Report of Implementation of FUSINTER Algorithm for Practical Work in AI

Florian Jaksik

Supervisor: Dr. Van Quoc Phuong Huynh

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, hence 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 adding them. The first ones are called bottom-up the later one top-down algorithms. FUSINTER falls into the first category.

In the following I will first 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. Then I will show some applications of the algorithm on different datasets. After that descriptions of the optimizations for the second part will follow. After an examination of the implementation in C++ I will end with reporting benchmarks.

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 greater 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 where 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 positive, 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 6 until there are no more splits or the criterion in 7) is not met.

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].

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

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

The variables in the preceding formulas are defined as:

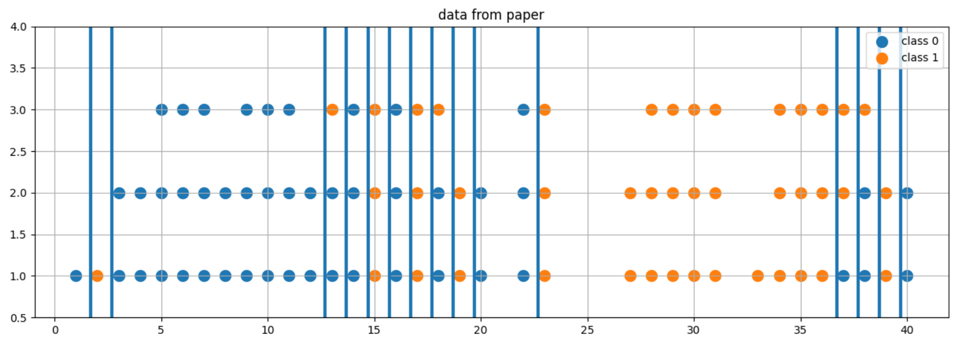

Figure 1: Initial splits created by steps 1-3 of the FUSINTER algorithm

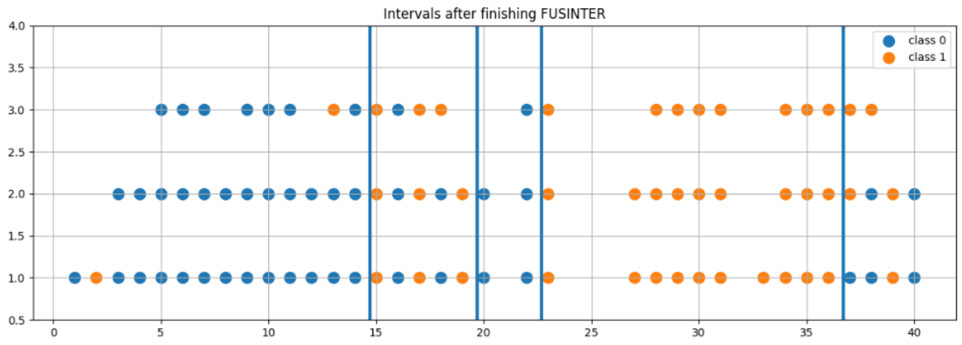

Figure 2: Final splits after completing FUSINTER algorithm

k : number of columns 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

If lambda is set to zero, ϕ_1 is identical to the definition of the shannon entropy. If lambda goes to infinity the fractions in the inner sum tend to $1/m$ and the second term gets larger and larger.

If alpha is set to zero the inner sum vanishes and second term completely defines the results. Since the second term does not depend on the distribution of the labels the results of the inner sum are also less affected by the distribution of examples and more by the amount of labels in each column.

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. Thereafter applications to data 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 essential 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 5 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. 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 run 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 $\lambda = 1$:

