Implementation – Email Classification using Transformers

Tim Terbach

June 2025

Contents

1 Einleitung

Nachdem in der theory.pdf verallgemeinert die theoretischen Grundbausteine dieses Projektes erklärt wurden, beschäftigt sich diese Datei nun mit der konkreten Anwendung dieses Wissens zum Lösen eines praktischen Problems.

1.1 Projekt

In diesem Projekt wollen wir das Postfach eines Versicherungsunternehmens simulieren. Genauer sollen dabei eingehende E-Mails in fünft verschiedene Kategorien eingeordnet werden:

1. Kfz-Schaden
   - Unfall, Blechschaden, Wildunfall
2. Hausrat-Schaden
   - Einbruch, Wasserschaden, Brandschaden
3. Haftpflicht
   - versehentliche Beschädigung fremder Dinge
4. Reiseschaden
   - Gepäckverlust, Stornierung, Krankheit im Urlaub
5. Tierkrankheit
   - OP beim Haustier, Tierarztrechnung

Anschließend können diese E-Mails dann den richtigen Sachbearbeitern zugeordnet werden. Da es sich hierbei um ein Demoprojekt handelt, werden diese zu klassifizierenden E-Mails eine verhältnissmäßig einfache Struktur haben und sich inhaltlich pro Label auf die 2-3 vordefinierte Themen fokussieren.

---

2 Projektstruktur

An dieser Stelle gehen wir kurz auf die allgemeine Projektstruktur ein, sowie die gewählten Architekturen und Technologien.

```
VectorMail/
 api/                 # Spring Boot backend (REST API)
 frontend/            # Vue.js frontend (Vite)
 python-service/      # Python-based classification module (Flask)
 docs/                # Theory and implementation details
 docker-compose.yml   # Launches the Flask engine container
 README.md            # Project overview
```

2.1 Frontend

Das Frontend wird mit Vue.js entwickelt und soll sich durch ein verspieltes, dennoch übersichtliches und intuitives Design auszeichnen.
Beim Styling habe ich mich für den BEM-Syntax entschieden, um ein besseres Verständnis für CSS-Prinzipien zu erlangen und gleichzeitig eine professionelle Benennung meiner CSS Elemente zu garantieren.
Für mehr Informationen betrachten Sie die README.md des frontend-Verzeichnisses.

2.2 Backend

Das Backend wird als REST-API umgesetzt und übernimmt das Klassifizieren der E-Mails mithilfe unseres trainierten LLMs. Es wird in einem Docker-Container mit Flask betrieben, um eine isolierte, skalierbare und leicht wartbare Architektur zu gewährleisten. Python wurde als Programmiersprache für das Training von BERT gewählt, da es weit verbreitet in KI-Anwendungen und Algorithmik ist und ich zudem meine Python Fähigkeiten ausbauen möchte.

2.2.1 RestAPI

Die REST-API dient als Schnittstelle zwischen dem Client und dem Backend. Der erste implementierte Endpunkt ist eine POST-Request, die die Wahrscheinlichkeitend der definierten Labels zurückggibt. Dabei dient das Backend (Spring Boot) als Vermittler zwischen Client und unserem Docker Container, welcher das Python Skript beinhaltet. Dort wird die eigentliche Klassifizierung durchgeführt.

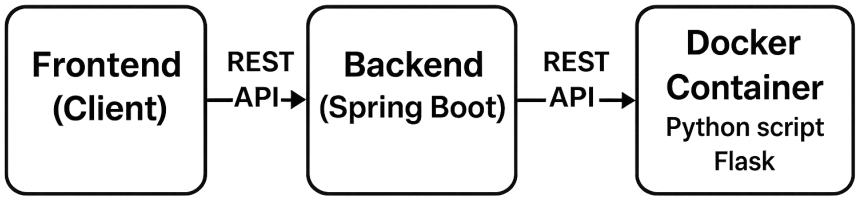


Figure 1: Symbolische Darstellung der Kommunikation

2.2.2 Docker

Um eine modulare und leicht wartbare Architektur sicherzustellen, wird das Python-Programm, wie bereits erwähnt, in einem separaten Docker-Container betrieben.

