Course Name: Natural Language Understanding

Assignment No. 1: Part of Speech Tagging using HMM

Submitted by:

Akaash Chatterjee
Roll No: M24CSE002

Bera Swaminath Ansuman Sabita
Roll No: M24CSE007

Raskar Varun Arun
Roll No: M24CSA026

Priyabrata Saha
Roll No: M24IRM004

Indian Institute of Technology Jodhpur

February 13, 2025

Google Colab Link

Contents

Chapter 1

Introduction

1.1 Introduction

Part-of-Speech (POS) tagging is an important task in Natural Language Processing (NLP) that involves assigning grammatical categories, such as nouns, verbs, or adjectives, to each word in a sentence. This process is required for understanding the syntax of text and is widely used in tasks such as information retrieval, machine translation, and sentiment analysis.
Hidden Markov Models (HMMs) are a popular statistical approach for POS tagging. HMMs treat POS tags as hidden states and words as observable outputs. They model the sequential nature of language by calculating two key probabilities: ***transition probabilities***, which represent the likelihood of one POS tag following another, and ***emission probabilities,*** which capture the likelihood of a word being associated with a specific POS tag. The Viterbi algorithm is commonly used with HMMs to decode the most probable sequence of tags for a given sentence.
HMM-based POS tagging is particularly effective because it considers both the context of words and their grammatical roles, enabling it to handle ambiguities in language. For instance, words with multiple possible tags (e.g., "lead" as a noun or verb) are disambiguated based on surrounding words and probabilities learned from training data. Despite its strengths, HMMs face challenges such as data sparsity and limited contextual scope due to the Markov assumption, which only considers immediate dependencies between states. However, they remain a foundational method for sequential labeling tasks in NLP.

1.2 Objective

The objective of our project is to develop and evaluate a Hidden Markov Model (HMM)-based Part-of-Speech (PoS) tagger using the English Penn Treebank (PTB) corpus. The task involves implementing three variations of HMM configurations: (a) First-Order HMM assuming the probability of a word depends only on the current tag, (b) Second-Order HMM assuming the probability depends only on the current tag, and (c) First-Order HMM assuming the probability of a word depends on both the current tag and the previous word. The dataset is split into an 80:20 ratio for training and testing, and the Viterbi algorithm is employed to compute the most probable sequence of tags for a given sentence. Performance is evaluated by calculating overall accuracy and tag-wise accuracy for both a 36-tag model and a collapsed 4-tag model, where tags are grouped into broader categories (N, V, A, O). Additionally, a default PoS tag is assigned to unseen words based on the most frequent tag in the dataset. The project aims to analyze and compare the performance of the 36-tag and 4-tag models, exploring how collapsing tags affect accuracy in light of transition and emission probabilities.

Chapter 2

Methodology

2.1 Data Loading and Processing

2.1.1 Importing Requried Libraries and Dataset Loading

We first import all required libraries, Since our penn-data file is in json format, we use the help of the json library to load it into our system.

```
[1]  1 #First we imported the necessary libraries.
     2 import matplotlib.pyplot as plt
     3 import seaborn as sns
     4 import json
     5 import numpy as np
     6 import pandas as pd
     7 from sklearn.metrics import confusion_matrix
     8 from collections import defaultdict, Counter
```

```
[2]  1 # We then load the dataset from a JSON file which we uploaded already to our colab system
     2 def load_data(file_path):
     3     with open(file_path, 'r') as f:
     4         data = json.load(f)
     5     return data
     6
     7 data = load_data('/content/penn-data.json')
```

2.1.2 Splitting our data for training and testing

We next split our data into 80% for training and 20% percent for testing according to the instructions.

```
[3]  1 # As per the instructions we then split our data into 80% train and 20% test
     2 np.random.seed(42)
     3 np.random.shuffle(data)
     4 split_idx = int(len(data) * 0.8)
     5 train_data, test_data = data[:split_idx], data[split_idx:]
```

