REPORT for
Lazy FCA Homework
Ordered Set

- by Abhishek Kardam

1. Introduction

This homework merges Lazy Learning with Formal Concept Analysis. Lazy Learning is a learning method in which generalization of the training data is,in theory, delayed until a query is made to the system, as opposed to eager learning, where the system tries to generalize the training data before receiving queries. Formal Concept Analysis (FCA) is a principled way of deriving a concept hierarchy or formal ontology from a collection of objects and their properties. Each concept in the hierarchy represents the objects sharing some set of properties; and each sub-concept in the hierarchy represents a subset of the objects in the concepts above it.

This homework introduces students to three main topics:

1. Typical machine learning project:
   The pipeline of loading a dataset; feature engineering, designing a new predictive algorithm, results comparison.
2. Lazy learning:
   Prediction labels for small or rapidly changing data;
3. Rule-learning (on part with FCA):
4. Viewing data as a binary description of objects (instead of points in a space of real numbers).

We first have to choose a dataset for adapting the pipeline.

2. Dataset

I have chosen the Car-evaluation dataset from the website UCI . The dataset contains 1728 instances with 7 Attributes and 0 missing values. Attributes in the dataset are, 'buying', 'maint', 'doors', 'persons', 'lug_boot', 'safety', 'class' , every attribute is categorical with three to four values. The class of the dataset has 4 values (‘unacc’, ’acc’, ’good’, ’vgood’).

We will have to binarize our data as features of our data set have large difference between their ranges, so we have to make every value binary(0 or 1)

3. Binarization

For Binarization, I have used the **One-hot encoding** method. One-hot encoding turns your categorical data into a binary vector representation. We have a python library 'Pandas', allows us to easily on-hot encode our categorical data with its function **'pandas.get_dummies()'**.
get_dummies() function is used for data manipulation and converts categorical data into dummy or indicator variables.
One-hot encoding can be very helpful in terms of working with categorical variables. One major drawback, however, is that it creates significantly more data. Because of this, it shouldn't be used when there are too many categories.

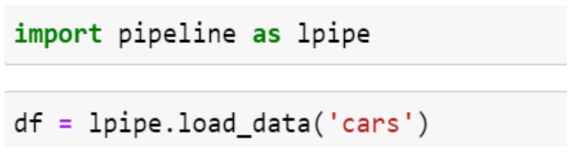
4. Prediction quality measure

For my project, I have used **'Accuracy'** and **'F1 score'** as prediction metrics.
Accuracy is the measure of all the correctly identified cases. It is most used when all the classes are equally important.
F1 score is the harmonic mean of precision and recall, which means that the F1 score will tell you the model's balanced ability to both capture positive cases(recall) and be accurate with the cases it does not capture(precision).

A poorly performing model which only predicts the majority class will seem accurate to other metrics, such as accuracy. However, the poor performance will be shown in F1 score as the model will have neither a good precision on the positive class.

For both Accuracy and F1-score 0 is the worst possible score and 1 is a perfect score indicating that the model predicts each observation correctly.

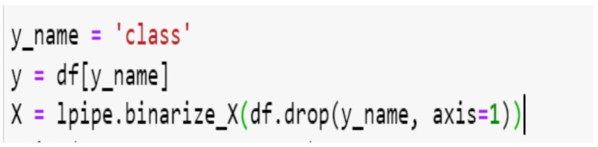
5. Adapt the pipeline



```
import pipeline as lpipe
```

```
df = lpipe.load_data('cars')
```

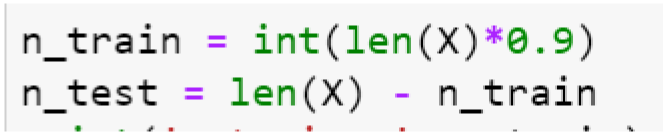
Now we have to adapt the lazy_pipeline.py, we first load our data using load_data() function, which calls another function(load_cars()) for data loading. We read our data from my github repository, then we rename the columns and binarize the class column. For binarizing the class column, we consider values 'good' and 'vgood' as True and values 'unacc' and 'acc' as False.



