CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Explainability of malware classsifiers

A surrogate model explanation approach applied to hierarchical multiple-instance data

Ondřej Vereš
Open Informatics
Cybersecurity

May 2024
Supervisor: Prof. Ing. Václav Šmídl, Ph.D

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Vereš Ondřej**	Personal ID number:	**492095**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Cyber Security**		

II. Master’s thesis details

Master’s thesis title in English:

Explainable classifiers of malware

Master’s thesis title in Czech:

Vysvětlitelné klasifikátory malwaru

Guidelines:

1. Describe hierarchical multi-instance data and summarize existing approaches for supervised learning from such data. Pay special attention to libraries Mill.jl and JsonGrinder.jl.
2. Train a classifier for data from the Avast-CTU Public CAPEv2 Dataset including hyper-parameter search. Train an alternative classifies (e.g. feature-based) on the same data.
3. Write a review of existing approaches for model explanations and critically review their suitability for their application to explanations of the HMil data. Choose the most promising methods for implementation.
4. Implement selected method within existing library ExplainMill.jl. Apply the method to the trained classifiers of the CAPEv2 data. Analyze performance of the methods on selected samples. Compare the results of the method with explanations of the alternative classifiers. Discuss advantages and disadvantages of the tested approaches.

Bibliography / sources:

1. Bosansky, B., Kouba, D., Manhal, O., Sick, T., Lisy, V., Kroustek, J. and Somol, P., 2022. Avast-CTU public CAPE dataset. arXiv preprint arXiv:2209.03188.
2. Mandlík, Š., Račinský, M., Lisý, V. and Pevný, T., 2022. JsonGrinder. jl: automated differentiable neural architecture for embedding arbitrary JSON data. The Journal of Machine Learning Research, 23(1), pp.13508-13512.
3. Pevný, T., Lisý, V., Bošanský, B., Somol, P. and Pěchouček, M., 2022. Explaining Classifiers Trained on Raw Hierarchical Multiple-Instance Data. arXiv preprint arXiv:2208.02694.

Name and workplace of master’s thesis supervisor:

doc. Ing. Václav Šmídl, Ph.D. Artificial Intelligence Center FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **23.01.2024** Deadline for master's thesis submission: ___________

Assignment valid until: **21.09.2025**

doc. Ing. Václav Šmídl, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgement / Declaration

I would like to sincerely thank my supervisor Prof. Ing. Václav Šmídl, Ph.D, and his colleagues doc. Ing. Tomáš Pevný, Ph.D. and Ing. Šimon Mandík for their continuous guidance and expertise which countlessly helped me to overcome challenging problems throughout the development of this work.

The access to the computational infrastructure of the OP VVV funded project Research Center for Informatics is also gratefully acknowledged.

Lastly, I would like to thank my family and friends for their support.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 24. May 2024

. . . .

Abstrakt / Abstract

Tato práce představuje metodu TreeLIME pro vysvětlování hierarchických víceinstančních modelů strojového učení. Naše metoda natrénuje zjednodušený logistický model pro konkrétní vstupní vzorek. TreeLIME následně vytvoří vysvětlení interpretací tohoto zjednodušeného modelu.

Zjistili jsme, že náš původní logistický model nebyl dobře interpretovatelný, protože některé prediktory byly na sobě částečně závislé. Proto jsme vyvinuli novou vrstvenou verzi TreeLIME, která trénuje logistický model pro každou vrstvu hierarchického víceinstančního vzorku zvlášť. Optimalizační proces obou verzí TreeLIME jsme zároveň vizualizovali, aby bylo snadnější ho pochopit a ověřit jeho korektnost.

Provedli jsme sérii experimentů a citlivostních studií pro různé hyperparametry metody TreeLIME na CAPEv2 [1] datasetu, který obsahuje tisíce skenů škodlivých souborů ve formátu JSON rozdělených do deseti tříd dle typu malwaru.

Výsledky vrstveného TreeLIME se výrazně zlepšily a byly srovnatelné s nejlepšími metodami pro vysvětlování hierarchických víceinstančních modelů v současné době.

Pro další zlepšení metody TreeLIME jsme identifikovali problémy s vrstveným přístupem a navrhli další vylepšení ploché verze TreeLIME pro pokračování v této práci.

Klíčová slova: vysvětlitelnost; hmil; malware;

This thesis introduces a method named TreeLIME to explain hierarchical multiple-instance machine learning models. Our method trains a surrogate logistic regression model for a specific input sample. Afterward, TreeLIME generates an explanation by interpreting the surrogate model.

We discovered that the initial Flat TreeLIME implementation suffered from correlations among the surrogate model predictors. Therefore, we developed an improved Layered TreeLIME method, which trains a surrogate model for each hierarchical multiple-instance data sample layer. Furthermore, we visualized the TreeLIME optimization process to make it more understandable and verify that it works correctly.

We conducted rigorous experiments and a sensitivity analysis for various hyperparameters of the TreeLIME method on the CAPEv2 [1] dataset, which contains JSON reports of thousands of malicious files divided into ten different malware classes.

The performance of the Layered TreeLIME improved dramatically and was comparable to the current state-of-the-art methods of explaining hierarchical multiple-instance models.

To improve TreeLIME further, we have identified issues in the Layered TreeLIME approach and suggested additional improvements to the Flat TreeLIME for future work.

Keywords: interpretability; hmil; malware;

Contents /

Tables / Figures

Chapter 1
Introduction into Explainability

This chapter will focus on why explainability is useful, what explainability is, and what are the most common ways approaches to explainability. The terms explainability and interpretability are used interchangeably in this work.

1.1 Why do we need explainability?

As we seek to apply machine learning in more complex applications, the underlying machine learning models have to get increasingly complicated. Complexity allows the model to understand and solve more complex tasks. On the other hand, complexity makes the model more difficult to understand. The model's decision is almost worthless without explanation in many fields, such as medicine, law, and security. Moreover, explaining the model's decision is helpful in virtually any field. Similar conclusions were made, for example, in [2].

If the model is perfect in every way, we would not need to understand the reasons behind its decisions. However, the models are often imperfect, for instance, due to flaws in training data or insufficient variety in training data.

Machine learning models make mistakes, and explainability offers a tool to learn why. This knowledge is then crucial for fixing those problems.

For example, the model might be highly sensitive to some unimportant features correlated with an important feature only in the training data but not in the real-world data. For example, the timestamp of a scan might be correlated with a label of the sample because, in the training data, some classes were scanned before others. In these situations, explaining the model is helpful because it presents what the model is sensitive to. If it is sensitive to the wrong features, we can fix it by, for instance, removing codependency from training data.

Without any methods to explain the model, the explanation would look like an enormously long list of mathematical operations with the input data and the model's parameters. That is not very helpful for humans.

1.2 Accuracy and explainability tradeoff

As was mentioned in section 1.1, as the models get more complex, they are harder to interpret. This can be understood as a tradeoff between accuracy and explainability. The more complex the model is, the more accurate it can get and the less explainable it is. On the other hand, the less complex the model is, the less accurate it can get and the more explainable it is. This tradeoff is shown in Fig. 1.1. Accuracy and explainability tradeoff is further discussed in [3], which features a similar Figure to 1.1.

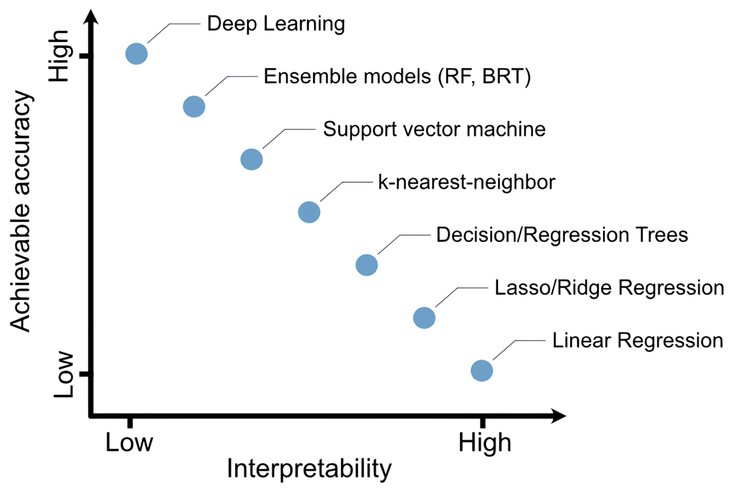

Figure 1.1. Visualization of a trade-off between accuracy and interpretability. On the bottom right are simple models, which are easier to explain, and on the top right are more complex models, which are harder to interpret. This chart is a reproduction from [4].

1.3 Model-based explanation

Model-based explanation approach uses interpretable models. These models are on the bottom right of Fig 1.1. We can derive an explanation by looking at the model's parameters. For instance, if we have a linear model, we can look at the weights of the model and their corresponding features to see which are more impactful and which are less impactful.

This approach limits the model's complexity, making it unusable for more complicated problems.

1.4 Post-hoc explanation

Post-hoc explanation techniques can be used for any model, but they are most commonly used with models from the top left part in Fig. 1.1.

These methods can be divided into black-box and white-box approaches regarding how we treat the model.

- **Black box approach** - The method does not access the model's internal parameters. It can only observe the model's input and output. These methods can be used for any model and are called **model-agnostic**. For further reading about the Black box approach see [5].
- **White box approach** - The method can access the model's internal parameters and structure. That means it can access, for example, gradients. These methods are called **model-specific**, which means they can be used only for specific models or some defined set of viable model types. For example, one possible white box approach is presented in [6].

1.5 What is an explanation of the model's decision?

An explanation of the model's decision might be a selection of the most essential input features. For instance, in image classification, the explanation would be a selection of the most essential superpixels out of all superpixels in the image. In other domains, the explanation should be a small enough subset of the original data to be comprehensible by humans. Explanations can be consistent or inconsistent.

An explanation might also be an interpretable model or a counterfactual sample, as we will see in the following sections.

1.6 Consistency

Explanations can be divided into two categories: consistent and inconsistent explanations.

- **Consistent explanation** - A consistent explanation is classified into the same class as the original sample. These explanations are preferred because they are less likely to be misleading.
- **Inconsistent explanation** - An inconsistent explanation is classified into a different class than the original sample. These explanations are less preferred than consistent explanations because they could be misleading.

In our work, we will be working only with consistent explanations.

1.7 Difference between global and local explanations

Another two types of explanations are local explanations and global explanations.

- **Global explanations** - Global explanations can be applied universally to any sample. Any feature marked as important is important for every input sample. One approach that uses global explanations was documented in [7].
- **Local explanations** - Local explanations explain the model's decision in a small neighborhood of the given data sample. This means that features selected as significant are not important for all input data samples but for similar input data samples that are in a close neighborhood to the original sample. Many methods use local explanations, for example, [8–10].

In our work, we will only work with local explanations as our models are too complicated to be explained globally.

1.8 Input data type

We can divide the explanation methods by the input data type they support. Different methods may be optimal for different data types.

Important for our work are these data types

- **Vectors/Matrices** - used for example in in [8–9].
- **Graphs with cycles** - used for example in [10].
- **Trees** - used for example in [11].

1.9 Explanation types

Explanation can take different forms and types. The most common explanation types are:

- **Feature importance** - This explanation type selects the most important part of the input. These features had the most significant impact on the model's decision. For example, if the model takes images as input, the explanation could be the most significant pixels or superpixels [12] of the image. This type was discussed, for example, in [13].
- **Counterfactual explanation** - This explanation is a new data sample or new data samples close to the original data sample, which is classified as a different class. This is an interesting alternative to the classical explanation approach. On one hand, counterfactual explanations are easy to understand. On the other hand, however, the amount of information they provide is limited. This method can generate several different counterfactual explanations to provide more information. Further reading about counterfactual explanations can be found in [14].
- **Surrogate model** - This method generates a new, simple and interpretable model, which is called **Surrogate model** . This surrogate model is approximately similar to the original model in the local area around the input data sample. Afterward, the model works as an explanation because of its simplicity and interpretability. Simple models, which can work as surrogate models, are sometimes called glass-box models. For example, the explanation method LIME [8] uses a surrogate model as an explanation.

Chapter 2
Hierarchical Multiple Instance Learning

This work is concerned with the explainability of Hmil models. Therefore, we must cover what Hmil is and why it is useful. Note that the terms instance and observations are going to be used interchangeably.

Hierarchical Multiple Instance Learning was introduced in [15] by Tomáš Pevný and Petr Somol in 2017.

As stated in Mill.jl [15] documentation, Multiple instance learning occurs when the sample x is a set of observations. Due to the properties of a set, the order of the items does not matter. This implies that all permutations of the same set should be treated equally. As stated in [11], hmil reflects the structure of data into the model.

The following example illustrates the difference between multiple-instance learning problems and standard fixed size vector based problems.

We want to analyze an application and predict if it might be dangerous. We decidee to analyze only the ports opened by the application. The application might not open any port, or it might open many ports. This would not be easy to represent as a vector of fixed size. However, representing the open ports as a set without fixed size or any inherent order of features is a more accurate representation of reality. Therefore hmil approach will likely yield better results.

To add more complexity to the problem, imagine we want to keep track of every packet received on that port. Now, we would have a set (ports and their packets) of sets (set of packets). Fixed-size feature vectors are not suitable for representing this information. Whereas, hmil natively supports this hierarchical structure.

2.1 Hmill node overview

In Hierarchical multiple-instance learning, we can represent a hierarchy by constructing a tree of nodes. Nodes can have the three following types:

- Array nodes
- Bag nodes
- Product nodes

Inner nodes of the tree representation can be either Bag nodes or Product nodes and leaf nodes are always Array nodes.

2.1.1 Array nodes

Array nodes store information about multiple observations or a single observation. Array nodes are represented as matrices. Each instance is represented as a column. Therefore, the number of columns is the same as number of observations.

The number of rows depends on the type of data stored in the observations. It is depicted in Table 2.1.

Type of data	Data representation	Number of rows
Noncategorical scalar	Scalar value	1
Noncategorical string	N-gram vector	Length of the n-gram vector
Categorical variable	One hot vector	Number of categories

Table 2.1. The table describes the number of rows based on the data type stored in observations.

The definition of the noncategorical or categorical variable depends on the number of unique values it has. The threshold can be set manually.

2.1.2 Bag nodes

Bag nodes contain a set of nodes or observations. Children of Bag nodes always have the same structure. Since leaf nodes cannot be Bag nodes, Bag nodes always have a child node. Moreover, Bag nodes always have a single child node. Bag nodes carry the following information:

- **Child node** - Child node might be an Array node, Product node, or Bag node.
- **Information about bags of the Bag node** - This information maps observations to different bags

Bag node can contain several bags. The information about bags of the Bag node includes how many bags it contains and which observations from the child node belong to which bag.

The following example will illustrate the need for multiple bags in the same Bag node.

We have a set of atoms, each with a set of bonds. These bonds have some properties, as illustrated in Figure 2.1.

```
ProductNode  # 1 obs, 176 bytes
  ├──────── lumo: ArrayNode(1×1)  # 1 obs, 53 bytes
  ├──────── inda: ArrayNode(3×1)  # 1 obs, 77 bytes
  ├──────── logp: ArrayNode(1×1)  # 1 obs, 53 bytes
  ├── mutagenic: ArrayNode(3×1)  # 1 obs, 77 bytes
  ├──────── indl: ArrayNode(3×1)  # 1 obs, 77 bytes
  ╰─────── atoms: BagNode  # 1 obs, 176 bytes
                    ╰── ProductNode  # 27 obs, 104 bytes
                          ├────── element: ArrayNode(8×27)  # 27 obs, 207 bytes
                          ├──────── bonds: BagNode  # 27 obs, 544 bytes
                          │                  ╰── ProductNode  # 68 obs, 56 bytes
                          │                        ├────── element: ArrayNode(8×68)  # 68 obs, 412 bytes
                          │                        ├── bond_type: ArrayNode(7×68)  # 68 obs, 412 bytes
                          │                        ├─────── charge: ArrayNode(1×68)  # 68 obs, 388 bytes
                          │                        ╰── atom_type: ArrayNode(37×68)  # 68 obs, 412 bytes
                          ├─────── charge: ArrayNode(1×27)  # 27 obs, 183 bytes
                          ╰── atom_type: ArrayNode(37×27)  # 27 obs, 207 bytes
```

Figure 2.1. Representation of a molecule in Mill.jl. Visualization was generated using [16].

