```
import pandas as pd

# 1. Define the raw POS-tagged abstract sentences
raw_tagged_sentences = [
    "The/DT quick/JJ brown/JJ fox/NN jumps/VBZ over/IN the/DT lazy/JJ dog/NN.",
    "A/DT cat/NN sat/VBD on/IN the/DT mat/NN.",
    "The/DT sun/NN shines/VBZ brightly/RB today/NN.",
    "He/PRP runs/VBZ fast/RB in/IN the/DT morning/NN.",
    "She/PRP is/VBZ reading/VBG a/DT book/NN about/IN history/NN."
]

# 2. Create an empty list to store the processed data
tagged_sentences = []

# 3. Iterate through each raw tagged sentence and process it
for sentence in raw_tagged_sentences:
    # Split the sentence string into individual word/tag pairs
    word_tag_pairs = sentence.split()

    current_sentence_tuples = []
    for pair in word_tag_pairs:
        # Split each word/tag pair into word and tag
        # Handle cases where a word might contain '/', taking only the last '/' a
        parts = pair.rsplit('/', 1)
        if len(parts) == 2:
            word, tag = parts
        else:
            # If no tag is found, assign a default or handle as error
            word = parts[0]
            tag = 'UNKNOWN'

        # Create a tuple (word, tag)
        current_sentence_tuples.append((word, tag))

    # Append the list of (word, tag) tuples for the current sentence to the tagg
    tagged_sentences.append(current_sentence_tuples)

# 4. Display the first few processed sentences to verify the format
print("First 3 processed sentences:")
for i, sentence in enumerate(tagged_sentences[:3]):
    print(f"Sentence {i+1}: {sentence}")

# You can also check the total number of processed sentences
print(f"\nTotal number of processed sentences: {len(tagged_sentences)}")
```

```
First 3 processed sentences:
Sentence 1: [('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'), ('fox', 'NN'), ('j
Sentence 2: [('A', 'DT'), ('cat', 'NN'), ('sat', 'VBD'), ('on', 'IN'), ('the', '
```

```
Sentence 3: [('The', 'DT'), ('sun', 'NN'), ('shines', 'VBZ'), ('brightly', 'RB')

Total number of processed sentences: 5
```

```
from collections import defaultdict

# 1. Initialize dictionaries
initial_tag_counts = defaultdict(int)
tag_counts = defaultdict(int)
tag_transitions = defaultdict(lambda: defaultdict(int))
emission_counts = defaultdict(lambda: defaultdict(int))

# 2. Iterate through each sentence in the tagged_sentences list
for sentence in tagged_sentences:
    # a. For the first word-tag pair in the current sentence
    if sentence:
        first_word, first_tag = sentence[0]
        initial_tag_counts[first_tag] += 1

    # b. For each word-tag pair (word, tag) in the current sentence
    for i, (word, tag) in enumerate(sentence):
        # i. Increment the count for tag in tag_counts
        tag_counts[tag] += 1

        # ii. Increment the count for word under its tag in emission_counts
        emission_counts[tag][word] += 1

        # c. For each consecutive pair of tags (tag_i, tag_j)
        if i > 0:
            prev_tag = sentence[i-1][1]
            current_tag = tag
            # i. Increment the count for the transition from tag_i to tag_j
            tag_transitions[prev_tag][current_tag] += 1

# 3. Calculate the probabilities

# a. Create an initial_probabilities dictionary
total_sentences = len(tagged_sentences)
initial_probabilities = {tag: count / total_sentences for tag, count in initial

# b. Create a transition_probabilities dictionary
transition_probabilities = defaultdict(lambda: defaultdict(float))
for prev_tag, next_tag_counts in tag_transitions.items():
    total_transitions_from_prev_tag = tag_counts[prev_tag]
    for next_tag, count in next_tag_counts.items():
        transition_probabilities[prev_tag][next_tag] = count / total_transitior

# c. Create an emission_probabilities dictionary
emission_probabilities = defaultdict(lambda: defaultdict(float))
for tag, word_counts in emission_counts.items():
    total_emissions_from_tag = tag_counts[tag]
```

```
        for word, count in word_counts.items():
            emission_probabilities[tag][word] = count / total_emissions_from_tag

    # 4. Display a few examples
    print("\n--- HMM Probabilities Examples ---")
    print(f"Initial probability of 'DT': {initial_probabilities.get('DT', 0):.4f}")

    # Example transitions
    print(f"Transition P('NN' | 'DT'): {transition_probabilities.get('DT', {}).get(
    print(f"Transition P('VBZ' | 'NN'): {transition_probabilities.get('NN', {}).get

    # Example emissions
    print(f"Emission P('fox' | 'NN'): {emission_probabilities.get('NN', {}).get('fo
    print(f"Emission P('quick' | 'JJ'): {emission_probabilities.get('JJ', {}).get('
    print(f"Emission P('jumps' | 'VBZ'): {emission_probabilities.get('VBZ', {}).get

    print("\nInitialization and calculation of HMM probabilities complete.")
