# PyIDS – Python Implementation of Interpretable Decision Sets Algorithm by Lakkaraju et al, 2016 *

Jiri Filip and Tomas Kliegr

Department of Information and Knowledge Engineering, Faculty of Informatics and Statistics, University of Economics, Prague {filj03,tomas.kliegr}@vse.cz

**Abstract.** Interpretable Decision Sets (IDS) by Lakkaraju et al, 2016 belongs to group of algorithms that perform classification based on association rules. Unlike most previous approaches, IDS provides means for balancing interpretability with prediction performance through user-set weights. Relying on submodular optimization, IDS is relatively computationally intensive. In this paper, we report on a new implementation of IDS, which is up to several orders of magnitude faster than the reference implementation released by Lakkaraju et al, 2016. The extensions to the reference implementation also include initial support for interoperability with other rule-based systems through the RuleML specification.

**Keywords:** Interpretable Decision Sets · Explainable Machine Learning · Classification · Rule Learning · Association Rules · Reproducibility · Replication · Benchmark



## 1 Introduction

"Black-box" models created by machine learning algorithms like deep neural networks provide state-of-the-art predictive performance. Owing to recent advances in machine learning, arbitrary models can also be explained using specialized postprocessing algorithms. However, some of the most popular model-agnostic explanation methods, such as LIME [11], have been reported to have multiple limitations including lacking understandability for humans and instability when used to explain a deep neural network [12]. Rule-based classifiers provide an alternative that has the potential to provide models that require no further explanation. Nevertheless, the comprehensibility of rule models is also not without caveats. For example, while individual rules may be well understandable, the complete rule model may lose its explainability if there are too many rules. Interpretable Decision Sets (IDS) [8] is a recently proposed rule learning algorithm that provides means for balancing model size, and also other facets of interpretability, with prediction performance through user-set weights.

* Supported by University of Economics, Prague by grant IGA 33/2018.

This paper is a first step of an on-going effort aimed at understanding and analysing strengths and weaknesses of the IDS algorithm. The motivation for creating a new implementation were some limitations of the reference implementation.

The pyIDS package covered in this paper implements the base IDS classifier as described in [8]. It also provides several enhancements, including a One-vs-All Classifier, a Coordinate Descent Optimizer (used to search for best IDS meta-parameters), a Random Search Optimizer, and a benchmarking module, which supports all interpretability metrics that the original article [8] defines. PyIDS is also accompanied with several Jupyter Notebooks, demonstrating different use cases of the software.

## 2 Background

IDS is a classification algorithm based on association rules. An association rule can be thought of as a set of conditions (antecedent), which – if met by a particular instance – predicts some class (consequent). Below is an example of several association rules describing how a default on a loan can be predicted from values of several attributes.

If age=(20,30) and family_status=married then default=no
If age=(30,40) and owns_car=yes then default=yes
If owns_car=yes then default=yes

Fig. 1: Example rules

IDS has multiple predecessor algorithms that applied association rule learning to the classification task [4, 9, 10, 14]. Among these, CBA (Classification Based On Associations) [9] is most widely used. These algorithms typically first mine large set of association rules using the Apriori algorithm [1] – or one of its successors – and then select a subset of these rules to form the classifier.

To select the subset of rules from the initial list, most association rule classification algorithms use heuristics. This results in inefficiencies, and also does not easily allow the user to balance predictive accuracy with the size of the model and other aspects of its interpretability.

IDS takes a different approach. By using submodular function optimization instead of a heuristic, it selects the subset of candidate rules for the final model. The "ideal" solution is a decision set, which reflects the following desiderata: small number of rules, rules with – on average – small number of conditions, the

`https://github.com/lvhimabindu/interpretable_decision_sets`
`https://github.com/jirifilip/pyIDS`

decision set correctly covers a very large portion of decision space, and small overlap between rules. These desiderata correspond to seven partial objectives, which are briefly and informally described below:

- $f_1$ – decreases with the number of rules,
- $f_2$ – decreases with the total number of conditions across all rules,
- $f_3$ – decreases with the number of overlaps (in terms of instances covered) between rules of the same class,
- $f_4$ – decreases with the number of overlaps (in terms of instances covered) between rules of different classes,
- $f_5$ – increases with the number of classes which are predicted by at least one rule,
- $f_6$ – decreases with the total number of incorrectly covered instances classified by individual rules,
- $f_7$ – increases with the number of data points which are correctly covered by at least one rule.

