Lazy FCA Report

Ordered sets

By Kenzin Igor

1. Introduction

Lazy learning and Formal Concept Analysis have merged in this FCA homework assignment.

Three key subjects are introduced:

1) An example of a machine learning project a process that includes loading a dataset, creating a new prediction algorithm, and comparing the outcomes.
2) Lazy learning: Predicting labels for sparse or quickly changing data.
3) Rule-learning: Analyzing information as binary descriptions of things (instead of points in a space of real numbers).

2. Description of selected dataset

Cardiovascular diseases (CVDs) are the number 1 cause of death globally, taking an estimated 17.9 million lives each year, which accounts for 31% of all deaths worldwide.
Heart failure is a common event caused by CVDs and this dataset contains 12 features that can be used to predict mortality by heart failure. Most cardiovascular diseases can be prevented by addressing behavioral risk factors such as tobacco use, unhealthy diet and obesity, physical inactivity and harmful use of alcohol using population-wide strategies. People with cardiovascular disease or who are at high cardiovascular risk (due to the presence of one or more risk factors such as hypertension, diabetes, hyperlipidemia or already established disease) need early detection and management wherein a machine learning model can be of great help.

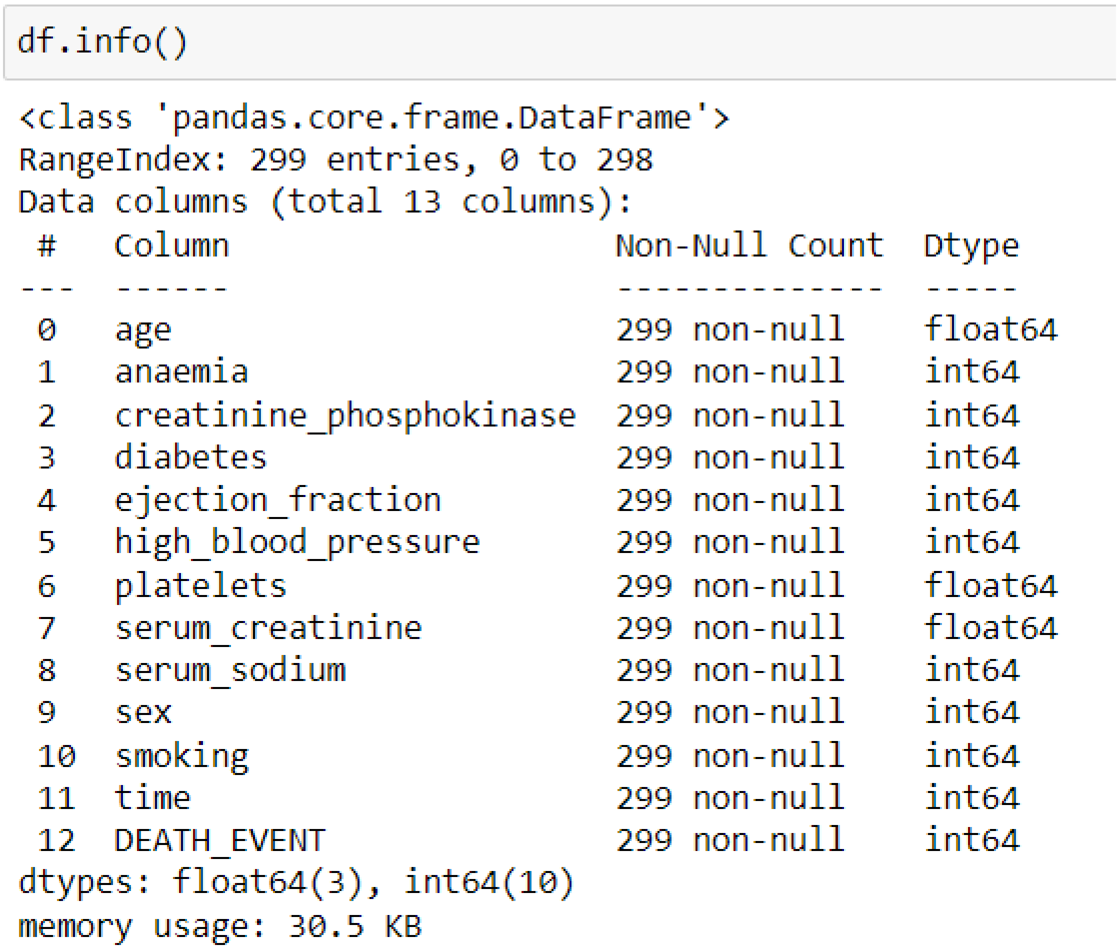
Each patient receives unique information, such as their gender, age, and blood pressure. A binary characteristic—the existence or absence of cardiac disease—was chosen as the target feature. All of these characteristics may be separated into three groups, each of which has been scaled differently:

