Lazy binary classifier based on Formal Concept Analysis

Jacob Malyshev

Model

We represent our data sets as ordered sets $K = (C, M, I)$, where C - set of objects, M - set of attributes and I - set of relations between C and M. We can write I in the following way:

$$I = \{(g, m) | g \in C, m \in M, gIm\}$$

. There is a target attribute in M set (m_t). To classify new object $g \in C_\tau$ (there is no information about relation of this object with m_t attribute) we use the following procedure:

1. We split our train data into two parts: C_+ - context with '+' examples, C_- - context with '-' examples.

$$C_+ = \{g \in C | (g, m_t) \in I\}$$
$$C_- = \{g \in C | (g, m_t) \notin I\}$$

2. Take $g_i \in C_\tau$ and paste it to our classifier

3. Select $g_i^+ \in C_+$ and calculate intersection of its description with the description g_τ, that is
$$\Delta_i^+ = (g_i^+)' \cap (g_\tau)'$$

4. Calculate the number of object in the set C_- for which the resulting intersection is included in the description of these objects.
$$N_i^+ = |\{g^- \in C_- | \Delta_i^+ \subseteq (g^-)')\}|$$

5. If N^+ doesn't exceed some threshold T, then we take this object into account and increase counter by one $NB^+ = NB^+ + 1$

6. Similarly, support for a negative decision is calculated: NB^-

7. Finally, our classifier predicts class of our new example using the following rule:

$$Class = \begin{cases} g_\tau \in C_+, & \text{if } NB^+ > NB^- \\ g_\tau \in C_-, & \text{if } NB^+ < NB^- \end{cases}$$

There is also situation, when $NB^+ = NB^-$. We have several solutions:

- Choose positive class
- Choose negative class
- Choose class randomly

In this project, we use first solution.

Some modifications

To improve quality of our algorithm we can add some modifications:

- Adding threshold parameter T_2, that defines min number of elements, that must be in intersection Δ_i^+ (must improve time of algorithm)
- Scaling NB^+ and NB^- by proportion of elements in target class (must imporve quality for unbalanced data)
- Use only a part of train data (must improve time, more suitable for big datasets)

Datasets

To test our algorithms we use two datasets:

- Tic-Tac-Toe End game Dataset UCI (https://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe+Endgame)

  This database encodes the complete set of possible board configurations at the end of tic-tac-toe games, where "x" is assumed to have played first. The target concept is "win for x" (i.e., true when "x" has one of 8 possible ways to create a "three-in-a-row").

  Number of Instances: 958 (legal tic-tac-toe endgame boards)

  Number of Attributes: 9, each corresponding to one tic-tac-toe square

  Missing Attribute Values: None

Class Distribution: About 65.3% are positive (i.e., wins for "x")

Attribute Information:

- V1 = top-left-square: x,o,b
- V2 = top-middle-square: x,o,b
- V3 = top-right-square: x,o,b
- V4 = middle-left-square: x,o,b
- V5 = middle-middle-square: x,o,b
- V6 = middle-right-square: x,o,b
- V7 = bottom-left-square: x,o,b
- V8 = bottom-middle-square: x,o,b
- V = bottom-right-square: x,o,b
- V10 = Class: positive,negative

To work with this data we have to preprocess it with binarization technique

- Mushroom Classification (https://archive.ics.uci.edu/ml/datasets/mushroom)

  This dataset includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family Mushroom drawn from The Audubon Society Field Guide to North American Mushrooms (1981). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy.

  Number of Instances: 8124

  Number of Attributes: 23

  Missing Attribute Values: None

  Class Distribution: About 51.8% are positive (i.e., 'e')

  Attribute Information:

  - cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
  - cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
  - cap-color: brown=n,buff=b,cinnamon=c,gray=g,
    green=r,pink=p,purple=u,red=e,white=w,yellow=y

- bruises: bruises=t,no=f
- odor: almond=a,anise=l,creosote=c,fishy=y,
  foul=f,musty=m,none=n,pungent=p,spicy=s
- gill-attachment: attached=a,descending=d,free=f,notched=n
- gill-spacing: close=c,crowded=w,distant=d
- gill-size: broad=b,narrow=n
- gill-color: black=k,brown=n,buff=b,chocolate=h,gray=g,
  green=r,orange=o,pink=p,purple=u,red=e,white=w,yellow=y
- stalk-shape: enlarging=e,tapering=t
- stalk-root: bulbous=b,club=c,cup=u,equal=e,rhizomorphs=z,rooted=r,missing=?
- stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
- stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
- stalk-color-above-ring:
  brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
- stalk-color-below-ring:
  brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
- veil-type: partial=p,universal=u
- veil-color: brown=n,orange=o,white=w,yellow=y
- ring-number: none=n,one=o,two=t
- ring-type: cobwebby=c,evanescent=e,flaring=f,
  large=l,none=n,pendant=p,sheathing=s,zone=z
- spore-print-color:
  black=k,brown=n,buff=b,chocolate=h,green=r,orange=o,purple=u,white=w,yellow=y
- population: abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y
- habitat: grasses=g,leaves=l,meadows=m,paths=p,urban=u,waste=w,woods=d

It worths to say, that we test our algorithms we use only 1000 entities for time reasons.

