Georgiy Voropayev T00570108

COMP 3710 Artificial Intelligence

November 9, 2019

''' Multiclass vs. multilable fitting 2d array'''

from sklearn.svm import SVC

from sklearn.multiclass import OneVsRestClassifier

from sklearn.preprocessing import LabelBinarizer

x = [[1, 2], [2, 4], [4, 5], [3, 2], [3, 1]]

y = [0, 0, 1, 1, 2]

classif = OneVsRestClassifier(estimator=SVC(gamma='scale', random\_state=0))

y = LabelBinarizer().fit\_transform(y)

print(classif.fit(x,y).predict(x))

**Output:**



''' The data should initially be in the (n\_samples, n\_features) shape.

    If it is not, then it should reshaped'''

digits = datasets.load\_digits()

print("Pringing shape of images: ")

print(digits.images.shape)

import matplotlib.pyplot as plt

plt.imshow(digits.images[-1], cmap=plt.cm.gray\_r)

data = digits.images.reshape((digits.images.shape[0], -1))

print("Reshaping images: ")

print(data)

''' KNN (k nearest neighbors) classification example '''

# Split iris data in train and test data

# A random permutation, to split the data randomly

import numpy as np

np.random.seed(0)

iris = datasets.load\_iris()

iris\_x = iris.data

iris\_y = iris.target

indices = np.random.permutation(len(iris\_x))

iris\_train\_x = iris\_x[indices[:-10]]

iris\_train\_y = iris\_y[indices[:-10]]

iris\_test\_x = iris\_x[indices[:-10]]

iris\_test\_y = iris\_y[indices[:-10]]

# Create and fit a nearest-neighbor classifier

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()

knn.fit(iris\_train\_x, iris\_train\_y)

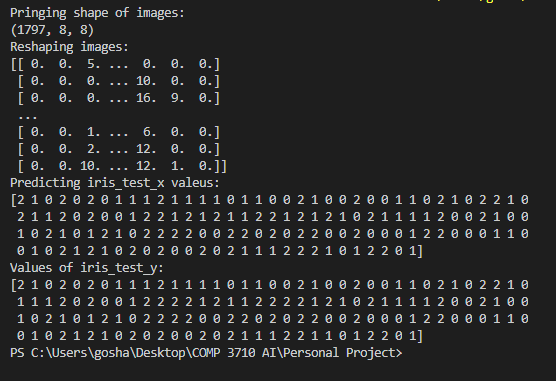
print("Predicting iris\_test\_x valeus: ")

print(knn.predict(iris\_test\_x))

print("Values of iris\_test\_y: ")

print(iris\_test\_y)

**Output:**



''' Linear model: from regression to sparsity '''

diabetes = datasets.load\_diabetes()

d\_x\_tr = diabetes.data[:-20]

d\_y\_tr = diabetes.data[-20:]

d\_x\_te = diabetes.target[:-20]

d\_y\_te = diabetes.target[-20:]

""" Linear Regression """

from sklearn import linear\_model

regr = linear\_model.LinearRegression()

regr.fit(d\_x\_tr, d\_y\_tr)

# print(regr.coef\_)

# print(regr.predict(d\_x\_te, d\_y\_te))

# Print the mean square error

print(np.mean((regr.predict(d\_x\_te) - d\_y\_te)\*\*2))

# 1 is perfect prediction and 0 means that there

# is no linear relationship between x nad y

regr.score(d\_x\_te, d\_y\_te)

""" Linear Regression """

from sklearn import linear\_model

# regr = linear\_model.LinearRegression()

# regr.fit(d\_x\_tr, d\_y\_tr)

# print(regr.coef\_)

# print(regr.predict(d\_x\_te, d\_y\_te))

# Print the mean square error

# print(np.mean((regr.predict(d\_x\_te) - d\_y\_te)\*\*2))

# 1 is perfect prediction and 0 means that there

# is no linear relationship between x nad y

# regr.score(d\_x\_te, d\_y\_te)

""" Shrinkage """

