aspBEEF: Explaining Predictions Through Optimal Clustering (preliminary report)

Pedro Cabalar[1], Rodrigo Martín[2], Brais Muñiz[1,2] and Gilberto Pérez[1,2]

[1]University of Coruña (UDC), A Coruña, Spain
[2]CITIC Research Center, A Coruña, Spain

Abstract
In this paper, we introduce aspBEEF, a tool for generating explanations for the outcome of an arbitrary machine learning classifier. The tool implements Grover's et al. framework known as Balanced English Explanations of Forecasts (BEEF) that generates explanations in terms of finite intervals over the values of the input features. In BEEF, outcomes of a classifier are clusterized in box-shaped clusters, whose limits are then used for explaining the behaviour of the model. Resulting clusters can be characterized in terms of three quality measures: purity, overlapping and inclusion. In general, it is preferable to maximize purity and inclusion and to minimize overlapping. The problem of obtaining the optimal BEEF clusters for a set of classified samples has been proved to be NP-hard, and so forth BEEF existing implementation computes an approximation. In this work we use instead an encoding into the Answer Set Programming paradigm, specialized in solving NP problems, to guarantee that the computed solutions are optimal.

Keywords
Knowledge Representation, Answer Set Programming, Explainable AI

1. Introduction

One of the biggest challenges to overcome in the field of machine learning (ML) is explainability. To able to interpret models produced by ML algorithms and to be able to explain their predictions and estimations is desirable in general but required in some cases. For example, in domains where decisions can seriously affect people's lives, domain experts need to understand the decisions of ML models in order to trust them, and for avoiding biased or unfair decisions. Examples of these domains could be healthcare or the legal domain. Besides, there also exist some regulations such as the General Data Protection Regulation (GDPR), which requires the system to provide the user with an explanation of how its decisions are made. Some ML algorithms, such as Decision Tree, produce transparent models by design, so it is easy to interpret them and to obtain local and/or global explanations from them. However, the best performance is often achieved by non-transparent models for any task, so an external method for explaining those models is required. Unfortunately, explaining why a non-transparent ML model makes a given prediction is often a non-trivial task, especially when the aim is to do it in human terms.

Explainable Artificial Intelligence (XAI) field studies how to obtain explainable models either by developing new transparent ML algorithms or by developing techniques that aim to explain any ML model disregarding its underlying algorithm. Such *model-agnostic* techniques, allow decoupling the process of obtaining a good model for prediction from the explanation step so the engineer has no longer to be limited to a transparent model which may not obtain the best performance for the specific task. Another important consequence is that they could be applied for explaining already deployed models, so there is no need to develop new models in order to meet the regulations. One interesting kind of *model-agnostic* techniques are those that make use of surrogate models, which are transparent models that approximate the outcome of the original model and provide explanations. The idea is to first develop a predictor model which may not be transparent, and then develop a surrogate model based not on actual data, but on the predictions of the first model. The original model is used for making predictions, and the surrogate model is used for obtaining explanations. Both resulting models do not always have to be in an agreement given a particular input. The level of agreement depends on the one hand on the quality of the predictions used to build the surrogate model in terms of quantity and completeness, and on the other hand on parameters specific to the selected surrogate model.

One *model-agnostic*, *surrogate model-based* is Balanced English Explanations of Forecasts (BEEF) [1]. The algorithm employs a special clustering method as a surrogate

ASPOCP 2021: Workshop on Answer Set Programming and Other Computing Paradigms 2021 co-located with ICLP 2021 Porto, Portugal, September 30, 2021
pedro.cabalar@udc.es (P. Cabalar); r.martin1@udc.es (R. Martín); brais.mcastro@udc.es (B. Muñiz); gpvega@udc.es (G. Pérez)
https://www.dc.fi.udc.es/~cabalar/ (P. Cabalar); https://github.com/trigork (R. Martín); https://github.com/bramucas (B. Muñiz)
0000-0001-7440-0953 (P. Cabalar); 0000-0002-9817-6666 (B. Muñiz); 000-0001-6169-6101 (G. Pérez)
© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

model for explaining the outcome of ML binary classifiers. The quality of the surrogate model that is computed is measured by three metrics (*overlapping*, *purity* and *inclusion*). Then, the surrogate clustering model is used for providing natural language explanations for local predictions in terms of ranges over the input features of the original model.

However, BEEF's finding of such clusters is heuristic-guided and does not provide the optimal solutions. In this paper, we present a prototype called `aspBEEF`, which provides an optimal solution given a preference over the quality metrics . It does so by the use of Answer Set Programming (ASP) for representing and solving the optimization problem and `asprin` for modelling the optimization preferences.

The contents of this paper are summarized in the following. Section 2 explains in more detail how the BEEF algorithm works. Section 3 provides a short description of ASP and `asprin`. Section 4 describes in detail how `aspBEEF` prototype works. Section 5 makes a short evaluation of the prototype. Finally, Section 6 makes some comments about the future work and concludes the paper.