The `atoms` Bag node has only one bag, including all 27 observations. However, the `bonds` Bag node has many bags, and each bag corresponds to the bonds of a single atom.

```
julia> ds.data.atoms.bags
AlignedBags{Int64}(UnitRange{Int64}[1:27])

julia> ds.data.atoms.data.data.bonds.bags
AlignedBags{Int64}(UnitRange{Int64}[1:3, 4:5, 6:8, … 60:62, 63:65, 66:68])
```

Figure 2.2. This Figure features examples of bag values of different Bag nodes

2.1.3 Product nodes

Product nodes contain a set of nodes that might have different structures. They can have multiple named child nodes. This structure is similar to objects in JavaScript or dictionaries in Python.

2.2 Uses of Hierarchical multiple-instance data

Any JSON or XML document can be represented inside the Mill.jl framework.

A lot of the data transferred on the web is JSON. Moreover, the output of many software analysis tools, such as [17], is in JSON. Therefore, being able to classify it well is critical, especially in cybersecurity.

2.3 Mill.jl library

Mill.jl [18] stands for Multiple instance learning library. It is a Julia library specifically designed to work with hierarchical multiple-instance data.

Mill.jl implements the nodes mentioned in Section 2.1. These nodes can be used to build complex hierarchical multiple-instance data structures.

Furthermore, Mill.jl also implements model nodes. The Hmil models follow the structure of the Hmil data sample or data samples. For each Hmil node, Mill.jl has an according model node. These model nodes form a similar hierarchical structure as the data sample. Therefore, the process of creating a model is called `reflecting`.

The final model is a tree structure of connected submodels, where each sub-model corresponds to the according hmil node.

Mill.jl is stored on GitHub at `https://github.com/CTUAvastLab/Mill.jl`

2.4 JsonGrinder.jl library

JsonGrinder.jl is a Julia module that converts JSON data to Mill nodes. The conversion can either use default pre-set parameters, which work well for most cases or be customized for specific demands.

For Example, JsonGrinder.jl includes a default set of rules to determine whether a variable should be treated as categorical or scalar. These rules can be overridden for any user-specific reasons.

JsonGrinder.jl is open source and available on GitHub at `https://github.com/CTUAvastLab/JsonGrinder.jl`

2.5 Alternative approaches to learning from Hierarchical multiple-instance data

As was stated in [19] by Tomáš Pevný, alternative approaches for learning from hierarchical multiple-instance data are

- **Feature engineering** - This approach takes the hierarchical multiple-instance data, selects some features, and converts them to vector, which omits the hierarchical information. This method is mainly based on the experience of human researchers [19].

- **JSON/XML/YAML/other formats as an image** - This approach converts hierarchical multiple-instance data to image format, and the convolutional neural network learns to classify the images into different classes based on different patterns in the image.
- **JSON/XML/YAML/other formats as a text** - This approach strips the data of hierarchical information and uses NLP models to classify the text data.

These methods lose information by stripping the hierarchical information from the data. Furthermore, some require human expertise, therefore, they cannot be automated easily.

The Hmil approach keeps the hierarchical information and reflects the hierarchy into the model. Moreover, it can be easily automated as it does not rely on human feature engineering.

Chapter 3
ExplainMill

ExplainMill.jl is an open-source library introduced in [11]. The implementation by Tomáš Pevný, Šimon Mandík and Matěj Račinský is stored on GitHub at `https://github.com/CTUAvastLab/ExplainMill.jl`. ExplainMill's primary focus is to explain models built using Mill.jl introduced in [18]. This chapter will explain the terms `confidence gap` and `confidence gap threshold`, describe how the masking system works, and what the explanation process looks like.

3.1 Confidence gap

The confidence gap for a particular class c and data sample x is the difference in the model's final softmax layer between class c and class b, where class b is the class with the highest value in the softmax layer excluding class c. The model classifies the sample x as a class c when the confidence gap is positive. When the confidence gap is negative, the model classifies the sample as the class b. The higher the confidence gap is, the more confident the model is in the classification of x as c and vice versa.

The confidence gap function is implemented in ExplainMill.jl.

3.1.1 Confidence gap threshold

Users can define a $threshold_{relative}$ or $threshold_{absolute}$. If the user chooses to define $threshold_{relative}$, ExplainMill.jl converts $threshold_{relative}$ to $threshold_{absolute}$ in the following way

$$threshold_{absolute} = threshold_{relative} * cg$$

where cg is the confidence gap of data sample x. ExplainMill.jl then guarantees that the explanation will have a confidence gap at least $threshold_{absolute}$. Further in this work, we will reference to the $threshold_{absolute}$ as τ.

3.1.2 Confidence gap and explanation size trade-off

The higher the confidence gap we want from the explanation, the larger the explanation will tend to be, and vice versa. This is generally true, but some exemptions can be found.

3.2 Masks

ExplainMill.jl introduces the concept of masks. Mask mk can be applied to a sample ds. This is noted as $mk[ds]$. The masked sample is pruned according to the mask values. The mask can mask individual observations or inner nodes in the tree representations. The problem of finding a good explanation can be redefined as finding a good mask mk for the sample ds, such that $ds[mk]$ is a good explanation.

ExplainMill.jl has a function `create_mask_structure`, which creates a mask mk from a data sample ds. The mask mk has a similar structure to ds, but it is not made from Mill nodes but from mask nodes. Representation of a mask of a molecule data sample from Fig. 2.1 is in Fig. 3.1.

```
typename(ExplainMill.ProductMask)
  ├────────── lumo: FeatureMask
  ├────────── inda: CategoricalMask
  ├────────── logp: FeatureMask
  ├──── mutagenic: CategoricalMask
  ├────────── ind1: CategoricalMask
  └───────── atoms: BagMask
                      └── typename(ExplainMill.ProductMask)
                            ├────── element: CategoricalMask
                            ├──────── bonds: BagMask
                            │                  └── typename(ExplainMill.ProductMask)
                            │                        ├──── element: CategoricalMask
                            │                        ├── bond_type: CategoricalMask
                            │                        ├───── charge: FeatureMask
                            │                        └── atom_type: CategoricalMask
                            ├─────── charge: FeatureMask
                            └──── atom_type: CategoricalMask
```

Figure 3.1. Representation of a mask of data sample featured in Fig. 2.1. Visualization was generated with HierarchicalUtils.jl [16].

The types of mask nodes will be discussed in the following subsections.

3.2.1 Product masks

Product masks do not mask anything. However, they are necessary for the structure of the mask tree, as they contain named child masks.

For example, in Fig. 3.1, the root node is a ProductMask, and it contains six child mask nodes of types FeatureMask, CategoricalMask, or BagMask.

3.2.2 Bag masks

BagMasks can be applied to BagNodes. BagMasks mask individual observations inside the child node.

3.2.3 FeatureMask

FeatureMasks can be applied to ArrayNodes. FeatureMasks can mask rows. In other words, specific values an observation can become. However, FeatureMasks cannot mask out individual observations because observations are represented as columns in ArrayNode.

This mask is primarily used with uncategorical numerical variables.

3.2.4 CategoricalMask

CategoricalMask can be applied to only ArrayNodes. CategoricalMask can mask out columns, equal to masking out individual observations of ArrayNode. On the other hand, CategoricalMask cannot mask rows inside ArrayNodes. Therefore, it cannot mask specific values an observation may become.

This mask is primarily used with categorical variables.

3.2.5 NGramMatrixMask

NGramMatrixMask can be applied to ArrayNodes that store n-grams. This mask is designed to work with strings and can mask out individual observations which is equal to masking columns.

3.3 How ExplainMill.jl finds an explanation

To find a good explanation of why model m classified input ds as class c.We are looking for the smallest subset of the sample ds such that the model's confidence gap on the explanation is positive or is as big as the user defines. Exhaustive search is not possible in most cases since the number of subsets of a set is 2^n, where n is the number of items in the original set.

In a tree with n nodes, the number of subtrees is smaller than 2^n because some subsets are disconnected. The disconnected subsets are called `forests`. Moreover, any forest, in our case, would be reduced to the connected part, which includes the root node, in other words, to a single tree t. Other trees than the tree t in the forest would be missing their parent node and, therefore, would be pruned out.

Computing the exact amount of subtrees is difficult due to the various branching factors in each Product node. However, the number will still be quite large, and enumerating every subtree will be computationally expensive.

Therefore, ExplainMill.jl implements several heuristics to search faster. They will be discussed further in this chapter.

ExplainMill.jl separates searching for the explanation into three phases. Each of these phases is independent of each other and can be configured independently. These three phases are described in the following subsections.

3.3.1 Subset selection

In this phase, we decide which subtrees we will investigate if they are important. The possible configurations are:

- **Flat search** - consider all nodes
- **Leaf search** - consider just leaves and keep paths to the leaves
- **Level by level search** - go level by level - and consider all nodes on that given level, except the children of previously removed nodes.

3.3.2 Search Type

In this phase, we can decide what search features we will use. Possible search features are:

- **Greedy addition** - Greedy addition evaluates all possible nodes to add to the existing explanation and picks the one with the highest gain of model confidence in the class we want to explain.
- **Heuristic addition** - Heuristic addition uses some heuristics to rank the importance of the nodes and chooses the first k nodes so that the explanation has an explanation gap greater than the confidence gap threshold τ.
- **Random Removal** - Random Removal is a process that can be performed after each addition. It shuffles the existing nodes in the explanation and looks for any node that can be removed without the model's confidence dropping below threshold τ. If the removal is successful, nodes are reshuffled, and the algorithm looks for other possible removal. If none is found, then the algorithm stops.
- **Fine Tuning** - Fine Tuning adds l elements to the explanation greedily. It removes the elements with the least loss of the model's confidence in the class we want to explain until no such can be found without going below the threshold τ. If the same nodes that were added were also removed, l would be increased. l starts at one and ends at $min(5, 2|S|)$ where S is the current explanation and $|S|$ is the number of nodes in S.

3.3.3 Heuristic sub-tree ranking

This phase ranks the available nodes based on heuristic h. These are the available heuristics implemented in ExplainMill.jl:

- **Constant heuristic** - Constant heuristic provides a constant heuristic for all nodes. It is used only as a way to measure the influence of other heuristics.
- **Gradient heuristic** - Gradient heuristic, as stated in [11], reads the model's gradient concerning embedding any data sample sub-trees. The higher the absolute value of this gradient is, the more likely it is an important feature.
- **Gnn heuristic** This method was introduced in [10] and is intended for graph data. This method was briefly described in Chapter 4. Here, it is used to provide heuristics and rank the subtrees.
- **Stochastic Heuristic** This heuristic assigns each node a random value. It was developed to be benchmarked against other heuristics.
- **Banzhaf heuristic and Shapley heuristic** These heuristics are implemented in the package [20], based on the papers [21–22]. Banzhaf heuristic estimates Banzhaf values originally introduced in [23]. Whereas Shapley heuristic estimates shapley values originally introduced in [24]. Either way, the nodes are sorted according to the calculated values, and their ranking is used as a heuristic.

Chapter 4
Overview of explanation methods

There are many methods for explaining the model's decision. We will focus only on the most relevant ones to our work with malware classification.

4.1 LIME

Lime was introduced in [8] in 2016 by researchers at the University of Washington. The acronym stands for local interpretable model-agnostic explanations.

Assume that f is the complex model, and g is the simple model used as a local explanation. The model g is sometimes also referred to as a surrogate model.

4.1.1 Interpretable representation of the data

X is the set of the input data. To provide good explanations, the raw input data X needs an interpretable representation of the data X'. X' needs to be simple enough for humans to understand it. The Lime paper [8] gives these two examples. Assume that $x \in X$ and $x' \in X'$.

- If x is a tensor representation of the image, then x' might be a vector of boolean values suggesting the presence or absence of superpixels [12]. (When a superpixel is absent, it is replaced with a grey color.)
- If x is text embedding, then x' might be a vector of boolean values suggesting the presence or absence of individual words of the original text.

So intuitively, we can think of x' as a mask of the presence or absence of some interpretable components of x.

4.1.2 Sampling perturbations

We sample instances by choosing a random amount of random non-zero values from x'. Other values are set to zero. These instances are called perturbations in [8] and are labeled z'. Intuitively, z' can be considered as a mask of the interpretable components of perturbation z.

Afterward, from z', we reconstruct the sample z that would correspond to mask z'. We can generate n perturbations and put them into the set Z for the reconstructed perturbations and Z' for interpretable masks of perturbations.

We will also generate a label for all $z' \in Z'$ as $f(z)$.

4.1.3 Computing the explanation

We solve the following problem and get the explanation g, where G is a set of all possible surrogate models.

$$\operatorname*{argmin}_{g \in G}(L(f, g, \pi_x) + \Omega(g)) \tag{1}$$

- $L(f, g, \pi_x)$ is the sum of square differences between the complex model output $f(z)$ and simple model output $g(z')$ scaled by the proximity to x defined by the function π_x

$$L(f, g, \pi_x) = \sum_{z \in Z, z' \in Z'} \pi_x(z)(f(z) - g(z'))^2 \tag{2}$$

- $\pi_x(z)$ calculates the proximity of x to z. Different implementations can be used, but the following function was used in [8]:

$$\pi_x(z) = exp(-D(x, z)^2/\sigma^2) \tag{3}$$

  Where D can be L2 distance, cosine distance, or other metric, exponentiality means that the importance of perturbations far from x will be close to zero. σ affects the width of the radius of proximity.
- Ω(g) represents the complexity of the model g. Two examples were provided:
  - If g is a decision tree, then $\Omega(g)$ might be the depth of the tree.
  - If g is a linear model, then $\Omega(g)$ might be the number of non-zero weights.

A visualization of LIME can be seen in Fig. 4.1.

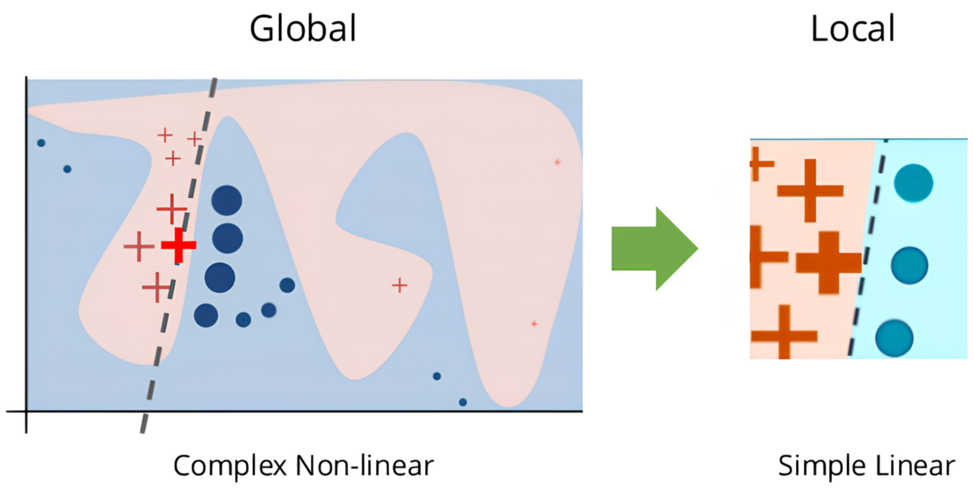

Figure 4.1. Visualization of the lime algorithm, providing a local explanation. A reproduction from [25].

4.2 SHAP

SHAP was introduced in [9] in 2017. SHAP method estimates Shapley values [24] for each feature based on how much the feature contributes to the model's output. The Shapley values are calculated by generating and evaluating many subsets of the original features. Due to the computationally expensivness of calculating Shapley values precisely, SHAP estimates them. Shapley value for feature i is equal to the feature i contribution to the model's output.

4.3 GNNExplainer

GNNExplainer was introduced in [10] in 2019. It is a method for explaining Graph neural networks. Since Hmil data follows a tree structure, it can be seen as a particular case of a graph network. It works by adjusting the mask of the graph to maximize the likelihood of a correct classification and to minimize the amount of unmasked nodes.

Chapter 5
TreeLIME

This chapter will introduce the main contribution of our work, the method TreeLIME. The name TreeLIME symbolizes that the method applies a technique similar to LIME [8] on tree data structures. Note that the terms Layered and Level by level are used interchangeably.

LIME includes the term `model-agnostic` in its acronym. Our method is model-specific. Therefore, the LIME in TreeLIME does not stand for the same acronym as the original LIME [8].

We chose this method because it seemed promising, and we wanted to evaluate it against other explanation techniques for hierarchical multiple-instance data.

