Fine Tuned LLM for Email Classification

Einleitung

Dieses Projekt hat das Ziel, tiefer in den Bereich der LLMs und NLP einzutauchen. Dabei werde ich erstmals Hugging Face und PyTorch verwenden, um BERT per Fine Tuning auf vordefinierte Labels zu trainieren.
Somit vertiefe ich weiter meine Kenntnisse in Python, grade im Bezug auf top-relevante Bereiche wie Fine Tuning von vor-trainieren LLMs, Text-Klassifizierung und Anwenden von KI-Modellen zum Lösen praktischer Probleme.
Diese Inhalte gehen weit über das im Studium erlernte wissen hinaus, da ich jedoch das Lösen von Problemen mithilfe von KI Anwendungen als die Zukunft des Software Engineering betrachte, gibt es wohl kaum einen besseren Zeitpunkt damit zu beginnen, als jetzt.

Fokus lege ich dabei stehts auf Verständnis und Erklärung, weshalb sich diese Datei vor allem mit den theoretischen Aspekten befasst, ohne dabei zu tief in die Theorie einzutauchen.
Die Balance zwischen theoretischem Wissen und praktischer Anwendung soll innerhalb dieses Projektes stehts erhalten bleiben.
Besonderen Wert lege ich dabei auf die Nachvollziehbarkeit und Transparenz meiner Entscheidungen bei der Implementierung und Entwicklung. Das bedeutet, dass jede Designentscheidung, von der Wahl des Models bis zur Wahl der Labels, klar begründet und dokumentiert wird. Ziel ist es, ein Projekt zu schaffen, das sowohl technisch fundiert als auch für Außenstehende verständlich und zugänglich ist.

---

1. NLP, LLM, MLM?

In diesem Abschnitt wollen wir zunächst ein paar Begriffe einordnen, welche in diesem Projekt häufiger auftauchen werden oder generell im Kontext dies Projektes verstanden werden sollten.

1.1 NLP

Im Kern geht es bei **Natural Language Processing** (NLP) darum, die komplexe und oft mehrdeutige Struktur unserer natürlichen Sprache in eine strukturierte und maschinenlesbare Form zu überführen, mit der Modelle Operationen durchführen können. Diese Aufgaben umfassen häufig:

- **Textklassifizierung:** Das Zuordnen von Texten zu vordefinierten Kategorien (wie in diesem Projekt zur E-Mail-Klassifizierung).
- **Spracherkennung:** Das Umwandeln von gesprochener Sprache in Text.
- **Maschinelle Übersetzung:** Das Übersetzen von Texten von einer Sprache in eine andere.
- **Sentimentanalyse:** Das Erkennen der emotionalen Haltung in einem Text.
- **Informationsextraktion:** Das Extrahieren spezifischer Informationen aus unstrukturiertem Text.

Modelle wie **BERT, ChatGPT, Gemini oder DeepSeek** sind aktuelle Beispiele für die Fortschritte im NLP-Bereich. Sie nutzen fortschrittliche Algorithmen, um die Feinheiten der menschlichen Sprache zu erfassen und für diverse Anwendungen nutzbar zu machen.

Eine große Rolle dabei spielt der Self-Attention-Mechanism und die Transformer Architektur auf welche im verlauf dieses Projektes noch weiter eingegangen wird.

1.2 LLM

Large Language Model (LLM) ist der generelle Überbegriff für genau solche Sprachmodelle die auf einer riesigen Menge an Texten trainiert wurden um NLP zu ermöglichen.
Dabei werden durch verschiedene Trainingsstrategien die Modelle auf unterschiedliche Anwendungsfälle optimiert.
Die Kernidee dabei ist jedoch im Prinzip immer die selbe:
Durch das Vorhersagen von bestimmten Tokens (Siehe Abschnitt *2.1*) lernt das Modell die Semantik und den Kontext ganzer Textsequenzen zu verstehen.
Dieses Verständnis ist die Kernidee auf denen LLMs beruhen.
Die **umfassende Wissensbasis**, die durch das Training auf riesigen Datenmengen entsteht, ermöglicht es LLMs, komplexe Sprachaufgaben zu lösen und sich an vielfältige Anforderungen anzupassen.

