Natural Language Processing Assignment 1

Tokenization and N-Gram Language Modeling: Implementation and Analysis

Author: Venkata Siva Sai Krishna Pokuri (2025202024)

Date: February 11, 2026

Contents

1 Tokenization **1**
1.1 Corpus Preprocessing 1
1.2 Tokenizer Implementations 1
1.3 Tokenization Analysis: Sensible and Problematic Examples 1
1.3.1 English Tokenization Analysis 1
1.3.2 Mongolian Tokenization Analysis 3

2 Language Modeling **5**
2.1 Implementation Details 5
2.2 Smoothing Methods 5
2.3 Perplexity Results on Test Set 6
2.4 Analysis of Training Data Size 6
2.5 Autocomplete Analysis 7
2.5.1 Correct Behaviors 7
2.5.2 Incorrect Behaviors and Failure Modes 8

3 Assumptions and Implementation Details **8**
3.1 Corpus and Preprocessing 8
3.2 BPE Tokenization 8
3.3 Language Modeling 9
3.4 Autocomplete 9

4 Conclusion **9**

List of Tables

1 Validation vs Test Perplexity for All 9 Model Configurations 6
2 Test Perplexity Comparison by Training Size (Kneser-Ney Smoothing) . 7

1 Tokenization

1.1 Corpus Preprocessing

Both English and Mongolian CC-100 corpora (1,000,000 lines each) were preprocessed with the following cleaning operations:

- Removal of null bytes and control characters (except newlines/tabs)
- Stripping of zero-width Unicode characters (U+200B to U+206F)
- Normalization of excessive whitespace
- Limitation of consecutive newlines to maximum of 2
- Removal of leading/trailing spaces per line

Data Partitioning: Each corpus was split using line-level random shuffling (seed=42) into:

- **Training**: 80% (800,000 lines)
- **Validation**: 10% (100,000 lines)
- **Testing**: 10% (100,000 lines)

Line-level splitting ensures sentences are not fragmented across partitions.

1.2 Tokenizer Implementations

Three tokenizers were implemented from scratch for each language:

1. **Whitespace Tokenizer**: Splits on whitespace and separates alphanumeric from non-alphanumeric characters using regex pattern `r'\w+|[^\w\s]'`.
2. **Regex Tokenizer**:
   - English pattern: `r"\w+(?:'\w+)?|[^\w\s]"` (preserves contractions)
   - Mongolian pattern: `r"[\u0400-\u04FF\w]+|[^\w\s]"` (handles Cyrillic U+0400 to U+04FF)
3. **BPE Tokenizer**: Character-level initialization with iterative merging of most frequent pairs.
   - Hyperparameters: `vocab_size=5000`, `min_frequency=3`, end-of-word marker `</w>`.

1.3 Tokenization Analysis: Sensible and Problematic Examples

This section presents a comprehensive analysis of all tokenizers with 3 sensible and 3 problematic examples each for both English and Mongolian.

1.3.1 English Tokenization Analysis

1. Whitespace Tokenizer

- **Sensible Examples**:
  1. Input: "The quick brown fox jumps over the lazy dog."
     Tokens: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', '.'] (10 tokens)
     Why Sensible: Perfect word-level separation. Ideal for simple sentences.
  2. Input: "I can't believe it's already 2024!"
     Tokens: ['I', 'can', "'", 't', 'believe', 'it', "'", 's', 'already', '2024', '!'] (11 tokens)
     Why Sensible: Consistent fragmentation of contractions ('can', "'", 't') allows the model to learn the pattern, effectively reducing vocabulary size by reusing common characters
  3. Input: "Dr. Smith works at the hospital."
     Tokens: ['Dr', '.', 'Smith', 'works', 'at', 'the', 'hospital', '.'] (8 tokens)
     Why Sensible: Abbreviations like 'Dr.' are split, but "Dr" + "." is a learnable sequence.

- **Problematic Examples**:

  1. Input: "don't can't won't shouldn't"
     Tokens: ['don', "'", 't', 'can', "'", 't', 'won', "'", 't', 'shouldn', "'", 't'] (12 tokens)
     Why Problematic: All contractions are fragmented identically. "won't" becomes ['won', "'", 't'], losing distinct semantics from "will not".
  2. Input: "http://www.example.com/path?query=value"
     Tokens: ['http', ':', '/', '/', 'www', '.', 'example', '.', 'com', '/', 'path', '?', 'query', '=', 'value'] (15 tokens)
     Why Problematic: URL is completely destroyed into characters and common words. The model cannot recognize this as a single entity.
  3. Input: "Price: $19.99 (20% off!!!)"
     Tokens: ['Price', ':', '$', '19', '.', '99', '(', '20', '%', 'off', '!', '!', '!', ')'] (14 tokens)
     Why Problematic: Currency and percentages are split. $19.99 becomes four tokens, losing numeric value.

