Report of Implementation of FUSINTER Algorithm for Practical Work in AI

Florian Jaksik

Winter Term 2023/24

Abstract

The goal of this project is the implementation of the FUSINTER discretization algorithm as it is described in [ZRR98]. It will consist out of three phases: The first one should be a naive implementation of the steps given in the paper in the Python Programming Language. The second phase will also be implemented in Python, but with optimizations to decrease the run time of the algorithm. The third and last phase is an implementation in C++ to further improve the run time. This last part is expected to be executable within Python.

1 Introduction

Discretization methods take continuous features and split them up into regions. As a basic example we can take the ages of a group of people (22, 85, 2, 30, 14, 5, 54). Now we can set a split point to separate minors from adults. In many countries that would be the age of 18. The discretized features are (adult, adult, minor, adult, minor, minor, adult).

Splitting by predefined values is probably the most basic method for discretization followed by those seeking to give either equal width between splitting points or equal frequency within each interval. In contrast to FUSINTER these algorithms don't use information provided by the labeling of data points. FUSINTER can be described as a supervised method.

Further we can distinguish between discretization algorithm that start with a set of splitting points and then iteratively remove them and those starting without any such points and iteratively add them. The first one are called bottom-up the later one top-down algorithms. FUSINTER falls into the first category.

In the following I will fist give a description of the steps of FUSINTER. Then I will go into the details of my implementation of the first phase of the project by describing the important parts of the code. Then I will show some applications of the algorithm on different datasets. ...

what to do in part 2/3

The code of this project can be found at https://github.com/floxo115/FUSINTER.

2 The FUSINTER Algorithm

The following steps of the FUSINTER algorithm can be found in [ZRR98, p. 315f]. We expect $\bar{x}_0$ to be a vector of real values and $\bar{y}_0$ to be a vector of corresponding labels with unique values $1, \ldots, m$

1. We sort the components of $\bar{x}_0$ in ascending order such that we get a new data vector $\bar{x}$ and corresponding label vector $\bar{y}$.

2. From left to right we sweep over the sorted data and form intervals for runs of examples with the same labels.

3. If there are multiple examples for the same value having different labels, we close the current interval and create a new interval from the value with the mixed labels up to but not including the next example that has a bigger value.

4. The boundary points of the intervals are the splitting points of the FUSINTER algorithm.

5. From the intervals $1, \ldots, k$ we calculate the table T with columns $T_1, \ldots, T_k$. The value of the i-th rows indicate the number of occurrences of examples labeled with i. If we would have two intervals with two classes we would write that as:

$$T = \begin{bmatrix} 3 & 0 \\ 2 & 2 \end{bmatrix}$$

   This would mean that in the first interval we have 5 examples with 3 of label 1 and 2 of label 2 as well as 2 examples in interval 2 wherer all have label 2.

6. If we merge the i-th and the $(i+1)$-th column of T we denote the new table with T_i. With the help of the real valued function ϕ we can calculate the index t such that

$$t = \arg\max_i \phi(T) - \phi(T_i)$$

7. if $\phi(T) - \phi(T_k) > 0$ we remove the k-th split and set $T := T_k$

8. We do this again from step 2 until there are no more splits or the criterion in 7) is not met.

2 is from the paper, but shouldn't it be step 5?

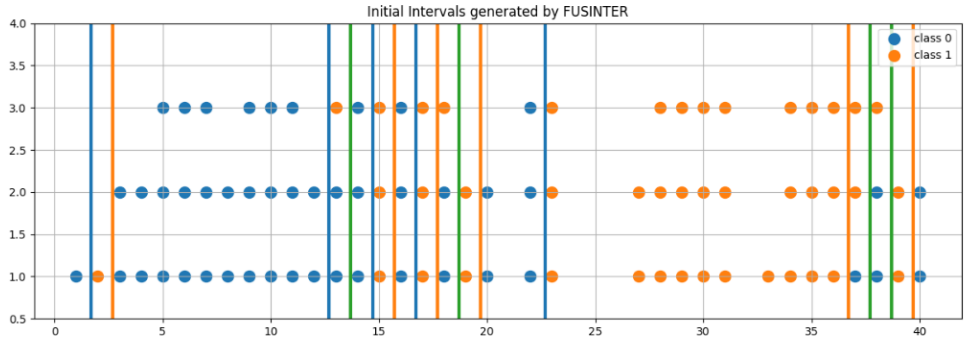

Figure 1: Initial splits created by steps 1-3 of the FUSINTER algorithm

all splits in same color

The ϕ function estimates the quality of the discretization. It is a function from a $m \times k$ matrix to the positive real numbers. In the original paper Shannon's Entropy and Quadratic Entropy is used [ZRR98, p. 318].

