Permutation Decision Tree Project Report

For Partial Fulfillment of CS F266

Pranav Srinivas
2020A7PS1694G
f20201694@goa.bits-pilani.ac.in

Ravi Sanker S
2020A7PS0142G
f20200142@goa.bits-pilani.ac.in

I. Introduction

Traditionally, decision tree nodes have been assessed for their impurity using well-established metrics like Shannon Entropy or Gini Impurity. However, these traditional measures come with a significant limitation: they assume that data follows the Independent and Identically Distributed (i.i.d.) assumption, a condition seldom met in real-world data-sets.

In the real world, data is rarely i.i.d., and it frequently exhibits complex order dependencies. Our decision-making processes often hinge on the specific sequence of events, not just the events themselves. To address these inherent limitations in the classical approach, Dr. Harikrishnan N.B. introduced a paper that proposes using Effort To Compress (ETC) as a metric to make decision trees, and thus introduced the terms Permutation Decision Tree and Permutation Decision Forests [1].

In [1], the specific algorithm employed to compute ETC is Non-sequential Recursive Pair Substitution (NSRPS). The paper only considered consecutive pairs of symbols. Thus, the number of consecutive symbols considered for employing the NSRPS algorithm could be considered a hyperparameter. There is a potential scope in fine-tuning the depth of the tree as well.

This project report aims to implement and explore the implications of the paper authored by Dr. Harikrishnan N.B. By conducting experiments and potentially tuning hyper-parameters, we seek to delve deeper into the utility of structural impurity, ETC Gain, and Permutation Decision Trees in enhancing decision tree-based machine learning models.

II. Methodology

A. Implementation of NSRPS Algorithm

We wrote our own implementation of the NSRPS algorithm which allows for the adjustment of the number of consecutive symbols considered during the transformation process. This value (the length of the sliding window) is taken as the function parameter. This allows us to use this values as a hyperparameter, as discussed in the Introduction. The results of our algorithm were then compared with the examples given in [1].

B. Implementation of ETC Gain

Instead of metrics such as Information Gain or Gini Impurity Index, this paper dealt with the implementation of ETC Gain. ETC is equal to the count of the number of iterations needed for the NSRPS algorithm to attain a homogeneous sequence when applied iteratively on the input sequence. The following formula was used to calculate as outlined in [1].

$$ETCGain(S,P) = ETC(S) - \sum_{V \in Values(P)} \left|\frac{S_V}{S}\right| .ETC(S_V)$$

P is the chosen parent attribute, S is the class label, V is the possible values of the chosen parent attribute P, $S_V \subset S$, and S_V consists of labels corresponding to the chosen parent attribute P taking the value V , $|S|$ represents the cardinality of the set S.

C. Implementationn of the Permutation Decision Tree

With the above two implementations, Permutation Decision Tree is a straight-forward implementation. With numerous implementation of Decision Trees available, and we ourselves having implemented it from scratch in our Machine Learning course (BITS F464), all that had to change was to replace Information Gain with ETC Gain. Our implementation was then thoroughly tested with the results given in [1].

D. Hyper-parameter Tuning of the Sliding Window Size

The number of consecutive symbols considered in NSRPS has the potential to be a hyper-parameter. We have used the variable name k to denote this in our implementation have used the phrase 'Sliding Window Size' to refer to this in the paper.

III. Experiments

A. Testing of NSRPS

Before testing the results of the Permutation Decision Tree, it was important to test if the results of our implementation of the NSRPS algorithm matched that of [**?**].

Sequence ID	Sequence	ETC (expected)	ETC (obtained)
A	111111	0	0
B	121212	1	1
C	222111	5	5
D	122112	4	4
E	211122	5	5

B. Testing of Decision Trees

Decision Trees were constructed for five different permutations given in [1]. We ran our implementation against the same five permutations and obtained the following trees.

1) *Permutation A: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14*

```
X_0 <= 2 ? 4.142857142857143
 left:1
 right:X_0 <= 4 ? 3.5
  left:X_1 <= 2 ? 3.0
    left:1
    right:0
  right:0
```

2) *Permutation B: 14, 3, 10, 12, 2, 4, 5, 11, 9, 8, 7, 1, 6, 13*

```
X_1 <= 2 ? 5.428571428571429
 left:X_0 <= 4 ? 3.0
  left:1
  right:0
 right:X_0 <= 2 ? 2.0
  left:1
  right:0
```

3) *Permutation C: 13, 11, 8, 12, 7, 6, 4, 14, 10, 5, 2, 3, 1, 9*

```
X_0 <= 4 ? 7.857142857142858
 left:X_1 <= 2 ? 2.6
  left:1
  right:X_0 <= 2 ? 1.0
    left:1
    right:0
 right:0
```

4) *Permutation D: 3, 2, 13, 10, 11, 1, 4, 7, 6, 9, 8, 14, 5, 12*

```
X_0 <= 4 ? 7.857142857142858
 left:X_0 <= 2 ? 2.6
  left:1
  right:X_1 <= 2 ? 1.0
    left:1
    right:0
 right:0
```

5) *Permutation E: 10, 12, 1, 2, 13, 14, 8, 11, 4, 7, 9, 6, 5, 3*

```
X_0 <= 2 ? 5.142857142857143
 left:1
 right:X_1 <= 2 ? 4.5
  left:X_0 <= 4 ? 1.0
    left:1
    right:0
  right:0
```

IV. Observations

Our results of the NSRPS algorithm match that of the examples given in the paper. The decision trees that are generated through our implementation also match that generated in the paper.

V. Conclusion

We have been successful in implementing the paper. There is still a lot of scope in hyper-parameter tuning and interpreting the subsequently generated decision trees. Specifically, we would like to tune the Sliding Window Size employed in the NSPRS algorithm. This will be the next course of action for us.

References

[1] Harikrishnan, N.B., Nithin Nagaraj (2023). Permutation Decision Tree. arXiv:2306.02617v2.