2. BEEF algorithm

BEEF is a model-agnostic cluster-based XAI framework for explaining the outcome of ML binary classifiers. Given a set of predictions, BEEF computes a set of clusters that globally describe the classifier's behaviour. From a single prediction or classification, BEEF uses those clusters for providing a set of what authors call *balanced explanations*. This *balanced explanations* explain any outcome of the classifier in terms of intervals over its input features.

Once fed with predictions from the ML model, some traditional clustering method such as KMeans is used for obtaining a set of clusters that accurately group the predicted classes. This is run in a vector space where the input features are the dimensions, and each classifier prediction is a point labelled with its predicted class. Using the obtained cluster centroids as a starting point, BEEF approximates a set of *axis-aligned hyperrectangular-shaped* clusters around them. We will be calling the new clusters *boxes* from now on for short. As they are *axis-aligned*, the obtained boxes can be described as a set of ranges over the input features of the classifier. Each box is associated with a predicted class, which is the majority predicted class within that cluster. As a result of this approximation process, boxes could overlap each other and predictions could fall outside any box. Figure 1 shows the general process of BEEF.

The obtained boxes generally describe the behaviour of the classifier and they can also be used for obtaining explanations from a single outcome. Given a single classified sample, a box where the point fell in can be used for building an explanation that argues why the example belongs to the box's associated class based on the box boundaries. BEEF explanations are expressed in natural language by selecting sentences from previously written text templates to explain each of the boxes to the user.

Although a single explanation may be preferable in most cases, as the boxes can overlap each other, a single classified sample could fall within more than a cluster, leading to multiple explanations or even none when it falls outside of any box. Besides, the class associated with an explanation does not always match the outcome of the original classifier, which leads to two types of explanations: *supporting explanations* when they match, and *opposing explanation* it does not. All the possible explanations (regardless of their types) presented together is what is called a *balanced explanations*.

The previously described properties depend on the quality of the obtained boxes, which can be quantified using the following three quality metrics:

1. **Overlapping**: two boxes can overlap each other when they share space for at least one dimension (or input feature). For example, if for dimension d_1, box B_1 lays within the range [2,10] and box B_2 lays within the range [-2, 6] then their overlap of that dimension is 4. The *overall overlapping* of two pairs of boxes is then defined following the geometric intersection, as the product of all their overlaps for all dimensions. When approximating the boxes, BEEF algorithm aims to minimize the overlapping or even suppress it.
2. **Purity**: it is defined as the percentage of samples of the majority class inside the box and is tried to be maximized for all boxes by the BEEF algorithm.
3. **Inclusion**: is the percentage of samples that are included in at least one box concerning the total of samples. Since the algorithm aims to explain the most elements possible, BEEF tries to maximize inclusion.

The original algorithm creates the sets of boxes from the initially classified data and then adjusts the boundaries of the boxes in an iterative manner, optimizing the metrics mentioned above until a given set of thresholds over the previously defined properties are satisfied. Then a feature selection is performed, trying to keep only the subset of relevant features in terms of the information they provide to the final explanations. This is done through a greedy algorithm that activates or deactivates the features that provide the largest or lowest marginal contribution, respectively.

Through this process, the original BEEF algorithm obtains good solutions at the cost of losing optimality. Our `aspBEEF` tool aims to obtain an optimal set of boxed

