N-Gramm Modell

Pötzsch Felix (580106) Borisov Timofei (580092)

Furitsch Simon (578153)

8. Juli 2024

Zusammenfassung

Dies ist ein Überblick zum Thema N-Gramme im Modul Aktuelle Themen der Informatik im Sommersemester 2024 an der Hochschule für Technik und Wirtschaft Berlin bei Prof. Dr. Tatiana Ermakova. Wir erklären Grundsätzlich was N-Gramme sind, wo sie eingeordnet werden können und wie mit dieser Technik umgegangen werden kann. Dazu zeigen wir einfache Optimierungen wie Smoothing, Back-Off Wording etc.. Des weiteren werden wir ein Beispielprogramm aufführen anhand dessen wir nocheinmal die Verwendung von N-Grammen verdeutlichen.

Inhaltsverzeichnis

1 Einführung (Pötzsch, Felix)

1.1 Sprachmodelle

Sprachmodelle sind der Versuch beispielsweise natürliche Sprache als mathematisches Modell darstellen zu können. Ihnen liegt meist ein stochastischer Prozess zugrunde. Bekannte Modelle sind:

- RNN (Rekurrente Neuronale Netze)[1, vgl.]
  - Abhängigkeiten zwischen Wörtern können erfasst werden und so kontextuell, kohärente Sätze gebildet werden.
  - Verschwindender Gradient. Abhängigkeiten in großer Entfernung können nicht mehr erkannt werden.
- LSTM (Long Short-Term Memory)[1, vgl.]
  - Art von RNN die das Problem des verschwindenden Gradienten durch Speicherzellen lösen.
  - Zelle ermöglicht selektives merken oder vergessen, wodurch langfristige Abhängigkeiten berücksichtigt werden können.
- GRU (Gated recurrent Unit)[1, vgl.]
  - Zielen darauf ab, die Architektur von LSTMs zu vereinfachen und gleichzeitig das Problem des verschwindenden Gradienten anzugehen.
  - Weniger Gating- Mechanismen aber dafür Rechenintensiver.
  - Haben eine vergleichbare Leistung wie LSTM.
- Transformer Modelle:[1, vgl.]
  - GPT (Generative Pre-Trained Transformer)
    - Im Gegensatz zu RNN benutzen Transformer Selbstaufmerksamkeitsmechanismen.
    - Transformatoren verarbeiten Wörter parallel, was zu höherer Effizienz führt.
  - GPT4 (Generative Pre-Trained Transformer 4)
    - Ist im Gegensatz zu GPT3 (1,76 Billionen Parameter) kein monolithisches Sprachmodell mehr sondern eher eine Mischung aus 8 x 220 Milliarden Parametermodellen. [2]

- Umfangreicher Textkorpus.
- Besonders gut für kohärente und kontextrelevante Texte.
- Kann für Vielzahl von Anwendungen verwendet werden, z.B.: Codegenerierung.

Im Voraus sollten bei diesen Modellen sowohl Kenntnisse über Sprache im Allgemeinen bestehen, sowie über die jeweilige Kommunikationssituation. Die oben genannten Modelle bestehen zudem aus mehreren Teilmodellen. Meist sind das Vokabular, Häufigkeiten und Grammatik. Die oben genannten Modelle treten zudem in Kombination auf. Sprachmodelle lassen sich in 4 verschiedene Sorten einteilen:

