**Project Report on:**

A study on the evaluation of tokenizers performance on different language families

**1. INTRODUCTION**

Thousands of languages have been adopted by various corners and cultures of our world. Since the dawn of humankind, languages lay the foundation for oral and written communication between humans from different cultures and backgrounds. Natural Language Processing attempts to bridge the gap between human language and computer language by providing well established and tested procedures for converting human language into an intermediate format that can be interpreted by our computers. The boom of state of the art (SoTA) Large Language Models with the introduction of the transformer architecture brought this domain to the spotlight. Research is being carried out with the focus on making this technology as accessible as possible to even the most remote parts of the world.

As stated earlier, Natural Language Processing (NLP for short) is a discipline in computer science (AI to be more specific) that focuses on developing tools and techniques (or leveraging existing ones) to process natural language. The purpose of this field is to provide a computer with the necessary tools to translate human readable language (Ex: English) into the machine’s native readable format, by storing text in an encoded form (ASCII, utf-8) which can be converted. Some of the core objectives of NLP is to enable word segmentation, speech recognition, text to speech, Named entity recognition, Tagging parts of speech in a sentence etc. Some of the methodologies adopted over the years to tackle such tasks is through using Rule based, Statistical methods and Deep Learning models for mapping text patterns.

The primary focus of this study is on **tokenization,** a foundational Natural Language Processing (NLP) technique that where sentences are segmented into their atomic format called **tokens.** Based on the procedures followed and algrithms used, a token can be termed as a word, a subword or a character. There are several approaches to tokenization: **Rule-based methods** use manually crafted rules such as punctuation and whitespace to segment text, which is straightforward but can be inflexible. **Knowledge-based** methods leverage linguistic resources like dictionaries and morphological rules, which are useful for handling specific terms but require extensive resources. **Statistical methods** rely on probabilistic models trained on large corpora to learn token boundaries and patterns, adapting to different contexts but needing significant computational resources. **Neural-based** tokenizers use deep learning techniques to learn patterns from large datasets, handling complex text robustly but requiring considerable computational power. Hybrid approaches combine elements of these methods to optimize accuracy and robustness, balancing precision and resource requirements

A Language is said to be a high resource language (Ex: English) when there are sufficient amounts of resources openly available in the text format on the internet, else it is called a low resource language (Ex: Urdu). Research focusing on these low resourceful languages is sparse, due to the lack of text data availability to perform experiments. Each language has its own diverse set of rules, making it very difficult to develop tools to efficiently process a large variety of language texts, and the various semantic and syntactic interpretations only adds to the difficulty to convert spoken data into text, and following NLP’s procedures for that language. Such challenges decrease the feasibility and scope of experimentation, making the results less reliable.

**1.1 Overview of this Report**

The purpose of this project is to empirically investigate the performance of open sourced tokenization utilities on different language families. The analysis was carried out for three target languages namely Urdu from the parso-arabic family, Simplified Chinese from the Logographic language family and Hindi from the Indic Language Family. This comprehensive report first aims to provide sufficient background information that is required to properly assess all the figures and tables in future sections. Keeping the same purpose in mind, this report has been divided into the following sections:

1. **Section – 2: Background** lays all the necessary foundations (on the surface level) along with a brief discussion of the different language families.
2. **Section – 3: Methodology** goes into all of the technical details of the benchmarking suite that has been setup using tools in the python programming language. This section also outlines some of the key decisions and evaluation criteria that were considered to objectively assess the tokenization’s performance.
3. **Section 4: Evaluation** this section presents an in-depth comparative analysis of the performance of tokenizers across the three language families. We examine the results, providing detailed insights into how each tokenizer performs with Urdu, Simplified Chinese, and Hindi. The analysis includes discussions on the effectiveness of each tokenizer and potential reasons for the observed performance trends.
4. **Section – 5: Conclusions and Limitations** hopes to provide a brief summary of the key findings and offer concluding statements. It also addresses the limitations of the study, highlighting factors that could influence the results and suggesting considerations for future research or applications based on this analysis.

**2. BACKGROUND**

**2.1 About the Unicode Standard**

Simply put, the unicode standard is an encoding format that is used for computers to better represent text that isn’t just english. Formally, the unicode standard is a text encoding standard designed to support the use of text in the digital format. The formulation of this standard was to solve the compatibility problem between encoding formats, where often the encoding representation of one format is interpreted as garbage by the other. Many normal characters have been unified under its umbrella, making it easier to eliminate language specific digital representation systems. The underlying principle is to encode every grapheme/grapheme-like units by assigning each grapheme with a **code point,** which enables it to support a wide variety of high resource and low resourceful languages.

The default encoding form of any language for the entire internet is the **Unicode Transformation Format (utf-8),** a lightweight unicode format. It is optimized for byte-oriented systems where backward compatibility with ASCIIis important, and is easier to parse compared t o legacy encoding schemes. This is important because the datasets that have been obtained during this study are from an internet repository, and each file is stored in the **utf-8** format.

**2.2 Target Languages**

According to wikipedia, a language family is a group of languages related through descent from a common ancestor, called a proto-language of that family. The divergence of the proto language into respective daughter languages happens due to geographical separation, with different regional dialects of the proto-language undergoing different languag changes. For this study, the following languages have been considered:

1. **Urdu (Perso-Arabic):** The Urdu language alphabet is a modified form of the Perso-Arabic script, which itself is derived from the Arabic script. It consists of 39 primary characters, including 27 consonants and 12 vowels. Urdu is written from right to left, and like Arabic, it is an abjad script, meaning it primarily represents consonants, with vowel sounds indicated by diacritics or implied through context. In the unicode standard, urdu falls under the arabic block, ranging from **0600-06FF**

A significant number of Urdu words are retained from Persian and Arabic, contributing to a rich vocabulary and diverse orthographic influences. One of the many key features of Urdu script is its use of ligatures, where characters combine into complex forms depending on their position in a word. This script's cursive nature requires that letters be connected, which can make reading and writing more fluid but also requires more intricate training.

1. **Simplified - Chinese (Logographic):** Simplified Chinese characters are a modernized writing system developed to boost literacy and streamline the traditional Chinese script. Officially introduced in the 1950s and 1960s, Simplified Chinese reduces stroke counts in many characters, enhancing learnability and ease of writing. This script is primarily used in Mainland China, Singapore, and Malaysia. It features fewer radicals and simplified versions of more complex traditional characters, significantly improving efficiency for both learners and writers.

**1. CJK Unified Ideographs (U+4E00 to U+9FFF)**

**2. CJK Unified Ideographs Extension A (U+3400 to U+4DBF)**

**3. CJK Unified Ideographs Extension B (U+20000 to U+2A6DF)**

While simplified characters maintain the core semantic elements of their traditional counterparts, they feature fewer strokes and less complex forms. This simplification aims to make reading and writing more accessible, particularly for those with limited formal education. Simplified Chinese maintains a strong connection to its traditional roots, preserving much of the language's rich historical and cultural context.