1) Binary (for example, gender, presence of disease). For these characteristics, 0 and 1 were used in place of the values. The particular value for which we must provide 0 (or 1) was picked at random because it has no bearing on the situation.
2) Specifiable (for example, the type of pain). One new one was added for each value for each such characteristic. If the specified attribute had this

value, the value 1 was set. The first category characteristics were then eliminated.

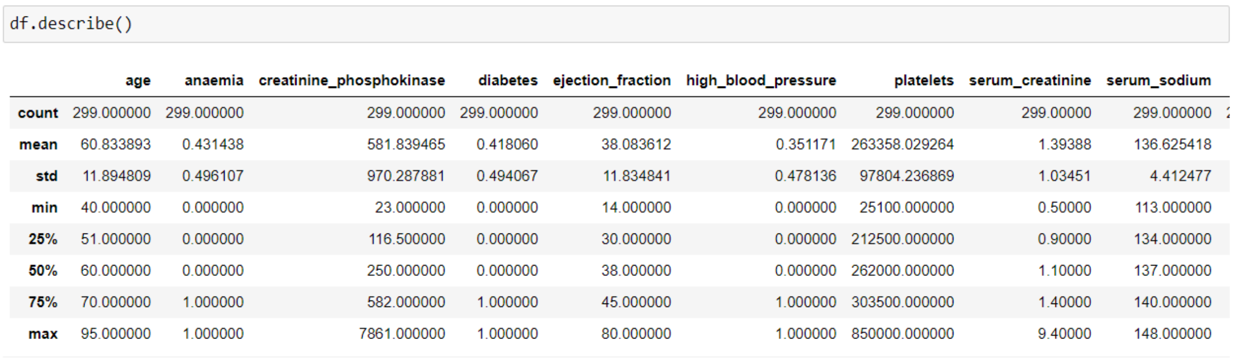
3) Numbers (for example, blood pressure). A range of permissible values of the form (min, max) was generated for each of these features. Additionally, each period was separated into 5 intervals by percentage, each of which was included as a new feature. If the object's original attribute's "greater than or equal to" requirement was satisfied, the value 1 was set. The original numerical characters were then deleted after that. As a result, a binary data set that the classifier could use was obtained.