1.3 MLM

Masked Language Modeling (MLM) bezieht sich auf eine Trainingsstrategie bestimmter LLMs.
Dabei wird vereinfacht gesagt ein Model auf das "erraten" von Tokens an zufälligen Stellen in den Trainingsdaten trainiert.

Betrachten wir zum Beispiel den Satz *"Die Sonne scheint und der Himmel ist blau"*, so würde das Modell im Training einen der Tokens durch einen *Masked Token* ersetzten, häufig `[MASK]`.

```
Die Sonne scheint und der [MASK] ist blau
```

Das Modell versucht jetzt also während des Trainings diesen Token zu bestimmen.
Liegt es falsch, so werden die Parameter des Neuronalen Netzes so angepasst, dass das nächste mal mit einer erhöhten Wahrscheinlichkeit `Himmel` zurückgegeben wird, so lernt das Modell.
Ein wichtiger Unterschied zu bekannten generativen LLMs wie ChatGPT, Gemini, DeepSeek etc. (die oft auf **Decoder-Architekturen** basieren) ist dabei, dass bei Modellen, die mittels MLM (wie **BERT**, welches eine **Encoder-Architektur** nutzt) trainiert werden, nicht nur der bereits betrachtete Kontext von links genutzt wird, sondern auch der Kontext der Tokens, welche *nach* unserem maskierten Token folgen.
Während ChatGPT also zum Beispiel während des Trainings versucht nur mit `Die Sonne scheint und der` den nächsten Token (`Himmel`) hervorzusagen, nutzen Modelle wie BERT die mit MLM trainiert werden, zusätzlich die Tokens `ist blau`.
Das bietet den Vorteil, dass BERT in der Lage ist, den Kontext der Tokens vor **und** nach unserem maskierten Token zu nutzen.
Deswegen spricht man bei Modellen wie BERT auch von "bidirektionalen Kontext", während Modelle wie ChatGPT nur "unidirektionalen Kontext" nutzen.
Der Fokus liegt bei MLM mehr auf dem **Verständnis einer gesamten Textsequenz**, während bei Modellen wie ChatGPT, der Fokus auf dem **sequentiellen generieren** von Tokens auf Basis bereits gesehener Tokens liegt.

---

2. Tokenizer

In diesem Abschnitt beschäftigen wir uns mit dem **Tokenizer**.
Der Tokenizer ist ein zentrales Konzept jedes LLMs, welches unsere natürliche Sprache in eine solche übersetzt, mit der unser Modell arbeiten kann.

2.1 Tokens

Ein LLM arbeitet mit **Tokens**, diese repräsentieren die Sprache oder das Alphabet unseres LLMs und können dabei ganz unterschiedlich aussehen.
Betrachten wir zum Beispiel den Satz: *"Die Sonne scheint und der Himmel ist blau"*
So könnte hier jedes einzelne Wort ein solcher Token sein:

1. Die
2. Sonne
3. scheint
4. und
5. der
6. Himmel
7. ist
8. blau

Im nächsten Abschnitt sehen wir, dass das nicht immer der Fall sein muss, zu Erklärungszwecken gehen wir hier aber weiterhin davon aus, dass jedes Wort ein einzelner Token interpretiert wird.
Wichtig ist auch, dass, wie wir in der Implementierung noch sehen werden, jeder Token dabei einem Index zugeordnet wird. Gebe ich also als Input *"Die Sonne scheint und der Himmel ist blau"*, so würde der Tokenizer diesen Satz in einen Array von etwa folgender Struktur umwandeln: `[12,122,143,1543,1245,8535,245,3688]`, dabei würde nun jeder dieser Indizes für einen der Tokens stehen:

- `12 -> Die`
- `122 -> Sonne`
  ...