A linear combination of these seven objectives comprises the final objective function:

$$f = \sum_{i=1}^{7} \lambda_i f_i, \tag{1}$$

where $\lambda_1, \ldots, \lambda_7$ are externally set weights for the individual partial objectives. For precise specification of $f_1, \ldots, f_7$ please refer to Lakkaraju et al, 2016.

In IDS, the goal is to select a subset from the premined set of association rules, which maximizes the function $f$. For this purpose, a number of algorithms with different balance between speed and guarantees for quality of the solution in terms of distance from the global optimum can be used [5]. For IDS, Lakkaraju [8] proposes to use the Smooth Local Search (SLS) optimization algorithm [5]. SLS is guaranteed to find a solution, which is at least $\frac{2}{5}$ of the optimum value [5].

## 3 Implementation

This section is structured by the phases of the IDS algorithm. First, frequent itemsets are extracted from data. From these, candidate rules are formed. The third phase – selection of rules from these candidates – is the core of IDS. The last phase is application of the model on unseen instances. These three phases can be repeated many items within the metaparameter tuning phase, which aims to set the $\lambda$ weights from Eq. 1. Once the final model has been learnt, the last step is to ensure its interoperability with other rule systems. The final subsection reports on our attempt to check pyIDS against the reference IDS implementation.

Note that our pyIDS implementation described in the following reuses some components from the pyARC package [6], which implements the CBA algorithm.

Note that IDS subtracts some of these criteria from a fixed constant to make them maximizable.

### 3.1 Generation of frequent itemsets

The first step of the IDS algorithm is generation of frequent itemsets that meet a user-specified support threshold.

The pyIDS implementation uses the high performing FIM package [3], which implements the Apriori [1] algorithm and is used for frequent itemset mining in a number of other machine learning libraries.

### 3.2 Generation of rules

IDS has a somewhat different approach to generating rules from frequent itemsets than some other association rule classification algorithms, such as CBA. While CBA generates only rules meeting a user-specified confidence threshold, IDS does not use the confidence threshold: for each input frequent itemset, IDS generates as many class rules as there are classes. The advantage of this approach is that it creates more candidates for selection in the optimization phase. The disadvantage is that it can be resource intensive.

In addition to rule generation without the confidence threshold as described in [8], pyIDS allows to apply the rule generation approach used in CBA, which involves the minimum confidence threshold. Additionally, for use cases when mining time and memory constraints are of priority, pyIDS integrates the heuristic tuning algorithm proposed in [7], which is reimplemented in the pyARC package. This algorithm iteratively changes confidence, support and maximum rule length constrains, until a user-specified number of rules is generated. Generated rules are passed directly to the optimization phase of IDS.

### 3.3 Rule selection

The essence of the selection process is as follows. All rules obtained in the previous step are used as candidates from which a subset is selected to the solution set. The solution set is created iteratively, with rules being added or removed across iterations based on the result of evaluation of the objective function (Eq. 1) on the interim solution set. The pyIDS package implements two variants of the rule selection algorithm.

*Smooth Local Search* The Smooth Local Search (SLS) algorithm is considered as the best algorithm for submodular optimization and as such is adopted in IDS [8]. Please refer to [5, 8] for further details on the SLS algorithm.

*Deterministic Local Search* From the implementation perspective, SLS introduces an element of randomness, which can be sometimes a disadvantage. In our experience, subsequent runs of the SLS algorithm can return quite different solution sets. This property of the SLS algorithm may complicate optimization of $\lambda$ metaparameters. As an alternative, the pyIDS package contains also Deterministic Local Search (DLS), which is also available in the reference implementation of IDS.

*Caching* Caching takes advantage of the observation that the same rule can be present in many candidate solutions. The reference implementation by Lakkaraju et al, 2016 did not support caching, which had adverse affect on computation time. The pyIDS implementation precomputes and caches rule cover and rule overlap.

The cache size grows quadratically with the number of input rules for the rule overlap partial objective function, and linearly for the rule cover partial objective function. For low memory setups pyIDS thus allows to disable caching.

### 3.4 Model application

The original reference implementation does not contain any specific function for model application (prediction). The pyIDS package implements prediction according to the description in Lakkaraju et al, 2016. From this it follows that rules are to be sorted according to the F1 metric and the first rule (in the order of F1 metric) matching the test instance is used to assign the class to the test instance. If no matching rule is found, a majority class in the training data is assigned.

