```
import nltk
from nltk import word_tokenize, pos_tag

# Download necessary NLTK resources (only required once)
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Input text
text = "NLTK is a powerful library in Python for working with human language data."

# Tokenize the text into words
words = word_tokenize(text)

# Perform part-of-speech tagging
pos_tags = pos_tag(words)

# Display the POS tags
print("Word with POS tags:")
for word, tag in pos_tags:
    print(f"{word}: {tag}")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
Word with POS tags:
NLTK: NNP
is: VBZ
a: DT
powerful: JJ
library: NN
in: IN
Python: NNP
for: IN
working: VBG
with: IN
human: JJ
language: NN
data: NNS
.: .
```

```
import nltk
from nltk.corpus import treebank
from collections import defaultdict, Counter
import math

# Download the NLTK treebank corpus if not already downloaded
nltk.download('treebank')

# Load the POS-tagged sentences from the NLTK treebank corpus
tagged_sentences = treebank.tagged_sents()

# Initialize dictionaries to store transition and emission probabilities
tag_transitions = defaultdict(Counter)
tag_emissions = defaultdict(Counter)
tag_counts = Counter()

# Calculate transition and emission probabilities
for sentence in tagged_sentences:
    previous_tag = "<START>"
    for word, tag in sentence:
        # Increment the transition and emission counts
        tag_transitions[previous_tag][tag] += 1
        tag_emissions[tag][word.lower()] += 1
        tag_counts[tag] += 1
        previous_tag = tag
    # Account for transition from last tag to end
    tag_transitions[previous_tag]["<END>"] += 1

# Convert counts to log-probabilities
def compute_log_probabilities(counter_dict):
    log_prob_dict = {}
    for key, counter in counter_dict.items():
        total_count = sum(counter.values())
        log_prob_dict[key] = {k: math.log(v / total_count) for k, v in counter.items()}
        return log_prob_dict

    transition_probs = compute_log_probabilities(tag_transitions)
    emission_probs = compute_log_probabilities(tag_emissions)

    # Set of unique tags
    tags = set(tag_counts.keys())

    # Step 3: Viterbi Algorithm for POS Tagging
    def viterbi(sentence, transition_probs, emission_probs, tags):
        # Initialization
        viterbi_probs = [{}]
        backpointer = [{}]

        for tag in tags:
            viterbi_probs[0][tag] = transition_probs.get("<START>", {}).get(tag, float("-inf")) + \
                                    emission_probs.get(tag, {}).get(sentence[0].lower(), float("-inf"))
            backpointer[0][tag] = "<START>"

        # Recursion
        for i in range(1, len(sentence)):
            viterbi_probs.append({})
            backpointer.append({})
            for tag in tags:
                max_prob, best_previous_tag = max(
                    (viterbi_probs[i-1][prev_tag] +
                     transition_probs.get(prev_tag, {}).get(tag, float("-inf")) +
                     emission_probs.get(tag, {}).get(sentence[i].lower(), float("-inf")),
                     prev_tag) for prev_tag in tags
                )
                viterbi_probs[i][tag] = max_prob
                backpointer[i][tag] = best_previous_tag

        # Termination
        max_prob, best_last_tag = max((viterbi_probs[-1][tag] + transition_probs.get(tag, {}).get("<END>", float("-inf")), tag) for tag in tags)

        # Backtrack to find the best path
        best_path = [best_last_tag]
        for i in range(len(sentence) - 1, 0, -1):
            best_path.insert(0, backpointer[i][best_path[0]])

        return list(zip(sentence, best_path))

    # Test the Viterbi POS Tagger
    test_sentence = ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
    tagged_sentence = viterbi(test_sentence, transition_probs, emission_probs, tags)