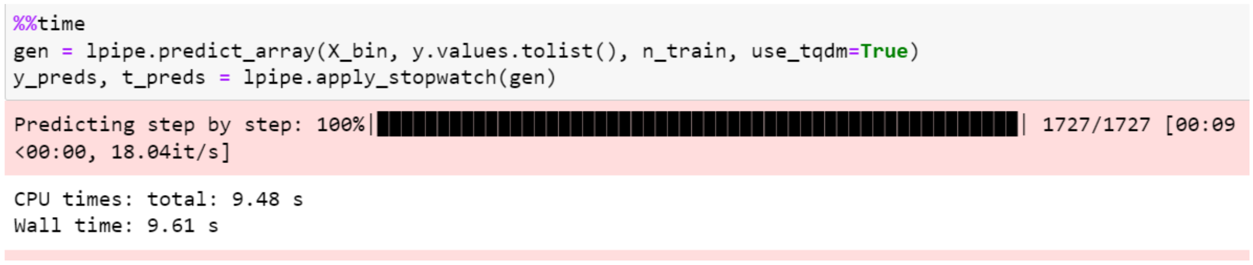
```
y_name = 'class'
y = df[y_name]
X = lpipe.binarize_X(df.drop(y_name, axis=1))
```

Then we call the binarize_X() function to binarize aur whole data, after dropping the class column. Inside the binarize_X() function we use the 'pandas.get_dummies()' function.

We shuffle the X(data) and order y(classes) to follow the order of rows from X(data). We convert our dataframe into sets (X_bin) and split our data into training and test data. Here we are taking 90% data as training data and remaining 10% data as test data.



```
n_train = int(len(X)*0.9)
n_test = len(X) - n_train
```



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

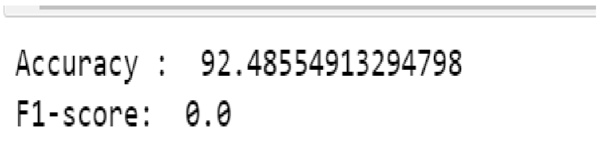
```
Predicting step by step: 100%|██████████████████████████████████████████████████| 1727/1727 [00:09
<00:00, 18.04it/s]
```

```
CPU times: total: 9.48 s
Wall time: 9.61 s
```

We then call our prediction function ‘lpipe.predict_array()’ which takes X_bin (set of our data), classes (y) as a list, training data size and boolean value use_tqdm for progress bar.. We also call the function ‘lpipe.apply_stopwatch()’ to show the prediction time.