5.1 TreeLIME parameters

TreeLIME supports multiple parameters to customize the explanation method. These are:

1. **n** - n determines how many perturbations will be generated.
2. **type** - *type* can be either `Flat` or `Layered`. The type determines whether the method works level by level or with the entire input x in one go.
   a) **direction** - *direction* is only relevant when the *type* is `Layered`. It defines in which order the layers should be gone through. The two possible values are `Up` and `Down`. In the `Up` direction, the furthest layer from the root is processed first, and the layer closest to the root is processed last. When the direction is set to `Down`, the process begins with the layer closest to the root and continues processing layers below.
3. **D** - D is a distribution which generates perturbation chance γ. Each perturbation generates its perturbation γ independently. D is always truncated to the interval $\langle 0, 1\rangle$, since γ is a probability.
4. **distance** - *distance* sets the distance function for TreeLIME. Two supported functions are `JsonDiff` and `Const`. If *distance* is `Const`, all perturbations' distance to the original sample equals constant 1. In other words, the distance between perturbations is ignored. If *distance* is `JsonDiff`, then a function `JsonDiff` from ExplainMill.jl is used as the distance metric. `JsonDiff` counts the differences between the JSON of the perturbation z and the JSON of the original input x.

5.2 Interpretable representation of the data

TreeLIME, like LIME, uses an interpretable representation for the input data set X. For TreeLIME, the interpretable representation of $x \in X$ is a mask m. The mask m is constructed by the ExplainMill.jl library for the sample X. This mask is a tree structure that can mask leaves and inner nodes of x.

5.3 Sampling perturbations

The process starts by generating mask m to fit the structure of x and filling all the children-masks from m with `True` values. Afterward, TreeLIME takes a set S of children-mask from the mask structure m. If *type* is `Flat`, S includes all children-masks with assignable values. That is, all children-masks that are not ProductMasks, as ProductMask only joins child masks together and does not mask anything by itself. If *type* is `Layered`, set S includes all children-masks from the mask structure m on some layer l. ProductMasks are ignored in the layers.

Each children-mask from S includes a binary list of some size. If the list has a value of `True` on index i, the corresponding node or observation in x is not masked and is included. If the value on index i is `False`, the corresponding node or observation will be masked with the value missing. During the evaluation of the Hmil models, missing values are replaced using various strategies with some default values.

We iterate over every value in every children-mask in S, and with perturbation chance γ, the initial value of `True` is overwritten with a `False` value. The resulting modified mask is named z' to correspond with the LIME notation. From any modified mask z', we can reconstruct a sample z using the original sample x.

However, editing children-masks in S might create a parentless observation that, according to the mask, should be included, but their parent node should be removed. Parentless observations are absent in the pruned sample z reconstructed from the modified mask z', where z' is a mask with parentless observations.

To be consistent with z and z', we need to remove masking patterns from z' that would correspond to creating parentless observation. This consistency is important for the TreeLIME method. It ensures that the model is not learning from misleading data. Lastly, we put all modified masks z' into set Z' and all reconstructed samples z into set Z.

5.4 Generating perturbation chance γ

Perturbation chance γ is the probability of removing nodes or observations from x. Perturbation chance γ is drawn from distribution D independently for each perturbation. All supported distributions are in Figure 5.1.

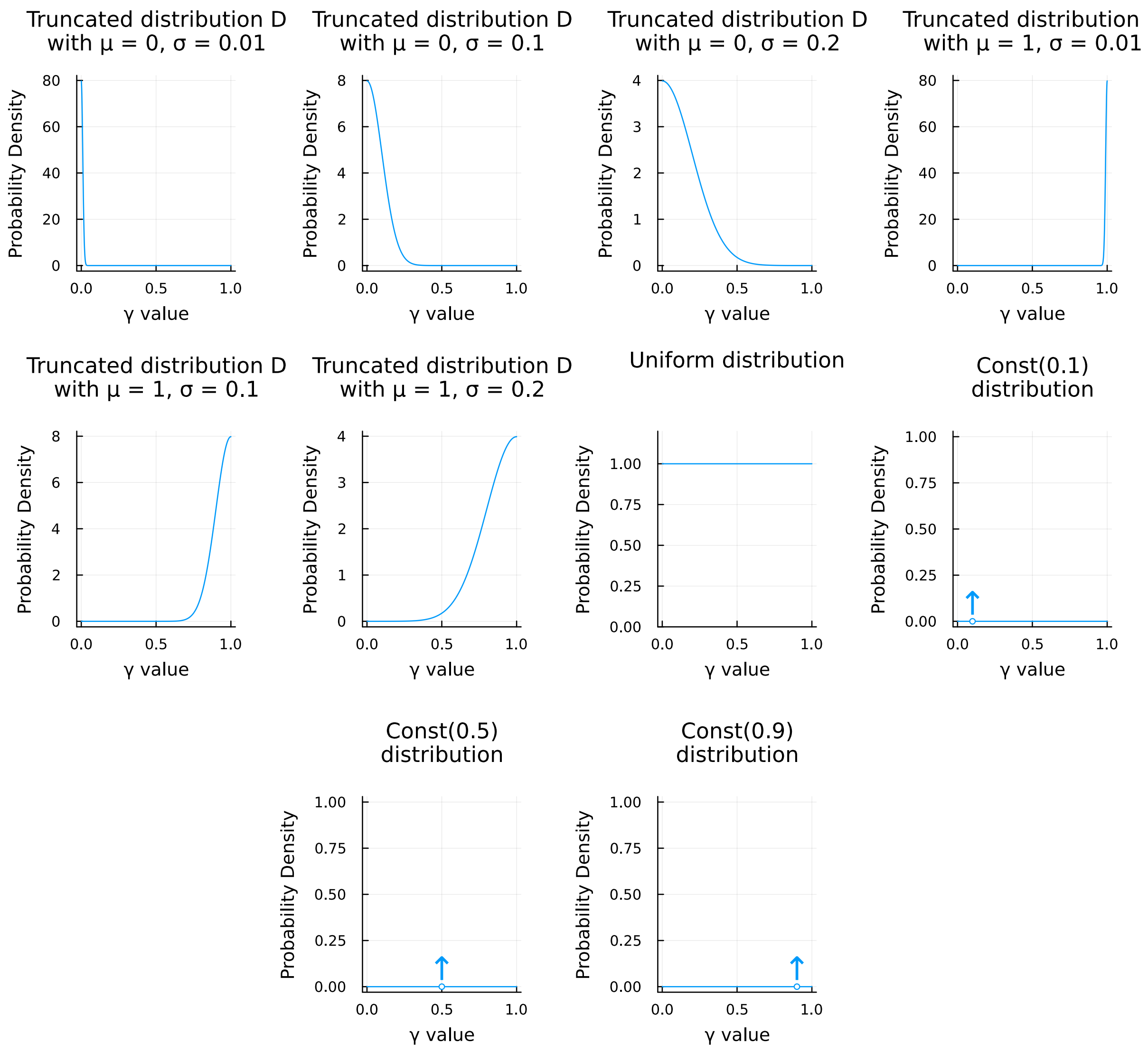

Figure 5.1. Probability density functions for different distributions D.

5.5 Computing the explanation

To compute the surrogate model g, we use logistic regression described in [26] with lasso regularization, introduced in 1996 in [27].

The surrogate model is trained using GLMNet.jl [28], a Julia wrapper for glmnet [29–30].

Input data for the logistic regression model g are Z', and the label for z' is computed as $f(z)$, where z is reconstructed from z'.

Logistic regression with lasso minimizes the following term in Eq. (1).

$$\sum_{i=1}^{u}(y_i - q_i)^2 + \lambda \sum_{j=1}^{p} |\hat{\beta}_j| \tag{1}$$

$$q_i = \frac{1}{1 + e^{-\eta_i}} \tag{2}$$

$$\eta_i = \hat{\beta}_0 + \sum_{j=1}^{p} \hat{\beta}_j x_{ij} \tag{3}$$

The equations (2) and (3) are a reproduction from Jiří Kléma's lecture called Generalized Linear Models, slide number 6 [31]. In the equations, u is the number of data samples, and p is the length of individual data samples.

As we can see, lasso minimizes the residual sum of squares plus a penalty for the sum of absolute values of coefficients. The parameter λ affects how strong the penalty for the size of the coefficients will be.

Because the penalty in lasso is the absolute value, it works very well as a feature selection tool because it tends to assign the coefficient value of zero. This phenomenon was explained, for example, in [32].

TreeLIME tries all possible lambda values in steps of `0.0005` between zero and the lambda value, which sets zero to the coefficients β_i. For each lambda, we take the coefficients $\beta_1, ...\beta_p$ and adjust the children masks in S according to them. If β_i is greater than zero, then the corresponding value in the corresponding children-mask from S will be set to one, otherwise to zero.

In this step, we interpret the surrogate model in the following way: If coefficient β_i is greater than zero, we want to put the predictor with index i into the explanation. If coefficient β_i is zero, then we do not want to put the predictor with index i into explanation, as it is not essential according to the model. Finally, suppose coefficient β_i is negative. In that case, we do not want to put the predictor with index i into the explanation, as it is important according to the model. However, its presence indicates that the sample is less likely to be the original class. The predictors with negative β_i would be useful for building a counterfactual sample, which is not the focus of this work.

Afterward, TreeLIME reconstructs the mask $m_{\lambda i}$ from all children-masks in S. Subsequently, TreeLIME applies the mask $m_{\lambda i}$ to the sample x, runs the pruned sample through the model, and measures the confidence gap.

After finishing this procedure for every lambda, we have a list of lambdas, a list of coefficients beta, and a list of confidence gaps.

Next, TreeLIME selects the highest value of λ, which has a corresponding confidence gap that satisfies the requested relative threshold.

If *type* is `Flat`, TreeLIME is finished. Otherwise, if the *type* is `Layered`, the method increases or decreases l, depending on the *direction*, and repeats this process for a new layer until no layer is left.

5.6 TreeLIME visualization

TreeLIME optimization is visualized in Figures 5.2, 5.3 and 5.4. Figure 5.2 visualizes the process when *type* is set to `Flat`. Figures 5.3 and 5.4 visualize layered optimization in directions `Up` and `Down`, respectively.

Figure 5.2 describes the optimization process of TreeLIME in Flat mode, with 200 perturbations and a relative tolerance of 50% of some example sample x. The process begins at the top left of both plots. Lambda is set to zero, and nothing has been pruned from the original sample. The dotted gray line marks the confidence gap of the original sample, and the original sample x is marked with a black dot. The method starts increasing λ with steps of `0.0005` until the λ is high enough that the whole sample is pruned and contains no leaves. In each step, the method takes the coefficients $\beta_{1,\ldots,p}$ and updates the children-masks according to them. If β_i is zero or negative, the algorithm prunes the according node or observation in the according children mask. At the same time, TreeLIME calculates the confidence gap of the original sample pruned by the according mask in each step. The algorithm chooses the mask with a confidence gap above the confidence gap threshold, which pruned the original sample mostly. A gray dashed line in both plots marks the confidence gap threshold. The λ and explanation size corresponding to the resulting mask are marked with a red dot with an orange stroke. Note that the zero confidence gap is marked with a simple gray line, and when any point is below this line, the model will classify the explanation as a different class than the original sample. The purple line changes color to gray when the confidence gap is below the confidence gap threshold to imply that it is an invalid explanation.

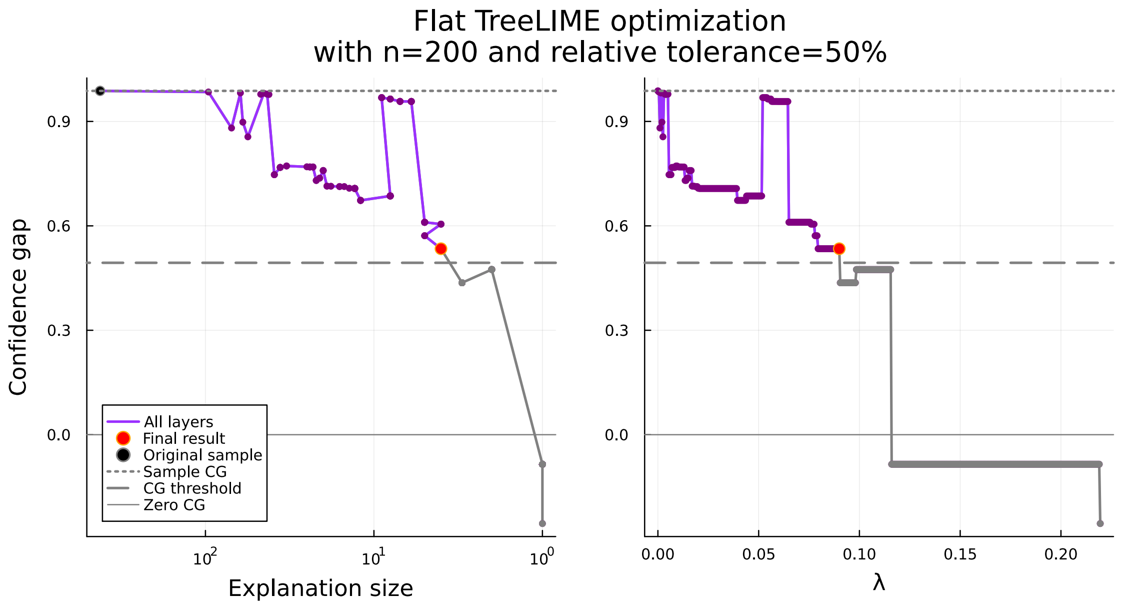

Figure 5.2. This figure describes the optimization process of TreeLIME in Flat mode, with 200 perturbations and a relative tolerance of 50%.

Figure 5.3 describes the `Layered` optimization of a different input sample x in the direction `Up`. The process begins at the lowest layer and moves upwards. This sample x is from the CAPEv2 dataset [17]. It has three layers. Layers 1 and 2 have roughly similar sizes, around 10^3 maskable values, and Layer 3 is much smaller. It has only around 10^1 maskable values. The process begins with Layer 3 and optimizes it in the same way as was described in Figure 5.2. However, the result of this optimization is not marked with the red dot with the orange stroke `Final result` but with a blue dot with the purple stroke `Partial result`. Some observations on this layer were pruned, and some were kept. Without making any further changes to Layer 3, the method starts optimizing Layer 2 similarly. Afterwards, the Layer 3. On each layer, TreeLIME starts with λ set to zero and increases it in the same step of `0.0005` until all values on that layer are pruned.

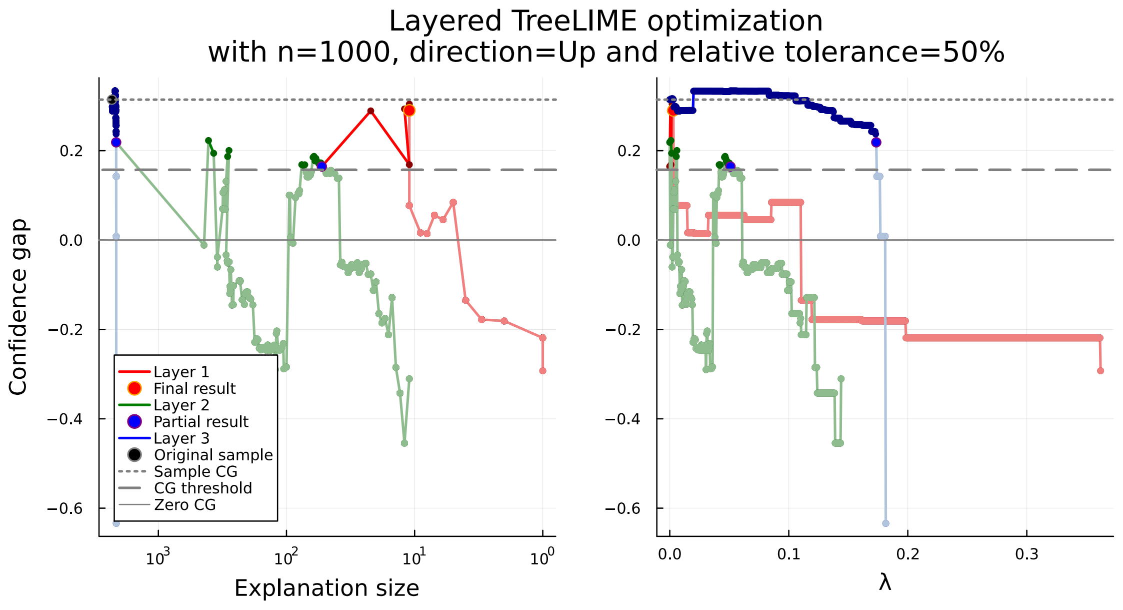

Figure 5.3. This figure describes the optimization process of TreeLIME in Layered-Up mode, with 200 perturbations and a relative tolerance of 50%.

Figure 5.4 describes the `Layered` optimization of a different input sample x in the direction `Down`. In the `down` direction, the algorithm begins with Layer 1, closest to the root, and then continues with layers below. Optimizing Layer 2 produced only explanations with a confidence gap below the confidence gap threshold in this example. That meant nothing was pruned on Layer 2, and the optimization process effectively skipped Layer 2 and continued with Layer 1. However, how the process tried to optimize Layer 2 is still visualized in this figure with a light green color.

