INSTITUTO FEDERAL MINAS GERAIS (IFMG) - CAMPUS BAMBUÍ
Mineração de Dados
Prof. Marcos Roberto Ribeiro

Lista de Exercícios 06

Exercício 1:
Por que é interessante que um conjunto de regras seja disjunto?

Exercício 2:
Quais os problemas das árvores de decisão que podem ser minimizados com o uso de regras de decisão?

Exercício 3:
Descreva a ideia básica do algoritmo de cobertura.

Exercício 4:
Diferencie as abordagens *top-down* e *bottom-up*

Exercício 5:
Considere o código de um classificador baseado em regras disponível no Apêndice A.

(a) Crie a classe **RulesTopDown** herdando da classe **Rules** para implementar a abordagem *top-down*. Além do método abstrato (**_learn_rule**), considere a implementação do seguinte método:

_expand_rule(data, class_value, rule): Expande uma regra (**rule**) considerando os registros de dados (**data**) e o valor de classe (**class_rule**).

(b) Crie a classe **RulesBottomUp** herdando da classe **Rules** para implementar a abordagem *bottom-up*. Além do método abstrato (**_learn_rule**), considere a implementação do seguintes métodos:

_rule_from(data, class_value): Cria uma regra para a classe **class_value** com base em um registro aleatório dos dados (**data**);

_reduce_rule(data, class_value, rule): Tenta reduzir a regra (**rule**) com base nos dados (**data**) e valor da classe (**class_value**).

Exercício 6:
Considere a fórmula $A = (p \wedge q) \vee (r \wedge s)$.

(a) Construa uma árvore de decisão para obter o resultado de A;

(b) Utilize o algoritmo de cobertura para construir um conjunto de regras para resolver A;

(c) Extraia as regras da árvore de decisão e compare com as regras geradas pelo algoritmo de cobertura.

Exercício 7:

Considere a base de dados de *Jogos* mostrada na Figura 1. Compare a precisão dos algoritmos implementados com os algoritmos baseados em regras disponíveis na ferramenta Weka.

Aparência	Temperatura	Umidade	Vento	Jogo
sol	quente	alta	falso	não
sol	quente	alta	verdade	não
encoberto	quente	alta	falso	sim
chuvoso	agradável	alta	falso	sim
chuvoso	frio	normal	falso	sim
chuvoso	frio	normal	verdade	não
encoberto	frio	normal	verdade	sim
sol	agradável	alta	falso	não
sol	frio	normal	falso	sim
chuvoso	agradável	normal	falso	sim
sol	agradável	normal	verdade	sim
encoberto	agradável	alta	verdade	sim
encoberto	quente	normal	falso	sim
chuvoso	agradável	alta	verdade	não

Figura 1: Dados para determinar se haverá jogo

Referências

FACELI, K. et al. **Inteligência artificial**: uma abordagem de aprendizagem de máquina. Rio de Janeiro: LTC, 2011.

Apêndice A Classificador baseado em regras

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from abc import abstractmethod
from classifier import Classifier

class Rules(Classifier):
    '''
    Simple decision tree classifier
    '''

    def __init__(self, csv_file, debugging):
        # Herda da classe pai
        Classifier.__init__(self, csv_file, debugging)
        # Nenhuma regra antes do treinamento
        self._rules_list = []

    def fit(self):
        self._build_rules()

    def _remove_covered(self, data, rule):
        '''
        Remove os registros cobertos por rule
        '''
        # Condições da regra
        rule_cond = rule[:-1]
        # Testa se há condições
        if rule_cond:
            # Lista de filtros baseda na regra
            filter_list = [cond[0] + '!="' + str(cond[1]) + '"'
                           for cond in rule_cond]
            # Retorna dados filtrados
            return data.query(' | '.join(filter_list))
        # Todos os registros são cobertos pela condição vazia
        return data[:0]

    def _build_rules(self):
        '''
        Algoritmo de cobertura
        '''
        self._debug('Buiding rules...')
        # Inicia com uma lista vazia de regras
        self._rules_list = []
        # Cópia dos dados
        data = self._data[:]
        # Contagem dos valores de classe (mais frequentes primeiro)
        class_count = data[self._class_att].value_counts()
        # Classe majoritária
        majoritary_class = class_count.index[0]
        while len(data) > 0:
            # Contagem dos valores de classe (mais frequentes primeiro)
            class_count = data[self._class_att].value_counts()
```

```
            # Pega o valor de classe mais frequente
            class_value = class_count.index[0]
            self._debug('Try to build rule for class: %s', class_value)
            # Aprende uma regra
            rule = self._learn_rule(data, class_value)
            if len(rule) > 1:
                self._debug('New rule learned: %s', rule)
                # Adiciona a regra à lista
                self._rules_list.append(rule)
            # Remove os registros cobertos pela regra
            data = self._remove_covered(data, rule)
        # Regra com condição vazia para a classe majoritária
        rule = [majoritary_class]
        self._debug('New rule learned: %s', rule)
        self._rules_list.append(rule)

    @abstractmethod
    def _learn_rule(self, data, class_value):
        '''
        Aprende uma regra
        '''
        pass

    def _get_covered(cls, data, rule):
        '''
        Retorna os registros cobertos por rule
        '''
        # Condição da regra
        rule_cond = rule[:-1]
        # Testa se há condições na regra
        if rule_cond:
            # Listra de filtros
            filter_list = [cond[0] + '=="' + str(cond[1]) + '"'
                           for cond in rule_cond]
            # Retorna dados filtrados
            return data.query(' & '.join(filter_list))
        # Retorna dados sem filtrar
        return data


    def _aval(self, data, rule):
        '''
        Avaliação da regra usando a métrica de Laplace
        '''
        # Número de classes
        class_count = len(data[self._class_att].value_counts())
        # Dados cobertos pela regra
        data_covered = self._get_covered(data, rule)
        # Dados classificados corretamente
        data_correct = data_covered[data_covered[self._class_att]==rule[-1]]
        # Número de dados cobertos
        covered = len(data_covered)
        # Número de dados classificados coretamente
        correct = len(data_correct)
        # Retorna a métrica de Laplace
```

```
        return (correct + 1) / (covered + class_count)

    def _is_covered(self, record, rule):
        '''
        Testa se um record é coberto por rule
        '''
        # Para cada condição da regra
        for cond in rule[:-1]:
            # Obtém atributo e seu valor
            att = cond[0]
            value = cond[1]
            # Testa se o registro possui valor diferente
            if record[att] != value:
                # Se tiver, não é coberto e interrompe o laço
                return False
        # Retorna verdadeiro (todas as condições foram atendidas)
        return True

    def print_rules(self):
        '''
        Lista regras
        '''
        # Para cada regra
        for rule in self._rules_list:
            # Condições da regra
            cond_list = rule[:-1]
            if len(cond_list) > 0:
            # Condições no formato (Atributo = Valor)
                str_cond_list = [cond[0] + ' = ' + str(cond[1])
                                 for cond in cond_list]
                # Escreve a condição
                print('IF', ' AND '.join(str_cond_list), 'THEN ', end='')
            # Escreve a classe
            print(self._class_att + ' = ' + str(rule[-1]))

    def classify_record(self, record):
        '''
        Classify a record
        '''
        # Para cada regra
        for rule in self._rules_list:
            # Testa se o registro é coberto pela regra
            if self._is_covered(record, rule):
                # Retorna a classe da regra (último elemento)
                return rule[-1]
        # Se nenhuma regra cobrir, retorna a classe majoritária
        return self._majoritary_class
```