json")
            self.model_type = model_type
            self.model_path = model_path + self.model_type
            self.model = SequenceTagger.load(self.model_path)

    # Extract metadata (sender and file ID) from file name
```

```
42      @staticmethod
43      def extract_meta_data(file_path):
44          file_name = os.path.basename(file_path)
45          match = re.match(r"([^\s]+)\s+([^\s]+)", file_name)
46          sender = match.group(1) if match else "unknown"
47          file_id = match.group(2).split(".")[0] if match else
                "unknown"
48
49          return sender.replace("korrigiert", ""), file_id
50
51      # Read all text files from the input folder
52      def read_files(self):
53          txt_files = glob.glob(os.path.join(self.
                input_data_folder, "*.txt"))
54          files_content = {}
55
56          for file_path in txt_files:
57              with open(file_path, "rb") as file:
58                  raw_data = file.read()
59                  encoding_info = detect(raw_data)
60                  detected_encoding = encoding_info['encoding']
61                  try:
62                      content = raw_data.decode(
                            detected_encoding)
63                      files_content[file_path] = content
64                  except Exception as e:
65                      print(f"Error reading file {file_path}: {
                            e}")
66                      continue
67
68          return files_content
69
70      # Classify named entities using the trained Flair model
71      def classify_entities(self, file_path, text):
72          sender, file_id = self.extract_meta_data(file_path)
73
74          sentence = Sentence(text, use_tokenizer=True)
75          self.model.predict(sentence)
76
77          entities = []
78          for entity in sentence.get_spans("ner"):
79              print("Entity:  ", entity)
80              entities.append([
81                  entity.text,
82                  entity.tag,
83                  entity.start_position,
84                  entity.end_position
85              ])
86
```

```
87          return {
88              "id": file_id,
89              "sender": sender,
90              "text": text,
91              "entities": entities
92          }
93
94      # Classify named entities using the standard Spacy model
95      def classify_entities_std_spacy(self, file_path, text):
96          sender, file_id = self.extract_meta_data(file_path)
97
98          nlp = spacy.load("de_core_news_lg")
99          doc = nlp(text)
100         entities = []
101         for ent in doc.ents:
102             label = "EVENT" if ent.label_ == "DATE" else ent.
                    label_
103             entities.append([ent.text, label, ent.start_char,
                     ent.end_char])
104
105         return {
106             "id": file_id,
107             "sender": sender.replace("korrigiert", ""),
108             "text": text,
109             "entities": entities
110         }
111
112     # Classify named entities using the standard Flair model
113     def classify_entities_std_flair(self, file_path, text):
114         sender, file_id = self.extract_meta_data(file_path)
115
116         model = "de-ner"
117         tagger = Classifier.load(model)
118         sentence = Sentence(text)
119         tagger.predict(sentence)
120
121         entities = []
122         for entity in sentence.get_spans('ner'):
123             label = "EVENT" if entity.tag == "DATE" else
                    entity.tag
124             entities.append([entity.text, label, entity.
                    start_position, entity.end_position])
125
126         return {
127             "id": file_id,
128             "sender": sender.replace("korrigiert", ""),
129             "text": text,
130             "entities": entities
131         }
```

```
132
133     # Process all text files and classify entities using the
            chosen model
134     def process_all_files(self):
135         all_results = {"results": []}
136
137         files_content = self.read_files()
138
139         for file_path, text in files_content.items():
140             if self.std_spacy:
141                 result = self.classify_entities_std_spacy(
                        file_path, text)
142             elif self.std_flair:
143                 result = self.classify_entities_std_flair(
                        file_path, text)
144             else:
145                 result = self.classify_entities(file_path,
                        text)
146             all_results["results"].append(result)
147
148         os.makedirs(self.json_data_folder, exist_ok=True)
149         with open(self.output_file, "w", encoding="utf-8") as
                json_file:
150             json.dump(all_results, json_file, indent=2,
                    ensure_ascii=False)
151
152         print(f"All results were saved in {self.output_file}.
                ")
153
154
155 # Run classification with the trained Flair model
156 ner = NERClassifier()
157 ner.process_all_files()
158
159 # Run classification with the best-trained Flair model
160 ner = NERClassifier(model_type="best-model.pt")
161 ner.process_all_files()
162
163 # Run classification using the standard Spacy model
164 ner = NERClassifier(std_spacy=True)
165 ner.process_all_files()
166
167 # Run classification using the standard Flair model
168 ner = NERClassifier(std_flair=True)
169 ner.process_all_files()
170
171 #------------------------------------------------------------
```

Listing 3: NER-Klassifikation mit Flair und Spacy

4.4 NER Evaluation