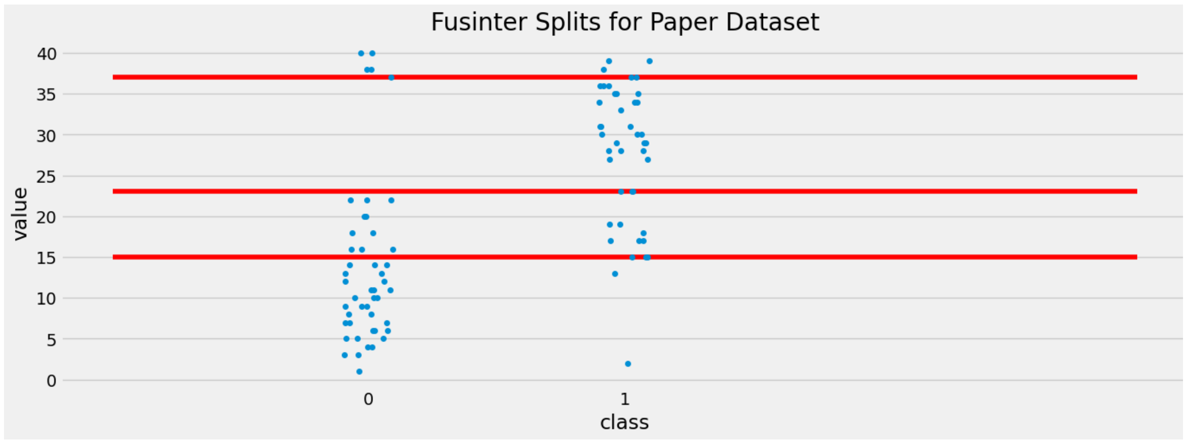

Figure 3: Splits with $\lambda = 1$ and $\alpha = 0.95$

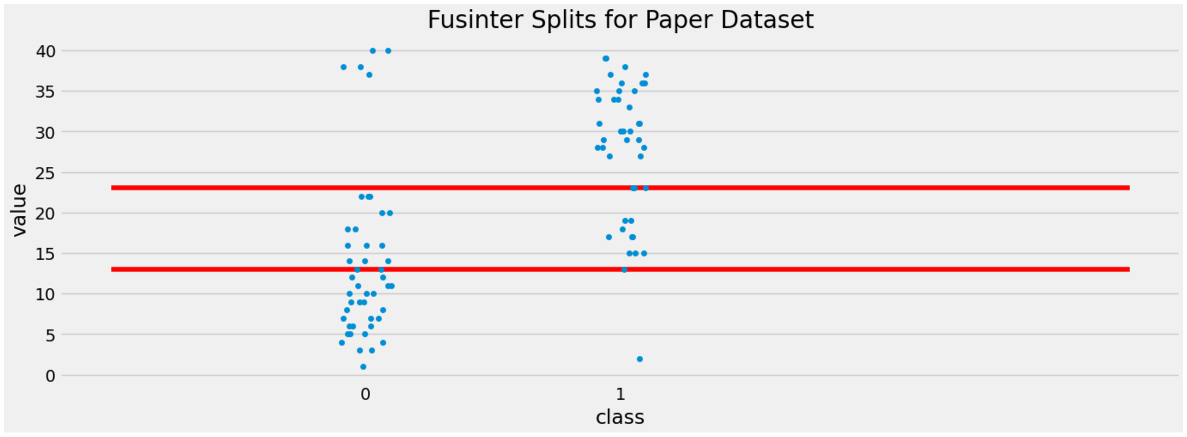

Figure 4: Splits with $\lambda = 1$ and $\alpha = 0.85$

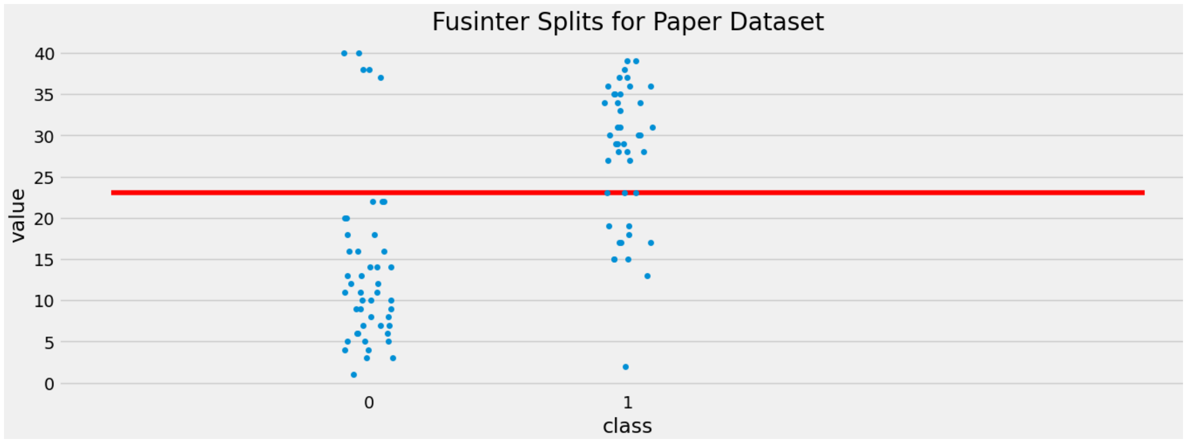

