Think of:
How words are pronounced

Sound patterns (like rhyming, stress, syllables)

Practical 03: Phonological and Morphological Analysis

```
!pip install epitran
```

Code Explanation:

This command installs the **epitran** library.

By using epitran, we can convert words to their IPA (International Phonetic Alphabet) representations, allowing us to examine features such as consonant clusters, vowels, stress patterns, and phoneme structures. This sets the foundation for the phonological part of the analysis.

```
!pip install panphon
```

Code Explanation:

This code installs the **panphon** library. It is used to work with sounds of words written in IPA. It helps compare how similar or different two words sound by looking at their phonemes.

```
!pip install pronouncing
```

Code Explanation:

This code Installs the pronouncing library, which helps work with the CMU Pronouncing Dictionary.
This library allows us to find pronunciations, rhymes, syllable counts, and phonetic patterns of English words. It's useful for tasks like rhyme detection and phonological analysis in NLP.

1. Phonetic Transcription Using Pronouncing

```
import pronouncing

words = ["phonetics", "phonology", "morphology", "analysis", "transcription"]
transcriptions = {}

for w in words:
    phones = pronouncing.phones_for_word(w)
    if phones:
        transcriptions[w] = phones[0]
    else:
        transcriptions[w] = "No transcription found"
print(transcriptions)
```

Code Explanation:

- import pronouncing: Loads the pronouncing library to work with phonetic data.
- words = [...]: Creates a list of five English words for analysis.
- transcriptions = {}: Initializes an empty dictionary to store results.
- The for loop goes through each word:
    - phones_for_word(w) looks up the word in the CMU Pronouncing Dictionary.
    - If a pronunciation is found, the first transcription is stored. (There might be more than one pronunciation for a word, so phones[0] picks the first one.)
    - If the word is not found in the dictionary, it stores "No transcription found"
- print(transcriptions): Shows the final phonetic transcriptions for all words.

Phonological analysis:

This code helps to extract the phonetic structure of words. Using the CMU Pronouncing Dictionary, we can see how each word is broken down into phonemes. This information is useful in phonological analysis, such as identifying stressed syllables, vowel/consonant patterns, and sound structures.

Output:

```
{'phonetics': 'F AH0 N EH1 T IH0 K S', 'phonology': 'F AH0 N AA1 L AH0 JH IY2', 'morphology': 'M AO0 R F AA1 L AH0 JH IY0', 'analysis': 'AH0 N AE1 L AH0
S AH0 S', 'transcription': 'T R AE2 N S K R IH1 P SH AH0 N'}
```

2. Phonological Feature Extraction with PanPhon

```python
import panphon

# Minimal ARPAbet to IPA mapping (extend as needed)
arpabet_to_ipa = {
    'P': 'p', 'T': 't', 'K': 'k',
    'B': 'b', 'D': 'd', 'G': 'g',
    'AH': 'ə', 'EH': 'ɛ', 'IY': 'i', 'S': 's'
    # You can add more ARPAbet-to-IPA mappings here
}

def arpabet_to_ipa_conversion(arpabet_str):
    ipa_str = ''
    for phoneme in arpabet_str.split():
        # Remove stress digits like "0" or "1"
        base = ''.join([c for c in phoneme if not c.isdigit()])
        ipa_str += arpabet_to_ipa.get(base, '')  # Add empty if phoneme not found
    return ipa_str

# Create a PanPhon FeatureTable object
ft = panphon.FeatureTable()

# Generate IPA features for words that have valid transcriptions
features = {
    w: ft.word_fts(arpabet_to_ipa_conversion(t))
    for w, t in transcriptions.items()
    if t != "No transcription found"
}

print(features)
```

```
import panphon

# Basic ARPAbet to IPA mapping
arpabet_to_ipa = {
    'P': 'p', 'T': 't', 'K': 'k',
    'B': 'b', 'D': 'd', 'G': 'g',
   'AH': 'ˆ', 'EH': 'e', 'IY': 'i', 'S': 's'
}

# Convert ARPAbet to IPA
def convert_to_ipa(arpabet_str):
    ipa = ''
    for phoneme in arpabet_str.split():
        phoneme = ''.join(c for c in phoneme if not c.isdigit())  # remove stress digits
        ipa += arpabet_to_ipa.get(phoneme, '')  # skip unknown phonemes
    return ipa

# PanPhon feature extractor
ft = panphon.FeatureTable()

# Get features for valid IPA transcriptions
features = {
    word: ft.word_fts(convert_to_ipa(transcription))
    for word, transcription in transcriptions.items()
    if transcription != "No transcription found"
}

print(features)
```