`/python-service` enthält alle notwendigen Dateien für den Python-Dienst, darunter:

- `requirements.txt` zur Definition der benötigten Python-Abhängigkeiten
- `Dockerfile` zur Containerisierung des Backends
- TODO...

Zunächst habe ich mich auch in diesem Projekt für `python:3.10-slim-buster` als Image entschieden, da es bereits in zuvorigen Projekten Probleme mit dem schlankeren python:3.10-slim gab.

Um dafür zu sorgen, dass sich der Dockercontainer selbstständig aktualisiert, Änderungen also selbstständig übernommen werden, ohne den Container mit `docker-compose up` neu bauen zu müssen, habe ich mich hier dafür entschieden diese Funktion mithilfe von Volumes zu ermöglichen.

Durch die Erweiterung der docker-compose.yml um

```
volumes:
  - ./python-service:/app
```

erkennt der Dockercontainer nun, falls Änderungen innerhalb des “python-service“ Verzeichnis vorgenommen werden und startet den Container automatisch neu.

---

3 Trainingsdaten

Um BERT zu Fine Tunen brauchen wir zunächst Trainingsdaten. Zum Start wollen wir dabei mit etwa 200 E-Mails pro Label beginnnen. Da es keine passende Datenbank gibt, die E-Mails zu diesen Kategorien liefert, habe ich mich an dieser Stelle dazu entschieden diese von einem weiteren, generativen LLM erzeugen zu lassen, in diesem Fall ChatGPT. Bei einer Menge von insgesamt 1000 E-Mails müssen wir dabei jedoch einen anderen Ansatz wählen, als einfach ChatGPT.com zu öffnen und nach 1000 E-Mails zu diesen Labeln zu fragen. Wir nutzen hier die sehr einfache API von ChatGPT um per Python-script diese E-Mails zu generieren. Obwohl dieser Schritt bei der Planung des Projektes zunächst nicht angedacht war, dient er als perfekte Erweiterung des Projektes, da wir auch hier weitere Konzepte aus der theory.pdf aufgreifen.

3.1 E-Mail Generierung mit der ChatGPT API

Betrachten wir zunächst mal das Python-script welches wir zu Generierung verwenden werden, genauer die `generate_email()` Funktion, welche iterativ ausgeführt wird und somit immer wieder neue E-Mails liefert.

```
def generate_email():
    response = client.chat.completions.create(
    model="gpt-4.1",
    messages=[
        {"role": "system", "content": "Du bist ein Generator f r realistische, formelle E-
        {"role": "user", "content": base_prompt}
    ],
    temperature=0.8
)
    text = response.choices[0].message.content
    try:
        return json.loads(text)
    except:
        print("Fehler beim Parsen der Antwort:")
        print(text)
        return None
```

Diese Funktion ist relativ intuitiv und bedarf keiner großartiger Erklärung, interessant sind jedoch vor allem die Paramter

- `temperature=0.8`
- `messages=[ "role": "system", "content": "Du bist ein Generator für realistische, formelle E-Mails an eine Versicherung. Gib nur JSON zurück.", "role": "user", "content": base_prompt ],`

Die **temprature** sagt hierbei aus, wie zufällig das Modell bei der Tokengenerierung vorgehen soll. Bei einer Temprature von 0.0 würde das Modell immer nur den warhscheinlichsten nächsten Token wählen, bei einem gleichbleibenden Prompt würde somit auch der Output bzw. die Sequenz generierter Tokens gleich bleiben, das wollen wir hier natürlich verhindern. Mit einem temprature-Wert von 0.8 sagen wir dem Modell

also, dass aus auch Tokens wählen darf, welche nicht am wahrscheinlichsten von allen Tokens als nächste vorkommen, um eine hohe Variation an verschiedenen E-Mails zu erzeugen.