2.1.3 Preprocessing the Dataset for POS Tagging

We preprocess the data set by converting it into a structured format suitable for Part-of-Speech (POS) tagging. We first transform the raw input into a list of (word, tag) tuples for both training and testing data. Next, we extract all unique tags and compute their frequencies, along with word occurrences in a case-insensitive manner. Finally, we determine the most frequent tag to serve as the default tag for future predictions.

```
[4]  # We then convert our loaded dataset to list of (word, tag) tuples to bring it to a suitable format
     formatted_train_data = [[(word, tag) for word, tag in zip(sent[0].split(), sent[1])] for sent in train_data]
     formatted_test_data = [[(word, tag) for word, tag in zip(sent[0].split(), sent[1])] for sent in test_data]
     train_data = formatted_train_data
     test_data = formatted_test_data
```

```
     # We then extract the words and tags and maintain their counts
     all_tags = set(tag for sent in train_data for _, tag in sent)
     tag_counts = Counter(tag for sent in train_data for _, tag in sent)
     word_counts = Counter(word.lower() for sent in train_data for word, _ in sent)
     default_tag = max(tag_counts, key=tag_counts.get)
```

2.2 Defining Necessary Functions

2.2.1 Transition Probability Functions

We define functions to compute transition probabilities for both First-Order and Second-Order Hidden Markov Models (HMMs).

- **First-Order HMM**:We compute the probability of transitioning from one tag to the next by iterating over the training data and counting occurrences of tag sequences. The probabilities are then normalized by the total transitions of each tag.
- **Second-Order HMM**: Here, we extend the transition model to consider the previous two tags when computing the probability of the next tag. This provides more contextual information and allows for a richer representation of tag dependencies.

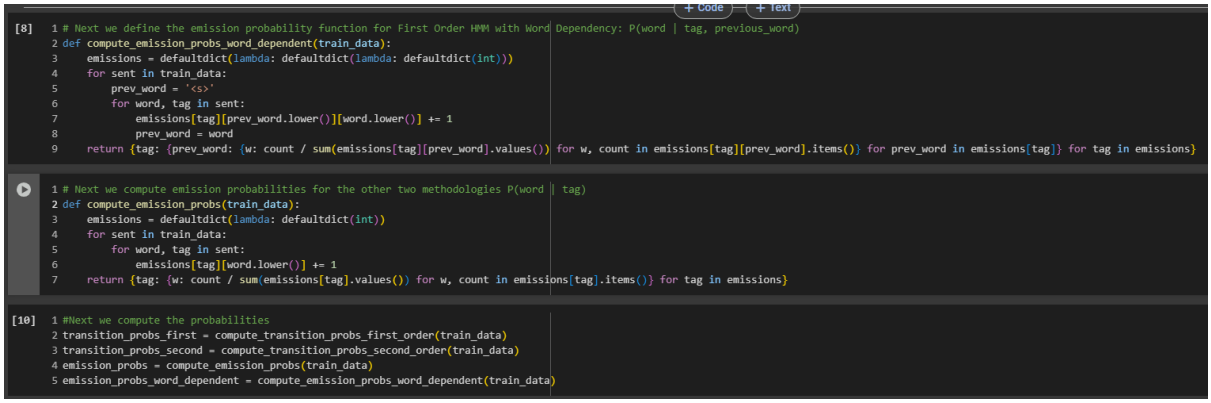
```
[6]  #First we define the function to compute transition probabilites for First Order HMM
     def compute_transition_probs_first_order(train_data):
         transitions = defaultdict(lambda: defaultdict(int))
         for sent in train_data:
             prev_tag = '<s>'
             for _, tag in sent:
                 transitions[prev_tag][tag] += 1
                 prev_tag = tag
         return {tag: {t: count / sum(transitions[tag].values()) for t, count in transitions[tag].items()} for tag in transitions}
```

```
     #Next we define the function to compute transition probabilites for Second Order HMM
     def compute_transition_probs_second_order(train_data):
         transitions = defaultdict(lambda: defaultdict(int))
         for sent in train_data:
             prev_tag1 = '<s>'
             prev_tag2 = '<s>'
             for _, tag in sent:
                 transitions[(prev_tag1, prev_tag2)][tag] += 1
                 prev_tag1, prev_tag2 = prev_tag2, tag
         return {tags: {t: count / sum(transitions[tags].values()) for t, count in transitions[tags].items()} for tags in transitions}
```

2.2.2 Emission Probability Functions

We define functions to compute emission probabilities for different methodologies in Hidden Markov Models (HMMs).

