Utilização do Modelo de Classificação SVM (Support Vector Machines)

Lucian Sturião, Aluno de Mestrado, PESC/COPPE - UFRJ
Carlos Eduardo Pedreira, Professor Associado - Pesquisador CNPq 1B / Cientista do Nosso Estado, PESC/COPPE - UFRJ

Abstract—A título de estudo foi usando o dataset ”banana” [2] e foi feita a comparação de sua classificação utilizando alguns conhecidos algoritmos: *Support Vector Machines* e *K-Nearest Neighbors*. Foram feitos os gráficos para visualização dos principais casos estudados com cada um dos algoritmos e também foi feita a comparação entre estes em questão de acurácia e desempenho.

Index Terms—Support Vector Machine, Classifier, Machine Learning, Artificial Intelligence.

1 Introdução

Este artigo tem como intenção estudar algumas diferentes técnicas de classificação utilizando como base o dataset ”banana” [2]. Serão utilizadas dois classificadores muito conhecidos: *Support Vector Machines* e *K-Nearest Neighbors*.

1.1 Support Vector Machines

Support Vector Machines é hoje em dia muito utilizado em diversas aplicações com resultados satisfatórios, e por alguns é considerado ”o melhor algoritmo de aprendizado de máquina” [11]. Entretando, esta é uma afirmação muito forte.

A ideia por trás no SVM é simples. Dado que se recorda como funcionam os *Perceptrons* [12], pode-se dizer que o SVM é uma extensão do *PLA*. O objetivo é conseguir separar datasets que não são linearmente separáveis utilizando hiperplanos e maximizar a margem das ”linhas” de separação entre os dados.

Os pontos que definem estas margens e o prórpio hiperplano são chamados de *Support Vectors*.

1.1.1 Kernels

Os Kernels são a base para o aprendizado do hiperplano. São eles que definem o cálculo utilizado na predição de uma entrada por meio dos vetores de suporte. Alguns Kernels utilizados foram: Linear, Plinomial, Sigmóide, Radial.

1.1.2 Parâmetro Gama

Este parâmetro é responsável pelo fator de consideração de proximidade dos pontos vizinhos na classificação de uma entrada. Foi criado também o gráfico com o Kernel Sigmóide e Gama 0.05 para ser observada mais claramente a diferença entre os valores de Gama. Quanto maior o valor de Gama, menos em consideração são levados pontos mais distantes. Quando menor o valor, mais em consideração são levados pontos distantes [13].

Pode se observar esta mudança nos gráficos de sigmoide de 1, 0.5, 0.05 e 0.01 gama. Quanto maior o gama, mais ”maleável” é a linha de classificação e quanto menor, mais ”rígida”. No gráfico de gama 0.01, por exemplo, a faixa de classificação nem se encontra mais no campo visível. Visto que existem mais pontos azuis que vermelhos, todos acabam sendo classificados como azuis.

1.2 K-Nearest Neighbors

O *KNN* é um dos algoritmos de classificação baseados em posicionamento espacial de seus registros. Este é altamente simples de ser implementado, e não pode se considerar que haja um ”treinamento” de fato visto que seu mecanismo funciona de forma que o resultado da classificação é baseado nas k instâncias mais próximas do registro ao qual se deseja predizer a classificação [9].

1.2.1 Escolha do algoritmo

Como constatado em 1.2, o algoritmo *KNN* é uma boa escolha para o dataset estudado. Observa-se em 3.1 que a classificação dos registros deste é relativamente baseada no posicionamento espacial dos mesmos, e por ser o *KNN* um algoritmo de fácil implementação (apesar de não ter sido implementado de fato no trabalho) e de baixo requisito de poder de processamento, foi escolhido.

