# OSDA Big Homework Report

## File Structure Description

- `framework/Experiment.ipynb` - core of the experiment, all the training steps and data preprocessing steps can be found there
- `framework/lazy_learning_framework.py` - implementation of lazy learning framework with all the algorithms
- `data/*` - all data files are stored here

## Introduction

The task of the big homework was to use Lazy Learning approaches for the binary classification task. The main goals of the homework included:

- Own dataset selection
- Dataset preprocessing/cleaning to fit into FCA framework
- Lazy Learning implementation
- Allow Online Learning approach in the dataset

## Dataset selection

I decided to choose the dataset which would be not too big to hold quick experiments, but also I wanted to include not only binary but also more complex features (categories, numerical, data) to apply binarization techniques.

I chose a well-known Breast Cancer Dataset from the UCI repository since I already worked with it and it was very interesting for me to compare Lazy Learning FCA algorithms to the classical Machine Learning approaches.

## Dataset Preprocessing & Cleaning

After loading the dataset I got the following structure:

```
[2]: data = pd.read_csv(DATA_PATH / 'breast-cancer.data', header = None)
     data.columns = ['class', 'age', 'menopause', 'tumor-size', 'inv-nodes', 'node-caps', 'deg-malig', 'breast', 'breast-quad', 'irradiat']
     data.head(2)
```

[2]:

| | class | age | menopause | tumor-size | inv-nodes | node-caps | deg-malig | breast | breast-quad | irradiat |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | no-recurrence-events | 30-39 | premeno | 30-34 | 0-2 | no | 3 | left | left_low | no |
| **1** | no-recurrence-events | 40-49 | premeno | 20-24 | 0-2 | no | 2 | right | right_up | no |

Upon inspection, I found that there were 2 types of data used in the dataset: binary and categorical features. I used a common approach for binarizing the dataset in this case called binning. It is very easy to implement, but yet effective to use with FCA algorithms.

```
[6]: # now let's deal with textual columns that have more than 2 unique values

     for col in ['age', 'menopause', 'tumor-size', 'inv-nodes', 'node-caps', 'deg-malig', 'breast-quad']:
         data = pd.concat([data, pd.get_dummies(data[col], prefix=f'{col}')], axis = 1).drop(columns = [col])

     # check that we didn't spoil the data

     assert(data.isna().sum().sum() == 0) # no missing values
     assert((data.nunique().unique()[0] == 2) & (data.nunique().unique().shape[0] == 1)) # all columns have only 2 unique values
```

## Implementation of Lazy Learning Frameworks And Classification Algorithms

For finding frequent itemsets I used 3 algorithms common for frequent itemsets mining: apriori, fpmax, fpgrowth. These algorithms were available in `mlxtend` Python package, so I didn't need to code them from scratch.

For lazy classification approach, we firstly need to find frequent item sets for positive and negative class separately and then use some algorithmic approach to classify incoming items from the test set. I implemented 3 separate approaches for such classification.

1. `most_itemsets` - for each incoming test set item I looked for itemsets from positive and negative classes, then I just predicted the class, which had more its' itemsets inside the test set item

```
def _algo_most_itemsets(self, X_train, Y_train, X_test):
    '''
        Classify each example from test set based on how many itemsets match frequent_zero_class itemsets and match frequent_one_class itemsets
        Think of, |g+ng| vs |g-ng|
        :param X_train: binarized items of the train set
        :param Y_train: target values for X_train
        :param X_test: binarized items of the test set
    '''

    frequent_zero_class = self._find_frequent_itemsets(X_train[Y_train == 0])
    frequent_one_class = self._find_frequent_itemsets(X_train[Y_train == 1])

    test_classes = []
    for (i, row) in X_test.iterrows():
        total_matches_one = 0
        for item in frequent_one_class['itemsets']:
            columns = np.array(list(item))
            if X_test.loc[i][columns].sum() == len(columns):
                total_matches_one += 1

        total_matches_zero = 0
        for item in frequent_zero_class['itemsets']:
            columns = np.array(list(item))
            if X_test.loc[i][columns].sum() == len(columns):
                total_matches_zero += 1

        if total_matches_one > total_matches_zero:
            output_label = 1
        elif total_matches_one < total_matches_zero:
            output_label = 0
        else:
            output_label = 1
        test_classes.append(output_label)

        if self.allow_online_learning:
            X_train = X_train.append(X_test.iloc[i:i+1, :]).reset_index(drop = True)
            Y_train = Y_train.append(pd.Series([output_label])).reset_index(drop = True)

            frequent_zero_class = self._find_frequent_itemsets(X_train[Y_train == 0])
            frequent_one_class = self._find_frequent_itemsets(X_train[Y_train == 1])

    return np.array(test_classes)
```