- Word-Dependent Emission Probabilities ($P(\text{word} \mid \text{tag}, \text{previous\_word})$): This method considers both the current tag and the previous word to compute the probability of a given word appearing. We iterate through the training data, tracking occurrences based on this dependency, and normalize the counts to obtain probabilities.
- Standard Emission Probabilities ($P(\text{word} \mid \text{tag})$): Here, we compute the probability of a word given a tag without considering the previous word. We count the occurrences of word-tag pairs in the training data and normalize them to derive the probability distribution for each tag.



```
[8]
# Next we define the emission probability function for First Order HMM with Word Dependency: P(word | tag, previous_word)
def compute_emission_probs_word_dependent(train_data):
    emissions = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))
    for sent in train_data:
        prev_word = '<s>'
        for word, tag in sent:
            emissions[tag][prev_word.lower()][word.lower()] += 1
            prev_word = word
    return {tag: {prev_word: {w: count / sum(emissions[tag][prev_word].values()) for w, count in emissions[tag][prev_word].items()} for prev_word in emissions[tag]} for tag in emissions}

# Next we compute emission probabilities for the other two methodologies P(word | tag)
def compute_emission_probs(train_data):
    emissions = defaultdict(lambda: defaultdict(int))
    for sent in train_data:
        for word, tag in sent:
            emissions[tag][word.lower()] += 1
    return {tag: {w: count / sum(emissions[tag].values()) for w, count in emissions[tag].items()} for tag in emissions}

[10]
#Next we compute the probabilities
transition_probs_first = compute_transition_probs_first_order(train_data)
transition_probs_second = compute_transition_probs_second_order(train_data)
emission_probs = compute_emission_probs(train_data)
emission_probs_word_dependent = compute_emission_probs_word_dependent(train_data)
```

2.3 Viterbi Algorithm

Next,we implement functions for the Viterbi Algorithm for our three methodologies.

```
[11]
#Next we define the viterbi algo for the 1st Methodology
def viterbi_first_order(sentence, transition_probs, emission_probs, tag_counts, unk_tag=default_tag):
    n = len(sentence)
    states = list(tag_counts.keys())
    viterbi = [{}]
    backpointer = [{}]

    for tag in states:
        viterbi[0][tag] = transition_probs.get('<s>', {}).get(tag, 1e-6) * emission_probs.get(tag, {}).get(sentence[0].lower(), 1e-6)
        backpointer[0][tag] = None

    for t in range(1, n):
        viterbi.append({})
        backpointer.append({})
        for tag in states:
            max_prob, prev_best = max(
                (viterbi[t-1][prev_tag] * transition_probs.get(prev_tag, {}).get(tag, 1e-6) * emission_probs.get(tag, {}).get(sentence[t].lower(), 1e-6), prev_tag)
                for prev_tag in states
            )
            viterbi[t][tag] = max_prob
            backpointer[t][tag] = prev_best

    best_path = []
    best_final_tag = max(states, key=lambda tag: viterbi[-1][tag])
    best_path.append(best_final_tag)

    for t in range(n-1, 0, -1):
        best_path.insert(0, backpointer[t][best_path[0]])

    return best_path
```

```
[12]
#Next we define the viterbi algo for the 2nd Methodology
def viterbi_second_order(sentence, transition_probs, emission_probs, tag_counts, unk_tag=default_tag):
    n = len(sentence)
    states = list(tag_counts.keys())
    viterbi = [{}]
    backpointer = [{}]

    for tag in states:
        viterbi[0][tag] = transition_probs.get(('<s>', '<s>'), {}).get(tag, 1e-6) * emission_probs.get(tag, {}).get(sentence[0].lower(), 1e-6)
        backpointer[0][tag] = None

    for t in range(1, n):
        viterbi.append({})
        backpointer.append({})
        for tag in states:
            max_prob, prev_best = max(
                (viterbi[t-1][prev_tag] * transition_probs.get((prev_prev_tag, prev_tag), {}).get(tag, 1e-6) * emission_probs.get(tag, {}).get(sentence[t].lower(), 1e-6), prev_tag)
                for prev_prev_tag in states for prev_tag in states
            )
            viterbi[t][tag] = max_prob
            backpointer[t][tag] = prev_best

    best_path = []
    best_final_tag = max(states, key=lambda tag: viterbi[-1][tag])
    best_path.append(best_final_tag)

    for t in range(n-1, 0, -1):
        best_path.insert(0, backpointer[t][best_path[0]])

    return best_path
```

```
[13]
#Next we define the viterbi algo for the 3rd Methodology
def viterbi_word_dependent(sentence, transition_probs, emission_probs, tag_counts, unk_tag=default_tag):
    n = len(sentence)
    states = list(tag_counts.keys())
    viterbi = [{}]
    backpointer = [{}]

    for tag in states:
        viterbi[0][tag] = transition_probs.get('<s>', {}).get(tag, 1e-6) * emission_probs.get(tag, {}).get('<s>', {}).get(sentence[0].lower(), 1e-6)
        backpointer[0][tag] = None

    for t in range(1, n):
        viterbi.append({})
        backpointer.append({})
        for tag in states:
            max_prob, prev_best = max(
                (viterbi[t-1][prev_tag] * transition_probs.get(prev_tag, {}).get(tag, 1e-6) * emission_probs.get(tag, {}).get(sentence[t-1].lower(), {}).get(sentence[t].lower(), 1e-6), prev_tag)
                for prev_tag in states
            )
            viterbi[t][tag] = max_prob
            backpointer[t][tag] = prev_best

    best_path = []
    best_final_tag = max(states, key=lambda tag: viterbi[-1][tag])
    best_path.append(best_final_tag)

    for t in range(n-1, 0, -1):
        best_path.insert(0, backpointer[t][best_path[0]])

    return best_path
```

