Universidade Federal do Agreste de Pernambuco

Bacharelado em Ciências da Computação

Prof. Tiago Buarque A. de Carvalho

Aprendizagem de Máquina:

Pré-processamento de Dados

Aluno: Vinícius Santos de Almeida

- 1. a) A melhor técnica de conversão de atributos categóricos para numéricos vai depender do tipo do atributo. Segue abaixo a melhor forma de lidar com cada um dos atributos da base:
    - **Binário, seja ordinal ou nominal**, todos os atributos abaixo se caracterizam como atributos binários, ou seja, podem assumir apenas dois valores na base e por isso podem ser convertidos para para forma binário {0,1} ou {-1, 1}:
        - school
        - sex
        - address
        - famsize
        - pstatus
        - schoolsup
        - famsup
        - paid
        - activities
        - nursery
        - higher
        - internet
        - romantic
    - **Não binário nominal**, todos os atributos abaixo podem ter seus valores convertidos em N colunas, dependendo da quantidade nominal de valores que o atributo pode assumir:
        - Mjob
        - Fjob
        - reason
        - guardian
    - As seguinte colunas já estão em formato numérico.
        - age
        - Medu
        - Fedu
        - traveltime
        - studytime
        - failures
        - famrel

    - freetime
    - goout
    - Dalc
    - Walc
    - health
    - absences
    - G1
    - G2
    - G3

- 1. b)
    - Converti para numéricos todos os atributos da questão **1(a)** listados em Binário e Não binário nominal, os demais atributos já estavam em formato numérico.
    - Ver arquivo `data/questao_1_b_export.csv` com a base com todos seus atributos em numéricos.
    - Implementação:

```
from utils import load_data, export_data

class BinaryConverter():

    def __init__(self, data, columns):
        self.data = data
        self.columns = columns

    def values_to_binary(self):
        possible_values = {}
        for key in self.columns:
            values = set()
            for r in self.data:
                values.add(r[key])

            possible_values[key] = values

        new_values = {}
        for key, value in possible_values.items():
            new_values[key] = dict(zip(value, (0,1)))

        return new_values

    def convert_data(self):
        b_values = self.values_to_binary()

        for index, row in enumerate(self.data):
            row_converted = row
            for key, value in row.items():
                if key in self.columns:
                    row_converted[key] = b_values[key][value]
                else:
                    row_converted[key] = value
```

```
            self.data[index] = row_converted

        return self.data

class NonBinaryConverter():

    def __init__(self, data, columns):
        self.data = data
        self.columns = columns
        print()

    def generate_columns(self):
        columns = []
        for column in self.data[0]:
            columns.append(column)
        return columns

    def new_columns(self):
        new_columns = {}
        for key in self.columns:
            values = set()
            for r in self.data:
                values.add(r[key])

            new_columns[key] = values

        return new_columns

    def convert_data(self):
        new_columns = self.new_columns()

        new_data = []
        for index, row in enumerate(self.data):
            row_converted = {i: v for i, v in row.items()}
            for key, value in row.items():
                if key in self.columns:
                    for column in new_columns[key]:
                        new_col_name = f'{key}_{column}'
                        if value == column:
                            row_converted[new_col_name] = 1
                        else:
                            row_converted[new_col_name] = 0
            for column in self.columns:
                del row_converted[column]
            new_data.append(row_converted)

        self.data = new_data
        return self.data

def main():
    data, _ = load_data('./data/student-mat.csv')

    binary_converter = BinaryConverter(data,
['sex','school','address','famsize','Pstatus','schoolsup','famsup
```

```
','paid','activities','nursery','higher','internet','romantic'])
    data = binary_converter.convert_data()

    nonbinary_converter = NonBinaryConverter(data,
['Mjob','Fjob','reason','guardian'])
    data = nonbinary_converter.convert_data()

    print('Base com atributos convertidos para numéricos: ')
    print(tuple(data[0].keys()))
    [print(tuple(row.values())) for row in data]
    export_data('./data/questao_1_b_export.csv', data,
nonbinary_converter.generate_columns())

if __name__ == '__main__':
    main()
```