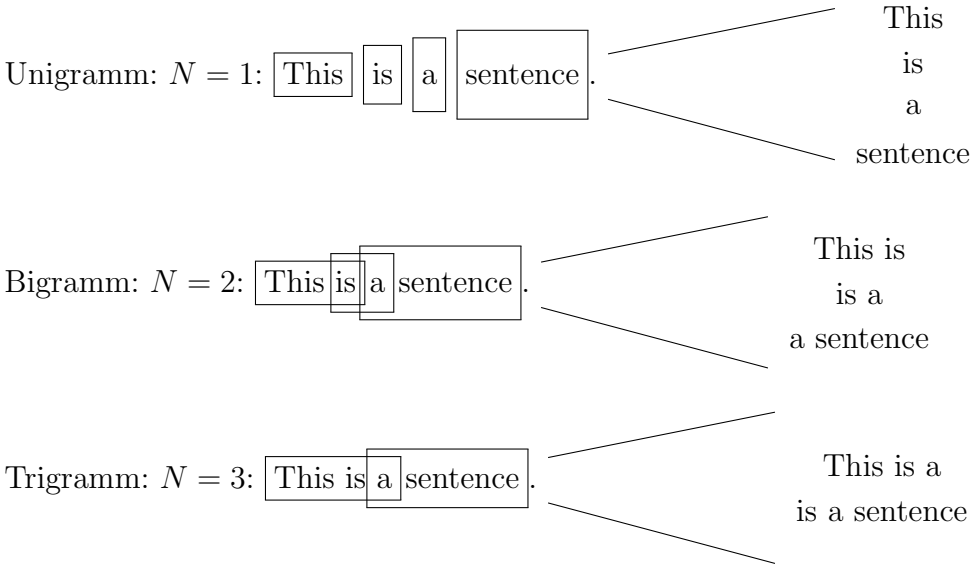
- Regelbasierte Systeme
  - Vordefinierte Sprachregeln, Vorlagen.
  - Fehlende Fähigkeit kontextrelevante dynamische Text zu generieren.
- Vorlagenbasierte Systeme
  - Vordefinierte Vorlagen, die mit dynamischen Inhalten gefüllt werden.
  - Gewisses Maß an Anpassung möglich (Grammatik).
  - Kein einzigartiger kreativer Text.
- Statistische Sprachmodelle
  - N-Gramme, Markov (HMM)
  - Mangel an semantischem Verständnis und Schwierigkeiten kontextrelevante Inhalte zu erzeugen.
- Neuronale Sprachmodelle
  - Dialoge sind möglich und es können sowohl Kontextrelevante als auch Kohärente Teile erkannt werden.
  - Siehe RNN, LSTM, GRU, GPT

1.2 N-Gramme

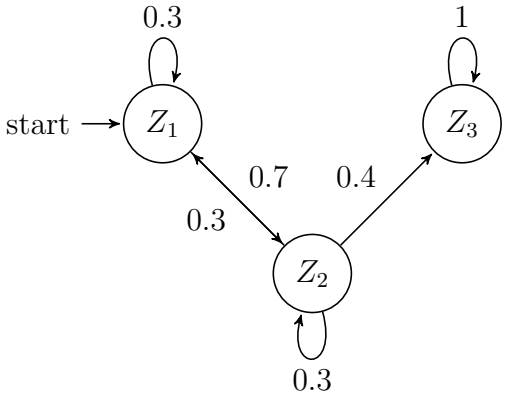
N-Gramme, sind in der Linguistik eine Sequenz von N aufeinanderfolgenden Fragmenten eines Textes. Fragmente können dabei Buchstaben, Silben, Laute oder Wörter sein. Das können von 1-Gramm (Uni-Gramme), über 2-Gramm (Bi-Gramme), 3-Gramm (Tri-Gramme) bis Multigrammen sein. Ihnen zugrunde liegt die Markov-Annahme. Die Markov-Annahme[4] erschließt eine Vorhersage über einen zukünftigen

Zustand nicht aus der Menge aller Zustände, sondern nur der unmittelbaren vorherigen eingetretenen Zustände. Die Zustände können in diesem Falle als Fragmente aufgefasst werden. Die klassischen Anwendungsgebiete für N-Gramme liegen in der Rechtschreibkorrektur, Textvervollständigung/Wörter vorhersage (z.B.: SMS schreiben), Textanalyse (Beurteilung), Spracherkennung (Audioanwendungen) und Handschrifterkennung.

Typische N-Gramme:



Die daraus folgende typische Markov Annahme als Prozessdiagramm und Übergangstabelle würde dazu folgend aussehen:



	Z_1	Z_2	Z_3
Z_1	0.3	0.3	0
Z_2	0.7	0.3	0
Z_3	0	0.4	1
$=$	1	1	1

Grundsätzlich haben wir es mit bedingten Wahrscheinlichkeiten zu tun die wie folgt lauten:

$$\begin{aligned} P(t) = P(w_1, w_2, ...w_n) &= P(w_1) * P(w_2 \mid w_1)...P(w_n \mid w_1, ...w_n - 1) \\ &= \prod_{i=1}^{n} P(w_i \mid w_1...w_i - 1) \end{aligned} \tag{1}$$

$$P(w_1 \mid w_1...w_i - 1) = \frac{P(w_1...w_i)}{P(w_1...w_i - 1)} \tag{2}$$

Also die Wahrscheinlichkeit, dass Wort 1 eintritt unter der Bedingung, dass ein Paar andere Wörter vorher schon in Kombination aufgetreten sind. Demnach wird Text auf Basis von statistischen Mustern generiert oder anders gesagt: Neu angeordnet. Es mangelt dadurch häufig an semantischem Verständnis. Insbesondere im Zusammenhang mit langfristigeren Abhängigkeiten (Kohärenz) sind Sprachmodelle die rein auf N-Grammen basieren sehr ungeeignet, um Texte zu schreiben. Da lediglich auf Basis einer *Maximum-Likelyhood*[4] Methode gearbeitet wird, kann Text nur *nacherzählt* werden aber nicht generiert.
Es folgen nun ein paar typischen Eigenheiten von N-Grammen. Einem Totalversagen entgegenzuwirken, muss zunächst die Wahl eines geeigneten N priorisiert werden. Denn auch wenn ein geeigneter (Text-) Korpus gut gewählt wurde kann es zu *Unseen N-Grams*[3] kommen. Ebenso wie *Unseen N-Grams*[3] gibt es auch *Unseen/Unknown Words*[3]. Prinzipiell aber sollte gesagt werden, dass auch ein großer oder geeigneter Korpus nicht immer zum Erfolg führt, da auch dort Wortkombinationen oder Wörter *Unseen* sein können (Chomsky, Zipfsches Gesetz).[3] Des Weiteren sollte auf den Underflow bei der Bewertung der Wahrscheinlichkeiten geachtet werden. Dieses Problem führt zum nächsten, denn es müssen nicht nur zu niedrige, sondern auch die vollständigen Nullwahrscheinlichkeiten beachtet werden.
Es müssen also vor allem die Informationen in Form der Angepasstheit sowie Auswahl vorab gut gewählt werden sowie die im Folgenden beschrieben Techniken des *Smoothings*[3], also der Re-evaluation von Nullwahrscheinlichkeiten und niedrigen Wahrscheinlichkeiten eigesetzt werden. Dazu zählt ebenfalls die zusätzlich im Folgenden beschriebene Perplexität als Evaluationstechnik, um auf *Back Off Words*[3], also Wörter, auf welche ausgewichen werden kann, zu kommen.

2 Fortgeschrittene Konzepte und Techniken in N-Gramm-Modellen (Borisov, Timofei)

2.1 Was ist un-seen N-Grams?

Da jeder Corpus begrenzt ist, fehlen darin zwangsläufig einige völlig akzeptable Wortfolgen. Das heißt, wir werden viele Fälle von vermeintlichen SZero Probability n-grams"haben, die in Wirklichkeit eine Nicht-Null-Wahrscheinlichkeit haben sollten [5].

Betrachten wir die Wörter, die auf das Bigram aus Daniel Jurafskys Buch basieren. Ein Textkorpus zusammen mit ihrer Anzahl:

- denied the allegations: 5
- denied the speculation: 2
- denied the rumors: 1
- denied the report: 1

Aber nehmen wir an, unser Testsatz enthält Sätze wie:

- denied the offer
- denied the loan

P(offer—denied the) ist 0! P(loan—denied the) ist 0!

Diese Nullen bedeuten, dass wir die Wahrscheinlichkeit anderer Wortkombinationen stark unterschätzt haben, was die richtige Entscheidung der Anwendung, die auf unserem Modell läuft, stark beeinflussen könnte.

Das Problem heißt "Data Sparsity"[6]. Also kurz gesagt, das bedeutet, dass viele Daten in einem Datensatz fehlen oder auf Null gesetzt sind, sodass die meisten Zellen in einer Tabelle leer bleiben. Dies tritt häufig bei spärlichen Matrizen oder hochdimensionalen Datensätzen auf, wo nicht alle Elemente beobachtet oder erfasst werden. Was können wir also mit den Wörtern machen, die wir verwenden und die nicht in den Kontext unserer Trainingsdaten passen?