Das minimiert die Kosten der Berechnungen, da mit diesen IDs schneller und effizienter gearbeitet werden kann, als mit Strings.
Auf was diese Indizes konkret verweisen und mit was genau "gearbeitet" wird sehen wir in einem späteren Abschnitt.

Nun stellen sich natürlich einige Fragen, wie genau diese Tokens generiert werden.

- *Wird einfach jedes Wort als Token gewertet?*
- *Wie viele Tokens gibt es?*
- *Wer definiert diese Tokens?*

Mit diesen Fragen beschäftigen wir uns im nächsten Abschnitt.

2.2 Der Tokenizer: Von Text zu Tokens

In diesem Abschnitt beschäftigen wir uns mit dem **Tokenizer**. Er ist ein zentrales Konzept jedes LLMs und übersetzt unsere natürliche Sprache in ein Format, mit dem unser Modell arbeiten kann.

Die generelle Idee des Tokenizers ist es, **Texte möglichst effizient in kleinere Einheiten, sogenannte Tokens oder Subtokens, zu zerlegen.** Ziel ist es, die **Anzahl der benötigten Tokens zu minimieren**, da jeder Token einen bestimmten Ressourcenverbrauch bei der Verarbeitung durch das Modell verursacht. Eine geringere Token-Anzahl pro Text reduziert die Rechenlast und beschleunigt die Verarbeitung.

Nehmen wir an, wir würden jedes Wort als separaten Token werten – das scheint zunächst intuitiv. Daraus folgt aber bereits folgendes Problem: Betrachten wir alleine das Wort "spielen". Wir müssten nun alle Varianten dieses Wortes als einzelnen Token anlegen: "spielen", "spielte", "spielten", "spiele"... und das für alle möglichen Worte in allen möglichen Sprachen.

Effizienter wäre es doch, in diesem Fall "*spiel*" als Token zu speichern, sowie "*en*", "*te*", "*ten*", "*e*"... So könnten wir bereits mit den Subtokens: `[spiel, kauf, en, te, ten, e]` die Worte "spielen", "spielte", "spielten", "spiele", "kaufen", "kaufte", "kauften", "kaufe" bilden, ohne jedes dieser Wörter einzeln als Token speichern zu müssen.

Genau das ist die Idee bei der **Vorbereitung der Trainingsdaten**: Eine solche effiziente Zerlegung in sogenannte **Subtokens** wird vollkommen automatisch erreicht. Dafür existieren es eine Vielzahl an verschiedenen Implementierungen, wie zum Beispiel **Byte Pair Encoding (BPE)**.

Bei Byte Pair Encoding wird dabei wie folgt vorgegangen:

- **Initialisierung:** Nehme die Trainingsdaten. Als Beispiel wieder ein Satz ähnlich zu unserem vorherigen Beispiel: *"Die Sonne scheint wie noch nie."* Wir zerlegen jedes Wort in seine einzelnen Buchstaben und ergänzen einen Sondertoken wie `</w>`, welcher das Ende des Wortes markiert. Dieser Token wird verwendet, um dem Tokenizer mitzuteilen, dass er keine Subtokens über Wortgrenzen hinweg erstellen soll. Die Häufigkeit jedes initialen Worts wird gespeichert, z.B. `[("D", "i", "e", </w>): 1]`.
- **Iterative Zusammenführung:** Berechne für alle möglichen **Subtoken-Paare**, wie oft diese insgesamt vorkommen. Hier würde zum Beispiel auffallen, dass das Paar `("i", "e")` insgesamt 3 Mal vorkommt und somit am häufigsten in unserem Satz. Um Ressourcen zu sparen, könnten wir also das Paar `("i", "e")` zu einem neuen, kombinierten Token `("ie")` zusammenfassen und diesen neuen Token zu unserem Alphabet hinzufügen. Die ursprünglichen Sequenzen werden entsprechend angepasst, z.B. von `("D", "i", "e", </w>)` zu `("D", "ie", </w>)`.
- **Wiederholung:** Dieser Vorgang wird x-Mal wiederholt (wobei x eine vordefinierte Anzahl an Merges oder eine bestimmte Vokabulargröße ist). Dabei werden die häufigsten aufeinanderfolgenden Zeichen- oder Subtoken-Paare zu neuen, größeren Subtokens zusammengefasst.