- 1. c) Para converter a coluna G3 para valores binários, é necessário usar a técnica limiar para conversão de atributos numéricos para categóricos, para isso vou assumir dois intervalos:
  - `x >= 14 && x <= 20`, aluno na média ou acima dela.
  - `x >= 0 && x < 14`, aluno abaixo da média.
  - Os resultados da conversão podem ser conferidos em `data/questao_1_c_export.csv`.
  - Implementação:

```
    from utils import load_data, export_data

    class BinaryCategorizer():

        def __init__(self, data: list[dict], columns: list,
intervals: tuple[tuple]):
            '''
                - interval: dict = {'value1': 0, 'value2': 0,
'value3': 1, 'value4': 1}
            '''
            self.data = data
            self.columns = columns
            self.intervals = intervals

        def categorize(self, value):
            return self.intervals[value]

        def convert_data(self):
            categorized = []
            for row in self.data:
                new_row = {k:v for k,v in row.items()}
                for col in self.columns:
                    new_row[col] = self.categorize(new_row[col])
                categorized.append(new_row)
            self.data = categorized
            return self.data
```

```
    def main():
        data, columns =
load_data('./data/questao_1_b_export.csv')

        intervals = {str(v): (0 if v < 14 else 1) for v in
range(0,21)}
        binary_categorizer = BinaryCategorizer(data, ['G3'],
intervals)
        data = binary_categorizer.convert_data()
        print(tuple(data[0].keys()))
        [print(tuple(row.values())) for row in data]
        export_data('./data/questao_1_c_export.csv', data,
columns)

    if __name__ == '__main__':
        main()
```

- 1. d) O intervalo de confiança é `[0.86, 0.93]`.
    - Implementação do classificador:

```
import numpy as np

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_fscore_support as
score
from sklearn.metrics import accuracy_score


class KNeighborsKFoldClassifier():

    def __init__(self, X, y, n_neighbors=1):
        self.X = X
        self.y = y
        self.n_neightbors = n_neighbors

    def load_data(self, filename):
        self.X = []
        self.y = []
        with open(filename, 'r') as file:
            for row in file.readlines():
                row_list = row.rstrip('\n').split(',')
                self.X.append([float(x) for x in row_list[:4]])
                self.y.append(row_list[4])
        self.X = np.array(self.X)
        self.y = np.ravel(self.y)

    def classify(self):
        self.accuracy_score = np.array([])
        for i in range(100):
            X_train, X_test, y_train, y_test =
```

```
train_test_split(self.X, self.y, test_size=.5)
            knn = KNeighborsClassifier(n_neighbors=1,
weights="distance", metric="euclidean")
            knn.fit(X_train, y_train)

            predicts = knn.predict(X_test)
            accuracy = accuracy_score(y_test, predicts)
            self.accuracy_score = np.append(self.accuracy_score,
accuracy)

    def intervalo_confianca(self):
        media = np.average(self.accuracy_score)
        desvio = self.accuracy_score.std()
        intervalo_confianca = (media-(1.96*desvio), media+
(1.96*desvio))
        return intervalo_confianca
```

- Implementação da questão:

```
import numpy as np

from my_kneighbors_kfold_classifier import
KNeighborsKFoldClassifier
from utils import load_data

data, columns = load_data('./data/questao_1_c_export.csv')
X = np.array([tuple([int(x) for x in list(row.values())[:28]]) +
tuple([int(x) for x in list(row.values())[29:46]]) for row in
data])
y = np.ravel([int(row['G3']) for row in data])
knn_kfold = KNeighborsKFoldClassifier(X, y)
knn_kfold.classify()
print(f'Intervalo de confiança:
{knn_kfold.intervalo_confianca()}')
```

- 2. a)
  - **Dados cíclicos**, pois são dados que representam informações cíclicas, que vão e se repetem sem existir um valor inicial ou um valor final, como dias da semana, meses do ano, horas do dia, minutos da hora etc. segue abaixo os atributos que podem ser convertidos para numéricos por essa técnica:
    - month
    - day
  - Os demais atributos, listados abaixo, já estão no formato numérico:
    - X
    - Y
    - FFMC
    - DMC
    - DC