1.3 Validação cruzada

A validação cruzada se faz necessária visto que no ”mundo real” não se tem a disposição todos dados nos quais o algoritmo será utilizado. O que acontece é que é feito o treinamento dos algoritmos com um conjunto de dados e é feito o teste com um conjunto diferente, que não utilizado no treinamento, para que não haja ”contaminação” do aprendizado. Esta técnica é utilizada para se ”medir a performance preditiva de um algoritmo” [8].

1.3.1 K-Fold

A técnica *K-Fold* é um procedimento simples, porém eficaz. Deve-se dividir o dataset original em K partes, inicialmente. Após esta divisão inicial, itera-se sobre as K partes e faz-se cada uma das partes por vez ser utilizada como teste e as partes restantes são utilizadas no treinamento.

Existe também variantes onde se faz validação cruzada interna e externa, para se otimizar parâmetros de um determinado algoritmo ou técnica utilizada. O que foi utilizado neste trabalho foi o procedimento simples de *K-Fold* explicado sucintamente acima.

2 METODOLOGIA

2.1 Utilização do Dataset

Vale mencionar que o dataset previamente a cada treino e teste foi embaralhado. Isto foi feito para que a ordem em que o dataset se apresentava inicialmente não influenciasse diretamente nos resultados obtidos. A ordem entre os atributos e classificação foi observada e não foi modificada, obviamente.

2.2 Método de visualização

Os gráficos foram acrescentados como forma de exemplificar visualmente o que cada um dos tipos de algoritmo e difentes parâmetros podem fazer na classificação do dataset escolhido. A visualização em si não representa a classificação obtida nas tabelas 2 e 4 pois em ambos os casos foi utilizado a técnica de validação cruzada *k-fold* como explicado na seção 1.3.1. Na visualização dos dados, como o objetivo não era de validar o algoritmo, foi utilizado sempre o dataset em sua completude. Sendo assim, todos registros originais se encontram em todos gráficos.

3 RESULTADOS

3.1 Visualização da classificação Original

Aqui pode ser observada o data como vem originalmente. Este foi o dataset utilizado como base para todo o trabalho.

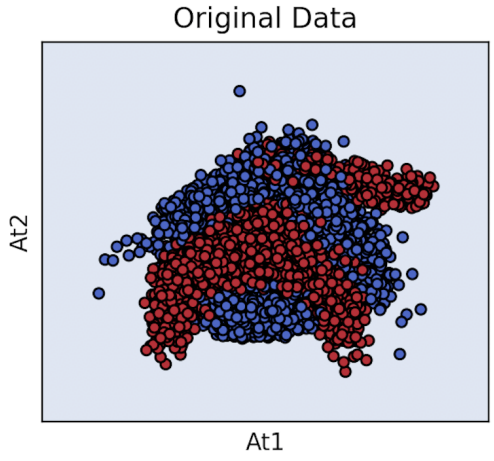

Fig. 1: Dataset Original

3.2 Utilização de SVMs para Classificação

3.2.1 Visualização dos Kernels do SVM

Cada um dos tipos de kernels e parâmetros utilizados podem ser observados nesta seção.

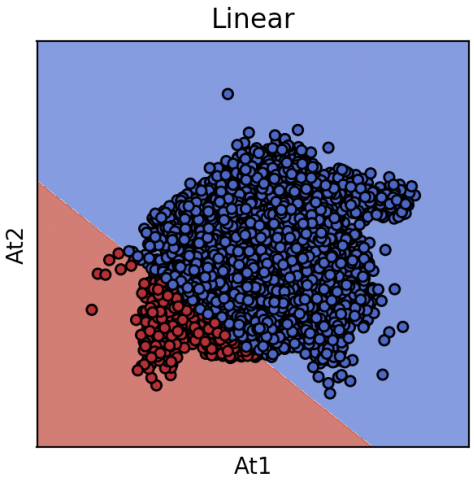

Fig. 2: Classificação do Kernel Linear

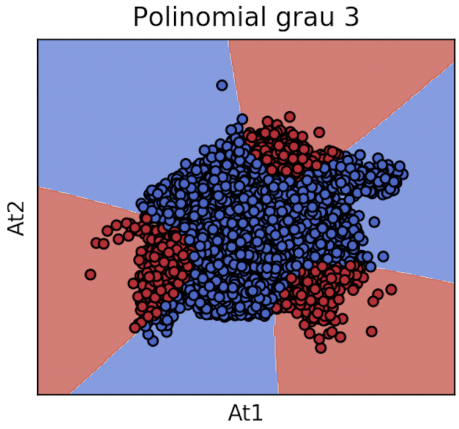

Fig. 3: Classificação do Kernel Polinomial (grau 3)

