лабораторная работа   
"изучение алгоритма apriori поиска ассоциативных правил"

**Цель работы**: знакомство с задачей поиска ассоциативных правил и алгоритмами ее решения на примере алгоритма *Apriory*.

**Задачи**:

– изучить задачу поиска ассоциативных правил, основные понятия и определения;

– изучить двухэтапный алгоритм *Apriory*: генерация часто встречающихся наборов и выделение ассоциативных правил;

– изучить реализацию алгоритма *Apriory* на языке *Python*;

– выбрать базы данных для тестирования алгоритма *Apriory* (каталог *data\arules*);

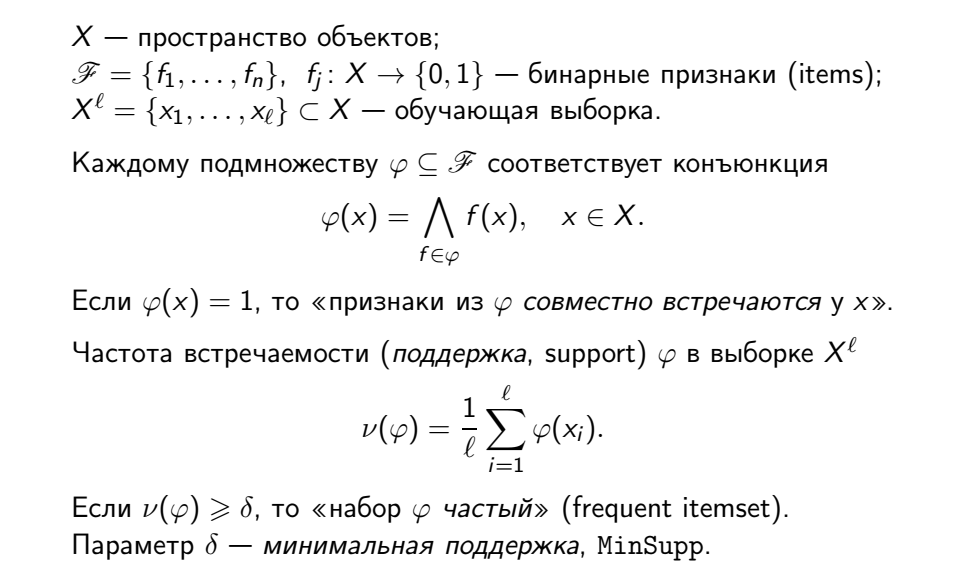
– реализовать алгоритм *Apriory* (любыми программными средствами) или использовать существующую реализацию;

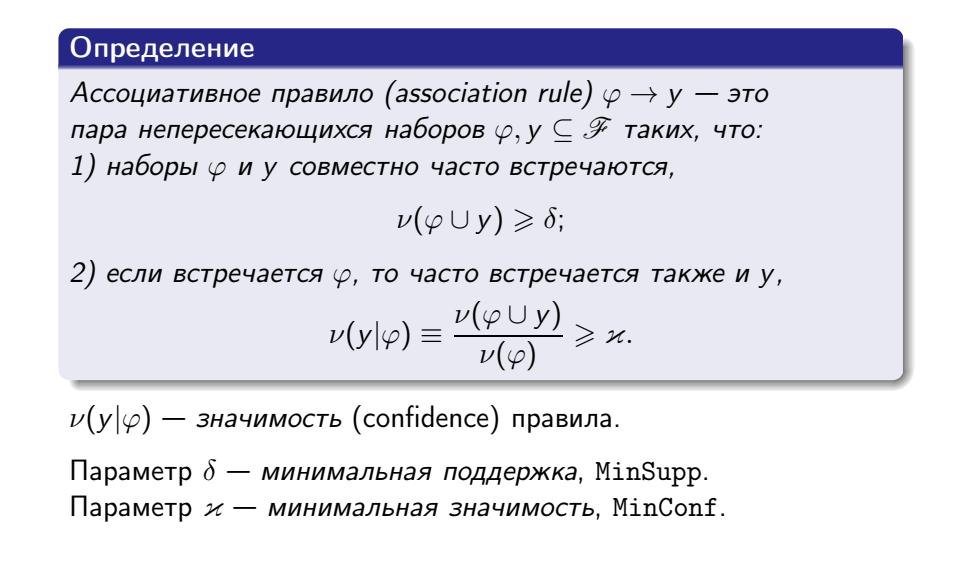
– реализовать функции загрузки данных для выбранной базы данных;

– провести тестирование алгоритма *Apriory*: получить ассоциативные правила; изучить время вычислений и количество часто встречающихся наборов в зависимости от величины поддержки; изучить количество правил в зависимости от значения значимости;

– реализовать вычисление дополнительных мер – характеристик интересности правила: поддержка (*Lift*, *Interest*, *Correlation*), убедительность (*Conviction*), влияние (*Leverage*, *Piatetsky*-*Shapiro*), покрытие (*Coverage*).