```
import os
import json
import re
from datetime import datetime
from difflib import SequenceMatcher
from charset_normalizer import detect
from matplotlib import pyplot as plt


class Evaluator:

    # Reads the content of a file with automatic encoding detection.
    def read_file_with_encoding(self, file_path):
        with open(file_path, "rb") as f:
            raw_data = f.read()
            encoding_info = detect(raw_data)
            detected_encoding = encoding_info['encoding']

            if detected_encoding:
                try:
                    return raw_data.decode(detected_encoding)
                except Exception as e:
                    print(f"Error decoding file {file_path}: {e}")
                    return ""
            else:
                print(f"Could not detect encoding for file {file_path}.")
                return ""

    # Extracts sender name and document ID from the file name
    @staticmethod
    def extract_meta_data(file_path):
        file_name = os.path.basename(file_path)
        match = re.match(r"([^\s]+)\s+([^\s]+)", file_name)
        sender = match.group(1) if match else "unknown"
        file_id = match.group(2).split(".")[0] if match else "unknown"
        return sender.replace("korrigiert", ""), file_id

    # Extracts named entities from the given annotated text.
    @staticmethod
    def extract_entities_from_text(text):
        entities = []
        offset = 0
        pattern = r"<(?P<label>[^<>]+)>(?P<text>[^<>]+)</\1>"
```

```
44          for match in re.finditer(pattern, text):
45              label = match.group("label")
46              entity_text = match.group("text")
47              start = match.start("text") - offset
48              end = match.end("text") - offset
49              entities.append([entity_text, label, start, end])
50          return entities
51
52      # Processes files to extract ground truth entities and
            save them in JSON format.
53      def extract_ground_truth(self):
54          output_file = "json_data/ground_truth.json"
55          results = []
56
57          file_paths = []
58          for root, dirs, files in os.walk("evaluate_data"):
59              for file in files:
60                  file_paths.append(os.path.join(root, file))
61
62          for file_path in file_paths:
63              if not os.path.exists(file_path):
64                  print(f"File not found: {file_path}")
65                  continue
66
67              sender, doc_id = self.extract_meta_data(file_path
                    )
68              content = self.read_file_with_encoding(file_path)
69              entities = self.extract_entities_from_text(
                    content)
70
71              results.append({
72                  "id": doc_id,
73                  "sender": sender,
74                  "entities": entities
75              })
76
77          with open(output_file, "w", encoding="utf-8") as
                json_out:
78              json.dump({"results": results}, json_out, indent
                    =4, ensure_ascii=False)
79          print(f"Processed data saved to {output_file}")
80
81      # Loads JSON data from a given file.
82      @staticmethod
83      def load_json(file_path):
84          with open(file_path, "r", encoding="utf-8") as f:
85              return json.load(f)
86
87      # Extracts entity tuples (text, label) from JSON results.
```

```
88      @staticmethod
89      def extract_entities(results):
90          entities = []
91          for result in results:
92              for entity in result["entities"]:
93                  word = entity[0]
94                  label = entity[1]
95                  entities.append((word, label))
96          return set(entities)
97
98      # Checks if two words are similar based on a similarity
            threshold.
99      @staticmethod
100     def is_similar(word1, word2, threshold=0.8):
101         similarity = SequenceMatcher(None, word1, word2).
                ratio()
102         return similarity >= threshold
103
104     # Matches predicted entities with ground truth using
            flexible matching criteria.
105     def match_flexible(self, true_entities,
            predicted_entities, similarity_threshold=1.0):
106         true_positives = set()
107         false_positives = set(predicted_entities)
108         false_negatives = set()
109
110         for true_word, true_label in true_entities:
111             matched = False
112             for pred_word, pred_label in predicted_entities:
113                 if true_label == pred_label and (
114                     true_word in pred_word or pred_word in
                        true_word or
115                     self.is_similar(true_word, pred_word,
                        similarity_threshold)
116                 ):
117                     true_positives.add((pred_word, pred_label
                        ))
118                     matched = True
119                     false_positives.discard((pred_word,
                        pred_label))
120             if not matched:
121                 false_negatives.add((true_word, true_label))
122
123         return {
124             "true_positives": true_positives,
125             "false_positives": false_positives,
126             "false_negatives": false_negatives,
127         }
128
```