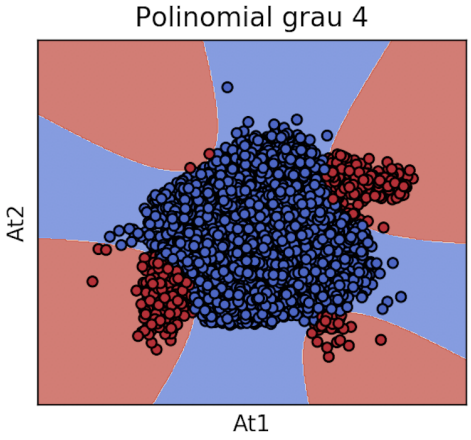

Fig. 4: Classificação do Kernel Polinomial (grau 4)

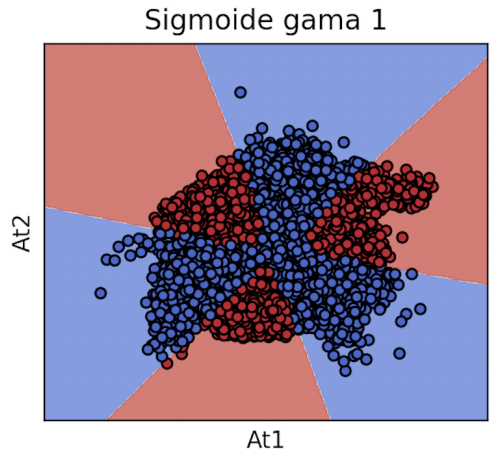

Fig. 5: Classificação do Kernel Sigmóide (gama 1)

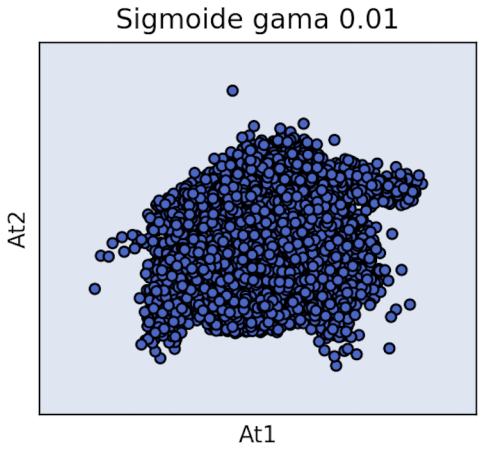

Fig. 8: Classificação do Kernel Sigmóide (gama 0.01)

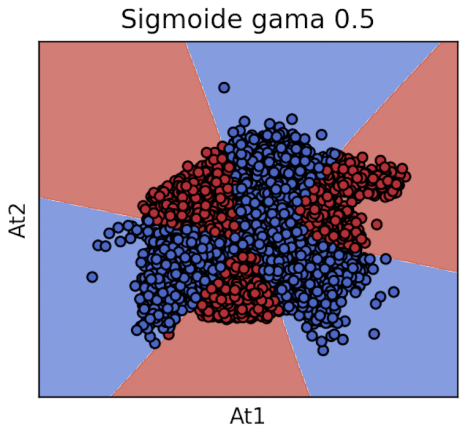

Fig. 6: Classificação do Kernel Sigmóide (gama 0.5)