Python implementation

To realize our algorithms we use Python. We create Python module **LazyFCA.py** with the following functions:

1. **preprocessing** - binarize our data:
   input: df - our data, target_column - name of target column, target_dict - dict, which defines what class is 1 and what is 0, shuffle - bool, if True - shuffle data, otherwise shuflle is not need
   output: df2 - binarized dataset

2. **LazyFCAclf** - main algorithm, which makes a prediction
   input : C_plus - C_+, C_minus - C_-, new_example - g_τ, max_int - L, min_elems - L_2, balance - scaling or not, prop - proportion of train data in use.
   output: class prediction

3. **Predict** - build predictions for test data **input**: data_dict - dict with C_+, C_-, C_τ, max_int, min_elems, balance, prop
   output: Y_pred - list of predictions

4. **cross_validation** - estimate quality of our model using accuracy, precision, recall and ROC AUC with KFolds cross-validtation
   input - df - preprocessed data, target_column, Kfolds - number of folds, shuffle, model - name of model in use, model_params - parameters of model
   output: dict of metrics

Results

Here I present results of 4 algorithms for two datasets:

1. Base algorithms with only one parameter L You can see results for Tic-Tac-Toe End game Dataset UCI in table 1 and for Mushroom Classification in table 2

2. Base algorithm plus L_2 parameter You can see results for Tic-Tac-Toe End game Dataset UCI in table 3 and for Mushroom Classification in table 4

3. Second algorithm plus scaling You can see results for Tic-Tac-Toe End game Dataset UCI in table 5 and for Mushroom Classification in table 6

4. Third algorithm plus random partition of data You can see results for Tic-Tac-Toe End game Dataset UCI in table 7 and for Mushroom Classification in table 8

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.699060	0.676768	1.000000	0.593220	04:08
1	1	0.777429	0.749104	0.995238	0.676518	04:16
2	2	0.787500	0.763441	0.990698	0.681063	04:17

Table 1: Base algorithm for Tic-Tac-Toe End game Dataset UCI

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.990991	0.982249	1.0	0.991018	22:46
1	1	1.000000	1.000000	1.0	1.000000	21:56
2	2	1.000000	1.000000	1.0	1.000000	23:16

Table 2: Base algorithm for Mushroom Classification

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.739812	0.705674	1.0	0.654167	01:21
1	1	0.824451	0.796364	1.0	0.720000	01:26
2	2	0.765625	0.734982	1.0	0.665179	01:19

Table 3: Second algorithm for Tic-Tac-Toe End game Dataset UCI

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.981982	0.965318	1.0	0.981928	03:11
1	1	0.981982	0.967213	1.0	0.980769	03:02
2	2	0.982036	0.966851	1.0	0.981132	03:04

Table 4: Second algorithm for Mushroom Classification

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.849530	0.890547	0.873171	0.840094	01:24
1	1	0.833856	0.870968	0.883178	0.808255	01:35
2	2	0.896875	0.922330	0.917874	0.888141	01:22

Table 5: Third algorithm for Tic-Tac-Toe End game Dataset UCI

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.993994	0.988764	1.000000	0.993631	03:18
1	1	0.987988	0.984043	0.994624	0.987108	03:15
2	2	0.964072	0.934132	0.993631	0.965742	03:12

Table 6: Third algorithm for Mushroom Classification

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.768025	0.855721	0.792627	0.754157	00:23
1	1	0.833856	0.911111	0.815920	0.840164	00:20
2	2	0.762500	0.823529	0.807692	0.743132	00:21

Table 7: Fourth algorithm for Tic-Tac-Toe End game Dataset UCI

	index	accuracy	precision	recall	ROC_AUC	Time
0	0	0.975976	0.961538	0.994318	0.974866	00:52
1	1	0.972973	0.965714	0.982558	0.972646	00:49
2	2	0.985030	0.977011	0.994152	0.984806	00:49

Table 8: Fourth algorithm for Mushroom Classification

Conclusion

We see, that results in tabels are very different if we use different algorithms. Table 1 shows, that mean ROC AUC score for Tic-Tac-Toe dataset is near 0.65, when we use base algorithm and it takes in average about 4 minutes. The best results for this dataset is in table 5, when we use third algorithm. ROC AUC mean is near 0.85 and time is about 1 min 30 sec. It's really great improvements in qualit and time. The quality increased mostly, because we use scaling technique for this unbalanced data.
Speaking about second dataset, let's look at table 2. We see, that ROC AUC is near 1, It's really high quality, however time is too slow: about 22 minutes for building predictions. However, let's look at table 8, we can see that ROC AUC decreased not so much, however average time is pretty small (about 50 seconds). It's mostly because we use L_2 threshold.

Appendix

Full code can be found on github: https://github.com/ostreech1997/LazyFCAclf
Main module: LazyFCA.py
Algorithms estimation: FinalReport.ipynb