Отчет по LazyFCA

Потемкин Павел

Для выполнения задания использовался, данный в исходном репозитории датасет Tic-Tac-Toe. Для удобства обработки датасет брался в исходном состоянии с UCI (https://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe+Endgame).

В данном датасете присутствует 2 target class: ‘positive’ / ‘negative’. ‘Positive’ означает, что X победили в данной расстановке на поле. Ниже можно увидеть пример нескольких записей из данного датасета:

```
['b' 'b' 'x' 'b' 'x' 'o' 'x' 'b' 'o' 'positive']
['b' 'o' 'o' 'b' 'o' 'x' 'x' 'x' 'x' 'positive']
['o' 'o' 'o' 'b' 'b' 'x' 'x' 'x' 'b' 'negative']
```

Как и указывалось в задании – данные сразу после считывания случайно перемешивались и делились на тестовую выборку и обучающую в отношении 1 к 9. Сразу же после разделения на обучающей выборке происходило разделение на + и – контексты. Положительным контекстом был ‘positive’ target class.

В моей работе записи из тестовой выборки поступают в классификатор по одному и каждому из них классифицируется по алгоритму FCA. Для каждого определенного элемента далее сравнивается ответ алгоритма с реальным ответом и записывается в массив результатов число, чтобы посчитать итоговые метрики. Эти числа напрямую являются значением True Positive и тд, просто введены для более удобного подсчета итоговых метрик:

1 - true positive
2 - false positive
3 - true negative
4 - false negative
-1 - метод не может дать точного ответа

На основе этого считались основные метрики для определения качества модели:

```python
def accuracy(results):
    return (np.count_nonzero(results == 1) + np.count_nonzero(results == 3)) / len(results)

def recall(results):
    return (np.count_nonzero(results == 1)) / \
    max((np.count_nonzero(results == 1) + np.count_nonzero(results == 4)) + np.count_nonzero(results == -1), 1)

def precision(results):
    return (np.count_nonzero(results == 1)) / max((np.count_nonzero(results <= 2)), 1)

def tn_rate(results):
    return (np.count_nonzero(results == 3)) / \
    max((np.count_nonzero(results == 3) + np.count_nonzero(results == 2)) + np.count_nonzero(results == -1), 1)

def print_results(results):
    print("Accuracy =", accuracy(results))
    print("Precision =", precision(results))
    print("Recall =", recall(results))
    print("True Negative Rate =", tn_rate(results))
```

Для тестирования использовалось 4 разных функции классификации.

1. Алгоритм из задания: объект классифицируется как +, если у всех его пересечений с объектами из + контекста минимально длины min_cdr не более max_cntr контрпримеров из – контекста.
2. Второй алгоритм работает по принципу большинства. Для каждого элемента тестовой части происходит голосование. Считается количество совпадений, не менее заданной длины и не имеющих пересечений с другим контекстом, с плюс и минус контекстом. Которых больше - к тому классу и причисляется элемент.
3. Очевидно, что второй алгоритм работает не очень хорошо, если какого-то класса сильно больше, чем другого. поэтому третий алгоритм решает эту проблему. Для контекста считается отношение похожих элементов к его размеру.
4. Теперь для каждого пересечения тестового элемента и элемента контекста считается его отношение к размеру контекста, данные отношения суммируются и опять делятся на размер контекстов.

Для каждого из алгоритмов перебирались параметры, чтобы добиться его наиболее точной работы. Для первого алгоритма менялось количество контрпримеров и длина пересечения. В остальных алгоритмах менялась только длина пересечения. Ниже в табличке указывается алгоритм, его настройки и метрика accuracy, которые давали достигались при лучшем результате:

	название	Max_cntr	Min_cdr	accuracy
1	Fca_example	19500	2	0.5157
2	Fca_abs	-	7	0.9789
3	Fca__relative	-	5	0.9894
4	Fca_partial	-	1	0.9473

Из этой таблицы видно, что первый алгоритм сильно проигрывает остальным. Результаты четвертого хуже чем 2 и 3.

3 и 2 показывают лучшие результаты, потому что, на мой взгляд, они лучше всего классифицируют объект. Однако 3 сглаживает разницу почти в 2 раза между контекстами (567+ 296-).