```
129     # Calculates precision, recall, F1-score, and accuracy
            for evaluation.
130     def calculate_metrics(self, true_entities,
            predicted_entities):
131         results = self.match_flexible(true_entities,
                predicted_entities)
132         tp = len(results["true_positives"])
133         fp = len(results["false_positives"])
134         fn = len(results["false_negatives"])
135 
136         precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0
137         recall = tp / (tp + fn) if (tp + fn) > 0 else 0.0
138         f1 = 2 * (precision * recall) / (precision + recall)
                if (precision + recall) > 0 else 0.0
139         total = tp + fp + fn
140         accuracy = tp / total if total > 0 else 0.0
141 
142         return precision, recall, f1, accuracy
143 
144     # Displays the evaluation results of different NER models
145     def display_results(self):
146         self.extract_ground_truth()
147 
148         # Extract ground truth entities from annotated text
                files
149         ground_truth_path = "json_data/ground_truth.json"
150         tool_final_path = "json_data/final-model.pt_output.
                json"
151         tool_best_path = "json_data/best-model.pt_output.json
                "
152         tool_std_path = "json_data/std_flair_model_output.
                json"
153         tool_spacy_path = "json_data/std_spacy_model_output.
                json"
154 
155         # Load the ground truth and model outputs from JSON
                files
156         ground_truth = self.load_json(ground_truth_path)["
                results"]
157         tool_final_results = self.load_json(tool_final_path)[
                "results"]
158         tool_best_results = self.load_json(tool_best_path)["
                results"]
159         tool_std_results = self.load_json(tool_std_path)["
                results"]
160         tool_spacy_results = self.load_json(tool_spacy_path)[
                "results"]
161 
162         # Extract entities from ground truth and model
```

```
            results
163         true_entities = self.extract_entities(ground_truth)
164         tool_final_entities = self.extract_entities(
                tool_final_results)
165         tool_best_entities = self.extract_entities(
                tool_best_results)
166         tool_std_entities = self.extract_entities(
                tool_std_results)
167         tool_spacy_entities = self.extract_entities(
                tool_spacy_results)
168
169         # Calculate evaluation metrics (Precision, Recall, F1
                -Score, Accuracy) for each model
170         metrics = {
171             "Trained Final Model": self.calculate_metrics(
                    true_entities, tool_final_entities),
172             "Trained Best Model": self.calculate_metrics(
                    true_entities, tool_best_entities),
173             "Standard Flair Model": self.calculate_metrics(
                    true_entities, tool_std_entities),
174             "Standard Spacy Model": self.calculate_metrics(
                    true_entities, tool_spacy_entities),
175         }
176
177         # Print evaluation results for each model
178         for tool, (precision, recall, f1, accuracy) in
                metrics.items():
179             print(f"{tool} - Precision: {precision:.2f},
                    Recall: {recall:.2f}, F1-score: {f1:.2f},
                    Accuracy: {accuracy:.2f}")
180
181         # plots evaluation results
182         self.plot_metrics(metrics)
183         self.plot_individual_metrics(metrics)
184
185     # Plots evaluation metrics for different models.
186     @staticmethod
187     def plot_metrics(metrics):
188         tools = list(metrics.keys())
189         precision = [metrics[tool][0] * 100 for tool in tools
                ]
190         recall = [metrics[tool][1] * 100 for tool in tools]
191         f1_score = [metrics[tool][2] * 100 for tool in tools]
192         accuracy = [metrics[tool][3] * 100 for tool in tools]
193         x = range(len(tools))
194
195         fig, ax = plt.subplots(figsize=(10, 6))
196
197         ax.scatter(x, precision, label='Precision', marker='o
```

```
            ', color='blue', linewidth=2)
198         ax.scatter(x, recall, label='Recall', marker='*',
                color='green', linewidth=2)
199         ax.scatter(x, f1_score, label='F1-Score', marker='P',
                 color='red', linewidth=2)
200         ax.scatter(x, accuracy, label='Accuracy', marker='X',
                 color='purple', linewidth=2)
201
202         for i, (p, r, f, a) in enumerate(zip(precision,
                recall, f1_score, accuracy)):
203             ax.text(i+0.1, p, f"{p:.1f}%", fontsize=10, ha='
                    center', va='bottom', color='blue')
204             ax.text(i+0.1, r, f"{r:.1f}%", fontsize=10, ha='
                    center', va='bottom', color='green')
205             ax.text(i+0.1, f, f"{f:.1f}%", fontsize=10, ha='
                    center', va='bottom', color='red')
206             ax.text(i+0.1, a, f"{a:.1f}%", fontsize=10, ha='
                    center', va='bottom', color='purple')
207
208         ax.set_xticklabels(labels=tools, rotation=45,
                fontsize=12)
209         ax.set_xticks(x)
210
211
212         ax.set_xlabel("NER Tools", fontsize=14)
213         ax.set_ylabel("Metrics (%)", fontsize=14)
214         ax.set_title("Comparison of NER Tools - Precision,
                Recall, F1-Score, and Accuracy", fontsize=16)
215
216         ax.legend(fontsize=12)
217
218         plt.tight_layout()
219         plt.show()
220
221     # Plots evaluation metrics for different models
            individual
222     @staticmethod
223     def plot_individual_metrics(metrics):
224         tools = list(metrics.keys())
225         num_tools = len(tools)
226
227         fig, axes = plt.subplots(2, 2, figsize=(16, 12),
                sharey=True)
228
229         colors = ['blue', 'green', 'red', 'purple']
230         markers = ['o', '*', 'P', 'X']
231         labels = ['Precision', 'Recall', 'F1-Score', '
                Accuracy']
232
```