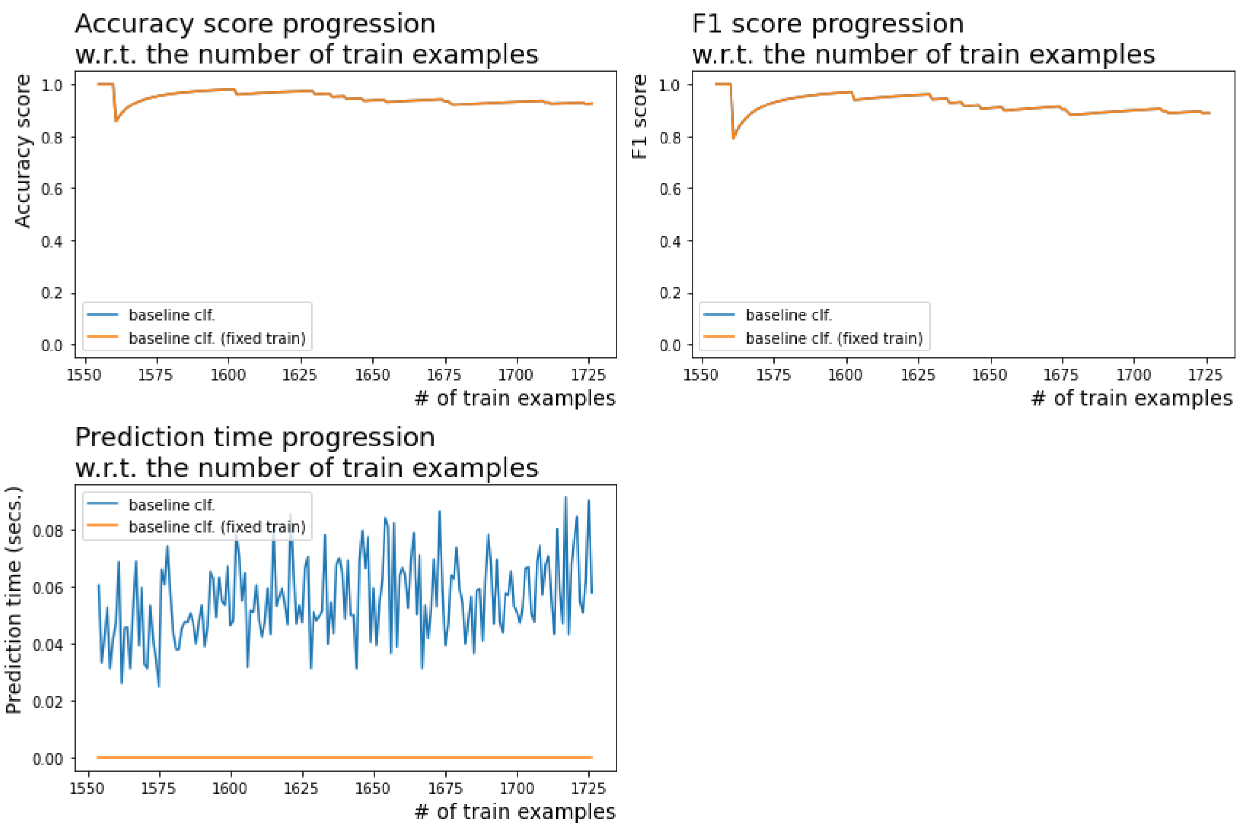
For our dataset the algorithm takes 9.48 sec to predict the classes to test examples.

We then compute prediction metrics (accuracy and f1-score) and plot them. For our dataset, algorithm have an accuracy of 92.48 % and f1-score of nearly 0.



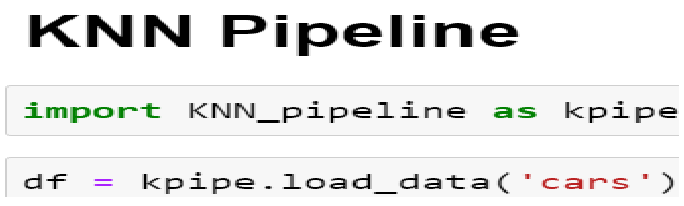
```
Accuracy :  92.48554913294798
F1-score:  0.0
```

Accuracy shows that our algorithm is correctly identifying 92.48% of data but f1-score shows that our algorithm is highly under-fitted.



6. Modify Pipeline

For modifying the pipeline, I have made a new python file 'KNN_pipeline.py', in which I have changed the prediction function. In the prediction function of knn pipeline, algorithm is taking one test example at a time, calculate the euclidean distance of that test example to every training example, sort the euclidean distance in ascending order, take first 5 distances(closest examples) of the sorted list, and classify the test example according to the most frequent class from the closest examples.
For example if the first 5 distances in the sorted list have classes [1,0,1,0,0] then the test example will be classified as class 0.



KNN Pipeline

```
import KNN_pipeline as kpipe
```

```
df = kpipe.load_data('cars')
```

Then we predict the same 'car-evaluation' dataset with the new algorithm.