2.4 Defining Evaluation Function for 36 tags and 4 tags

Next we define the evaluation function to test our models performance for the 36 tag model and 4 tag model for each configuration.

```
[14]
#We define an evaluation function to find the accuracy metrics for our methodologies
def evaluate_model_with_tagwise(test_data, viterbi_func, transition_probs, emission_probs, tag_counts, collapsed=False):
    correct = 0
    total = 0
    tag_correct = defaultdict(int)
    tag_total = defaultdict(int)

    for sentence in test_data:
        words = [word for word, _ in sentence]
        true_tags = [tag for _, tag in sentence]
        predicted_tags = viterbi_func(words, transition_probs, emission_probs, tag_counts)

        min_len = min(len(predicted_tags), len(true_tags))

        for i in range(min_len):
            tag_total[true_tags[i]] += 1
            if predicted_tags[i] == true_tags[i]:
                correct += 1
                tag_correct[true_tags[i]] += 1
            total += 1

    overall_accuracy = correct / total if total > 0 else 0
    tagwise_accuracy = {tag: tag_correct[tag] / tag_total[tag] if tag_total[tag] > 0 else 0 for tag in tag_total}


    if collapsed:
        print(f"Overall Accuracy: {overall_accuracy:.4f}")
        print("\Tag-wise Accuracy:")
        for tag in ['N', 'V', 'A', 'O']:
            acc = tagwise_accuracy.get(tag, 0)
            print(f"{tag}: {acc:.4f}")
    else:
        print(f"Overall Accuracy: {overall_accuracy:.4f}")
        print("\nTag-wise Accuracy:")
        for tag, acc in sorted(tagwise_accuracy.items()):
            print(f"{tag}: {acc:.4f}")
    return overall_accuracy, tagwise_accuracy
```