3. Data in the dataset



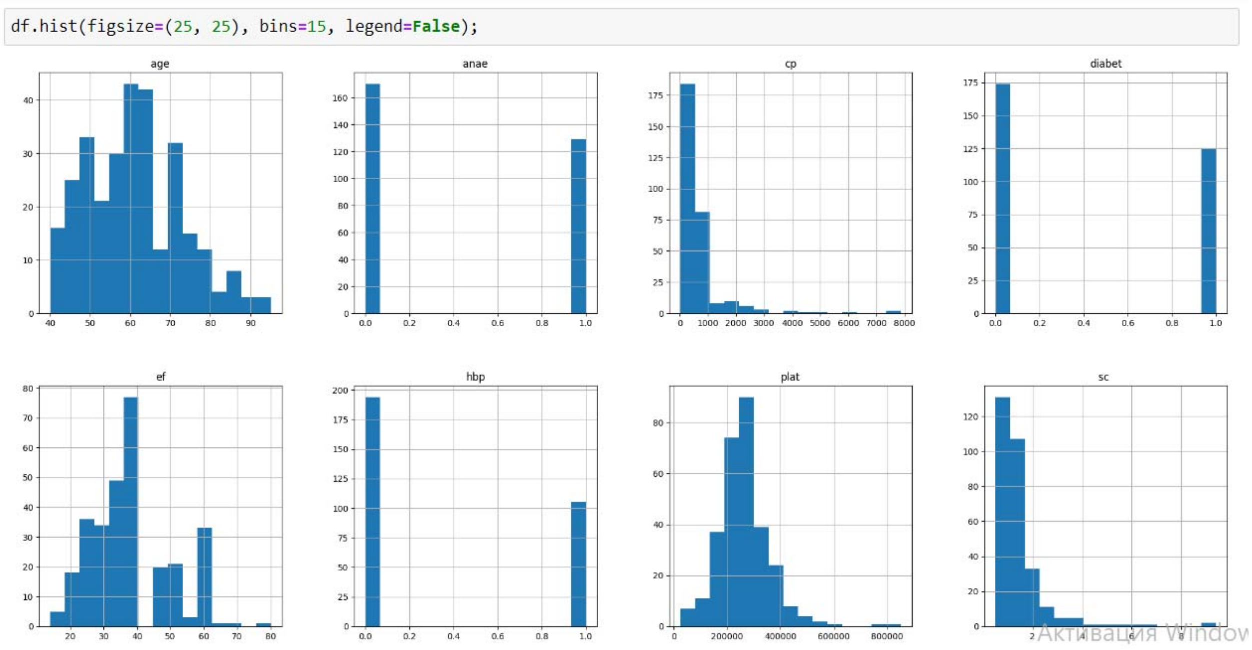
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 299 entries, 0 to 298
Data columns (total 13 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   age                       299 non-null    float64
 1   anaemia                   299 non-null    int64
 2   creatinine_phosphokinase  299 non-null    int64
 3   diabetes                  299 non-null    int64
 4   ejection_fraction         299 non-null    int64
 5   high_blood_pressure       299 non-null    int64
 6   platelets                 299 non-null    float64
 7   serum_creatinine          299 non-null    float64
 8   serum_sodium              299 non-null    int64
 9   sex                       299 non-null    int64
 10  smoking                   299 non-null    int64
 11  time                      299 non-null    int64
 12  DEATH_EVENT               299 non-null    int64
dtypes: float64(3), int64(10)
memory usage: 30.5 KB
```



```
df.describe()
```

	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets	serum_creatinine	serum_sodium
count	299.000000	299.000000	299.000000	299.000000	299.000000	299.000000	299.000000	299.00000	299.000000
mean	60.833893	0.431438	581.839465	0.418060	38.083612	0.351171	263358.029264	1.39388	136.625418
std	11.894809	0.496107	970.287881	0.494067	11.834841	0.478136	97804.236869	1.03451	4.412477
min	40.000000	0.000000	23.000000	0.000000	14.000000	0.000000	25100.000000	0.50000	113.000000
25%	51.000000	0.000000	116.500000	0.000000	30.000000	0.000000	212500.000000	0.90000	134.000000
50%	60.000000	0.000000	250.000000	0.000000	38.000000	0.000000	262000.000000	1.10000	137.000000
75%	70.000000	1.000000	582.000000	1.000000	45.000000	1.000000	303500.000000	1.40000	140.000000
max	95.000000	1.000000	7861.000000	1.000000	80.000000	1.000000	850000.000000	9.40000	148.000000