Ein weiterer interessanter Punkt sind die übergebenen **messages**. Mit `messages` wird jeweils eine Rolle sowie ein Prompt (Content) übergeben. Dabei legen wir mit `role="system"` fest, dass es sich um einen System Prompt handelt. Hier wird dem Modell kurz und knapp gesagt, wie es sich verhalten soll, in unserem Fall legen wir hier vor allem fest, dass das Modell nur mit einer JSON antworten soll, da wir mehr Output hier nicht brauchen, bzw. dieser zusätzliche Output mit mehr Tokens und somit höheren Kosten verbunden wäre.

Anschließend übergeben wir den eigentlicher Nutzer-Prompt, dementsprechend mit `role="user"`. Dieser Prompt bezieht sich vor allem auf ein paar inhaltliche Richtlinien und Beispiele der E-Mails für die verschiedenen Labels. Den kompletten User-Prompt hat dabei ChatGPT selbst auf Basis dieser Richtlinien und Beispiele erstellt, mit dem Ziel ein weiteres LLM dazu anzuweisen, einen für diese Aufgabe optimalen Datensatz zu erzeugen, diesen können Sie in dem folgenden Abschnitt betrachten.

3.1.1 User Prompt

Bitte generiere jeweils genau **eine E-Mail** für jedes Label im folgenden JSON-Format:

```
[
    {
    "label": "<schadenstyp1>",
    "text": "<realistische E-Mail in deutscher Sprache>"
    },
    {
    "label": "<schadenstyp2>",
    "text": "<realistische E-Mail in deutscher Sprache>"
    },
    {
    "label": "<schadenstyp3>",
    "text": "<realistische E-Mail in deutscher Sprache>"
    },
    {
    "label": "<schadenstyp4>",
    "text": "<realistische E-Mail in deutscher Sprache>"
    },
    {
    "label": "<schadenstyp5>",
    "text": "<realistische E-Mail in deutscher Sprache>"
    },
]
```

Regeln:

1. Die E-Mail soll **formell, aber alltagsnah** klingen – wie eine echte Nachricht von Privatpersonen oder Kunden.
2. Der E-Mailtext soll **3–4 Sätze** lang sein, in natürlichem Deutsch.
3. Verwende **keine sensiblen Daten** wie echte Adressen, Telefonnummern oder reale Versicherungsnummern.
4. Die E-Mail soll **thematisch zum jeweiligen Schadenstyp passen**, ohne zu ausschweifend oder künstlich zu wirken.
5. Verwende **normale, höfliche Sprache**, gerne mit kleinen individuellen Variationen im Ausdruck.

6. Gib **nur die Keys `label` und `text`** zurück – keine Metadaten, Überschriften oder Kommentare.

Mögliche Labels (zufällig auswählen):

- `"kfz-schaden"` → z. B. Unfall, Blechschaden, Wildunfall
- `"hausrat-schaden"` → z. B. Einbruch, Wasserschaden, Brandschaden
- `"haftpflichtschaden"` → z. B. versehentliche Beschädigung fremder Dinge
- `"reiseschaden"` → z. B. Gepäckverlust, Stornierung, Krankheit im Urlaub
- `"tierkrankheit"` → z. B. OP beim Haustier, Tierarztrechnung

Thematische Richtlinien für jede Kategorie:

Kfz-Schaden:

- Betreff: Schäden an Autos oder Motorrädern
- Typische Begriffe: Unfall“, Fahrzeug“, Blechschaden“, Wild“, Polizei“

Hausrat-Schaden:

- Betreff: Schäden am Inventar in Wohnung oder Haus
- Typische Begriffe: Einbruch“, Rohrbruch“, Fernseher“, gestohlen“

Haftpflichtschaden:

- Betreff: Absender hat Dritten Schaden zugefügt
- Typische Begriffe: versehentlich“, mein Kind“, beschädigt“, Nachbar“

Reiseschaden:

- Betreff: Probleme im Urlaub, z. B. Verlust, Krankheit
- Typische Begriffe: Koffer“, Flug“, Storno“, Rechnung“, Ausland“

Tierkrankheit:

- Betreff: Tierarztkosten, Behandlungen von Haustieren
- Typische Begriffe: Hund“, Katze“, OP“, Tierarzt“, Rechnung“

