PUC-RIO

Trabalho Final de Data Mining

Problema de Classificação

Professora: Manoela Kohler

Antonio Carlos de Oliveira e Resende

Gabriel Alboretti de Souza

15/1/2019

Sumário

1) Problema

O problema consiste em utilizar 27 atributos que descrevem o estado de saúde de cavalos e conseguir descobrir se o cavalo vai sobreviver, morrer ou sofrer eutanásia. Esse problema de classificação é apresentado em 2 arquivos, 1 para o treinamento dos modelos criados e outro para o teste dos modelos.

2) Proposta

Para o problema, foi proposto utilizar a linguagem de programação python devido a sua facilidade de implementação e por ser amplamente utilizada na criação de modelos peditórios. A IDE escolhida foi *a Jupyter Notebook* devido a sua praticidade para criar e debugar o código.

2.1) Bibliotecas Utilizadas

A principal biblioteca utilizada para a solução do problema é a *pandas*, porém as demais bibliotecas utilizadas estão abaixo.

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn import tree
from sklearn.svm import SVC

import sklearn.metrics as metrics
import sklearn.preprocessing as preprocessing
import pandas as pd
import numpy as np

pd.options.display.max_rows = 999
```

2.2) Arquivos Utilizados

Os 2 arquivos utilizados (horse.csv e horseTest.csv) foram lidos diretamente via função *read_csv* da biblioteca *pandas*. Um é utilizado para a criação e treinamento dos modelos e o outro para o teste dos modelos.

```
#Load in the data with `read_csv()`
horsesDataSet = pd.read_csv('horse.csv', header=0, delimiter=',')
horsesDataSetTest = pd.read_csv("horseTest.csv", header=0, delimiter=',')

