JOHANNES KEPLER
UNIVERSITÄT LINZ

Practical Work in AI

Working with
paper "Computing Optimal Decision Sets with SAT"
chapter "4.1 Iterative SAT Model"
(Yu et al., 2020)

Table of contents

1. Topic Overview

Look at underlying paper "Computing Optimal Decision Sets with SAT" (Yu et al., 2020)

- Paper is trying to create classifiers with help of logical constraints
- For this purpose create a rule/decision set which contains several rules which decide the class of a sample
- Each rule consists of nodes
- Each node **j** carries information about the used binary feature **r** and its truth value **t**=0/1

  node representation: s_jr

Look at underlying paper "Computing Optimal Decision Sets with SAT" (Yu et al., 2020)

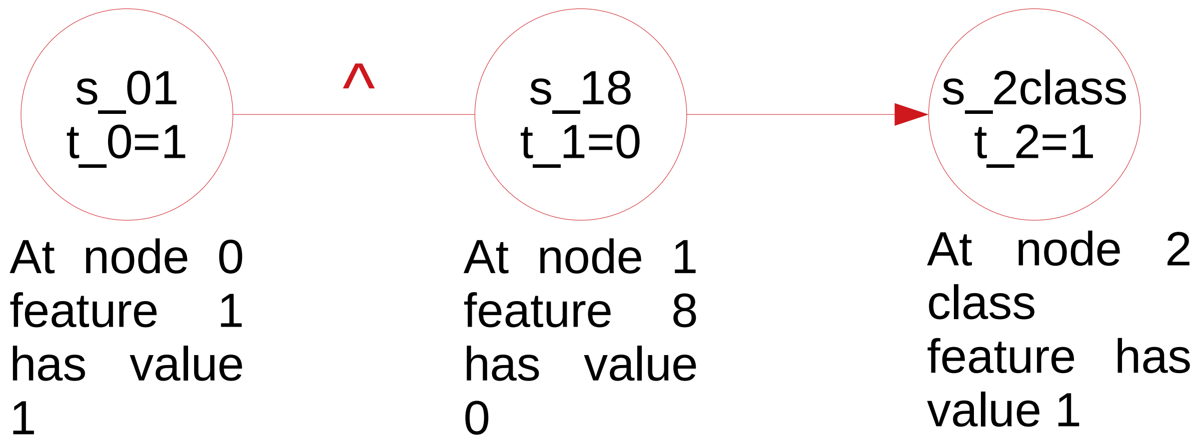
- If one sample **i** fulfills all rule nodes' conditions (with exception of last one), the last node of the rule predicts the class of that sample through its truth value **t**=0(-class)/1(+ class)

  Representation when sample **i** fits to node **j**:

  v_ij
- When sample fulfills several rules majority vote leads to final class prediction

Look at underlying paper "Computing Optimal Decision Sets with SAT" (Yu et al., 2020)

- e. g. rule:



Sample with feature1=1 AND feature8=0 fulfills rule's conditions
→ sample class prediction=1

Aim of the practical work

- Understanding the paper's ideas
- Implementation of the constraints from the chapter **"4.1 Iterative SAT Model"**
- Test them on several datasets, especially the Mushroom Dataset
- Compare the implementation to another rule-based classification algorithm (well performing RIPPER chosen)

2. Class DecisionSetClassifier

2.1. Structure/methods

methods

1) fit(data, path_to_kissat_solver)
2) predict(data, y_test=None)
3) score(y_test)
4) Ruleset_()
5) ruleset_performance()

fit(data, path_to_kissat_solver)

- Implementation of constraints and building a decision set
- Arguments:

  → data: numpy array with the y-column

  → path_to_kissat_solver: string path leading to the KISSAT storage place
- Output:

  → 1: decision set successfully built

  → 0: no model for the given number of nodes found

  → -1: computations were too complex for the system

predict(data, y_test=None)

- Predict the y-values for given data by using the in fit() found decision set. Majority vote decides the class prediction of a sample
- Input:

  data: numpy array without label-column

  y_test: true y-labels to compare them with predictions,

  for evaluation purpose needed
- Output:

  an array that contains a prediction for each inputted sample

score(y_test)

- Precondition: firstly, execute predict()
- Input: y_test: true data labels
- Output: accuracy

ruleset_()

- Only visualization purpose. Uses the in fit() found decision set to show output as a nicer visualization in form of a string as output
- e. g. brings '¬' and '∧' into game
- However, use ruleset_performance() instead

ruleset_performance()