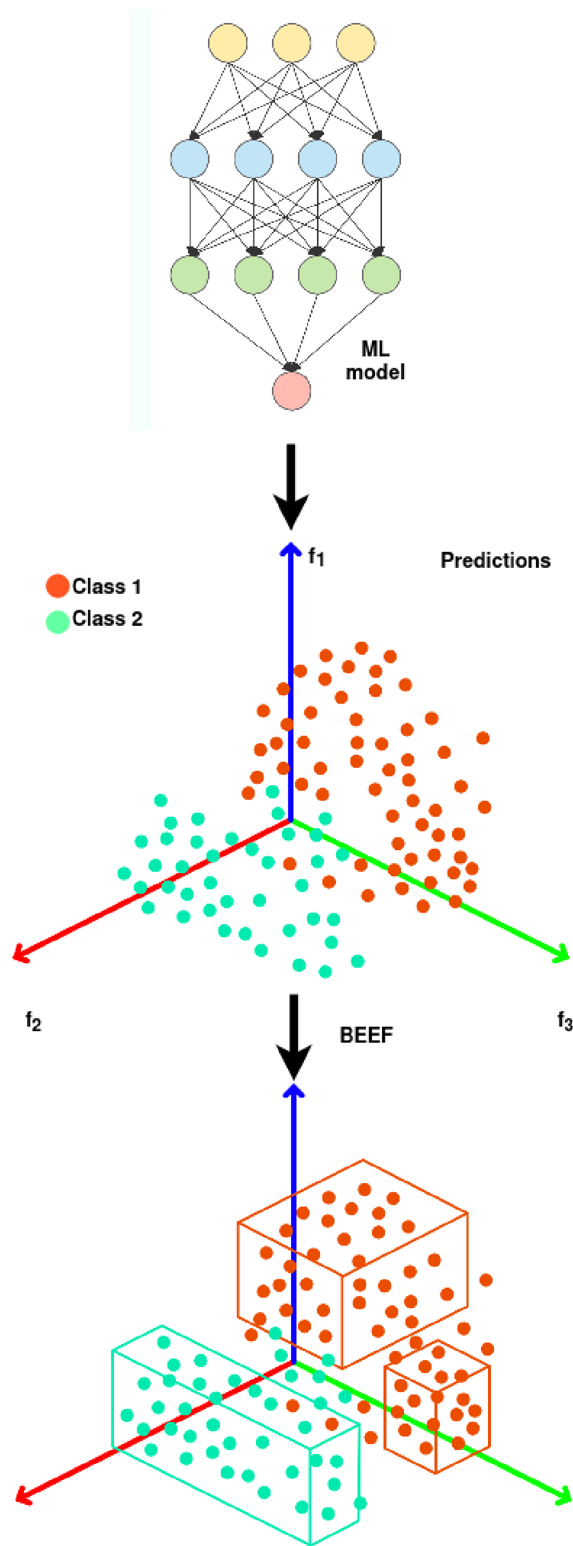


Figure 1: The general process of BEEF framework. Predictions from a ML model are represented in a vector space where axis are the model's input features (e.g., f_1, f_2 and f_3 in the figure). Points are then grouped in hyperrectangular axis-aligned clusters which will be used for explaining the model's behaviour in terms of intervals over the input features. Unlike aspBEEF, BEEF only works with binary classifiers.

given a set of predictions and a defined order over the boxes properties (*overlapping*, *purity* and *inclusion*).

3. ASP and asprin

aspBEEF relies in ASP [2, 3, 4] for both specifying and solving the optimization problem stated above. We assume basic familiarity with ASP for this article. In particular, aspBEEF is built on top of clingo [5], through the use of its python api [6], for both grounding and solving. In addition, we make use of asprin for modelling the optimization preferences instead of making direct use of ASP *#minimize* statements.

asprin [7] is an ASP extension tool that allows the specification of preferences easily and flexibly. It provides a *#preference* statement that allows declaring user-defined preferences which allow more expressiveness and functionality than ASP. In the following example from the authors,

```
#preference(costs, less(weight)){40:sauna, 70:
    dive}.
#optimize(costs).
```

the *#preference* statement declares a preference identified by *costs* with type *less(weight)* and some weights assigned to the atoms *sauna* and *dive*. These preferences can be used for declaring new ones, so the use of the *#optimize* statement is mandatory for pointing asprin which preference relation must be used for the optimization. With the previous setup, stable models which include those atoms will sum their associated cost, and those with lower cost will be preferred during solving. The computation of these preferred stable models is done via what is called *preference programs*. The last computed stable model is then provided as the optimal one.

4. The aspBEEF Tool

The BEEF algorithm first clusters the classified samples and uses them as the starting point for finding a set of box-shaped clusters which perform well in terms of the quality metrics defined in Section 2 The problem has been proven to be NP-complete by the authors, so therefore the original algorithm performs a greedy search to obtain a good set of boxes. We present aspBEEF as an ASP implementation of such an algorithm for obtaining the optimal set of boxes. The presented implementation is an ongoing development and has some differences from the original one that will be explained throughout this section.

As in the original algorithm, aspBEEF initially takes as an input a set of already classified samples, typically the outcome of an already trained ML classifier. The data must be introduced into the tool in CSV format, whereas the columns encode all the input features and the outcome class from the ML model. The name of the column which encodes the classification outcome is needed to be input into the tool by the user. The samples are then clustered using a traditional method such as KMeans or any other. As a result of the clustering, each sample is now related with its original class (the outcome of the ML model) but also with its cluster. Note that, if the user has already clustered the data (for example, it is known a domain-dependent way to cluster the samples), this can be input into the tool by the user for using it as the starting point instead. The clustered data is then encoded to ASP facts to use alongside the rest of the ASP rule set. See figures 2 and 3 for an example of the encoding. Note that, since clingo cannot work with floating-point numbers, all of the numerical data is automatically transformed to an integer by multiplying the original values by the smallest necessary power of ten.

In the example, the features from the samples (the input features of the ML classifier, also the columns from the input CSV file) are represented using the `attribute/1` predicate. There is an additional attribute *predtarget*, which stands for the name of the generated column (named *predtarget*) which encodes the cluster for each sample. Each sample is then encoded row by row (i.e. sample by sample) using `value/3` predicate. Its arguments specify, respectively, the identifier of the sample (as an integer number), the feature which is encoded and the value of the sample for that very feature.