Figure 5: Splits with $\lambda = 1$ and $\alpha = 0.8$

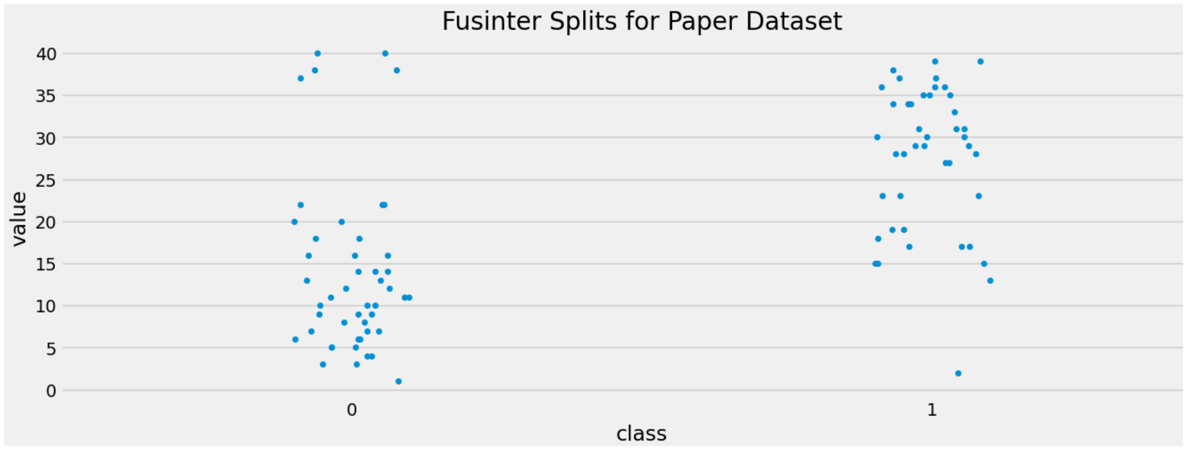

Figure 6: Splits with $\lambda = 1$ and $\alpha = 0.25$

We see that the higher the value of α the more splits are generated.