Die Methode zur Lösung dieses Problems heißt: Smoothing oder Discounting.

2.2 Smoothing Techniques

2.2.1 Laplace-Glättung. Add One (Add X).

Die einfachste Art der Glättung besteht darin, zu allen n-Gramm-Zählungen einen Wert zu addieren, bevor wir sie zu Wahrscheinlichkeiten normalisieren. Alle Zählungen, die vorher Null waren, haben nun den Wert 1, die Zählungen von 1 werden zu 2 usw. Dieser Algorithmus wird als Laplace-Glättung oder Additive Glätterung bezeichnet [7].

Statt

$$\#(w_1, w_2, w_3, \ldots, w_{n-1}, w_n)$$

Wir setzen

$$\#(w_1, w_2, w_3, \ldots, w_{n-1}, w_n) + 1$$

Da das Vokabular aus V-Wörtern besteht und jedes Wort inkrementiert wurde, müssen wir auch den Nenner anpassen, um die zusätzlichen V-Beschachtungen zu berücksichtigen.

$$P_L(w_n|w_1, w_2, w_3, \ldots, w_{n-1}) = \frac{\#(w_1, w_2, w_3, \ldots, w_{n-1}, w_n) + 1}{\#(w_1, w_2, w_3, \ldots, w_{n-1}) + \mid V \mid}$$

[8]

Eine Alternative zur Glättung durch Addition besteht darin, einen etwas geringeren Anteil der Wahrscheinlichkeitsmasse von den gesehenen zu den ungesehenen Ereignissen zu verschieben. Anstatt zu jeder Zählung 1 zu addieren, fügen wir einen Bruchteil der Zählung k hinzu.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + k}{C(w_{n-1}) + kV}$$

[5].

2.2.2 Good-Turing Glättung

Wie berechne ich die Wahrscheinlichkeit von un-seen N-Grams?

Der erste Schritt der Berechnung besteht darin, die Wahrscheinlichkeit abzuschätzen, dass ein zukünftig beobachtetes Fall zu einer bisher nicht gesehenen Art gehört. Diese Schätzung ist:

$$P_{GT}(unseen) = \frac{N_1}{Z}$$

Der nächste Schritt besteht darin, die Wahrscheinlichkeit zu schätzen, dass das nächste beobachtete Fall einer Art angehört, die bereits r-mal beobachtet wurde. Für eine einzelne Art beträgt diese Schätzung:

$$P_{GT}(x) = \frac{r*}{Z}$$

Good-Turing-Smoothing löst dieses Dilemma, indem es die Häufigkeitswerte anpasst. Es wird so getan, als ob Arten, die eigentlich r-mal vorkommen, r^*-mal vorkommen, mit

$$r* = \frac{(r+1)N_{r+1}}{N_r}.$$

[8],[9]

2.2.3 Katz Backoff Glättung

Wenn ein benötigtes n-Gramm in einem Backoff-N-Gramm-Modell keine Vorkommen aufweist, gehen wir zum (n-1)-Gramm über. Dieser Prozess wird fortgesetzt, bis wir eine Sequenz finden, die einige Vorkommen hat.
Damit ein Backoff-Modell eine korrekte Wahrscheinlichkeitsverteilung ergibt, müssen die n-Gramme höherer Ordnung reduziert werden, um etwas Wahrscheinlichkeitsmasse für die n-Gramme niedrigerer Ordnung zu reservieren. Ähnlich wie bei der Add-One-Glättung führt die Verwendung der nicht diskontierten MLE-Wahrscheinlichkeit dazu, dass bei der Ersetzung eines n-Gramms mit einer Wahrscheinlichkeit von Null durch ein n-Gramm niedrigerer Ordnung zusätzliche Wahrscheinlichkeitsmasse hinzugefügt wird. Dies würde dazu führen, dass die Gesamtwahrscheinlichkeit, die das Sprachmodell allen möglichen Zeichenketten zuweist, größer als 1 ist. Neben diesem expliziten Abzinsungsfaktor benötigen wir eine Funktion a, um diese Wahrscheinlichkeitsmasse auf die n-Gramme niedrigerer Ordnung zu verteilen.