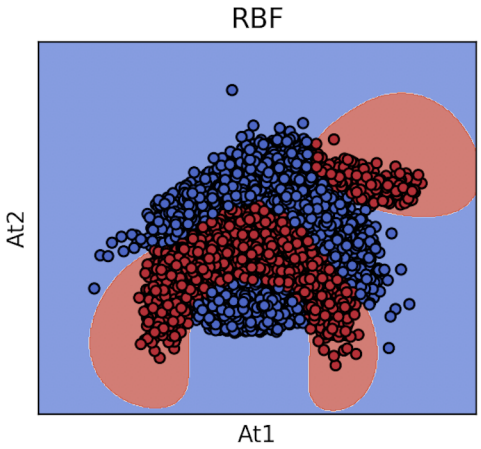

Fig. 9: Classificação do Kernel RBF

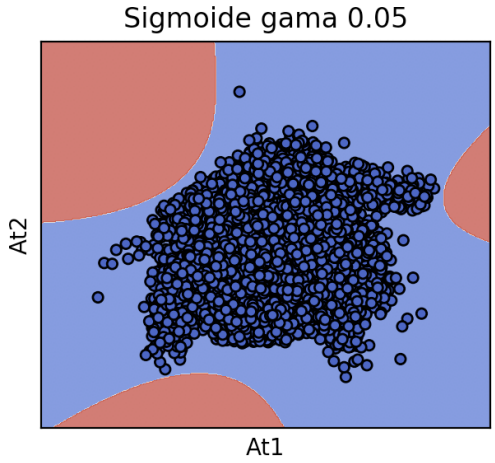

Fig. 7: Classificação do Kernel Sigmóide (gama 0.05)

3.2.2 Kernels e respectivos E_{out}

Para cada um dos Kernels e parâmetros foi anotado qual erro no dataset de treino (E_{out}) foi observado para cada um tipo de *k-fold* testado na tabela 1. Foram utilizado *2-fold*, *5-fold*, e *10-fold* como explicado em 1.3.1.

3.2.3 Melhores resultados e Respectidos K-Fold

Após a comparação entre os métodos foi anotado para qual dos *K-fold* se obteve melhor resultado e qual este resultado na tabela 2.

3.3 Utilização de KNN para Classificação

3.3.1 Visualização da Classificação usando KNN

Similarmente a 3.2.1, alguns gráficos de diferentes números de vizinhos foram gerados para se observar visualmente qual resultado gerado na classificação usando *KNN*.

K-Fold	SVM Kernel	E_{out}
2	Linear	0.4177
	Polinomial (grau 3)	0.3558
	Polinomial (grau 4)	0.3630
	Sigmoide (gama 1)	0.7135
	Sigmoide (gama 0.5)	0.7045
	Sigmoide (gama 0.01)	0.4452
	RBF	0.0947
5	Linear	0.4245
	Polinomial (grau 3)	0.3547
	Polinomial (grau 4)	0.3481
	Sigmoide (gama 1)	0.6971
	Sigmoide (gama 0.5)	0.6839
	Sigmoide (gama 0.01)	0.4330
	RBF	0.0933
10	Linear	0.4207
	Polinomial (grau 3)	0.3226
	Polinomial (grau 4)	0.3339
	Sigmoide (gama 1)	0.6924
	Sigmoide (gama 0.5)	0.6735
	Sigmoide (gama 0.01)	0.4150
	RBF	0.0679

TABLE 1: Tabela de resultados por K-Fold

SVM Kernel	K-Fold	E_{out}
Linear	2	0.4177
Polinomial (grau 3)	10	0.3226
Polinomial (grau 4)	10	0.3339
Sigmoide (gama 1)	10	0.6924
Sigmoide (gama 0.5)	10	0.6735
Sigmoide (gama 0.01)	10	0.4150
RBF	10	0.0679