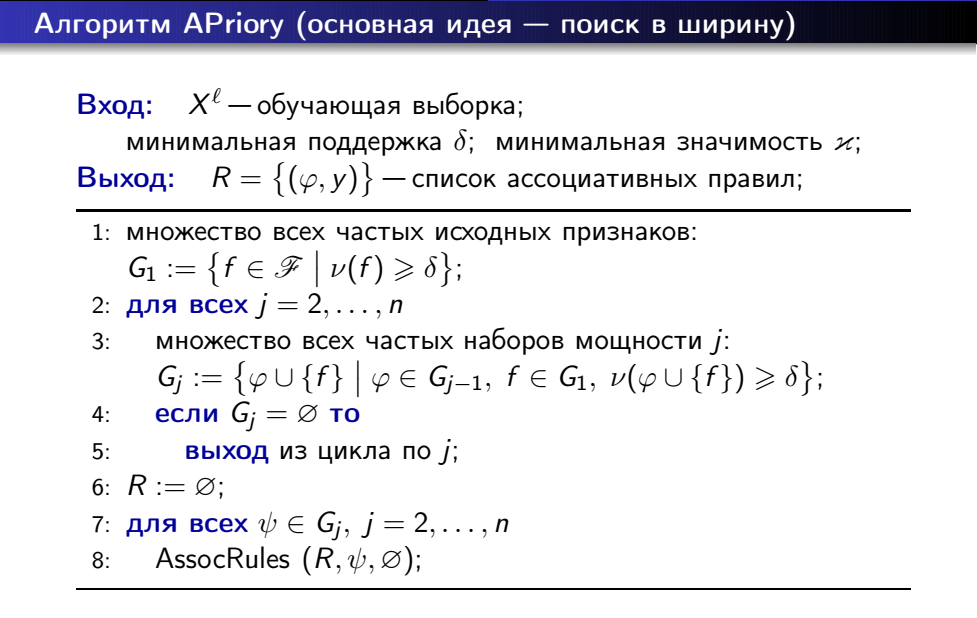
# Основные определения и обозначения



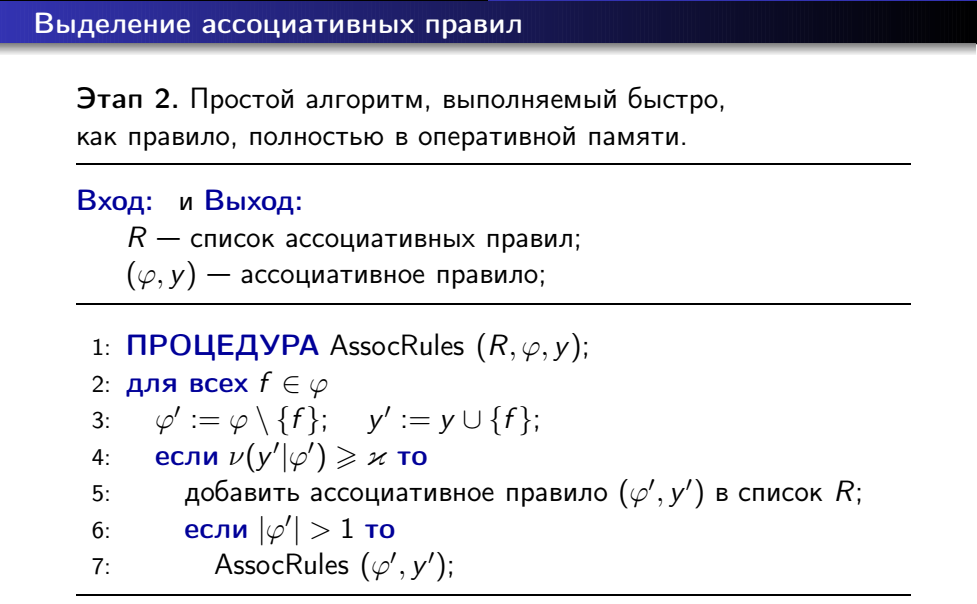


# Алгоритм Apriory

## Поиск часто встречающихся наборов



## Выделение ассоциативных правил



# пример реализации алгоритма apriory

Реализация алгоритма на языке Python представлена следующим набором классов:

– transaction: транзакция, имеющая идентификатор и содержащая набор элементов;

– database: база данных, представляющая собой список транзакций;

– rule: класс для работы с ассоциативными правилами;

– apriory: класс, инкапсулирующий алгоритм поиска ассоциативных правил.

