DS 33 – Textgenerierung und Sampling-Strategien

Informatik 10 – MINT-Zweig | Künstliche Intelligenz

Lernziel

Ich benenne unterschiedliche Sampling-Strategien und wende sie zur Auswahl möglicher Folgewörter an; ich beschreibe die Auswirkungen unterschiedlicher n-Gramm-Längen auf den generierten Text.

🕗 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Aus einer bekannten Tabelle das nächste Wort zunächst von Hand auswählen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Greedy, Random und Top-k mit Karten, Würfel oder Zufallszahlen fragend-entwickelnd vergleichen.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Einen kurzen Text gemeinsam erzeugen und anschließend die vorbereitete Simulation ausführen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Den Korpus wechseln und Veränderungen des erzeugten Textes untersuchen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Auswirkungen von Sampling, Trainingsdaten und n-Gramm-Länge erklären lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Top-k- oder Trigramm-Programmierung ausschließlich als optionale Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Auf den Kontext … folgt mit der Häufigkeit/Wahrscheinlichkeit … das Wort …“
- „Der erzeugte Text verändert sich, weil …“
- „Im Vergleich zum n-Gramm-Modell kann das LLM …, jedoch …“

📌 Merksätze

- **Textgenerierung (next word prediction):** Ausgehend von einem Startwort/-kontext wird mithilfe der n-Gramm-Tabelle immer wieder das **nächste Wort** bestimmt und an den Text angehängt.
- **Sampling-Strategien** legen fest, **wie** aus mehreren möglichen Folgewörtern eines ausgewählt wird:
  - **Greedy:** Es wird **immer** das **wahrscheinlichste** Folgewort gewählt → deterministisch, aber oft eintönig/sich wiederholend.
  - **Random:** Es wird **zufällig** unter allen möglichen Folgewörtern gewählt, **gewichtet nach ihrer Wahrscheinlichkeit** → abwechslungsreicher, aber manchmal unsinnig.
  - **Top-k:** Es werden nur die `k` wahrscheinlichsten Folgewörter betrachtet, unter denen dann zufällig (gewichtet) ausgewählt wird → Kompromiss zwischen Vorhersagbarkeit und Abwechslung.

- Die **Länge des n-Gramms** beeinflusst die Textqualität: Größere n (z. B. Trigramme statt Bigramme) erzeugen meist **kohärentere**, aber dafür **weniger variantenreiche** Texte, da der Kontext spezifischer und damit seltener im Korpus vorkommt.

✍️ Von Hand: Die ersten Schritte der Textgenerierung

Bevor ihr die Textgenerierung per Code laufen lasst: Nehmt eure Bigramm-Tabelle aus DS 31/32 (oder die Beispieltabelle mit "der hund läuft..." aus DS 32). Beginnt beim Wort **"der"** und wählt **von Hand** jeweils das **wahrscheinlichste** Folgewort (Greedy-Prinzip), bis ihr 5 Wörter notiert habt. Vergleicht euer von Hand erzeugtes Ergebnis anschließend mit der Greedy-Ausgabe der Code-Zelle unten – sie sollten übereinstimmen!