1. **Hindi (Indic):** Hindi, one of the major languages of India, has its origins in the Indo-Aryan branch of the Indo-European language family. It evolved from Sanskrit through the Prakrit and Apabhramsha languages, with its roots traceable to ancient Vedic texts. Hindi began to take shape during the medieval period, influenced by various regional dialects and languages, including Persian, Arabic, and Turkic, due to historical invasions and trade. The unicode range for Hindi is **U+0900 to U+0967**

Uniquely, Hindi is written in the Devanagari script, which is known for its phonetic accuracy and distinct horizontal line running across the top of the letters. The language features a rich system of sounds, including retroflex consonants and nasalized vowels, which contribute to its melodic and rhythmic quality. Additionally, Hindi’s diverse vocabulary incorporates elements from various languages, reflecting its historical interactions and cultural amalgamations.

**2.3 Dataset info**

The CC100 dataset is a comprehensive large-scale collection of text corpora gathered in multiple languages. The primary objective of this dataset is to enhance the accessibility of natural language processing (NLP) tasks across a broad range of languages. The CC100 dataset comprises texts collected through web scraping from diverse sources, including books, articles, and websites scattered across the internet. This comprehensive collection ensures a wide representation of topics and styles, making it highly valuable for training and evaluating language models. The creation of CC100 reflects a concerted effort to diversify and enrich the corpus of high-quality text available for training artificial intelligence models

For the purposes of this study, the following language datasets have been extracted from the CC100 collection:

1. **CC100 – Urdu (Code word: ur)**
2. **CC100 - Chinese (Code word: zh-Hans)**
3. **CC100 – Hindi (Code word: hi)**

These subsets of the CC100 dataset provide a significant amount of text in their respective languages, facilitating various NLP tasks such as language modeling, translation, and text classification. By offering extensive and diverse text data, CC100 supports advancements in multilingual NLP research and helps build more robust and inclusive language models.

**2.4 Tokenization**

Tokenization is a fundamental technique in Natural Language Processing (NLP) that involves breaking down text into its basic units, known as tokens. This process is crucial for converting a corpus of text into manageable pieces, which can be words, subwords, or characters, depending on the method used. A tokenizer is a tool or program designed to perform this segmentation, generating a list of tokens from the text.

There are several approaches to building tokenizers, each with its own characteristics:

1. **Rule-Based Tokenization**: This method segments text using predefined rules based on explicit patterns such as punctuation marks and whitespace. It is straightforward and effective for languages with clear delimiters but can be inflexible with complex or ambiguous text. This approach often requires continual updates as language use evolves.
2. **Knowledge-Based Tokenization**: This approach relies on linguistic resources like dictionaries, lexicons, and morphological rules to identify tokens. It is effective for handling domain-specific terms and proper nouns but may need extensive and specialized resources to cover all text variations.
3. **Statistical**: Statistical methods use probabilistic models trained on large text corpora to learn patterns and boundaries for tokens. These models can adapt to different contexts and handle text variations effectively, though they often require substantial computational resources and large datasets.
4. **Neural-Based Tokenization**: Neural-based tokenizers employ deep learning techniques and neural networks to learn tokenization patterns from extensive training data. They are capable of managing complex and ambiguous text across various languages but require significant computational power and large annotated corpora for effective training.
5. **Hybrid Approaches:** Hybrid tokenizers combine elements of rule-based, knowledge-based, statistical, and neural methods to leverage the strengths of each. This approach aims to balance accuracy, adaptability, and resource efficiency, providing a more robust and precise tokenization solution.

Other techniques such as Byte Pair Encoding, SentencePiece, WordPiece, and n-gram models also contribute to the diverse landscape of tokenization methods, each offering distinct advantages for different NLP tasks. There are many natural language processing libraries that are openly available for developers with built in tokenization utilities with various implementations. This study’s focus is on the widely used tokenizers namely: **NLTK (Natural Language Toolkit), SpaCy, Stanza (Stanford NLP), jieba, IndicNLP.** Each of these libraries have implemented their tokenization utilities either as a pure rule based/ neural based or they have a hybrid between rule based and nerual implementations.

**3. METHODOLOGY**

**3.1 Key Libraries**

To enhance the efficiency of our experimental setup and streamline the process of loading and storing input and output files, we have utilized several key libraries. These libraries play a crucial role in optimizing performance and managing large datasets effectively. Here’s a detailed overview of the libraries employed:

1. **mmap**: The mmap module enables memory-mapped file I/O, which allows for lazy loading of text files. This technique minimizes the consumption of main memory by mapping files directly into memory and providing random access to large files without the need to read them entirely into memory. This approach ensures optimal performance and efficient handling of substantial text data.
2. **Plyvel**: Plyvel is a Python wrapper for LevelDB, a high-performance key-value store. Plyvel is particularly effective for managing large datasets, making it well-suited for storing unique tokens generated during the tokenization process. Its ability to handle rapid read and write operations significantly enhances the efficiency of data storage and retrieval.
3. Natural Language Processing (NLP) Packages:
   1. **NLTK (Natural Language Toolkit)**: NLTK is a comprehensive library that supports symbolic and statistical NLP tasks. It provides a broad range of tools for text processing, including tokenization, stemming, and part-of-speech tagging, making it a versatile choice for various NLP applications.
   2. **spaCy**: spaCy is a powerful NLP library designed for industrial use, featuring pre-trained statistical models and word vectors. It excels in tasks such as efficient tokenization, named entity recognition, and dependency parsing, offering robust performance and accuracy.
   3. **Stanza**: Developed by the Stanford NLP Group, Stanza is a Python NLP package that supports multilingual text analysis. It includes tools for tokenization, multi-word token expansion, lemmatization, and dependency parsing, facilitating comprehensive language processing capabilities.
   4. **IndicNLP**: This library is tailored for processing Indian languages and includes tools for tokenization and other language-specific tasks. It helps in handling the linguistic nuances of Indic languages effectively.
   5. **Jieba**: Jieba is a popular library for Chinese text segmentation. It is well-suited for tokenizing Chinese text into meaningful units, leveraging both dictionary-based and statistical methods to handle the complexities of Chinese language segmentation.

**3.2 Experimental Setup**

The suite of benchmarks were developed using the python programming language. Python was chosen due to it’s ease of use and a wide collection of libraries (all of the ones mentioned at the end of previous section) for performing large scale NLP tasks. A custom pipeline was built for streamlining the data extraction and processing tasks, while making sure that each process is timed and logged properly. The purpose of this pipeline is to clean the data to remove irregularities and generate a word list (vocabulary) which only contains all the unique tokens from the tokenization process. For a given corpus of input text, the following are the procedures used to efficiently extract and tokenize the text:

1. **Loading and Cleaning**

* **Loading:** The raw text file is read into memory to prepare the content for processing. This step ensures that the entire text data is accessible for subsequent operations.
* **Cleaning:** Separate every sentence based on full stops and exclamation marks and apply language-specific cleaning rules to normalize the text. This includes removing unnecessary characters, standardizing formatting, and addressing any language-specific issues to ensure consistency and readability. Regular Expressions (re) were used for automating this process.