## Класс transaction

Свойства класса:

– tid: идентификатор транзакции;

– itemset: набор элементов в транзакции, тип – frozenset;

Методы класса:

–\_\_init\_\_ – конструктор класса;

– get\_list – получение списка элементов;

– get\_boolean – получение списка признаков присутствия элементов в транзакции.

class transaction:

    '''

    Транзакция, имеющая идентификатор и содержащая набор элементов

    '''

    def \_\_init\_\_(self, tid, itemset=frozenset()):

        self.tid = tid

        self.itemset = itemset

    def get\_list(self, fields):

        return list(self.itemset)

    def get\_boolean(self, fields):

        digit = lambda value: "1" if value in self.itemset else "0"

        return [digit(item) for item in fields]

## Класс database

Свойства класса:

– fields – список полей базы данных;

– transactions – список транзакций;

Методы класса:

– \_\_init\_\_ – конструктор класса;

– add\_transaction – добавление транзакции в базу данных;

– print\_as\_set – печать в формате "множество элементов"

– print\_as\_boolean – печать в формате "булевый набор принзаков"

– calc\_support – вычисление поддержки набора;

– load – загрузка данных.

class database:

    '''

    База данных, состоящая из транзакций

    '''

    def \_\_init\_\_(self, fields=[], transactions=[]):

        self.fields = fields

        self.transactions = transactions

    def add\_transaction(self, item):

        self.transactions.append(item)

    def print\_as\_set(self):

        for t in self.transactions:

            print(t.tid, " ".join(t.get\_list(self.fields)))

        print()

    def print\_as\_boolean(self):

        print("#", " ".join(self.fields))

        for t in self.transactions:

            print(t.tid, " ".join(t.get\_boolean(self.fields)))

        print()

    def calc\_support(self, itemsets, support):

        count = {itemset:0 for itemset in itemsets}

        for transaction in self.transactions:

            for itemset in itemsets:

                if itemset <= transaction.itemset:

                    count[itemset] += 1

        total = len(self.transactions)

        for itemset in itemsets:

            support[itemset] = count[itemset] / total

    def load(self, file\_name):

        pass

## Класс rule

Свойства класса:

– antecedent – левая часть импликации, предпосылка;

– consequent – правая часть импликации, следствие;

– support – значение подержки;

– confidence – значение значимости.

Методы класса:

– \_\_init\_\_ – конструктор класса;

– \_\_str\_\_ – преобразование в строку;

– description – отображение описания правила в терминах полей базы данных.

class rule:

    def \_\_init\_\_(self, antecedent, consequent, support, confidence):

        self.antecedent = antecedent

        self.consequent = consequent

        self.support = support

        self.confidence = confidence

    def \_\_str\_\_(self):

        return f"{list(self.antecedent)} => {list(self.consequent)}"

    def description(self, names):

        ant = [names[item] for item in list(self.antecedent)]

        cons = [names[item] for item in list(self.consequent)]

        return f"{ant} => {cons}"

## Класс apriori

Свойства класса:

– database – используемая база данных;

– min\_support – минимальная поддержка;

– min\_confidence – минимальная достоверность;

– itemsets – список наборов;

– support – словарь, хранящий значение поддержки набора;

– rules – список сгенерированных правил.

Методы класса:

– \_\_init\_\_ – конструктор класса;

– filter – формируем список наборов с поддержкой, большей min\_support

– step\_0 – формируем одноэлементные наборы, рассчитываем их поддержку;

– step\_k – формируем наборы следующего уровня, рассчитываем их поддержку;

– generate\_rules – генерация ассоциативных правил;

– run – запуск алгоритма;

–print\_itemsets – печать наборов;

– print\_support – вывод значения поддержки;

– print\_rules – печать ассоциативных правил;

–print\_description – печать ассоциативных правил с описанием полей;

– itemset\_count – расчет общего количества наборов.

class apriori:

    def \_\_init\_\_(self, database, min\_support, min\_confidence):

        self.database = database

        self.min\_support = min\_support

        self.min\_confidence = min\_confidence

        self.itemsets = []

        self.support = {}

        self.rules = []

    def filter(self, items, support):

