Text Preprocessing

Tokenization

Tokenization is the process of dividing a given corpus into units called tokens. The unit of a token varies depending on the context, but tokens are usually defined as meaningful units.

Tokenization > Sentence Tokenization

- *Sentence Tokenization:* Dividing a corpus into sentences. Also known as sentence segmentation.

- **Necessity:** Often required when the corpus is not already divided into sentences.
- **Naive Approach:** Splitting on "?", ".", or "!" seems intuitive but is not always accurate.
- **Challenges with Periods (.):**
- "?" and "!" are relatively clear sentence boundaries.
- Periods are ambiguous because they can appear within words or abbreviations.
- **Custom Rules:** You can define rules based on the language and specific characteristics of the corpus.
- **Limitations of Rules:** Achieving 100% accuracy is difficult due to typos or poorly formed sentences.
- **NLTK's sent_tokenize:** NLTK provides sent_tokenize for sentence tokenization in English.

★ tasks like cleaning or standardizing text, heavily depends on the characteristics of the corpus it's applied to.

Considerations

Language Specificity: Rules designed for one language will likely fail or produce incorrect results when applied to another. Languages have different grammatical structures, punctuation conventions, and character sets.

Special Characters and Punctuation: The way special characters and punctuation are used varies across languages and even within different contexts within a single language. A rule that assumes a period always marks the end of a sentence might fail if the text contains abbreviations (e.g., "Mr.") or decimal numbers (e.g., "3.14"). Understanding the corpus's specific conventions is crucial.

Corpus Quality: As you pointed out, the quality of the corpus is a significant factor. If the corpus contains typos, grammatical errors, inconsistent formatting, or other inconsistencies, it will be very difficult to create rules that work reliably. "Garbage in, garbage out" applies here.

Achieving 100% Accuracy: It's almost impossible to achieve 100% accuracy with rule-based systems, especially when dealing with real-world text data. Natural language is inherently complex and ambiguous, and there will always be edge cases that the rules don't cover.

Hybrid Approaches: In some cases, a hybrid approach that combines rule-based methods with machine learning techniques can be more effective. Rule-based methods can be used to handle common cases, while machine learning models can be trained to handle more complex or ambiguous cases.

Solution

Analyze Corpus
Understand the language, character set, punctuation conventions, and common errors

Start with a basic set of rule
Focus on the most frequent and consistent patterns

Test and Evaluate
Evaluate rules on the representative sample of the corpus

Code

```
from nltk.tokenize import sent_tokenize

text = "I am actively looking for Ph.D. students. and you
are a Ph.D student."
print('sent_Tokenization' :',sent_tokenize(text))
```

Output

```
sent_Tokenization : ['I am actively looking for Ph.D. students.',
'and you are a Ph.D student.']
```

Tokenization > Word Tokenization

- *Word tokenization* is the process of dividing text into tokens based on words.
- **Limitations:** Real-world tokenization is more complex than simply removing punctuation. Removing punctuation can change the meaning, and whitespace splitting is insufficient for languages like Korean.

Decision Points During Tokenization

During tokenization, unexpected situations arise, requiring **consideration of tokenization standards.**
These choices should be made based on the intended use of the data, selecting criteria that don't negatively impact that use. For example, let's consider the problem of how to classify words containing apostrophes (') in English-speaking languages.

sample text : Don't be fooled by the dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a pastry shop.

Multiple options of breakdown of tokenization options for Don't and Jone's

e.g.
Don't:
- **Don't:** Keep the contraction as a single token.
- **Don t:** Separate the word and the apostrophe, but keep the 't' attached.
- **Dont:** Remove the apostrophe altogether, merging the words.
- **Do n't:** Separate the word, apostrophe, and 't' into individual tokens.

Jone's:
- **Jone's:** Keep the possessive form as a single token.
- **Jone s:** Separate the word and the apostrophe s.
- **Jone:** Keep only the base word, removing the possessive marker.
- **Jones:** Convert to the standard spelling of the name, removing the apostrophe.

Consideration points:

1 **Punctuation as Tokens:**
- Punctuation can be meaningful tokens in themselves.
- Example: Periods (.) can indicate sentence boundaries.