    - ISI
    - temp
    - RH
    - wind
    - rain
    - area

- 2. b)
    - O base de dados convertida assim como solicitado está em `data/questao_2_b_export.csv`.
    - Implementação:

```
import numpy as np
import math

from utils import export_data, load_data

WEEK_DAYS = {
'sun': 1,
'mon': 2,
'tue': 3,
'wed': 4,
'thu': 5,
'fri': 6,
'sat': 7,
}

MONTHS = {
'jan': 1,
'feb': 2,
'mar': 3,
'apr': 4,
'may': 5,
'jun': 6,
'jul': 7,
'aug': 8,
'sep': 9,
'oct': 10,
'nov': 11,
'dec': 12,
}

class CyclicConverter():

    def __init__(self, data, columns):
        self.data = data
        self.columns = columns

    def order_cyclic_table(self):
        for col in self.columns:
            if col == 'month':
```

```
                ordered = {}
                for month in self.cyclic_table[col]:
                    ordered[month] = MONTHS[month]
                self.cyclic_table[col] = ordered
            if col == 'day':
                ordered = {}
                for day in self.cyclic_table[col]:
                    ordered[day] = WEEK_DAYS[day]
                self.cyclic_table[col] = ordered

    def create_cyclic_table(self):
        self.cyclic_table = {}
        for col in self.columns:
            possible_values = list(set(row[col] for row in
self.data))
            self.cyclic_table[col] = possible_values

        self.order_cyclic_table()
        return self.cyclic_table

    def convert_data(self):
        self.create_cyclic_table()

        for index, row in enumerate(self.data):
            new_row = {k:v for k,v in row.items()}
            for key, value in row.items():
                if key in self.columns:
                    new_row[f'{key}_sin'] =
CyclicConverter.get_sin_attr(self.cyclic_table[key][value],
len(self.cyclic_table[key]))
                    new_row[f'{key}_cos'] =
CyclicConverter.get_cos_attr(self.cyclic_table[key][value],
len(self.cyclic_table[key]))
                    del new_row[key]
            self.data[index] = new_row
        return self.data

    def get_new_columns(self, columns):
        new_columns = [col for col in columns if col not in
self.columns]
        for col in columns:
            if col in self.columns:
                new_columns.append(f'{col}_sin')
                new_columns.append(f'{col}_cos')
        return new_columns

    @staticmethod
    def get_sin_attr(index, length):
        return round(math.sin(2 * math.pi * index / length),2)

    @staticmethod
    def get_cos_attr(index, length):
        return round(math.cos(2 * math.pi * index / length),2)
```

```
def main():
    data, columns = load_data('./data/forestfires.csv', sep=',')
    cyclic_converter = CyclicConverter(data, ['month', 'day'])
    data = cyclic_converter.convert_data()

    print(tuple(data[0].keys()))
    [print(tuple(row.values())) for row in data]
    export_data('./data/questao_2_b_export.csv', data,
cyclic_converter.get_new_columns(columns))


if __name__ == '__main__':
    main()
```

- 3. a) Os atributos a seguir podem ser convertidos para numéricos pela técnica de **não binário ordinal** pois seus valores dão ideia de order, a maioria do baixo (low) ao muito alto (v-high), em outros casos, aqueles que já tinham alguns dados numéricos porém podiam assumir um valor categórico (5-more, more) pode convertê-lo para numérico ordinal da mesma forma:

    - buying
    - maint
    - doors
    - persons
    - lug_boot
    - safety

    * A classe é categórica e também pode ser convertida para numérica pela técnica **não binário ordinal**.

- 3. b)

    - O resultado da conversão pode ser conferido em `data/questao_3_b_export.csv`.
    - Implementação:

```
import numpy as np
import math

from utils import export_data, load_data

class OrdinalConverter():

    def __init__(self, data, columns):
        self.data = data
        self.columns = columns

    def convert_data(self):
        for index, row in enumerate(self.data):
            new_row = {k:v for k,v in row.items()}
            for key, value in row.items():
                if key in self.columns:
```

```
                    new_row[key] = self.columns[key][value]
            self.data[index] = new_row
        return self.data

    @staticmethod
    def get_sin_attr(index, length):
        return round(math.sin(2 * math.pi * index / length),2)

    @staticmethod
    def get_cos_attr(index, length):
        return round(math.cos(2 * math.pi * index / length),2)

def main():
    data, columns = load_data('./data/car.csv', sep=',')
    ordinal_converter = OrdinalConverter(data,
        {
            'buying': {'vhigh': 3, 'high': 2, 'med': 1, 'low':
0},
            'maint': {'vhigh': 3, 'high': 2, 'med': 1, 'low': 0},
            'doors': {'2': 0, '3': 1, '4': 2, '5more': 3},
            'persons': {'2': 0, '4': 1, 'more': 2},
            'lug_boot': {'small': 0, 'med': 1, 'big': 2},
            'safety': {'low': 0, 'med': 1, 'high': 2},
            'class': {'unacc': 0, 'acc': 1, 'good': 2, 'vgood':
3}
        })
    data = ordinal_converter.convert_data()

    print(tuple(data[0].keys()))
    [print(tuple(row.values())) for row in data]
    export_data('./data/questao_3_b_export.csv', data, columns)


if __name__ == '__main__':
    main()
```

- 4. a) e b)
    - Intervalo de confiança: [0.45, 0.80].
    - Adaptei o algoritmo usado na semana anterior para fazer o 1nn dividido como solicitado com 100 repetições, como vai poder ver abaixo. O ponto mais importante do algoritmo são as seguintes funções:
        - `missing_cols` que percorre toda a base determinando as colunas que tem algum valor faltante.
        - `missing_cols_avg` que percorre a saída de `missing_cols`, ou seja, percorre cada coluna que não tem dado, salvando os valores existentes e que no final retorna um dicionário com a média para cada coluna.
        - `fill_missing` que percorre os dados preenchendo onde estiver vazio com a média obtida na função `missing_cols_avg`.