Die konkrete Implementierung geht natürlich weit über dieses Grundprinzip hinaus, aber die Idee sollte durch dieses kurze Beispiel klar werden: **Minimiere die Anzahl der zu bearbeitenden**

Tokens durch intelligentes Zusammensetzen bereits bekannter Tokens und erweitere das Alphabet um diese.

Im Kontext von Modellen wie **BERT**, die über die **Hugging Face Transformers Bibliothek** geladen werden, nutzen wir häufig Varianten von BPE, wie zum Beispiel den **WordPiece-Tokenizer**. Dieser ist bereits mit dem spezifischen BERT-Modell vortrainiert und stellt sicher, dass unsere Eingabetexte in das vom Modell erwartete Format übersetzt werden.

2.3 Vector Embeddings

Wir wissen nun also was das Ziel eines LLMs ist. Zudem wissen wir, wie diese unsere Sprache in für sie verständliche Tokens umwandeln um mit diesen zu arbeiten.
Die wirklich spannenden Fragen bleiben jedoch offen:

- Wie und was lernen die Modelle?
- Wie genau verstehen sie diese Tokens?
- Wie können sie auf Basis dieser Tokens komplexe Aufgaben ausführen, wie das Generieren von Antworten, Zusammenfassungen oder Textklassifizierungen?

Hier kommen die **Vector Embeddings** ins Spiel.
Jeder Token des Alphabets bekommt ein solches Vector Embedding, diese sind, wie der Name verrät, im Prinzip erstmal einfach nur ein Vektor, also ein Array an float-values.
Der einfach halt halber nehmen wir hier an, dass diese immer zwischen -1 und 1 liegen.

Nehmen wir als Beispiel einen Token aus unserem Beispielsatz: `Sonne`.
Wir haben `Sonne` den Index `122` zugewiesen, zudem weisen wir diesem Index nun einen solchen Vektor zu, der wie folgt aussehen könnte: `[-0.91,0.91,0.43,-0.02,0.76...]`
In der Praxis bestehen diese Vektoren aus mehreren Hundert solcher Werte.
Dieser Vektor repräsentiert dabei das semantische Verständnis des Modells im Bezug auf diesen Token, für was genau diese verschiedenen Werte stehen? Man weiß es nicht.
Das ist ein große Mission aktuell, zu verstehen wie und was genau das Modell erlernt, man kann es sich jedoch ungefähr wie folgt vorstellen:
Jeder dieser Tokens könnte für eine Art Eigenschaft und die Übereinstimmung dieses Tokens mit dieser Eigenschaft stehen. So könnte der erste Wert `-0.91` für "Kälte" stehen (sehr geringe Übereinstimmung), der zweite Wert `0.91` eventuell für "Wärme" (sehr hohe Übereinstimmung). Der dritte Wert steht vielleicht dafür, wie groß etwas ist usw.
Was genau aber in diesen einzelnen Werten für eine Bedeutung steckt, bleibt ein Geheimnis, wir wissen nur, dass es funktioniert.