2. Regex Tokenizer

- **Sensible Examples**:

  1. Input: "The quick brown fox jumps over the lazy dog."
     Tokens: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', '.'] (10 tokens)
     Why Sensible: Confirms the regex correctly handles the "base case" of clean text, ensuring valid words aren't accidentally split.
  2. Input: "I can't believe it's already 2024!"
     Tokens: ['I', "can't", 'believe', "it's", 'already', '2024', '!'] (7 tokens)
     Why Sensible: Major improvement! "can't" and "it's" are single tokens, preserving semantic meaning.
  3. Input: "Dr. Smith works at the hospital."
     Tokens: ['Dr', '.', 'Smith', 'works', 'at', 'the', 'hospital', '.'] (8 tokens)
     Why Sensible: Correctly isolates punctuation, allowing the model to learn that a period can follow an abbreviation without ending the sentence.

- **Problematic Examples**:

  1. Input: "don't can't won't shouldn't"
     Tokens: ["don't", "can't", "won't", "shouldn't"] (4 tokens)
     Why Problematic: While sensible for tokenization, this increases sparsity. "won't" is now a specific vocabulary item that needs its own training examples.
  2. Input: "http://www.example.com/path?query=value"
     Tokens: ['http', ':', '/', '/', 'www', '.', 'example', '.', 'com', '/', 'path', '?', 'query', '=', 'value'] (15 tokens)
     Why Problematic:Fails to recognize the URL as a structural entity, breaking it into meaningless components just like whitespace tokenization.
  3. Input: "Price: $19.99 (20% off!!!)"
     Tokens: ['Price', ':', '$', '19', '.', '99', '(', '20', '%', 'off', '!', '!', '!', ')'] (14 tokens)
     Why Problematic: Destroys numeric magnitude by splitting on the decimal point, making "19.99" appear as two separate integers.

3. BPE Tokenizer

- **Sensible Examples**:

  1. Input: "The quick brown fox jumps over the lazy dog."
     Tokens: ['The</w>', 'quick</w>', 'br', 'own</w>', ... 'do', 'g.</w>'] (16 tokens)
     Why Sensible: Subwords capture frequent patterns ('br', 'own'). Common words ('The') are single tokens.

2. Input: "I can't believe it's already 2024!"
   Tokens: ['I</w>', "can't</w>", 'believe</w>', "it's</w>", 'already</w>', '20', '24', '!</w>'] (8 tokens)
   Why Sensible: "can't" and "it's" are frequent enough to be single tokens. '2024' is split into '20' '24', allowing numeric generalization.
3. Input: "Dr. Smith works at the hospital."
   Tokens: ['Dr.</w>', ... 'hosp', 'it', 'al.</w>'] (10 tokens)
   Why Sensible: 'hospital' becomes 'hosp' 'it' 'al', showing morphological decomposition.

- **Problematic Examples**:

1. Input: "don't can't won't shouldn't"
   Tokens: ["don't</w>", "can't</w>", "won't</w>", 'sh', "ouldn't</w>"] (5 tokens)
   Why Problematic: Inconsistent! "don't" is one token, but "shouldn't" is split ('sh' + "ouldn't"). This inconsistency confuses the model.
2. Input: "http://www.example.com/path?query=value"
   Tokens: ['http://www.', 'example', '.com/', 'pa', 'th', '?', 'qu', 'er', 'y', '=', 'value</w>'] (11 tokens)
   Why Problematic: While it captures common web prefixes ('http://www.'), the rest is fragmented into arbitrary sub words that don't reflect the URL structure.
3. Input: "Price: $19.99 (20% off!!!)"
   Tokens: ['Pr', 'ic', 'e:</w>', '$', '19', '.', '9', '9</w>', '(', '20', '%</w>', 'off', '!!', '!)</w>'] (14 tokens)
   Why Problematic: Over-fragments rare sequences like prices, splitting digits ('9', '9') and merging punctuation ('!)'), making arithmetic reasoning impossible.

1.3.2 Mongolian Tokenization Analysis

Mongolian presents unique challenges due to Cyrillic script and agglutinative morphology.