For the new algorithm we do not convert our data into sets (X_bin) instead we convert the binary data to 0,1 (for calculation in distance).

```
import numpy
X = numpy.multiply(X, 1)
X.head()
```

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

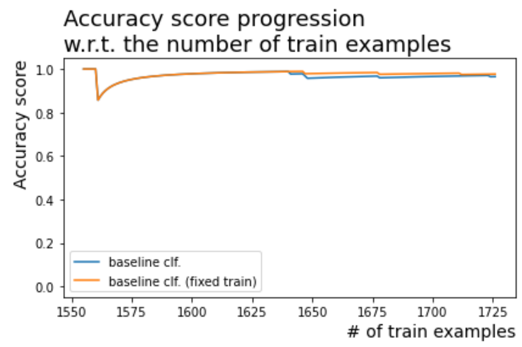
```
Predicting step by step: 100%|██████████████████████████████████████████████████████| 1727/1727 [00:33
<00:00,  5.22it/s]
```

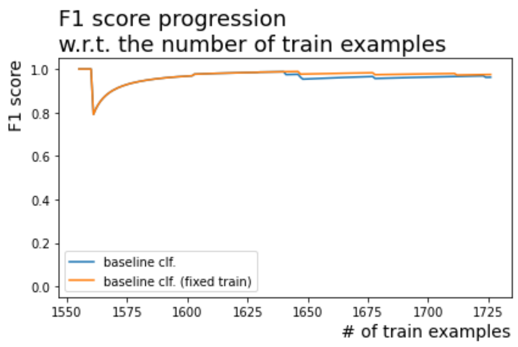
```
CPU times: total: 33.5 s
Wall time: 33.1 s
```

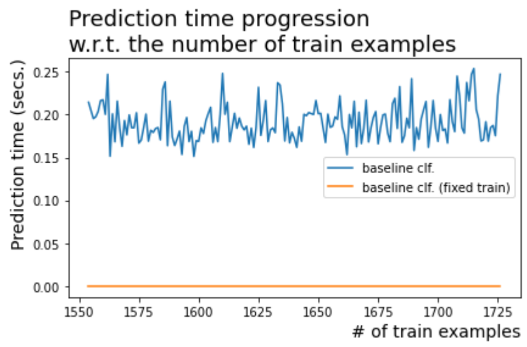
We then predict our 10% test data and the new algorithm takes 33.5 secs to predict all test examples which is slower that the previous algorithm which took 9.48 secs.

```
Accuracy :  96.53179190751445
F1-score:  72.72727272727273
```

We then compute prediction metrics (accuracy and f1-score) and plot them. For our dataset, the new algorithm has an accuracy of 96.5 % and f1-score 0.72, which shows that our new algorithm is working very better than the previous lazy pipeline algorithm and solves the problem of under-fitting.



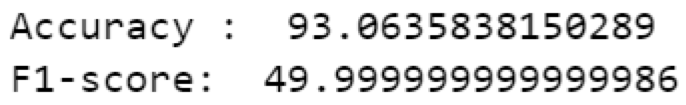




7. Comparison with other Algorithm

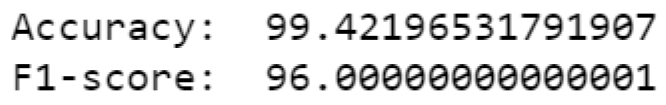
We now compare our new algorithm with 2 popular rule-based models.

1. **Decision Tree**
   We predict the same ‘car-evaluation’ dataset with the decision tree algorithm, we take 90% data as training data and 10 % data as test data. In this algorithm we take the gini index to classify the training examples and then predict the test examples.
   This algorithm has accuracy of 93.06% and f1 score of 0.499, which shows that this algorithm is also great for prediction but it is still not better than our KNN_pipeline algorithm which has accuracy of 96.5 % and f1-score 0.72.



```
Accuracy :  93.0635838150289
F1-score:  49.999999999999986
```

2. **Random Forest**
   We predict the same ‘car-evaluation’ dataset with the Random Forest algorithm, we take 90% data as training data and 10 % data as test data. In this algorithm we use ‘sklearn.ensemble import RandomForestClassifier’ library to predict the test examples.
   This algorithm is already trained so it has accuracy of 99.42% and f1 score of 96, which is far better than our KNN_pipeline algorithm which has accuracy of 96.5 % and f1-score 0.72.



