CS771 Introduction to Machine Learning

Assignment 2

Naveen Kumar	Tejaswa	Gargi Jain
210654	211110	210379
knaveen21@iitk.ac.in	tejaswa21@iitk.ac.in	gargijain21@iitk.ac.in

Priyanshu Biswas
210779
priyanshu21@iitk.ac.in

Question 1

To design the ML algorithm for the task, we undertook the following design decisions and their rationale:

1. Feature Extraction with BiGram:

Decision:
We used character bigrams as features.
Rationale:
Character bigrams effectively capture local dependencies in words, making them suitable for distinguishing between words with similar structures. Bigrams (which are sequences of two characters) offer a balance between capturing sufficient context and maintaining a manageable feature space compared to higher-order n-grams.

We utilized CountVectorizer with analyzer='char' and `ngram_range=(2, 2)` to extract character bigrams from the input words. The CountVectorizer is a feature extraction method provided by the sklearn library. It converts a collection of text documents to a matrix of token counts. For our purpose, we configured it to work with character bigrams as follows:

- **Analyzer**:
  We set the analyzer parameter to 'char', which tells the CountVectorizer to treat each character in the input text as a token. This allows us to create n-grams of characters rather than words.

- **N-gram Range**:
  We set the `ngram_range` parameter to (2, 2), specifying that we want to extract 2-character sequences (bigrams) from the text. This configuration ensures that only bigrams are considered as features.

For Example, for input '['hello' , 'world']'.
After applying the CountVectorizer, the bigrams extracted might be:

- For 'hello': 'he', 'el', 'll', 'lo'
- For 'world': 'wo', 'or', 'rl', 'ld'

2. Decision Tree Classifier

We employed a Decision Tree Classifier for word classification.
Raionale:
Decision Trees are intuitive and can handle both categorical and numerical data. They are well-suited for text data represented as bag-of-bigrams, and their tree structure aids in understanding which bigrams are most important for classification. In our machine learning model, we employed the DecisionTreeClassifier from the sklearn library for classifying words based on their character bigrams

1. Splitting Criterion

We used the default splitting criterion, which is **'gini'** as follows:

- **Gini Impurity:**
  $\text{Gini}(p) = 1 - \sum_{i=1}^{C} p_i^2$.
  where p_i is the proportion of samples belonging to class i, and C is the number of classes.

The Gini impurity is minimized when a node is split such that the classes are well separated. It is computationally efficient and often works well for classification tasks.

2. Stopping Rules:

We set the `max_depth` and `min_samples_leaf` parameters to control the growth of the decision tree.

- `max_depth` : his parameter limits the maximum depth of the tree. By controlling the depth, we prevent the tree from becoming too complex. It defines the maximum number of levels in the tree.
- `min_samples_leaf` : This parameter sets the minimum number of samples required to be at a leaf node. By setting this parameter, we avoid creating nodes that are too specific to a subset of the data.It ensures that a leaf node has atleast a certain number of samples.

3. Hyperparameter Tuning with Grid Search:

We opted for hyperparameter tuning using Grid Search.
Rationale:
Grid Search systematically explores multiple combinations of parameter values and cross-validates them to determine the best parameters. This approach ensures that the chosen model performs optimally for the given hyperparameters.

Pruning Strategies:

We used the `max_depth` parameter for implicit pruning rather than explicit pruning techniques. Pruning helps to reduce the complexity of the tree and prevent overfitting. In our algo we used pre-pruning through `max_depth` and `min_samples_leaf`.

4. Handling Small Class Sizes:

In our model, we used `GridSearchCV` to find the optimal hyperparameters for the `DecisionTreeClassif`
One important aspect of this process was determining the appropriate number of folds for cross-validation, which was influenced by the smallest class size in our dataset.
Rationale:
This adjustment ensures that the cross-validation process does not fail due to insufficient data in any class.

Adjusting Cross-Validation Folds Based on Minimum Class Size

Issue: In datasets with very small class sizes, using too many folds in cross-validation can cause problems such as empty folds and unrepresentative training/validation sets.

Solution: We adjusted the number of cross-validation folds to be the smaller of 5 or the minimum class size. This ensures that every fold contains samples from all classes, preventing issues such as empty folds.

Implementation Code:

```
if min_class_size >= 2:
    grid_search = GridSearchCV(DecisionTreeClassifier(), param_grid,
    cv=min(5, min_class_size))
```

Detailed Steps and Justifications:

- **Determine Minimum Class Size:**

  ```
  unique, counts = np.unique(words, return_counts=True)
  min_class_size = min(counts)
  ```

  This step identifies the smallest class in the dataset to ensure that every fold can contain samples from all classes.