Beim Katz-Backoff stützen wir uns auf eine diskontierte Wahrscheinlichkeit P*, wobei wir rekursiv auf die Katz-Wahrscheinlichkeit für das (n-1)-Gramm mit kürzerer Vorgeschichte zurückgehen. Die Wahrscheinlichkeit für ein Backoff n-Gramm PBO wird also wie folgt berechnet:

$$P_{bo}(w_i \mid w_{i-n+1} \cdots w_{i-1})$$

$$= \begin{cases} d_{w_{i-n+1} \cdots w_i} \dfrac{C(w_{i-n+1} \cdots w_{i-1} w_i)}{C(w_{i-n+1} \cdots w_{i-1})} & \text{if } C(w_{i-n+1} \cdots w_i) > k \\ \alpha_{w_{i-n+1} \cdots w_{i-1}} P_{bo}(w_i \mid w_{i-n+2} \cdots w_{i-1}) & \text{otherwise} \end{cases}$$

Zur Berechnung von α ist es sinnvoll, zunächst eine Größe β zu definieren, die die verbleibende Wahrscheinlichkeitsmasse für das $(n-1)$-Gramm darstellt:

$$\beta_{w_{i-n+1} \cdots w_{i-1}} = 1 - \sum_{\{w_i : C(w_{i-n+1} \cdots w_i) > k\}} d_{w_{i-n+1} \cdots w_i} \frac{C(w_{i-n+1} \cdots w_{i-1} w_i)}{C(w_{i-n+1} \cdots w_{i-1})}$$

Anschließend:

$$\alpha_{w_{i-n+1} \cdots w_{i-1}} = \frac{\beta_{w_{i-n+1} \cdots w_{i-1}}}{\sum_{\{w_i : C(w_{i-n+1} \cdots w_i) \leq k\}} P_{bo}(w_i \mid w_{i-n+2} \cdots w_{i-1})}$$

2.3 Vergleich von N-Grammen und neuronalen Netzen.

Im Allgemeinen sind neuronale Netze fortschrittlichere und komplexere Instrumente als N-Gramme. Ihr Vorteil liegt in vielen Aspekten. Zum Bespie: N-Gramme erfassen nur den lokalen Kontext. Ein Trigramm-Modell berücksichtigt nur zwei vorangehende Wörter, um das nächste Wort vorherzusagen. Sie sind nicht in der Lage, weitreichende Abhängigkeiten zu verstehen. Mit steigendem Wert von "n"wächst die Zahl der möglichen n-Gramme exponentiell, was zu hohem Speicherverbrauch und Sparsamkeitsproblemen führt. Probleme mit Wörtern, die mehrere Bedeutungen haben (Polysemie), oder mit verschiedenen Wörtern, die gleich klingen (Homonymie), weil ihnen ein tiefes Verständnis des Kontextes fehlt. N-Grams erfordern manuelles Feature-Engineering, um die relevanten n-Gramme auszuwählen, und führen oft zu hochdimensionalen, spärlichen Vektoren usw. [10] [11]
Neuronale Netze sind mittlerweile kontextbewusst. Insbesondere mit Architekturen wie LSTM (Long Short-Term Memory) oder Transformer-Modellen können neuronale Netze weitreichende Abhängigkeiten und kontextuelle Informationen über ganze Sätze oder sogar Absätze hinweg erfassen. Sie verfügen über ein automatisches Lernen von Merkmalen während des Trainingsprozesses, wodurch sich der Bedarf an umfangreicher manueller Merkmalstechnik verringert usw.Wir müssen jedoch den Vorteil von N-Grammen anerkennen. Bei kurzen Texten sind N-Gramm-Methoden dem neuronalen Netz überlegen. [12]

3 Konkrete Schritte zur Implementierung eines N-Gramm Modells in Pyhton (Furitsch, Simon)

3.1 Datenvorbereitung

Die Daten müssen zuerst in ein für das Training geeignetes Format gebracht werden. Hierzu wird der Text zuerst in Kleinbuchstaben konvertiert und die Sonderzeichen und Zahlen werden entfernt. Textfehler, wie Beispielsweiße doppele Leerzeichen werden gemeinsam mir den **Stoppwörtern** bereinigt

3.1.1 Stoppwörter

Stoppwörter sind Füllwörter, welche keine richtige Bedeutung für die Textvorhersage habe, ein Beispiel hierfür ist das Wort: „aber“