As an alternative to the prediction process described in [8], pyIDS supports a second option, which is inspired by CBA. The default class is computed from instances, which are not covered by any of the rules in the final model (rather than from all training data).

### 3.5 Optimization of Lambda metaparameters

The $\lambda$ parameters allow the user to set the accuracy-interpretability tradeoff. There are two principle ways in which $\lambda$ parameters can be set:

1. user sets custom weights for each of the seven partial objective functions,
2. weights are optimized to maximize AUC (Area Under Curve) on a validation dataset.

Since the individual partial objectives have a different scale, the $\lambda$ parameters do not directly correspond to values of the interpretability metrics. This complicates setting of the weights by the user. While this problem could be possibly addressed by normalization of the partial objectives, this is not covered in the original approach by Lakkaraju et al, 2016.

The second option amounts to imposing specific requirements on the values of interpretability metrics through AUC optimization, as also described in the the original paper introducing IDS [8]. Here, the user sets the limits for the individual metrics directly. An example is a user-set limit on maximum 100 rules in the model. The metaparameters are then optimized so that AUC is maximized. Any combination of $\lambda$ values resulting in solution not meeting the user-set requirements (e.g. resulting in a rule set with more than 100 rules) is discarded.

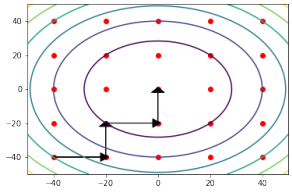



Fig. 2: Coordinate Ascent

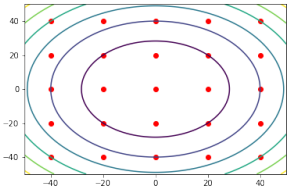



Fig. 3: Grid Search

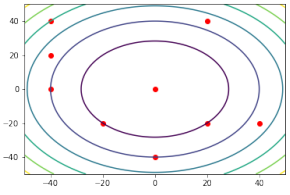



Fig. 4: Random Search

PyIDS contains implementation of the coordinate ascent optimization algorithm as proposed for tuning IDS metaparameters in [8]. In addition, our implementation also offers grid search and random search. Before all these standard optimization algorithms are described, we provide some details on the way AUC of an IDS model is computed in our implementation.

*Measuring interpretability* The user requirements on model interpretability are set through the following measures:

1. **Average rule length:** average number of conditions in the antecedent.
2. **Fraction classes:** proportion of classes that are covered by the model.
3. **Fraction overlap:** to what degree the rules in the final model overlap. Values range between 0 and 1, with 0 being the best.
4. **Fraction uncovered:** how many instances in the data set are not covered by the rules in the model. The value ranges between 0 and 1, with 0 being the best.
5. **Ruleset length:** rule count in the model.

The same measures are used to evaluate model interpretability in Lakkaraju et al, 2016:

*Computation of AUC* AUC is calculated with probabilities that are equal to confidence of the rule that classified the instance. This is the same approach as adopted by Lakkaraju et al, 2016 [8]. As an alternative option, the pyIDS implementation allows to compute AUC based on a match between prediction and true label. If the predicted label matches, the probability for the given instance is set to 1.0, if the instance is misclassified, the probability is set to 0.

*Coordinate ascent optimization* Informally, coordinate ascent can be thought of as an alternative of gradient descent. Instead of minimization – as in gradient descent – we are maximizing the model's AUC score and – more importantly – we are increasing or decreasing only one coordinate (parameter). This is different than in gradient descent, where steps are taken along the direction of the derivative. An illustration of the main idea of coordinate ascent is present in Figure 2. Only one parameter/direction is changed at a time.

*Grid search* An illustration of this standard optimization algorithm can be seen in Figure 3. In order to find the function's maximum value, the model is evaluated at every combination of the parameters. The default parameter grid is depicted in Figure 5.

| | | | | | |
|---|---|---|---|---|---|
| $\lambda_0$ | 0 | 20 | 40 | 60 | 80 |
| $\lambda_1$ | 0.5 | 5 | 50 | 500 | 5000 |
| ... | ... | ... | ... | ... | ... |
| $\lambda_6$ | 0.3 | 542 | 1000 | 0.92 | 23 |

Fig. 5: Parameter grid for $\lambda$ optimization

*Random search* In this optimization approach, a sample of the parameter grid is chosen at random. For each setting, the AUC is calculated. The setting with the greatest AUC associated with it is then chosen. For pyARC, a NumPy [13] random generator is used.

