Part-of-Speech Tagging using Hidden Markov Models

Comparing decoding techniques and exploring adversarial training strategies

By Kayvan Shah

Project Background

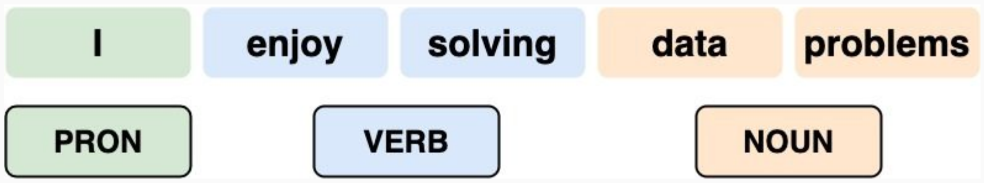
- HMMs enable the modeling of the sequential nature of language by associating hidden states (POS tags) with observed words, capturing the underlying grammar and syntax.
- HMMs can achieve accuracy in the range of *85-90%*, more advanced models often surpass these scores, reaching up to *95%* or higher, especially when trained on large datasets with diverse linguistic patterns.
- In this project, we are utilizing HMMs for POS tagging and comparing decoding techniques and exploring adversarial training strategies for enhanced accuracy and robustness in sequence labeling tasks.

Part-of-Speech Tagging

- Structure prediction task
- ***Problem***: Assign every token a values from a discrete label-space
- ***Challenge***: Ambiguity - different syntactic role of word

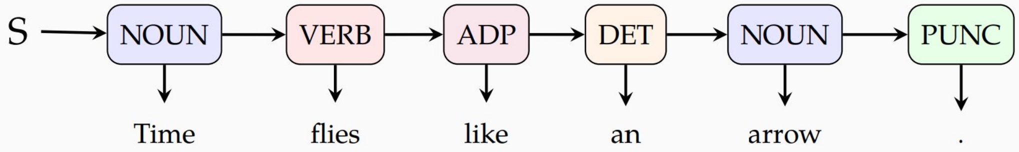
POS tagging is useful for:

1. Text Parsing
2. Text Classification
3. Speech synthesis
4. Machine translation



CC	conjunction, coordinating	PRP$	pronoun, possessive
CD	numeral, cardinal	RB	adverb
DT	determiner	RBR	adverb, comparative
EX	existential there	RBS	adverb, superlative
FW	foreign word	RP	particle
IN	preposition or conjunction, subordinating	RRB	right round bracket
JJ	adjective or numeral, ordinal	SYM	symbol
JJR	adjective, comparative	TO	"to" as preposition or infinitive marker
JJS	adjective, superlative	UH	interjection
LRB	left round bracket	VB	verb, base form
LS	list item marker	VBD	verb, past tense
MD	modal auxiliary	VBG	verb, present participle or gerund
NN	noun, common, singular or mass	VBN	verb, past participle
NNP	noun, proper, singular	VBP	verb, present tense, not 3rd person singular
NNPS	noun, proper, plural	VBZ	verb, present tense, 3rd person singular
NNS	noun, common, plural	WDT	WH-determiner
PDT	pre-determiner	WP	WH-pronoun
POS	genitive marker	WP$	WH-pronoun, possessive
PRP	pronoun, personal	WRB	Wh-adverb

Hidden Markov Models



- Probabilistic models - models sequence of observations based on hidden states and observed emissions.
- Entire system evolves over time.
- Computes the maximum likelihood estimate for occurrence of tags wrt to word.
- Probability of a tag depends on the previous tag.
- Probability of word at given state depends only on current tag.

Space and Time Complexity:

1. Transition params matrix of size N * N | Linear time
2. Emission params matrix of size N * M | Polynomial time
3. Prior probability vector of size N * 1 | Polynomial time

Where,
N = Number of states i.e. number of distinct tags
M = Number of observable symbols i.e. number of distinct words

$$
\begin{aligned}
t(s'|s) &= \frac{\text{count}(s \rightarrow s')}{\text{count}(s)} \\
e(x|s) &= \frac{\text{count}(s \rightarrow x)}{\text{count}(s)} \\
\pi(s) &= \frac{\text{count}(null \rightarrow s)}{\text{count}(num\_sentences)}
\end{aligned}
$$

Where,

- t is the transition parameter
- e is the emission parameter
- π is the initial state or prior probabilities