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

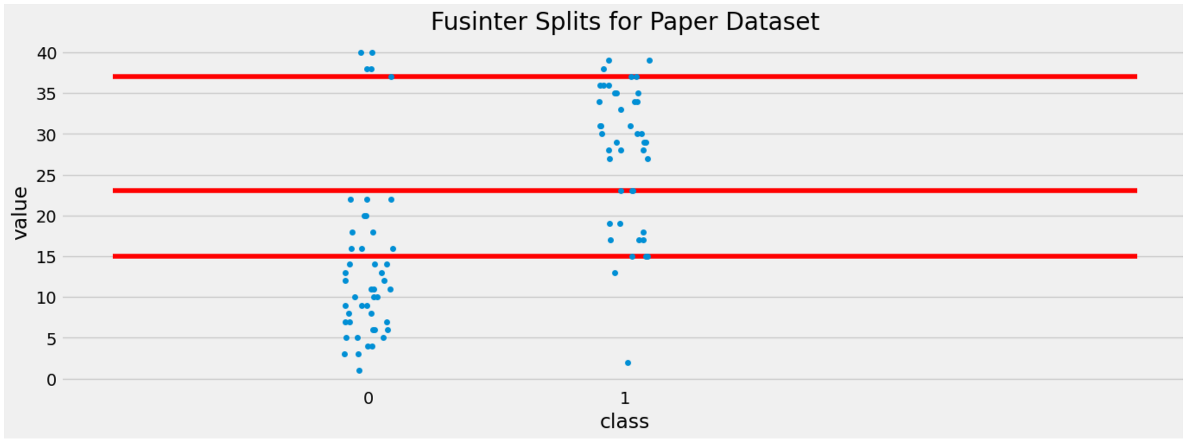

Figure 7: Splits with $\lambda = 1$ and $\alpha = 0.95$

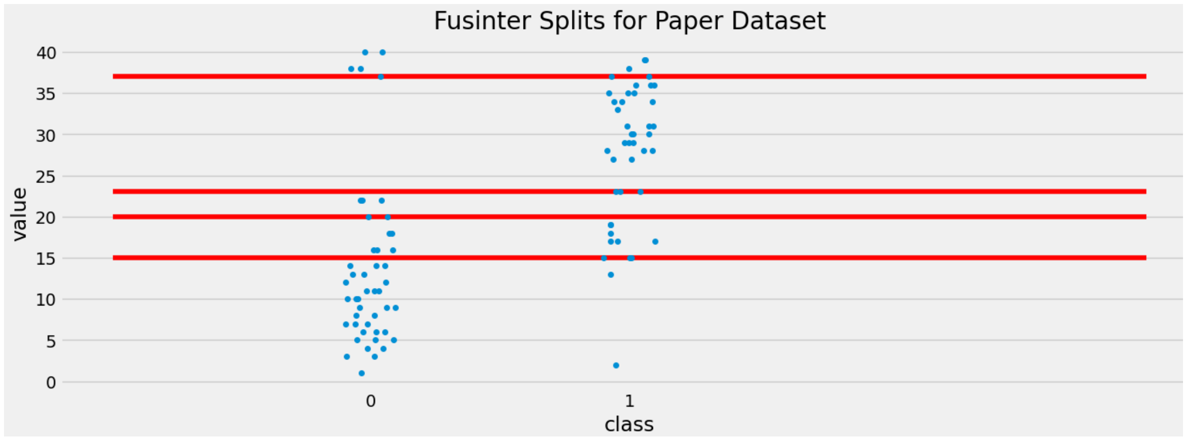

Figure 8: Splits with $\lambda = 0.85$ and $\alpha = 0.95$

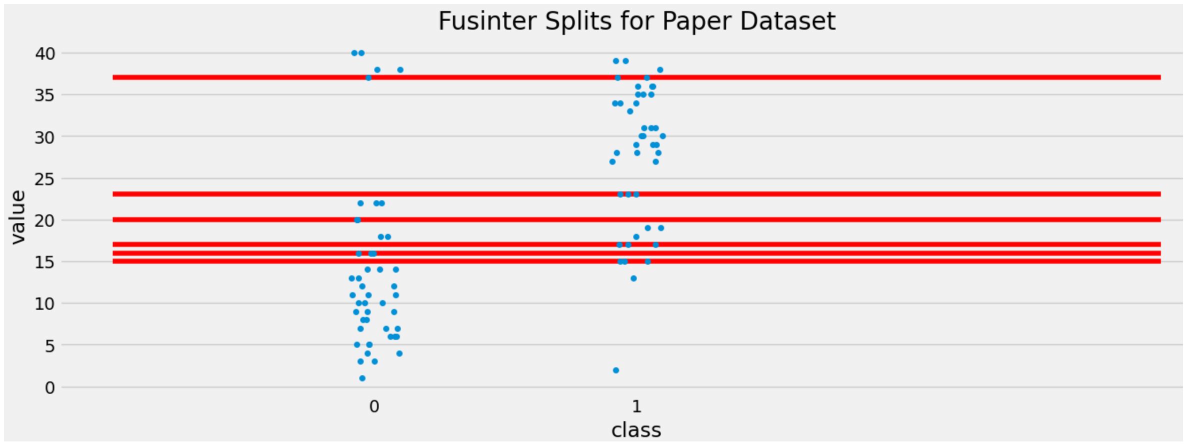