1. **Creating the intermediate file**

* Store the cleaned text in a new file, formatted with each line representing a distinct segment of text. A segment in this file is a single line that was separated by a new line character.
* A set number of individual sentences are then sampled randomly (seeding is done) to be stored in a different file, which is later sent for tokenization.

1. **Tokenization**

* Various tokenization methods suited to the target language to process the intermediate file were utilised, namely:
  + **For Urdu:** NLTK’s Treebank tokenizer, SpaCy’s hybrid tokenizer, and Stanza’s Urdu tokenization process.
  + **For Simplified Chinese:** Jieba’s tokenizer, SpaCy’s tokenizer, and Stanza’s Chinese tokenization processor.
  + **For Hindi:** IndicNLP’s rule based implementation, SpaCy’s Hindi tokenizer and stanza’s Hindi tokenization processor.
* Clean the tokenized data by removing duplicates and irrelevant tokens to refine the list. Compile a final list of unique tokens from the filtered data.
* The tokens are then stored in a key-value store (such as LevelDB) which can be later retrieved as a CSV/TXT file when desired.

**3.3 Evaluation Criteria**

Assessing the performance of a tokenizer on a given corpus can be challenging, particularly when dealing with the complexities of linguistic rules and semantics. In the absence of language experts and due to time constraints, this study focuses on a quantitative approach using statistical methods to evaluate tokenizer performance. The evaluation is based on the following key metrics:

1. Choosing the appropriate tokenizers.
2. Includes the time required to tokenize a file and the total number of unique tokens generated. These metrics provide initial insights into the efficiency and scale of the tokenizer.
3. To measure the minimum, maximum, and average lengths of the token lists produced by each tokenizer. This helps in understanding the distribution and consistency of the generated tokens.
4. To analyze the frequency distribution curves of token lengths. This involves examining how token lengths are distributed and interpreting trends in relation to the tokenizer's implementation and behavior.
5. Evaluates the quality of tokenization by assessing factors such as accuracy and relevance of tokens in the context of the text. This metric helps determine how well the tokenizer preserves the meaning and structure of the original text.
6. Determines the performance ranking of each tokenizer based on the aforementioned metrics. This ranking provides a comparative analysis of the tokenizers, highlighting their relative strengths and weaknesses.

**4. EVALUATION**

**4.1 Overview**

For each language, the analysis includes:

1. An overview of the key linguistic rules and characteristics that can affect tokenization.
2. Choice of tokenizers used.
3. Evaluation of performance metrics, such as tokenization time, unique token count, and token length statistics.
4. Assessment of the quality of the generated tokens through a hit-ratio based statistic.

**4.2 Urdu**

**4.2.1 Language Characteristics Affecting Tokenization**

1. **Right-to-Left Script**: Urdu is written from right to left, unlike English, which is left-to-right. This directionality impacts text layout and requires tokenizers to be adapted for right-to-left processing.
2. **Diacritics and Punctuation**: Urdu uses diacritical marks which are essential for pronunciation and meaning but can complicate tokenization. Handling these elements accurately is crucial for proper text processing.
3. **Mixed Scripts**: Urdu texts often include words from other languages, particularly English, written in Latin script. This mixing of scripts requires tokenizers to effectively manage and process multiple alphabets within the same text.
4. **No Clear Word Boundaries**: Urdu frequently features compound words and phrases with ambiguous word boundaries. This necessitates advanced tokenization techniques to accurately segment text despite these ambiguities.

**4.2.2 Choice of tokenizers and preliminary results**

The following four tokenizers have been selected for analyzing a text file containing 5 million sentences randomly sampled from the Urdu dataset in the CC100 corpus: NLTK, spaCy, Stanza, and IndicNLP. These tokenizers include NLTK’s Treebank tokenizer, spaCy's tokenizer, Stanza's Urdu tokenizer, and IndicNLP's tokenization processor.

A comparison of key metrics, such as the time taken for tokenization and the number of unique tokens generated, is provided in the below table. It is worth noting that the tokenization time varies significantly across these tools. A plausible explaination to this variance can be attributed to the unique implementation and design strategies followed for each tokenizer.

For example, NLTK's Treebank tokenizer is known for its rule-based approach. In contrast, spaCy's tokenizer uses efficient, pre-trained models and optimized processing pipelines. Stanza's Urdu tokenizer, developed specifically for multilingual support, uses neural models that balance accuracy and performance. IndicNLP's tokenizer, tailored for Indian languages, incorporates language-specific rules that may influence both speed and token accuracy.

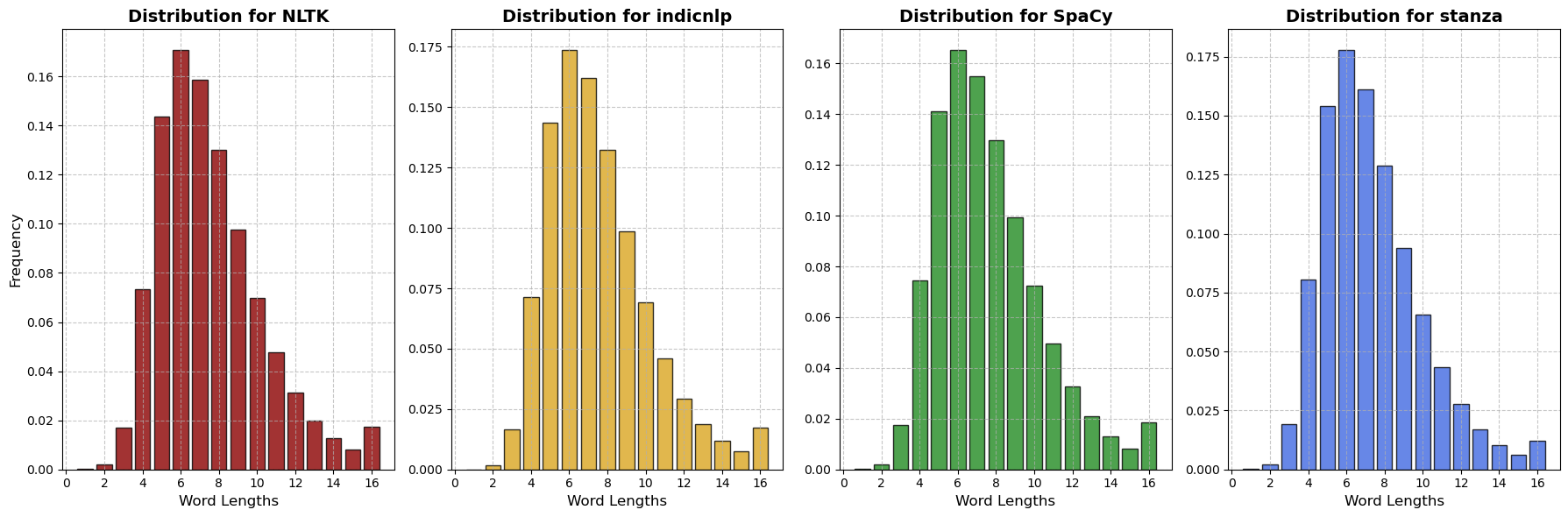
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Time taken (mins/hrs)** | **Unique tokens** | **Min length** (in unicode characters) | **Max length** (in unicode characters) | **Average length** (in unicode characters) |
| **NLTK** | 12.3 Mins | 10,14,248 | 1 | 323 | 7 |
| **SpaCy** | 10.5 Mins | 9,41,392 | 1 | 323 | 7 |
| **IndicNLP** | **7 Mins** | **9,21,436** | **1** | **323** | **7** |
| **Stanza** | 5 hrs | 9,02,171 | 1 | 250 | 7 |