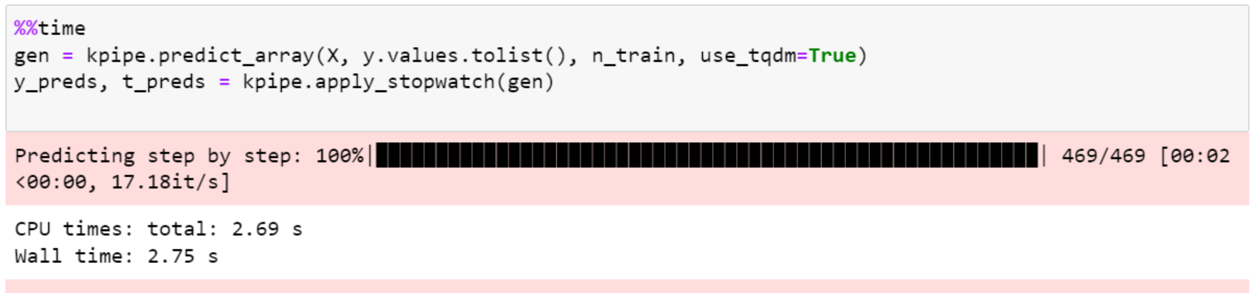
```
Accuracy:  99.42196531791907
F1-score:  96.00000000000001
```

8. More Datasets

I have tested the KNN_pipeline with 3 more datasets (all the datasets are taken from website UCI and are available in my github repository).

1. **Thoracic Surgery dataset**
   This dataset is dedicated to classification problem related to the post-operative life expectancy in the lung cancer patients
   This dataset has 17 attributes and 470 instances.
   Attributes of this dataset are('DGN', 'PRE4', 'PRE5', 'PRE6', 'PRE7', 'PRE8', 'PRE9','PRE10', 'PRE11','PRE14', 'PRE17', 'PRE19', 'PRE25', 'PRE30', 'PRE32', 'AGE', 'class').

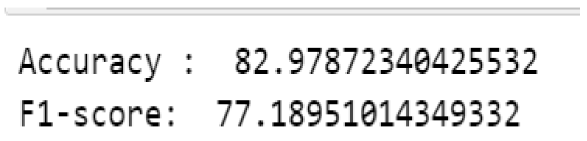
After loading and binarizing the data we take 90% data as training data and 10% data as test data. We then predict the test data with our KNN_pipeline algorithm, it takes 2.7 sec to predict all the test examples.



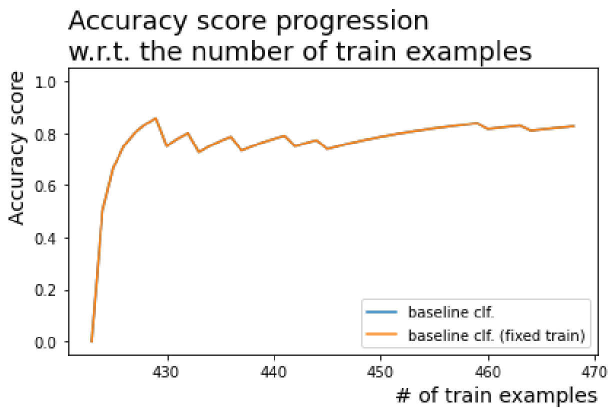
```
%%time
gen = kpipe.predict_array(X, y.values.tolist(), n_train, use_tqdm=True)
y_preds, t_preds = kpipe.apply_stopwatch(gen)
```