3.1.2 Stemming und Lemming

Eine Möglichkeit das Modell anzupassen ist es die Wörter in eine andere Form zu bringen. Dies kann sich sowohl positiv als auch negativ auf das Modell auswirken, hierzu sei aber der Anwendungsfall von Bedeutung. Insbesondere ist hierbei auf das Evaluieren durch die in Kapitel 4.5 genannte **Perplexität** hingewießen. Diese kann dabei helfen die Auswirkung von Stemmung und Lemming zu ermitteln.
Beim **Stemming** wird das Wort einfach in seinen Wortstamm gesetzt: aus singen wird sing und aus gehen wird geh
Beim **Lemming** hingegen wird das Wort in seine Grundform gebracht. aus Häuser wird Haus und aus ging wird gehen

3.2 Grundlagen

3.2.1 Bibliothek: NLTK

Im Kapitel 4.3 wird für den Code die Bibiothek NLTK verwendet. Natural Language Toolkit ist die populärste Bibliothek in der neurolighuistischen Programmierung. Diese Bibliothek macht unter anderem das Tokenisieren, die Erzeugung von Grams, das importieren verschiedener Textkörper und vieles weiteres zum Einzeiler. Auch in den folgenden Kapiteln werden die Funtionalitäten näher beleuchtet.

3.2.2 Tokenisierung

Bei der Tokenisierung handelt es sich um die Aufteilung einer Zeichenkette in kleinere Teile. Die für n-Gram meist genutzte Tokenisierungsmethode ist die Toke-

nisierung in Wörtern da ja auch die Vorhersahe auf Wörtern basiert. Das bedeutet jedes Wort wird ein Token in der Lösungsmenge. Aus „Das ist ein Satz“, wird „das“,“ist“,“ein“,“satz“. Diese Menge wird dann verwendet, um die n-Grams zu erzeugen
In **NLTK** wird der Worttokenisierer mitgeliefert

```
def tokenize(text):
    return nltk.word_tokenize(text.lower())
```

3.2.3 Grams erzeugen

Nun werden die Tokens verwendet, um n-Gramme zu erzeugen. Die konkrete Zahl des n hängt stark von der Kontextanforderung an das Modell ab. Während niedrige n zum Beispiel Unigram (1-gram) und Biagram (2-gram) nützlich für Wortverbindungen und einen geringen Kontext sind, sind Modelle mit hohen n nützlich für die Schätzung von größeren Sätzen bzw. über einen längeren Kontext hinaus.
In **NLTK** ist dies wieder ganz einfach

```
def getGram(gram, tokens):
    return list(nltk.ngrams(tokens, gram))
```

Dies ist eine Methode welche das n und die Tokens übergeben bekommt und daraus das gewünschte Gram erzeugt. Zum Beispiel n=2 alle Biagramme der Tokens. Für die Ermittlung der in Kapitel 4.2.4 genannten bedingten Wahrscheinlichkeiten ist es jedoch auch notwendig die Wahrscheinlichkeiten und somit auch die Grams der darunter liegenden Dimensionen zu ermitteln. Geht man also von n=3 aus sollten alle Unigramme, Biagramme und Triagramme erzeugt werden.

```
def getGrams(n, tokens):
    grams = [[] for _ in range(n)]
    for i in range(1, n + 1):
        grams[i - 1] = getGram(i, tokens)
    return grams
```

3.2.4 Bedingte Wahrscheinlichkeiten ermitteln

Die Wahrscheinlichkeiten werden nun ermittelt damit das Modell damit die Vorhersage bilden kann.
Dies passiert in zwei Schritten:

```
def buildModel(grams):
    model = defaultdict(lambda: defaultdict(lambda: 0))
    for gram_list in grams:
        for gram in gram_list:
```

```
            prefix = gram[:-1]
            suffix = gram[-1]
            if prefix:  # Keine () -> Wort predictions
                model[prefix][suffix] += 1
    return model
```

Hier wird das Model einfach in einem zweidimensionalen Dictionary gespeichert. Der Schlüssel der ersten Dimension ist der Prefix, also alles bis auf das letzte Wort. Die zweite Dimension ist der Suffix, also das letze Wort. Der Wert dahinter ist die Anzahl wie oft der Suffix auf den Prefix gefolgt ist.