- **Set Cross-Validation Folds:**

  ```
  grid_search = GridSearchCV(DecisionTreeClassifier(), param_grid,
  cv=min(5, min_class_size))
  ```

This code sets the number of folds for cross-validation to the smaller of 5 or the minimum class size, ensuring valid cross-validation while balancing practical constraints.

Example Scenario: If the smallest class has only 2 samples, setting 5 folds would not be practical. Instead, we use 2 folds to ensure each fold contains samples from every class.

Impact of Proper Cross-Validation: Properly chosen cross-validation folds ensure that the model evaluation process is both effective and feasible, providing reliable results without introducing data constraints.

5. Custom Prediction Logic:

We developed a custom scoring function to rank potential matches for a given bigram list.

Rationale:

While Decision Trees classify words, predicting based on bigrams requires a flexible scoring mechanism that can match words based on bigram presence and sequence.

The `calculate_match_score` function is designed to evaluate how well a given word matches a list of bigrams. This function is crucial for ranking words based on their similarity to a target bigram sequence.

```
def calculate_match_score(self, bigram_list, word):
    score = 0
    word_bigrams = [word[i:i+2] for i in range(len(word)-1)]
    bigram_sequence_score = sum(1 for i, bigram in enumerate(bigram_list)
    if bigram in word_bigrams[i:i+1])
    if bigram_sequence_score == len(bigram_list):
        score += bigram_sequence_score * 2
    else:
        score += sum(word.count(bigram) for bigram in bigram_list)
    return score
```

Detailed Breakdown

Generating Bigrams from the Word

`word_bigrams = [word[i:i+2] for i in range(len(word)-1)]`

What It Does: This line generates a list of all bigrams (two-character sequences) present in the given `word`.

Why It's Important: Bigrams capture patterns in the text, essential for tasks like text analysis or word games. Extracting bigrams allows for comparison with the target bigram list.

Example: For the word `"hello"`, the bigrams are `["he", "el", "ll", "lo"]`.

Code Explanation: A list comprehension creates bigrams by sliding a two-character window over the length of the word.

Scoring Exact Bigram Sequences

`bigram_sequence_score = sum(1 for i, bigram in enumerate(bigram_list) if bigram in word_bigrams[i:i+1])`

What It Does: Scores the word based on the presence of bigrams from the `bigram_list` in the exact sequence.

Why It's Important: This step evaluates if the bigrams appear in the correct order, which is crucial for tasks where sequence matters.

Example: If `bigram_list` is `["he", "el"]` and `word_bigrams` is `["he", "el", "ll", "lo"]`, both "he" and "el" are present in the correct sequence, so the score is 2.

Code Explanation: The generator expression counts how many bigrams appear in the correct sequence.

Scoring Based on Exact Matches

```
if bigram_sequence_score == len(bigram_list):  score += bigram_sequence_score * 2
```

What It Does: Adds a score that is twice the number of bigrams if the exact sequence is matched.

Why It's Important: Rewards exact matches by giving a higher score for perfectly matched sequences.

Example: For a `bigram_list` with 2 bigrams, if both bigrams appear in the correct sequence, the function adds 4 points to the score (2 bigrams * 2).

Code Explanation: If all bigrams in the `bigram_list` appear in the correct sequence in the `word`, the score is doubled

Scoring Based on Bigram Counts

```
else:  score += sum(word.count(bigram) for bigram in bigram_list)
```

What It Does

If the exact sequence of bigrams from the `bigram_list` is not found in the `word`, this part of the function scores the word based on the total number of occurrences of each bigram in the `bigram_list`.

> *The function sums up the counts of each bigram from the* ***`bigram_list`*** *in the* ***`word`*** *to form the final score.*

Why It's Important

This fallback scoring method ensures that even if the word does not contain the bigram sequence exactly as specified, it is still evaluated for how many of the target bigrams are present. This approach is particularly useful for handling cases where exact matches are rare, but partial matches are still valuable.

> *By evaluating the presence of individual bigrams, this method allows the system to give a score to words that are partially relevant, thus increasing the robustness of the matching process.*

Returning the Score

```
return score
```

What It Does

Finally, this line of code returns the computed score for the `word`.