#description of dataSet
descriptionHorsesDataSet = horsesDataSet.describe(include='all')
descriptionHorsesDataSetTest = horsesDataSetTest.describe(include='all')
```

3) Análise exploratória

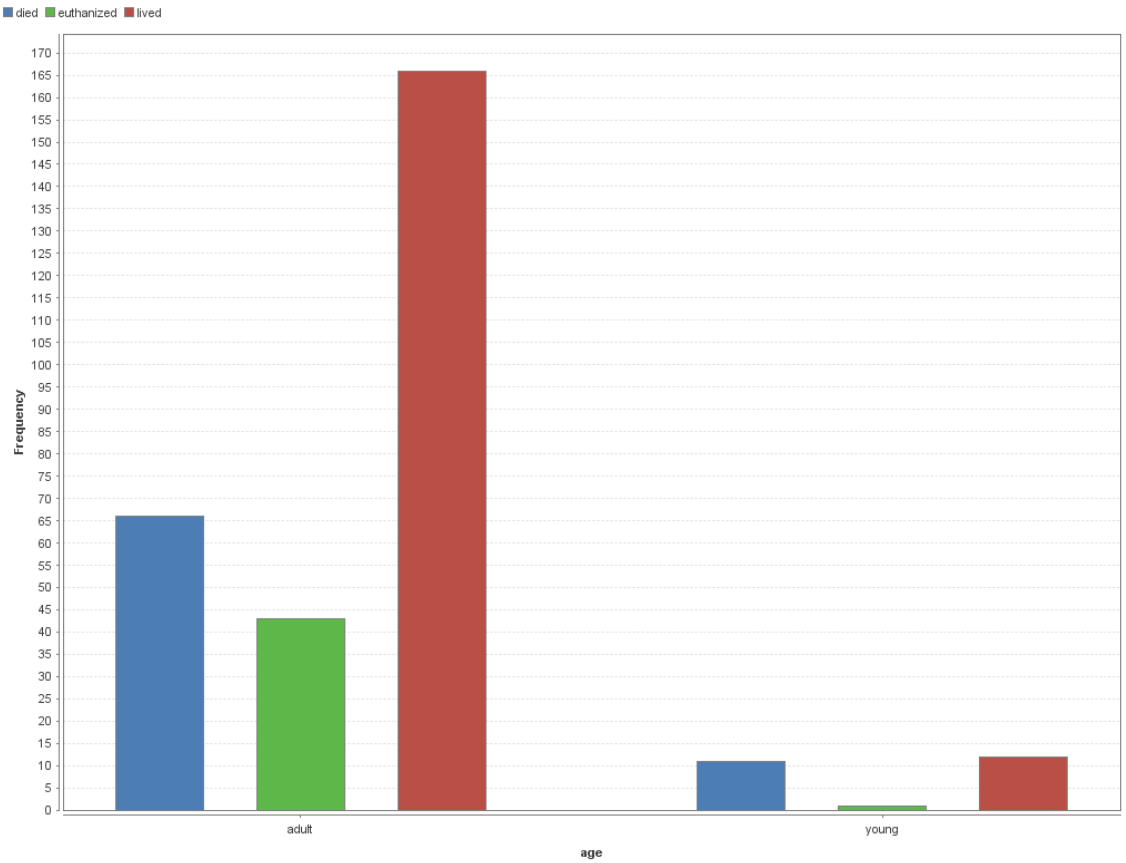
Nesta etapa temos como objetivo conhecer a base de dados para trabalhar melhor com o problema, extraindo o máximo de informações sobre o mesmo por meio de gráficos e observações.

Podemos notar que a base possui 27 atributos sem conta com o atributo alvo que queremos classificar. Foi verificado que os atributos *rectal_temp*, *respiratory_rate*, *nasogastric_tube*, *nesogastric_reflux*, *nesogastric_tube_ph*, *rectal_exam_feces*, *abdomen*, *packed_cell_volume*, *total_protein*, *abdomo_appearence*, *abdomo_protein* possuem seus maiores valores marcados como NA, o que indica que os mesmos possuem uma falta de informação grande. Na seção de pré-processamento explicamos melhor o critério utilizado para tratamento desses atributos.

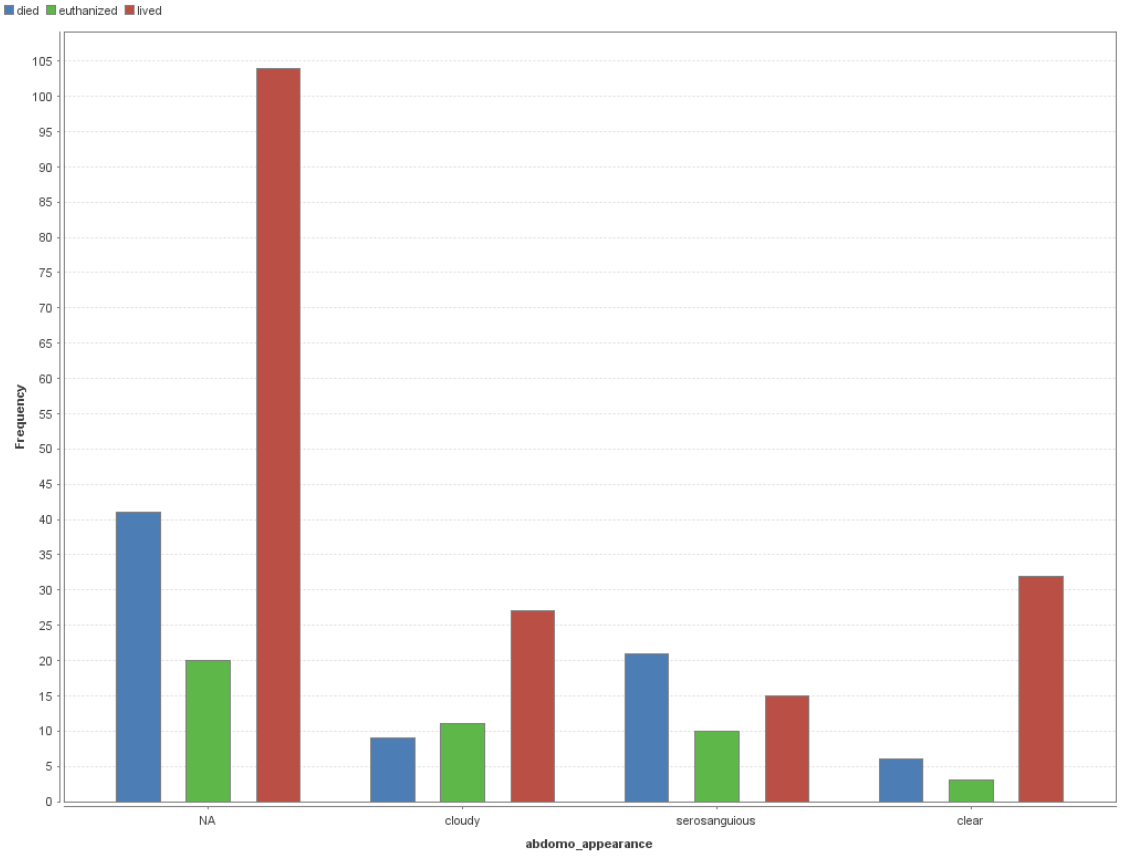
Na coluna de *target* (*outcome*), percebemos que para fins de predizer se um cavalo vive ou morre a classificação *euthanized* é irrelevante, no sentido em que no final um cavalo que sofreu eutanásia tem o mesmo fim que um cavalo dado como morto (*died*). Provavelmente podemos melhorar a classificação dada pelos algoritmos tornando o problema atual que seria de ordem 3 em um de classificação binária.

Possuímos também colunas que têm aspecto de id, ou seja, não devem ser muito úteis para a classificação já que não trazem informações relevantes ou que influenciam o resultado final do destino do animal.

A base é predominantemente composta por dados de cavalos adultos, e quando jovens dificilmente os cavalos sobrevivem.



No atributo *abdomo_appearence*, temos uma quantidade grande de informação faltante pois a maioria dos cavalos que morrem não possuem está informação, deixando um pouco forte a hipótese que tal atributo não deve impactar muito na nossa predição já que nos informa muito pouco.


died
euthanized
lived
Frequency
105
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
NA
cloudy
serosanguious
clear
abdomo_appearance

4) Pré processamento

Esta etapa do trabalho consiste em preparar os dados para que os modelos possam ser executados. A etapa pode ser dividida em 3 fases:

1) Analisando atributos desnecessários e/ou redundantes;
2) Tratamento de dados faltantes;
3) Transformação dos atributos para uma melhor análise dos modelos.

4.1) Atributos Desnecessários

Nessa fase foi avaliado a quantidade de dados faltantes para cada atributo, e foi decidido deletar e não utilizar atributos que possuem mais dados faltantes que um determinado *threshold* estipulado previamente. O *threshold* utilizado foi de 0.5, ou seja, atributos com mais de 50% de dados faltantes foram deletados da base de treino dos modelos.