TABLE 2: Melhores resultados por Kernel

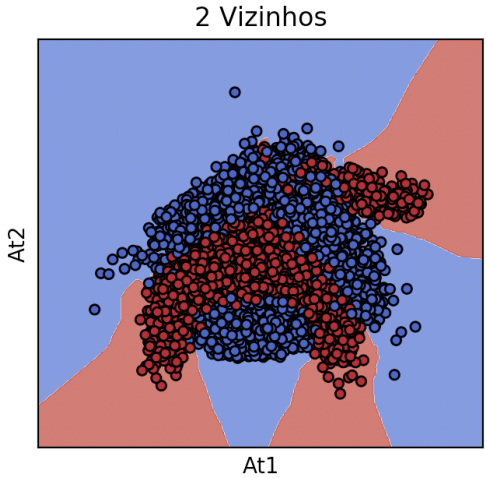

Fig. 10: Classificação com 2 Vizinhos

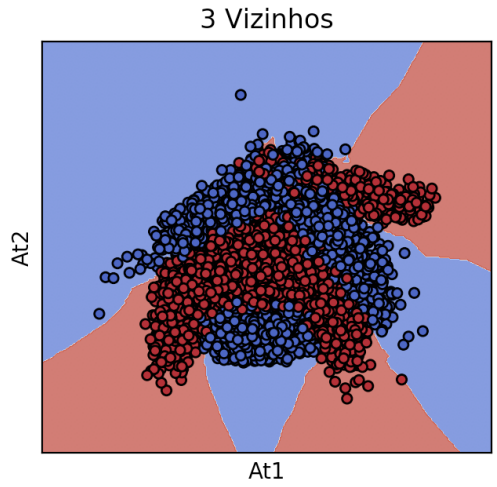

Fig. 11: Classificação com 3 Vizinhos

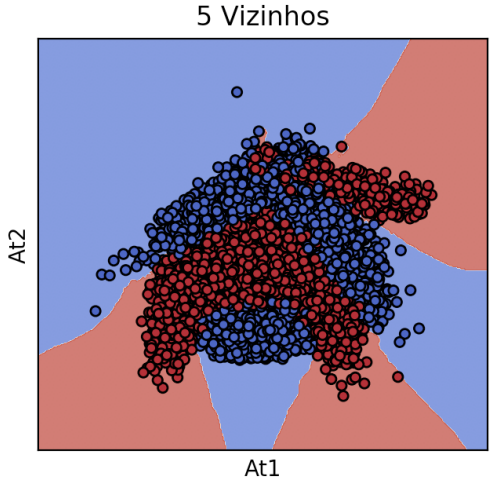

Fig. 12: Classificação com 5 Vizinhos

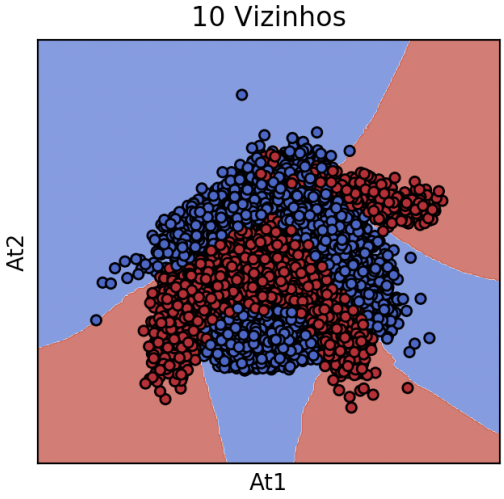

Fig. 13: Classificação com 10 Vizinhos

3.3.2 Kernels e respectivos E_{out}

O mesmo que 3.2.2 foi feito para o algoritmo KNN na tabela 3.