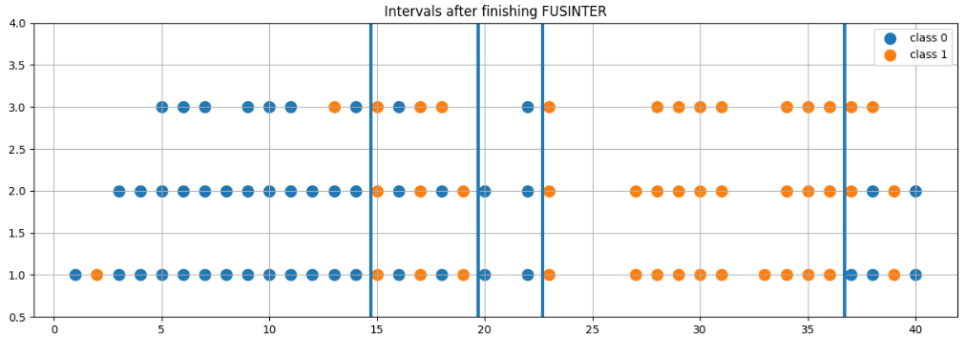

Figure 2: Final splits after completing FUSINTER algorithm

$$\text{Shannon's Entropy } = \phi_1(T) := \sum_{j=1}^{k} \Big\{ \alpha \frac{n_j}{n} \Big(- \sum_{i=1}^{m} \frac{n_{ij} + \lambda}{n_j + m\lambda} \log_2 \frac{n_{ij} + \lambda}{n_j + m\lambda} \Big) + (1 - \alpha) \frac{m\lambda}{n_j} \Big\}$$

$$\text{Quadratic Entropy } = \phi_2(T) := \sum_{j=1}^{k} \Big\{ \alpha \frac{n_j}{n} \Big(\sum_{i=1}^{m} \Big(\frac{n_{ij} + \lambda}{n_j + m\lambda} \Big) \Big(1 - \frac{n_{ij} + \lambda}{n_j + m\lambda} \Big) \Big) + (1 - \alpha) \frac{m\lambda}{n_j} \Big\}$$

The variables in the preceding formulas are defined as:

k : number of splits in table

n : number of examples in table

m : number of unique labels

n_j : number of examples in j-th column

n_{ij} : value of cell in row i and column j

λ **and** α : tuning parameters

3 Phase 1: Naive Implementation in Python

For implementing the algorithm the Numpy library was used since it is the de facto standard for handling data in machine learning.

The code mainly consists out of three classes (Splitter, TableManager, FUSINTERDiscretizer) that will be described in the following. After that applications to data and benchmarks will be shown.

3.1 Classes

3.1.1 Splitter

The responsibility of the Splitter is to generate the initial split given already sorted data. So it constitutes the most important part for steps 1 to 4. Its interface only consists of a single public method `apply()` that runs these steps on data.

The most important part of the Splitter is the while loop inside the `apply()` method: Here we iterate over all examples. At each step we check if there are multiple examples at the same value. And if the label of the current value are of the same type (1...m) or if they are mixed (-1). The label of the current value and the next index is returned by `_get_label_of_next_value(index)`. If the label of the current value is different from the label of the last value or if it is mixed, we create a new split point. If the label stays the same, we go on with the loop.

This approach is able to create an array of splitting points in linear time since each example is examined exactly one time.

The code can be found at here

3.1.2 TableManager

The responsibility of the TableManager is to create a table T as described in step 6 and also to merge such tables like in step 6. For this it uses the `create` and `compress_table` methods respectively.

The `create` method takes the data and the splits generated by the Splitter and creates a $m \times k$ matrix for data with m labels and k initial splits. The table is generated by iterating over the data and counting the appearances of all labels in the current interval and then setting them as columns in the matrix. This process examines each example exactly once and hence runs in linear time.

The `compress_table` method takes a matrix like the one generated with the `create` function and an index to create a new table but with the columns i and $i+1$ merged together. So the table

$$T_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

with $i = 1$ becomes

$$T_2 = \begin{bmatrix} 1 & 5 & 4 \\ 5 & 13 & 8 \end{bmatrix}$$

Also this function looks at each example exactly once and hence runs in linear time.