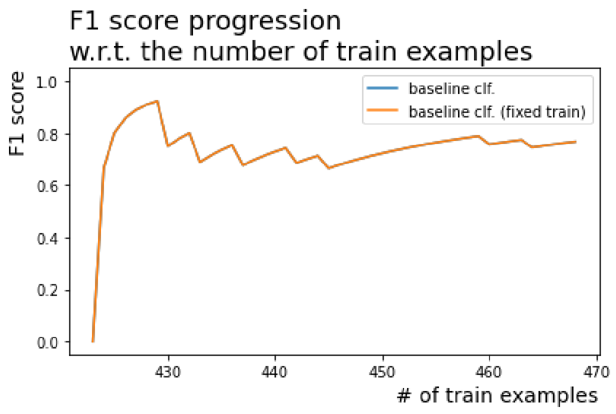
```
Predicting step by step: 100%|██████████████████████████████████████████████████| 469/469 [00:02
<00:00, 17.18it/s]

CPU times: total: 2.69 s
Wall time: 2.75 s
```

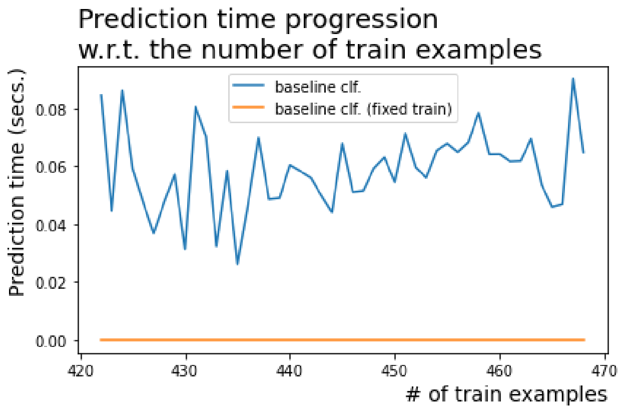
We then compute prediction metrics (accuracy and f1-score) and plot them. For Thoracic Surgery dataset, the KNN_pipeline algorithm has an accuracy of 82.97 % and f1-score 0.77, which shows that on decreasing the size of the dataset, the accuracy of the algorithm also decreases.



```
Accuracy :  82.97872340425532
F1-score:  77.18951014349332
```

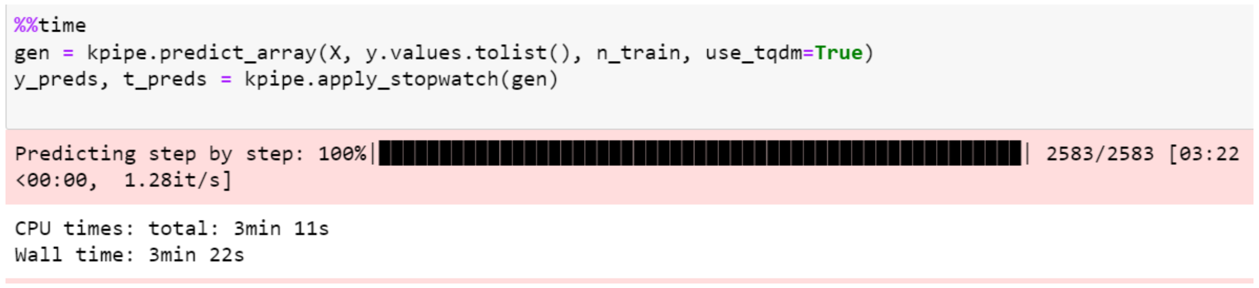






2. **Seismic bumps dataset**
   This dataset describes the problem of high energy seismic bumps forecasting in a coal mine.
   This dataset has 19 attributes and 2584instances.
   Attributes of this dataset are('seismic', 'seismoacoustic', 'shift', 'genergy', 'gpuls', 'gdenergy', 'gdpuls', 'ghazard', 'nbumps', 'nbumps2', 'nbumps3', 'nbumps4', 'nbumps5', 'nbumps6', 'nbumps7', 'nbumps89', 'energy', 'maxenergy', 'class').