HMM implementation

```python
class HMM:
    def _compute_prior_params(self, train_data):
        """Compute the prior probabilities

        Formula: π(s) = count(null -> s) / count(num_sentences)
        """
        tag_to_index = {tag: i for i, tag in enumerate(self.labels)}
        num_sentences = len(train_data)

        for sentence in train_data:
            label = sentence[0][1]
            state_idx = tag_to_index[label]
            self.priors[state_idx] += 1

        self.priors = self.priors / num_sentences
        self.priors = self._smoothen_propabilities(self.priors, self.smoothing_constant)

    def _compute_transition_params(self, train_data):
        """Compute transition parameters

        Formula: t(s'|s) = count(s -> s') / count(s)
        """
        tag_to_index = {tag: i for i, tag in enumerate(self.labels)}

        for sentence in train_data:
            label_indices = [tag_to_index.get(label) for _, label in sentence]

            for i in range(1, len(label_indices)):
                prev_state = label_indices[i - 1]
                curr_state = label_indices[i]
                self.transitions[prev_state, curr_state] += 1

        row_agg = self.transitions.sum(axis=1)[:, np.newaxis]
        self.transitions = self.transitions / row_agg
        self.transitions = self._smoothen_propabilities(self.transitions, self.smoothing_constant)
```

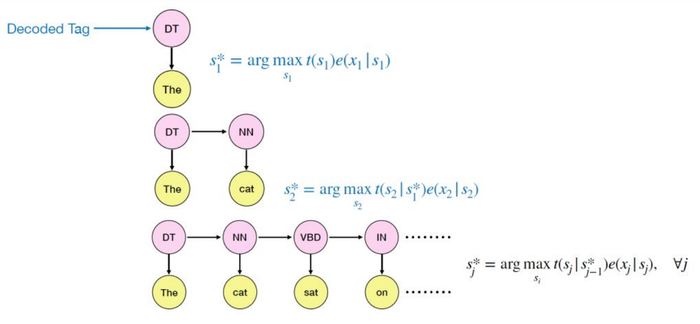
```python
def _compute_emission_params(self, train_data):
    """Compute emission parameters

    Formula: e(x|s) = count(s -> x) / count(s)
    """
    word_to_index = dict(zip(self.vocab["word"], self.vocab["index"]))
    tag_to_index = {tag: i for i, tag in enumerate(self.labels)}

    for sentence in train_data:
        for word, label in sentence:
            state_idx = tag_to_index[label]
            word_idx = word_to_index.get(word, word_to_index[VocabConfig.UNKNOWN_TOKEN])
            self.emissions[state_idx, word_idx] += 1

    row_agg = self.emissions.sum(axis=1)[:, np.newaxis]
    self.emissions = self.emissions / row_agg
    self.emissions = self._smoothen_propabilities(self.emissions, self.smoothing_constant)
```

Greedy Decoding



- **Local Optimization**: Selects the most likely tag for each word, focusing on maximizing likelihood at each step, independently.
- Start from the first word and decode one state at a time
- Does not consider overall sequence context

Input:
HMM with

- Transition params matrix t(s | s') of size N * N
- Emission params matrix e(w | s) of size N * M
- Prior probability vector π(s) of size N * 1

Where,
- N = Number of states i.e. number of distinct tags
- M = Number of observable symbols i.e. number of distinct words

Output:
- List of tags for the given sequence of words

TIme Complexity: O(T * N)
- Given a sequence of length T and N number of states
- For each word in the sequence, it computes the most probable tag.

Space Complexity: O(N^2 + NM + N)

Greedy Decoding

```python
class GreedyDecoding:

    def _decode_single_sentence(self, sentence):
        predicted_tags = []

        prev_tag_idx = None

        for word in sentence:
            word_idx = self.word_to_index.get(word, self.word_to_index[VocabConfig.UNKNOWN_TOKEN])

            if prev_tag_idx is None:
                # scores = self.priors * self.emissions[:, word_idx]
                scores = self.priors_emissions[:, word_idx]
            else:
                scores = self.transitions[prev_tag_idx] * self.emissions[:, word_idx]

            prev_tag_idx = np.argmax(scores)
            predicted_tags.append(self.states[prev_tag_idx])

        return predicted_tags

    def decode(self, sentences):
        predicted_tags_list = []

        for sentence in tqdm(sentences):
            predicted_tags = self._decode_single_sentence([word for word, tag in sentence])
            predicted_tags_list.append(predicted_tags)

        return predicted_tags_list
```

Viterbi Decoding

- **Dynamic Programming Basis -** Utilizes dynamic programming, breaking down the sequence into smaller subproblems. It explores all possible paths and retains the most probable sequence found so far.
- **Optimal Path Identification** - Aims to identify the most probable sequence of hidden states given the observed sequence, utilizing a trellis structure.
- **Backtracking mechanism** - It retraces the path of maximum probabilities to identify the most probable sequence of states

Input:
HMM with

- Transition params matrix t(s | s') of size N * N
- Emission params matrix e(w | s) of size N * M
- Prior probability vector π(s) of size N * 1

Where,

- N = Number of states i.e. number of distinct tags
- M = Number of observable symbols i.e. number of distinct words

Output:

- List of tags for the given sequence of words

TIme Complexity: O(T * N^2)

- Given a sequence of length T and N states:
- It constructs a trellis or matrix of size T * N to compute the most probable path.