The code can be found here

3.1.3 FUSINTERDiscretizer

The main part of FUSINTER is implemented here. After generating initial splits and an initial table T we iteratively compute which split points to remove. This is done in the `apply()` method.

Here we iteratively compute the value from step 6 for all possible merges and then test if the maximum value is strictly positive like in step 7.

The implementation is as close as possible to the algorithm described above and is in that sense naive. We do many computation multiple times and allocate many unnecessary tables. For n iterations we calculate $n + (n-1) + (n-2) + \cdots + 1$ table merges in the worst case. That gives a run time complexity of $O(n^2)$. This will be a major point for optimization in phase 2 of the project.

Further the implementation of the Shannon's Entropy and the Quadratic Entropy although they are of linear complexity do heavily rely on loops, which is also bad for performance in Python.

The code can be found here.

3.2 Applications

In the following we are applying the FUSINTER algorithm to various datasets and observe the behavior given different values for the parameters α, λ.

3.3 Dataset From the Original Paper

This dataset is taken from the original paper [ZRR98, p. 315 (Figure 5)]. It consists out of integer values being labeled with 2 classes.

We first give results with $\lambda = 1$:

Figure 3: Splits with $lambda = 1$ and $\alpha = 0.95$

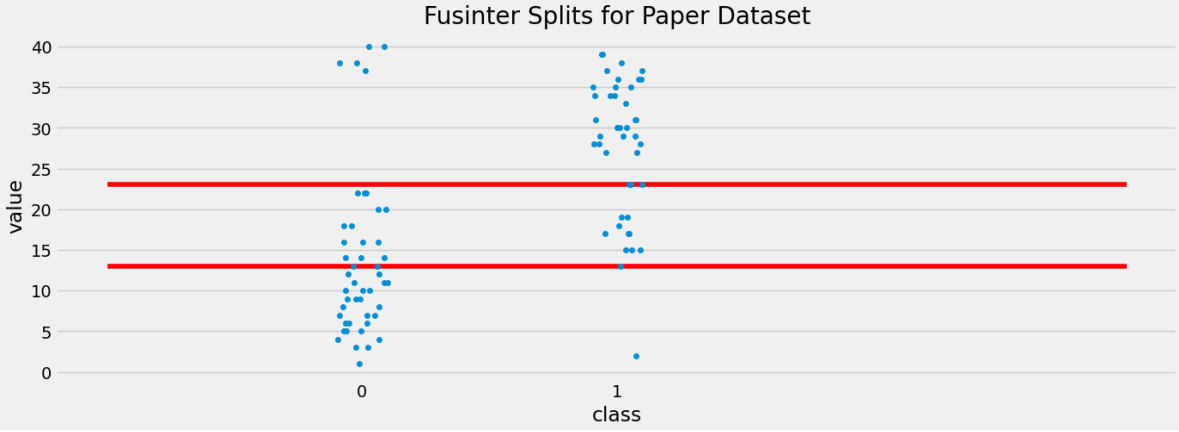

Figure 4: Splits with $lambda = 1$ and $\alpha = 0.85$

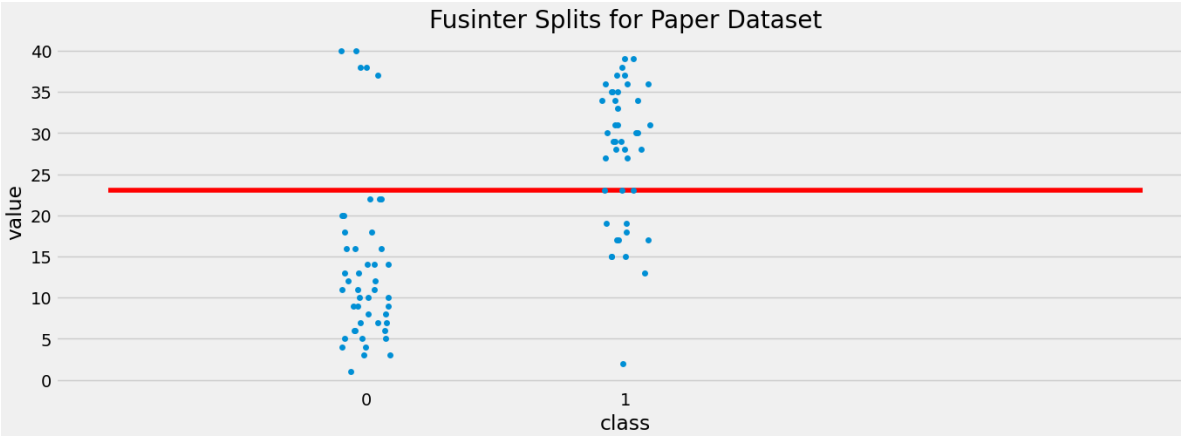

Figure 5: Splits with $lambda = 1$ and $\alpha = 0.8$

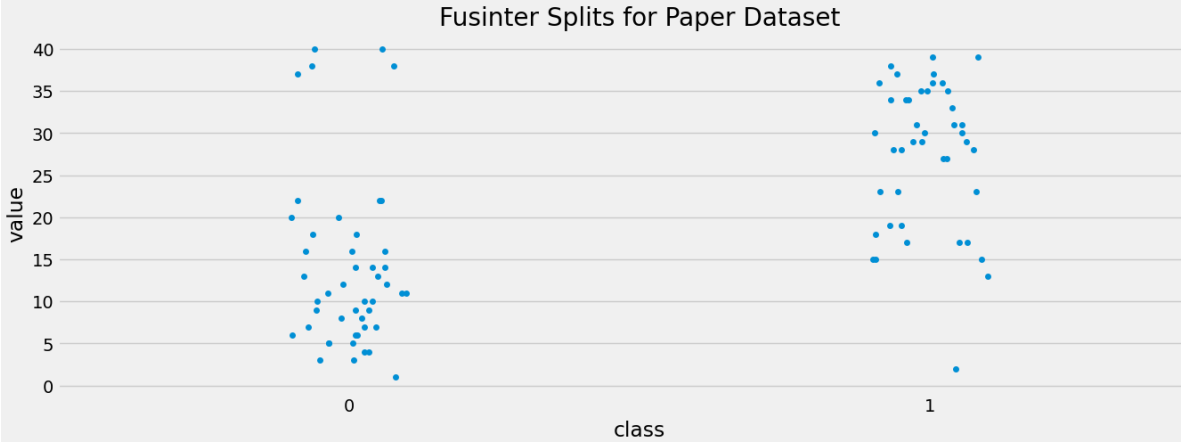

Figure 6: Splits with $lambda = 1$ and $\alpha = 0.25$

We see that the higher the value of α the move splits we are are going to produce.

Now we keep the value $\alpha = 0.95$ and vary the λ:

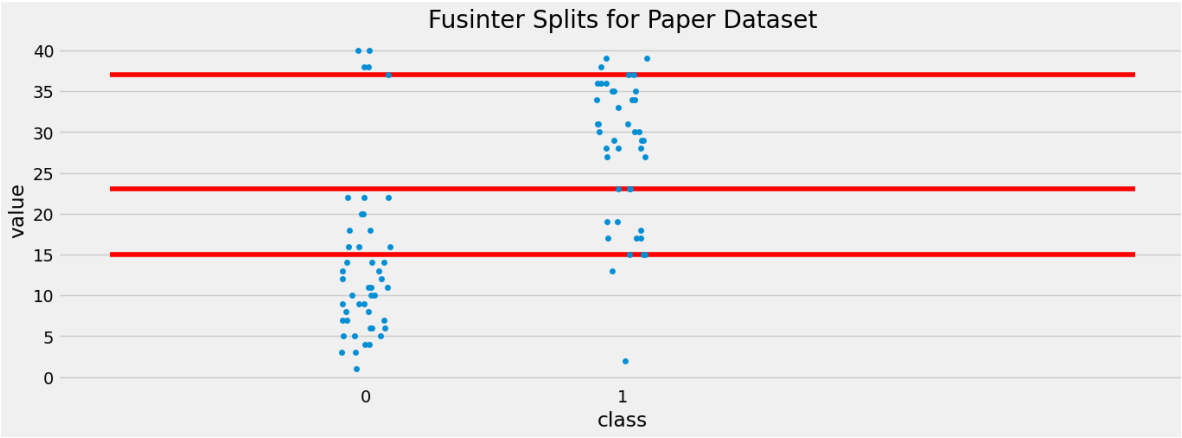

Figure 7: Splits with $lambda = 1$ and $\alpha = 0.95$

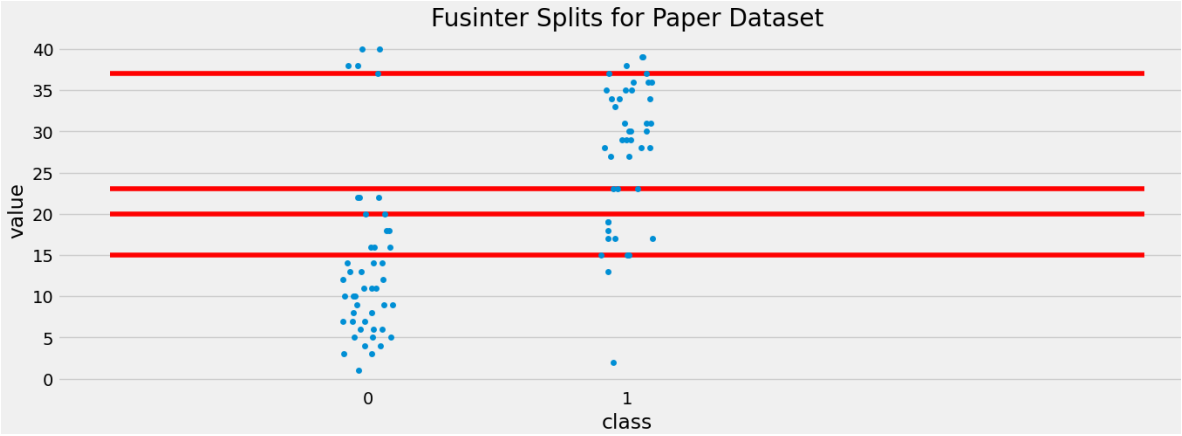

Figure 8: Splits with $lambda = 0.85$ and $\alpha = 0.95$

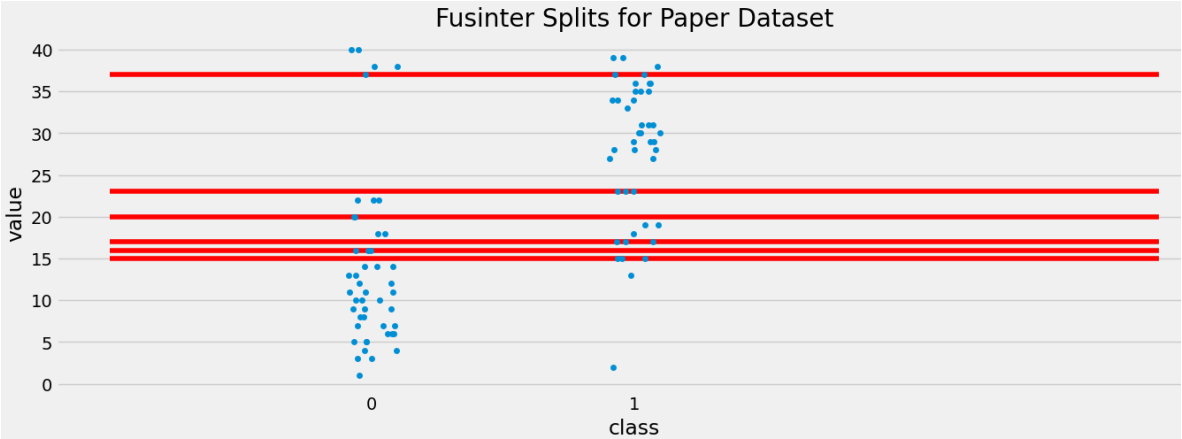

Figure 9: Splits with $lambda = 0.35$ and $\alpha = 0.95$

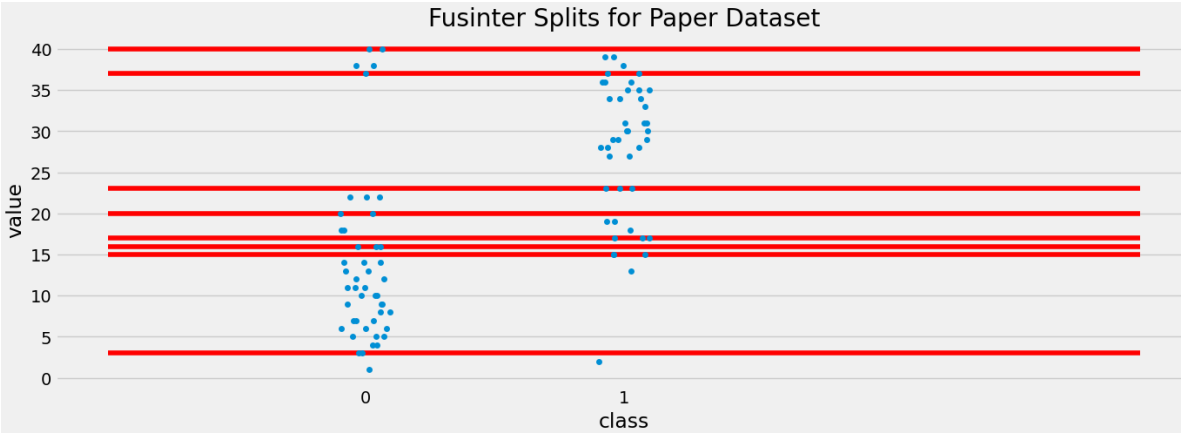

Figure 10: Splits with $lambda = 0.15$ and $\alpha = 0.95$

Figure 11: Splits with $lambda = 0.05$ and $\alpha = 0.95$

We see that the lower the value of λ the less likely the algorithm is to remove initial splits. If it approaches zero all initial splits remain in the output.