```
# iterate through each attribute and define the percentage of missing values
# populate array with zeros with column dimensions of dataset
qtd_nan = [0 for x in range(horsesDataSet.shape[1])]

# populate array with zeros with column dimensions of dataset
qtd_total = [0 for x in range(horsesDataSet.shape[1])]

i = 0
while i < horsesDataSet.shape[1]:
    # get array of boolean describing each line as null or not for i attribute
    attributeLinesIsNA = pd.isna(horsesDataSet.iloc[:, i])

    # get current attribute label name
    currentAttributeLabel = list(horsesDataSet)[i]

    qtd_nan[i] = horsesDataSet.loc[attributeLinesIsNA, currentAttributeLabel].shape[0]
    qtd_total[i] = horsesDataSet.loc[:, currentAttributeLabel].shape[0]
    i = i+1
```

```
# dropping atributes
threshold = 0.5
PreProcessedHorseDataSet = horsesDataSet
PreProcessedHorseDataSetTest = horsesDataSetTest
i = 0
while i < horsesDataSet.shape[1]:
    if percentageArray[i] > threshold:
        # get current attribute label name
        currentAttributeLabel = list(horsesDataSet)[i]

        # drop attribute column if na values > threshold
        PreProcessedHorseDataSet =
PreProcessedHorseDataSet.drop(columns=currentAttributeLabel)