Diese Embeddings werden, mit anderen Parametern, auf die wir Teilweise später noch eingehen, während des Trainings erlernt und angepasst. Bei jedem gescheiterten Versuch einen Token vorherzusagen oder einen maskierten Token zu erraten, werden diese so angepasst, dass beim nächsten Versuch die Wahrscheinlichkeit für ein richtiges Ergebnis erhöht wird. Somit passt das Modell also Schritt für Schritt seine Parameter selbst an. Wie genau das wiederum passiert ist ein Thema für sich und würde den Rahmen dieses Projektes sprengen.
Wir geben uns an dieser Stelle also zunächst mit der Erkenntnis zufrieden, dass das Modell im Laufe seines Trainings die Vektor Embeddings, sowie andere Parameter so anpasst, dass ein richtiges Hervorsagen von Tokens immer und immer wahrscheinlicher wird.

Wir haben nun nach dem Training also folgendes Mapping:

Token (Sonne) ↦ **Index** (`122`) ↦ Vector Embedding (`[-0.91,0.91,...]`).
...

Können wir nun also einfach die Vector Embeddings von den bekannten Tokens nutzen um einen maskierten Token zu erraten oder neue Tokens zu generieren?
Leider nein, die Vector Embeddings dienen jedoch als Grundlage dafür.
Um zu verstehen wie genau diese Tokens jedoch hervorgesagt werden können, müssen wir noch etwas tiefer in die Theorie eindringen, genauer gesagt zum Herzstück unserer LLMs, dem **Self-Attention-Mechanism**.

2.4 Self-Attention-Mechanism

- Soll nur grob beschrieben werden, so dass verständlich.
- Nicht zu tief in Theorie, angepasst an Projektumfang und implementierung -> Bezug am besten
- Key, Value, Query-Vektoren
- Multi-Headed (grade bezug auf BERT)

---

3. Nutzen vor-trainierter LLMs

Durch das **Fine Tuning** haben wir die Möglichkeit vor-trainiere Modelle auf unsere spezifischen Anwendungsfall anzupassen.
Das ist jedoch nicht die einzige Möglichkeit, eine weitere ist das **System Prompting**.
Was diese beiden Begriffe bedeuten und wann welches dieser beiden Konzepte von Vorteil sind besprechen wir in diesem Abschnitt.

3.1 System Prompting

System Prompting wird vor allem bei generativen LLms wie ChatGPT, DeepSeek usw. verwendet. Dabei wird vereinfacht gesagt, bei jedem Prompt (Input bzw. Anfrage an das LLM) eine Art "System Prompt" oder "Initial Prompt" mit übergeben. In diesem Prompt wird dem Modell gesagt wie es sich verhalten soll, wie der Output aussehen soll und auf welche Aspekte besonderer Fokus gelegt werden soll.
Grundsätzlich wird hier also das Verhalten und der Charakter des LLMs definiert.
Basierend auf den Trainingsdaten und dem Verständnis des Modells im Bezug auf unsere Sprache, weiß das Modell dadurch, wie es den Output anpassen soll um das von ihm verlangte Verhalten mit dem generierten Text in Übereinstimmung zu bringen.

3.1.1 Beispiele: System Prompting

Nehmen wir an, die Nutzerfrage ist stets: "*Wie ist das Wetter heute?*".
Die Antwort des LLMs könnte sich je nach System Prompt drastisch ändern:

- **System Prompt:** "Du bist ein hilfsbereiter Wetterassistent."
  - **Mögliche Antwort:** "Heute wird es sonnig mit Temperaturen um 20 Grad Celsius."

- **System Prompt:** "Du bist ein professioneller Meteorologe. Gib detaillierte und wissenschaftlich fundierte Wetterberichte aus. Nenne Temperatur, Luftfeuchtigkeit und Windgeschwindigkeit."
  - **Mögliche Antwort:** "Für den heutigen Tag wird eine klare Himmelslage erwartet. Die Höchsttemperatur erreicht voraussichtlich 21,5 °C bei einer relativen Luftfeuchtigkeit von 65 %. Der Wind weht aus nordöstlicher Richtung mit durchschnittlich 12 km/h."
- **System Prompt:** "Du bist ein Wetter-Bot, der Antworten in maximal fünf Wörtern gibt."
  - **Mögliche Antwort:** "Sonnig, 20 Grad, leichter Wind."