Code Explanation:

- import panphon: Loads the PanPhon library, which helps extract detailed p features from IPA symbols.
- arpabet_to_ipa: This is a dictionary that maps ARPAbet phonemes (used in CMU dictionary) to IPA symbols.
- arpabet_to_ipa_conversion():
    - This function takes a string of ARPAbet phonemes.
    - It removes digits (like 0 or 1) which represent stress.
    - it converts each cleaned phoneme to its IPA equivalent using the arpabet_to_ipa dictionary.

- FeatureTable(): Creates an object that can extract phonological features from IPA strings. These features include:
    - Voicing (Tells us if our vocal cords vibrate when we say a sound),
    - Place of articulation (Describes where in our mouth or throat the sound is made),
    - Manner of articulation (Explains how the air flows when we say a sound.),
    - Nasality (Tells if the air comes out through your nose when saying a sound), and more.
- The features dictionary:

    - for w, t in transcriptions.items() - This goes through each word (w) and its ARPAbet transcription (t) in the transcriptions dictionary we created earlier.
    - Converts each transcription into IPA using arpabet_to_ipa_conversion.
    - Uses ft.word_fts() to generate a feature matrix — which is a list of binary values representing the phonological properties of each sound in the word.
- print(features): Shows the extracted phonological features. This lets us compare the internal sound patterns of different words.

Output:

```
{'phonetics': 'F AH0 N EH1 T IH0 K S', 'phonology': 'F AH0 N AA1
L AH0 JH IY2', 'morphology': 'M AO0 R F AA1 L AH0 JH IY0',
'analysis': 'AH0 N AE1 L AH0 S AH0 S', 'transcription': 'T R AE2
N S K R IH1 P SH AH0 N'}
```

Phonological Analysis:

This code focuses on phonological analysis by converting phonetic transcriptions from ARPAbet to IPA and then analyzing their sound features. The IPA form represents how a word is pronounced, and PanPhon helps break it down into phonological traits such as voicing, place of articulation, and manner of articulation. This helps us understand the sound pattern and structure of the word.

3. PhonemeFrequencyAnalysis

```
from collections import Counter

# Combine all transcriptions into a single string (without spaces)
all_phones = ''.join([t.replace(' ', '') for t in transcriptions.values() if t != "No transcription found"])

phoneme_counts = Counter(all_phones)

print(phoneme_counts)
```

Code Explanation:

- from collections import Counter: Imports Python's Counter tool to count how many times each phoneme appears.
- all_phones = ''.join([...]):
    - Takes all transcriptions from the transcriptions dictionary.
    - Removes spaces between phonemes (e.g., 'P AH0 N EH1 T IH0 K S' becomes 'PAH0NEH1TIH0KS'),
    - Joins all the phoneme strings into one long string like 'PAH0NEH1TIH0KSP...'.
- Counter(all_phones):

    - Counts how many times each phoneme (letter/symbol) appears in total.
- print(phoneme_counts): Shows the frequency of each phoneme in the list of words.

Output:

```
Counter({'A': 15, 'H': 14, '0': 11, 'N': 5, '1': 5, 'S': 5, 'I': 4, 'F': 3, 'E': 3, 'L': 3, 'R': 3, 'T': 2, 'K': 2, 'J': 2, 'Y': 2, '2': 2, 'M': 1, 'O': 1, 'P': 1})
```

Phonological Analysis:

This code performs a basic phonological frequency analysis. It counts how often each phoneme (sound unit) appears in the dataset. This is useful to understand which types of sounds are common in a group of words. Phoneme frequency is important for phonology because it helps analyze pronunciation trends, compare sound systems, and build pronunciation models in natural language processing or speech technology.