# If there are few data points per dimension, noise in

# the observations induces high variance:

x = np.c[.5,1].T

y = np.c[.5,1]

test = np.c\_[0,2].T

regr = linear\_model.LinearRegression()

import matplotlib.pyplot as plt

plt.figure()

np.random.seed(0)

for \_ in range(6):

    this\_x = .1 \* np.random.normal(size = (2,1)) + x

    regr.fit(this\_x, y)

    plt.plot(test, regr.predict(test))

    plt.scatter(this\_x, y, s=3)

 # If you have high-dimentional statistical learning

 # then you have to shrink your data to zero. Any two

 # random set are likely to be uncorrelated

 # (Called Ridge regression)

regr = linear\_model.Ridge(alpha = .1)

plt.figure()

for \_ in range(6):

    this\_x = .1 \* np.random.normal(size = (2,1)) + x

    regr.fit(this\_x, y)

    plt.plot(test, regr.predict(test))

    plt.scatter(this\_x, y, s=3)

# Due to some issues with the dataset, I cannot retrieve

# an output

# Choosing alpha to minimize left out error

alphas = np.logspace(-4, -1, 6)

print([regr.set\_params(alhpa=alpha).fit(d\_x\_tr, d\_y\_tr).score(d\_x\_te, d\_y\_te)

      for alpha in alphas])

'''

            ===========Exercise===========

Try classifying the digits dataset with nearest neighbors

and a linear model. Leave out the last 10% and test prediction

performance on these observations.

 '''

from sklearn import datasets, neighbors, linear\_model

digits = datasets.load\_digits()

x\_digits = digits.data / digits.data.max()

y\_digits = digits.target

n\_samples = len(x\_digits)

d\_x\_train = x\_digits[:int (.9 \* n\_samples)]

d\_y\_train = y\_digits[:int (.9 \* n\_samples)]

d\_x\_test = x\_digits[:int (.9 \* n\_samples):]

d\_y\_test = y\_digits[int (.9 \* n\_samples):]

model = neighbors.KNeighborsClassifier()

logistic = linear\_model.LogisticRegression(solver='lbfgs', max\_iter=1000,

                                           multi\_class='multinomial')

print("Model Score: " , model.fit(d\_x\_train, d\_y\_train).score(d\_x\_test, d\_y\_test))

print("Logistic score: " , logistic.fit(d\_x\_train, d\_y\_train).score(d\_x\_test, d\_y\_test))

**Output:**



'''

            ===========Exercise 2===========

Try classifying classes 1 and 2 from the iris dataset with SVMs,

with the 2 first features. Leave out 10% of each class and test

prediction performance on these observations.

Warning: the classes are ordered, do not leave out the last 10%,

you would be testing on only one class.

 '''

from sklearn import svm

import numpy as np

import matplotlib.pyplot as plt

iris = datasets.load\_iris()

x = iris.data

y = iris.target

x = x[y != 0, :2]

y = y[y != 0]

samples = len(x)

np.random.seed(0)

order = np.random.permutation(samples)

x = x[order]

y = y[order].astype(np.float)

iris\_train\_x = x[:int (.9 \* samples)]

iris\_train\_y = y[:int (.9 \* samples)]

iris\_test\_x = x[int (.9 \* samples):]

iris\_test\_y = y[int (.9 \* samples):]

for kernel in ('linear', 'poly', 'rbf'):

    svc = svm.SVC(kernel = kernel, gamma=10)

    svc.fit(iris\_train\_x, iris\_train\_y)

    plt.figure()

    plt.clf()

    plt.scatter(x[:,0], x[:, 1], c=y, zorder=10, cmap=plt.cm.Paired,

                edgecolor='k', s=20)

    # Circle out the test data

    plt.scatter(iris\_test\_x[:, 0], iris\_test\_x[:, 1], s=80, facecolors='none',

                zorder=10, edgecolor='k')

    plt.axis('tight')

    x\_min = x[:, 0].min()

    x\_max = x[:, 0].max()

    y\_min = x[:, 1].min()

    y\_max = x[:, 1].max()

    XX, YY = np.mgrid[x\_min:x\_max:200j, y\_min:y\_max:200j]

    Z = svc.decision\_function(np.c\_[XX.ravel(), YY.ravel()])