The previously described facts are used alongside the ASP rule set encoding the BEEF search problem. Unlike BEEF, aspBEEF receives the number of input features to be used, which is encoded in the ASP rule set as a constant named `selectcount`. That constant is used for selecting which features to use for building the boxes in the following choice rule,

```
selectcount {selattr(A):attribute(A), not
    classtarget(A), not predtarget(A)}
    selectcount.
```

where the `classtarget/1` indicates which is the class feature (i.e. the outcome classification) and `predtarget` indicates which is the cluster feature. The user is allowed to specify some fixed features through a command-line option, which are called *fixed features*. Those are forced to be selected by adding `selattr` facts to the ASP program, so they no longer depend on being chosen by the choice rule. The rest used features are called `free features`. Note that fixed features considerably prune the search space as evaluation results will show.

The user defines the number of boxes to find, which is passed to the clustering algorithm and therefore

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
```

```
attribute('sepal_length'). attribute('sepal_width'). attribute('petal_length').
attribute('petal_width'). attribute('species'). attribute('predtarget').

value(0,'sepal_length',51). value(0,'sepal_width',35). value(0,'petal_length',14).
value(0,'petal_width',2). class(0, 'c_iris_setosa'). cluster(0, 'c1').
value(1,'sepal_length',49). value(1,'sepal_width',30). value(1,'petal_length',14).
value(1,'petal_width',2). class(1, 'c_iris_setosa'). cluster(1, 'c1').
value(2,'sepal_length',47). value(2,'sepal_width',32). value(2,'petal_length',13).
value(2,'petal_width',2). class(2, 'c_iris_setosa'). cluster(2, 'c2').
```

Figure 2: Sample of the original CSV data clustered and parsed to ASP facts

```
selattr('sepal_length') selattr('sepal_width') selattr('petal_length') selattr('petal_width')
rectinliercount('p_0',50) rectinliercount('p_1',55) rectinliercount('p_2',35)
rectcluster('p_0','p_0') rectcluster('p_1','p_1') rectcluster('p_2','p_2')
minrectval('p_0','sepal_length',43,58) minrectval('p_0','sepal_width',23,44)
minrectval('p_0','petal_length',10,19) minrectval('p_0','petal_width',1,6)
minrectval('p_1','sepal_length',49,64) minrectval('p_1','sepal_width',20,34)
minrectval('p_1','petal_length',30,51) minrectval('p_1','petal_width',10,24)
minrectval('p_2','sepal_length',62,79) minrectval('p_2','sepal_width',26,38)
minrectval('p_2','petal_length',50,69) minrectval('p_2','petal_width',16,25)
outliercount(10) overlapcount(0) impurecount(0)
```

Figure 3: Output facts of an aspBEEF execution over the IRIS dataset

match the number of pre-computed clusters. For each box (identified using `rectangle/1` predicate) and selected feature, a pair of existing values of the dataset are chosen to define the lower and upper limits for the particular feature. The following choice rule generates the pair of values V for each box R and feature A.

```
2 {rectval(R,A,V) : fringevalue(R,A,V)} 2 :-
     rectangle(R), selattr(A).
```

`fringevalue/3` encodes the values V in the feature A which fall within the fringe of the precomputed cluster R. The fringe is defined as the outer rim of each cluster, its thickness can be configured by parameter. By default this fringe covers all of the cluster data, but it can be made thinner so we avoid using the innermost values. This helps speeding up the process but we risk losing the optimal values when selecting the box bounds. For each cluster, R a box is defined as the set of `rectval` pairs for each feature.

For each defined box, quality metrics are computed. To deal with them, it is necessary to know which samples fall inside which boxes. The following code takes care of this.

```
attrinlier(R,I,A) :- value(I,A,V), rectval(R,A,
     V0), rectval(R,A,V1), V>=V0, V<=V1, V0<V1.
attroutlier(R,I,A) :- value(I,A,V), rectval(R,A,
     V0), rectval(R,A,V1), V<V0, V0<V1
attroutlier(R,I,A) :- value(I,A,V), rectval(R,A,
     V0), rectval(R,A,V1), V>V1, V0<V1.

rectoutlier(R, I) :- attroutlier(R,I,A).
rectinlier(R, I) :- attrinlier(R,I,A), not
     rectoutlier(R,I).

rectinliercount(R,C) :- rectangle(R), C=#count{
     I : rectinlier(R,I)}.
```

Predicates `attrinlier/3` and `attroutlier/3` (respectively) state if a sample I, fall inside the range of box R (or not) for the feature A. Consequently, predicates `rectoutlier/2` and `rectinlier/2` (respectively) state if a sample I, fall inside the box R (or not), when taking into account all the selected features. Finally, the `rectinliercount/2` predicate tells the total number of samples C which fall inside the box R.

There exist some differences in the way some quality

metrics are handled by aspBEEF concerning the original approach. For instance, while BEEF maximizes the inclusion (previously defined in Section 2), aspBEEF instead equivalently minimizes the exclusion (i.e. samples which fall outside any box, or *outlier samples*). Using the previous rules, the following ASP predicates point out which points I are outliers and which are not.