```
233         for idx, tool in enumerate(tools):
234             row, col = divmod(idx, 2)  # Bestimme Zeile und
                Spalte (2x2-Layout)
235             ax = axes[row, col]  # Hole den aktuellen
                Achsenbereich
236 
237             precision = metrics[tool][0] * 100
238             recall = metrics[tool][1] * 100
239             f1_score = metrics[tool][2] * 100
240             accuracy = metrics[tool][3] * 100
241 
242             values = [precision, recall, f1_score, accuracy]
243             x = range(len(values))  # Positionen auf der X-
                Achse
244 
245             for j, (value, color, marker) in enumerate(zip(
                values, colors, markers)):
246                 ax.scatter(j, value, color=color, marker=
                    marker, label=labels[j], s=100)
247 
248                 # Werte oberhalb der Punkte anzeigen
249                 ax.text(j, value + 2, f"{value:.1f}%", ha='
                    center', fontsize=10)
250 
251             ax.set_title(f"{tool}", fontsize=14)
252             ax.set_xticklabels(labels, rotation=45, fontsize
                =10)
253             ax.set_xticks(range(len(labels)))
254             ax.set_ylim(0, 110)
255 
256             ax.spines['top'].set_linewidth(2)
257             ax.spines['right'].set_linewidth(2)
258             ax.spines['bottom'].set_linewidth(2)
259             ax.spines['left'].set_linewidth(2)
260 
261         for idx in range(len(tools), 4):  # Wenn weniger als
            4 Tools vorhanden sind
262             row, col = divmod(idx, 2)
263             axes[row, col].axis('off')  # Nicht genutzte
                Achsen deaktivieren
264 
265         fig.text(0.04, 0.5, "Metrics (%)", va='center',
            rotation='vertical', fontsize=14)
266 
267         fig.suptitle("Metrics Comparison per Tool (
            Scatterplots)", fontsize=16)
268 
269         plt.tight_layout(rect=[0.04, 0.04, 1, 0.95])
270         plt.savefig(f"plots/evaluate_result_{datetime.now().
```

```
            strftime("%Y_%m_%d%H_%M_%S")}.png")
271
272         plt.show()
273
274
275
276
277 # Run evaluation
278 evaluator = Evaluator()
279 evaluator.display_results()
280
281 #------------------------------------------------------------
```

Listing 4: NER Evaluation Pipeline

Literatur

[1] Alan Akbik, Duncan Blythe und Roland Vollgraf. „Flair: An Easy-to-Use Framework for State-of-the-Art NLP“. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. 2019, S. 54–59. URL: https://aclanthology.org/N19-4010/.

[2] Papers with Code. *Named Entity Recognition (NER) - Papers with Code*. Zugriff: 07.11.2024. 2024. URL: https://paperswithcode.com/task/named-entity-recognition-ner.

[3] Matthew Honnibal u. a. *spaCy: Industrial-strength Natural Language Processing in Python*. 2020. DOI: 10.5281/zenodo.1212303. URL: https://github.com/explosion/spaCy.

[4] Matthew Honnibal u. a. *spaCy: Industrial-strength Natural Language Processing in Python*. 2020. URL: https://spacy.io/models/de.

[5] Srinivasa Rao Kundeti u. a. „Clinical named entity recognition: Challenges and opportunities“. In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016, S. 1937–1945. DOI: 10.1109/BigData.2016.7840814.

[6] Flair NLP. *How Model Training Works*. Accessed: 2023-12-10. 2023. URL: https://flairnlp.github.io/flair/master/tutorial/tutorial-training/how-model-training-works.html.

[7] Flair NLP. *Tagging entities*. 2023. URL: https://flairnlp.github.io/docs/tutorial-basics/tagging-entities.

[8] Mudasar Ghafoor und Salman Naseer. *Named Entity Recognition (NER) in NLP: Techniques, Tools, Accuracy and Performance*. Zugriff: 16.11.2024. 2022.

[9] *WossiDia*. Zugriff: 16.11.2024. URL: `https://www.wossidia.de/`.