```
df.hist(figsize=(25, 25), bins=15, legend=False);
```

4. **Classification**

The dataset set was split into positive and negative classes for the proposed classification technique, and after that, the algorithm tried to assign items from the unclassified set to either the positive or negative class. The classes' intentions would be made by the algebra (+ and -). Then, any unclassified item would realize its intentions. The positive and negative items would then collide with every unclassified object after that. There is a little issue, though, that we need to take into account: if an undeclared object exclusively interacts with good intentions, we define it as positive, just as we do for the negative. A paradox exists when there is both positive and negative information. Use ordered sets to represent the training data set (context) K = (G,M,I), where G is a collection of objects, M is a collection of attributes, and I is a collection of relationships between G and M. I can be defined as a set of the form $\{(g,m)|g \in G, m \in M, gIm\}$. The target attribute, or mt, is one of all attributes. The initial set of items is then split into two sets in accordance with the following rule:

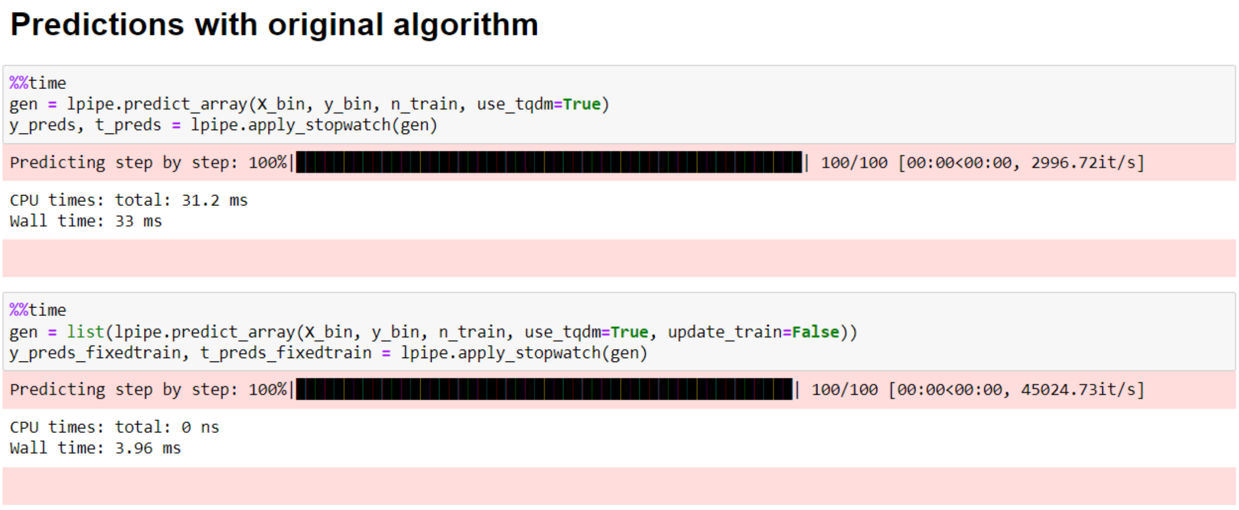
$$G_+ = \{g \in G|(g,m_t) \in I\}$$

$$G_- = \{g \in G|(g,m_t) \notin I\}.$$

Next, we will have a simple operator for the FSA, which can be defined as:
$A' = \{m \in M|(g,m) \in I, \forall g \in A\}$

$B' = \{g \in G|(g,m) \in I, \forall m \in B\}$

5. Prediction



Predictions with original algorithm

```
%%time
gen = lpipe.predict_array(X_bin, y_bin, n_train, use_tqdm=True)
y_preds, t_preds = lpipe.apply_stopwatch(gen)
```

```
Predicting step by step: 100%|██████████████████████████████████████████████████| 100/100 [00:00<00:00, 2996.72it/s]
```

```
CPU times: total: 31.2 ms
Wall time: 33 ms
```

```
%%time
gen = list(lpipe.predict_array(X_bin, y_bin, n_train, use_tqdm=True, update_train=False))
y_preds_fixedtrain, t_preds_fixedtrain = lpipe.apply_stopwatch(gen)
```

```
Predicting step by step: 100%|██████████████████████████████████████████████████| 100/100 [00:00<00:00, 45024.73it/s]
```