3.1.2 Generierter Datensatz

Das Generieren von 2000 E-Mails hat auf diese Art und Weise insgesamt knapp $2.00 gekostet, bei 160000 generierten Tokens. Durch die Formulierung des User-Prompts haben wir nun also 2000 E-Mails in folgender Struktur vorliegen:

```
... ],
 [
{
  "label": "kfz-schaden",
  "text": "ich möchte einen Kfz-Schaden melden, der sich gestern an meinem Auto ereignet ...
},
{
  "label": "hausrat-schaden",
  "text": "leider kam es in meiner Wohnung zu einem Wasserschaden durch einen Rohrbruch in der ...
{
  "label": "haftpflichtschaden",
  "text": "mein Sohn hat beim Spielen versehentlich die Fensterscheibe unseres Nachbarn ...
```

```
  },
  {
    "label": "reiseschaden",
    "text": "bei unserer Rückkehr aus dem Urlaub wurde festgestellt, dass unser Koffer am ...
  },
  {
    "label": "tierkrankheit",
    "text": "unser Hund musste letzte Woche wegen einer akuten Verletzung operiert werden. Die ...
  }
],
[...
```

Es fällt auf, dass sich das Modell sehr stark an die vorgegebenen Begriffe gehalten hat, sodass sehr viele E-Mails eine ähnliche Struktur aufweisen. Es lassen sich jedoch später sehr einfach neue E-Mails generieren, in welchem wir Fokus auf andere Worte und Szenarios legen können um für eine divärsere Trainingsgrundlage zu sorgen und somit Overfitting zu verhindern. Im Rahmen dieses Projektes sollten die generierten 2000 E-Mails jedoch erstmal ausreichen.

---

4 Fine Tuning

Nachdem wir nun also unsere E-Mails unter anderem mit Hilfe von System Prompting generiert haben, wollen wir diese Daten für das Fine Tuning eines BERT Modells nutzen.

4.1 Erste Schritte

Um über Hugging Face auf Modelle zugreifen zu können und dieses trainieren zu können, müssen wir zunächst ein paar Bilbliotheken importieren:

- `from datasets import DatasetDict, Dataset`
  Die `datasets`-Bibliothek bietet die speziellen Datenstrukturen **Dataset** und **DatasetDict**. Diese sind für das effiziente Management und die Vorverarbeitung großer Textdatensätze optimiert, was die Kompatibilität mit den weiteren Hugging Face Tools stark vereinfacht.
- `from transformers import AutoTokenizer, AutoModelForSequenceClassification, TrainingArguments, Trainer, DataCollatorWithPadding`
  - **AutoTokenizer**: Lädt automatisch den korrekten Tokenizer für ein gewähltes Modell, um Rohtext in für das Modell verständliche numerische IDs umzuwandeln.
  - **AutoModelForSequenceClassification**: Initialisiert das passende vortrainierte Modell (wie BERT) mit einem zusätzlichen Klassifikations-Head, der speziell für Aufgaben wie die E-Mail-Klassifikation trainiert wird.
  - **TrainingArguments**: Definiert alle wichtigen Hyperparameter und Konfigurationen für den Trainingsprozess, wie Lernrate, Batch-Größe und Epochen.
  - **Trainer**: Dies ist die zentrale Klasse, die den gesamten Trainings- und Evaluations-Loop handhabt und die Komplexität des Modells, der Daten und der Optimierung abstrahiert.
  - **DataCollatorWithPadding**: Sorgt dafür, dass die Textsequenzen innerhalb eines Batches auf eine einheitliche Länge gebracht werden (Padding), um sie als Tensoren an das Modell übergeben zu können.
- `import evaluate`
  Die `evaluate`-Bibliothek von Hugging Face ermöglicht den einfachen Zugriff auf eine Vielzahl von Standard-Bewertungsmetriken (z.B. Accuracy, F1-Score). Diese sind entscheidend, um die Leistung des trainierten Modells während und nach dem Fine-Tuning zu beurteilen.