2. **Punctuation Within Words:**
- Some words inherently contain punctuation.
- Examples: m.p.h, Ph.D, AT&T.

3 **Special Characters with Meaning:**
- Dollar signs ($) and slashes (/) can convey meaning.
- Example: $45.55 (price), 01/02/06 (date).
- You might want to treat "45.55" as a single token rather than splitting it into "45" and "55".

4 **Commas in Numbers:**
- Commas are often used in numbers to group digits.
- Example: 123,456,789.

5 **Contractions and Clitics:**
- Apostrophes (') in English often represent contractions (shortened forms of words).
- Examples: "what're" (what are), "we're" (we are), "I'm" (I am).
- "re" and "m" in these examples are called "clitics" (the contracted part).
- Tokenization may involve expanding these contractions.

6 **Words with Internal Spaces:**
- Some words or phrases contain spaces within them but are treated as single units.
- Examples: "New York", "rock 'n' roll".
- Tokenization needs to be able to recognize these as single tokens if the application requires it.

Penn Treebank Tokenization rules

1 **Hyphenated words are kept as one token.**
2 **Apostrophe-attached clitics (like in "doesn't") are separated.**

Input

"Starting a home-based restaurant may be an ideal. it doesn't have a food chain or restaurant of their own."

Code

```
from nltk.tokenize import TreebankWordTokenizer
tokenizer = TreebankWordTokenizer()
text = "Starting a home-based restaurant may be an ideal. it
doesn't have a food chain or restaurant of their own."
print('treebankTokenizer :',tokenizer.tokenize(text))
```

Output

```
treebankTokenizer : ['Starting', 'a', 'home-based', 'restaurant',
'may', 'be', 'an', 'ideal.', 'it', 'does', "n't", 'have', 'a', 'food',
'chain', 'or', 'restaurant', 'of', 'their', 'own', '.']
```

★Check out
how NLTK's word_tokenize and WordPunctTokenizer handle apostrophes in tokenization

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.tokenize import WordPunctTokenizer
```

word_ tokenize:

```
print('word_tokenize', tokenize("Don't be fooled by the dark
sounding name, Mr. Jone's Orphanage is as cheery as cheery
goes for a pastry shop."))
```

Output:

```
word_tokenize,: ['Do', "n't", 'be', 'fooled', 'by', 'the', 'dark', 'sounding',
'name', ',', 'Mr.', 'Jone', "'s", 'Orphanage', 'is', 'as', 'cheery', 'as', 'cheery',
'goes', 'for', 'a', 'pastry', 'shop', '.']
```

wordPunctTokenizer:

```
print('wordPunctTokenizer:',WordPunctTokenizer().tokenize("D
on't be fooled by the dark sounding name, Mr. Jone's
Orphanage is as cheery as cheery goes for a pastry shop."))
```

Output:

```
['Don', "'", 't', 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr', '.',
'Jone', "'", 's', 'Orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'goes', 'for', 'a',
'pastry', 'shop', '.']
```

Compare:
- **word_tokenize:** This tokenizer splits "Don't" into "Do" and "n't", and keeps "Jone's" as "Jone's". It attempts to handle contractions in a somewhat intelligent way.
- **WordPunctTokenizer:** This tokenizer splits "Don't" into "Don", "'", and "t", and "Jone's" into "Jone", "'", and "s". It separates *all* punctuation into separate tokens.
•

Learning:
word_tokenize offers a more sophisticated approach to handling contractions, while WordPunctTokenizer provides a more granular separation of words and punctuation. The choice of which tokenizer to use depends on the specific needs of your application. If you need to preserve contractions, word_tokenize is a better choice. If you need to analyze punctuation separately, WordPunctTokenizer is more suitable.

Can consider Keras' text_to_word_sequence

```
print('keras_tokenization :',text_to_word_sequence("Don't be fooled by the
dark sounding name, Mr. Jone's Orphanage is as cheery as cheery goes for a
pastry shop."))
```

```
['Don', "'", 't', 'be', 'fooled', 'by', 'the', 'dark', 'sounding', 'name', ',', 'Mr', '.', 'Jone', "'", 's',
'Orphanage', 'is', 'as', 'cheery', 'as', 'cheery', 'goes', 'for', 'a', 'pastry', 'shop', '.']
```