1. Whitespace Tokenizer (Mongolian)
Sensible Examples:

1. Input: Монгол улсын нийслэл Улаанбаатар хот юм.
   Tokens: ['Монгол', 'улсын', 'нийслэл', 'Улаанбаатар', 'хот', 'юм.']
   Count: 6 tokens
   Why Sensible: Clean word-level separation. "Улаанбаатар" (Ulaanbaatar) is kept whole as a semantic unit.

2. Input: Би өнөөдөр сургуульд явна.
   Tokens: ['Би', 'өнөөдөр', 'сургуульд', 'явна.']
   Count: 4 tokens
   Why Sensible: Suffixes like "-д" (dative) stay attached ("сургуульд"), preserving case information which is crucial for Mongolian grammar.

3. Input: Сайн байна уу?
   Tokens: ['Сайн', 'байна', 'уу?']
   Count: 3 tokens
   Why Sensible: Standard greeting handled correctly. Punctuation attached to the last word is a minor issue but the semantic units are preserved.

Problematic Examples:

1. Input: Утас: +976-11-123456 (24/7)
   Tokens: ['Утас', ':', '+', '976', '-', '11', '-', '123456', '(', '24', '/', '7', ')']
   Count: 13 tokens
   Why Problematic: Phone number destroyed into individual components. The model cannot learn full number patterns or recognize the country code +976 as a unit.

2. Input: www.монгол.mn эсвэл http://example.mn
   Tokens: ['www', '.', 'монгол', '.', 'mn', 'эсвэл', 'http', ':', '/', '/', 'example', '.', 'mn']

Count: 13 tokens
Why Problematic: Mixed script URL completely fragmented. Domain parts are split, destroying the web address semantics.

3. Input: Үнэ: 50,000 (10%-ийн хөнгөлөлттэй!!!)
Tokens: ['Үнэ', ':', '50', ',', '000', '', '(', '10', '%', '-', 'ийн', 'хөнгөлөлттэй', '!', '!', '!', ')']
Count: 16 tokens
Why Problematic: Price and percentage completely fragmented. The comma in 50,000 splits the number, and the currency symbol is isolated.

2. Regex Tokenizer (Mongolian)
Sensible Examples:

1. Input: Би өнөөдөр сургуульд явна.
Tokens: ['Би', 'өнөөдөр', 'сургуульд', 'явна.']
Count: 4 tokens
Why Sensible: Identical to Whitespace performance on clean text. Accurately captures word boundaries.

2. Input: Монгол Улс
Tokens: ['Монгол', 'Улс']
Count: 2 tokens
Why Sensible: Proper nouns are preserved as single tokens.

3. Input: 1990 он
Tokens: ['1990', 'он']
Count: 2 tokens
Why Sensible: Simple year and word separated correctly.

Problematic Examples:

1. Input: Утас: +976-11-123456 (24/7)
Tokens: ['Утас', ':', '+', '976', '-', '11', '-', '123456', '(', '24', '/', '7', ')']
Count: 13 tokens
Why Problematic: Identical to Whitespace; regex pattern lacks phone number matching, treating punctuation as splitters.

2. Input: www.монгол.mn эсвэл http://example.mn
Tokens: ['www', '.', 'монгол', '.', 'mn', 'эсвэл', 'http', ':', '/', '/', 'example', '.', 'mn']
Count: 13 tokens
Why Problematic: Identical to Whitespace; regex pattern lacks URL matching.

3. Input: Үнэ: 50,000 (10%-ийн хөнгөлөлттэй!!!)
Tokens: ['Үнэ', ':', '50', ',', '000', '', '(', '10', '%', '-', 'ийн', 'хөнгөлөлттэй', '!', '!', '!', ')']
Count: 16 tokens
Why Problematic: Identical to Whitespace; regex pattern lacks currency/numeric matching.

3. BPE Tokenizer (Mongolian)
Sensible Examples:

1. Input: Монгол улсын нийслэл Улаанбаатар хот юм.
Tokens: ['Монгол</w>', 'улсын</w>', 'нийслэл</w>', 'Улаанбаатар</w>', 'хот</w>', 'юм.</w>']
Count: 6 tokens
Why Sensible: Excellent frequent word handling. "Улаанбаатар" is a single token, which is much more efficient than character-level splitting.

2. Input: Манай гэр бүлд таван хүн байна.
Tokens: ['Манай</w>', 'гэр</w>', 'бүл', 'д</w>', 'таван</w>', 'хүн</w>', 'байна.</w>']
Count: 7 tokens
Why Sensible: "бүлд" becomes "бүл" + "д</w>". Correctly identifies the stem "family" and dative suffix "to". Best morphological handling.