```
CPU times: total: 0 ns
Wall time: 3.96 ms
```

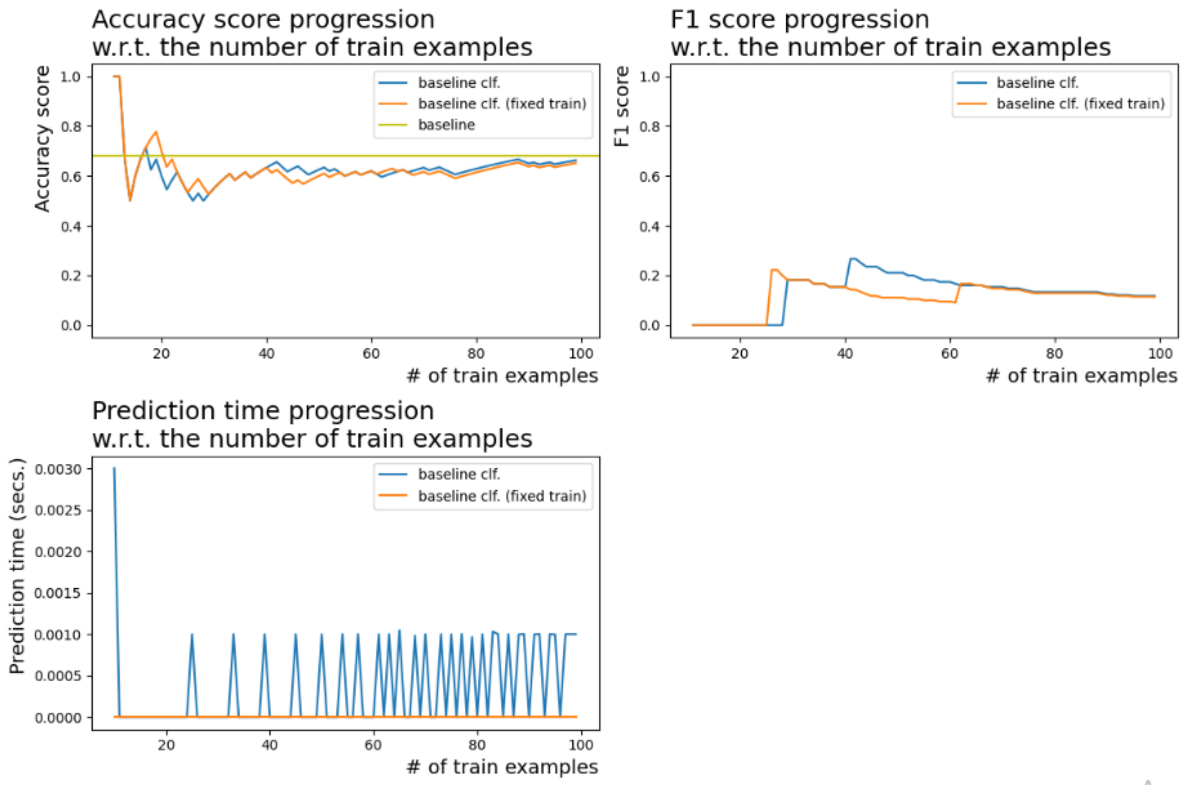
To complete the task, we take two metrics.

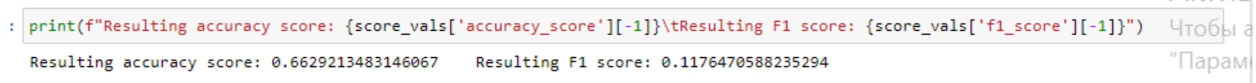
- The first one will be the accuracy score since our dataset is balanced and the metric is simple.

- The second take F1 to minimize the false negative forecast

Using this code, we see the time during which the action was performed with the ready method.

Results analytics shows:



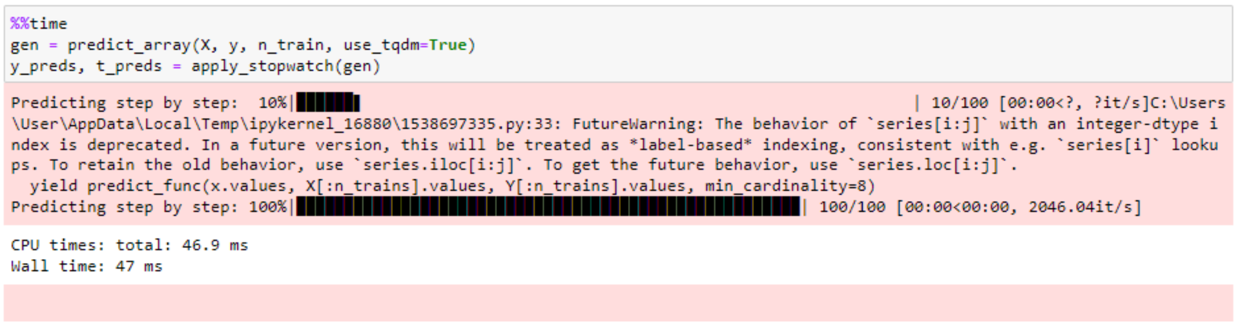


