Assignment 2: Word Sequencing

NEURAL NEXUS

Group Members:

1. Lekansh Bhatnagar 220586
2. Kollamoram Karthik 220538
3. Jiyanshu Dhaka 220481
4. Harshit Agarwal 220438
5. Rohit Karwa 220911
6. Ayush 220259

Date of Submission: July 17th, 2024

University: Indian Institute of Technology Kanpur

Course: CS771 Introduction to Machine Learning

Instructor: Purushottam Kar **Total Marks:** 40

Contents

Abstract

This report presents our approach and implementation for the Word Sequencing problem as part of the CS771: Introduction to Machine Learning course. We describe the design decisions, algorithms used, and our final implementation.

1 Introduction

The Word Sequencing problem involves guessing a word given a set of bigrams. This task is inspired by genome sequencing in genetics. We developed a machine learning algorithm to solve this problem and report our findings and methodologies in this document.

2 Design Decisions

This section provides an in-depth look at the various design decisions and considerations that were made during the development of our machine learning algorithm. The objective was to create a robust, efficient, and interpretable model capable of predicting words based on the presence of bigrams. Each decision was carefully crafted to balance model complexity, performance, and generalization capability.

2.1 Choice of Algorithm

We chose a decision tree for its interpretability and effectiveness in handling text data features like bigrams. Decision trees excel in capturing complex relationships in data through recursive splitting based on feature presence, making them ideal for tasks where specific word combinations (bigrams) are influential. This transparency helps in identifying key bigrams for classification, while also offering flexibility in balancing model complexity and interpretability through parameter tuning.

2.2 Splitting Criteria

The decision tree algorithm selects the splitting criteria based on the most frequent bigram present in the dataset. This approach is optimized for text classification tasks involving bigrams due to the following reasons:

2.2.1 Bigram Frequency Selection

The algorithm identifies the most frequently occurring bigram that has not been used in previous splits (tracked in the `history` set). This bigram is chosen as the splitting criterion for each node in the decision tree.

2.2.2 Entropy Reduction

Splitting based on the most frequent bigram maximizes information gain, which measures the reduction in uncertainty (entropy) in the dataset. This approach effectively separates the dataset into more homogeneous subsets, enhancing classification accuracy.

2.3 Optimality

- **Effective Feature Selection**: By leveraging frequently occurring bigrams, the algorithm capitalizes on informative features that are indicative of specific classes or categories in the text data.
- **Computational Efficiency**: Frequency-based selection is computationally efficient as it focuses on a subset of bigrams likely to provide significant discriminatory power, optimizing both time and space complexity.

2.4 Space Optimization

- **Memory Usage**: The algorithm minimizes memory usage by storing only necessary data structures (e.g., `history` set) and clearing unnecessary data after training each node.
- **Data Structures**: Efficient data structures like sets (`history`) ensure quick lookup and insertion operations, optimizing both memory and processing time during training.

In conclusion, the decision tree algorithm's strategy of selecting the most frequent bigram for splitting is optimal and space-efficient. It maximizes information gain while minimizing computational overhead, making it well-suited for text classification tasks based on bigram analysis.

2.5 Stopping Criteria

The decision tree algorithm employs specific stopping criteria to determine when to stop splitting nodes. This approach is crucial for controlling model complexity and ensuring generalizability in text classification tasks, particularly with bigram analysis. The chosen stopping criteria are based on the following considerations:

2.5.1 Minimum Leaf Size

The algorithm stops splitting a node further if the number of instances (words) in the node falls below a specified minimum leaf size. This prevents the model from overfitting by ensuring that each leaf node contains enough instances to make reliable predictions.

2.5.2 Reasons for Choosing Stopping Criteria

- **Computational Efficiency**: Setting clear stopping criteria reduces unnecessary computations, optimizing the training process and improving runtime performance.
- **Interpretability**: A shallower tree (controlled by maximum depth) and well-populated leaf nodes (controlled by minimum leaf size) enhance the interpretability of the model, making it easier to understand and analyze.

2.6 Implementation in the Decision Tree Algorithm

In practice, these stopping criteria are implemented within the decision tree algorithm for text classification. They are integrated to balance between capturing complex relationships in the data and maintaining model simplicity and interpretability.