2. `itemsets_intersection_sums` - for each incoming test set item I looked for itemsets from positive and negative classes, for each itemset I calculated how much common it had with the coming test set item, then I just summed all the "similarity" values and predicted the class, that got larger similarity score

```python
def _algo_itemsets_intersection_sums(self, X_train, Y_train, X_test):
    '''
        Classify each example from test set based on the total sums of proportion of matching each itemset
        to the items found for the validation sample
        :param X_train: binarized items of the train set
        :param Y_train: target values for X_train
        :param X_test: binarized items of the test set
    '''

    frequent_zero_class = self._find_frequent_itemsets(X_train[Y_train == 0])
    frequent_one_class = self._find_frequent_itemsets(X_train[Y_train == 1])

    test_classes = []
    for (i, row) in X_test.iterrows():
        total_matches_one = 0
        for item in frequent_one_class['itemsets']:
            columns = np.array(list(item))
            total_matches_one += X_test.loc[i][columns].sum() / len(columns)

        total_matches_zero = 0
        for item in frequent_zero_class['itemsets']:
            columns = np.array(list(item))
            total_matches_zero += X_test.loc[i][columns].sum() / len(columns)

        if total_matches_one > total_matches_zero:
            output_label = 1
        elif total_matches_one < total_matches_zero:
            output_label = 0
        else:
            output_label = 1
        test_classes.append(output_label)

        if self.allow_online_learning:
            X_train = X_train.append(X_test.iloc[i:i+1, :]).reset_index(drop = True)
            Y_train = Y_train.append(pd.Series([output_label])).reset_index(drop = True)

            frequent_zero_class = self._find_frequent_itemsets(X_train[Y_train == 0])
            frequent_one_class = self._find_frequent_itemsets(X_train[Y_train == 1])

    return np.array(test_classes)
```

3. `itemsets_intersection_probas` - this is very similar to `itemsets_intersection_sums` , but instead of summation I used multiplication, this was kinda inspired by Naive Bayes and probabilistic concepts (unfortunately, this approach worked worst of all)

```python
def _algo_itemsets_intersection_probas(self, X_train, Y_train, X_test):
    '''
        Classify each example from test set based on the multiple of probabilities of itemset match the prediction item
        :param X_train: binarized items of the train set
        :param Y_train: target values for X_train
        :param X_test: binarized items of the test set
    '''

    frequent_zero_class = self._find_frequent_itemsets(X_train[Y_train == 0])
    frequent_one_class = self._find_frequent_itemsets(X_train[Y_train == 1])

    test_classes = []
    for (i, row) in X_test.iterrows():
        total_matches_one = 1
        for item in frequent_one_class['itemsets']:
            columns = np.array(list(item))
            total_matches_one *= X_test.loc[i][columns].sum() / len(columns)

        total_matches_zero = 1
        for item in frequent_zero_class['itemsets']:
            columns = np.array(list(item))
            total_matches_zero *= X_test.loc[i][columns].sum() / len(columns)

        if total_matches_one > total_matches_zero:
            output_label = 1
        elif total_matches_one < total_matches_zero:
            output_label = 0
        else:
            output_label = 1
        test_classes.append(output_label)

        if self.allow_online_learning:
            X_train = X_train.append(X_test.iloc[i:i+1, :]).reset_index(drop = True)
            Y_train = Y_train.append(pd.Series([output_label])).reset_index(drop = True)

            frequent_zero_class = self._find_frequent_itemsets(X_train[Y_train == 0])
            frequent_one_class = self._find_frequent_itemsets(X_train[Y_train == 1])

    return np.array(test_classes)
```

## Best hyperparameters selection, Results, Comparisson to Sklearn Algorithms

For parameter selection I created a simple GridSearch for loop, where I iterated over all possible combinations of parameters and selected best combination based on Out-Of-Fold accuracy score.

```
[14]: # grid search loop
# I will limit the search space a bit to save time

grid_search_params = {
    'min_support': [0.2, 0.3, 0.4],
    'allow_online_learning': [False, True],
    'fca_algorithm': ['fpgrowth', 'apriori'],
    'prediction_algorithm': ['most_itemsets', 'itemsets_intersection_sums']
}

best_acc = 0
best_params = {}
iteration_idx = 0
for min_support in grid_search_params['min_support']:
    for allow_online_learning in grid_search_params['allow_online_learning']:
        for fca_algorithm in grid_search_params['fca_algorithm']:
            for prediction_algorithm in grid_search_params['prediction_algorithm']:

                print(f'\n\n\n---------------------------------Running GridSearch; Iteration {iteration_idx}---------------------------------\n')
                iteration_idx+=1

                params = {
                    'min_support': min_support,
                    'allow_online_learning': allow_online_learning,
                    'fca_algorithm': fca_algorithm,
                    'prediction_algorithm': prediction_algorithm
                }

                acc = validate_algo(data, params)
                if acc > best_acc:
                    best_acc = acc
                    best_params = params
```

Here is the best hyperparameter set:

```
Best Accuracy 0.7447552447552448
Best Params {'min_support': 0.3, 'allow_online_learning': True, 'fca_algorithm': 'fpgrowth', 'prediction_algorithm': 'itemsets_intersec
```

This is infact better then sklearn models with default hyperparameters, I suspect that it might be even better then models with optimal hyperparameters, but I haven't tested it:

```
Logistic Regression OOF accuracy 0.7132867132867133
KNN OOF accuracy 0.6993006993006993
Decision Tree 0.6468531468531469
```

In general I noticed that for the dataset Lazy Models were not only better in terms of accuracy, but also much better in terms of f1-score, which is much more important. The only drawback I could notice is a slight inconsistency between folds, meanwhile Sklearn models were very consistent between folds.

All the validation and testing results for the models are available in the `framework/Experiment.ipynb` file.