K-Fold	Número de Vizinhos	E_{out}
2	2 Vizinhos	0.1215
	3 Vizinhos	0.1113
	5 Vizinhos	0.1037
	10 Vizinhos	0.0947
	20 Vizinhos	0.0939
	50 Vizinhos	0.0932
	100 Vizinhos	0.0981
5	2 Vizinhos	0.1103
	3 Vizinhos	0.1047
	5 Vizinhos	0.0924
	10 Vizinhos	0.0924
	20 Vizinhos	0.0801
	50 Vizinhos	0.0896
	100 Vizinhos	0.0858
10	2 Vizinhos	0.0905
	3 Vizinhos	0.0849
	5 Vizinhos	0.0811
	10 Vizinhos	0.0641
	20 Vizinhos	0.0698
	50 Vizinhos	0.0754
	100 Vizinhos	0.0773

TABLE 3: Tabela de resultados por K-Fold

Número de Vizinhos	K-Fold	E_{out}
2 Vizinhos	10	0.0905
3 Vizinhos	10	0.0849
5 Vizinhos	10	0.0811
10 Vizinhos	10	0.0641
20 Vizinhos	10	0.0698
50 Vizinhos	10	0.0754
100 Vizinhos	10	0.0773

TABLE 4: Melhores resultados por Vizinhos

3.3.3 Melhores resultados e Respectidos K-Fold

O mesmo que 3.2.3 foi feito para o algoritmo KNN na tabela 4.

4 CONCLUSÃO

Observou-se que com ambos algoritmos utilizados obteve-se uma boa acurácia de por volta de 93% nos melhores casos. E apesar do *SVM* ser citado em alguns momentos como "melhor algoritmo de aprendizado de máquina" no caso observado aqui foi ultrapassado por um simples algoritmo de comparação a vizinhos: o *KNN*.

O melhor caso do *SVM* teve E_{out} de 0.0679 contra 0.0641 do *KNN*. É pouca diferença, mas, considerando a diferente complexidade entre os algoritmos, significante.

É fato que independente do algoritmo de aprendizado de máquina escolhido, se bem utilizado, e dependendo do problema, pode-se obter ótimos resultados. O caso visto neste trabalho foi um ótimo exemplo.

No *SVM* o melhor kernel observado foi o *RBF*, enquanto os outros kernels obtiveram baixíssima acurácia, com o segundo melhor atingindo por volta de 68% de acerto. Já em contrapartida, o *KNN* obteve bons resultados independente do número de vizinhos utilizados, sendo 10 vizinhos o melhor caso, com o segundo melhor não muito atrás (também por volta de 93% de acerto).

Todos códigos podem ser encontrados em [1] ou uma parte nos apêndices A e B.

REFERENCES

[1] Github: luciansr/svm-testing. Acesso em 7/09/2018.
[2] KEEL - Banana data set. Acesso em 4/09/2018.
[3] Plot different SVM classifiers in the iris dataset. Acesso em 5/09/2018.
[4] 1.4. Support Vector Machines. Acesso em 5/09/2018.
[5] An introduction to machine learning with scikit-learn. Acesso em 6/09/2018.
[6] sklearn.neighbors.KNeighborsClassifier. Acesso em 6/09/2018.
[7] Nearest Neighbors Classification. Acesso em 6/09/2018.
[8] Why every statistician should know about cross-validation. Acesso em 7/09/2018.
[9] A Complete Guide to K-Nearest-Neighbors with Applications in Python and R. Acesso em 7/09/2018.
[10] Support Vector Machines for Machine Learning. Acesso em 7/09/2018.
[11] Why Support Vector Machine(SVM) - Best Classifier?. Acesso em 7/09/2018.
[12] What the Hell is Perceptron?. Acesso em 7/09/2018.
[13] SVM Gamma Parameter. Acesso em 7/09/2018.