4. Stress Pattern Analysis with CMU Dictionary

```
import nltk
nltk.download('cmudict')
from nltk.corpus import cmudict

d = cmudict.dict()
stress_patterns = {}

for w in words:
    if w in d:
        transcription = d[w][0]  # First pronunciation variant
        # Extract stress digits (1=primary, 0=otherwise)
        pattern = ''.join(['1' if '1' in ph else '0' for ph in transcription])
        stress_patterns[w] = pattern

print(stress_patterns)
```

Code Explanation:

- import nltk: Loads the NLTK library
- nltk.download('cmudict'): Downloads the CMU Pronouncing Dictionary, which provides ARPAbet phonetic transcriptions with stress information.
- from nltk.corpus import cmudict: Imports the CMU dictionary from NLTK.
- d = cmudict.dict(): Loads the CMU dictionary into variable d, where each word maps to a list of possible pronunciations (each as a list of phonemes).
- stress_patterns = {}: Prepares an empty dictionary to store stress patterns for each word.
- for w in words: Loops through the list of target words.
- if w in d: Checks if the word is available in the dictionary.

- transcription = d[w][0]: Takes the first available transcription of the word.
- pattern = ''.join(['1' if '1' in ph else '0' for ph in transcription]):
    - For each phoneme, it checks whether it has a primary stress (1).
    - It builds a string like '010', representing where the stress falls in the word.
- print(stress_patterns): Displays each word and its corresponding stress pattern.

Output:

```
[nltk_data] Downloading package cmudict to /root/nltk_data...
[nltk_data]   Unzipping corpora/cmudict.zip.
{'phonetics': '00010000', 'phonology': '00010000', 'morphology': '000010000', 'analysis': '00100000', 'transcription': '000000010000'}
```

Phonological analysis:

This code focuses on phonological stress analysis. Stress refers to how strongly a syllable is pronounced in a word. Using the CMU Pronouncing Dictionary, this code identifies where primary stress (1) appears among the syllables in a word.

5. Vowel and Consonant Analysis

```
from collections import Counter

vowels = set('aeiou')
phonemes = ''.join([t.replace(' ', '').lower() for t in transcriptions.values() if t != "No transcription found"])

vowel_count = Counter([ch for ch in phonemes if ch in vowels])
consonant_count = Counter([ch for ch in phonemes if ch not in vowels])

print("Vowels:", vowel_count)
print("Consonants:", consonant_count)
```

Code Explanation:

- from collections import Counter: Imports a tool to count how often each letter appears.
- vowels = set('aeiou'): Defines which letters are considered vowels.
- phonemes = ''.join([t.replace(' ', '').lower() for t in transcriptions.values() if t != "No transcription found"]):
    - Joins all transcriptions (ignoring ones marked “No transcription found”).
    - Removes spaces and converts everything to lowercase for consistency.
- vowel_count = Counter([ch for ch in phonemes if ch in vowels]):
    - Counts how many times each vowel (a, e, i, o, u) appears.
- consonant_count = Counter([ch for ch in phonemes if ch not in vowels])
    - Counts how many times each letter that is a consonant appear.

Output:

```
Vowels: Counter({'a': 15, 'i': 4, 'e': 3, 'o': 1})
Consonants: Counter({'h': 14, '0': 11, 'n': 5, '1': 5, 's': 5, 'f': 3, 'l': 3, 'r': 3, 't': 2, 'k': 2, 'j': 2, 'y': 2, '2': 2, 'm': 1, 'p': 1})
```

Phonological analysis:

The code tries to find out how many vowels and consonants are used in the given word transcriptions. This helps understand the sound structure of the words (how different sounds, like vowels and consonants, are arranged and used in words), which is useful in studying how words are spoken and how sounds are used in a language.