- `import numpy as np`
  NumPy ist die grundlegende Python-Bibliothek für numerische Berechnungen und die Arbeit mit Arrays. Sie wird häufig benötigt, um die numerischen Ausgaben des Modells zu verarbeiten (z.B. Logits in Klassifikations-Labels umzuwandeln) oder für allgemeine mathematische Operationen bei der Metrikberechnung.

4.2 Auswahl des BERT Modells

Zunächst müssen wir uns natürlich die Frage stellen, welches BERT Modell wir für unser Projekt überhaupt benutzten. Seit seiner Veröffentlichung 2018 gibt es viele abgewandelte oder optimierte Versionen dieses LLMs, bekannte Beispiele sind dabei:

- **DistilBERT** (HuggingFace): Eine kleinere und schnellere Version von BERT, die durch Destillation trainiert wurde und etwa 95% der Performance des originalen BERT beibehält.
- **RoBERTa** (Meta): Eine robust optimierte Variante von BERT, die durch längeres Training auf mehr Daten und ohne die Next-Sentence Prediction Aufgabe eine verbesserte Leistung erzielt.
- **ALBERT** (Google): Eine "Lite BERT"-Version, die durch Parameter-Sharing und Embedding-Faktorisierung die Modellgröße drastisch reduziert, während die Performance erhalten bleibt.

Neben Variationen, welche Training und Parameteranzahl optimieren, gibt es auch Varianten, welche auf spezifischen Sprachen trainiert wurden, darunter auch `bert-base-german-cased`. Dieses Modell bietet sich für unser Projekt besonders gut an, da es durch seinen deutschen Trainingssatz besonders gut darin ist, den semantischen und kontextuellen Inhalt einer deutschen Textsequenz zu ergreifen und verstehen, genau das, was wir für dieses Projekt benötigen.

4.3 Modell auf Training vorbereiten

Als erstes laden wir unser Modell und fügen den Classification Head hinzu. Mit Hugging Face und unseren eben getätigten imports geht das wie folgt:

```
# pre-trained model path
model_path = "bert-base-german-cased"

# load tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_path)

# add classification head

id2label = {
    0: "Kfz-Schaden",
    1: "Hausrat-Schaden",
    2: "Haftpflichtschaden",
    3: "Reiseschaden",
    4: "Tierkrankheit"
}

label2id = label2id = {
    "Kfz-Schaden": 0,
    "Hausrat-Schaden": 1,
    "Haftpflichtschaden": 2,
    "Reiseschaden": 3,
    "Tierkrankheit": 4
}
```

```
    model = AutoModelForSequenceClassClassification.from_pretrained(
        model_path,
        num_labels = 5,
        id2label = id2label,
        label2id = label2id
    )
```

Nachdem wir das Modell nun also mit einem Classification Head versehen haben, müssen wir festlegen welche Parameter wir während des Trainings anpassen wollen. Wie bereits in der theory.pdf erläutert, werden wir hier nicht nur den Classification Head trainieren, sondern auch die letzten, sogenannten **Pooling Layers** unseres Modells:

```
    # freeze base model parameters
    for name, param in model.base_model.named_parameters():
        param.requires_grad = False

    # unfreeze base model pooling layers
    for name, param in model.base_model.named_parameters():
        if "pooler" in name:
            param.requires_grad = True
```

4.4 Trainingsdaten vorbereiten

Da wir nun das entsprechende Modell geladen haben, müssen wir als nächstes natürlich auch unsere Trainingsdaten aus unserer JSON laden und für das Training vorbereiten.

```
with open("trainingsdata/emails.json", 'r', encoding='utf-8') as f:
    data_from_json = json.load(f)

# Create a Hugging Face Dataset object from the list of email dictionaries.
full_dataset = Dataset.from_list(data_from_json)

# Split the full dataset into training and testing (or validation) sets.
train_test_split = full_dataset.train_test_split(test_size=0.2, seed=42)

# Create a DatasetDict to store the train and validation splits.
dataset_dict = DatasetDict({
    'train': train_test_split['train'],          # Assign the training split
    'validation': train_test_split['test']       # Assign the test split as the validation set
})


# ================== Pre-Procces Data ==============================

# define text preprocessing
def preprocess_function(examples):
    # return tokenized text with truncation
    return tokenizer(examples["text"], truncation=True, max_length=128)

# preprocess all datasets
tokenized_data = dataset_dict.map(preprocess_function, batched=True)

# create data collator
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Schauen wir uns auch hier die wichtigsten Schritte nochmal etwas genauer an.