Figure 9: Splits with $\lambda = 0.35$ and $\alpha = 0.95$

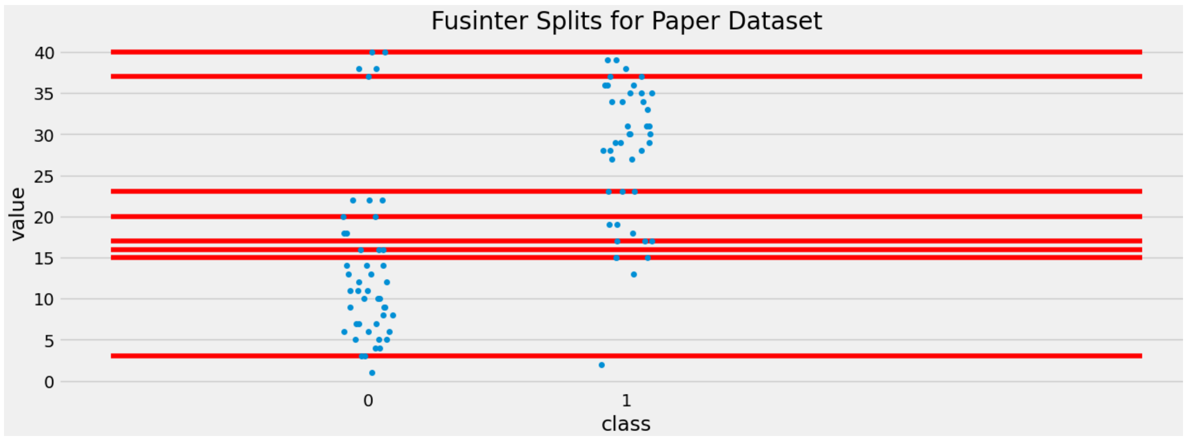

Figure 10: Splits with $\lambda = 0.15$ and $\alpha = 0.95$

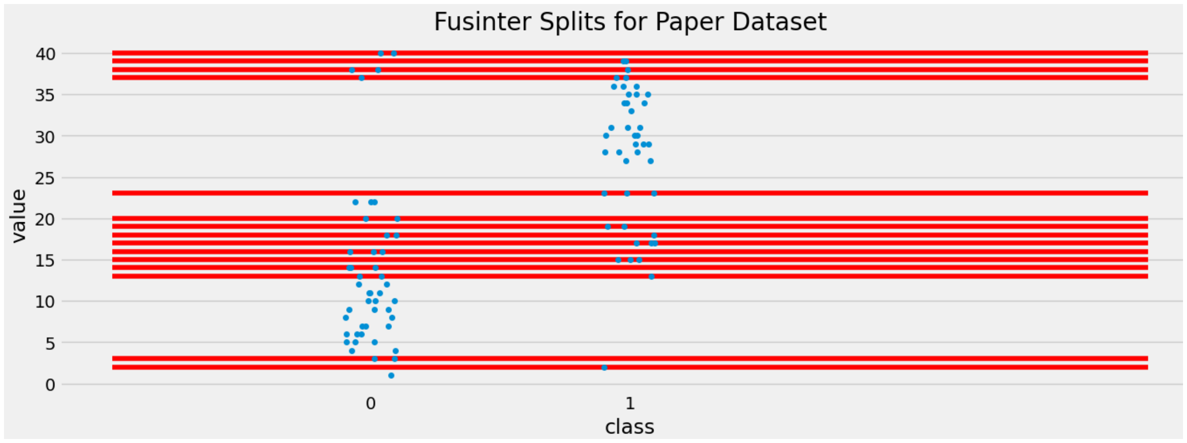

Figure 11: Splits with $\lambda = 0.05$ and $\alpha = 0.95$

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.