6. Phonological Rule Application

```
voiceless_to_voiced = {'p': 'b', 't': 'd', 'k': 'g'}

def apply_voicing_rule(word):
    return ''.join([voiceless_to_voiced.get(ch, ch) for ch in word])

new_transcriptions = {
    w: apply_voicing_rule(t.replace(' ', ''))
    for w, t in transcriptions.items()
    if t != "No transcription found"
}

print(new_transcriptions)
```

Code Explanation:

- voiceless_to_voiced = {'p': 'b', 't': 'd', 'k': 'g'}: This defines a basic rule that changes voiceless consonants (p, t, k) to their voiced counterparts (b, d, g).
- apply_voicing_rule(word): This function goes through each letter in the given word. If the letter is voiceless (like p), it replaces it with the voiced version (like b), using the dictionary.
- new_transcriptions = {...}: For each word with a valid transcription:
    - It removes spaces from the phoneme string.
    - Applies the voicing rule using the function.
- print(new_transcriptions): Displays the updated transcriptions with voicing changes.

Output:

```
{'phonetics': 'FAH0NEH1TIH0KS', 'phonology': 'FAH0NAA1LAH0JHIY2', 'morphology': 'MAO0RFAA1LAH0JHIY0', 'analysis': 'AH0NAE1LAH0SAH0S', 'transcription': 'TRAE2NSKRIH1PSHAH0N'}
```

Phonological analysis:

This code does a simple phonological analysis by changing some voiceless sounds like *p, t,* and *k* into their voiced forms *b, d,* and *g*. This shows how sounds can change in speech, which is a common rule in many languages. By applying this rule, the code helps us understand how sounds are used and how words might be pronounced differently in real speech.

7. Morphological Analysis

7.1. Text Tokenization and Stopword Removal

```
import nltk

nltk.download('gutenberg')
nltk.download('punkt')
nltk.download('stopwords')

from nltk.corpus import gutenberg, stopwords
from nltk.tokenize import word_tokenize

text = gutenberg.raw('austen-emma.txt')
words = word_tokenize(text)

stop_words = set(stopwords.words('english'))

filtered = [w for w in words if w.lower() not in stop_words]

print(filtered[:50])
```

Code Explanation:

- nltk.download(...): Downloads required datasets
    - *Gutenberg* collection (contains books like *Emma* by Jane Austen)
    - *punkt* (tokenizer)
    - *stopwords* (common English words like "the", "is", "and")
- from nltk.corpus import gutenberg, stopwords: Imports books and stopword lists.
- from nltk.tokenize import word_tokenize: Allows breaking the text into words.
- text = gutenberg.raw('austen-emma.txt'): Loads the full text of *Emma* by Jane Austen.
- words = word_tokenize(text): Splits the text into individual words (tokens).
- stop_words = set(stopwords.words('english')): Gets a list of common English stopwords.
- filtered = [w for w in words if w.lower() not in stop_words]: Filters out common stopwords from the text.
- print(filtered[:50]): Prints the first 50 words after removing stopwords.

Output:

```
['[', 'Emma', 'Jane', 'Austen', '1816', ']', 'VOLUME', 'CHAPTER', 'Emma', 'Woodhouse', ',', 'handsome', ',', 'clever', ',', 'rich', ',', 'comfortable',
'home', 'happy', 'disposition', ',', 'seemed', 'unite', 'best', 'blessings', 'existence', ';', 'lived', 'nearly', 'twenty-one', 'years', 'world', 'littl
e', 'distress', 'vex', '.', 'youngest', 'two', 'daughters', 'affectionate', ',', 'indulgent', 'father', ';', ',', 'consequence', 'sister', "'s", 'marria
ge']
```

Morphological analysis

This preprocessing step is essential before morphological analysis, which studies word structure and formation (eg: root + suffix). The code first breaks the text into individual words using tokenization. Then, it removes common stopwords like "the", "is", and "and" that don't add much meaning. After that, we retain content words which often include morphemes (prefixes, suffixes, stems).

7.2. Stemming and Lemmatization

```
from nltk.stem import PorterStemmer, WordNetLemmatizer
nltk.download('wordnet')

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

stemmed = [stemmer.stem(w) for w in filtered]
lemmatized = [lemmatizer.lemmatize(w) for w in filtered]

print("Stemmed words:", stemmed[:20])
print("Lemmatized words:", lemmatized[:20])
```

Code Explanation:

- from nltk.stem import PorterStemmer, WordNetLemmatizer: Imports tools to reduce words to their base forms.
- nltk.download('wordnet'): Downloads the WordNet database for lemmatization.
- stemmer = PorterStemmer(): Creates a stemmer object (removes suffixes from words).
- lemmatizer = WordNetLemmatizer(): Creates a lemmatizer object (returns the base dictionary form of a word).
- stemmed = [stemmer.stem(w) for w in filtered]: Applies stemming to each filtered word.
- lemmatized = [lemmatizer.lemmatize(w) for w in filtered]: Applies lemmatization to each filtered word.
- Print statemets: Shows the first 20 stemmed and lemmatized words.