```

```
--- HMM Probabilities Examples ---
Initial probability of 'DT': 0.6000
Transition P('NN' | 'DT'): 0.4286
Transition P('VBZ' | 'NN'): 0.5000
Emission P('fox' | 'NN'): 0.2500
Emission P('quick' | 'JJ'): 0.3333
Emission P('jumps' | 'VBZ'): 0.2500

Initialization and calculation of HMM probabilities complete.
```

```
import pandas as pd

# 1. Convert the transition_probabilities dictionary into a pandas DataFrame
transition_matrix = pd.DataFrame(transition_probabilities).fillna(0)

# 2. Reorder columns and index to ensure consistency and readability
# Get all unique tags from both keys and nested keys to ensure all tags are pre
all_tags = sorted(list(set(transition_matrix.index.tolist() + transition_matrix

transition_matrix = transition_matrix.reindex(index=all_tags, columns=all_tags,


# 3. Display the resulting transition matrix DataFrame
print("--- HMM Transition Matrix ---")
print(transition_matrix)

print("\nTransition matrix extracted and displayed successfully.")
```

```
--- HMM Transition Matrix ---
           DT    IN        JJ    NN  NN.  PRP   RB  VBD  VBG   VBZ
DT   0.000000  0.75  0.000000  0.00  0.0  0.0  0.0  0.0  1.0  0.00
IN   0.000000  0.00  0.000000  0.25  0.0  0.0  0.5  1.0  0.0  0.25
JJ   0.285714  0.00  0.333333  0.00  0.0  0.0  0.0  0.0  0.0  0.00
```

```
NN   0.428571  0.00  0.333333  0.00  0.0  0.0  0.0  0.0  0.0  0.00
NN.  0.285714  0.25  0.333333  0.00  0.0  0.0  0.5  0.0  0.0  0.00
PRP  0.000000  0.00  0.000000  0.00  0.0  0.0  0.0  0.0  0.0  0.00
RB   0.000000  0.00  0.000000  0.00  0.0  0.0  0.0  0.0  0.0  0.50
VBD  0.000000  0.00  0.000000  0.25  0.0  0.0  0.0  0.0  0.0  0.00
VBG  0.000000  0.00  0.000000  0.00  0.0  0.0  0.0  0.0  0.0  0.25
VBZ  0.000000  0.00  0.000000  0.50  0.0  1.0  0.0  0.0  0.0  0.00

Transition matrix extracted and displayed successfully.
```

```
print("--- HMM Emission Probability Examples (Top 3 for selected tags) ---")

# Select a few interesting POS tags to display
sample_tags = ['NN', 'JJ', 'VBZ', 'DT', 'IN', 'PRP']

for tag in sample_tags:
    if tag in emission_probabilities:
        print(f"\nTag: {tag}")
        # Get the emission probabilities for the current tag
        word_probs = emission_probabilities[tag]

        # Sort words by their probability in descending order
        sorted_words = sorted(word_probs.items(), key=lambda item: item[1], rev

        # Print the top N most likely words
        for i, (word, prob) in enumerate(sorted_words[:3]): # Display top 3 wor
            print(f"  '{word}': {prob:.4f}")
    else:
        print(f"\nTag: {tag} - No emission probabilities found.")

print("\nEmission probability examples extracted and displayed successfully.")
```

```
--- HMM Emission Probability Examples (Top 3 for selected tags) ---

Tag: NN
  'fox': 0.2500
  'cat': 0.2500
  'sun': 0.2500

Tag: JJ
  'quick': 0.3333
  'brown': 0.3333
  'lazy': 0.3333

Tag: VBZ
  'jumps': 0.2500
  'shines': 0.2500
  'runs': 0.2500

Tag: DT
  'the': 0.4286
  'The': 0.2857
  'A': 0.1429
```

```
Tag: IN
  'over': 0.2500
  'on': 0.2500
  'in': 0.2500

Tag: PRP
  'He': 0.5000
  'She': 0.5000

Emission probability examples extracted and displayed successfully.
```

```
print("--- Most Frequent Tag Transitions ---")

# 1. Initialize an empty list to store transitions with probabilities
frequent_transitions = []

# 2. Iterate through the transition_probabilities dictionary
for prev_tag, next_tag_probs in transition_probabilities.items():
    for next_tag, probability in next_tag_probs.items():
        frequent_transitions.append((prev_tag, next_tag, probability))

# 3. Sort the frequent_transitions list in descending order based on the probab
frequent_transitions.sort(key=lambda x: x[2], reverse=True)

# 4. Print the top N (e.g., top 10) sorted transitions
num_transitions_to_display = 10
print(f"Top {num_transitions_to_display} Transitions:")
for i, (prev_tag, next_tag, prob) in enumerate(frequent_transitions[:num_transi
    print(f"  {i+1}. {prev_tag} -> {next_tag}: {prob:.4f}")

print("\nMost frequent tag transitions identified and displayed successfully.")