### 3.6 Export to RuleML

To enhance interopertability with other systems for processing rules, the pyIDS implementation supports export to the RuleML format [2]. One example exported rule is present in Figure 6. Rule quality statistics, such as confidence and support are not currently exported.

### 3.7 Correctness of our implementation

To check that that our pyIDS implementation adheres to the original paper by Lakkaraju et al, 2016, we use the Deterministic Local Search (DLS) version of the rule selection step in IDS. The comparison was performed on the well-known Titanic dataset. Both implementations provided the same output solution set.

The script containing the comparison test can be downloaded from pyIDS repository. The script also contains a set of functions, which convert data structures from the reference implementation to data structures of pyIDS.

The alternative Smooth Local Search (SLS) version contains a random element, which makes comparison of two implementations difficult.

`https://github.com/jirifilip/pyIDS/blob/master/scripts/other/reference_implementation_testing.ipynb`

```
<Implies>
    <head>
        <Atom>
            <Var>class</Var>
            <Rel>Iris-setosa</Rel>
        </Atom>
    </head>
    <body>
        <Atom>
            <Var>name</Var>
            <Rel>sepallength</Rel>
        </Atom>
        <Atom>
            <Var>value</Var>
            <Rel>-inf_to_5.55</Rel>
        </Atom>
        <Atom>
            <Var>name</Var>
            <Rel>sepalwidth</Rel>
        </Atom>
        <Atom>
            <Var>value</Var>
            <Rel>3.35_to_inf</Rel>
        </Atom>
        <Atom>
            <Var>name</Var>
            <Rel>petalwidth</Rel>
        </Atom>
        <Atom>
            <Var>value</Var>
            <Rel>-inf_to_0.8</Rel>
        </Atom>
    </body>
</Implies>
```

Fig. 6: Export to RuleML

While we obtained the same result on one particular problem, this does not guarantee perfect match between pyIDS and the reference implementation, or even with the description of IDS as intended by its authors in [8]. Also note that the package is still in the process of development.

## 4 Benchmarking

The initial objective of our work was to provide an implementation that is faster than the reference one provided by Lakkaraju et al, 2016. In this section, we evaluate the results of our effort on two benchmarks between pyIDS and the reference implementation.

### 4.1 Datasets and Setup

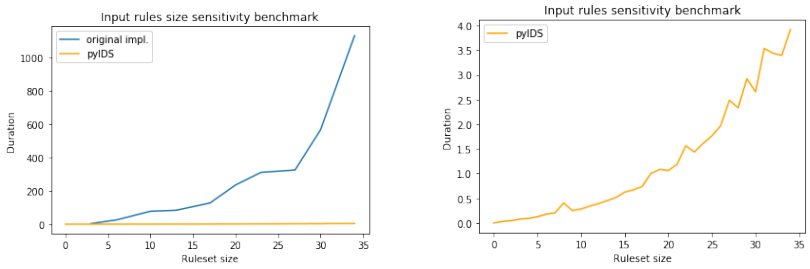
Since datasets originally used for evaluation in [8] are not publicly available, we used the iris dataset, which can be retrieved from the UCI repository. We also tried to use other open datasets of similar size to those used in [8], but the computational cost was prohibitive for both pyIDS and the reference implementation, probably due to many expensive evaluations of the objective function.

For rule generation, we used the heuristic tuning algorithm, described in Section 3.2. The $\lambda$ metaparameters were left at their defaults, as their values were only marginally relevant for the performance benchmark.

### 4.2 Rule count sensitivity

In this benchmark, the number of rules with which we train the IDS model is increased iteratively and the dataset size is held constant. As can be seen from

Figure 7, pyIDS is consistently faster than the original reference implementation. For example, pyIDS takes approximately 4 seconds to train a model with 35 input rules, which is several orders of magnitude faster than the reference implementation.