💡 Beispiel: Textgenerierung mit Greedy- und Random-Sampling

```python
import random

korpus = """
der hund läuft im park der hund spielt mit dem ball
die katze sitzt auf dem baum die katze beobachtet den hund
der hund bellt laut der hund läuft zur katze
die katze springt vom baum die katze rennt weg
der hund läuft ihr hinterher der hund ist müde
"""

def erstelle_bigramm_tabelle(text):
    woerter = text.split()
    tabelle = {}
    for i in range(len(woerter) - 1):
        a, b = woerter[i], woerter[i + 1]
        tabelle.setdefault(a, {})
        tabelle[a][b] = tabelle[a].get(b, 0) + 1
    return tabelle

tabelle = erstelle_bigramm_tabelle(korpus)

def naechstes_wort_greedy(tabelle, wort):
    folgewoerter = tabelle.get(wort)
    if not folgewoerter:
        return None
    return max(folgewoerter, key=folgewoerter.get)

def naechstes_wort_random(tabelle, wort):
    folgewoerter = tabelle.get(wort)
    if not folgewoerter:
        return None
    woerter = list(folgewoerter.keys())
    gewichte = list(folgewoerter.values())
    return random.choices(woerter, weights=gewichte, k=1)[0]

def generiere_text(tabelle, startwort, laenge, strategie):
    text = [startwort]
    aktuelles_wort = startwort
    for _ in range(laenge - 1):
        naechstes = strategie(tabelle, aktuelles_wort)
        if naechstes is None:
            break
        text.append(naechstes)
        aktuelles_wort = naechstes
    return " ".join(text)

print("Greedy:", generiere_text(tabelle, "der", 10, naechstes_wort_greedy))
print("Random:", generiere_text(tabelle, "der", 10, naechstes_wort_random))
print("Random:", generiere_text(tabelle, "der", 10, naechstes_wort_random))
```

```
Greedy: der hund läuft im park der hund läuft im park
Random: der hund spielt mit dem ball die katze die katze
Random: der hund bellt laut der hund läuft zur katze rennt
```

📝 Übung 1 – Beobachtung Greedy vs. Random

Führt die Zelle oben **mehrfach** aus. Was fällt euch beim Vergleich der Greedy- und der Random-Ausgabe auf? Warum liefert Greedy immer denselben Text, Random aber unterschiedliche?

(Eure Beobachtung hier)

🚀 Optionale Vertiefung A – Top-k-Sampling programmieren

Implementiert eine Funktion `naechstes_wort_top_k(tabelle, wort, k=2)`, die nur unter den `k` häufigsten Folgewörtern (gewichtet nach Häufigkeit) zufällig auswählt.

```
# OPTIONAL: naechstes_wort_top_k implementieren

def naechstes_wort_top_k(tabelle, wort, k=2):
    pass

# Test:
# print(generiere_text(tabelle, "der", 10, lambda t, w: naechstes_wort_top_k(t, w, k=2)))
```

🚀 Optionale Vertiefung B – Einfluss der n-Gramm-Länge

Baut analog zu DS 31/32 eine **Trigramm-Tabelle** auf (Kontext = Wortpaar statt Einzelwort) und passt die Textgenerierung entsprechend an. Vergleicht die generierten Texte: Wirken sie mit Trigrammen "sinnvoller" als mit Bigrammen? Warum bricht die Trigramm-Generierung bei kleinem Korpus schneller ab?

📝 Übung 2 – Abhängigkeit von den Trainingsdaten

Ersetzt im Beispielcode oben den `korpus` durch einen **inhaltlich ganz anderen** Text, z. B.:

```
korpus = """
heute scheint die sonne heute regnet es stark
morgen scheint die sonne morgen ist es bewölkt
heute ist es bewölkt heute regnet es leicht
"""
```

Führt die Textgenerierung mit `generiere_text(tabelle, "heute", 8, naechstes_wort_random)` erneut aus.

1. Wie stark unterscheidet sich der erzeugte Text vom Hund/Katze-Beispiel?
2. Was bedeutet das für die Aussage "Ein Sprachmodell erzeugt objektiven, neutralen Text"?
3. **Optional:** Probiert dieselbe Beobachtung mit SoekiaGPT aus, indem ihr dort zwischen zwei unterschiedlichen Dokumentensammlungen wechselt – das Prinzip ist identisch.

Lösung

✅ Lösung Übung 2

Lösung

```
def naechstes_wort_top_k(tabelle, wort, k=2):
    folgewoerter = tabelle.get(wort)
    if not folgewoerter:
        return None
    top_k = dict(sorted(folgewoerter.items(), key=lambda item: item[1], reverse=True)[:k])
    woerter = list(top_k.keys())
    gewichte = list(top_k.values())
    return random.choices(woerter, weights=gewichte, k=1)[0]

print(generiere_text(tabelle, "der", 10, lambda t, w: naechstes_wort_top_k(t, w, k=2)))
```

```
der hund spielt mit dem ball die katze beobachtet den
```