2.5 Evaluating for the 36 tag models

Using the above function,we generate the accuracy metrics for our 3 methodologies for 36 tags.The outputs have been discussed in the Results section.

```
[15]
# Now we evaluate each model
print("Evaluating models...")
first_order_acc, first_order_tagwise = evaluate_model_with_tagwise(test_data, viterbi_first_order, transition_probs_first, emission_probs, tag_counts)
second_order_acc, second_order_tagwise = evaluate_model_with_tagwise(test_data, viterbi_second_order, transition_probs_second, emission_probs, tag_counts)
word_dep_acc, word_dep_tagwise = evaluate_model_with_tagwise(test_data, viterbi_word_dependent, transition_probs_first, emission_probs_word_dependent, tag_counts)

# Printing the results
print(f"First Order HMM Overall Accuracy with 36 tags: {first_order_acc:.4f}")
print("Tag-wise accuracy:")
for tag, acc in first_order_tagwise.items():
    print(f"{tag}: {acc:.4f}")

print(f"\nSecond Order HMM Overall Accuracy with 36 tags: {second_order_acc:.4f}")
print("Tag-wise accuracy:")
for tag, acc in second_order_tagwise.items():
    print(f"{tag}: {acc:.4f}")

print(f"\nWord Dependent HMM Overall Accuracy with 36 tags: {word_dep_acc:.4f}")
print("Tag-wise accuracy:")
for tag, acc in word_dep_tagwise.items():
    print(f"{tag}: {acc:.4f}")
```

2.6 Evaluating for 4 tag models

For the 4 tag models,all the steps are the same as the 36 tag model except we map the 36 tags into 4 tags as shown below.

```
[18]
#Now we do implementation for the collapsed tags part with 4tags instead of 36
collapsed_tags = {
    'N': {'NN', 'NNS', 'NNP', 'NNPS'},
    'V': {'VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ'},
    'A': {'JJ', 'JJR', 'JJS', 'RB', 'RBR', 'RBS'},
    'O': set()  # Will be updated after all_tags is known
}

# We update 'O' to include remaining tags after all_tags is defined
all_original_tags = set(tag for sent in train_data for _, tag in sent)
collapsed_tags['O'] = all_original_tags - (collapsed_tags['N'] | collapsed_tags['V'] | collapsed_tags['A'])
def get_collapsed_tag(tag):
    for key, values in collapsed_tags.items():
        if tag in values:
            return key
    return 'O'  # Fallback
```

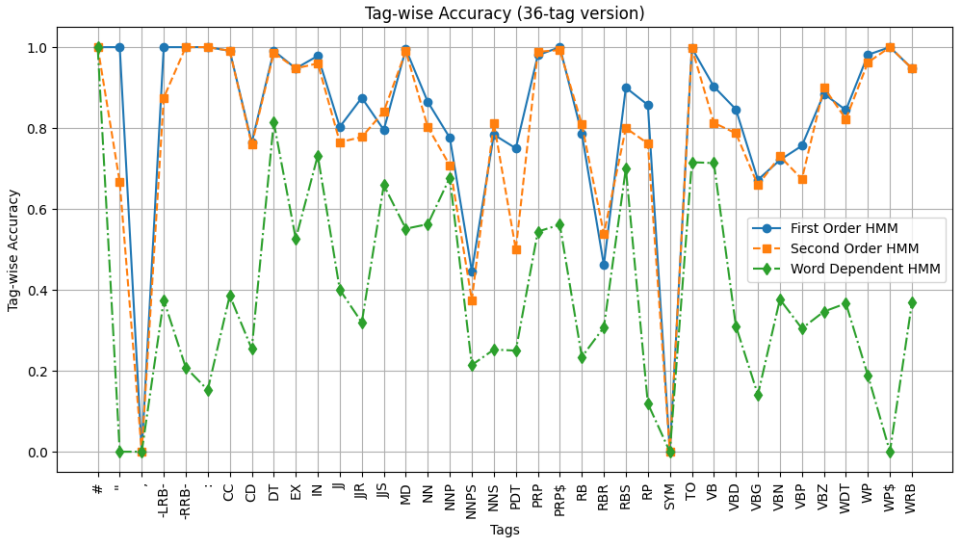
Next,we again use that evaluation function to generate the necessary metrics for the 4 tag models.

```
#Now we find the metrics to judge our model performance and print those
print("First Order HMM:")
first_order_acc, first_order_tagwise = evaluate_model_with_tagwise(
    test_data, viterbi_first_order, transition_probs_first, emission_probs, tag_counts
)

print("\nSecond Order HMM:")
second_order_acc, second_order_tagwise = evaluate_model_with_tagwise(
    test_data, viterbi_second_order, transition_probs_second, emission_probs, tag_counts
)

print("\nWord Dependent HMM:")
word_dep_acc, word_dep_tagwise = evaluate_model_with_tagwise(
    test_data, viterbi_word_dependent, transition_probs_first, emission_probs_word_dependent, tag_counts
)