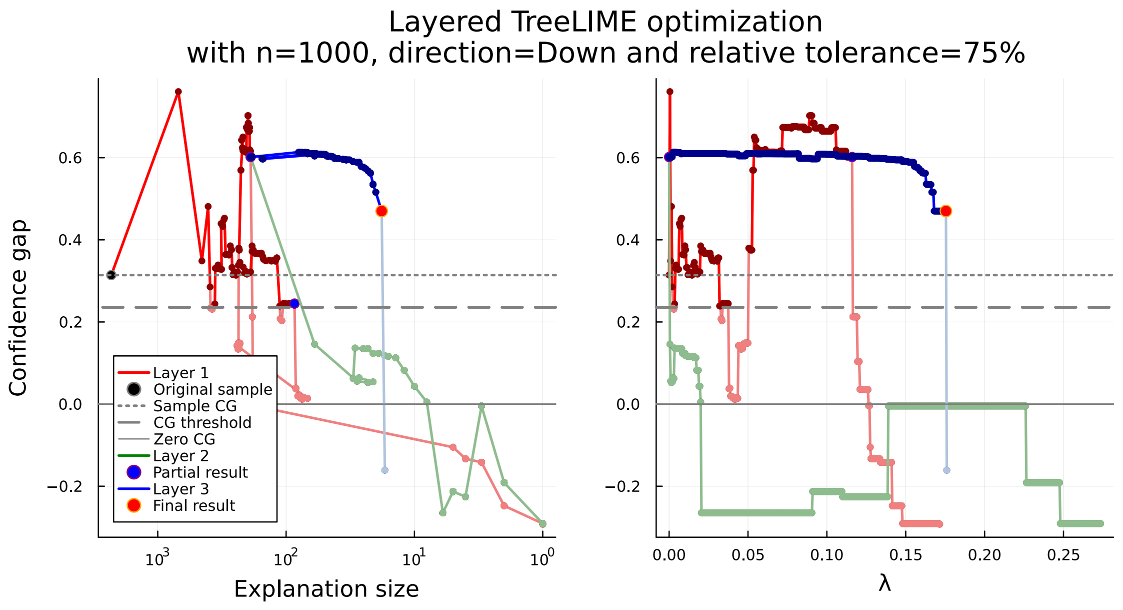

Figure 5.4. This figure describes the optimization process of TreeLIME in Layered-Down mode, with 200 perturbations and a relative tolerance of 50%.

5.7 TreeLIME implementation

TreeLIME was implemented in Julia [33] to be compatible with the ExplainMill.jl [11], Mill.jl [18], Flux.jl [34], and JsonGrinder.jl [18]. As mentioned, the surrogate model is trained with [28]. The distributions D were implemented in Distributions.jl [35]

Chapter 6
Experiments

In this chapter, we will analyze how each parameter affects the performance of TreeLIME. Finally, we will compare the performance of TreeLIME to other explanation techniques available in ExplainMill.jl.

6.1 CAPEv2 dataset

This chapter's experiments explained classifications of the Avast-CTU Public CAPEv2 dataset [1]. This dataset includes 48,976 scans of malicious programs from the software CAPEv2 [17]. The malicious programs are classified into ten different families. These are:

- Adload - adware
- Emotet - banking and other information-stealing Trojan
- HarHar - drive encryption ransomware
- Lokibot - information-stealing Trojan
- njRAT - remote access Trojan
- Qakbot - banking Trojan
- Swisyn - remote access Trojan
- Ursnif - banking Trojan
- Zeus - banking Trojan

6.2 Experiment environment

All experiments run as Slurm [36] jobs on the Czech Technical University Research Center for Informatics cluster on the upgraded AMD nodes from 2021 equipped with AMD EPYC 7543. GPU acceleration was not used.

6.3 Information about the experiments

We produced a classifier f using the Mill.jl library and trained it to classify malware scans into one of the ten malware families. The classifier f achieved a classification of 86.1% percent on test data. The implementation of the training process was based on the example script in [17], where the hyperparameter search was already conducted, which resulted in using 32 neurons in the penultimate layer. Given more time for training, higher accuracy could be achieved, but because the main focus of this work was improving TreeLIME to match the current state-of-the-art approaches.

Furthermore, the model was trained with a drop-out ratio of 1 original sample to 9 subsamples with the same label. Subsamples were created from the original samples by applying random masks. Drop-out makes the model more used to the small samples. Otherwise, the model never sees the small subsamples in the training process, and therefore, the model could behave unexpectedly when classifying the small subsamples.

The Gnn and Grad heuristics were not tested on this dataset because of implementation issues that were not directly connected to this work. Unfortunately, we did not have the capacity to resolve those issues. However, in experiments on smaller datasets, `mutagenesis`, `device id`, and `hepatitis` from [11], Grad and Gnn heuristics systematically showed worse performance than Shapley and Banzhaf heuristics.

6.4 Experiments evaluation criteria

Our evaluation of the criteria is the smallness of the explanation and the amount of time the method took to create the explanation. Smaller explanations are easier to understand and give more dense information about the model. Therefore, the explanations are more valuable. A shorter time to produce the explanation makes the method more practical.

In [37] are the following metrics to evaluate how good the explanation method is:

- **Stability** - Stability is a metric that determines how similar the explanations are when explaining the same sample on similar models with almost identical performance. Small changes in the training process, such as a random forest model with 100 or 101 trees, should not significantly affect the explanation.
- **Robustness** - Robustness is a metric that determines how sensitive a model is to minor changes to the input data sample x. Remarkably, small changes to x should not lead to significant differences in explanations.
- **Effectiveness** - The first two metrics could be fulfilled by an explanation method that always returns the same explanation to every sample x and every model f. Therefore, the article [37] introduced the metric effectiveness. Effectiveness determines how important the explanation is to the model's f classification. The model's decision should change if the explanation is removed from x.

All the mentioned metrics cannot be evaluated for a single explanation but for an explanation technique that performs many different explanations.

These metrics were not implemented in this work because implementing TreeLIME was more challenging than we initially anticipated, and we had to overcome many problems. As a result, we would need more time to evaluate all these metrics. Nonetheless, explanation size and time are essential in the real-world applicability of this method.

6.5 Evaluation of distance

The method LIME [8] uses the distance between the reconstructed perturbation and the original sample as the weight of observation when training the surrogate model. We tried to implement something similar by using JsonDiff as a metric function. JsonDiff calculates d, which is the number of differences (the number of missing elements in JSON a which are present in JSON b plus the number of missing elements in JSON b which are present in JSON a) in the original sample and the reconstructed perturbation. The d is inverted and used as a weight of the perturbation mask inside the surrogate model.

As shown in Figure 6.1 and 6.2, the use of JsonDiff had a negligible impact on performance. In Figure 6.3, we can see that computing JsonDiff makes TreeLIME run longer. We concluded that JsonDiff provides an equal amount of misleading information (the difference between `weight:67` and `weight:68` is the same as between `weight:67` and `weight: No data`) as helpful information, therefore as calculating `JsonDiff` takes time, it should be omitted.

We were unaware of any other metric that could be used for the distance from perturbation to the original sample.

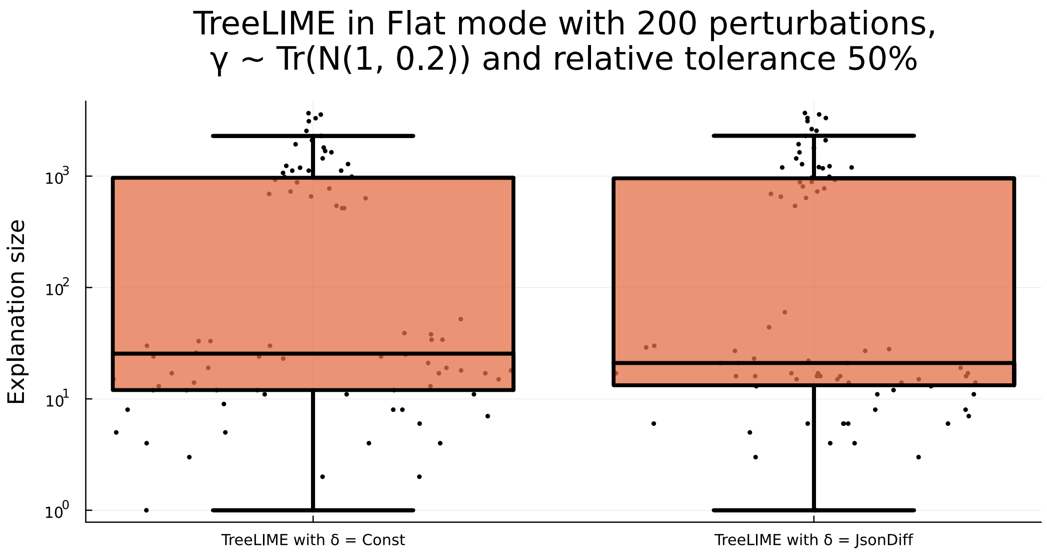

Figure 6.1. This Figure compares a Flat TreeLIME with a constant distance function and a Flat TreeLIME with a JsonDiff distance function.

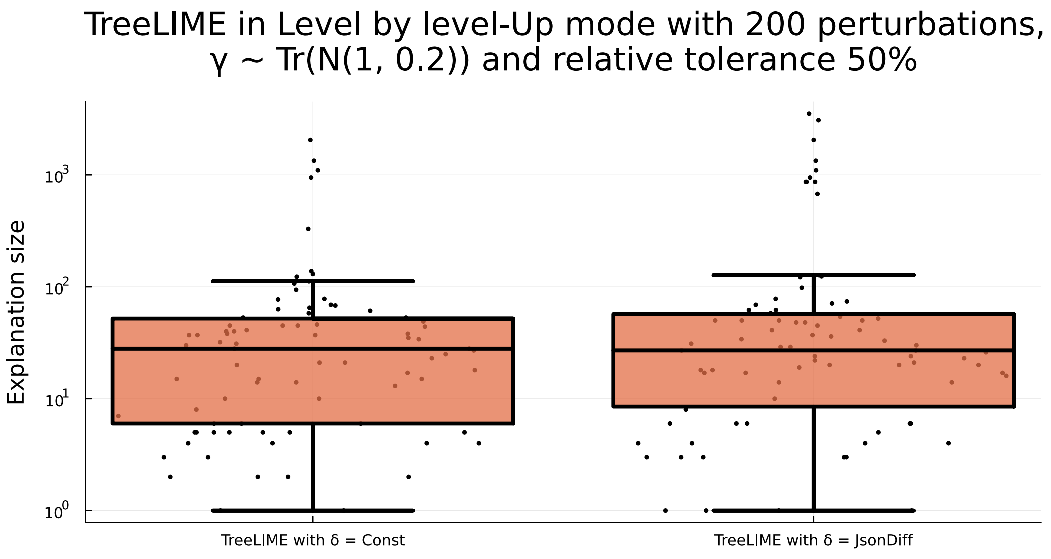

Figure 6.2. This Figure compares a Layered TreeLIME using a constant distance function and JsonDiff as a distance function.

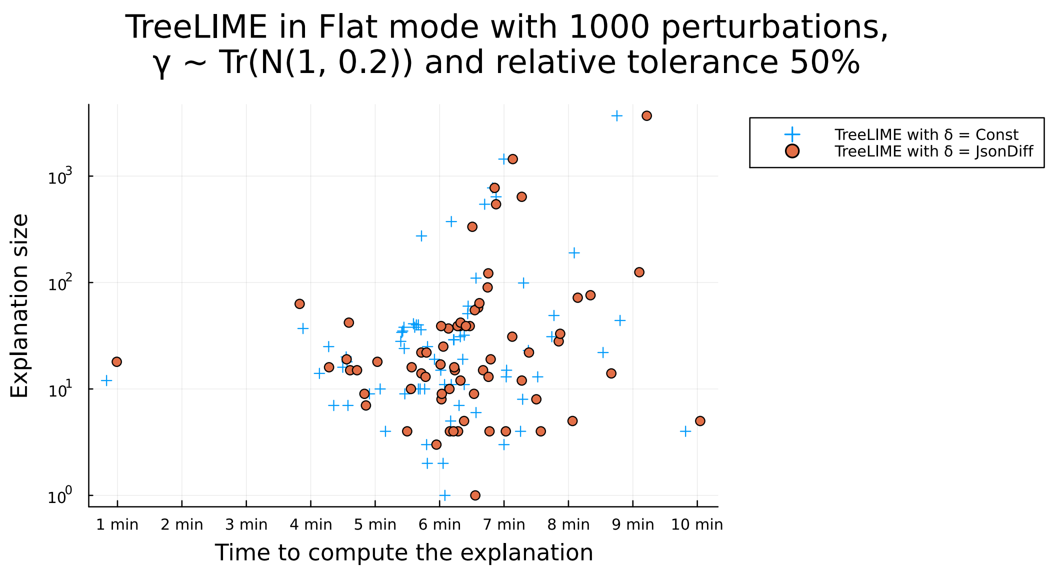

Figure 6.3. This Figure compares a Flat TreeLIME with 1000 perturbations using a constant distance function and JsonDiff as a distance function regarding time.

6.6 Evaluation of the TreeLIME mode

TreeLIME supports three different modes. They are described throughout Chapter 5.

TreeLIME modes are:

- Flat
- Level by level - Up (Sometimes referred to as Layered - Up)
- Level by level - Down (Sometimes referred to as Layered - Down)

The Figures 6.4, 6.5 and 6.6 present interesting results. Overall, Flat TreeLIME has the weakest performance. The TreeLIME mode Level by level - Down performs usually the best. Moreover, TreeLIME in mode Level by level - Up performs slightly usually worse and sometimes slightly better than Level by level - Down, but still a lot better than the Flat mode.

The reasons for the worse performance of Flat TreeLIME are described in 7.1. The Down mode could perform better than the Up mode because the upper layers prune more than the lower layers, and therefore, it is efficient to prune more first and then work with already pruned layers.

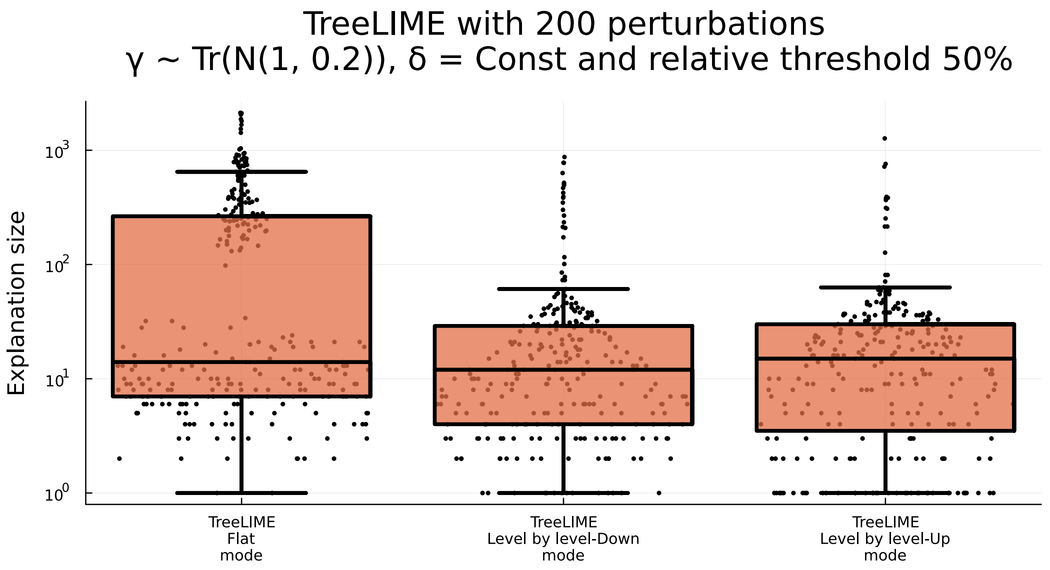

Figure 6.4. This Figure compares a Flat TreeLIME, Level by level - Down TreeLIME, and Level by level - Up TreeLIME with 200 perturbations and a relative threshold of 50%.