APPENDIX A
CÓDIGO DE VISUALIZAÇÃO

```
from sklearn import svm, datasets
import pandas
import numpy as np
from sklearn.utils import shuffle
import matplotlib.pyplot as plt

## parameters
# datasetName = filename
def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min,
        x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy


def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a
        classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to
        contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(),
        yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out

def plot(clf, title, ax, xx, yy, X0, X1, y,
    plt):
    plot_contours(ax, clf, xx, yy,
                  cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y,
        cmap=plt.cm.coolwarm, s=20,
        edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

    return

def test_svm(datasetName):


    ## k folds
    k_folds = [2,5,10]

    ## setup all svms
    C = 1.0 # SVM regularization parameter
    typesOfSVMs = [
        {
            'name': 'Linear',
            'learner': svm.LinearSVC()
        },
        {
            'name': 'Polinomial grau 3',
            'learner': svm.SVC(kernel='poly',
                degree=3, C=C)
        },
        {
            'name': 'Polinomial grau 4',
            'learner': svm.SVC(kernel='poly',
                degree=4, C=C)
        },
        {
            'name': 'Sigmoide gama 1',
            'learner': svm.SVC(kernel='sigmoid',
                gamma=1, C=C)
        },
        {
            'name': 'Sigmoide gama 0.5',
            'learner': svm.SVC(kernel='sigmoid',
                gamma=0.5, C=C)
        },
        {
            'name': 'Sigmoide gama 0.05',
            'learner': svm.SVC(kernel='sigmoid',
                gamma=0.05, C=C)
        },
        {
            'name': 'RBF',
            'learner': svm.SVC(kernel='rbf', C=C)
        },
        {
            'name': 'Original Data',
            'learner': svm.SVC(kernel='linear',
                C=C)
        },
    ]

    ## get data
    data = pandas.read_csv('./dataset/'+
        datasetName)

    X = data.values[:, :2]
    Y = data.values[:,2]

    svm_learner = svm.SVC(kernel='poly',
        degree=3)
    svm_learner.fit(X, Y)

    typesOfSVMs = ({
        'name': item['name'],
        'learner': item['learner'].fit(X, Y)
    } for item in typesOfSVMs)

    typesOfSVMs = ({
        'name': item['name'],
        'learner': item['learner'],
        'resultY': [1.0 if x > 0.0 else -1.0
            for x in
            item['learner'].predict(X)] if
```

```
            item['name'] != 'Original Data'
            else Y
    } for item in typesOfSVMs)

    # Set-up 2x2 grid for plotting.
    fig, sub = plt.subplots(2, 4)
    plt.subplots_adjust(wspace=0.4, hspace=0.4)

    X0, X1 = X[:, 0], X[:, 1]
    xx, yy = make_meshgrid(X0, X1)

    for SVMitem, ax in zip(typesOfSVMs,
        sub.flatten()):
        clf = SVMitem['learner']
        title = SVMitem['name']
        resultY = SVMitem['resultY']
        plot_contours(ax, clf, xx, yy,
                cmap=plt.cm.coolwarm, alpha=0.8)
        ax.scatter(X0, X1, c=resultY,
            cmap=plt.cm.coolwarm, s=20,
            edgecolors='k')
        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xlabel('At1')
        ax.set_ylabel('At2')
        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_title(title)


    plt.show()

    return None

test_svm('banana-data.dat')
```