```

Chapter 3

Results

3.1 36-Tag Models



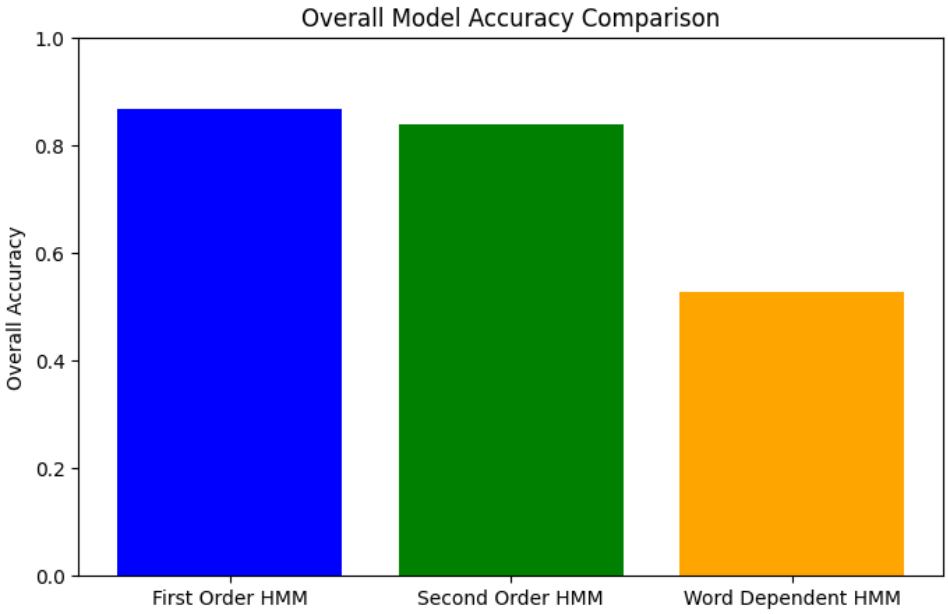
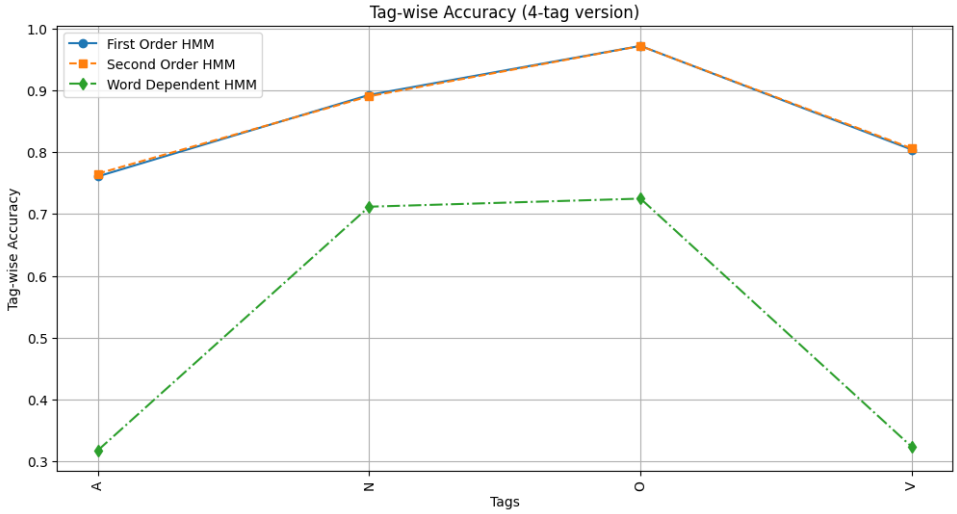