```

```
--- Most Frequent Tag Transitions ---
Top 10 Transitions:
  1. VBD -> IN: 1.0000
  2. PRP -> VBZ: 1.0000
  3. VBG -> DT: 1.0000
  4. IN -> DT: 0.7500
  5. NN -> VBZ: 0.5000
  6. VBZ -> RB: 0.5000
  7. RB -> NN.: 0.5000
  8. RB -> IN: 0.5000
  9. DT -> NN: 0.4286
  10. JJ -> JJ: 0.3333

Most frequent tag transitions identified and displayed successfully.
```

```
import numpy as np

# 1. Define a new, untagged abstract sentence
new_sentence = "The study analyzes brain activity in response to stimuli."
print(f"New sentence: {new_sentence}")
```

```
# 2. Tokenize the new sentence into a list of words
new_sentence_words = [word.lower() for word in new_sentence.replace('.', '').spl
print(f"Tokenized words: {new_sentence_words}")

# Get all unique tags from the training data, including 'NN.' if present
# Ensure 'all_tags' is defined, if not, reconstruct it from emission_probabiliti
if 'all_tags' not in locals():
    all_tags = sorted(list(set(emission_probabilities.keys()) | set(tag for sub_

# Adding a potential 'START' tag if it's implicitly used by initial_probabilitie
# For simplicity, we assume initial_probabilities keys cover all starting tags

# Small constant for unknown words/transitions to avoid zero probabilities
# Using a small float to avoid issues with log(0) in other HMM implementations
# Here we are just using probabilities, so a small value like 1e-100 is fine
UNKNOWN_PROB = 1e-100

# 3. Implement the Viterbi algorithm

# Initialize Viterbi path probabilities (V) and backpointers (path)
# V[t][tag] = max probability of any path ending in 'tag' at time 't'
# path[t][tag] = previous tag in the best path ending in 'tag' at time 't'

num_words = len(new_sentence_words)

V = [{tag: -np.inf for tag in all_tags} for _ in range(num_words)] # Using log p
path = [{tag: None for tag in all_tags} for _ in range(num_words)]

# Log probabilities to prevent underflow, handle 0 probabilities with a small va
def log_safe(prob):
    return np.log(prob) if prob > 0 else -np.inf

# Initialization step for the first word
first_word = new_sentence_words[0]
for tag in all_tags:
    initial_prob = initial_probabilities.get(tag, UNKNOWN_PROB)
    emission_prob = emission_probabilities.get(tag, {}).get(first_word, UNKNOWN_
    V[0][tag] = log_safe(initial_prob) + log_safe(emission_prob)

# Iteration step for the rest of the words
for t in range(1, num_words):
    word = new_sentence_words[t]
    for current_tag in all_tags:
        max_log_prob = -np.inf
        best_prev_tag = None

        emission_prob_current_word = emission_probabilities.get(current_tag, {})
        log_emission_prob = log_safe(emission_prob_current_word)

        for prev_tag in all_tags:
            if V[t-1][prev_tag] == -np.inf: # Skip paths that are already impos
```

```
                if V[t-1][prev_tag] == -np.inf: # Skip paths that are already impossible
                    continue

                transition_prob = transition_probabilities.get(prev_tag, {}).get(current_tag, 0)
                log_transition_prob = log_safe(transition_prob)

                current_log_prob = V[t-1][prev_tag] + log_transition_prob + log_emission_prob

                if current_log_prob > max_log_prob:
                    max_log_prob = current_log_prob
                    best_prev_tag = prev_tag

            V[t][current_tag] = max_log_prob
            path[t][current_tag] = best_prev_tag

    # 4. Backtrack to reconstruct the most likely tag sequence
    predicted_tags = []

    # Find the last tag in the best path
    last_word_idx = num_words - 1
    last_tag = max(V[last_word_idx], key=V[last_word_idx].get)
    predicted_tags.append(last_tag)

    # Backtrack through the path
    for t in range(last_word_idx, 0, -1):
        last_tag = path[t][last_tag]
        predicted_tags.insert(0, last_tag)

    # 5. Print the original sentence, its tokenized words, and the predicted POS tags
    print("\n--- HMM POS Tagging Result ---")
    print(f"Original Sentence: {new_sentence}")
    print(f"Tokenized Words: {new_sentence_words}")
    print(f"Predicted POS Tags: {predicted_tags}")

    print("\nHMM applied to new sentence and tags predicted successfully.")
```

```
New sentence: The study analyzes brain activity in response to stimuli.
Tokenized words: ['the', 'study', 'analyzes', 'brain', 'activity', 'in', 'respon

--- HMM POS Tagging Result ---
Original Sentence: The study analyzes brain activity in response to stimuli.
Tokenized Words: ['the', 'study', 'analyzes', 'brain', 'activity', 'in', 'respon
Predicted POS Tags: ['DT', 'JJ', 'NN', 'VBZ', 'RB', 'IN', 'DT', 'NN', 'VBZ']

HMM applied to new sentence and tags predicted successfully.
```