APPENDIX B
CÓDIGO DE COMPARAÇÃO DE ACURÁCIA

```
from sklearn import svm, datasets
import pandas
import numpy as np
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
import sys


def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min,
        x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy


def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a
        classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to
        contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(),
        yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out

def plot(clf, title, ax, xx, yy, X0, X1, y,
    plt):
    plot_contours(ax, clf, xx, yy,
                cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y,
        cmap=plt.cm.coolwarm, s=20,
        edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

    return

def test_svm(datasetName):


    ## k folds
    k_folds = [2,5,10]

    ## setup all svms
    C = 1.0 # SVM regularization parameter
    typesOfSVMs = [
        {
            'name': 'Linear',
            'learner': svm.LinearSVC(),
            'E_out': None,
            'E_out_description': None
        },
        {
            'name': 'Polinomial grau 3',
            'learner': svm.SVC(kernel='poly',
                degree=3, C=C),
            'E_out': None,
            'E_out_description': None
        },
        {
            'name': 'Polinomial grau 4',
            'learner': svm.SVC(kernel='poly',
                degree=4, C=C),
            'E_out': None,
            'E_out_description': None
        },
        {
            'name': 'Sigmoide gama 1',
            'learner': svm.SVC(kernel='sigmoid',
                gamma=1, C=C),
```

```
            'E_out': None,
            'E_out_description': None
        },
        {
            'name': 'Sigmoide gama 0.5',
            'learner': svm.SVC(kernel='sigmoid',
                gamma=0.5, C=C),
            'E_out': None,
            'E_out_description': None
        },
        {
            'name': 'Sigmoide gama 0.01',
            'learner': svm.SVC(kernel='sigmoid',
                gamma=0.01, C=C),
            'E_out': None,
            'E_out_description': None
        },
        {
            'name': 'RBF',
            'learner': svm.SVC(kernel='rbf',
                C=C),
            'E_out': None,
            'E_out_description': None
        }
    ]

    ## get data
    data = pandas.read_csv('./dataset/'+
        datasetName)

    #print(svm_learner.predict([X[0, :]]))
    #print(X)
    #print(Y)

    row_number = data.values.shape[0]

    results_dic = dict()

    for k in k_folds:
        shuffledData = shuffle(data)
        X = shuffledData.values[:, :2]
        Y = shuffledData.values[:,2]

        for i in range(k):
            k_range = (row_number+1)/k
            fold_range = [int(k_range * i),
                int(k_range*(i+1))]

            fold_test_data_X = X[fold_range[0]:
                fold_range[1], :]
            fold_test_data_Y = Y[fold_range[0]:
                fold_range[1]]

            X_trainning_part1 = X[0:
                fold_range[0], :]
            X_trainning_part2 = X[fold_range[1]:
                row_number, :]

            Y_trainning_part1 = Y[0:
                fold_range[0]]
            Y_trainning_part2 = Y[fold_range[1]:
                row_number]

            fold_trainning_data_X =
                np.concatenate((X_trainning_part1,
                X_trainning_part2))
            fold_trainning_data_Y =
                np.append(Y_trainning_part1,
                Y_trainning_part2)

            #print(fold_test_data_X.shape[0] +
                fold_trainning_data_X.shape[0])
            for item in typesOfSVMs:
                clf_name = item['name']
                clf = item['learner']

                result_key_name = str(k) + ' -
                    k_fold - ' + clf_name

                if not(result_key_name in
                    results_dic):
                    results_dic[result_key_name] =
                        {
                        'E_out': None,
                        'E_out_description': None
                    }

                dict_item =
                    results_dic[result_key_name]

                clf.fit(fold_trainning_data_X,
                    fold_trainning_data_Y)
                clf_prediction =
                    clf.predict(fold_test_data_X)

                test_row_number =
                    fold_test_data_X.shape[0]
                errorsInPrediction = 0

                for prediction, real_test_Value
                    in zip(clf_prediction,
                    fold_test_data_Y):
                    if(prediction !=
                        real_test_Value):
                        errorsInPrediction += 1

                E_out =
                    errorsInPrediction/test_row_number

                if(dict_item['E_out'] == None or
                    dict_item['E_out'] > E_out):
                    dict_item['E_out'] = E_out
                    dict_item['E_out_description']
                        = 'Min E_out = ' +
                        str(E_out) + ' on k_fold '
                        + str(k)

                if(item['E_out'] == None or
                    item['E_out'] > E_out):
                    item['E_out'] = E_out
                    item['E_out_description'] =
                        'Min E_out = ' +
                        str(E_out) + ' on k_fold '
                        + str(k)
    ## end testing

    for key,value in results_dic.items():
        print(key + ': ' +
            value['E_out_description'])

    print('\nfinal result:')

    for item in typesOfSVMs:
        print(item['name']+ ': ' +
            item['E_out_description'])


    return None

test_svm('banana-data.dat')
```