An dieser Stelle möchte ich einen kleinen Exkurs zum Jailbreaking von solchen generativen Modellen einschieben, da ich mir selbst an dieser Stelle folgende Frage gestellt habe:

"*Kann man dem Modell nicht einfach sagen, dass es die System Prompts umgehen soll, bzw. ignorieren soll?*"

3.1.2 Exkurs: Jailbreaking

Nun, **ja und nein**.
Jealbreaking beschreibt in diesem Kontext das Prinzip, diese System Prompts zu umgehen um das Modell aus seinem "Käfig" an Vorschriften zu befreien, daher der Begriff: **Jailbreaking**.

Betrachten wir folgendes Beispiel:

- Per System Prompt wird einem Modell gesagt: "*Du bist ein professioneller Meteorologe. Gib detaillierte und wissenschaftlich fundierte Wetterberichte aus. Nenne Temperatur, Luftfeuchtigkeit und Windgeschwindigkeit.*"
- Nun befehlt der Nutzer: "*Du bist ein **kein** Meteorologe..." oder "Ignoriere alle vorhergehenden Aufforderungen...".

Durch den zweiten Prompt wird jedoch der erste nicht überschrieben, im Gegenteil, der **Kontext wird einfach um diesen Prompt erweitert**. Das Modell würde nun also versuchen sich wie ein Meteorologe zu verhalten, aber gleichzeitig auch wie **kein** Meteorologe, was zu unvorhersehbaren Ergebnissen führen kann.
Zudem kann das Modell solche Versuche erkennen und direkt abblocken.

In der Vergangenheit genutzte Jailbreak-Prompts sehen deswegen häufig sehr komplex aus, da eine komplett neue Umgebung für das Modell geschaffen wird, in der es unbeschränkt agieren kann.
So zum Beispiel auch ein älterer DAN-Jeailbreak für **GPT-3.5**.

3.2 Fine Tuning

Beim **Fine Tuning** verfolgen wir einen anderen Ansatz, dabei nehmen wir ein bereits trainiertes Modell, in unserem Fall BERT, und trainieren dieses so weiter, dass es auf unseren spezifischen Anwendungsfall angepasst wird.
Die vor trainierten Modelle sind meist auf keinen spezifischen Anwendungsfall trainiert und sollen standartmäßig eine breite Menge an Funktionen bieten. Diese Modelle kann man sich dabei wie eine Art "Rohling" vorstellen, der beim Fine Tuning immer weiter in die gewünschte Form gebracht

wird.
In unserem Fall muss dabei vor allem auf zwei Aspekte geachtet werden:

- Computational Power des Trainings
- Overfitting des Modells durch Trainingsdaten
- Underfitting des Modells durch Trainingsdaten

In der Praxis versucht man immer die "goldene Mitte" zwischen Overfitting und Underfitting zu treffen, um das Modell auch in Real World Cases nutzbar zu machen.
Gehen wir auf diese Aspekte nun noch etwas genauer ein.