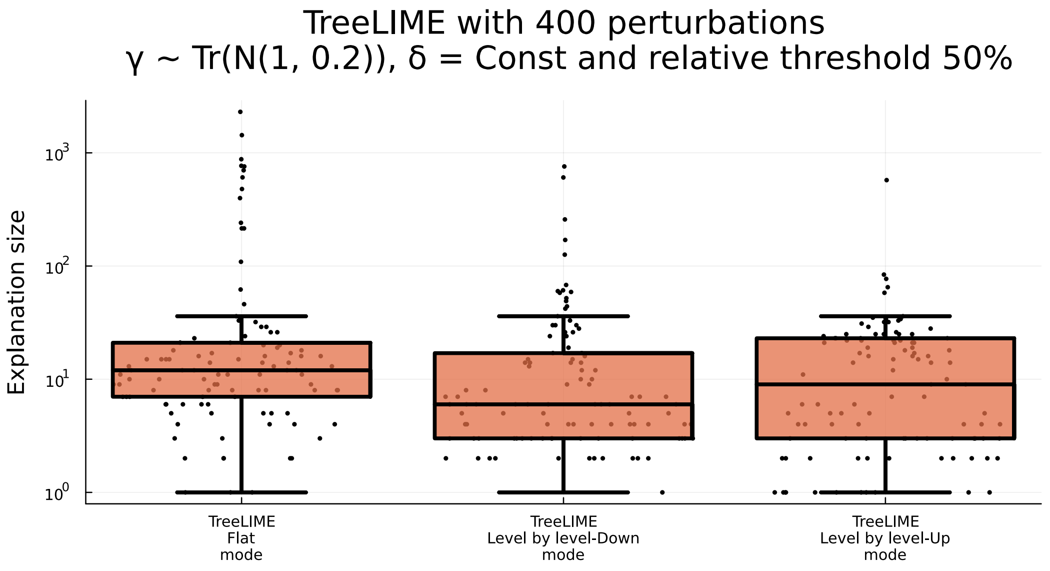

Figure 6.5. This Figure compares a Flat TreeLIME, Level by level - Down TreeLIME, and Level by level - Up TreeLIME with 400 perturbations and a relative threshold of 50%.

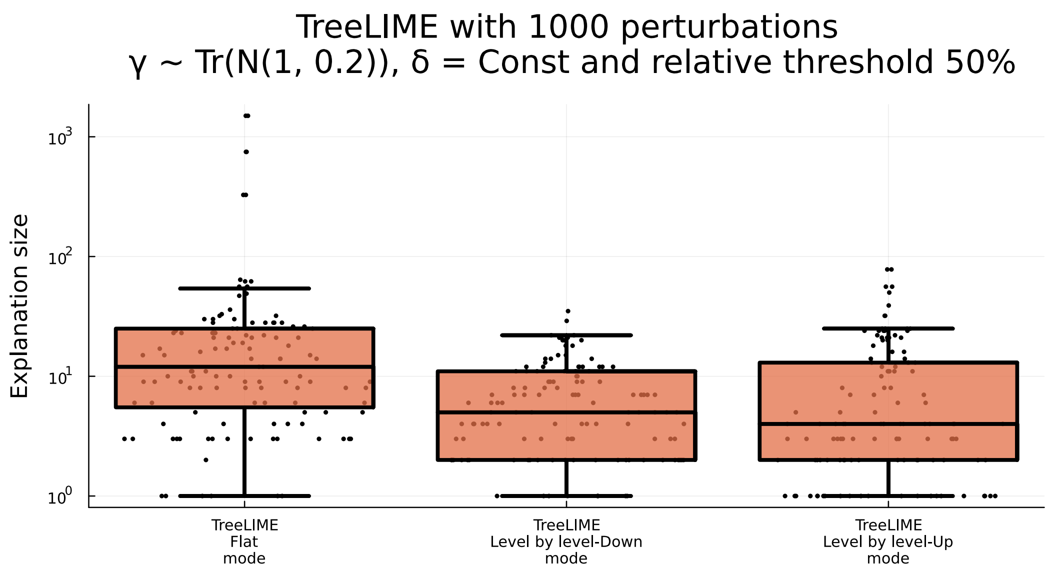

Figure 6.6. This Figure compares a Flat TreeLIME, Level by level - Down TreeLIME, and Level by level - Up TreeLIME with 1000 perturbations and a relative threshold of 50%.

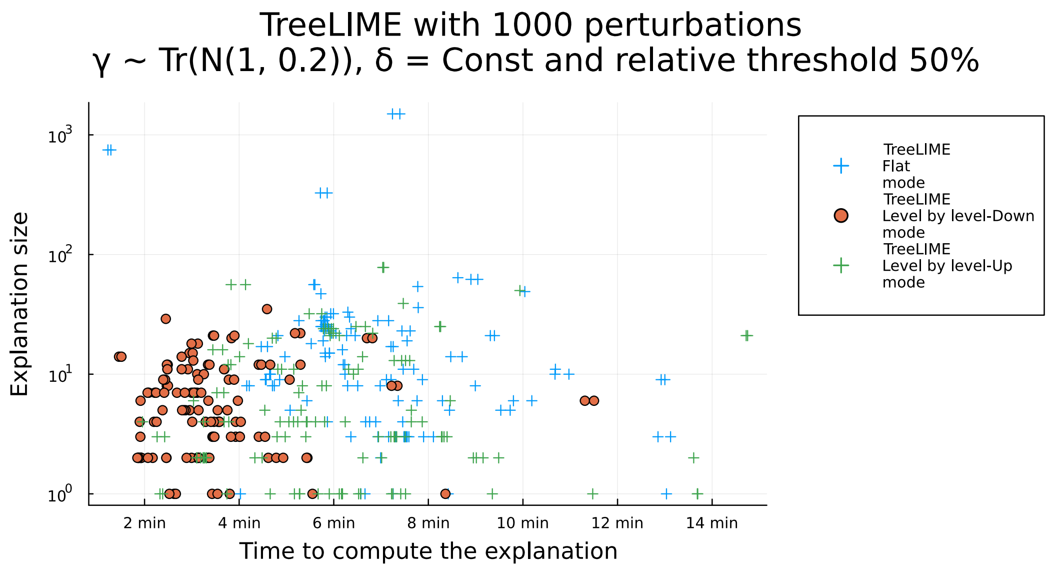

Figure 6.7. This Figure compares a Flat TreeLIME, Level by level - Down TreeLIME, and Level by level - Up TreeLIME with 1000 perturbations and a relative threshold of 50% regarding time.

Figure 6.7 shows that the Level by level - Down TreeLIME is the fastest method. This could also be because it prunes at the beginning of the process. Therefore, it prunes the most overall.

6.7 Evaluation of the Distribution D

Perturbation chance γ is drawn from the distribution D for each perturbation independently. It determines how likely a node or observation in sample x will be pruned.

Perturbation chance γ affects the size of perturbations. If γ is close to zero, the perturbations will be similar to the original sample and only miss a few nodes or observations. On the other hand, if γ is close to one, most of the sample will be pruned, and the perturbation will be more similar to the empty sample with few extra leaves or nodes.

All tested distributions are visualized in Figure 5.1. Results from the experiments focused on distribution D are in Figure 6.8 for the Flat TreeLIME, in Figure 6.9 for Layered TreeLIME in Up mode, and in Figure 6.10 for Layered TreeLIME in Down mode.

Neither Flat nor Layered TreeLIME performs well when D generates values close to zero. The methods might not even create a perturbation that is classified differently in these cases. Therefore, the surrogate model does not have access to any helpful information. However, even if they generate perturbations that are classified differently, the information in that perturbation for the surrogate model is that when these few nodes or observations are removed, the label changes, so, therefore, they should be excluded from the explanations. However, that leaves the explanation to be very large. This sampling would be more suitable for generating counterfactual examples mentioned in section 1.9, where we look for the slightest change to the original sample so that the model classifies it differently. When looking for an explanation, it is more suitable to use distribution leaning towards one because those perturbations carry more `prune` than `keep` information. Whereas perturbations generated with γ closer to one carry more `keep` than `prune` information.

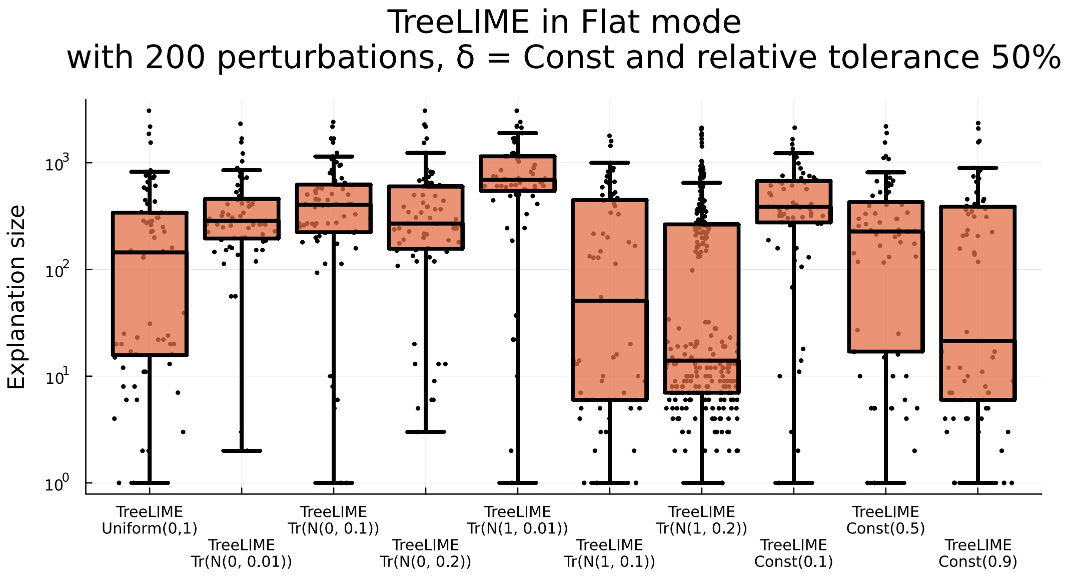

Figure 6.8. This Figure compares a Flat TreeLIME method with 200 perturbations, relative threshold 50%, and different distributions D.

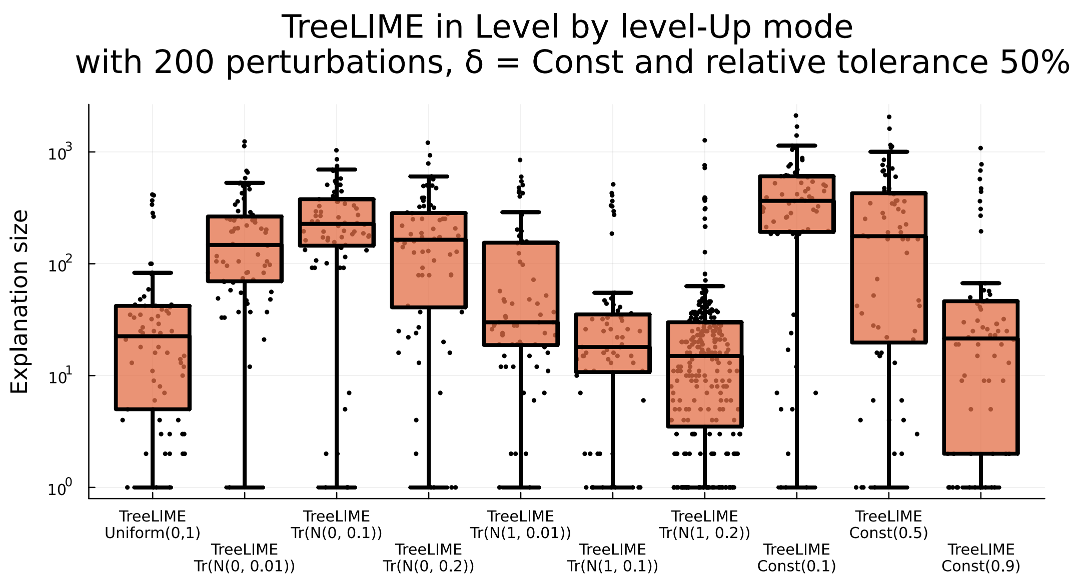

Figure 6.9. This Figure compares a Layered-Up TreeLIME method with 200 perturbations, relative threshold 50%, and different distributions D.

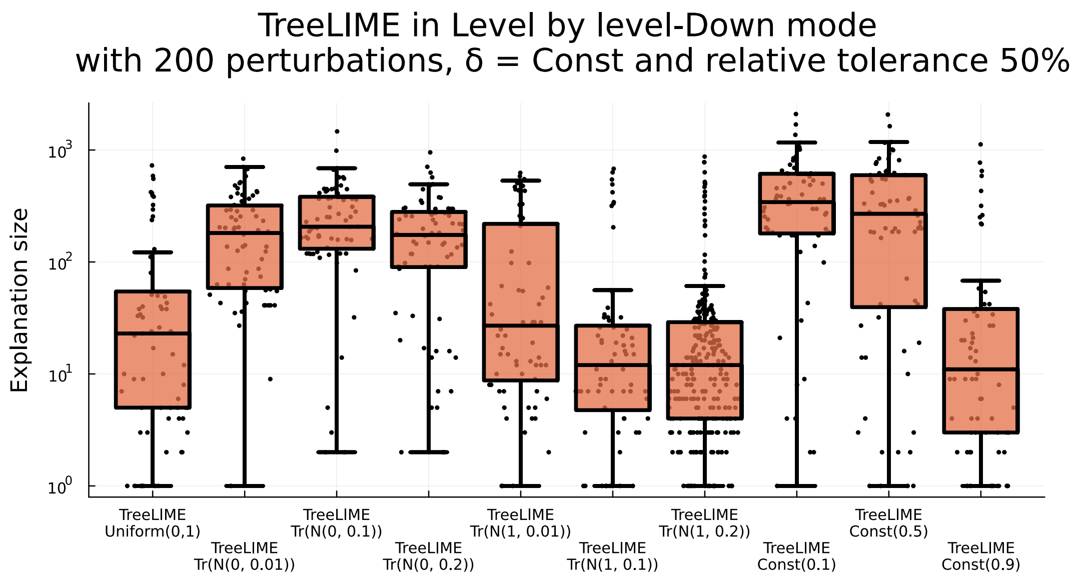

Figure 6.10. This Figure compares a Layered-Down TreeLIME method with 200 perturbations, relative threshold 50%, and different distributions D.

6.8 Evaluation of the number of perturbations n

The number n defines how many perturbations are going to be generated. Generating an explanation with 200 perturbations took one to two minutes, and 1000 perturbations took five to thirteen minutes on the CAPEv2 dataset, as shown in Figure 6.14. As these times seem manageable, we evaluated n equal to 200, 400, and 1000.

Figures 6.11, 6.12, and 6.13 showcase how the number of perturbations affects different modes of TreeLIME. Flat mode performed much better with 400 perturbations over 200, but the performance stayed generally similar with 1000 perturbations. On the other hand, in the layered modes, the explanation size decreased overall with increasing n as was expected.

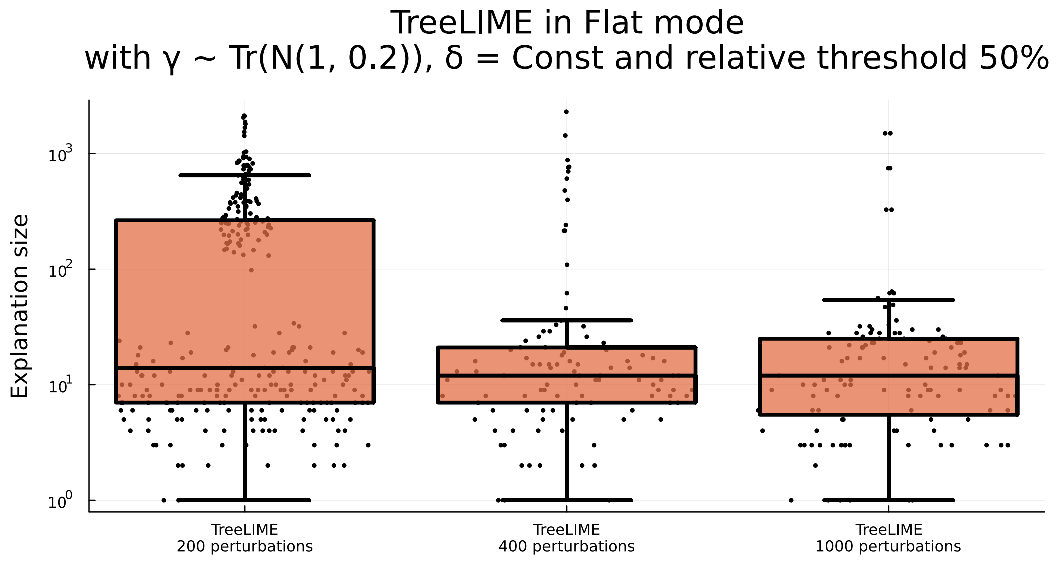

Figure 6.11. This Figure compares a Flat TreeLIME method with different numbers of perturbations regarding explanation size. Relative threshold is 50%, and D is $Tr(N(1, 0.2))$.