```
: print(f"Resulting accuracy score: {score_vals['accuracy_score'][-1]}\tResulting F1 score: {score_vals['f1_score'][-1]}")

  Resulting accuracy score: 0.6629213483146067    Resulting F1 score: 0.1176470588235294
```

Accuracy that we get only 0.66 and F1 score is 0.11.

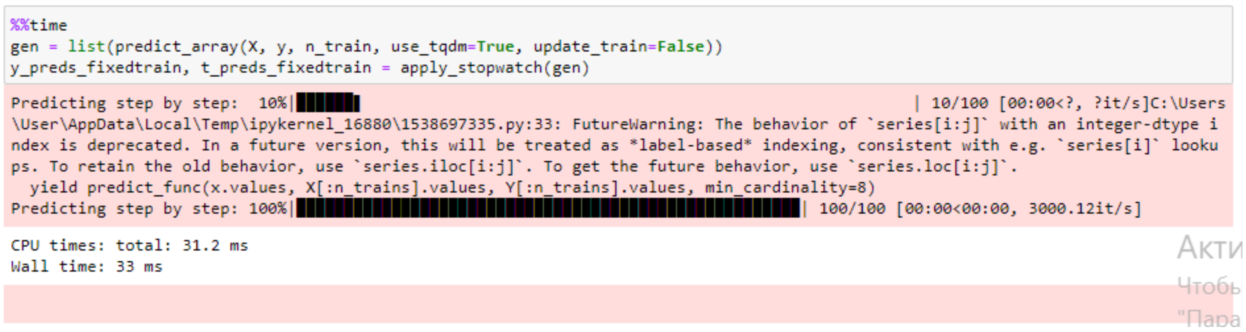
For the second method that we used on this Dataset we can see that the time has increased significantly compared to the previous one.



```
%%time
gen = predict_array(X, y, n_train, use_tqdm=True)
y_preds, t_preds = apply_stopwatch(gen)

Predicting step by step:  10%|█████                                             | 10/100 [00:00<?, ?it/s]C:\Users
\User\AppData\Local\Temp\ipykernel_16880\1538697335.py:33: FutureWarning: The behavior of `series[i:j]` with an integer-dtype i
ndex is deprecated. In a future version, this will be treated as *label-based* indexing, consistent with e.g. `series[i]` looku
ps. To retain the old behavior, use `series.iloc[i:j]`. To get the future behavior, use `series.loc[i:j]`.
  yield predict_func(x.values, X[:n_trains].values, Y[:n_trains].values, min_cardinality=8)
Predicting step by step: 100%|██████████████████████████████████████████████████| 100/100 [00:00<00:00, 2046.04it/s]