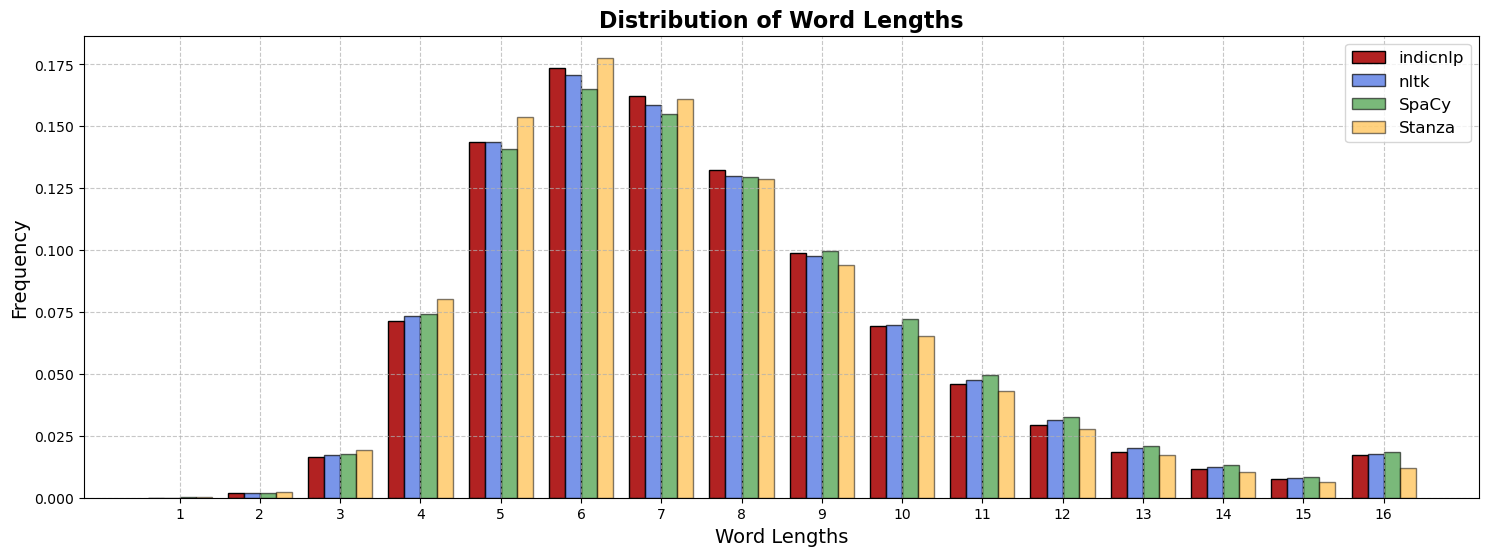
It is worth noting that the lengths of each token was calculated using the python’s len() function. According to the standard library’s documentation, the len() function calculates the length of a string by counting the number of individual unicode characters present in that string. Therefore, it is important to check the unicode chart for each language to identify characters.

**4.2.3 Evaluation based on frequency distributions**

The next experiment was to quantitatively analyse the existing research and determine whether the tokenizer’s results align with the same. In a paper written by *AbdelRaouf et al*, it was shown that the average length of an Arabic word is 5 characters. It is worth noting the results from the above paper were obtained after performing character recognition of street signs. To check the validity of our tokenizers, the research was taken as the ground truth, and the character lengths for each unique token was measured and stored in the database.

These stored lengths were then binned and the frequency distribution for each bin are then visually plotted, as shown below:



****

From the above plots, it can be observed that the frequency distributions from the resultant token list is similar across all three selected tokenizers. The regions from lengths 4-9 represent up to 70 % of the overall distribution. It can be concluded that the distributions for all four tokenizers is almost similar, we proceed to use more metrics to arrive at a clear choice

**4.2.3 Quality Assessment**

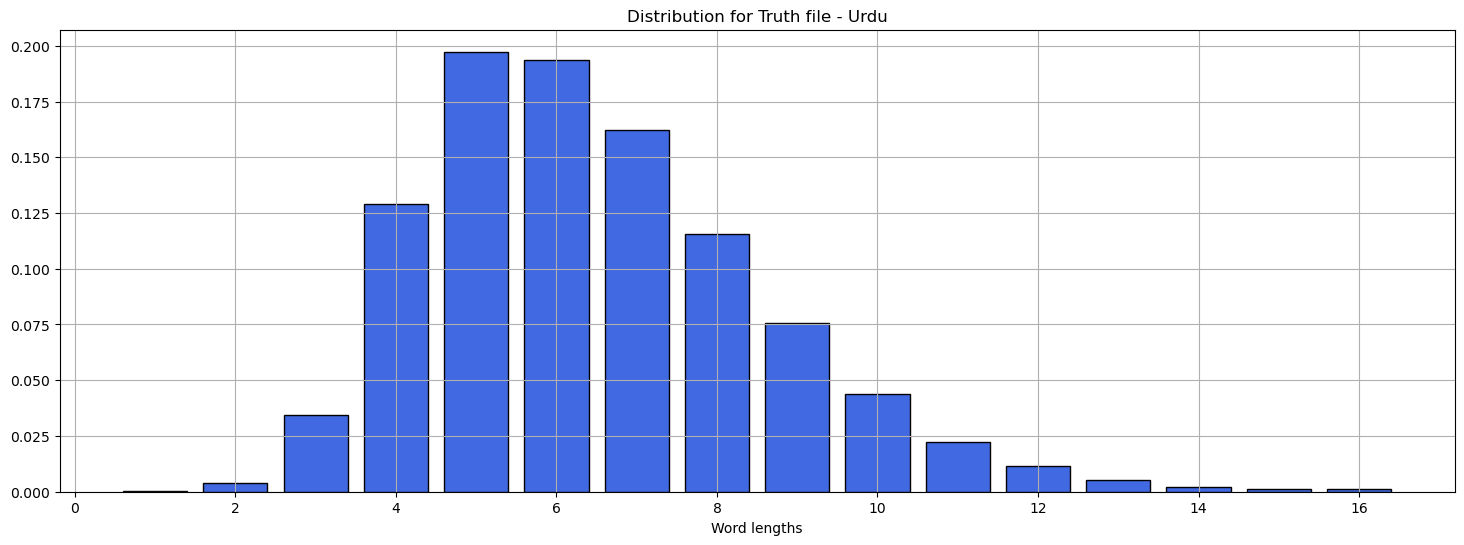
The **hit ratio** is a key metric used to assess the effectiveness of various systems, including tokenizers, by measuring the proportion of successful outcomes relative to the total number of attempts. In the context of tokenizer evaluation, the hit ratio gauges how well a tokenizer identifies words from a predefined reference list. This list typically comprises important or relevant terms that serve as a benchmark for performance.