        #drop from test
        PreProcessedHorseDataSetTest =
PreProcessedHorseDataSetTest.drop(columns=currentAttributeLabel)

    i = i + 1
```

4.2) Tratamento de dados Faltantes

Com os atributos restantes, nessa fase foram feitas as inserções de dados para auxiliar os modelos em seus treinamentos. Para atributos numéricos, foi decidido

substituir os valores nulos pela média de todos os outros existentes. Para os atributos categóricos, foi decidido substituir os valores faltantes pelo valor mais presente dentre os não nulos. Lembrando que também foi necessário fazer o mesmo método na base de teste, porém os valores utilizados para preencher os campos nulos são provenientes da base de treino.

```
# fill remaining lines with mean values (only numerical)
PreProcessedHorseDataSet = PreProcessedHorseDataSet.fillna(horsesDataSet.mean())

# Show Statistics of DataSet
StatisticsPreProcessedHorseDataSet = PreProcessedHorseDataSet.describe(include='all')

# Altering Categorical missing values to Mode Value (value that appear the most often)
i = 0
while i < PreProcessedHorseDataSet.shape[1]:
    # return the most frequent value (first index because mode() returns a DataFrame)
    attributeMode = PreProcessedHorseDataSet.mode().iloc[0, i]
    currentAttributeLabel = list(PreProcessedHorseDataSet)[i]
    PreProcessedHorseDataSet[currentAttributeLabel] =
PreProcessedHorseDataSet[currentAttributeLabel].fillna(attributeMode)
    i = i+1

# Altering missing values [DATASET TEST]
#Saving values from train to insret in TEST with variable v
v = [0 for x in range(horsesDataSet.shape[1])]
i=0
while i < PreProcessedHorseDataSet.shape[1]:
    if PreProcessedHorseDataSet.dtypes[i] == 'O':
        v[i] = PreProcessedHorseDataSet.mode().iloc[0, i]
    else:
        v[i] = PreProcessedHorseDataSet.iloc[0, i].mean()

    currentAttributeLabel = list(PreProcessedHorseDataSetTest)[i]
    PreProcessedHorseDataSetTest[currentAttributeLabel] =
PreProcessedHorseDataSetTest[currentAttributeLabel].fillna(v[i])
    i = i+1
```

4.3) Transformação dos Atributos

Para finalizar, nesta última fase foram feitas três modificações nas bases de treino e de teste. A primeira modificação foi fazer com que o modelo tivesse que optar por escolher se o cavalo iria viver ou morrer, ou seja, a categoria que classificava que o cavalo seria submetido a eutanásia foi transformada na mesma categoria em que o cavalo morreria.

```
# Change values from euthanized to died
AttributesHorseDataSet = PreProcessedHorseDataSet.drop('outcome', axis=1)
TargetHorseDataSet = PreProcessedHorseDataSet.loc[:, 'outcome']

# mapping 'euthanized' values to 'died' to tune fitting
TargetHorseDataSet = TargetHorseDataSet.map(lambda x: 'died' if x == 'euthanized' else x)

PreProcessedHorseDataSet = pd.concat([AttributesHorseDataSet, TargetHorseDataSet], axis=1)

# Change values from euthanized to died [DATASET TEST]
AttributesHorseDataSetTest = PreProcessedHorseDataSetTest.drop('outcome', axis=1)
TargetHorseDataSetTest = PreProcessedHorseDataSetTest.loc[:, 'outcome']

# mapping 'euthanized' values to 'died' to tune fitting
TargetHorseDataSetTest = TargetHorseDataSetTest.map(lambda x: 'died' if x == 'euthanized' else x)
PreProcessedHorseDataSetTest = pd.concat([AttributesHorseDataSetTest, TargetHorseDataSetTest],
axis=1)
```

A Segunda modificação na base foi feita para de fato transformar os atributos categóricos. O método utilizado foi o de binarização, que consiste em criar uma coluna na base para cada valor que o atributo pode possuir preenchendo essas novas colunas com 0 ou 1, onde 1 informa que aquela coluna é o valor do atributo e 0 informa o contrário. Dessa forma, os dados podem passar por qualquer modelo com mais facilidade, uma vez que cada coluna nova só poderá assumir 2 valores.