Table 3.1: Comparison of HMM Variations with 36 Tags

Tag	First Order HMM	Second Order HMM	Word Dependent HMM
#:	1.0000	1.0000	1.0000
":	1.0000	0.6667	0.0000
,:	0.0000	0.0000	0.0000
-LRB-	1.0000	0.8750	0.3750
-RRB-	1.0000	1.0000	0.2083
::	1.0000	1.0000	0.1525
CC	0.9911	0.9911	0.3862
CD	0.7643	0.7586	0.2543
DT	0.9909	0.9861	0.8145
EX	0.9474	0.9474	0.5263
IN	0.9785	0.9599	0.7320
JJ	0.8029	0.7652	0.4005
JJR	0.8750	0.7778	0.3194
JJS	0.7955	0.8409	0.6591
MD	0.9951	0.9902	0.5512
NN	0.8646	0.8024	0.5624
NNP	0.7759	0.7077	0.6775
NNPS	0.4464	0.3750	0.2143
NNS	0.7837	0.8125	0.2525
PDT	0.7500	0.5000	0.2500
PRP	0.9801	0.9886	0.5442
PRP$	1.0000	0.9928	0.5612
RB	0.7854	0.8091	0.2343
RBR	0.4615	0.5385	0.3077
RBS	0.9000	0.8000	0.7000
RP	0.8571	0.7619	0.1190
SYM	0.0000	0.0000	0.0000
TO	0.9976	0.9976	0.7156
VB	0.9025	0.8129	0.7135
VBD	0.8463	0.7872	0.3108
VBG	0.6721	0.6590	0.1410
VBN	0.7209	0.7302	0.3767
VBP	0.7563	0.6738	0.3047
VBZ	0.8845	0.9003	0.3465
WDT	0.8444	0.8222	0.3667
WP	0.9811	0.9623	0.1887
WP$	1.0000	1.0000	0.0000
WRB	0.9474	0.9474	0.3684
Overall Accuracy	0.8681	0.8407	0.5272

3.2 4-Tag Models



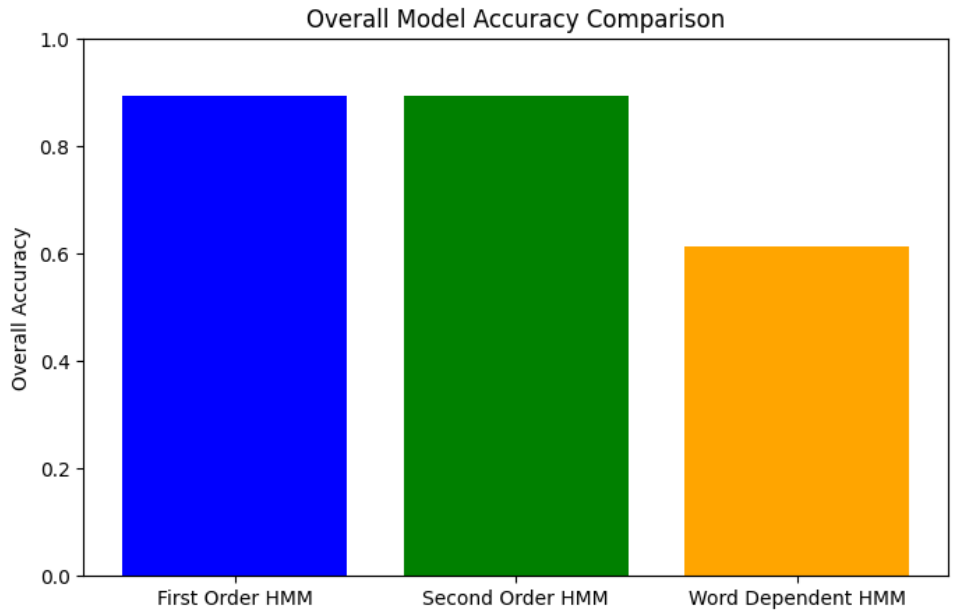


Table 3.2: Comparison of HMM Variations with 4 Tags

Tag	First Order HMM	Second Order HMM	Word Dependent HMM
A	0.7607	0.7652	0.3176
N	0.8926	0.8904	0.7118
O	0.9719	0.9717	0.7249
V	0.8040	0.8060	0.3232
Overall Accuracy	0.8954	0.8954	0.6139

Chapter 4

Discussion and Analysis

4.1 Performance of 36-Tag vs. 4-Tag Model across Three HMM Configurations:

The results of the experiment compare the performance of three different Hidden Markov Model (HMM) configurations—**First-Order HMM, Second-Order HMM, and Word-Dependent HMM**—across two different tag sets: **36-tag** (detailed Part-of-Speech (POS) tagging) and **4-tag** (simplified POS tagging).