3. Input: ээж аав
   Tokens: ['ээж</w>', 'аав</w>']
   Count: 2 tokens
   Why Sensible: Common family terms are single tokens.

Problematic Examples:

1. Input: Утас: +976-11-123456 (24/7)
   Tokens: ['Утас', ':</w>', '+', '9', '7', '6', '-1', '1', '-1', '23', '4', '5', '6</w>', '(', '24', '/', '7', ')</w>']
   Count: 18 tokens
   Why Problematic: Severe fragmentation. Digits split arbitrarily (e.g., '9', '7', '6'), making numeric learning impossible. BPE fails on unseen numeric sequences.

2. Input: www.монгол.mn эсвэл http://example.mn
   Tokens: ['www.', 'монгол', '.mn</w>', 'эсвэл</w>', 'http', '://', 'e', 'x', 'am', 'p', 'le', '.mn</w>']
   Count: 12 tokens
   Why Problematic: Latin URL characters ('example') fragmented into single letters because Latin is rare in Mongolian corpus.

3. Input: Үнэ: 50,000 (10%-ийн хөнгөлөлттэй!!!)
   Tokens: ['Үнэ', ':</w>', '50', ',', '000', '</w>', '(', '10', '%', '-ийн</w>', 'хөнгөл', 'өл', 'т', 'тэй', '!!', '!', ')</w>']
   Count: 17 tokens
   Why Problematic: Numeric and suffix fragmentation. Suffix '-ийн' (genitive) handled well, but 'хөнгөлөлттэй' broken into 4 pieces.

2 Language Modeling

2.1 Implementation Details

A 4-gram language model was implemented from scratch with:

- $n = 4$ (context window of 3 previous tokens)
- Special tokens: `<s>` (start), `</s>` (end), `<unk>` (unknown)
- Probability estimation: Maximum Likelihood Estimation (MLE) with backoff.

2.2 Smoothing Methods

Smoothing is essential to handle the "zero-count" problem where specific n-grams observed in the test set were never seen during training. In the absence of smoothing, if an n-gram probability is zero, the perplexity effectively tends toward infinity.

- **No Smoothing (MLE with Backoff)**: This method calculates the raw Maximum Likelihood Estimation:

$$P_{MLE}(w_i|h) = \frac{C(h, w_i)}{C(h)}$$

  If the context h is unseen ($C(h) = 0$), the model recursively backs off to the lower-order $(n-1)$-gram probability with a fixed backoff weight (e.g., 0.4). **The Zero Probability Problem:** It is important to note that if a word is still unseen after all backoff steps, the probability becomes zero. This causes the perplexity to explode, as $\ln(0)$ is undefined. In this implementation, a floor of 10^{-10} is used to prevent mathematical errors, which results in the extremely high perplexity scores (e.g., $> 150,000$) seen in the results.

- **Witten-Bell Smoothing**: This method allocates probability mass to unseen events based on $T(h)$, the number of unique types (continuations) following a context h. The probability for a seen n-gram is:

$$P_{WB}(w_i|h) = \frac{C(h, w_i)}{C(h) + T(h)}$$

The reserved mass for backoff to lower-order models is weighted by $\lambda(h) = \frac{T(h)}{C(h)+T(h)}$. The intuition is that if a context has many unique followers, it is more likely to be followed by something new, so we should trust lower-order models more.

- **Kneser-Ney Smoothing**: This implements absolute discounting, where a fixed constant $d = 0.75$ is subtracted from non-zero counts to save mass for unseen words.

$$P_{KN}(w_i|h) = \frac{\max(C(h, w_i) - d, 0)}{C(h)} + \lambda(h)P_{cont}(w_i)$$

Crucially, for lower-order backoff, it uses the *continuation probability* (P_{cont}), which represents how likely a word is to appear in a variety of new contexts, rather than its raw frequency:

$$P_{cont}(w) = \frac{|\{v : C(v, w) > 0\}|}{\sum_{w'} |\{v : C(v, w') > 0\}|}$$

This property helps Kneser-Ney generalize better to unseen data, as confirmed by its superior perplexity results.

2.3 Perplexity Results on Test Set

All 9 model configurations were trained on English data and evaluated.