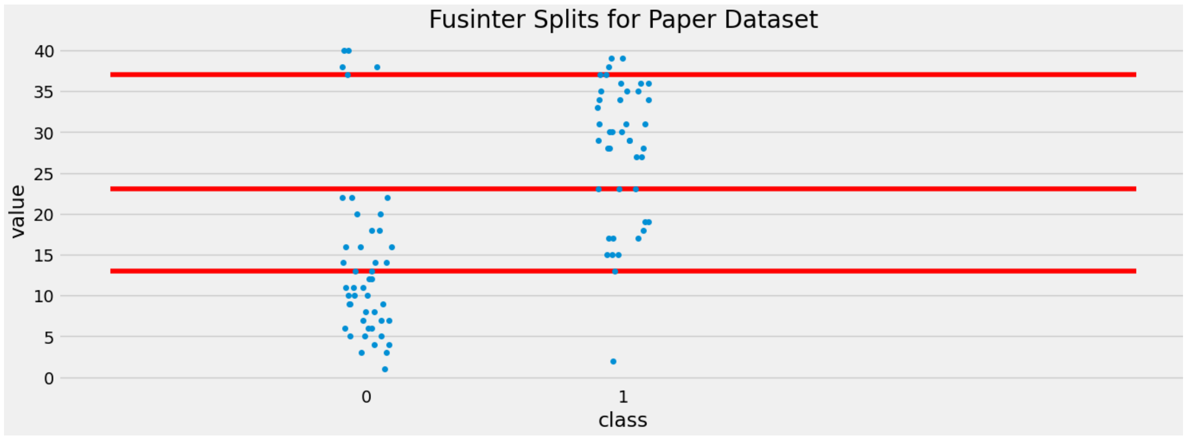

Figure 12: Splits with $\lambda = 0.05$ and $\alpha = 0.45$

3.4 Iris Dataset

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

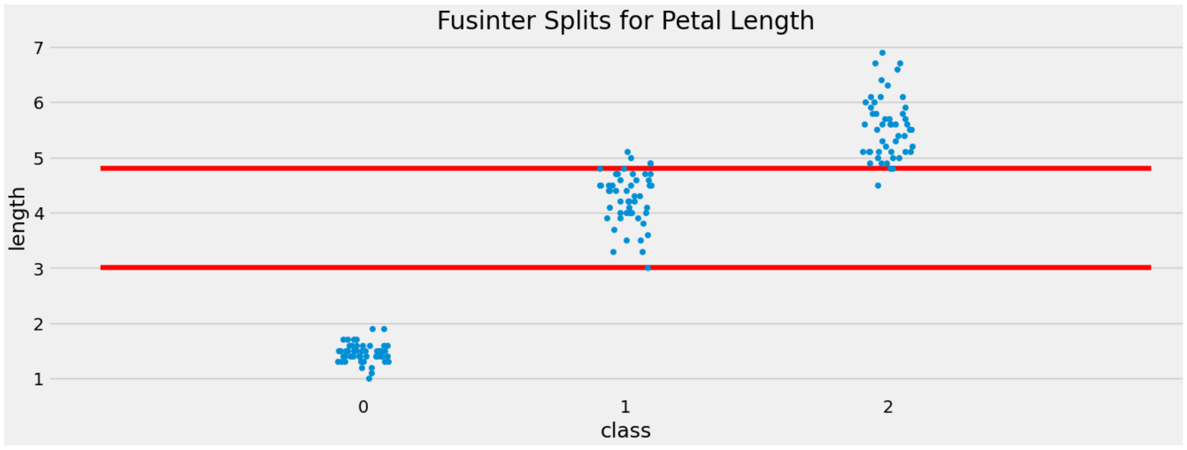

Figure 13: Splits with $\lambda = 1$ and $\alpha = 0.95$

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 is completely labeled 1 the second one is completely labeled 2.

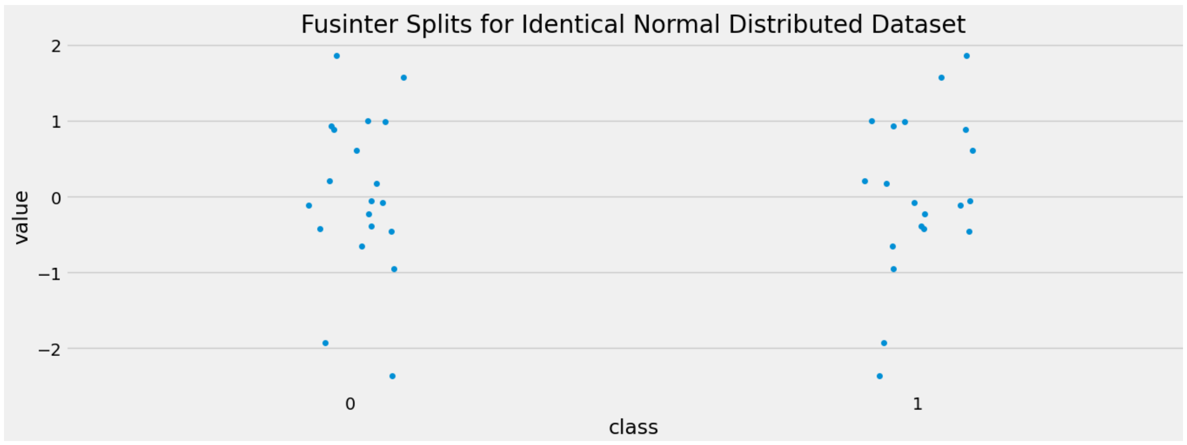

Figure 14: Duplicated Normal Distribution yields no splits.

This is the expected output.