> *The function outputs the score that reflects how well the word matches the bigram list, which can be used for further ranking or selection tasks.*

Why It's Important

Returning the score is crucial as it enables the comparison of different words based on their relevance to the target bigram sequence. This score can be used in subsequent steps of the algorithm to rank words and select the most appropriate matches.

By returning the score, the function provides a quantitative measure of a word's relevance, which is essential for ranking words and making decisions in applications such as text analysis or word games.

6. Thresholding and Top Matches:

We determined a threshold score to filter top matches.
Rationale:
The threshold helps in retaining high-quality predictions while filtering out low-confidence matches, ensuring the most likely candidates are selected.
We used a threshold of 70% of the highest score to filter matches.
In our implementation, we used a threshold of 70% of the highest score to filter matches. This section explains the concept of the threshold, why 70% was chosen, and how it affects the results.

What is a Threshold?

A threshold is a cutoff value used to make decisions or classifications. In the context of our model, a threshold helps determine which matches are considered acceptable or noteworthy based on their scores.

*Why Use a 70% Threshold?

Purpose: The 70% threshold ensures that only the most relevant matches are selected while excluding less relevant ones.

Reason for 70%:

- **High Threshold Benefits:** A higher percentage of the best score ensures that we focus on the most relevant matches, filtering out weaker results.
- **Balancing Inclusivity and Exclusivity:** A 70% threshold is high enough to be selective but not so high that it excludes too many potential matches.
- **Avoiding Overwhelming Results:** A lower threshold might include too many matches, many of which might be of little use. A 70% threshold strikes a balance between filtering and inclusivity.

*How It Works

Step-by-Step Process:

1. **Calculate Scores:** Compute scores for each potential match based on their criteria.
2. **Determine the Maximum Score:** Find the highest score among all matches.
3. **Compute the Threshold Score:** Calculate the threshold as 70% of the maximum score.

```
max_score = max(scores)  % scores is a list of all calculated scores
threshold_score = 0.7 * max_score
```

4. **Filter Matches:** Select matches with scores above or equal to the threshold score.

```
filtered_matches = [match for match in matches if match.score >=
threshold_score]
```

Example: If the highest score is 85, the threshold score is $0.7 \times 85 = 59.5$. Matches with scores of 85 and 72 are kept, while those below 59.5 are filtered out.

Advantages of Using a 70% Threshold

- **Focus on Best Matches:** Ensures that only the most relevant results are considered.
- **Balanced Approach:** A 70% threshold provides a good balance between inclusivity and exclusivity.
- **Adjustable Flexibility:** The percentage can be adjusted based on specific needs; a lower threshold for more inclusive filtering and a higher threshold for stricter requirements.

Precision Score:

We calculated the precision score for the prediction.
Rationale:
This calculation helps in evaluating the model's performance, particularly when the correct word is among the top predictions. In the context of our matching system, we use the precision score to evaluate the effectiveness of the top-ranked results. The precision score measures the proportion of relevant results among the top results retrieved by the system.

Definition of Precision

Precision is calculated as:

$$\text{Precision} = \frac{\text{Number of Relevant Results}}{\text{Number of Retrieved Results}}$$

Implementation Steps

To calculate precision in our system, follow these steps:

1. **Retrieve Top Matches:** After ranking, select the top N words based on their scores.

```
top_words = [word for word, score in sorted_words[:N]]
```

2. **Identify Relevant Matches:** Check which of these top N words are relevant.

```
relevant_words = set(top_words).intersection(set(test_relevant_words))
```

3. **Calculate Precision:** Compute the precision score as the ratio of relevant words to the total number of top results.

```
precision = len(relevant_words) / N
```

4. **Example Calculation:** If the top 5 words are ["example", "sample", "test", "trial", "demo"] and the relevant words are ["example", "test", "demo"], the precision score is:

$$\text{Precision} = \frac{3}{5} = 0.6$$

Importance of Precision

- **Measures Effectiveness:** Precision shows how well the top-ranked results meet the target criteria.
- **Focus on Quality:** It evaluates the quality of the top results, unlike recall, which measures overall completeness.
- **Model Fine-Tuning:** High precision indicates that the model's top predictions are often correct, which is important for improving model performance.

In summary, the precision score is a key metric for evaluating the performance of a matching or ranking system. It measures the proportion of relevant results among the top-ranked items, offering insights into the effectiveness of the model's predictions and helping to fine-tune the model for better performance.
Precision score is calculated but not used in the main function.

These design decisions ensure that the model is both effective and efficient in predicting words based on bigram sequences.