```
# categorical attribute binarization
categoricalHorseDataSet = PreProcessedHorseDataSet.select_dtypes(include='object')
categoricalHorseDataSet = categoricalHorseDataSet.drop('outcome', axis=1)
categoricalHorseDataSetDummy = pd.get_dummies(categoricalHorseDataSet)
PreProcessedHorseDataSet = pd.concat([categoricalHorseDataSetDummy,
PreProcessedHorseDataSet.loc[:, 'outcome']], axis=1)

# categorical attribute binarization [DATASET TEST]
categoricalHorseDataSetTest = PreProcessedHorseDataSetTest.select_dtypes(include='object')
categoricalHorseDataSetTest = categoricalHorseDataSetTest.drop('outcome', axis=1)
categoricalHorseDataSetDummy = pd.get_dummies(categoricalHorseDataSetTest)
PreProcessedHorseDataSetTest = pd.concat([categoricalHorseDataSetDummy,
PreProcessedHorseDataSetTest.loc[:, 'outcome']], axis=1)
```

Quanto a terceira transformação, ela foi realizada para trocar o tipo da variável de saída de *object* para *category*, sabendo que alguns modelos só aceitam esse tipo de variável.

```
# Convertendo objetos para categóricos
i= 0
while i < PreProcessedHorseDataSet.shape[1]:
    if PreProcessedHorseDataSet[list(PreProcessedHorseDataSet)[i]].dtypes == 'O':
        PreProcessedHorseDataSet[list(PreProcessedHorseDataSet)[i]] =
PreProcessedHorseDataSet[list(PreProcessedHorseDataSet)[i]].astype('category')
    i = i+1

i= 0
while i < PreProcessedHorseDataSetTest.shape[1]:
    if PreProcessedHorseDataSetTest[list(PreProcessedHorseDataSetTest)[i]].dtypes == 'O':
        PreProcessedHorseDataSetTest[list(PreProcessedHorseDataSetTest)[i]] =
PreProcessedHorseDataSetTest[list(PreProcessedHorseDataSetTest)[i]].astype('category')
    i = i+1
```

5) Aplicação dos modelos

Esta etapa do trabalho consiste na criação e aplicação dos modelos de *Machine Learning*. Foram utilizados basicamente 2 modelos, KNN (*k-nearest neighbors*), Árvore de Decisão e SVM, e para cada um deles foi utilizado o método de Regressão Logística (LR) para potencializá-los.

5.1) KNN

Esse modelo consiste em associar a saída do objeto testado ao seu vizinho lógico mais próximo, ou seja, procura na base de treino o objeto mais parecido com o testado e desse modo o associa a ele. Como na biblioteca utilizada era obrigatório informar o número vizinhos, foi decidido rodar o mesmo algoritmo variando o número de vizinhos de 0 a 100 e escolhendo o melhor número de vizinhos ótimos àquele que possuísse o melhor resultado.

```
result_lr = 0
melhor_qty = 0
resultados = []

for j in range(1,len(PreProcessedHorseDataSet)):
    model = LogisticRegression()
    feature_qty = j
    dataset = PreProcessedHorseDataSet
    rfe = RFE(model, feature_qty)
    rfe = rfe.fit(dataset.drop('outcome', axis = 1), dataset.outcome.values)

    col = []
    for i in range(0,rfe.support_.size):
        if rfe.support_[i] == True:
            col.append(i)

    dataset_after_rfe = dataset[dataset.columns[col]]
    x_train, x_test, y_train, y_test = train_test_split(dataset_after_rfe, dataset['outcome'], random_state = 0)
    tamanho = 100
    maior_knn = 0
    pos = 0
    vet =[]
    for i in range(1,tamanho):
        knn = KNeighborsClassifier(n_neighbors = i)
        knn.fit(x_train, y_train)
        result = knn.score(x_test, y_test)
        vet.append(result)
        if result > maior_knn:
            maior_knn = result
            pos = i