    print(tagged_sentence)
```

```
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
[('The', '``'), ('quick', '``'), ('brown', '``'), ('fox', '``'), ('jumps', '``'), ('over', '``'), ('the', '``'), ('lazy', '``'), ('dog',
```

```
    import re

    # Sample sentence to be tagged
    sentence = "The quick brown fox jumps over the lazy dog"

    # Split the sentence into words
    words = sentence.split()

    # Initialize an empty list to store (word, tag) tuples
    tagged_sentence = []

    # Define basic POS tagging rules using regular expressions
    for word in words:
        if re.match(r'^[A-Z][a-z]*$', word):
            # Rule: Capitalized words (like proper nouns) are tagged as 'NNP' (Proper Noun)
            tag = 'NNP'
        elif re.match(r'.*ing$', word):
            # Rule: Words ending in 'ing' are tagged as 'VBG' (Gerund/Present Participle Verb)
            tag = 'VBG'
        elif re.match(r'.*ed$', word):
            # Rule: Words ending in 'ed' are tagged as 'VBD' (Past Tense Verb)
            tag = 'VBD'
    elif re.match(r'.*es$', word) or re.match(r'.*s$', word):
        # Rule: Words ending in 'es' or 's' are tagged as 'NNS' (Plural Noun)
        tag = 'NNS'
    elif re.match(r'.*ly$', word):
        # Rule: Words ending in 'ly' are tagged as 'RB' (Adverb)
        tag = 'RB'
    elif re.match(r'.*ous$|.*ful$|.*able$|.*ic$', word):
        # Rule: Words with common adjective suffixes are tagged as 'JJ' (Adjective)
        tag = 'JJ'
    elif re.match(r'the|a|an$', word.lower()):
        # Rule: 'the', 'a', and 'an' are tagged as 'DT' (Determiner)
        tag = 'DT'
    elif re.match(r'^[0-9]+$', word):
        # Rule: Words that are purely numeric are tagged as 'CD' (Cardinal Number)
        tag = 'CD'
    else:
        # Default rule: tag as 'NN' (Noun) if no other rule applies
        tag = 'NN'

    # Append the tagged word to the result list
    tagged_sentence.append((word, tag))

# Print the tagged sentence
print(tagged_sentence)
```

```
[('The', 'NNP'), ('quick', 'NN'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'NNS'), ('over', 'NN'), ('the', 'DT'), ('lazy', 'NN'), ('dog
```

```
import re

# Sample sentence to be tagged
sentence = "The quick brown fox jumps over the lazy dog"

# Split the sentence into words
words = sentence.split()

# Step 1: Initial Tagging - Start by tagging each word as 'NN' (Noun)
tagged_sentence = [(word, 'NN') for word in words]

# Step 2: Define Transformation Rules
# Each rule will have a condition and the action to apply if the condition is met
# Format: (pattern, current_tag, new_tag)
rules = [
    (r'^[Tt]he$', 'NN', 'DT'),        # Rule: "The" or "the" is a determiner (DT)
    (r'.*ing$', 'NN', 'VBG'),         # Rule: Words ending with "ing" are gerunds/present participles (VBG)
    (r'.*ed$', 'NN', 'VBD'),          # Rule: Words ending with "ed" are past tense verbs (VBD)
    (r'.*ly$', 'NN', 'RB'),           # Rule: Words ending with "ly" are adverbs (RB)
    (r'^[A-Z][a-z]*$', 'NN', 'NNP'),  # Rule: Capitalized words are proper nouns (NNP)
]

# Step 3: Apply Transformation Rules
for i, (word, tag) in enumerate(tagged_sentence):
    for pattern, current_tag, new_tag in rules:
        # If the word matches the pattern and has the current tag, apply the new tag
        if re.match(pattern, word) and tag == current_tag:
            tagged_sentence[i] = (word, new_tag)
            break  # Stop once a rule has been applied to avoid conflicting transformations

# Print the tagged sentence after applying transformation rules
print(tagged_sentence)
```

```
[('The', 'DT'), ('quick', 'NN'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'NN'), ('over', 'NN'), ('the', 'DT'), ('lazy', 'NN'), ('dog',
```