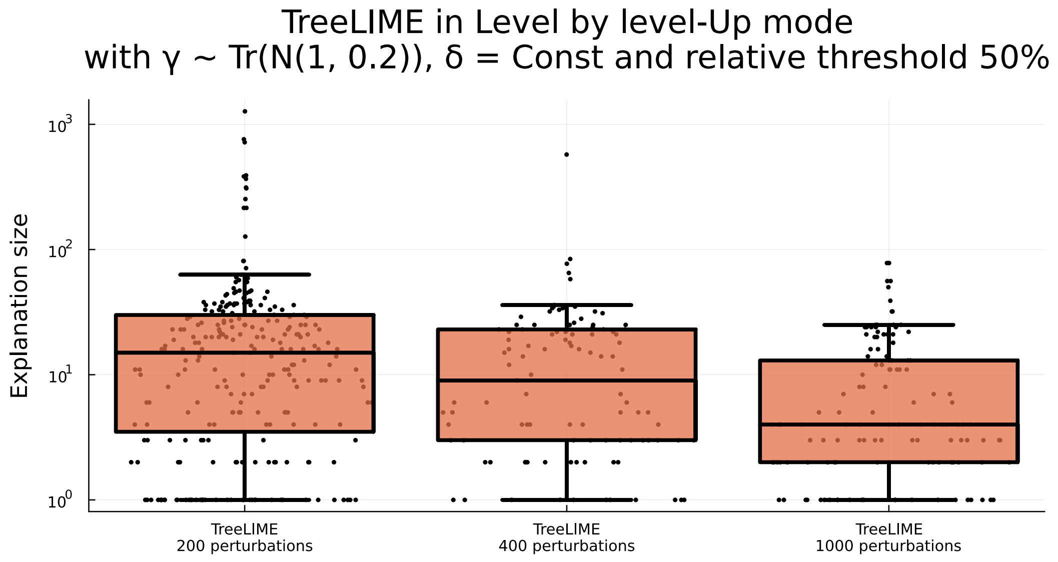

Figure 6.12. This Figure compares a Layered-Up TreeLIME method with different numbers of perturbations regarding explanation size. Relative threshold is 50%, and D is $Tr(N(1, 0.2))$.

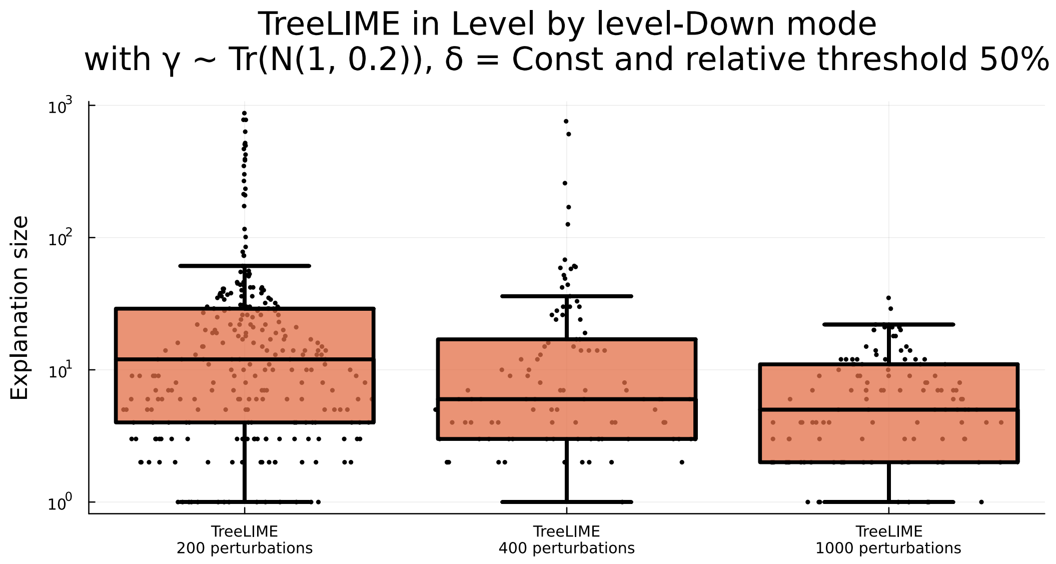

Figure 6.13. This Figure compares a Layered-Down TreeLIME method with different numbers of perturbations regarding explanation size. Relative threshold is 50%, and D is $Tr(N(1, 0.2))$.

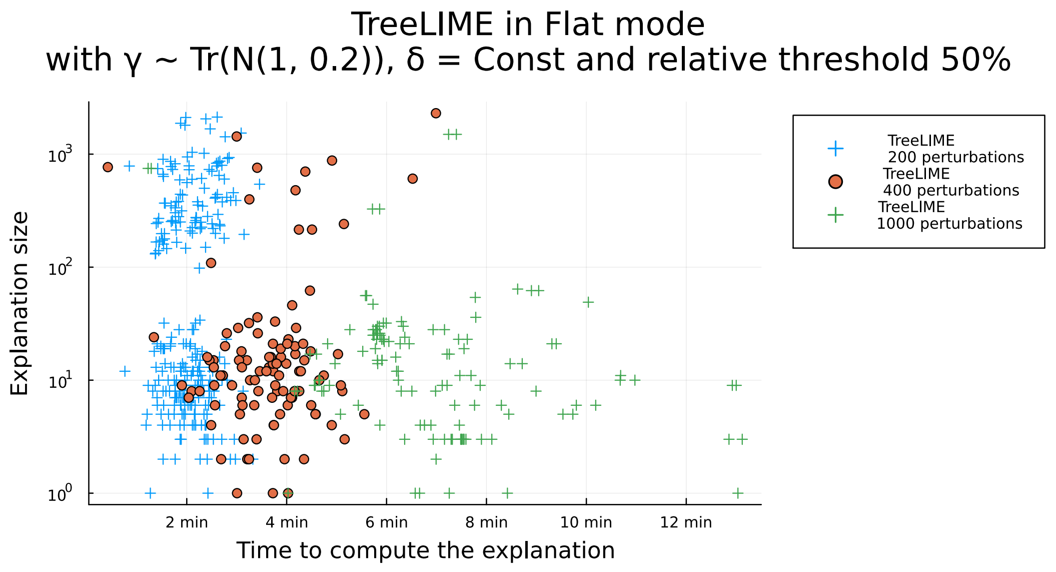

Figure 6.14. This Figure compares a Flat TreeLIME method with different numbers of perturbations regarding time. Relative threshold is 50%, and D is $Tr(N(1, 0.2))$.

6.9 Evaluation of the required relative tolerance

Required relative tolerance limits the lowest confidence gap the explanation can have. Figures 6.15, 6.16, and 6.17 show how the TreeLIME, Bazhaf heuristic and Shapley heuristic explanation methods are affected by the relative tolerance.

To conclude the results from Figures 6.15, 6.16, and 6.17. All TreeLIME methods with high relative tolerance perform significantly worse than Banzhaf or Shapley heuristics with high relative tolerance.

The Layered TreeLIME with high relative tolerance performs almost as poorly as Flat TreeLIME with high relative tolerance. The reasons why Layered TreLIME is sensitive to high relative tolerance are discussed in section 7.3.

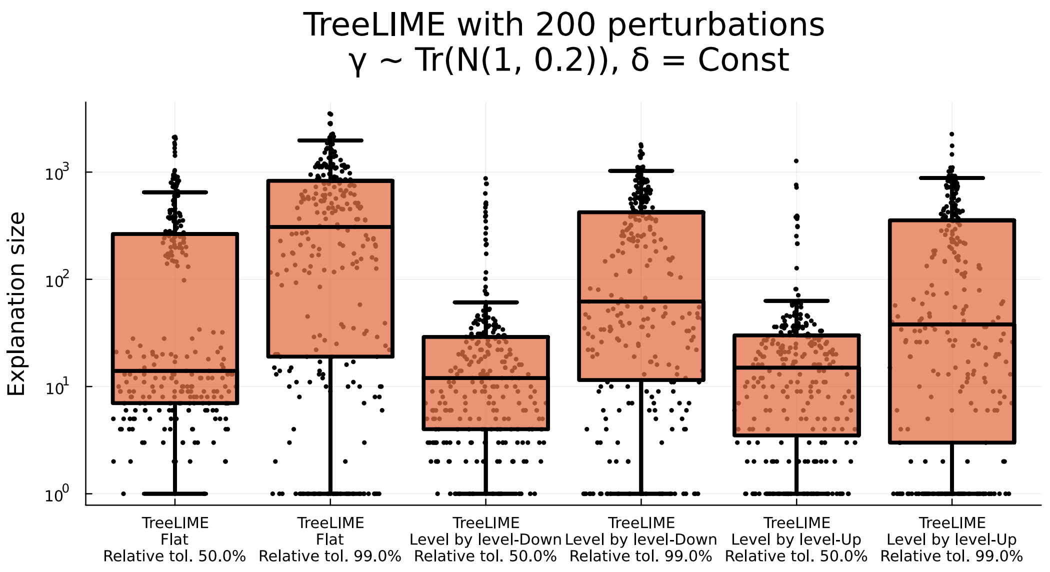

Figure 6.15. This Figure compares how relative tolerance affects TreeLIME with 200 perturbations.

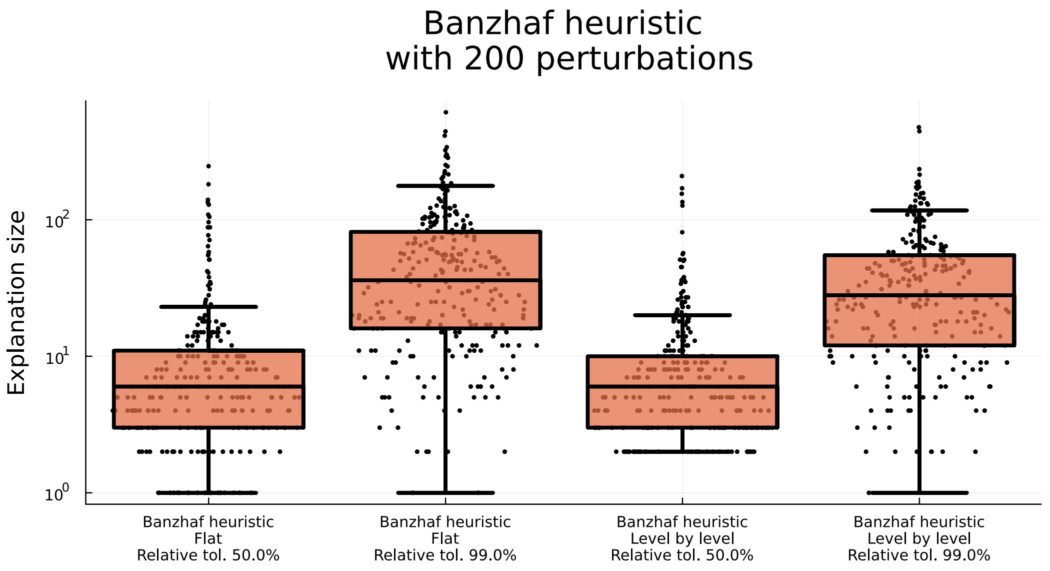

Figure 6.16. This Figure compares how relative tolerance affects Banzhaf heuristic with 200 perturbations.

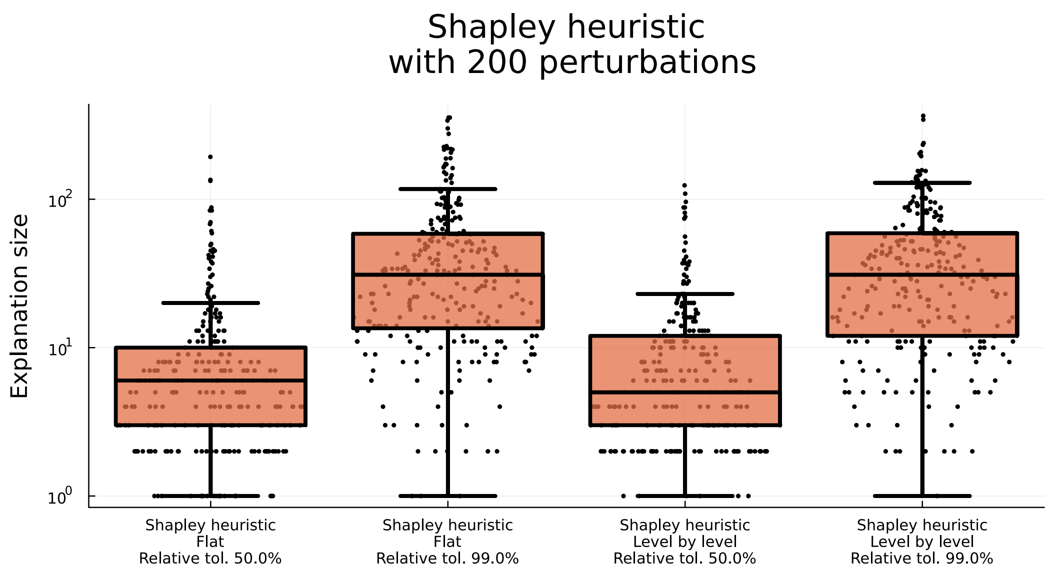

Figure 6.17. This Figure compares how relative tolerance affects the Shapley heuristic with 200 perturbations.

6.10 Comparison of the best TreeLIME method to other methods.

In this section, we will compare the best versions of TreeLIME against other methods already implemented in ExplainMill.jl.

Figures 6.18, 6.19 and 6.20 showcase Layered TreeLIME with different numbers of perturbations, and Figures 6.21, 6.22 and 6.23 are featuring Flat TreeLIME with varying numbers of perturbations.

Although TreeLIME is comparable to the current state-of-the-art methods, it is noticeably worse. The reasons TreeLIME did not perform at the same level as Shapley or Banzhaf are discussed in Chapter 7.

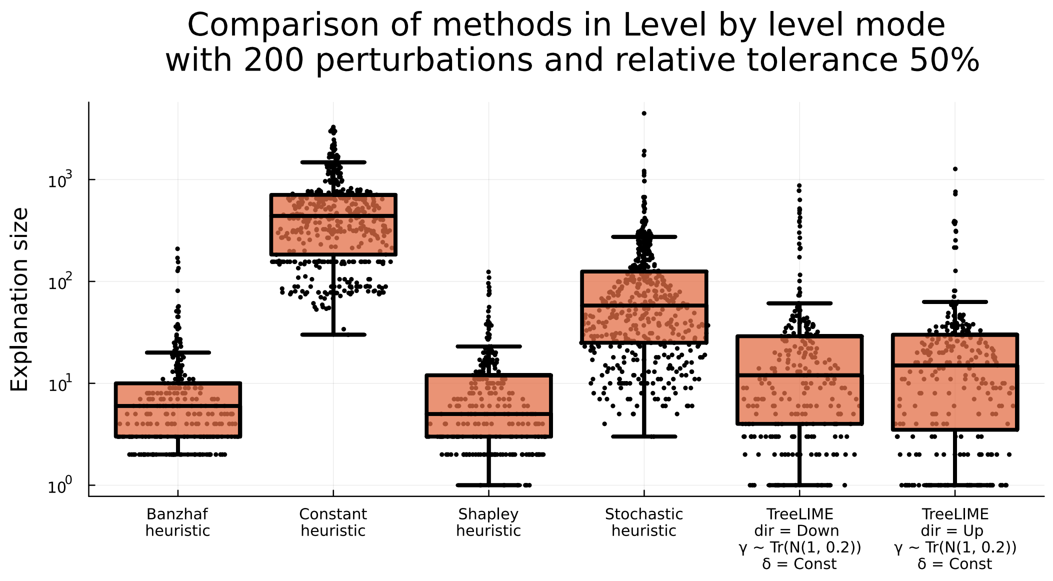

Figure 6.18. This figure compares ExplainMill.jl methods in Level by level mode with 200 perturbations against Layered TreeLIME with 200 perturbations.