To calculate the hit ratio, first identify how many words from the reference list are correctly found in the tokens generated by the tokenizer. Next, count the total number of words in the reference list. For instance, if a reference list contains 100 words and the tokenizer correctly identifies 80 of them, the hit ratio would be 80%.

A higher hit ratio indicates that the tokenizer is effective in capturing relevant words, suggesting strong performance. Conversely, a lower hit ratio implies that the tokenizer may be missing significant terms, reflecting potential areas for improvement. For calculating the hit ratio, a reference file with approximately **150K urdu words** has been chosen. The same file was obtained from urduhack, a library specifically built for processing urdu text, which unfortunately was not included in this comparision due to dependency errors which couldn’t be resolved in time. This file has also been passed into the cleaner utility of the pipeline to ensure consistency across both the files.

The average length (in unicode characters) of all the words was found to be 6, and maximum length to be 22. While the average length is approximately what we got during our experiments, the maximum length obtained from the tokenization experiments was ranging from 250 – 300 unicode characters, indicating that there are many outliers generated. These outliers can either be cleaned further and inspected by language experts or can be removed. The hit ratio results have been calculated by keeping these outliers.

The frequency distribution plots of this file was plotted following earlier procedures. The length of each word (in unicode characters) was obtained, which were then used for binning with the respective frequency count and plotted as shown in the figure below:

 **Fig:Frequency Distribution plot for urduhack’s word dictionary**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tokenizer** | NLTK | spaCy | IndicNLP | Stanza |
| **Hit Ratio** | **38.30 %** | **38.57 %** | **38.30 %** | **38.59%** |

Based on the table provided above, it can be observed that the hit ratio amongst all four of the chosen tokenizers is similar. NLTK and IndicNLP have the exact same hit ratios as their implementations are rule based and they focus more on splitting the input text based on whitespaces. Stanza has a marginally higher hit ratio than spaCy, but if time is considered then stanza took a significantly longer amount of time.

NLTK and IndicNLP can be considered desirable, if there is a time and compute constraint. If there’s no time and compute constraint, then one can consider using stanza/spacy for better accuracy.

**4.3 Simplified Chinese**

**4.3.1 Language Characteristics Affecting Tokenization**

1. **Lack of Word Delimiters:** Chinese lacks spaces between words, which makes it difficult to determine word boundaries.
2. **Complex Character Combinations:** Characters can represent whole words or concepts, and multi-character words increase the complexity of tokenization.
3. **Segmentation Issues**: Accurate word segmentation is difficult due to the high density of homophones and similar-sounding words.
4. **Mixed-Language Texts:** Chinese text may include foreign words or mixed languages, requiring tokenizers to handle multiple languages.

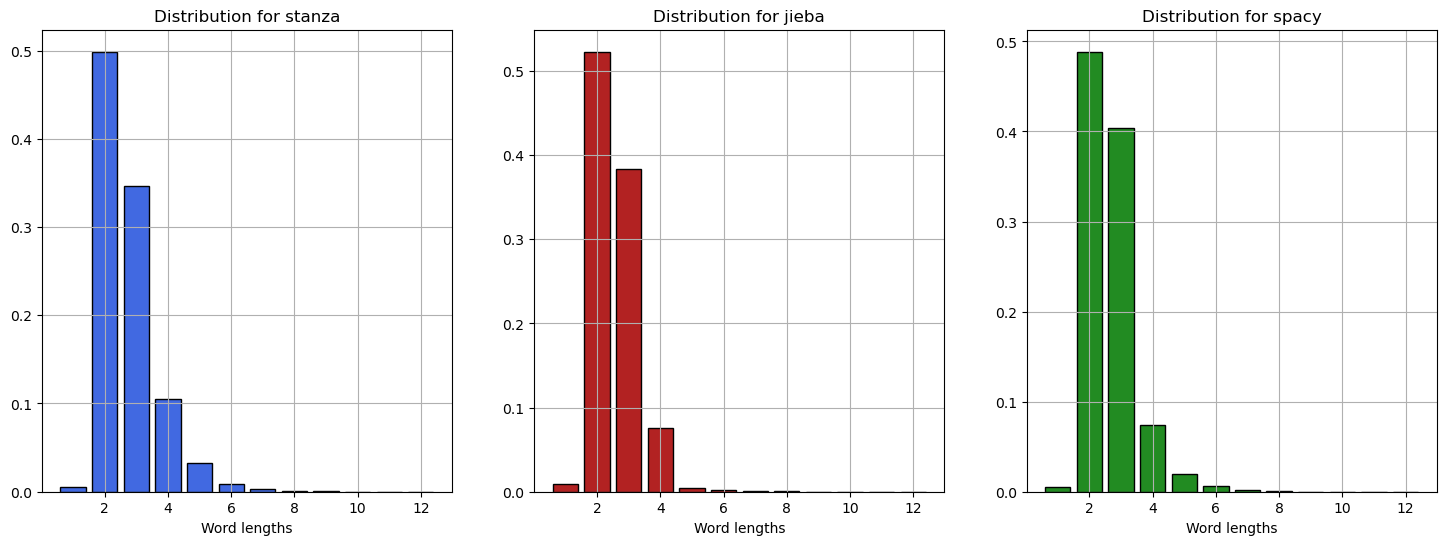
**4.3.2 Choice of tokenizers and preliminary results**