Space Complexity: O(N^2 + NM + N)

Viterbi Decoding

```python
class ViterbiDecoding:
    def _decode_single_sentence(self, sentence):
        V, path, word_idx = self._initialize_variables(sentence)

        V[0] = np.log(self.priors_emissions[:, word_idx[0]])

        for t in range(1, len(sentence)):
            # Compute scores
            scores = (
                V[t - 1, :, np.newaxis]
                + np.log(self.transitions)
                + np.log(self.emissions[:, word_idx[t]])
            )
            V[t] = np.max(scores, axis=0)
            path[t] = np.argmax(scores, axis=0)

        # Backtracking
        predicted_tags = [0] * len(sentence)
        predicted_tags[-1] = np.argmax(V[-1])

        for t in range(len(sentence) - 2, -1, -1):
            predicted_tags[t] = path[t + 1, predicted_tags[t + 1]]

        predicted_tags = [self.states[tag_idx] for tag_idx in predicted_tags]
        return predicted_tags
```

Adversarial Strategies

- Laplace Smoothing with Variation:
  - Apply Laplace smoothing with varying smoothing constants to transition and emission probabilities.
- Noise Addition:
  - Introduce controlled noise to the emission probabilities matrix
- Random Ranged Perturbations
  - Introduce variations in probabilities by modifying them slightly without drastically changing the structure by randomly generated noise from a selected range
- Controlled Adjustments based perturbation
  - Modify the actual probabilities by slight percentage
  - Select those which are least probable

Pipeline

Data Preparation

- Case insensitive
- Vocabulary generation
  - Word to index mapping
  - Drop words low frequency
- Sequence processing
  - Replace OOV words
  - Enhance the sequence with special tokens for better context

Seq2Seq labelling

- For training an HMM following are the inputs:
  - The vocabulary
  - Labels sequences
- Outputs 3 matrices:
  - Prior probabilities
  - Transition probabilities
  - Emission probabilities

Sequence Decoding

- Reveal the most probable sequence of hidden states given observations
- Decoding Techniques:
  - Greedy Decoding
  - Viterbi decoding
- Evaluation
  - For every true and predicted sequences doing a tag-by-tag comparison.

For this task, **Wall Street Journal** Dataset is used. The POS tagging dataset (PENN Treebank) has ~40 unique tags.

	index	sentence	labels
0	0	[pierre, vinken, ,, 61, years, old, ,, will, j...	[NNP, NNP, ,, CD, NNS, JJ, ,, MD, VB, DT, NN, ...
1	1	[mr., vinken, is, chairman, of, elsevier, n.v....	[NNP, NNP, VBZ, NN, IN, NNP, NNP, ,, DT, NNP, ...
2	2	[rudolph, agnew, ,, 55, years, old, and, forme...	[NNP, NNP, ,, CD, NNS, JJ, CC, JJ, NN, IN, NNP...

Results

- Approach using default and laplace smoothing strategy have nearly same accuracy
- Viterbi algorithm outperforms greedy decoding algorithm for standard and laplace smoothing strategy
- Greedy algorithm outperform Viterbi decoding algorithm when perturbations are introduced.
- There is significant drop in accuracy with introduction of perturbations.

	Accuracy *(computed by tag-by-tag comparison)*	
Strategies	**Greedy Decoding**	**Viterbi Decoding**
Standard	0.9156	0.9321
Laplace Smoothing	0.9155	0.9323
Noise Addition to Transition Params	0.9124	0.0907
Random Ranged Perturbations	0.7031	0.0945
Controlled Adjustments based Perturbations	TBD	TBD

Analysis

- Greedy algorithm is fast
  - Rapid and independent decision-making per token.
  - Can produce results quickly but may yield suboptimal outcomes due to its localized decisions
- Viterbi Algorithm finds optimal results
  - Focuses on finding the optimal sequence, considering the entire observation sequence and their associated probabilities.
  - Yields the best possible tags for a given sequence of words but involves higher computational complexity.
- Random perturbation are disruptive in nature
  - Random perturbations in HMMs are inherently disruptive and can distort learned patterns within the model.
  - Particularly, they heavily impact emission probabilities, leading to significant disturbances in the decoding process.
- Drastic drop in Viterbi decoding accuracy
  - Exhibits high sensitivity to changes in emission probabilities.
  - The disruption caused by perturbations often leads to a drastic drop in Viterbi decoding accuracy.
  - This sensitivity affects the model's ability to maintain the sequential integrity of the output.

Future Work

To improve model accuracy and robustness

1. Adversarial Training Data:
    a. Incorporate adversarial training techniques, where the model is trained on both clean and perturbed data
2. Limited Perturbation:
    a. Apply perturbations to a subset of the emission matrix or restrict perturbation to less critical probabilities. This way, the model's core learned patterns may remain intact.
3. Focused Perturbation Strategy:
    a. Focused perturbations on specific subsets of emission probabilities.
    b. Introduce perturbations only to infrequent or ambiguous observations.

THANK YOU