	Das	ist	ein	Beispieltext
Das	0	1	0	0
ist	0	0	1	0
ein	0	0	0	1
Beispieltext	0	0	0	0
(Das ist)	0	0	1	0
(ist ein)	0	0	0	1
(Das ist ein)	0	0	0	1

```
def convertToProbabilities(model):
    for prefix in model:
        total_count = sum(model[prefix].values())

        for suffix in model[prefix]:
            model[prefix][suffix] /= total_count
```

Hier wird nun ermittelt wie viel mögliche Suffixe es für einen Prefix gibt. Anschließend wird das Vorkommen der einzelnen Suffixe geteilt durch die Anzahl aller Suffixe denn das liefert uns die benötigten bedingten Wahrscheinlichkeiten.

3.2.5 Vorhersage

```
def predict(model, prefix):
    if prefix in model:
        possible_words = model[prefix]
        return max(possible_words, key=possible_words.get)
```

Es wird einfach ein gegebener Prefix übergeben, dann wird im Modell geschaut ob ein Suffix dafür existiert, falls dies der Fall ist wird der Suffix mit der höchsten Wahrscheinlichkeit zurück gegeben.

3.3 Code

Der weitere Code also alle Hilfsmethoden und die Programmschleife können Sie unter folgendem Link abrufen
https://github.com/println4debug/n-gram

3.4 Evaluierung eines N-Gramm Modells

Um nun zu ermitteln, wie gut unser Modell arbeitet, können wir einfach schauen wie gut das Modell auf unvorhergesehene Inputs reagiert. Wenn wir aber genauer evaluieren möchten bietet es sich an einen Testsatz zu verwenden und zu schauen wie wahrscheinlich es ist dass das Modell diesen Satz vorhersagen kann. Für dieses Verfahren wollen wir also eine Metrik, die diesen Zustand numerisch zusammenfasst.

3.4.1 Perplexität

Die Perplexität ist das wichtigste Maß, um die Qualität eines n-Gram Modells zu beschreiben. Sie ist definiert durch

$$Perplexity(W) = (\prod_{i=1}^{N} P(w_i|w_1, w_2, ..., w_{i-1}))^{-\frac{1}{N}}$$

Es werden also die Wahrscheinlichkeiten aller Präfix/Suffix Kombinationen des Testsatzes in ein Produkt umgewandelt und dieses wird durch den negativen Durchschnitt poteziert. Doch diese Variante bringt einige Probleme mit sich. Zum einen können ohne die Verwendung von Smoothing Techniken Nullwahrscheinlichkeiten mit in das Produkt einfließen was das Peodukt in jedem Fall 0 macht. Außerdem ist diese Variante unstabil bei sehr vielen sehr kleinen Werten da der Endwert dadurch immer kleiner wird. Deshalb gibt es eine äquivalente Umformung

$$Perplexity(W) = e^{-\frac{1}{N}\sum_{i=1}^{N} lnP(w_i|w_1,w_2,...,w_{i-1})}$$

Es werden alle ermittelten Wahrscheinlichkeiten logarithmiert, anschließend summiert, durch den negativen Wert der Anzahl der Wahrscheinlichkeitselementmenge geteilt. Dieser Wert wird in e-Funktion gegeben um anschließend einen endgültigen Wert zu bringen. Je niedriger dieser ist, umso besser konnte der Testsatz von dem Modell verhergesagt werden.
Das Logarithmieren dient dazu, vor allem sehr kleine Zahlen wieder größer zu skalieren, um mit ihnen besser arbeiten zu können. Da das Logarithmieren eine monotone

Transformation ist, das bedeutet der Wert wird zwar verändert aber sein Verhältnis zu den anderen Werten wird nicht verändert. Durch die Logarithmusgesetzte wissen wir dass wir das Produkt nun in Summen umwandeln können und durch die E-Funtion machen wird das Logarithmieren wieder rückgängig und kehren zurück in den Raum der Wahrscheinlichkeit. Außerdem erlaubt sie uns auch den Exponenten der Produktform in einen Multiplikator umzuwandeln.

```
def perplexity(model, test_data_tokens):
    grams = getGrams(len(test_data_tokens), test_data_tokens)
    accumulator = 0.0

    for gram_list in grams:
        for gram in gram_list:
            prefix = tuple(gram[:-1])
            suffix = gram[-1]
            if len(prefix) > 0:
                if prefix in model and suffix in model[prefix]:
                    propabilitie = model[prefix][suffix]
                    accumulator += math.log(propabilitie)
                else:
                    accumulator += math.log(1e-10)

    perplexity_result = math.exp(accumulator / -len(test_data_tokens
        ))
    print(f"Perplexity: {perplexity_result}")
```