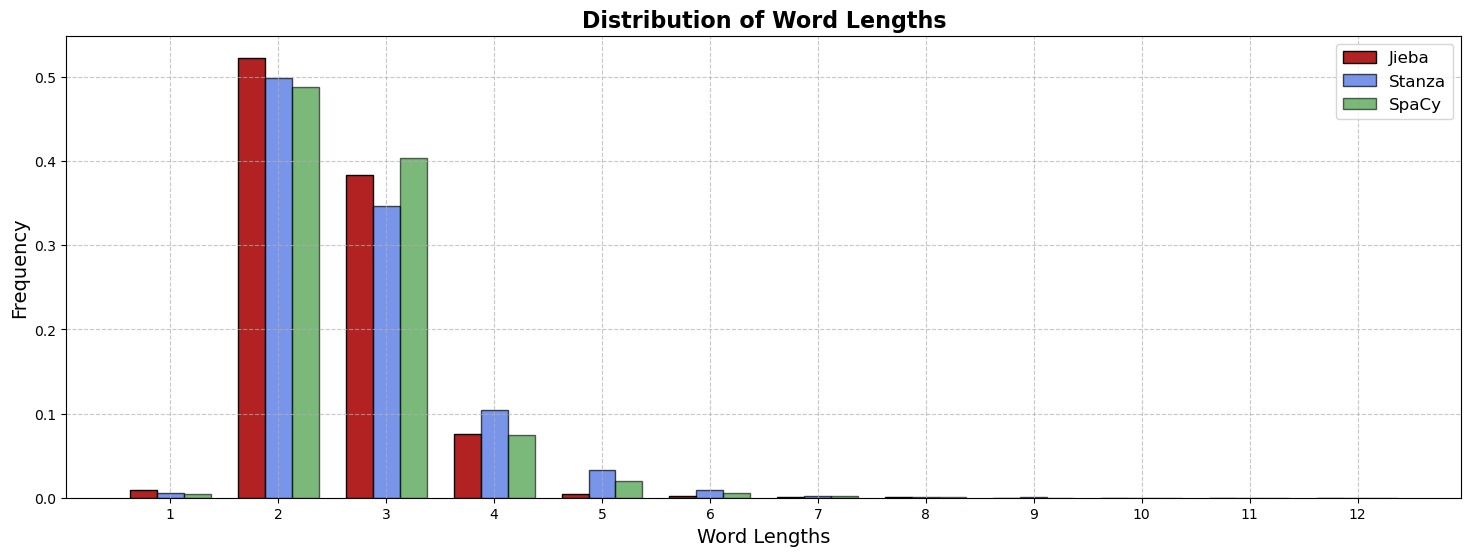
The following three tokenizers have been selected for analyzing a text file containing 5 million sentences randomly sampled from the Simplified chinese dataset in the CC100 corpus: Jieba, spaCy, Stanza. The metrics discussed above for urdu have also been applied here to ensure consistency across all the languages.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Time taken** | **Unique tokens** | **Min length** (in unicode characters) | **Max length** (in unicode characters) | **Average length** (in unicode characters) |
| **Jieba** | **17 Mins** | **8,75,311** | **1** | **16** | **2** |
| **SpaCy** | 4 hours | 14,40,220 | 1 | 17 | 2 |
| **Stanza** | 3.8 hours | 16,79,448 | 1 | 27 | 2 |

**4.3.3 Evaluation based on frequency distributions**

Following similar procedures from Urdu, the ground truth for chinese was considered as the research conducted by *Zhang.Y et al.* The research claims that the average character length of a chinese token was found to be ~2, after processing the bilingual corpus dataset. Since the type of chinese text (Simplified or Traditional) was not mentioned, the results may vary based on different implementations. The following plots visually show the frequency distributions for the three chosen tokenizers from **Jieba, spaCy and Stanza**

 **(a)**

 **(b)**

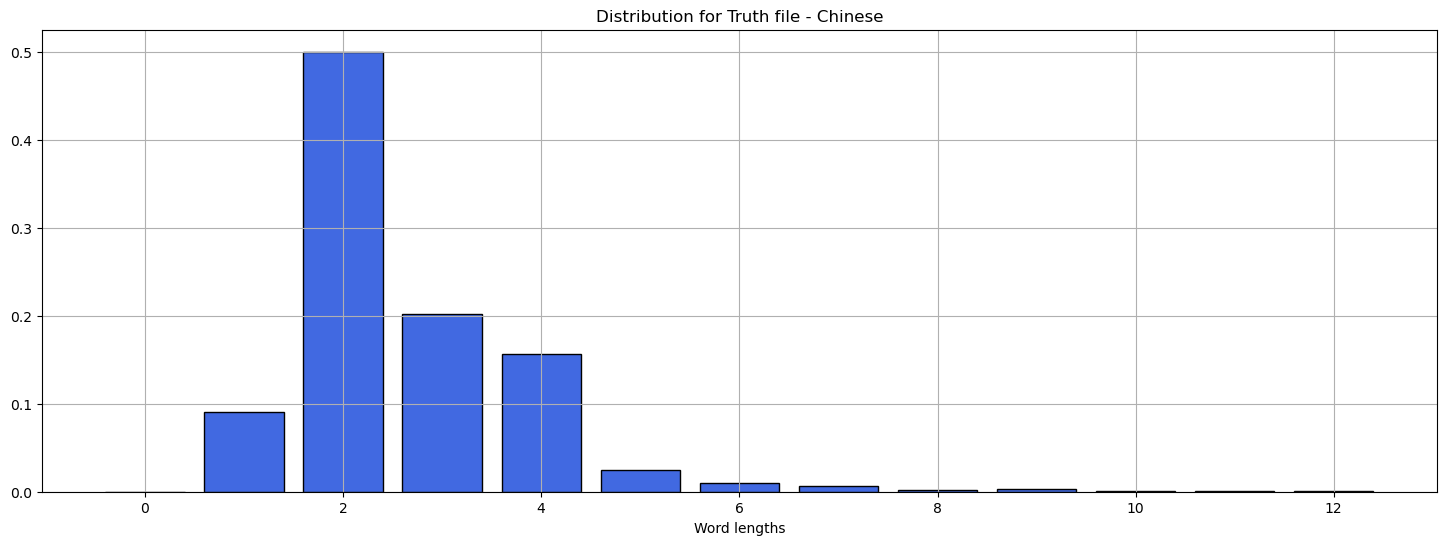
**Fig: (a), (b)Frequency Distributions of Jieba, spaCy and Stanza**

From the above plots, it can be observed that the frequency distributions from the resultant token list is similar across all

three selected tokenizers. The regions from lengths 2-4 represent up to 95 % of the overall distribution. It can be concluded that the distributions for all four tokenizers is almost similar, we proceed to use more metrics to arrive at a clear choice

**4.3.4 Quality Assessment**

The ground truth file considered for chinese is CEDICT, a widely used chinese dictionary for training tokenizers. The simplified chinese column from the CEDICT was extracted and saved as a csv file. This file contains approximately 130K unique simplified chinese words, although it is to be noted that the file has been cleaned using the same procedures as the original text to keep the file consistent and remove unnecessary words. The maximum length of a word in this file is 37 unicode haracters long, while the average is 2 characters, which reinforces our prior results.

The following plot was made to examine the frequency distribution of word lengths using the same procedure as above:

**Fig: Frequency Distribution for CEDICT**

From the above plot, it can be concluded that the tokenization went as expected, although few anomalies are left to be explained such as the maximum character length obtained in our experiment was higher than the ground truth file. We can consider these as outliers and remove them, as they lie in the minority (<5 %) class of the distribution in our experimental results.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tokenizer:** | Jieba | spaCy | Stanza |
| **Hit Ratio** | **69.7 %** | **67%** | **59.8%** |

The above table represents the respective hit ratios for Jieba, SpaCy and Stanza respectively. Based on our earlier observations, it can be concluded that **Jieba**, a library specially designed for processing chinese text, is capable of giving the desired results.