```
inlier(I) :- rectinlier(R,I).
outlier(I) :- pointid(I), not inlier(I).
```

Overlapping is handled more differently. In the original approach, overlapping is defined as the geometrical intersection areas of each pair of boxes. For simplicity, aspBEEF instead consider *overlapping samples* which are samples that fall inside more than one box. Such samples are handled in the following rule reusing the `rectinlier/2` predicate shown before.

```
overlappoint(I,R1,R2) :- rectinlier(R1,I),
     rectinlier(R2,I), R1<R2.
```

Finally, as with inclusion, instead of maximizing purity, aspBEEF minimizes impurity. To handle impurity, samples of the minority class within each box, or *impure samples*, are counted. That is handled by the following rules.

```
impure(R) :- rectangle(R), rectinlier(R,I1),
     rectinlier(R,I2), I1<I2, cluster(I1,C1),
     cluster(I2,C2), C1!=C2.
impurity(R,CL1,IC) :- impure(R), cluster(CL1),
     IC=#count{I: cluster(I,CL2), CL2!=CL1,
     rectinlier(R,I)}.
impurity(R,M) :- impure(R), M=#min{CLI :
     impurity(R,CL,CLI)}.
```

The impure boxes are defined as those with at least two samples belonging to different classes with the predicate `impure/1`. `impurity/3` predicate counts how many samples IC from the class $CL1$ are inside box R. Finally, `impurity/2` predicate states that count M but just for the minority class for each box R.

Using the rules given so far, the global quality metrics for a particular set of boxes are defined as follows,

```
outliercount(C) :- C=#count{I : outlier(I)}.
overlapcount(C) :- C=#count{I : overlappoint(I,
     R1,R2)}.
impurecount(C) :- C=#count{I : impurepoint(I)}.
```

where C is the respective count.

Given the metric definitions, aspBEEF relies in asprin preferences for finding the global optimal solution. For each metric, an asprin *#preference* statement is declared as in the code below.

```
#preference(overlap,less(cardinality)){
   overlappoint(I, R1, R2) : rectangle(R1),
        rectangle(R2), pointid(I)
}.
#preference(impurity,less(cardinality)){
   impurepoint(I): pointid(I)
}.
#preference(outlier,less(cardinality)){
   outlier(I) : pointid(I)
}.
#preference(all,lexico){
   overlapprio::**overlap;
   impurityprio::**impurity;
   outlierprio::**outlier
}.
#optimize(all).
```

The three preferences (respectively identified by `overlap`, `impurity` and `outlier`), are defined with the type *less(cardinality)*. They are aggregated within a *lexico* preference identified by *all*. Besides, they are sort by priority values which are encoded as the following ASP constants: *overlapprio*, *impurityprio* and *outlierprio*. These priorities define the order of preference for their respective quality metrics to be optimized. By default, the order is overlapping, then impurity and finally exclusion. Finally, the *#optimize* sentence indicates the *all* preference to be optimized.

The explained code so far is provided to asprin which iteratively finds better and better solutions (i.e. sets of boxes) until the optimal is found. Once found, this optimal is presented to the user as the set of ASP facts describing the ranges of each box as shown in Figure 4 or as a set of rules in form of text as shown in Figure 5.

Besides, our tool is also able to generate graphical HTML reports in the form of dynamic 2D scatter plots to graphically show the found solutions. In those reports, the user is allowed to select the features of the dataset by pairs and visualize the projection of the boxes for those two very dimensions alongside the classified samples. A summary of the quality of the solution metrics is also shown. An example of the visualization can be seen in Figure 6. Additionally, a sample live version of an HTML report using the IRIS dataset and four free features can be viewed in the authors' repository: http://trigork.github.io/asp/aspbeef/IRIS/

There exist some other differences in addition to those already explained. The first one is that aspBEEF implementation can handle more than two classes, unlike BEEF which only deals with binary problems. However, while BEEF also adjusts the number of boxes, the current implementation of aspBEEF needs this number to be fixed by the user to work.

The second one is that aspBEEF activates and deactivates features globally. This means that given a solution, all boxes use the same features. The original approach

```
selattr('sepal_length') selattr('sepal_width') selattr('petal_length') selattr('petal_width')
rectinliercount('p_0',50) rectinliercount('p_1',55) rectinliercount('p_2',35)
rectcluster('p_0','p_0') rectcluster('p_1','p_1') rectcluster('p_2','p_2')
minrectval('p_0','sepal_length',43,58) minrectval('p_0','sepal_width',23,44)
minrectval('p_0','petal_length',10,19) minrectval('p_0','petal_width',1,6)
minrectval('p_1','sepal_length',49,64) minrectval('p_1','sepal_width',20,34)
minrectval('p_1','petal_length',30,51) minrectval('p_1','petal_width',10,24)
minrectval('p_2','sepal_length',62,79) minrectval('p_2','sepal_width',26,38)
minrectval('p_2','petal_length',50,69) minrectval('p_2','petal_width',16,25)
outliercount(10) overlapcount(0) impurecount(0)
```

Figure 4: Output facts of an aspBEEF execution over the IRIS dataset