    knn = KNeighborsClassifier(n_neighbors = pos)
    knn.fit(x_train, y_train)
    TESTE = PreProcessedHorseDataSetTest[list(dataset[dataset.columns[col]].columns)]

    h = knn.score(TESTE, PreProcessedHorseDataSetTest.outcome)
    resultados.append(h)
    if h > result_lr:
        result_lr = h
        melhor_qty = j
        TargetHorseDataSet_prediction = knn.predict(x_test)
        TargetHorseDataSet_test = y_test
```

5.2) KNN + LR

A única diferença deste modelo para o anterior é o acréscimo do LR. Tal qual foi utilizado para achar a quantidade ótima de atributos para que o modelo funcione da melhor forma. Uma vez que o LR realiza uma classificação dos atributos avaliando quais são os mais relevantes para a solução do problema. Como na biblioteca utilizada era obrigatório informar o número de atributos desejados, foi decidido rodar o mesmo algoritmo variando o número de atributos de 1 até o número total de atributos.

Dessa forma, foi aplicado a variação do KNN de 0 a 100 para cada quantidade de atributos e a quantidade de atributos ótima foi escolhida levando em consideração o melhor score final.

5.3) Árvore de Decisão

Esse modelo consiste em classificar os dados criando diversas ramificações baseadas em comparações com algum valor até que ache a “folha” que melhor represente aquele dado.

5.4) Árvore de Decisão + LR

Como foi feito com o KNN, foi utilizado o LR para potencializar o valor do modelo anterior. E o score final é referente a quantidade de atributos ótima.

```
for j in range(1,len(PreProcessedHorseDataSet)):
    model = LogisticRegression()
    feature_qty = j
    dataset = PreProcessedHorseDataSet
    rfe = RFE(model, feature_qty)
    rfe = rfe.fit(dataset.drop('outcome', axis = 1), dataset.outcome.values)

    col = []
    for i in range(0,rfe.support_.size):
        if rfe.support_[i] == True:
            col.append(i)

    dataset_after_rfe = dataset[dataset.columns[col]]

    # label encoder
    labelEncoder = preprocessing.LabelEncoder()
    labelEncoder.fit(TargetHorseDataSet.values)
    TargetHorseEncodedArray = labelEncoder.transform(TargetHorseDataSet.values)
    TargetHorseEncodedDataSet = pd.DataFrame(TargetHorseEncodedArray, columns=['outcome'])

    # split train and test data and target
    AttributesHorseDataSet_train, AttributesHorseDataSet_test, TargetHorseDataSet_train,
TargetHorseDataSet_test = train_test_split(dataset_after_rfe, TargetHorseEncodedDataSet,
random_state=1)

    # initialize model parameters
    decisionTreeModel = tree.DecisionTreeClassifier()

    # fit model using training data
    decisionTreeModel.fit(AttributesHorseDataSet_train, TargetHorseDataSet_train)

    # predict our test data using fitted model
    TargetHorseDataSet_prediction = decisionTreeModel.predict(AttributesHorseDataSet_test)
    accuracyScore = metrics.accuracy_score(TargetHorseDataSet_test, TargetHorseDataSet_prediction)

    if accuracyScore > result_lr:
        result_lr = accuracyScore
        melhor_qty = j
        TargetHorseDataSet_prediction_Store = decisionTreeModel.predict(AttributesHorseDataSet_test)
        TargetHorseDataSet_test_Store = TargetHorseDataSet_test
```

5.5) SVM e SVM + LR

O SVM tenta traçar um hiperplano entre a base de dados, no caso clássico dividindo a base de dados entre dois tipos. Como de antemão binarizamos o problema, o SVM foi uma escolha natural para classificarmos a base.