Output:

```
Stemmed words: ['[', 'emma', 'jane', 'austen', '1816', ']', 'volum', 'chapter', 'emma', 'woodhous', ',', 'handsom', ',', 'clever', ',', 'rich', ',', 'co
mfort', 'home', 'happi']
Lemmatized words: ['[', 'Emma', 'Jane', 'Austen', '1816', ']', 'VOLUME', 'CHAPTER', 'Emma', 'Woodhouse', ',', 'handsome', ',', 'clever', ',', 'rich',
',', 'comfortable', 'home', 'happy']
```

Morphological analysis:

This code performs morphological analysis, which is the study of word structure. It looks at morphemes, the smallest units of meaning in a word.

- Stemming (removes endings like *-ing*, *-ed*) cuts off word endings, which helps identify the root of a word but may not always produce real words.
- Lemmatization finds the actual base form (lemma) of the word using vocabulary and grammar rules.

This helps us to extract meaning from variations of a word to study patterns in grammar, tenses, and word formation.

7.3. Morphological Complexity and Affix Analysis

```
from collections import Counter

lengths = [len(w) for w in filtered]

print("Average word length:", sum(lengths) / len(lengths))

prefixes = Counter([w[:3] for w in filtered if len(w) > 3])
suffixes = Counter([w[-3:] for w in filtered if len(w) > 3])

print("Top prefixes:", prefixes.most_common(10))
print("Top suffixes:", suffixes.most_common(10))
```

Code Explanation:

- from collections import Counter: Imports the Counter class, which is used to count how often items (like words or letters) appear.
- lengths = [len(w) for w in filtered]: Creates a list of word lengths by calculating the number of characters in each word from the filtered list.
- print("Average word length:", sum(lengths) / len(lengths)): Calculates and prints the average word length by dividing the total character count by the number of words.
- prefixes = Counter([w[:3] for w in filtered if len(w) > 3]): Extracts the first 3 letters (prefix) from each word longer than 3 characters and counts how often each prefix appears in the text.
- suffixes = Counter([w[-3:] for w in filtered if len(w) > 3]): Extracts the last 3 letters (suffix) from each word longer than 3 characters and counts how often each suffix appears in the text.
- print("Top prefixes:", prefixes.most_common(10)): Prints the 10 most common prefixes based on frequency.

- print("Top suffixes:", suffixes.most_common(10)): Prints the 10 most common suffixes based on frequency.

Output:

```
Average word length: 4.591895359748741
Top prefixes: [('cou', 1001), ('thi', 947), ('con', 918), ('Emm', 865), ('eve', 843), ('com', 825), ('wou', 820), ('Mrs', 668), ('Har', 667), ('mus', 610)]
Top suffixes: [('ing', 4227), ('uld', 1654), ('ion', 1473), ('ent', 1083), ('ght', 1012), ('her', 972), ('mma', 860), ('nce', 847), ('ton', 842), ('ted', 808)]
```

Morphological analysis:

This code demonstrates morphological analysis by examining how words are formed and structured. It first calculates the average length of words to understand their general complexity. Then, it extracts and counts the most common three-letter prefixes and suffixes from the words in the text. By identifying these frequent patterns, we gain insights into how new words are created or modified in English, which is the core idea of morphological analysis.

7.4. Syllable and Compound Word Analysis

```
import pyphen

dic = pyphen.Pyphen(lang='en')
syllables = [len(dic.inserted(w).split('-')) for w in filtered]

print("Average syllables per word:", sum(syllables) / len(syllables))
```

Code Explanation:

- import pyphen: Imports the pyphen library, which helps to split words into syllables.
- pyphen.Pyphen(lang='en'): Creates a pyphen dictionary object for English to find where syllables break in words.
- Syllables = [len(dic.inserted(w).split('-')) for w in filtered]: For each word in the filtered list, inserts hyphens at syllable breaks, splits the word at hyphens, and counts the number of syllables in the word.
- print("Average syllables per word:", sum(syllables) / len(syllables)): This calculates the average number of syllables per word and prints it.