```
Rule(s) for Class p_0
 Rule #0
  sepal_length is between 4.3 and 5.8
  sepal_width is between 2.3 and 4.4
  petal_length is between 1.0 and 1.9
  petal_width is between 0.1 and 0.6
Rule(s) for Class p_1
 Rule #0
  sepal_length is between 4.9 and 6.6
  sepal_width is between 2.0 and 3.4
  petal_length is between 3.0 and 5.1
  petal_width is between 1.0 and 1.9
Rule(s) for Class p_2
 Rule #0
  sepal_length is between 6.2 and 7.9
  sepal_width is between 2.6 and 3.8
  petal_length is between 5.0 and 6.9
  petal_width is between 1.6 and 2.5
```

Figure 5: Output rules of an aspBEEF execution over the IRIS dataset

is instead able to turn on and off features for each of the clusters individually, depending on if those features provide valuable information or not. In an attempt to mitigate this effect, as was explained through this section, aspBEEF allows to manually select relevant features (fixed features) alongside the total number of features to use. Leaving a possible feature selection process up to the user.

5. Evaluation

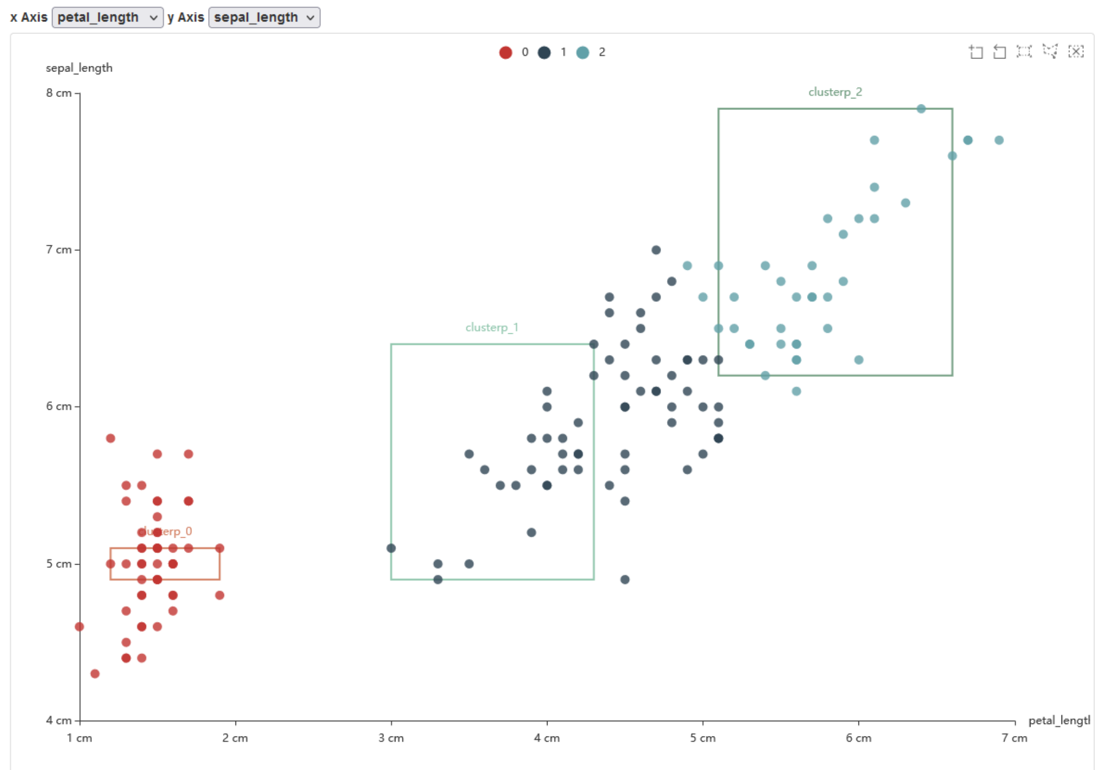
A short evaluation of the tool has been performed. The IRIS dataset, which is publicly available, has been used for the evaluation. In particular, the instance of the used IRIS data set is available in https://archive.ics.uci.edu/ml/datasets/iris For simplicity, the ground truth classes of the samples in the dataset were used to feed the algorithm, instead of using the outcome of a ML trained classifier. The dataset is composed of 150 samples alongside its ground truth class and 4 features: *sepal lenght*, *sepal width*, *petal length* and *petal width*.

Several tests were performed by using random samples of the dataset and used features to evaluate the influence of those factors in the computational cost of aspBEEF. Three random samples were extracted from the whole data set, each one with a different size: 60 samples, 90 samples and 150 samples. Tests were performed using 2, 3 and 4 features and both fixed and free feature selection methods were tested. Each test was performed 100 times and then averaged to smooth out the outlying values in the final measure. Both grounding and solving times were reported to see which of both require a bigger effort. Results are shown in Table 5

As can be seen in the table, solving time requires a bigger effort. On average, solving time represents 63.25% of the invested time taking all the tests into account. If analyzed by sample sizes, results show how solving time increases with sample sizes. Note 40.04% of average solving time for 60 sample size tests, 60.65% for 90 and 90.27% for 150. This makes sense since aspBEEF use values from the samples as possible thresholds, so as the number of samples increases, the number of possible boxes also increases exponentially.

However, that is not what increases the computational cost the most. When dealing with free features, increasing its number has a huge impact on the invested time. In general, if results for 2 and 4 free features are compared, a big time decrease is observed, a 57.68% for sample size 60 from (1.028 to 0.435), a 51.49% for 90 from (from 1.309 to 0.635) and a 63.47% for 150 (from 33.117 to 12.099). Given a total number of available features N and several free features to be selected f, the number of feature combinations to be tested by aspBEEF is $C_{N,f}$ (e.g combinations without repetitions of f elements within a set of N). This means that the more free features are used, the lower the number the combinations that have to be tested and hence the invested time, up to the minimal when $f = N$ and all the features are used.