- Firstly, run fit() and predict()
- Outputs a table with columns:

1) 'rules'

   gained from ruleset_(), one rule per row

2) '# test samples fulfilling rule conditions'

   how many samples fit to a certain rule

3) '# mistakenly fitting test samples'

   how many samples accidently fit to a rule

4) '% mistaken fits'

   3)/2)

2.2. fit method

"4.1 Iterative SAT Model" constraint implementation

Logical constraints:

(1) Each node carries exactly one feature

$$\forall_{j \in [N]} \sum_{r=1}^{K+1} s_{jr} = 1$$

(2) The final node is a leaf

$$s_{Nc}$$

(3) At the first node all samples are valid

$$\forall_{i \in [M]} \, v_{i1}$$

"4.1 Iterative SAT Model" constraint implementation

Logical constraints:

- (4) Each sample is valid at the first node of a rule or when the previous node's feature value fits to the feature value of the sample

$$\forall_{i \in [M]} \forall_{j \in [N-1]} \, v_{ij+1} \leftrightarrow s_{jc} \vee \left(v_{ij} \wedge \bigvee_{r \in [K]} \left(s_{jr} \wedge \left(t_j = \pi_i[r]\right)\right)\right)$$

- (5) If a training sample fulfills rule's conditions the final leaf shall predict the correct label

$$\forall_{i \in [M]} \forall_{j \in [N]} \left(s_{jc} \wedge v_{ij}\right) \rightarrow \left(t_j = c_i\right)$$

"4.1 Iterative SAT Model" constraint implementation

Logical constraints:

(6) Each training sample has to fully fit to at least one rule

$$\forall_{i \in [M]} \bigvee_{j \in [N]} (s_{jc} \wedge v_{ij})$$

"4.1 Iterative SAT Model" constraint implementation

Implementation boolean variables:

- s_jr → s[j,r] (paper j>=1, implementation j>=0)
- v_ij → v[i,j]
- t_j → t[j]
- π_i [r] → 0/1 (describes the feature value of sample **i** in feature **r**

"4.1 Iterative SAT Model" constraint implementation

Because of the universal quantifier ∀, the implementation consists of for-loops.

Problems:

- for (4) three loops are nested in each other
- Depending on the number of training samples the loop output can become very long

Complexity issues. However, the author Alexey Ignatiev verified that the for-loops are unavoidable.

"4.1 Iterative SAT Model" classifier implementation

- Implement the constraints with help of for-loops. Formulate them as strings.
- Concatenate the strings by conjunction '&'

"4.1 Iterative SAT Model" classifier implementation

- Package pyeda:
  - convert the string to an expression
  - Transform the expression into CNF using tseitin transformation such that exponential overhead can be avoided

    ```
    a = string.tseitin()
    ```

  - Transform the CNF into DIMACS format which can be read by SAT solvers

    ```
    b = pyeda.boolalg.expr.expr2dimacscnf(a)[1]
    ```

"4.1 Iterative SAT Model" classifier implementation

- Store the DIMACS formula in a file such that the classifier can read the formula

```
file = open('my_cnf.cnf', 'w')
file.write(f'{b}')
file.close()
```

"4.1 Iterative SAT Model" classifier implementation

- Use the KISSAT solver of Prof. Dr. Armin Biere which is known to be fast and quite new

  (https://github.com/arminbiere/kissat)

  ! '{path_including_kissat}' my_cnf.cnf

  (!: a terminal execution within a jupyter notebook)

- Convert DIMACS solution back to our variables: numeric_ID → s[j,r]; …

  with help of the mapping dictionary:

  pyeda.boolalg.expr.expr2dimacscnf(a)[0]

"4.1 Iterative SAT Model" classifier implementation

- Alternative solver: PICOSAT. It is already included in the pyeda package. So, nothing has to be installed. However, it is slower and older.

  Relating code in notebook under

  '# When using PICOSAT'

"4.1 Iterative SAT Model" classifier implementation

- Transfer all used feature indices **r** into one array, e. g.

  [1, 9, class_index, 3, class_index]

  → Having 2 rules.

  First one makes use of features 1, 9. The second of 3.

- For t_j create an zero_array and put 1 into every indexed **j** place where t_j=1, e. g.

  [0, 0, 1]

  → for node 2 we have t_2=1

3. Data sets and Results

Preconditions data sets

- Need to have binary labels
- Need to be binary in general.