4 Phase 2

4.1 Improvement Idea V1

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

Algorithm 1 Apply Method of FUSINTERDiscretizer Version 1

```
Require: $splits$: list of initial splits
Require: $table$: initial table
Require: $H$: Function to compute the entropy of a given table
  while $splits \neq \emptyset$ do
      $max\_idx = \arg\max_i H(table_i)$        ▷ $table_i$ is the original table merged at index $i$
      if $H(table) - H(table_{max\_idx}) \geq 0$ then
          $table = table_{max\_idx}$
          delete $splits[max\_idx]$
      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 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 values.

The only operation applied on the tables is merging two adjacent columns. 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 calculate the entropy value for each column separately and store them too. To compute $H(table) - H(table_i)$ we simply add up those columns except for the columns i and $i+1$ and then add the entropy value of the column created by merging columns i and $i+1$.

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

Algorithm 2 Apply Method of FUSINTERDiscretizer Version 2.1

```
Require: $splits$: list of initial splits
Require: $table$: initial table
Require: $MVC$: Class MergedValueComputer
  while $splits \neq \emptyset$ do
      $mvc = MVC(table)$
      $max\_val = \max mvc.compute\_split\_values()$
      $max\_idx = \arg\max mvc.compute\_split\_values()$
      if $max_val \geq 0$ then
          $table = table_{max\_idx}$          ▷ $table_i$ is the original table merged at index $i$
          delete $splits[max\_idx]$
      else
          break
      end if
  end while

  return $splits$
```

The code can be found here (FUSINTERDiscretizer) and here (MergeValueComputer)

4.1.1 MergeValueComputer V1

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

Labels in Intervals		
5	0	2
3	1	0

we get the following entropy values per column

Column Entropy Values		
0.68333	0.17931	0.19013

which gives the table entropy of

Total Entropy
1.05277

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. The only new value that has to be computed is the entropy of the new column created by merging.

4.2 Improvement Idea V2

The described approach reduced the runtime, but it also requires the computation of the entropy values of all possible merges in each iteration of the while loop. We can reduce this even further by observing that if the merge takes place at index i, only possible merge values at $i-1$ and $i+1$ are affected. So most of the entropy values stay the same over one iteration.

For this I implemented a new version of the `MergeValueComputer` that stores the entropies of the given table and then computes the values $\Delta_i = H(table) - H(table_i)$ by the method used in `V1`. Then after a merge the new table is given to the `update()` method to remove the delta corresponding to the merge index and also update the neighboring Δ's of the merge index.

For the paper dataset we compute the input table with the following entries.

Labels in Intervals				
1	0	26	2	3
0	1	0	1	0

The `MergeValueComputer` computes the following column entropies and stores them:

Column Entropy Values				
0.12566	0.12566	0.16533	0.11472	0.09385

The deltas following from this are:

Column Delta Values			
0.14543	0.01415	-0.00304	0.05589

The `FUSINTER` algorithm forces us to remove the maximum of the deltas. In this case it is the first one. This gives a new table:

Labels in Intervals			
1	26	2	3
1	0	1	0

When given to the *update()* method of the new `MergeValueComputer` it updates the entropies as follows:

Column Entropy Values			
0.12566	0.16533	0.11472	0.09385

The deltas are computed as:

Column Delta Values		
0.18511	-0.00304	0.05589

In this example only the second element has to be updated, all other entropies and deltas stay the same. If the maximum delta lies between two other values, the predecessor and successor have to be recalculated. If it lies at the end, only the predecessor needs to be recomputed.

For this to work we also need to keep the instance of the *MergeValueComputer* in the *FUSINTER-Discretizer*'s while loop.

Algorithm 3 Apply Method of FUSINTERDiscretizer Version 2.2

```
Require: $splits$: list of initial splits
Require: $table$: initial table
Require: $MVC$: Class MergedValueComputer
  $mvc = MVC(table)$
  while $splits \neq \emptyset$ do
    $max\_val = \max mvc.compute\_split\_values()$
    $max\_idx = \arg\max mvc.compute\_split\_values()$
    if $max\_val \geq 0$ then
      $table = table_{max\_idx}$                ▷ $table_i$ is the original table merged at index $i$
      delete $splits[max\_idx]$
      $mvc.update(table, max\_idx)$
    else
      break
    end if
  end while