        # формируем список наборов с поддержкой, большей min\_support

        return [item for item in items if support[item] >= self.min\_support]

    def step\_0(self):

        self.itemsets = []

        self.support = {frozenset():-1}

        self.rules = []

        # формируем одноэлементные наборы

        items = [frozenset([item]) for item in self.database.fields]

        # рассчтываем поддержку одноэлементных наборов

        self.database.calc\_support(items, self.support)

        # добавляем в список

        self.itemsets.append(self.filter(items, self.support))

        return self.itemsets[-1] != []

    def step\_k(self):

        if self.itemsets[-1] == []:

            return False

        items = []

        for e1 in self.itemsets[0]:

            for e2 in self.itemsets[-1]:

                if not e1 <= e2:

                    s = frozenset(e2.union(e1))

                    if not s in items:

                        items.append(s)

        # рассчтываем поддержку наборов

        self.database.calc\_support(items, self.support)

        # добавляем в список

        self.itemsets.append(self.filter(items, self.support))

        # возвращаем признак того, что можно продолжать

        return self.itemsets[-1] != []

    def generate\_rules(self):

        # перебрать все наборы из itemsets, начиная с двухэлементых

        for level in self.itemsets[1:]:

            for itemset in level:

                # генерируем все подмножества множества itemset

                subsets = get\_subsets(itemset)

                # формируем посыл и следствие правила

                for antecedent in subsets:

                    consequent = itemset.difference(antecedent)

                    # рассчитать поддержку и достоверность

                    rule\_support = self.support[itemset]

                    rule\_confidence = rule\_support / self.support[antecedent]

                    # если достоверность больше пороговой, добавить в список

                    if rule\_confidence > self.min\_confidence:

                        self.rules.append(rule(antecedent, consequent, rule\_support, rule\_confidence))

        self.rules.sort(key=lambda rule: rule.confidence)

    def run(self, debug=False):

        if debug:

            print('Runnung Apriori:')

        if self.step\_0():

            if debug:

                print('.')

            while self.step\_k():

                if debug:

                    print('.')

            self.generate\_rules()

            if debug:

                print('.done!')

    def print\_itemsets(self):

        k = 0

        for level in self.itemsets:

            print('level', k)

            for item in level:

                print(list(item))

            k += 1

        print()

    def print\_support(self):

        for key in self.support.keys():

            print(list(key), f"{self.support[key]:.2}")

    def print\_rules(self, top=0):

        for item in self.rules[:top]:

            print(item, f"supp = {item.support:.2} conf = {item.confidence:.2}")

    def print\_description(self, top=0, names={}):

        for item in self.rules[:top]:

            print(item.description(names), f"supp = {item.support:.2} conf = {item.confidence:.2}")

    def itemset\_count(self):

        return sum([len(item) for item in self.itemsets])

Пример использования классов на тестовом наборе данных:

# инициализация объекта базы данных списком полей

    db = database(['A','B','C','D','E','F'])

    # создаем транзакции

    t1 = transaction(0, frozenset(['A','B','C']))

    t2 = transaction(1, frozenset(['A','C']))

    t3 = transaction(2, frozenset(['A','D']))

    t4 = transaction(3, frozenset(['B','E','F']))

    # добавляем транзакции в базу данных

    db.add\_transaction(t1)

    db.add\_transaction(t2)

    db.add\_transaction(t3)

    db.add\_transaction(t4)

    # выводим базу данных

    db.print\_as\_set()

    db.print\_as\_boolean()

    # инициализируем и запускаем алгоритм Apriori

    alg = apriori(db, 0.30, 0.50)

    alg.run()

    # печатаем результаты

    alg.print\_itemsets()

    alg.print\_support()

    alg.print\_rules()

Результат выполнения программного кода:

0 B C A

1 C A

2 D A

3 B F E

# A B C D E F

0 1 1 1 0 0 0

1 1 0 1 0 0 0

2 1 0 0 1 0 0

3 0 1 0 0 1 1

level 0

['A']

['B']

['C']

level 1

['C', 'A']

level 2

[] -1

['A'] 0.75

['B'] 0.5

['C'] 0.5

['D'] 0.25

['E'] 0.25

['F'] 0.25

['B', 'A'] 0.25

['C', 'A'] 0.5

['C', 'B'] 0.25

['C', 'B', 'A'] 0.25

['A'] => ['C'] supp = 0.5 conf = 0.67

['C'] => ['A'] supp = 0.5 conf = 1.0