Table 1: Validation vs Test Perplexity for All 9 Model Configurations

Rank	Tokenizer	Smoothing	Val PPL	Test PPL
1	BPE	Kneser-Ney	87.38	87.71
2	BPE	Witten-Bell	113.52	114.01
3	Whitespace	Kneser-Ney	278.73	281.36
4	Whitespace	Witten-Bell	290.05	292.67
5	Regex	Kneser-Ney	303.94	306.52
6	Regex	Witten-Bell	314.75	317.36
7	BPE	None	147,880.78	150,201.32
8	Whitespace	None	252,915.65	258,799.41
9	Regex	None	315,486.20	322,278.15

Analysis of Results: Kneser-Ney smoothing consistently outperforms Witten-Bell and No Smoothing across all tokenizers, demonstrating its robustness in handling data sparsity and unseen events. BPE tokenization yields significantly lower perplexity (87.71 vs 280-300) because it reduces the vocabulary size and handles rare words via subwords, assigning non-zero probabilities more effectively. The close alignment between Validation PPL and Test PPL (<1% difference) indicates that our models are generalizing well and not overfitting to the training data. The slight increase in test perplexity is expected as the test set represents strictly unseen data.

Regex vs. Whitespace Perplexity: While the Regex tokenizer is more linguistically sophisticated, it yields higher perplexity than the Whitespace tokenizer due to **data sparsity**. By keeping contractions like "can't" or "won't" as single units, the model treats them as unique, low-frequency vocabulary items. In contrast, the Whitespace tokenizer fragments these into high-frequency sub-components (e.g., ['can', "'", 't']), allowing the 4-gram model to learn generalized negation patterns more easily and assign higher statistical probability to those sequences.

2.4 Analysis of Training Data Size

We analyzed the effect of training data size on model performance by comparing models trained on subsets (100k, 200k lines) against the full 800k dataset.

Table 2: Test Perplexity Comparison by Training Size (Kneser-Ney Smoothing)

Tokenizer	100k Lines	200k Lines	Full (800k)
Whitespace	538.34	432.10	281.36
Regex	591.61	473.55	306.52
BPE	145.29	96.18	87.71

Comprehensive Performance Analysis:

- **Data Efficiency**: BPE is remarkably data-efficient. A BPE model trained on just 100k lines (PPL 145.29) significantly outperforms Whitespace (PPL 281.36) and Regex (PPL 306.52) models trained on the full 800k dataset. This suggests BPE captures linguistic structure much faster than word-level models.
- **Scaling Behavior**:
  - **Whitespace/Regex**: Show strictly linear improvement with data size. Doubling data from 100k to 200k reduces perplexity by ∼20%. Increasing to 800k yields further massive gains (432 → 281), indicating these models are still "hungry" for data and haven't saturated.
  - **BPE**: Shows diminishing returns. The jump from 100k to 200k yields a large gain (145 → 96), but the massive increase from 200k to 800k (4× more data) only yields a modest improvement (96 → 87). This suggests BPE learns the core statistical structure of the language very quickly, likely due to its smaller, more focused vocabulary.
- **Smoothing Robustness**: As seen in the full results, Kneser-Ney consistently provides the best performance across all dataset sizes, confirming it as the most robust smoothing technique for both sparse (100k) and dense (800k) data regimes.

2.5 Autocomplete Analysis

Autocomplete demonstrates the model's generative capabilities. I have analyzed 5 distinct prefixes across all models.

2.5.1 Correct Behaviors

- **Common Phrase Completion ("The president of")**:
  - **Output**: "The president of the United States..." (All models)
  - **Analysis**: The 4-gram "president of the United" is extremely frequent. Even models without smoothing get this right. It shows that high-frequency patterns are robust to tokenization choices. BPE continues further: "...United States and Canada. The hotel offers a", showing it constructs longer coherent sequences.
- **Grammatical Consistency ("I can't believe")**:
  - **Output (BPE+Kneser-Ney)**: "I can't believe that I have a few more things to do in"
  - **Analysis**: This is a grammatically perfect sentence. The model correctly predicts the complementizer "that", subject "I", verb "have", and object "things". BPE's subword vocabulary allows it to form complex structure ("have...things") better than word-level models which often get stuck.
- **Contraction Handling ("She was")**:
  - **Output (Regex+Kneser-Ney)**: "She was a little bit of a pain to get the best"
  - **Analysis**: The Regex tokenizer produces very clean output because contractions are single tokens. Unlike Whitespace, there are no spacing artifacts. Kneser-Ney smoothing allows for a colloquial, diverse continuation ("a little bit of a pain") rather than just the most frequent dry statement.