The value of *lambda* gives the maximal amount of possible splits and the value of *alpha* then is responsible for removing splits from the possible ones. We see that when we set $\lambda = 0.05$ and $\alpha = 0.45$ the output is equal to $\lambda = 0.1$ and $\alpha = 0.95$

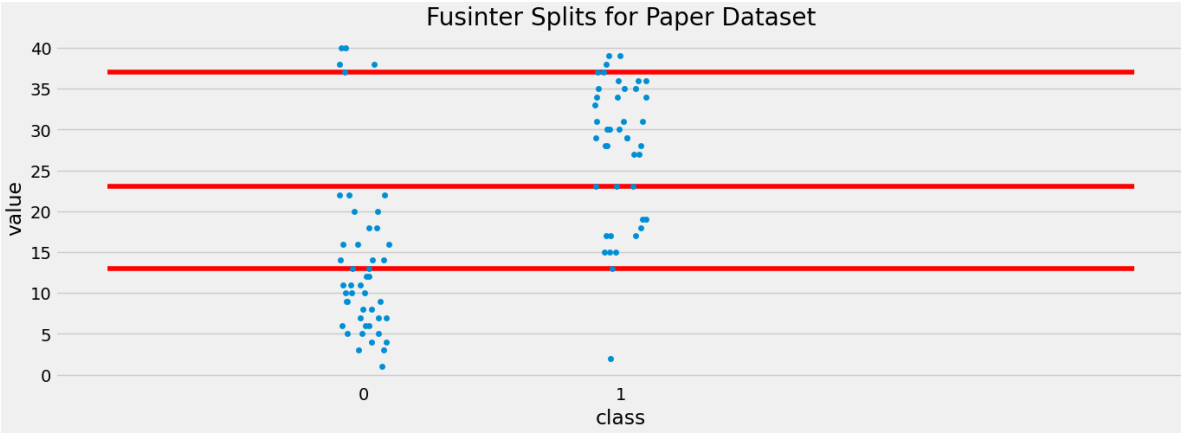

Figure 12: Splits with $lambda = 0.05$ and $\alpha = 0.45$

Heuristic of alpha lambda behavior given the the phi formula

3.4 Iris Dataset

The application of the algorithm to the petal length feature of the iris dataset [iri]:

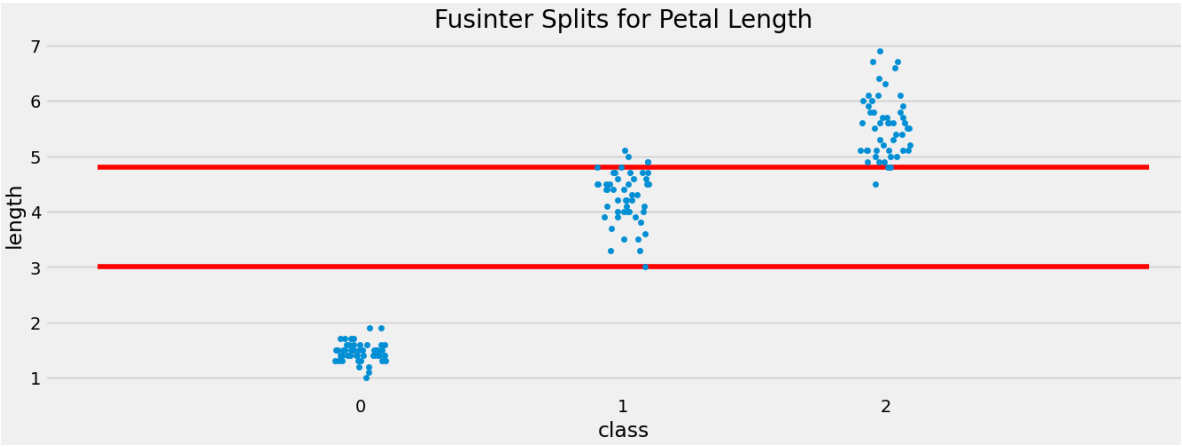

Figure 13: Splits with $lambda = 1$ and $\alpha = 0.95$

3.5 Duplicated Data

As it is stated in the paper, the algorithm should yield zero splits if there is no meaningful discretization possible. [ZRR98, p. 316 (Step 8)] We put that to the test by generating a normal distributed set of datapoints and duplicated it. The first set was labeled 1 the second one was labeled 2.

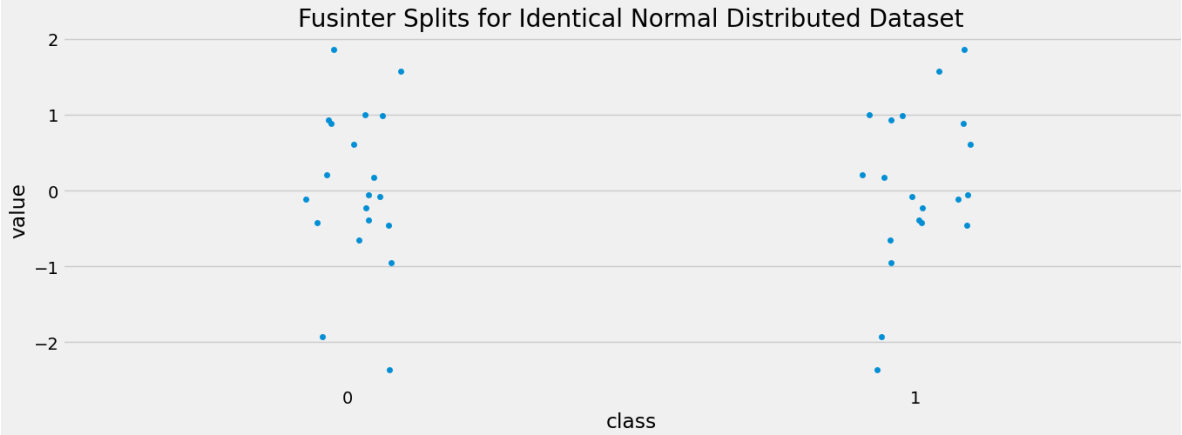

Figure 14: Duplicated Normal Distribution yields no splits for $\lambda \neq 0$

3.6 Benchmarks

I ran benchmarks against the paper dataset [ZRR98, p. 315 (Figure 5)] and extended it by repeating it multiple times from 1 to 10 times. So a multiplication with 1 gives the original dataset with 90 examples, a multiplication with 10 gives the original dataset 10 times repeated resulting in 900 examples. The results were the following:

multiplicator	average time in seconds
1	0.0495
2	0.3541
3	1.1334
4	2.6202
5	5.005
6	8.5336
7	13.5499
8	20.587
9	30.1785
10	40.6007

We see that the time does not scale linearly with the input. To further get insights into the execution of the program I ran it with a profiler.

Name	Call Count	Time (ms)	Own Time (ms)
quadratic_entropy	456000	87682 91.3%	42912 44.7%
<method 'reduce' of 'numpy.ufunc	5818000	16160 16.8%	16160 16.8%
_wrapreduction	5818000	28286 29.5%	8486 8.8%
sum	5816000	37023 38.6%	7415 7.7%
sum	5816000	46846 48.8%	4687 4.9%
<built-in method numpy.core._mul	5870011	41984 43.7%	4333 4.5%
<dictcomp>	5818000	2896 3.0%	2896 3.0%
compress_table	228000	5095 5.3%	2860 3.0%

Figure 15: Output from Profiler

It is obvious that most of the time is spent inside the quadratic entropy function. This is the case because Python is notoriously slow at running loops. In phase 2 we will apply just in time compilation for the entropy functions.

It is also noteworthy that the part we considered to run slow, the loop inside the `apply` method of the FUSINTERDiscretizer does not contribute that much to the runtime. This is probably the case because Numpy is already applying a lot of optimization when it comes to allocate and deallocate memory for the huge amount of tables that are created.