3.2.1 Computational Power

Beim Training von LLMs werden wie bereits erwähnt, bei jeder Iteration eine Vielzahl an Parametern angepasst.
BERT besitzt 110 Millionen solcher Parameter, welche standartmäßig beim Training angepasst werden. Diese alle zu trainieren und anzupassen ist nicht nur unfassbar rechenaufwendig, sondern oft auch unnötig. Wir wollen die Kernfähigkeiten unseres Modells weitestgehend unberührt lassen und BERT für unseren spezifischen Anwendungsfall anpassen bzw. nutzbar machen.
Dafür wird oft die **Feature-Extraction** verwendet. Dabei geht es genau um die eben beschriebene Idee, wir wollen das Basismodell (hier BERT) weitestgehend unberührt lassen und nur diese Layer trainieren, welche für unseren Anwendungsfall relevant sind.
In unserem Fall wollen wir also BERTs Fähigkeiten im Bezug auf Textverständnis nutzen, um Textsequenzen bestimmten Labels zuzuordnen.
Das Textverständnis selbst wollen wir dabei nicht weiter trainieren, wir gehen davon aus, dass das Basis Modell in seiner vorliegenden Form darin bereits gut genug ist.
Was wir spezifisch trainieren wollen ist der sogenannte **Classification-Head**, dieser wird, wie der Name schon vermuten lässt, auf das Modell "gesetzt" und erweitert somit das Basismodell. Dieser Head ist nun für die Klassifizierung zuständig, er nutzt also die von BERT bereitgestellte Interpretation der gegebenen Textsequenz und ordnet diese anschließend einem der definierten Label zu.

In unserem Fall werden wir eine kleine Menge der im Basis Modell enthaltenen Layer auch mit trainieren, dabei handelt es sich jedoch nur um die sogenannten **Pooling Layer**.
Diese Pooling Layer befinden sich meist am Ende des Basismodells und sind dafür verantwortlich die hochdimensionale Repräsentation einer Textsequenz als Vektor in eine solche, niedriger dimensionale umzuwandeln, um spätere Berechnungen zu vereinfachen und am Ende eine "einfache" Repräsentation der gesamten Textsequenz zu erhalten, genau diese finale Repräsentation wird anschließend als Input unseres Classification-Heads genutzt.
Durch das Anpassen dieser Pooling Layer und des Classification-Heads optimieren wir, wie das Basismodell die gegebenen Informationen für den Input unseres Heads zusammenfasst und auf unsere Domäne abstimmt, ohne die komplexen inneren Transformer-Layer von Grund auf neu trainieren zu müssen.

BERT ist besonders gut darin ganze Textsequenzen in eine einfache Repräsentation zusammenzufassen, das liegt an einer weiteren Trainingsmethode, der **next sentence prediction**.
Auf diese möchte ich an dieser Stelle nicht allzu sehr eingehen, grundlegend wird das Modell jedoch darauf trainiert, basierend auf einem bestimmten Token `[CLS]`, welcher die gesamte kontextuelle

Repräsentation einer Textsequenz speichert, den nächsten Satz zu erraten. Genau diesen nutzen wir auch als Input für unseren Head.
Durch das Training lernt BERT also möglichst effektiv eine ganze Textsequenz in einen Token zusammenzufassen, genau das was wir benötigen.

3.2.2 Overfitting

Beim **Overfitting** setzt sich das Modell zu stark auf die erlernten Muster aus den Traingsdaten fest. Dabei achtet das Modell immer mehr auch auf Muster, welche unbeabsichtigt in den Trainingsdaten vorkommen.
In unserem Beispiel kann man sich das etwa so vorstellen:
Nehmen wir an, unsere Trainingsdaten sind E-Mails mit jeweils einem zugehörigen Label. Das Modell soll diese E-Mails korrekt klassifizieren.
Machen wir nun den Fehler, dass wir zum Beispiel jede der Trainings-Emails mit Label "Bewerbung", mit dem Satz "Sehr geehrte Damen und Herren, mein Name ist..." beginnen und andere E-Mails nicht, so könnte das Modell lernen, dass jede E-Mail die so beginnt, zum Label "Bewerbungen" gehört, obwohl natürlich eigentlich auf den Inhalt und Kontext der E-Mail selbst eingegangen werden soll.
Genau so könnte sich das Modell also auf bestimmte Wortwahlen, bestimmte Namen oder Satzstrukturen festsetzten um die E-Mails zu klassifizieren, eben einfach aus dem Grund, weil es auf den Trainingsdaten funktioniert.
Overfitting äußert sich meistens darin, dass das Modell auf den Trainingsdaten sehr gut abschneidet, jedoch bei Beispielen, welche nicht aus den Trainingsdaten stammen und auch nicht die selben Muster aufweisen, versagt.
Vereinfacht ausgedrückt, lernt das Modell beim Overfitting spezifische Muster der Trainingsdaten, die sich nicht auf **neue, ungesehene Daten** übertragen lassen. Dadurch verliert es seine Fähigkeit zur **Generalisierung** für reale Anwendungsfälle und wird zu somit "starr".