2.5.2 Incorrect Behaviors and Failure Modes

- **Repetitive Loops (No Smoothing)**:
  - **Prefix**: "The quick brown"
  - **Output**: "The quick brown. The details of the problem. I am not"
  - **Analysis**: Without smoothing, if the model hasn't seen "brown fox" enough, it crashes to backoff and switches topics violently to "The details of...". It lacks the "soft" probability mass to pick a semantically related but unseen word, often getting stuck in high-frequency generic loops.
- **Tokenization Artifacts (Whitespace)**:
  - **Prefix**: "I can't believe"
  - **Output**: "I can' t believe it. I' m not sure if this is"
  - **Analysis**: The output contains "can' t" and "I' m". This happens because Whitespace tokenizes "can't" as ['can', "'", 't']. When generating, it adds spaces between these tokens. This makes the text look machine-generated and unpolished, a major downside of simple whitespace tokenization.
- **Semantic Drift ("In the")**:
  - **Prefix**: "In the"
  - **Output (BPE+None)**: "In the meantime, I am sure that the people who are"
  - **Analysis**: While grammatically correct, this shows the generic nature of n-gram models. "In the meantime" is a safe, high-probability bridge. The model quickly drifts to generic filler text ("I am sure that...") rather than generating specific content. This is a limitation of the short context window (n=4).

3 Assumptions and Implementation Details

In addition to the requirements specified in the assignment, the following assumptions and implementation details were adopted:

3.1 Corpus and Preprocessing

- **Line Independence**: The corpus partition (80-10-10 split) assumes line-level independence. Shuffling was performed at the line level, not the document level.
- **Unicode Cleaning**: I explicitly removed zero-width characters (e.g., U+200B) and specific control characters (U+0000-U+0008) to prevent tokenization artifacts.
- **Regex Patterns**:
  - **English**: Assumes that apostrophes inside words (e.g., "don't") should be preserved, represented by `r"\w+(?:'\w+)?|[^\w\s]"`.
  - **Mongolian**: Assumes the Cyrillic block U+0400 to U+04FF covers all necessary characters.

3.2 BPE Tokenization

- **Tie-Breaking**: During BPE training, if multiple pairs have the same highest frequency, the tie is broken based on the iteration order (first encountered pair is merged).
- **Base Vocabulary**: To minimize `<UNK>` tokens, the base vocabulary was initialized with all unique characters found in the training corpus, not just the top-k frequent ones.
- **Vocabulary Size**: Fixed at 5000 as per common practice for small-medium corpora, balancing granularity and sequence length.

3.3 Language Modeling

- **Perplexity Normalization**: Perplexity is normalized by the number of tokens plus the end-of-sentence marker ($N + 1$).
- **Backoff Strategy**:
  - **Unknown Contexts**: If a context h was never seen during training ($C(h) = 0$), the model recursively backs off to the lower-order $(n-1)$-gram probability.
  - **Zero Probabilities**: In the rare case of zero probability (even after smoothing), a floor of 10^{-10} is used to prevent mathematical errors in log-calculation.

3.4 Autocomplete

- **Greedy Decoding**: We assumed that finding the most likely sentence is equivalent to iteratively selecting the locally most probable next token (Greedy Search). Beam search was not used.
- **Generation Constraint**: The model is explicitly prevented from generating `<UNK>` or `<s>` (start) tokens during autocomplete to ensure readable output.

4 Conclusion

This assignment provided a comprehensive analysis of tokenization and N-gram language modeling across English and Mongolian corpora. Our experiments yielded several key insights:

- **Tokenization Impact**: BPE tokenization consistently outperformed word-level approaches, achieving the lowest perplexity (87.71) and showing remarkable data efficiency. By using subwords, BPE captures linguistic structure much faster than Whitespace or Regex models, even with significantly less training data
- **Smoothing Effectiveness**: **Kneser-Ney smoothing**proved to be the most effective technique across all configurations. Its use of continuation probabilities handles the "long tail" of language more effectively than Witten-Bell or simple MLE backoff
- **Generative Limitations**: Use of a 4-gram model with greedy decoding demonstrated coherent local context generation but revealed the limitations of statistical approaches in maintaining long-range semantic coherence, often drifting into generic phrases.

In summary, the optimal configuration for a statistical language model in this constrained setting is **BPE Tokenization combined with Kneser-Ney Smoothing**, offering the best balance of vocabulary size, generalization to unseen data, and computational efficiency.