```
for j in range(1,PreProcessedHorseDataSet.shape[1]):
    model = LogisticRegression()
    feature_qty = j
    dataset = PreProcessedHorseDataSet
    rfe = RFE(model, feature_qty)
    rfe = rfe.fit(dataset.drop('outcome', axis = 1), dataset.outcome.values)

    col = []
    for i in range(0,rfe.support_.size):
        if rfe.support_[i] == True:
            col.append(i)

    dataset_after_rfe = dataset[dataset.columns[col]]

    # split train and test data and target
    AttributesHorseDataSet_train, AttributesHorseDataSet_test, TargetHorseDataSet_train,
TargetHorseDataSet_test = train_test_split(AttributesHorseDataSet, TargetHorseEncodedDataSet,
random_state=1)

    svmClassifier = SVC(kernel='linear', C = 1.0)
    svmClassifier.fit(AttributesHorseDataSet_train, TargetHorseDataSet_train)

    TargetHorseDataSet_prediction = svmClassifier.predict(AttributesHorseDataSet_test)

    accuracyScore = metrics.accuracy_score(TargetHorseDataSet_test, TargetHorseDataSet_prediction)

    if accuracyScore > result_lr:
        result_lr = accuracyScore
        melhor_qty = j
        TargetHorseDataSet_prediction_Store = svmClassifier.predict(AttributesHorseDataSet_test)
        TargetHorseDataSet_test_Store = TargetHorseDataSet_test
```

6) Resultado dos modelos

Para a avaliação dos modelos foram utilizadas quatro métricas:

1) Acurácia;
2) Recall;
3) Cohen's Kappa;
4) Matriz de Confusão.

```
accuracyScore = metrics.accuracy_score(TargetHorseDataSet_test_Store,
TargetHorseDataSet_prediction_Store)
print(accuracyScore)
recallScore = metrics.recall_score(TargetHorseDataSet_test_Store,
TargetHorseDataSet_prediction_Store, average=None)
print(recallScore)
kappaScore = metrics.cohen_kappa_score(TargetHorseDataSet_test_Store,
TargetHorseDataSet_prediction_Store)
print(kappaScore)

#TargetHorseDataSet_test = labelEncoder.inverse_transform(TargetHorseDataSet_test)
#TargetHorseDataSet_prediction = labelEncoder.inverse_transform(TargetHorseDataSet_prediction)

confusionMatrix = pd.DataFrame(
    metrics.confusion_matrix(TargetHorseDataSet_test_Store, TargetHorseDataSet_prediction_Store),
    columns=['Predicted Died', 'Predicted Lived'],
    index=['True Died', 'True Lived']
)
```

6.1) KNN

```
Accuracy Score:
0.7466666666666667
Recall Score:
[0.70588235 0.7804878 ]
Cohens Kappa Score:
0.4875943905070118
            Predicted Died  Predicted Lived
True Died               24               10
True Lived               9               32
```

6.2) KNN + LR

```
Accuracy Score:
0.7733333333333333
Recall Score:
[0.64705882 0.87804878]
Cohens Kappa Score:
0.5345016429353779
            Predicted Died  Predicted Lived
True Died               22               12
True Lived               5               36
```

6.3) Árvore de Decisão

```
Accuracy Score:
0.7333333333333333
Recall Score:
[0.78125    0.69767442]
Cohens Kappa Score:
0.46770759403832507
            Predicted Lived  Predicted Died
True Lived               25               7
True Died                13              30
```

6.4) Árvore de Decisão + LR

```
Accuracy Score:
0.8266666666666667
Recall Score:
[0.65625    0.95348837]
Cohens Kappa Score:
0.6324915190350546
            Predicted Died  Predicted Lived
True Died               21               11
True Lived               2               41
```

6.5) SVM e SVM + LR

```
Accuracy Score:
0.76
Recall Score:
[0.78125    0.74418605]
Cohens Kappa Score:
0.5171673819742489
            Predicted Died  Predicted Lived
True Died               25               7
True Lived              11              32
```

7) Conclusão

Após a avaliação dos modelos apresentados, o melhor resultado obtido foi o que aplicamos uma Árvore de decisão seguido da regressão logística. Foi percebido que o modelo possui uma taxa de acerto maior para os casos em que o cavalo morre.

Foram obtidos melhores resultados após a escolha de binarizar os atributos e transformar os casos de eutanásia em morte.

Reduzir o número de atributos por meio de regressão logística melhorou a acurácia e o coeficiente kappa em todos os classificadores, salvo o SVM que manteve seu resultado igual.