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

Lista de Exercícios 07

Exercício 1:
Diferencie classificadores ávidos de classificadores preguiçosos.

Exercício 2:
Como funciona o classificador de rota e quais seus problemas?

Exercício 3:
Por que é importante fazer a padronização ou normalização dos dados antes de utilizar classificadores que trabalham com distância?

Exercício 4:
Implemente uma rotina para ler os dados de um arquivo CSV, calcular o z-score dos mesmos e salvar o resultado em outro arquivo.

Exercício 5:
Discuta sobre a escolha do valor de k para o algoritmo KNN.

Exercício 6:
Considere a classe de um classificador genérico do Apêndice A. Implemente um classificador KNN herdando de **Classifier**. Durante a implementação implemente os métodos abstratos da classe pai, exceto o método **fit()**, e considere a criação do seguinte método:

_distance(rec1, rec2) : Calcula a distância entre **rec1** e **rec2**.

Observações:

- Na inicialização da classe, tenha como opção aplicar o *zscore* utilizando a rotina do Exercício 4 para normalizar os dados;
- Por que o método **fit()** não precisa ser implementado?

Exercício 7:
Calcule a precisão do algoritmo implementado na questão anterior para a base de dados *Iris*. Compare a precisão obtida com a precisão de alguns algoritmos disponíveis na ferramenta Weka.

Apêndice A Classificador genérico

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

import pandas as pd
import logging
from abc import abstractmethod


LOG = 'classifier.log'


class Classifier():
    '''
    Classifier template class
    '''
    def __init__(self, csv_file, debugging=False):
        # Flag de depuração
        self._debugging = debugging
        # Configura log
        self._config_log()
        # Lê arquivo CSV para _data
        self._data = pd.read_csv(csv_file, skipinitialspace=True)
        # Pega último atributo para classe
        self._class_att = str(self._data.columns[-1])
        # Demais atributos são atributos de dados
        self._att_list = list(self._data.columns[:-1])

    @abstractmethod
    def fit(self):
        '''
        Treinamento do classificador
        '''
        pass

    @abstractmethod
    def classify_record(self, record):
        '''
        Classifica um registro
        '''
        pass

    def _debug(self, msg, *args, **kwargs):
        '''
        Exibe mensagens de debug
        '''
        self._log.debug(msg, *args, **kwargs)

    def _config_log(self):
        '''
        Configura o log
        '''
        # Cria log
```

```
        self._log = logging.getLogger(LOG)
        # Formato de mensagens
        str_format = '%(levelname)s - %(message)s'
        log_format = logging.Formatter(str_format)
        # Arquivo
        file_handler = logging.FileHandler(LOG)
        file_handler.setFormatter(log_format)
        # Console
        console_hanler = logging.StreamHandler()
        console_hanler.setFormatter(log_format)
        self._log.addHandler(file_handler)
        self._log.addHandler(console_hanler)
        # Nível de log
        if self._debugging:
            self._log.setLevel(logging.DEBUG)
        else:
            self._log.setLevel(logging.INFO)

    def disable_debug(self):
        '''
        Desabilita o debug
        '''
        self._log.setLevel(logging.INFO)

    def precision(self, test_data):
        '''
        Calcula a precisão considerando os dados de test_data
        '''
        # Contagem de erros
        errors = 0
        # Para cada registro de dado
        for _, rec in test_data.iterrows():
            # Classifica o registro
            class_pre = self.classify_record(rec)
            # Compara com a classe correta do registro
            if class_pre != rec[self._class_att]:
                # Se for diferente conta o erro
                errors += 1
        # Retorna a porcentagem de acertos
        return (len(test_data) - errors) / len(test_data)

    def training_precision(self):
        '''
        Calcula a precisão usando todo o conjunto de treinamento
        '''
        return self.precision(self._data)

    def k_fold_precision(self, k):
        '''
        Calcula a precisão usando validação cruzada
        '''
        # Tamanho de cada partição
        fold_len = len(self._data) // k
        # Soma da precisão para fazer a média
        precision = 0
```

```
        # Backup dos dados originais
        bkp_data = self._data
        # Para cada partição
        for cont in range(k):
            # Calcula o início e fim dos dados de teste
            fold_start = cont*fold_len
            fold_end = (cont+1)*fold_len
            # Selciona os dados de teste
            test_data = bkp_data[fold_start:fold_end]
            # Os dados de treinamento são os demais dados
            train_data = bkp_data[:fold_start].append(bkp_data[fold_end+1:])
            # Atribui os dados de teste ao classificado
            self._data = train_data
            # Faz o treinamento
            self.fit()
            # Soma a precisão
            precision += self.precision(test_data)
        # Retorna a média das precisões
        return precision / k
```