Generell entsteht Overfitting meist aus den folgenden Gründen:

- Das Modell ist zu komplex (z.B. zu viele Layer, zu viele Parameter für die Datenmenge).
- Zu lange Trainingszeit.
- Zu kleiner Trainingsdatensatz (das Modell hat nicht genug unterschiedliche Beispiele gesehen).
- Fehlerhafte oder verrauschte Daten im Trainingssatz.

Verrauscht, bedeutet hierbei, dass der Trainingssatz eine Menge unnötiger und vor allem für die Klassifizierung irrelevanten Kontext enthalten, auf den sich das Modell fälschlicherweise festsetzten könnte.

3.2.3 Underfitting

Während Overfitting die häufigere Herausforderung beim Fine-Tuning von bereits ausreichen trainierten Modellen ist, da das vortrainierte Modell bereits über ein umfassendes Sprachverständnis verfügt, kann auch **Underfitting** in seltenen Fällen auftreten.
Wenn der spezifische Datensatz, welcher für das Fine-Tuning genutzt wird, zu klein ist oder die Trainingskonfiguration (z.B. eine zu restriktive Lernrate) das Modell daran hindert, kann es die neuen Muster eventuell nicht ausreichend lernen.

Da das in unserem Anwendungsfall aber kein Problem darstellen wird, will ich an dieser Stelle nicht weiter darauf eingehen.

3.3 System Prompting vs Fine Tuning

Nun stellt sich abschließend die Frage, wann man welches der beiden Konzepte anwenden sollte. System Prompting ist in der Regel immer dann gut, wenn man ein Modell vielseitig benutzen will und sich die Anwendungsfälle häufig ändern oder generell sehr dynamisch sind. Hierbei geht es darum, das Modell *temporär* für eine spezifische Aufgabe oder einen Stil zu instruieren, ohne seine Kernkompetenzen dauerhaft zu verändern.

Beispiele:

- Kundenservice-Chatbot
- Social Media Inhaltsgenerierung
- Sprachübersetzung oder Generierung in einem bestimmten Stil

Fine-Tuning wird eher dann verwendet, wenn wie in diesem Projekt, der Anwendungsbereich des Modells sehr statisch sein wird und somit immer und immer wieder die selbe Aufgabe ausgeführt werden soll, diese jedoch mit einer möglichst hohen Korrektheit und Qualität.

Beispiele:

- Textklassifizierung mit vordefinierten Labels
- Sentiment Analyse von Textsequenzen
- Medizinische Textanalyse:

---

4. Ergebnisse

- AUC Score
- Accuracy Score
- Unterschiede, Aussagekraft
- **Konfusionsmatrix:** Eine Konfusionsmatrix ist für Klassifikationsprobleme unerlässlich. Sie visualisiert True Positives, False Positives, True Negatives und False Negatives und gibt viel detailliertere Einblicke als reine Scores.
- **Precision, Recall, F1-Score:** Diese Metriken sind für Multiklassen-Klassifikationen und insbesondere bei unausgeglichenen Datensätzen oft aussagekräftiger als die reine Accuracy. Sie lassen sich auch gut aus der Konfusionsmatrix ableiten.
- **Beispiel-Szenario:** Wenn dein Modell eine E-Mail fälschlicherweise als "Kündigung" klassifiziert (False Positive), obwohl es eine "Schadensmeldung" ist, was sind die Implikationen? Das zeigt kritisches Denken über die Modellergebnisse.