Despite the long computational times when using free



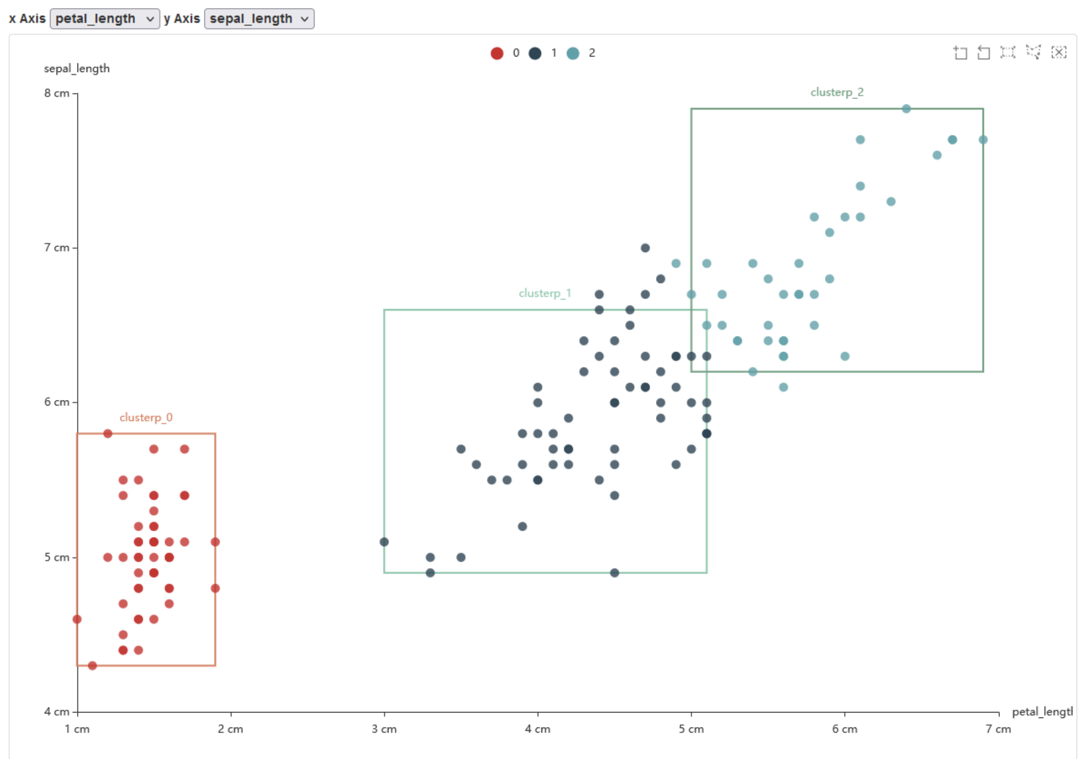


Figure 6: Visualizations of the `petal_length` and `sepal_length` features for the solutions 32 and 101 (final solution) of the search process for the whole IRIS Dataset

Sample Size	Used Features	Time w/ Free Features	Time w/ Fixed Features
60	2	1.028 (0.70)	0.471s (0.13)
60	3	0.736 (0.41)	0.560 (0.23)
60	4	0.435 (0.11)	0.400s (0.09)
90	2	1.309s (0.82)	1.269s (0.82)
90	3	3.558s (3.11)	1.402s (0.95)
90	4	0.635 (0.17)	1.134 (0.62)
150	2	33.117s (32.37)	7.282s (6.53)
150	3	24.029s (23.29)	4.255s (3.51)
150	4	12.099 (11.33)	3.979 (3.23)

Table 1
Times table for a data set of 150 points and 4 features. We provide total time (i.e. grounding and solving) and just the solving time alongside between parenthesis.

features, the automatic selection prunes the less useful features and so provides additional insight about the behavior of the model and also more concise explanations. Although optimizations are planned as future work, even large computational times could be worth it given that the algorithm is adding additional value by explaining already developed ML models, which usually do not have such a feature. In cases where the useful (or desired) features for explaining are known beforehand, then fixed features would be the recommended approach to prune the search space. When the opposite happens and some features are to be dispensed with, a mixed approach could also be useful. Undesired features should not be provided in the data to prevent `aspBEEF` to explore them, known useful features should be forced as fixed features for pruning the search space and finally the rest can be used as free features and let the algorithm decide if they are going to be included or not in the explanations.

6. Conclusions and Future Work

We have introduced the tool `aspBEEF` which obtains optimal BEEF explanations of machine learning classifiers by implementing the framework in ASP. The tool computes the optimal set of BEEF box-shaped clusters given a specific order over the quality measures (*overlapping*, *purity* and *inclusion*) and describes the found clusters. For doing so, it receives the predictions of a machine learning classifier (in *csv* format), the number of boxes to find, a set of fixed features and a set of free features. `asprin` is used for declaring the preferences and actually for performing the optimization process. The evaluation shows that the technique has a high computational cost, especially with the increase of free features. One option for tackling the problem would be to externally perform a feature selection process for fixing just the important features. Anyway, the effort is reasonable having in mind that our goal is not simple classification but obtaining rich explanations of the outcome of already trained ML classifiers, which rarely present such a feature.