Output:

```
Average syllables per word: 1.4706034433230557
```

Morphological Analysis:

This code helps analyze the internal structure of words by estimating the number of syllables, which is a key component in morphology. Syllables often align with morphemes, the smallest units of meaning, though not always exactly. By counting syllables using the pyphen library, we get an idea of how complex or simple a word might be morphologically. For example, longer

words with more syllables often contain multiple morphemes, such as prefixes, roots, and suffixes. This kind of analysis is useful in studying word formation, language patterns, and text complexity.

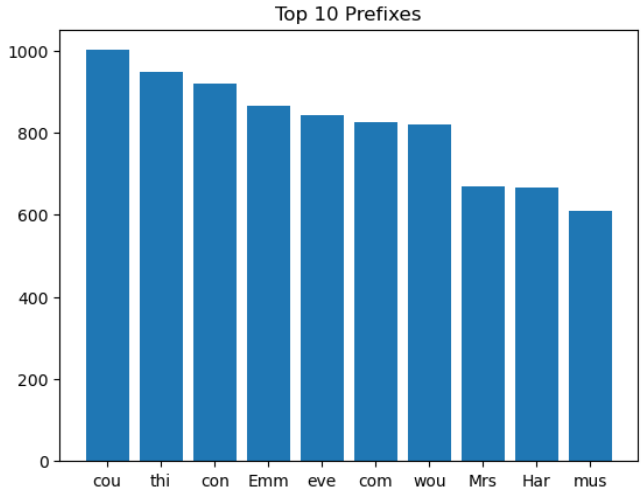
7.5. Visualization

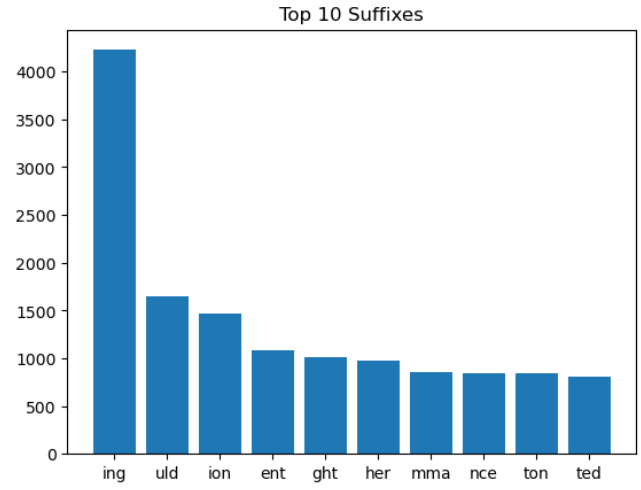
```
import matplotlib.pyplot as plt
labels, values = zip(*prefixes.most_common(10))
plt.bar(labels, values)
plt.title("Top 10 Prefixes")
plt.show()
labels, values = zip(*suffixes.most_common(10))
plt.bar(labels, values)
plt.title("Top 10 Suffixes")
plt.show()
```

Code Explanation:

- import matplotlib.pyplot as plt: Imports the matplotlib library for creating visual plots.
- labels, values = zip(*prefixes.most_common(10)): Extracts the top 10 most common prefixes and separates them into two lists, labels for the prefixes and values for their counts.
- labels, values = zip(*suffixes.most_common(10)): Extracts the top 10 most common suffixes and separates them into two lists, labels for the prefixes and values for their counts.
- plt.bar(labels, values): Draws a bar chart using the prefix labels and their frequencies.
- plt.title("Top 10 Prefixes"): Adds a title to the prefix bar chart.
- plt.show(): Displays the prefix bar chart.

Output:





This code provides a visual analysis of word structure by showing the most frequent prefixes (beginnings of words) and suffixes (endings of words) in the filtered text. In morphology, prefixes and suffixes are called affixes. By identifying the most common affixes, we can understand how words are formed, modified, or inflected in English.