CPU times: total: 46.9 ms
Wall time: 47 ms
```



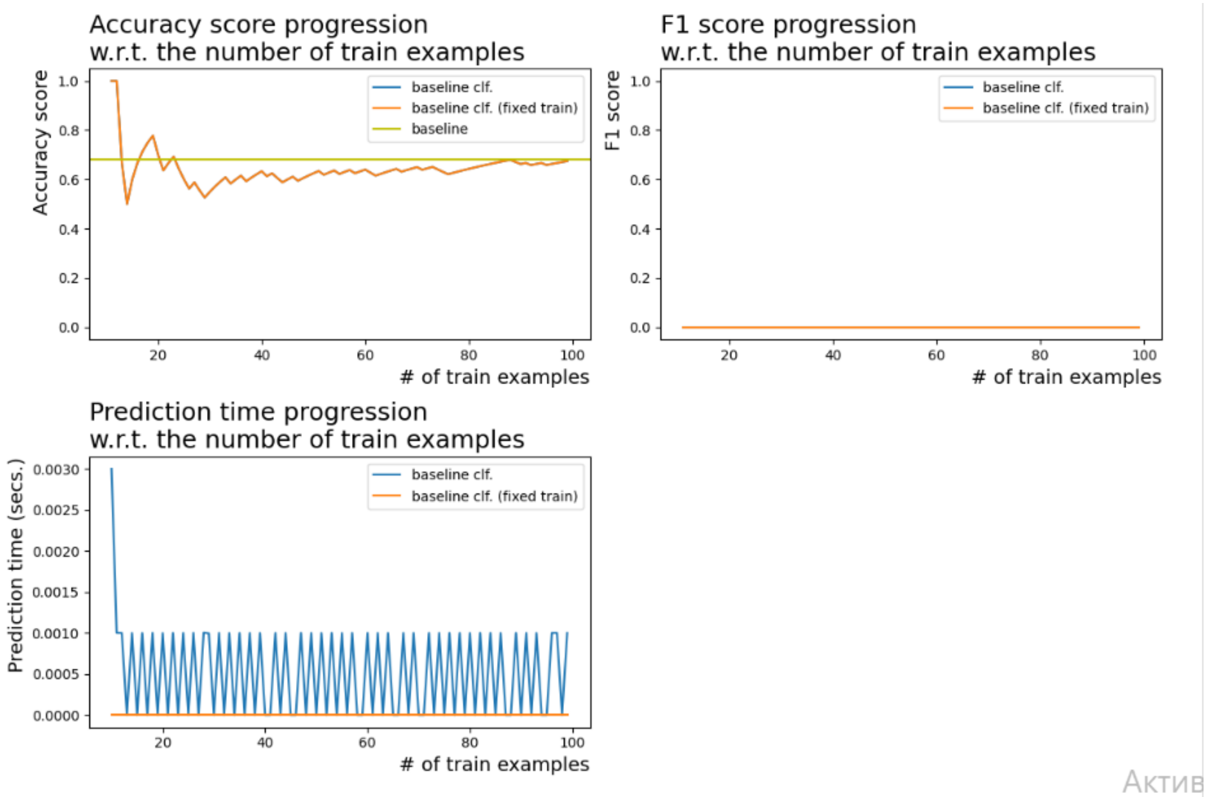
```
%%time
gen = list(predict_array(X, y, n_train, use_tqdm=True, update_train=False))
y_preds_fixedtrain, t_preds_fixedtrain = apply_stopwatch(gen)

Predicting step by step:  10%|█████                                             | 10/100 [00:00<?, ?it/s]C:\Users
\User\AppData\Local\Temp\ipykernel_16880\1538697335.py:33: FutureWarning: The behavior of `series[i:j]` with an integer-dtype i
ndex is deprecated. In a future version, this will be treated as *label-based* indexing, consistent with e.g. `series[i]` looku
ps. To retain the old behavior, use `series.iloc[i:j]`. To get the future behavior, use `series.loc[i:j]`.
  yield predict_func(x.values, X[:n_trains].values, Y[:n_trains].values, min_cardinality=8)
Predicting step by step: 100%|██████████████████████████████████████████████████| 100/100 [00:00<00:00, 3000.12it/s]

CPU times: total: 31.2 ms
Wall time: 33 ms
```

However, when checking the results, our data is slightly higher than last time.

The values have not changed much, but there is a positive shift.



```
8]: print(f"Resulting accuracy score: {score_vals['accuracy_score'][-1]}\tResulting F1 score: {score_vals['f1_score'][-1]}")

    Resulting accuracy score: 0.6741573033707865     Resulting F1 score: 0.0
```

6. Conclusion

As a result, we managed to improve the code, because the accuracy value increased from 0.6629 to 0.6741.

The original code has a good analytic probability. If it is slightly modified, then it will be possible to use it fully.

CPU times: total: 31.2 ms & Wall time: 33 ms for original method. CPU times: total: 62.5 ms & Wall time: 36 ms for the second method.

The work done will have a good impact on the further understanding of the FCA and help develop in this direction. There may be minor bugs in the work, so to further improve the code, you will need to return after a while. We managed to get a better understanding of FCA, however, other methods can be applied, such as XGBoostClassifier, CatBoostClassifier and others, which can also give a higher value.