Configuration	36-Tag Model Accuracy	4-Tag Model Accuracy
First-Order HMM	0.8681	0.8954
Second-Order HMM	0.8407	0.8954
Word-Dependent HMM	0.5272	0.6139

Table 4.1: Comparison of accuracy across different HMM configurations

From these values, we observe the following trends:

- **First-Order HMM performs the best** among all three configurations in both 36-tag and 4-tag models.
- **Second-Order HMM provides slightly lower accuracy** than First-Order HMM in the 36-tag model but achieves the same performance as First-Order HMM in the 4-tag model.
- **Word-Dependent HMM performs significantly worse** than the other two configurations in both tag sets, with particularly low accuracy in the 36-tag model (52.72%) compared to 4-tag (61.39%).

4.2 Overall Performance Comparison of the 36-Tag vs 4-Tag Model

- The **4-tag model consistently outperforms the 36-tag model** across all three HMM configurations.
- The performance gap is most evident in the **Word-Dependent HMM** configuration, where the **36-tag model achieves only 52.72% accuracy**, while the **4-tag model achieves 61.39% accuracy**.
- The difference is less significant in the First-Order and Second-Order HMMs, but the **4-tag model still achieves a higher accuracy of 89.54% in both cases, compared to 86.81% (First-Order) and 84.07% (Second-Order) in the 36-tag model**.

4.3 Why Does the 4-Tag Model Outperform the 36-Tag Model?

The superior performance of the **4-tag model over the 36-tag model** can be explained by the fundamental assumptions of HMMs regarding **transition and emission probabilities**:

4.3.1 Sparsity and Data Availability

- In the **36-tag model**, there are **more possible tag transitions and word-tag combinations**, leading to **sparse transition and emission probabilities**.
- With **fewer training instances for each tag**, the model struggles to generalize well, especially for rare tags like *NNPS (Proper Plural Nouns)* or *WP$ (Possessive Wh-Pronoun)*.
- In contrast, the **4-tag model groups multiple similar tags together**, resulting in **denser probability distributions and more robust parameter estimation**.

4.3.2 Transition Probability Estimation

- The **First-Order HMM assumes that the probability of a tag depends only on the previous tag**. When we have 36 tags, the number of possible transitions is much larger, making it harder to obtain accurate transition probabilities from the training data.
- The **4-tag model reduces the number of possible transitions**, making the transition probability estimates more reliable.

4.3.3 Emission Probability Estimation

- The **Word-Dependent HMM performs particularly poorly for the 36-tag model (52.72%)**, as it **assumes a dependence on both the tag and the previous word**. This leads to a **large number of unique word-tag transitions, many of which are rarely observed**, making probability estimation unreliable.
- In the **4-tag model, fewer tags mean that each tag has more associated words**, leading to **better emission probability estimates** and **smoother generalization**.

4.3.4 Overfitting in the 36-Tag Model

- With **36 tags, the model has to learn more parameters** compared to the 4-tag model, leading to **greater chances of overfitting** to specific word-tag pairs that appear infrequently in the training data.
- The **4-tag model generalizes better** because it reduces the number of parameters, making it **less sensitive to noise and rare occurrences** in the training data.

Chapter 5

Conclusion

1. The 4-tag model is more suitable for tasks requiring broader POS categories, such as text classification or speech recognition, where fine-grained distinctions between tags are not necessary.

2. The 36-tag model, despite its lower accuracy, provides finer linguistic granularity, making it preferable for syntactic parsing, machine translation, and linguistic research where detailed tag distinctions matter.

3. The Word-Dependent HMM configuration is not ideal for either model due to its reliance on complex dependencies that are difficult to estimate accurately, especially in large-tag scenarios.

4. Thus, while the 4-tag model provides higher accuracy due to denser probability distributions and better generalization, the 36-tag model remains valuable when detailed POS distinctions are required, even if it comes at the cost of lower performance.