  Also discrete datasets possible when we transfer it into a binary one (see mushroom dataset)

- All used datasets already contained in the zip-folder

Data set from underlying paper

	Item No.	1	2	3	4	5	6	7	8
Features	0 L	1	1	0	1	0	1	0	0
	1 C	0	0	0	1	0	1	1	0
	2 E	1	0	1	0	0	1	1	1
	3								
	4 S	0	1	0	0	1	1	0	1
	Class H	0	0	1	0	1	0	0	1

Fit() and predict() get full dataset to check if our decision set classifier learnt the same rules as stated in the paper:

$$L \Rightarrow \neg H$$ 0 → ¬class

$$\neg L \land \neg C \Rightarrow H$$ ¬0 ∧ ¬1 → class

$$C \Rightarrow \neg H$$ 1 → ¬class

My result using ruleset_performance():

rules	# test samples fulfilling rule conditions	# mistakenly fitting test samples	% mistaken fits
0 → ¬class	4	0	0.0
¬0 ∧ ¬1 → class	3	0	0.0
1 → ¬class	3	0	0.0

Data set from underlying paper

Comparing with RIPPER:

using my function tableizer(...):

best accuracy scores for dataset from underlying paper

dataset forms	our decsision set classifier	RIPPER
full dataset of shape (8, 5)	1.0	0.875

Data handling with functions

- Create a table comparing RIPPER with our classifier in accuracy:

  tableizer(current_dataset_name, *row_values_lists)

- Working with Stratified Cross Validation with number of folds k=5 like in the underlying paper

  stratified_cross_validation(dataset, clf_ripper_dsc, number_folds, path_tokissat_solver)

  - No need of calling class DecisionSetClassifier directly
  - Usable for RIPPER

- Data cleaning: Remove duplicates and samples with same feature values but different classes:

  remove_duplicates(data_array)

Binary data set from research gate

- Binary data set
- Reason of taking: it fulfills all preconditions

- Retrieved from

  https://www.researchgate.net/post/A_dataset_with_binary_data_for_a_two-class_classification_problem_How_to_decide_if_it_is_linear_or_non-linear_How_to_choose_a_good_classifier

Binary data set from research gate

Results:

data set forms	our decsision set classifier	RIPPER
full data set of shape (250, 751); #folds=5	computation too complex	0.62
shortened data set of shape (59, 11); #folds=5	None with 28 nodes	0.6666666666666666

Problems:

- The decision set classifier had problems of computing such a high data amount (see for-loop problem)
- 28 rule nodes don't seem to be enough. However, with higher node number we run into complexity issues.

Recruitment data

- Contains information about job success and education
- Reason of taking: we can make use of some binary features to predict job status

- Retrieved from

  https://www.kaggle.com/benroshan/factors-affecting-campus-placement

Recruitment data

Results:

data set forms	our decsision set classifier	RIPPER
full data set of shape (41, 8); #folds=5	None with 29 nodes	0.75
shortened data set of shape (18, 6); #folds=5	0.6666666666666666 with 20 nodes	0.75

- Here problem of producing rules with more than 29 nodes (complexity issues)
- Our classifier was able to generate rules for shortened dataset. However, the test dataset was also very small here.
- RIPPER outperforms our decision set classifier

Mushroom data set

- Use different properties of mushrooms to decide if they are ediable/poisonous
- Reason of taking:
  - it has discrete features but they can be transformed into binary ones
  - binary class
  - famous dataset to work with

- Retrieved from

  https://archive.ics.uci.edu/ml/datasets/Mushroom

Mushroom data set

Results:

data set forms	our decision set classifier	RIPPER
full data set of shape (8124, 118); #folds=5	problem too complex with 20 nodes	1.0
shortened data set of shape (33, 18); #folds=5	0.7142857142857143 with 25 nodes	0.7142857142857143
shortened binary data set of shape (23, 12); #folds=5	0.6 with 33 nodes	1.0
shortened discrete data set of shape (130, 52); #folds=5	1.0 with 33 nodes	1.0

- complexity issues for full dataset
- For shortened datasets our classifier performs quite good but still worse than RIPPER
- The two latter datasets are only considering features which were used by RIPPER for full dataset. Penultimate: binary features, last: discrete features

4. Conclusion

- Like in the paper RIPPER outperforms our Decisionset classifier 'opt'
- opt has problems with the scalability. Nevertheless, one of the authors stated that the for-loops are a must have.

  Idea: using cloud computing service to get more computational power.

  However: I tried out the service 'aws' 'SageMaker' of 'amazon'. No access to high computational power because of charges.