   After loading and binarizing the data we take 90% data as training data and 10% data as test data. We then predict the test data with our KNN_pipeline algorithm, it takes 3 minutes and 22 seconds to predict all the test examples.

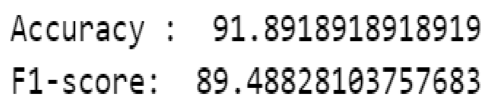


```
%%time
gen = kpipe.predict_array(X, y.values.tolist(), n_train, use_tqdm=True)
y_preds, t_preds = kpipe.apply_stopwatch(gen)
```

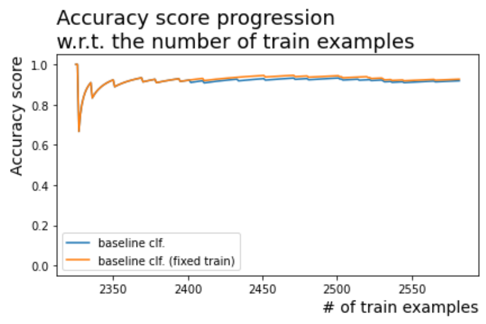
```
Predicting step by step: 100%|██████████████████████████████████████████████████████| 2583/2583 [03:22
<00:00,  1.28it/s]
```

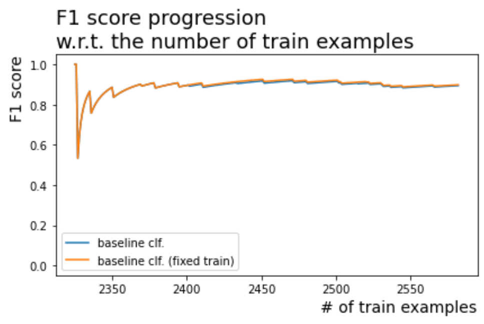
```
CPU times: total: 3min 11s
Wall time: 3min 22s
```

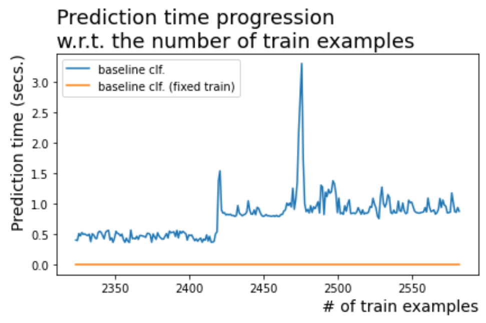
We then compute prediction metrics (accuracy and f1-score) and plot them. For Seismic bumps dataset, the KNN_pipeline algorithm has an accuracy of 91.89 % and f1-score 0.89, which shows that our algorithm works very well with big datasets.



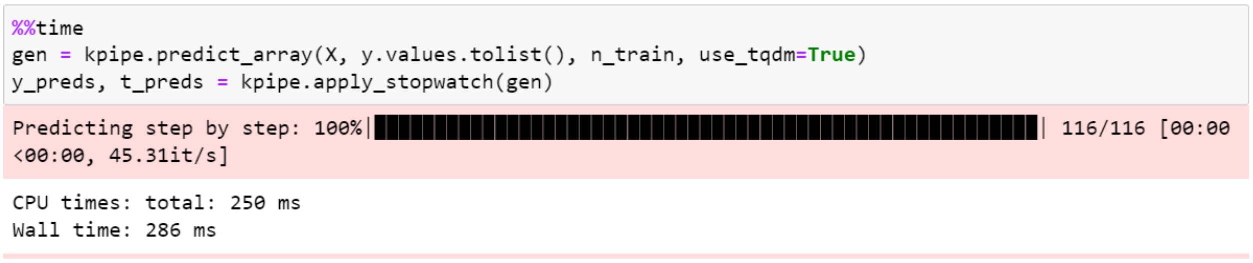
```
Accuracy :  91.8918918918919
F1-score:  89.48828103757683
```