**4.4 Hindi**

**4.4.1 Language Characteristics Affecting Tokenization**

1. **Complex Script**: Hindi is written in the Devanagari script, which features ligatures and complex character combinations, making it harder to separate words accurately.
2. **Inflections and Derivations**: Hindi words often change form through inflections and derivations, adding complexity to tokenization as tokenizers must handle different word forms and their variations.
3. **Out-Of-Vocabulary (OOV) Words**: New words, slang, or domain-specific terms not present in standard dictionaries can pose challenges for tokenizers.
4. **Mixed-Language Texts**: Hindi text may include English or other language elements, requiring tokenizers to manage multiple languages within a single document.

**4.4.2 Choice of tokenizers and preliminary results**

The following three tokenizers have been selected for analyzing a text file containing 5 million sentences randomly sampled from the Simplified chinese dataset in the CC100 corpus: Jieba, spaCy, Stanza. The metrics discussed above for urdu have also been applied here to ensure consistency across all the languages.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Time taken | Unique tokens | Min length | Max length | Average length |
| IndicNLP | **24.2 Mins** | **7,94,002** | **1** | **195** | **7** |
| spaCy | **24.2 Mins** | **7,94,002** | **1** | **195** | **7** |
| Stanza | 5 hrs | 7,94,481 | 1 | 195 | 7 |

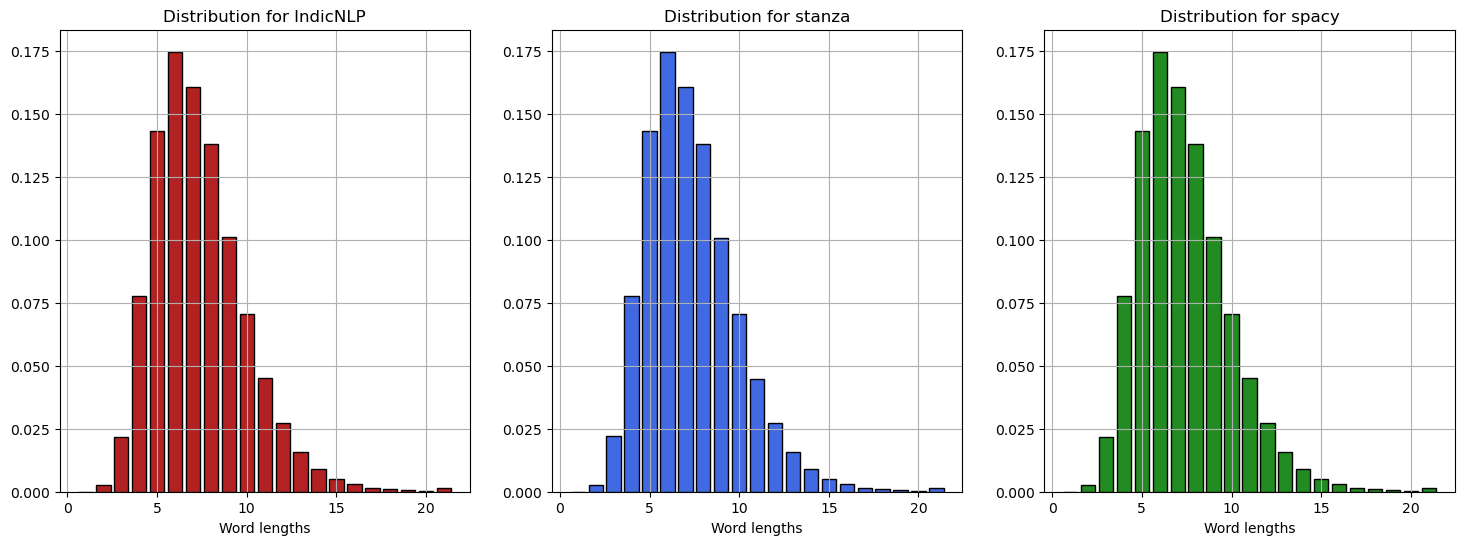
After observing from urdu and chinese, a recurring trend lies in the abnormally long times that stanza is taking for tokenization, while for each language all the others are finishing it in less than an hour, stanza seems to be taking more than 3 hours consistently. It is worth underlining that stanza’s implementations are neural network based, which makes it computational requirements more demanding compared to other systems.

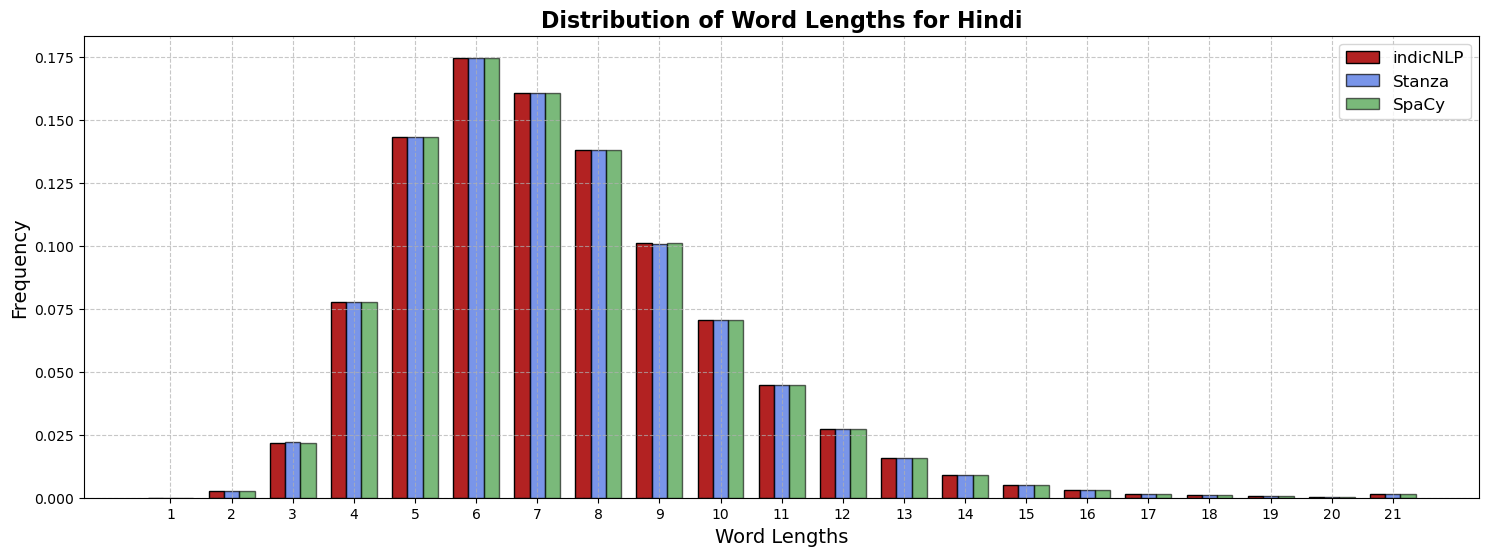
It is also worth noting that all the times listed through these tables are a result of isolating the tokenization and storing process away from the cleaning process, to give a fair opportunity for the tokenization process and to objectively evaluate the amount of time each tokenizer takes to process a language.