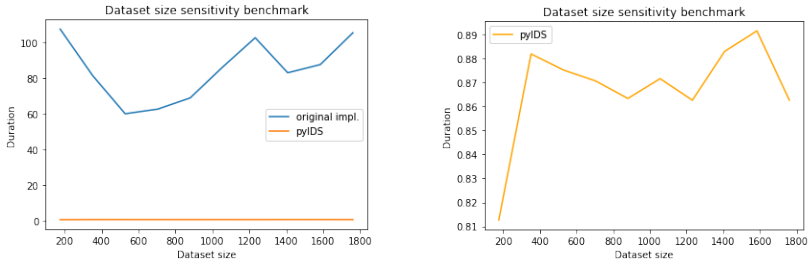

(a) Both implementations (b) pyIDS only

Fig. 7: Data size sensitivity benchmark

### 4.3 Data size sensitivity

In this benchmark, we vary dataset size, keeping constant the number of rules on IDS input.

As can be seen from Figure 8, the pyIDS implementation is again at least an order of magnitude faster than the reference implementation. Somewhat surprisingly, the runtime of IDS does not appear to depend on the dataset size. We attribute this to caching.




(a) Both implementations (b) pyIDS only

Fig. 8: Data size sensitivity benchmark

## 5 Conclusion

The presented reimplementation of the Interpretable Decision Sets algorithm by Lakkaraju et al, 2016 provides an alternative to the reference implementation provided by the original authors. Our pyIDS implementation is faster, contains more options for tuning the metaparameters, and supports some vital functionality omitted from the reference implementation, such as the ability to apply the learnt model.

. While pyIDS performed in our benchmarks up to several orders of magnitude faster than the original reference implementation, we have to note that it does not scale to some larger datasets and larger input rule sets. This is contrary to expectations that would follow from experiments reported in [8]. Future work can use the pyIDS implementation to investigate the causes of this discrepancy. The software is available under MIT license at `https://github.com/jirifilip/pyIDS`.

## References


1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) Proc. 20th Int. Conf. Very Large Data Bases, VLDB. pp. 487–499. Morgan Kaufmann (12–15 September 1994)
2. Boley, H., Zou, G.: Perspectival Knowledge in PSOA RuleML: Representation, Model Theory, and Translation. CoRR **abs/1712.02869, v3** (2019), `http://arxiv.org/abs/1712.02869`
3. Borgelt, C.: Efficient implementations of Apriori and Eclat. In: FIMI03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations (2003)
4. Elkano, M., Galar, M., Sanz, J.A., Fernndez, A., Barrenechea, E., Herrera, F., Bustince, H.: Enhancing multiclass classification in FARC-HD fuzzy classifier: On the synergy between $n$-dimensional overlap functions and decomposition strategies. IEEE Transactions on Fuzzy Systems **23**(5), 1562–1580 (Oct 2015). https://doi.org/10.1109/TFUZZ.2014.2370677
5. Feige, U., Mirrokni, V.S., Vondrák, J.: Maximizing non-monotone submodular functions. SIAM Journal on Computing **40**(4), 1133–1153 (2011)
6. Filip, J., Kliegr, T.: Classification based on associations (CBA)-a performance analysis. In: RuleML+RR (Supplement) (2018)
7. Kliegr, T., Kuchař, J.: Tuning hyperparameters of classification based on associations (CBA). In: Proceedings of ITAT 2019. CEUR-WS (2019), to appear
8. Lakkaraju, H., Bach, S.H., Leskovec, J.: Interpretable decision sets: A joint framework for description and prediction. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 1675–1684. ACM (2016)
9. Liu, B., Hsu, W., Ma, Y.: Integrating classification and association rule mining. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. pp. 80–86. KDD'98, AAAI Press (1998)
10. Pei, W.L.J.H.J., et al.: Cmar: Accurate and efficient classification based on multiple class-association rules. In: Proceedings of IEEE-ICDM. pp. 369–376 (2001)

11. Ribeiro, M.T., Singh, S., Guestrin, C.: Why should I trust you?: Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 1135–1144. ACM (2016)
12. Stiffler, M., Hudler, A., Lee, E., Braines, D., Mott, D., Harborne, D.: An analysis of reliability using lime with deep learning models. In: Annual Fall Meeting of the Distributed Analytics and Information Science International Technology Alliance, AFM DAIS ITA (2018)
13. Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering **13**(2), 22 (2011)
14. Yin, X., Han, J.: CPAR: Classification based on predictive association rules. In: Proceedings of the 2003 SIAM International Conference on Data Mining. pp. 331–335. SIAM (2003)