  return $splits$
```

This approach makes it possible to keep the already computed deltas and only recompute the ones that will change according to the table merge.

The code can be found here (FUSINTERDiscretizer) and here (MergeValueComputer)

5 Phase 3

For the third part I implemented the optimized version V2 in C++. The goal was to be able to call the code from within Python and the execution time to be superior to V2 implemented in Python.

The classes were implemented in a header only style and are named identical to the to the Python versions: FUSINTERDiscretizer, TableManager, Splitter, MergeValueComputer.

To make the code callable from Python I used the Pybind11 library. The goal of Pybind11 is to expose "[...] C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. "[Pyb] which together with a relatively straight forward interface made it a good choice for this project.

Since the working of the code is – with the exception of some C++ idioms – identical to the Python code I will focus on the implementation of the wrapping approach. First I added the Pybind11 repository as a git submodule to the project to circumvent the usage of system wide imports. Since I am working with CMake to compile the project I had to declare the submodule in the `CMakeLists.txt`:

```
add_subdirectory(external/pybind11)
```

Further I needed to set variables so that the current virtual environment is used for compilation instead of the global Python version:

```
set(Python_VIRTUALENV FIRST)
```

Finally I had to tell CMake where to find the Pybind11 cpp file that governs the compilation of the library:

```
pybind11_add_module(FUSINTER_v3_pybind pybind11.cpp)
```

The `pybind11.cpp` file only imports the Pybind11 header files and then defines the objects and modules that are to be compiled. In my case that was just the `FUSINTERDiscretizer` file:

```
PYBIND11_MODULE(FUSINTER_v3_pybind, m) {
    pybind11::class_<lib::FUSINTERDiscretizer>(m, "FUSINTERDiscretizer")
            .def(pybind11::init<float, float>())
            .def("fit", &lib::FUSINTERDiscretizer::fit)
            .def("transform",&lib::FUSINTERDiscretizer::transform);
}
```

Here we define the Python module `FUSINTER_v3_pybind` that contains the class `FUSINTERDiscretizer` and its methods.

For compilation I created an empty build directory and called CMake to generate the according Makefile:

```
    cmake -DCMAKE_BUILD_TYPE=Release -S. -B build
```

After the Makefile is generated we can `cd` into the build directory and generate a shared library object by calling `make`. For my system that yields a `FUSINTER_v3_pybind.cpython310x86_64linux-gnu.so` file that can be imported and used in Python:

```
    import FUSINTER_v3_pybind
    discretizer = FUSINTER_v3_pybind.FUSINTERDiscretizer(alpha, lambda)
```

6 Interface

- $FUSINTERDiscretizer(alpha : float,\ lambda : float) \rightarrow \text{instance of discretizer} :$ Instantiates the discretizer
- $discretizer.fit(x : \text{array of floats},\ y : \text{array of integers}) \rightarrow \text{array of splits} :$ learns a discretization based on alpha, lambda the data x and the labels y.
- $discretizer.transform(x : \text{array of floats}) \rightarrow \text{array of integers} :$ applies the learned discretization to the data x

7 Benchmarks

I ran the algorithms against three datasets: The dataset from the paper[ZRR98, p. 316] with 90 examples and 1 continuous feature; The Waveform Database Generator[BS88] with 5000 examples and 21 continuous features; The Covertype Dataset[Bla98] with 581012 examples and 10 continuous out of 52 total features; The SUSY Dataset[Whi14] has 5000000 examples and was testes for only the first feature.

I ran the algorithm over all continuous features and got the following results. The results are given in seconds and OOT stands for out of time, which occurred for run times longer than two hours:

	paper	wave	covertype	susy
python w/o optimization	0.0538	OOT	OOT	OOT
python w optimization V1	0.8452	101.0536	810.8134	OOT
python w optimization V2	0.2749	5.5638	39.0544	OOT
C++	0.0003	0.0543	0.82	3762

References

[Bla98] Jock Blackard. Covertype. UCI Machine Learning Repository, 1998. DOI: https://doi.org/10.24432/C50K5N.

[BS88] L. Breiman and C.J. Stone. Waveform Database Generator (Version 1). UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C5CS3C.

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

[Pyb] Pybind. Pybind/pybind11: Seamless operability between c++11 and python.

[Whi14] Daniel Whiteson. SUSY. UCI Machine Learning Repository, 2014. DOI: https://doi.org/10.24432/C54606.

[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.