* `fill_missing` é usada no treino e no teste, como solicitado.

** Eventualmente, dependendo do split, todos os valores de uma coluna estavam nulos, o que retorna um NaN para média desses valores, nesses casos coloquei 0, pois a biblioteca não aceita NaN ao fazer a classificação.

- Implementação:

```
import numpy as np

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold,
StratifiedShuffleSplit, train_test_split
from sklearn.metrics import precision_recall_fscore_support as
score
from sklearn.metrics import accuracy_score

from utils import load_data


class KNeighborsKFoldClassifier():

    def __init__(self, X, y, n_neighbors=1):
        self.X = X
        self.y = y
        self.n_neightbors = n_neighbors

    def load_data(self, filename):
        self.X = []
        self.y = []
        with open(filename, 'r') as file:
            for row in file.readlines():
                row_list = row.rstrip('\n').split(',')
                self.X.append([float(x) for x in row_list[:4]])
                self.y.append(row_list[4])
        self.X = np.array(self.X)
        self.y = np.ravel(self.y)

    @staticmethod
    def missing_cols(data):
        missing = set()
        for row in data:
            for k, v in enumerate(row):
                if np.isnan(v):
                    missing.add(k)
        return list(missing)

    @staticmethod
    def missing_cols_avg(missing_cols, data):
        missing_cols_dict = {}
        for col in missing_cols:
            values = np.array([])
```

```
            for row in data:
                if not np.isnan(row[col]):
                    values = np.append(values, row[col])
            missing_cols_dict[col] = np.average(values if
len(values) > 0 else [0])
        return missing_cols_dict

    def fill_missing(self, data):
        missing_cols =
KNeighborsKFoldClassifier.missing_cols(data)
        missing_cols_avg =
KNeighborsKFoldClassifier.missing_cols_avg(missing_cols, data)
        for i, row in enumerate(data):
            new_row = np.copy(row)
            for k, v in enumerate(row):
                if np.isnan(v):
                    new_row[k] = missing_cols_avg[k]
            data[i] = new_row
        return data

    def classify(self, k_neighbors=1, test_size=.5,
n_splits=100):
        sss = StratifiedShuffleSplit(n_splits=n_splits,
test_size=test_size)

        self.acertos = np.array([])
        for train_index, test_index in sss.split(self.X, self.y):
            X_train, X_test = self.X[train_index],
self.X[test_index]
            y_train, y_test = self.y[train_index],
self.y[test_index]
            X_train = self.fill_missing(X_train) # <----- letra a
            X_test = self.fill_missing(X_test) # <----- letra b
            knn = KNeighborsClassifier(n_neighbors=k_neighbors,
weights="distance", metric="euclidean")
            knn.fit(X_train, y_train.astype(float))

            acertos = knn.score(X_test, y_test)
            self.acertos = np.append(self.acertos, acertos)

    def intervalo_confianca(self):
        media = np.average(self.acertos)
        desvio = self.acertos.std()
        intervalo_confianca = (media-(1.96*desvio), media+
(1.96*desvio))
        return intervalo_confianca


def main():
    data, _ = load_data('./data/processed.hungarian.csv',
sep=',')
    X = np.array([tuple([int(x) if str(x).isnumeric() else np.NaN
for x in list(row.values())[:13]]) for row in data])
    y = np.ravel([int(row['num']) for row in data])
```

```
    knn_kfold = KNeighborsKFoldClassifier(X, y)
    knn_kfold.classify(test_size=.1)
    print(knn_kfold.intervalo_confianca())


if __name__ == '__main__':
    main()
```