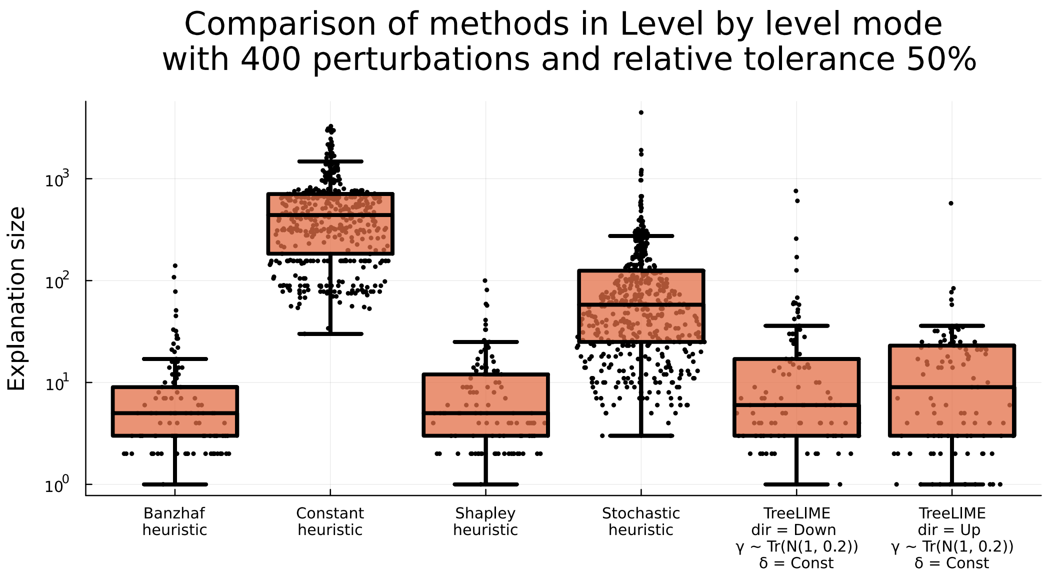

Figure 6.19. This figure compares ExplainMill.jl methods in Level by level mode with 400 perturbations against Layered TreeLIME with 400 perturbations.

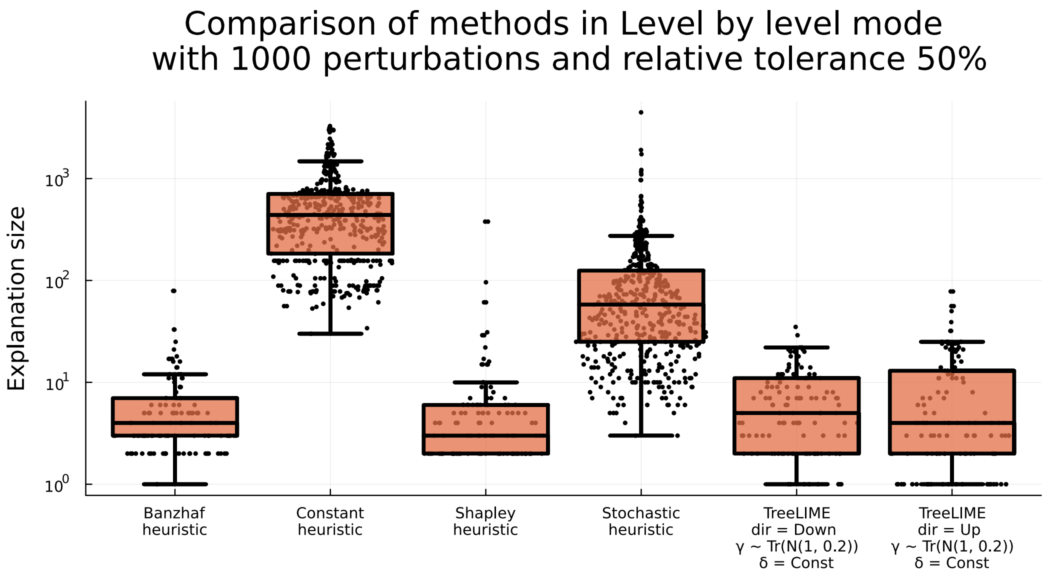

Figure 6.20. This figure compares ExplainMill.jl methods in Level by level mode with 1000 perturbations against Layered TreeLIME with 1000 perturbations.

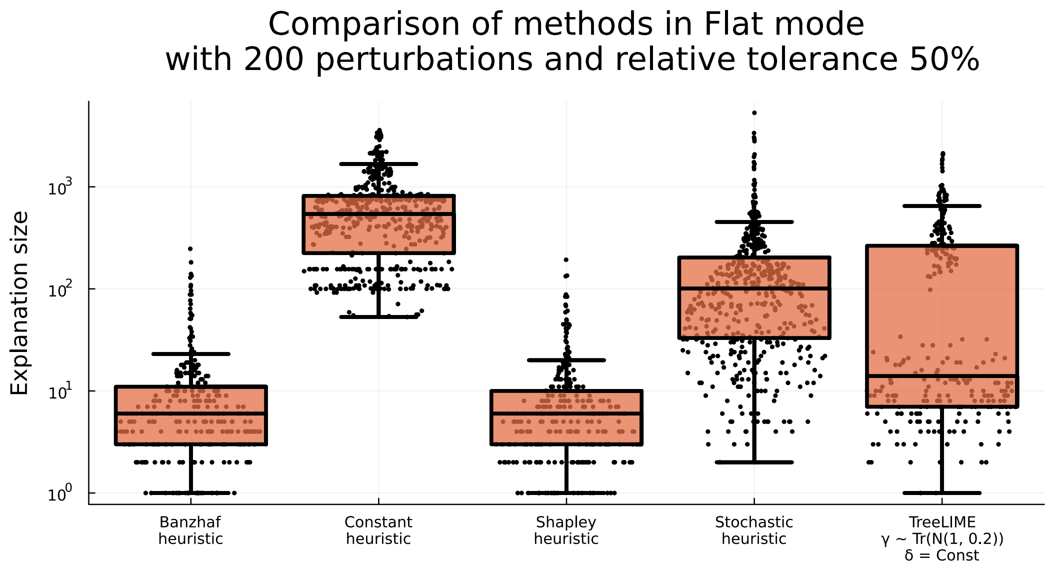

Figure 6.21. This figure compares ExplainMill.jl methods in Flat mode with 200 perturbations against Flat TreeLIME with 200 perturbations.

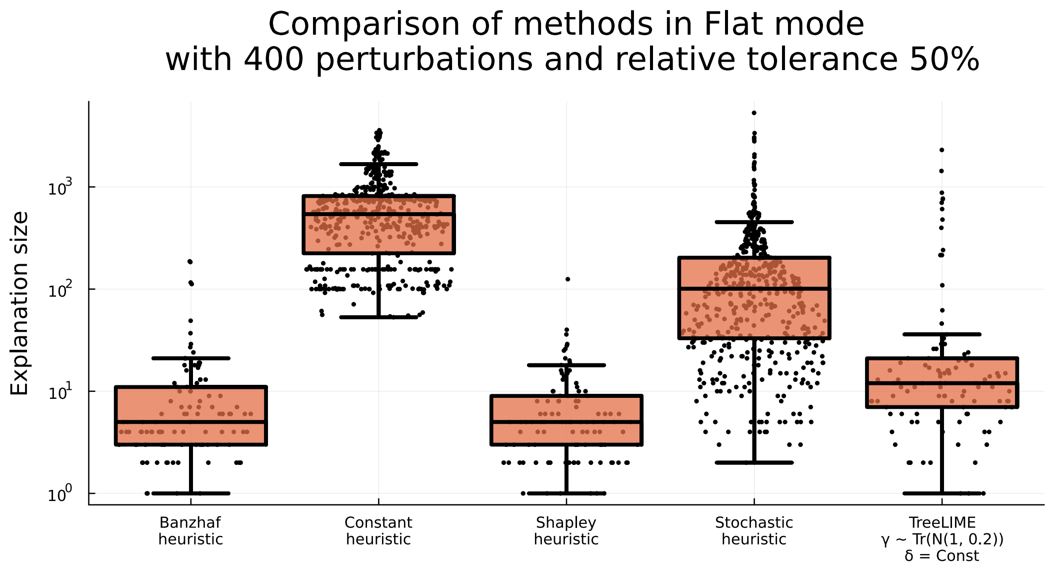

Figure 6.22. This figure compares ExplainMill.jl methods in Flat mode with 400 perturbations against Flat TreeLIME with 400 perturbations.

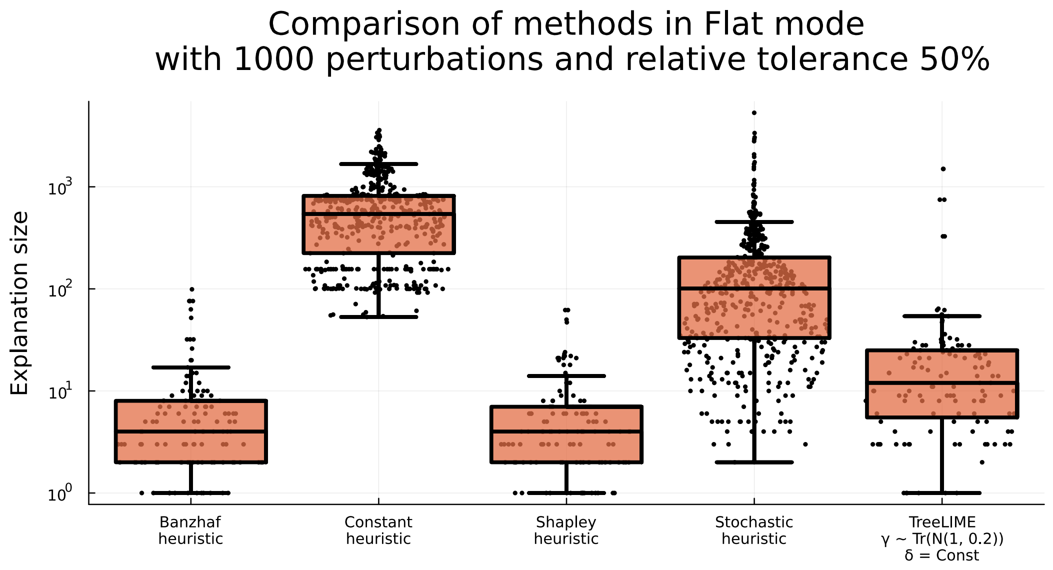

Figure 6.23. This figure compares ExplainMill.jl methods in Flat mode with 1000 perturbations against Flat TreeLIME with 1000 perturbations.

Chapter 7
TreeLIME analysis

This chapter will analyze the weak points of the TreeLIME method.

7.1 Flat TreeLIME analysis

Flat TreeLIME had a significantly worse performance than Shapley or Banzhaf heuristic. The following text will analyze why.

ExplainMill.jl uses a system of the mask to prune a sample. However, these masks are not independent of each other. Masking elements higher in the hierarchy affect the elements lower in the hierarchy. This problem is illustrated on datasets with smaller samples than CAPEv2 [17] to be easier to visualize. Figure 7.1 is a mask of the sample from the mutagenesis classification problem. Next to each mask is the size of the mask. Product masks do not have any size next to them, as their size is always zero. They do have the ability to mask their children. Product masks only group the children's masks and serve to construct the necessary hierarchy.

Mask in Figure 7.1 has three levels. Masks on the same level have the same distance from the root. Table 7.1 describes each child's mask level.

```
typename(ExplainMill.ProductMask)
  ├────────── lumo: FeatureMask: (1,)
  ├────────── inda: CategoricalMask: (1,)
  ├────────── logp: FeatureMask: (1,)
  ├──── mutagenic: CategoricalMask: (1,)
  ├────────── ind1: CategoricalMask: (1,)
  └───────── atoms: BagMask: (27,)
                      └── typename(ExplainMill.ProductMask)
                            ├──────── element: CategoricalMask: (27,)
                            ├────────── bonds: BagMask: (68,)
                            │                     └── typename(ExplainMill.ProductMask)
                            │                           ├──────── element: CategoricalMask: (68,)
                            │                           ├── bond_type: CategoricalMask: (68,)
                            │                           ├────────── charge: FeatureMask: (1,)
                            │                           └── atom_type: CategoricalMask: (68,)
                            ├────────── charge: FeatureMask: (1,)
                            └──── atom_type: CategoricalMask: (27,)
```

Figure 7.1. Sizes of masks of mask featured in Fig. 3.1. Next to each mask is the size of the mask in brackets. The size corresponds to how many observations or nodes the mask can prune or keep.

Masks in Layer 1	size	Masks in Layer 2	size	Masks in Layer 3	size
FeatureMask	1	CategoricalMask	27	CategoricalMask	68
CategoricalMask	1	BagMask	68	CategoricalMask	68
FeatureMask	1	FeatureMask	1	FeatureMask	1
CategoricalMask	1	CategoricalMask	27	CategoricalMask	68
CategoricalMask	1				
BagMask	27				

Table 7.1. This table describes the layer for each mask in 7.1. The colors are used to highlight the dependencies of the masks.

The Table 7.1 highlights the dependencies between masks. These dependencies are caused because there are multiple ways to prune a node or observation. The following dependencies are in the mask 7.1:

- All green masks in all Layers and the blue BagMask in Layer 1 are independent. Any other values do not restrict them. They can freely have `True` or `False` values.
- The blue and purple masks in Layer 2 depend on the blue BagMask in Layer 1. If the blue BagMask has a `True` value on index i, then the blue CategoricalMasks can have free values of `True` or `False` on index i. However, if the blue BagMask has a `False` value on index i, then the blue CategoricalMasks also have `False` values on index i. That is because if they had a `True` value on index i. This mask would be misleading for the model because the according label would be generated with that observation pruned due to BagMask on Layer 1. However, the model would believe that the `True` value in Layer 2 had some impact on the label, but it did not.
- The purple BagMask depends on the blue BagMask in a more complicated manner. The purple BagMask has 27 bags, and each of those bags is dependent on a different index in the Blue BagMask. So if the blue BagMask contains a `False` value on index i, all values in bag i of the purple BagMask must be `False`. Otherwise, the mask would be misleading.
- Lastly, all purple CategoricalMasks depend on the purple BagMask, which, as mentioned, depends on the blue BagMask.

For example, if the blue BagMask contained a `False` value on index i, this forces two more `False` values in the blue CategoricalMasks. Assuming that the bag with index i has a size of three. Three more forced `False` values would be in the purple BagMask. Moreover, they would force three `False` values in each of the purple CategoricalMasks. To summarize, one `False` value in the blue BagMask could force 14 other values to be `False`. Depending on the bag size with index i in purple BagMask, it could be more or less. The average bag size in the purple BagMask is $68 \div 27 \approx 2.5$.

Flat TreeLIME tries to explain this sample by taking all masks from all layers and putting them into a flat binary vector of ones. This vector has a size of $1+1+1+1+1+27+27+68+1+27+68+68+1+68 = 360$. It iterates over all items in this vector, and with perturbation chance γ, it overrides the item from `True` to `False`. Then, it sets zero to all items with a parent with a zero to remove parentless observations. Even though this pruning is necessary to remove incorrect representations of masks, it also introduced relationships and correlations between the predictors.

Regression with correlated predictors performs poorly, and the resulting coefficients are unsuitable for interpretability. Doc. Ing. Jiří Kléma, Ph.D. in [32] on slide 13 states that: “Correlations among predictors cause problems, the variance of all coefficients tends to increase, sometimes dramatically, interpretations become hazardous...”

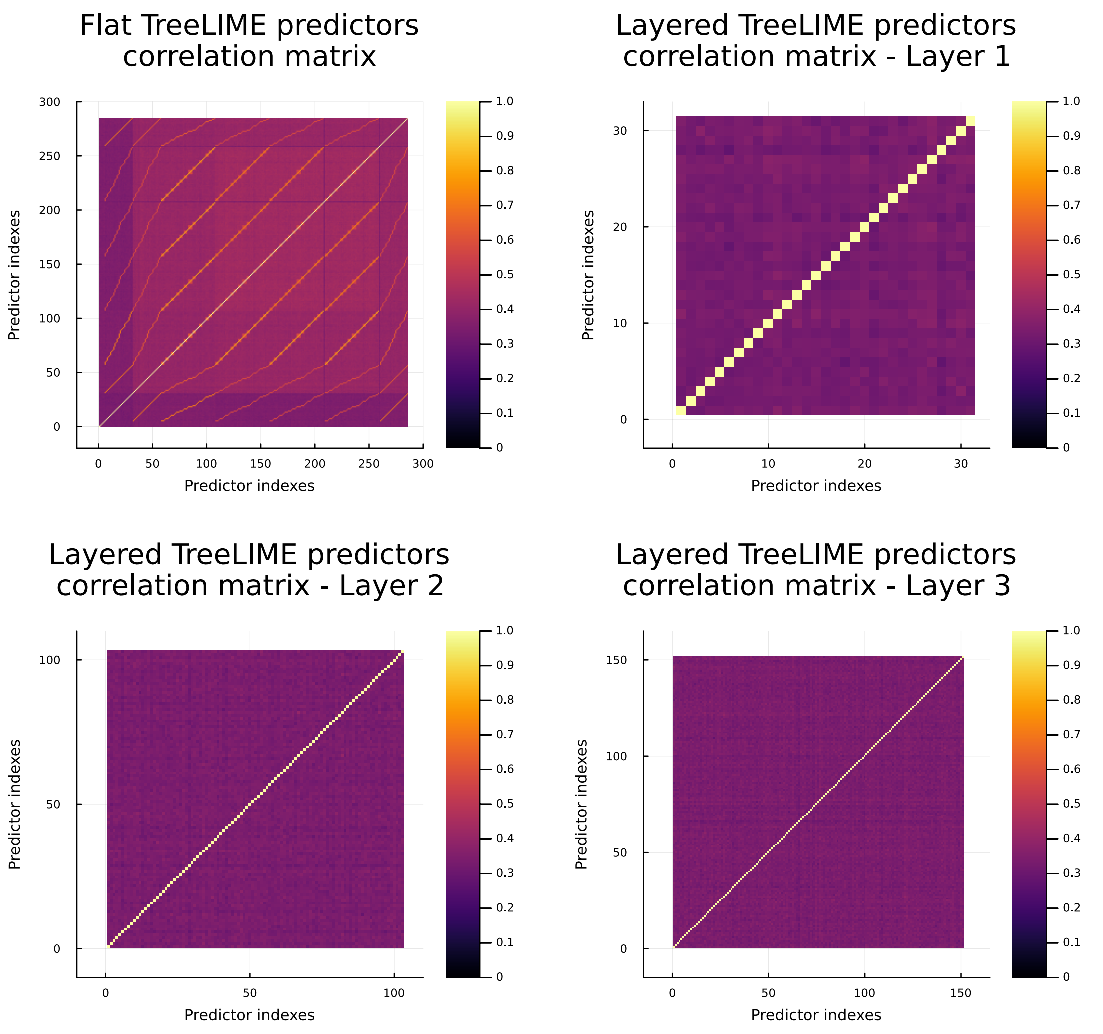

Figure 7.2. This figure is blurred in Apple Preview or PDF readers with enabled image smoothing. For optimal reading experience, use Adobe Acrobat Reader or turn off image smoothing in your preferred PDF reader. This figure visualizes correlation matrices between the predictors used in logistic regression.