3. **Breast Cancer Dataset**
   This dataset has 10 attributes and 116 instances. Class has only two values, breast cancer present or absent,.
   Attributes of this dataset are('Age', 'BMI', 'Glucose', 'Insulin', 'HOMA', 'Leptin', 'Adiponectin', 'Resistin', 'MCP.1', 'class')
   After loading and binarizing the data we take 90% data as training data and 10% data as test data. We then predict the test data with our KNN_pipeline algorithm, it takes 250 milli-seconds to predict all the test examples.

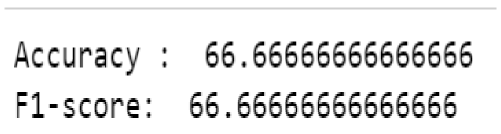


```
%%time
gen = kpipe.predict_array(X, y.values.tolist(), n_train, use_tqdm=True)
y_preds, t_preds = kpipe.apply_stopwatch(gen)
```

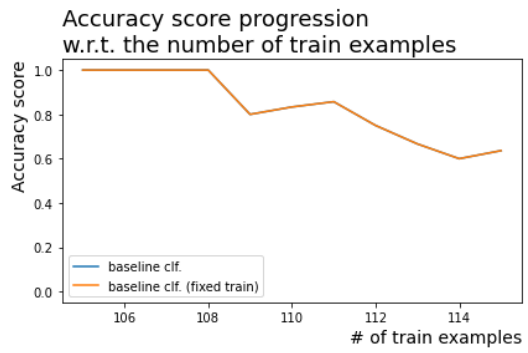
```
Predicting step by step: 100%|██████████████████████████████████████████| 116/116 [00:00
<00:00, 45.31it/s]
```

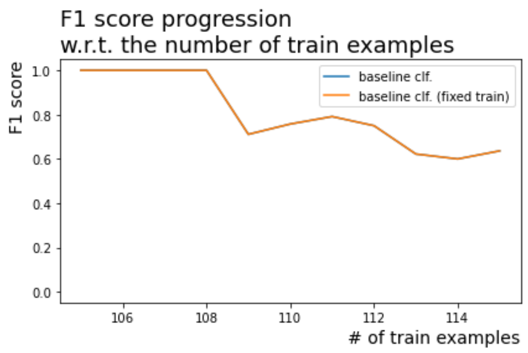
```
CPU times: total: 250 ms
Wall time: 286 ms
```

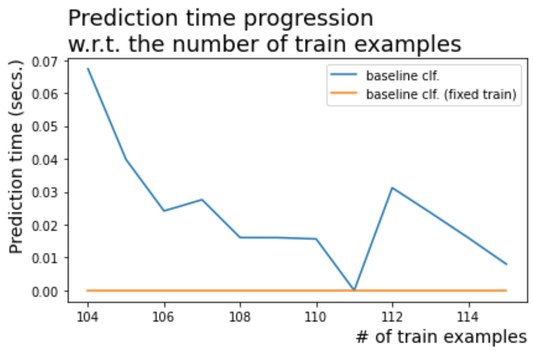
We then compute prediction metrics (accuracy and f1-score) and plot them. For Seismic bumps dataset, the KNN_pipeline algorithm has an accuracy of 66.66 % and f1-score 0.66.



```
Accuracy :  66.66666666666666
F1-score:  66.66666666666666
```







9. Conclusion

Our new KNN_pipeline algorithm takes more computational time as compared to the original pipeline algorithm but the new algorithm predicts test examples with more accuracy than the original algorithm.
Our new KNN_pipeline algorithm work better than big data and predict most of the test examples correctly.