- 5. a) e b)
    - Saída:
        - Intervalo confiança original: (0.6309764707967276, 0.7973381359448459)
        - Intervalo confiança intervalo [0,1]: (0.9058531997642713, 0.9905512946177514)
        - Intervalo confiança padronização: (0.907280078171628, 0.9868772251991588)
    - Conclusão:
        - Entre o original e intervalo [0,1] não há sobreposição dos intervalos, e por isso, é possível dizer que há diferença significativa entre os dois, tendo como prefência o classificador com intervalo, dando preferência ao de maior acurácia se isso for um fator determinante.
        - Entre o original e o com padronização também não há sobreposição, por isso, há diferença significativa entre eles, ainda dando preferência ao de maior acurácia se isso for um fator determinante.
        - Entre o intervalo [0,1] e o com padronização HÁ sobreposição e por isso não é possível determinar significativamente qual classifica melhor no quesito acurácia.
    - Criei um algoritmo, na classe `RescaleStdConverter` que é possui dois métodos que converte o conjunto de dados para as intervalo [0,1] e para padronização. Depois de geradas bases convertidas, uso a classe `KNeighborsKFoldClassifier` já usada em questões anteriores para classificar 1nn com 100 repetições 50/50.
    - Implementação (`KNeighborsKFoldClassifier` já foi definida em questões anteriores):

```
import numpy as np

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold,
StratifiedShuffleSplit, train_test_split
from sklearn.metrics import precision_recall_fscore_support as
score
from sklearn.metrics import accuracy_score

from my_kneighbors_kfold_classifier import
KNeighborsKFoldClassifier

from utils import load_data, export_data

class RescaleStdConverter():

    def __init__(self, data, columns):
        self.data = data
        self.columns = columns
```

```
    def map_columns_values(self):
        mapped_values = {}
        for col in self.columns:
            mapped_values[col] = np.array([])
            for row in self.data:
                for k,v in row.items():
                    if k not in mapped_values:
                        mapped_values[k] = np.array([float(v)])
                    else:
                        mapped_values[k] =
np.append(mapped_values[k], float(v))
        return mapped_values

    @staticmethod
    def calc_rescale(v, min, max):
        return round((v - min) / (max - min),2)

    def rescale_data(self):
        mapped_values = self.map_columns_values()
        self.rescaled_data = []
        for i, row in enumerate(self.data):
            new_row = {k:v for k,v in row.items()}
            for k, v in row.items():
                if k in self.columns:
                    max = mapped_values[k].max()
                    min = mapped_values[k].min()
                    new_row[k] =
RescaleStdConverter.calc_rescale(float(v), min, max)
                else:
                    new_row[k] = v
            self.rescaled_data.append(new_row)
        return self.rescaled_data

    @staticmethod
    def calc_std(v, avg, std):
        return round((v - avg) / std, 2)

    def standardize_data(self):
        mapped_values = self.map_columns_values()
        self.standardized_data = []
        for i, row in enumerate(self.data):
            new_row = {k:v for k,v in row.items()}
            for k, v in row.items():
                if k in self.columns:
                    avg = np.average(mapped_values[k])
                    std = mapped_values[k].std()
                    new_row[k] =
RescaleStdConverter.calc_std(float(v), avg, std)
                else:
                    new_row[k] = v
            self.standardized_data.append(new_row)
        return self.standardized_data
```

```
def main():
    data, columns = load_data('./data/wine.csv', sep=',')
    rescaler_sdt = RescaleStdConverter(data, columns[1:])

    rescaled = rescaler_sdt.rescale_data()
    export_data('./data/questao_5_scaled.csv', rescaled, columns)
    standardized = rescaler_sdt.rescale_data()
    export_data('./data/questao_5_std.csv', standardized,
columns)

    X = np.array([tuple([float(x) for x in list(row.values())
[1:]]) for row in data])
    y = np.ravel([float(row['class']) for row in data])
    knn_kfold = KNeighborsKFoldClassifier(X, y)
    knn_kfold.classify()
    print(f'Intervalo confiança original:
{knn_kfold.intervalo_confianca()}')

    data_scaled, _ = load_data('./data/questao_5_scaled.csv')
    X = np.array([tuple([float(x) for x in list(row.values())
[1:]]) for row in data_scaled])
    y = np.ravel([float(row['class']) for row in data_scaled])
    knn_kfold = KNeighborsKFoldClassifier(X, y)
    knn_kfold.classify()
    print(f'Intervalo confiança intervalo [0,1]:
{knn_kfold.intervalo_confianca()}')

    data_std, _ = load_data('./data/questao_5_std.csv')
    X = np.array([tuple([float(x) for x in list(row.values())
[1:]]) for row in data_std])
    y = np.ravel([float(row['class']) for row in data_std])
    knn_kfold = KNeighborsKFoldClassifier(X, y)
    knn_kfold.classify()
    print(f'Intervalo confiança padronização:
{knn_kfold.intervalo_confianca()}')


if __name__ == '__main__':
    main()
```