4 Phase 2

4.1 Improvement Idea

The major problem of the naive implementation lies withing the `apply()` method of the `FUSINTER` class.

Algorithm 1 Apply Method of FUSINTER Version 1

Require: *splits*: list of initial splits
Require: *table*: initial table
Require: H: Function to compute the entropy of a given table
  while *splits* is not empty **do**
    merged_tables ← all possible merges of table

    split_values ← empty list
    for *merged_table* ∈ *merged_tables* **do**
      append $H(table) - H(merged\_table)$ to *split_values*
    end for

    max_split_value ← the maximum value in *split_values*
    max_split_value_index the index of the table corresponding to maximum split value

    if at least one value in *split_values* ≥ 0 **then**
      table ← table in *merged_tables* index *max_split_value_index*
      delete element with index *max_split_value_index* from *splits*
    else
      break
    end if
  end while

  return *splits*

Here we compute the split values for all possible splits for each pass of the algorithm. If we have a look at the actual computation of these values we see that many computation steps are redundant. The entropy functions on page 2 are defined over two sums: The inner one computes weighted values for each column (depending on λ, α and the number of elements in the table n as well as the number of rows in the table k. The outer one just adds up those column values to give a scalar value.

The only operation applied on the tables is merging two adjacent columns. That means increasing the values of one column by the values of the other one and then deleting the second one. This neither alters the amount of rows in the table k nor does it alter the amount of elements in the table n.

If we create a class that stores the values n, k we can also store the values computed by the inner loop. Then entropy value of the original table is just the sum of those stored value and the entropy value of a merged table – say with merge index i – is just the entropy value of the original table minus

the stored values with index i and $i+1$ + the entropy value of the merged columns with index i and $i+1$ computed with n and k.

This allows a new implementation of the `apply()` method:

Algorithm 2 Apply Method of FUSINTER Version 2

Require: *splits*: list of initial splits
Require: *table*: initial table
Require: MVC: Class `MergedValueComputer`
  while *splits* is not empty **do**
    $mvc \leftarrow MVC(table)$
    $split\_values \leftarrow mvc.compute\_split\_values$ ▷ returns all possible split values

    $max\_split\_value \leftarrow$ the maximum value in $split\_values$
    $max\_split\_value\_index$ the index of the table corresponding to maximum split value

    if at least one value in $split\_values \geq 0$ **then**
      $table \leftarrow$ merged table from $table$ merged at index $max\_split\_value\_index$
      delete element with index $max\_split\_value\_index$ from *splits*
    else
      break
    end if
  end while

  return *splits*

4.1.1 MergeValueComputer

The `MergeValueComputer` is a class that takes a table created by a `TableManager` together with values for α and λ as input. Then it computes a list of values corresponding to the inner loop of weighted entropy functions as they are described on page 2.
E.g.: For the following input table

5	0	2
3	1	0

we get the following entropy values per column

0.68333859	0.17930737	0.19012986

which gives the table entropy of

1.0527758178799638

now for computing the split value for the first two columns we only need to take the entropy value, subtract the first two stored values from it and add the entropy value of the merged column $[5, 4]$ with the original parameters n, k, λ, α.

This approach saves us from computing the split value for each possible merged table and reduces it to compute the entropy of the original table once for each pass and then reuse most of the column entropies. The only new value that has to be computed is the entropy of the new column created by merging.

4.2 Path not taken

A further improvement is possible by not instantiating a new `MergeValueComputer` in each pass, but to instantiate one at the beginning of `apply()` and then update the stored column entropies depending on the chosen new table. In the end I did not follow this path since the improvements were minimal compared to the ones achieved by the original approach. The simplicity seems to be more important to me than an improvement of some milliseconds.

4.3 Benchmarks

I ran the same tests as for version 1 of the algorithm with the following results:

multiplicator	average time in seconds
1	0.1807
2	0.0113
3	0.0274
4	0.0395
5	0.0591
6	0.0804
7	0.0804
8	0.1437
9	0.1787
10	0.2121

The runtime seems to be very constant. This seems to indicate that the biggest part is used for setting things up. My hopes are that by using C++ these times can be further reduced.

5 Phase 3

References

[iri] https://scikit-learn.org/stable/datasets/toy_dataset.html#iris-dataset. Accessed: 04.11.2023.

[ZRR98] D. A. Zighed, S. Rabaséda, and R. Rakotomalala. Fusinter: A method for discretization of continuous attributes. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(03):307–326, 1998.