In Hindi, an interesting observation would be the identical number of tokens and time taken for both IndicNLP and spaCy to tokenize the Hindi sentences. To ensure reproducibility, the same procedures have been run again for both of them and the results are the same. One plausible explanation might be the implementations of spaCy’s Hindi processor, which also might be tokenizing based on whitespace characters like IndicNLP.

**4.4.3 Evaluation based on frequency distributions**

Following similar procedures from Urdu, the ground truth for chinese was considered as the research conducted by \_\_*.* The research claims that the average character length of a hindi token was found to be \_, after processing the bilingual corpus dataset. Since the type of chinese text (Simplified or Traditional) was not mentioned, the results may vary based on different implementations. The following plots visually show the frequency distributions for the three chosen tokenizers from **IndicNLP, SpaCy and Stanza**

 (a)

 (b)

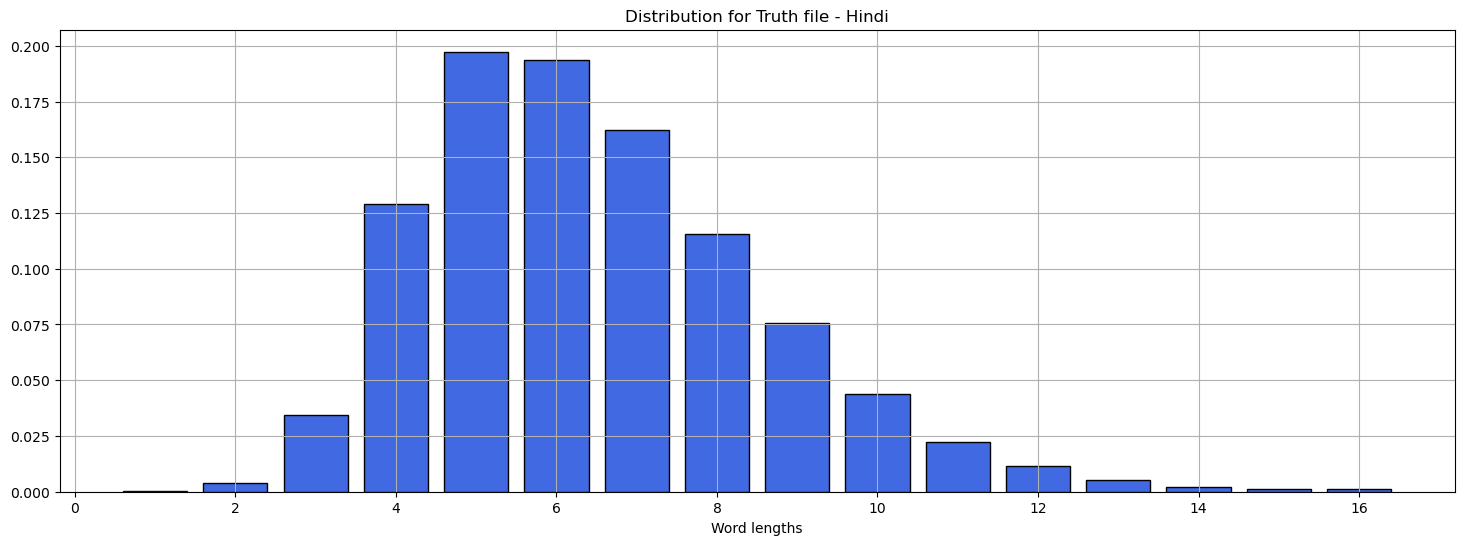
**Fig: (a), (b) Frequency Distributions for Hindi Tokenizers: IndicNLP, spaCy, Stanza**

From the above plots, it can be observed that the frequency distributions from the resultant token list is similar across all

three selected tokenizers. The regions from lengths 4-9 represent up to 80 % of the overall distribution. It can be concluded that the distributions for all three tokenizers are similar, we proceed to use more metrics to arrive at a clear choice.

**4.4.4 Quality Assessment**

Following the same steps as Urdu and Chinese, the ground truth file for hindi has been obtained from a github repository. This file has approximately 200k words, which have been cleaned to maintain consistency. The following plot shows the frequency distribution of these words, binned at different lengths:

**Fig: Frequency Distribution for Hindi dict**

The average word length of this file is 7 unicode characters and the maximum word length is 37, which is much lower than the 195 obtained from our experiment.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tokenizer:** | IndicNLP | spaCy | Stanza |
| **Hit Ratio** | **37.3 %** | **37.3 %** | **37.3 %** |

**Table: Hit Ratios for Hindi’s tokenized text, measured against the ground truth file**

Based on the table provided above, it can be observed that the hit ratio amongst all three chosen tokenizers is similar. IndicNLP can be considered desirable, if there is a time and compute constraint. If there’s no time and compute constraint, then one can consider using Stanza/spaCy for better accuracy.

**5. CONCLUSIONS AND LIMITATIONS**

**5.1 Conclusion**

**5.2 Limitations and Future Scope**

While the study was performed with a purpose of thoroughly covering the effect of tokenizers on as many langauge families as possible, due to time constraints the following limitations have been imposed:

1. The sample size of the input file for tokenization was taken to be 5M sentences, but it was found that taking larger sample sizes for random sampling can have a positive impact on hit ratios.
2. The choice of tokenizers was fairly limited to each language. This can be addressed by experimenting with more custom built hybrid implementations.
3. More metrics can be devised to perform a more rigorous and thorough examination of the tokens lists generated.
4. The pipeline for cleaning and processing raw data can be further optimized to improve speed, more data storage and compute options can be experimented with and validated.
5. Using the improved pipeline and a more robust suite of metrics, the overall process can be implemented for more language families such as the Latin (English, Latin), Indic (Bengali, Sanskrit, Marathi) etc.

**REFERENCES**

[1] AbdelRaouf, A., Higgins, C. A., & Khalil, M. (2008). A database for Arabic printed character recognition. In *Image Analysis and Recognition: 5th International Conference, ICIAR 2008, Póvoa de Varzim, Portugal, June 25-27, 2008. Proceedings 5* (pp. 567-578). Springer Berlin Heidelberg.

[2] Zhang, Y., Brown, R. D., Frederking, R., & Lavie, A. (2001). Pre-processing of bilingual corpora for Mandarin-English EBMT. In *Proceedings of Machine Translation Summit VIII*