2.7 Hyperparameters

The decision tree algorithm for text classification utilizes several hyperparameters to optimize model performance and control its behavior. These hyperparameters include:

2.7.1 Minimum Leaf Size

The `min_leaf_size` hyperparameter determines the minimum number of instances (words) required to form a leaf node in the decision tree. Nodes with fewer instances than this threshold will not be split further, helping to prevent overfitting. It is kept as 1 to avoid splitting of nodes that already have only one word.

2.7.2 Maximum Depth

The `max_depth` hyperparameter limits the depth of the decision tree. It specifies the maximum number of levels that the tree can grow. We have kept it as **None** so that we can promote over-fitting in our model.

2.7.3 Verbose Mode

The `verbose` hyperparameter controls whether detailed information about the training process is printed during execution. When set to `True`, it enables printing of information such as tree structure and splitting criteria, aiding in debugging and understanding the model's behavior.

3 Implementation

This section describes the implementation details of our algorithm, including the key functions and their roles.

We first preprocess the given words to extract and sort the bigrams, removing duplicates and retaining only the first five bigrams.

3.1 Model Training

The `my_fit` function takes the dictionary as input and trains the Decision Tree model. The steps involved are:

1. **Splitting the data based on bigrams:**
   The `Node` class is used to represent each node in the tree. The `fit` method splits the data recursively to create decision nodes and leaves.

```
def fit(self, words, verbose=False):
    self.words = words
    self.root = Node(depth=0, parent=None)
    if verbose:
        print("root")
        print("", end='')
    self.root.fit(all_words=self.words,
    my_words_idx=np.arange(len(self.words)),
    min_leaf_size=self.min_leaf_size,
       max_depth=self.max_depth, verbose=verbose)
    # Clear unnecessary data after training
    self.root.clear_history()
```

2. **Recursively creating decision nodes and leaves:**
   The `Node` class handles the creation of decision nodes and leaves, deciding whether to split further based on the stopping criteria

```
def fit(self, all_words, my_words_idx, min_leaf_size, max_depth,
 fmt_str="    ", verbose=False):
   self.my_words_idx = my_words_idx
   if len(my_words_idx) <= min_leaf_size or self.depth >=
   max_depth:
    self.is_leaf = True
    self.process_leaf(self.my_words_idx)
    if verbose:
    print('')
   else:
     self.is_leaf = False
     self.query, split_dict = self.process_node(all_words,
     my_words_idx, self.history)
     if self.query is None:
        self.is_leaf = True
         self.process_leaf(self.my_words_idx)
           if verbose:
                print('')
           else:
               if verbose:
                    print(self.query)
                for i, (response, split) in enumerate(split_dict.items()):
                   if verbose:
                     if i == len(split_dict) - 1:
                        print(fmt_str + "", end='')
                        fmt_str += "    "
                     else:
                        print(fmt_str + "", end='')
                        fmt_str += "   "
       self.children[response] = Node(depth=self.depth + 1,
       parent=self)
       history = self.history.copy()
       history.add(self.query)
```

```
            self.children[response].history = history
            self.children[response].fit(all_words, split, min_leaf_size,
            max_depth, fmt_str, verbose)
        # Clear history to save space after training
        self.clear_history()
```

3.2 Prediction

The `my_predict` function takes the trained model and a tuple of bigrams to predict the possible words. The function returns a list of guesses.

```
def predict(self, bigrams):
    node = self.root
    while not node.is_leaf:
        node = node.get_child(node.get_query() in bigrams)
    return [self.words[i] for i in node.my_words_idx][:5]
```

4 Results

We assessed our model's performance using the provided dictionary, and here are the summarized findings:

4.1 Training Time

- The training time for our model was **0.905 seconds** on average.

4.2 Model Size

- The on-disk size of our trained model (after pickle dump) was **1.26 MB** (1330235.0 bytes) on average.

4.3 Testing Time

- The total testing time for our model was **0.089 seconds** on average.

4.4 Precision

- The precision of our model, calculated as the average precision across all test points, was **0.965** on average.

5 Conclusion

- We successfully developed a machine learning algorithm to guess words from bigrams. Our approach was validated using a public dictionary, showing promising results.