In Figure 7.2, all correlation matrices are visualized as heatmaps due to their size. The top left plot shows the correlation matrix for Flat TreeLIME. We see that the correlation between the same predictors is always one, which creates the yellow diagonal line from the lower left corner to the top right corner. However, there are more correlation patterns in this plot. These correlation patterns result from ensuring that the final mask does not have parentless observations and is, therefore, non-misleading for the model. Other plots do not have any correlation patterns other than the diagonal line, which is expected. Also, we can see that the sum of the number of predictors in Layers 1, 2, and 3 matches the number of predictors in the Flat TreeLIME. Note: The number of predictors is not 360 because different sample x was chosen, but the figure would be highly similar for any sample x.

Flat TreeLIME does not perform well due to the correlations and dependencies between the predictors, making the coefficients β unsuitable for interpretability.

7.2 Layered TreeLIME analysis

Layered TreeLIME performed significantly better than Flat TreeLIME, however, Shapley and Banzhaf heuristics still performed noticeably better. In this section, we will analyze why.

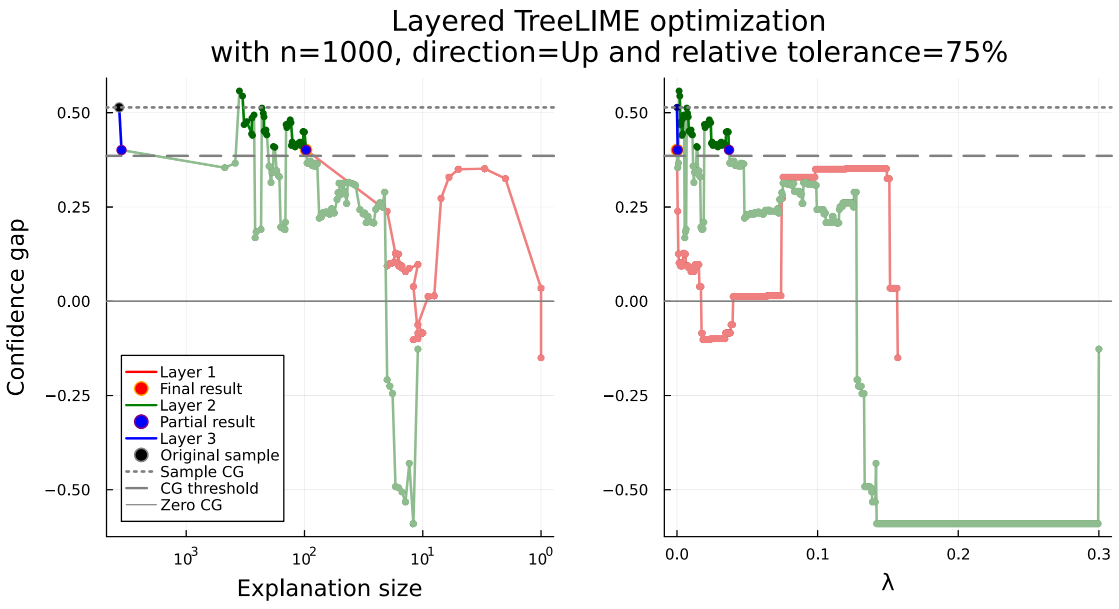

Figure 7.3. Example of Layered TreeLIME failing to find a small explanation.

Figure 7.3 Layered TreeLIME found an explanation with the number of leaves $\approx 10^2$. We know that a smaller solution exists, which can be found using more advanced Search types mentioned in 3.3.2, such as Random Removal explained in 1.9. So why TreeLIME have not found it? We believe the optimization path, defined in subsection 7.2.1, has additional restrictions.

7.2.1 TreeLIME optimization path

TreeLIME optimization graph G contains a vertex for every mask parameters setting TreeLIME evaluated. For example, in Figure 7.3, every point scattered in the plot, no matter its color, would be a single vertex in G.

There is an edge between any two vertices u and v only if TreeLIME evaluated u and v in the same layer one directly after the second or if TreeLIME has chosen u as the best point from a given layer and vertex v is the first vertex in the next layer.

The TreeLIME optimization path for Figure 7.3 is the connected line from the point labeled as `Original sample` to the point labeled as `Final result`, which is in the case of Figure 7.3 drawn behind the point labeled as `Partial result`. Hence, the only thing visible is the orange stroke of the point `Final result`. Note: The TreeLIME optimization path can include points below the confidence gap threshold and points generated inside different layers. Therefore, it might be represented with multiple colors in the plots.

For this work, the TreeLIME optimization path shall be defined as a path from the vertex representing the original sample to the vertex representing the resulting mask setting returned by TreeLIME. Path in the graph is a series of distinct vertices, and because the graph G is a tree, there is always a single TreeLIME optimization path.

7.2.2 TreeLIME optimization path restrictions

In Flat TreeLIME, the only restriction on the TreeLIME optimization path is that the last point has a confidence gap above the confidence gap threshold. This restriction is necessary to achieve the target result. The TreeLIME optimization path of Flat TreeLIME does not add any further restrictions.

However, Layered TreeLIME adds some additional restrictions to the TreeLIME optimization path. These restrictions are that each transition from a layer to a different layer has to happen on a point with a confidence gap above the confidence gap threshold. For n layers, this adds $n-1$ additional restrictions. These additional restrictions make it more difficult for layered TreeLIME to provide smaller explanations. It is possible that the optimization path leading to a better explanation does not include these $n-1$ points above the confidence gap threshold. When n is small, we could try a brute-force method of starting the next layer from each point of the previous layer. However, as the number of layers increases, the number of combinations increases exponentially. Therefore, it is not a practical approach.

Chapter 8 Conclusion

In this chapter, we will draw the main conclusions from this work.

8.1 Future work based on TreeLIME analysis

Flat TreeLIME has problems with dependencies and correlation among predictors but does not have problems with multiple restrictions on its optimization path. Layered TreeLIME has problems with additional restrictions on the optimization path, but it has no dependencies or correlations among its predictors.

Removing the issues of Layered TreeLIME seems very difficult. However, resolving issues with Flat TreeLIME seems more straightforward. We would need to develop a masking system that cannot mask a single node or observation in two or more ways. It would probably look like a leaf masking method. This method would probably be used only inside TreeLIME as ExplainMill.jl masking offers more features such as:

- Masking an entire bag
- Masking a feature in every bag
- Masking only some features in some bags

The new masking system would support only the last of these three properties. Therefore, it would not be applicable in most ExplainMill.jl use cases where more sophisticated masking is necessary.

Unfortunately, due to time constraints, we were not able to implement the new masking system, partly because the issues became apparent only in later stages of development and were not noticeable on initial smaller datasets like `mutagenesis`, `hepatitis` and `device_id` from [11], where Flat TreeLIME performed well.

8.2 Conclusion

To conclude, this thesis introduced a new method for explaining Hmil models called TreeLIME. The TreeLIME method was heavily tested on the CAPEv2 dataset [1] and showed disappointing performance. This work analyzed the method, revealed the underlying issues, and proposed an improved Layered TreeLIME version. Layered TreeLIME had a comparable performance to the current state-of-the-art methods. Lastly, this thesis described how the TreeLIME method could be further improved in future work.

Appendix A
Used AI software

The following AI software was used:

- **Grammarly** - Grammarly without any generative AI features was used to correct the grammar and improve the sentence structure of this thesis.
- **GitHub Copilot** - GitHub Copilot was used to speed up the development by generating easy-to-understand lines or very short snippets of Julia code that were manually modified to fit into the current context.
- **Large language models** - No large language model was used to generate or rephrase any part of the text of this thesis.

Appendix B
Source code

All the source code developed for is open source and available on GitHub in two repositories:

- **MyExplainMill** - MyExplainMill is a fork of ExplainMill further modified to fit the needs of this thesis. TreeLIME is implemented inside MyExplainMill. MyExplainMill is available at `https://github.com/ondraveres/MyExplainMill`
- **myscripts** - The repository `myscripts` stores all the code for running experiments, training models, plotting visualizations and everything else, which was developed for this thesis. This repository was originally based on folder `scripts` in code for the article [11]. The repository `myscripts` is available at `https://github.com/ondraveres/myscripts`

References

[1] Branislav Bosansky, Dominik Kouba, Ondrej Manhal, Thorsten Sick, Viliam Lisy, Jakub Kroustek, and Petr Somol. *Avast-CTU Public CAPE Dataset*. 2022. `https://arxiv.org/abs/2209.03188`.

[2] Prarthana Dutta, Naresh Muppalaneni, and Ripon Patgiri. *A Survey on Explainability: Why Should We Believe the Accuracy of A Model?* 2020.

[3] Philipp Hacker, Ralf Krestel, Stefan Grundmann, and Felix Naumann. Explainable AI under contract and tort law: legal incentives and technical challenges. *Artificial Intelligence and Law*. 2020, 28 DOI 10.1007/s10506-020-09260-6.

[4] Maximilian Pichler, and Florian Hartig. *Machine Learning and Deep Learning – A review for Ecologists*. 2022.

[5] Vikas Hassija, Vinay Chamola, Atmesh Mahapatra, Abhinandan Singal, Divyansh Goel, Kaizhu Huang, Simone Scardapane, Indro Spinelli, Mufti Mahmud, and Amir Hussain. Interpreting Black-Box Models: A Review on Explainable Artificial Intelligence. *Cognitive Computation*. 2023, 16 DOI 10.1007/s12559-023-10179-8.

[6] Farhad Shakerin, and Gopal Gupta. *White-box Induction From SVM Models: Explainable AI with Logic Programming*. 2020.

[7] Mark Ibrahim, Melissa Louie, Ceena Modarres, and John Paisley. *Global Explanations of Neural Networks: Mapping the Landscape of Predictions*. 2019.

[8] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. 2016.

[9] Scott Lundberg, and Su-In Lee. *A Unified Approach to Interpreting Model Predictions*. 2017.

[10] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. *GNNExplainer: Generating Explanations for Graph Neural Networks*. 2019.

[11] Tomáš Pevný, Viliam Lisý, Branislav Bošanský, Petr Somol, and Michal Pěchouček. *Explaining Classifiers Trained on Raw Hierarchical Multiple-Instance Data*. 2022.

[12] David Stutz, Alexander Hermans, and Bastian Leibe. Superpixels: An evaluation of the state-of-the-art. *Computer Vision and Image Understanding*. 2018, 166 1–27. DOI 10.1016/j.cviu.2017.03.007.

[13] Mirka Saarela, and Susanne Jauhiainen. Comparison of feature importance measures as explanations for classification models. *SN Applied Sciences*. 2021, 3 DOI 10.1007/s42452-021-04148-9.

[14] Sahil Verma, Varich Boonsanong, Minh Hoang, Keegan E. Hines, John P. Dickerson, and Chirag Shah. *Counterfactual Explanations and Algorithmic Recourses for Machine Learning: A Review*. 2022.

[15] Tomas Pevny, and Petr Somol. *Discriminative models for multi-instance problems with tree-structure*. 2017.

[16] Šimon Mandlík, and Matěj Račinský. *HierarchicalUtils.jl is a package providing abstract functionality over hierarchical structures*. 2021.
`https://github.com/CTUAvastLab/HierarchicalUtils.jl`.

[17] Kevin O'Reilly. *CAPE: Malware Configuration And Payload Extraction*. 2019.
`https://github.com/kevoreilly/CAPEv2`.

[18] Simon Mandlik, Matej Racinsky, Viliam Lisy, and Tomas Pevny. *Mill.jl and JsonGrinder.jl: automated differentiable feature extraction for learning from raw JSON data*. 2021.

[19] Tomáš Pevný. *Hierarchical Multiple Instance Learning | Tomas Pevny | JuliaCon 2021*. 2021.
`https://www.youtube.com/watch?v=Bf0CvltIDbE`.

[20] Tomáš Pevný, and Matěj Račinský. *Dependency aware feature selection is a simple but effective method*. 2018.
`https://github.com/pevnak/Duff.jl`.

[21] Petr Somol, Jiří Grim, and Pavel Pudil. *Fast dependency-aware feature selection in very-high-dimensional pattern recognition*. In: *2011 IEEE International Conference on Systems, Man, and Cybernetics*. 2011. 502-509.

[22] Shaheen S. Fatima, Michael Wooldridge, and Nicholas R. Jennings. A linear approximation method for the Shapley value. *Artificial Intelligence*. 2008, 172 (14), 1673-1699. DOI https://doi.org/10.1016/j.artint.2008.05.003.

[23] J.F. Banzhaf. Weighted voting doesn't work: A mathematical analysis. *Rutgers Law Review*. 1965, 19 (2), 317-343.

[24] Lloyd S Shapley. A Value for n-Person Games. 1953, 307–317.

[25] Manu Joseph. *"Interpretability part 3: opening the black box with LIME and SHAP*. 2018.
`https://www.kdnuggets.com/2019/12/interpretability-part-3-lime-shap.html`.

[26] Joanne Peng, Kuk Lee, and Gary Ingersoll. An Introduction to Logistic Regression Analysis and Reporting. *Journal of Educational Research - J EDUC RES*. 2002, 96 3-14. DOI 10.1080/00220670209598786.

[27] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*. 1996, 58 (1), 267–288.

[28] Simon Kornblith, Jack Dunn, Simon Byrne, and Alex Arslan. *Julia wrapper for fitting Lasso/ElasticNet GLM models using glmnet*. 2013.
`https://github.com/JuliaStats/GLMNet.jl`.

[29] Jerome Friedman, Robert Tibshirani, and Trevor Hastie. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*. 2010, 33 (1), 1–22. DOI 10.18637/jss.v033.i01.

[30] J. Kenneth Tay, Balasubramanian Narasimhan, and Trevor Hastie. Elastic Net Regularization Paths for All Generalized Linear Models. *Journal of Statistical Software*. 2023, 106 (1), 1–31. DOI 10.18637/jss.v106.i01.

[31] Jiří Kléma. *Generalized linear models*. 2022.
`https://cw.fel.cvut.cz/b231/_media/courses/b4m36san/san_glms.pdf`.

[32] Jiří Kléma. *Multiple (non) linear regression*. 2022.
`https://cw.fel.cvut.cz/b231/_media/courses/b4m36san/san_regression_2022.pdf`.

[33] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*. 2017, 59 (1), 65–98.

[34] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable Modelling with Flux. *CoRR*. 2018, abs/1811.01457

[35] Dahua Lin, John Myles White, Simon Byrne, Douglas Bates, Andreas Noack, John Pearson, Alex Arslan, Kevin Squire, David Anthoff, Theodore Papamarkou, Mathieu Besançon, Jan Drugowitsch, Moritz Schauer, and other contributors. *JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions*. 2019.
`https://doi.org/10.5281/zenodo.2647458`.

[36] M Jette, C Dunlap, J Garlick, and M Grondona. SLURM: Simple Linux Utility for Resource Management. 2002.

[37] Ming Fan, Wenying Wei, Xiaofei Xie, Yang Liu, Xiaohong Guan, and Ting Liu. *Can We Trust Your Explanations? Sanity Checks for Interpreters in Android Malware Analysis*. 2020.