```
train_test_split = full_dataset.train_test_split(test_size=0.2, seed=42)
```

Hier spalten wir das zuvor erstellte `Dataset full_dataset` in Trainings- und Validierungsdaten auf, um später Aussagen über die Leistung unseres Modells treffen zu können. Mit `test_size = 0.2` wird dabei festgelegt, dass 20% der Daten als solche Validierungsdaten genutzt werden sollen. Der Seed wird festgelegt, um später diese Aufteilung reproduzieren zu können.
Anschließend speichern wir genau diese Aufteilung in einer weiteren HugginFace Datenstrukur, einem `DatasetDict`, die Datenstruktur die später vom `Trainer` erwartet wird.

```
# Create a DatasetDict to store the train and validation splits.
dataset_dict = DatasetDict({
    'train': train_test_split['train'],
    'validation': train_test_split['test']
})
```

```
tokenized_data = dataset_dict.map(preprocess_function, batched=True)
```

In dieser Zeile lassen wir den Tokenizer unsere Trainingsdaten und Token-IDs und andere erforderliche Formate (wie Attention Masks) umwandeln, sodass unser Modell mit diesen numerischen Werten später arbeiten und trainieren kann. Durch die Map Funktion wird hierbei einfach unsere zuvor definierte `preprocess_function` auf allen Trainings- und Validierungsdaten angewendet.
Mit `batches=True` legen wir fest, dass unsere `preprocess_function` nich für jedes Beispiel einzeln, sondern auch für mehrere Beispiele auf einmal aufgerufen wird, um diese somit gleichzeitig zu verarbeiten, was die Geschwindigkeit der Datenvorbereitung erhöht.

```
data_collator = DataCollatorWithPadding(tokenizer=tokenizer))
```

Damit auch später unser Modell beim Training mit Batches, also Gruppen von Trainingsdaten, arbeiten kann, müssen wir noch einen `DataCollatorWithPadding` erstellen.
Das Padding sorgt dabei dafür, dass alle Trainingsdaten auf die selben Länge gebracht werden, das wird für die späteren Berechnungen und die Umwandlung in Tensoren verrausgesetzt.

4.5 Metriken definieren

Bevor wir zum Training übergehen, sollten wir noch unsere Metriken definieren, welche wir zum Auswerten des Trainings nutzen wollen.

TODO

4.6 Training des Modells

Wir haben unser Modell mit einem Classification-Head geladen, die zu trainierenden Paramter festgelegt sowie unsere Trainingsdaten für das Training vorbereitet.
Abschließend müssen wir das Modell also nur noch trainieren, dafür nutzen wir zwei weitere Datenstrukturen von HuggingFace, `TrainingArguments` und `Trainer`.
Die `TrainingsArguments` beschreiben dabei, wie beim Training vorgegangen werden soll und wann unsere Metriken während des Trainings berechnet werden sollen.
Dem `Trainer`, welcher als zustäzliche Abstraktionsschicht sich anschließend um das Training kümmert, werden anschließend diese `TrainingsArguments` sowie andere Paramter übergeben.

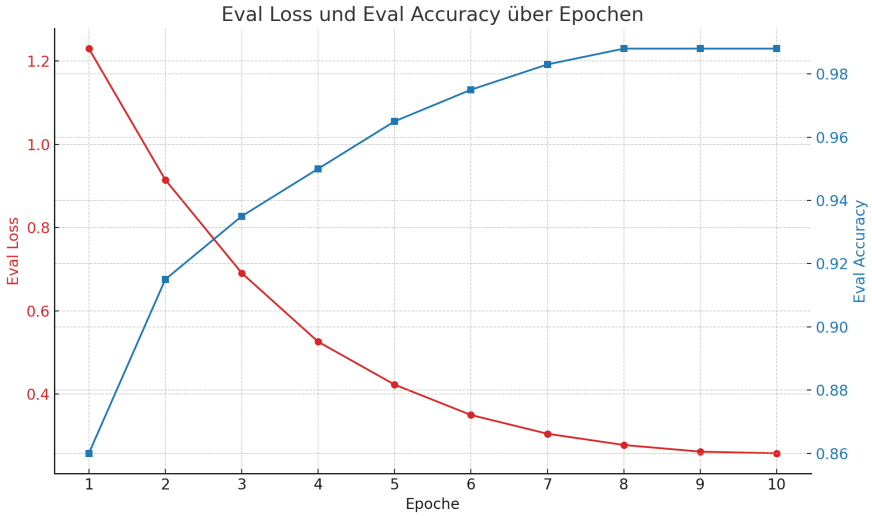
```
# hyperparameters
lr = 2e-5
batch_size = 8
num_epochs = 10