As future work, we aim to modify the way `aspBEEF` handle features by allowing the activation of features to be box-dependent. For some cases, some features could be useful for explaining one class predictions but could be no more than noise for explaining the other. Currently, all classes have to be explained using the same features, since features are enabled or disabled globally for all boxes during optimization. By allowing the boxes to use different sets of features, those features which improve quality the least could be disregarded for some boxes, which would also simplify the explanations. In such a setting, each resulting box would only use the important features for explaining the predictions within it. This would be convenient from the perspective of an end-user since explanations would be more concise and differences between classes would be clearer rather than having. However, solutions whose boxes use more features would always have at least the same quality that equivalent ones that disregard some features, since having more features decrease the probability of having overlapping boxes. Hence, a new quality measure should be introduced for preferring those solutions whose boxes use fewer features. Furthermore, we aim to improve flexibility by adding an option to make `aspBEEF` find the best number of boxes in terms of quality. Usually using the same number of boxes as output classes works fine, but this has not always to be the case. The reasonable option with the current implementation is to externally perform an estimation with some already well-known techniques like the elbow method. The trivial option for introducing the feature in the tool would be to automatically perform this step before the clustering method.

Another option would be to add the number of boxes as a quality metric to be optimized. Since the optimal quality solution would be to trivially have a box around each prediction, a new simplicity quality measure probably would need to be introduced to encourage solutions with fewer boxes. Instead of receiving a fixed number, the tool could receive a threshold over the maximum number of boxes and find the optimal number in terms of quality but also simplicity.

The aspBEEF prototype is open source and already available at
https://github.com/trigork/aspBEEF.

Acknowledgments

This work has been partially supported by MINECO (grant TIN2017-84453-P) and Xunta de Galicia (grants GPC ED431B 2019/03 and ED431G 2019/01 for CITIC center).

References

[1] S. Grover, C. Pulice, G. I. Simari, V. S. Subrahmanian, BEEF: balanced english explanations of forecasts, IEEE Trans. Comput. Soc. Syst. 6 (2019) 350–364. URL: https://doi.org/10.1109/TCSS.2019.2902490. doi:10.1109/TCSS.2019.2902490.

[2] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (1999) 241–273.

[3] V. W. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: K. R. Apt, V. W. Marek, M. Truszczyński, D. Warren (Eds.), The Logic Programming Paradigm, Artificial Intelligence, Springer Berlin Heidelberg, 1999, pp. 375–398. URL: http://dx.doi.org/10.1007/978-3-642-60085-2_17. doi:10.1007/978-3-642-60085-2_17.

[4] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103. URL: http://doi.acm.org/10.1145/2043174.2043195. doi:10.1145/2043174.2043195.

[5] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: M. Carro, A. King (Eds.), Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16), volume 52 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:15.

[6] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own asp-based system?!, ArXiv abs/2008.06692 (2020).

[7] G. Brewka, J. P. Delgrande, J. Romero, T. Schaub, asprin: Customizing answer set preferences without a headache, in: AAAI, AAAI Press, 2015, pp. 1467–1474.