3.5 Zukunft von N-Gramm Modellen

3.5.1 Vergleich zu Transformermodellen

Während n-Gram Modelle eine sehr effiziente und recht triviale Lösung für einfache Worterkennung sind ist es den modernen Modellen in einigen Punkten unterlegen. Zu erwähnen sind insbesondere die Transformer Modelle, welche insbesondere in der Lage sind, Kontexte von großen Texten viel besser modellieren können als n-Grame, da ihr Wahrscheinlichkeitsmodell nicht auf die Kontextlänge N-1 vorangegangenen Wörtern beschränkt ist. Außerdem haben sie damit eine erhöhte Flexibilität und gesteigerte Effizienz. Das sind dann keine einfachen Modelle mehr. Die Datenmenge, die benötigt wird, um solche Modelle zu trainieren ist noch viel größer als die von N-Gramen. Auch wenn bei einem N-Gram Modell schon zwei Wörter genügen würden werden sie dennoch erst mit großen Datenmengen nützlich. Dennoch lassen sich n-Gramme viel einfacher implementieren und sind deshalb für kleinere Projekte eine gute Wahl.

3.6 Ausblick

Deshalb lässt sich auch sagen das vor allem im privaten Sektor und bei kleineren Projekten bei denen zum Beispiel kein langer Wortkontext wichtig ist n-Gramme weiterhin zum Einsatz kommen und auch weiterhin werden neue Ideen zur Verbesserung dieser Modelle kommen, welche das Modell relevant halten. Zum Beispiel bessere Glättungstechniken. Dennoch werden sich vor allem in State of the Art Projekte und große Firmen eher die Tranformermodelle etablieren da sie weitaus besser und effizienter sind. Erwähnenswert währen hierbei vor allem neuste Technologien wie ChatGPT4o.

Literatur

[1] AIML, *Compare the different sequence models (RNN, LSTM, GRU, and Transformers)*, https://aiml.com/compare-the-different-sequence-models-rnn-lstm-gru-and-transformers/

[2] Kai Spriesterbach, *Update: Geheimnis gelüftet! Soviele Parameter hat GPT-4*, https://www.afaik.de/gpt-4-anzahl-parameter/

[3] Katja Markert, *N-gram Modeling*, Institut für Computerlinguistik Uni Heidelberg

[4] Tobias Scheffer und Thomas Vanck, *Statistische Sprachmodelle*, Universtät Potsdam, https://docplayer.org/115275693-Statistische-sprachmodelle.html 2019.

[5] Daniel Jurafsky and James H. Martin, *Speech and Language Processing*, 3rd Edition, Pearson, 2023.

[6] Dremio, "Data Sparsity," https://www.dremio.com/wiki/data-sparsity/, Accessed: June 1, 2024.

[7] David Foster, *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*, O'Reilly Media, 2020.

[8] Prof. Dr. Johannes Maucher https://griesshaber.pages.mi.hdm-stuttgart.de/nlp/04ngram/01ngram.html, Accessed: June 1, 2024.

[9] William A. Gale, "Good–Turing smoothing without tears," *Journal of Quantitative Linguistics*, 1995.

[10] Roshmita Dey , Accessed: June 1,*Understanding Language Modeling: From N-grams to Transformer-based Neural Models* 2024.

[11] Alexander Clark, Gianluca Giorgolo, and Shalom Lappin, *Statistical Representation of Grammaticality Judgements: the Limits of N-Gram Models*, Department of Philosophy, King's College London

[12] A. Suresh Babu, Kumar P.N.V.S.Pavan *Comparing Neural Network Approach with N-Gram Approach for Text Categorization*, January 2010International Journal on Computer Science and Engineering 2(1)