training_args = TrainingArguments(
    output_dir="bert-label-classifier_teacher",
    learning_rate=lr,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=num_epochs,
    logging_strategy="epoch",
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
)


# ================= Define Trainer ================================


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_data["train"],
    eval_dataset=tokenized_data["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

trainer.train()
```

---

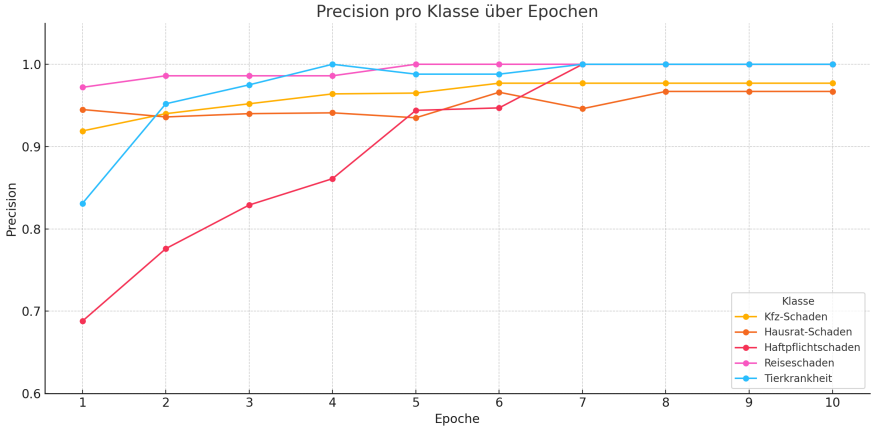
5 Auswertung

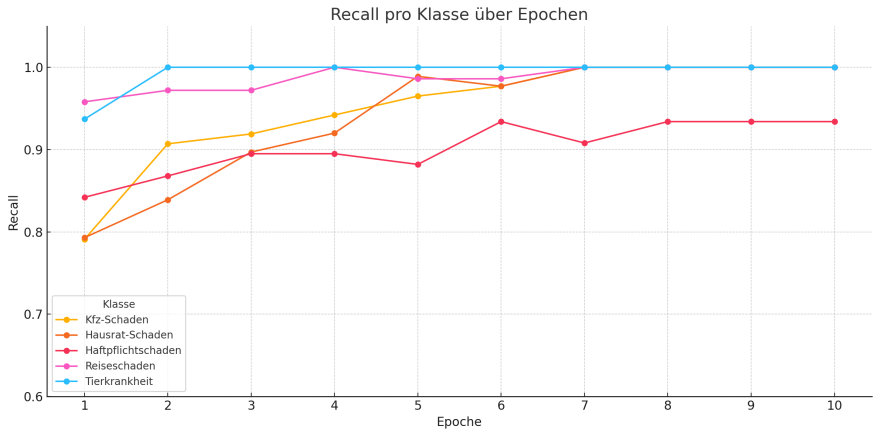
Nach dem Training auf unseren 2000 Trainingsdaten, betrachten wir nun die definierten Metriken um die Qualität und Funktionalität unseres Modells einzuordnen.

5.1 Allgemeine Metriken



5.2 Label-spezifische Metriken





TODO