    # Put the result into a color plot

    Z = Z.reshape(XX.shape)

    plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)

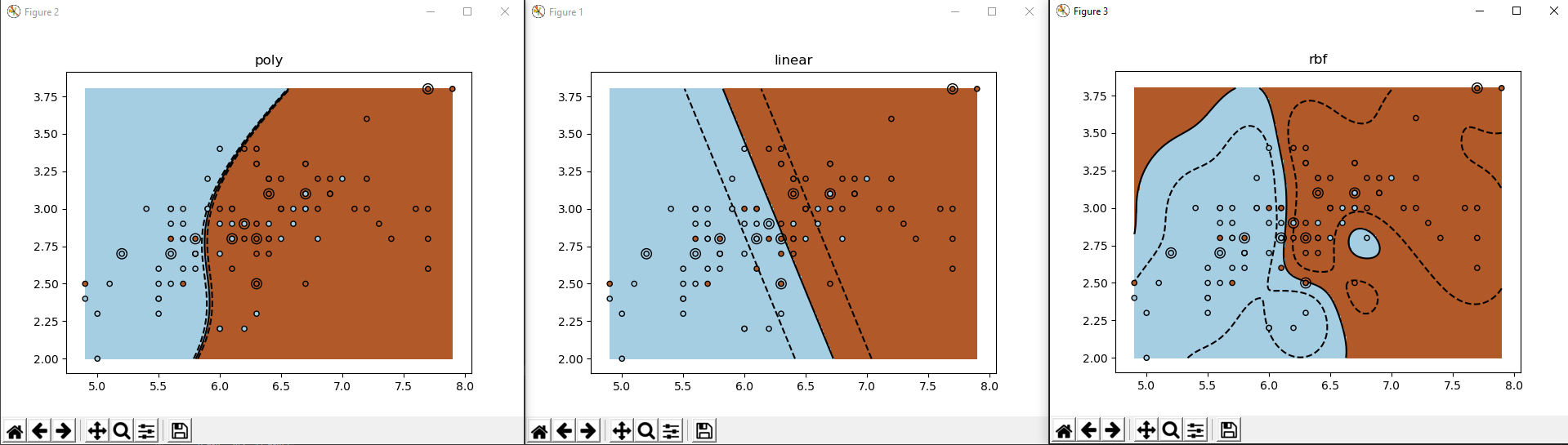
    plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],

                linestyles=['--', '-', '--'], levels=[-.5, 0, .5])

    plt.title(kernel)

plt.show()

**Output:**



''' Model selection: choosing estimators and their parameters '''

# Every estimator exposes a score method that can judge the quality

# of the fit (or the prediction) on new data. Bigger is better.

from sklearn import datasets, svm

digits = datasets.load\_digits()

x\_digits = digits.data

y\_digits = digits.target

svc = svm.SVC(C=1, kernel='linear')

svc.fit(x\_digits[:-100], y\_digits[:-100]).score(x\_digits[-100:], y\_digits[-100:])

# To get a better measure of prediction accuracy

# (which we can use as a proxy for goodness of fit of the model),

#  we can successively split the data in folds that we use for

# training and testing:

import numpy as np

x\_folds = np.array\_split(x\_digits, 3)

y\_folds = np.array\_split(y\_digits, 3)

scores = list()

for k in range(3):

    # Using 'list' to copy, in order to 'pop' later on

    x\_train = list(x\_folds)

    x\_test = x\_train.pop(k)

    x\_train = np.concatenate(x\_train)

    y\_train = list(y\_folds)

    y\_test = y\_train.pop(k)

    y\_train = np.concatenate(y\_train)

    scores.append(svc.fit(x\_train, y\_train).score(x\_test, y\_test))

print(scores)

**Output:**



''' Preprocessing data'''

""" Standardization, or mean removal and variance scaling """

# If a feature has a variance that is orders of magnitude larger

# than others, it might dominate the objective function and make

# the estimator unable to learn from other features correctly as expected.

# The function scale provides a quick and easy way to perform this

# operation on a single array-like dataset:

from sklearn import preprocessing

import numpy as np

x\_train = np.array([[1., -1., 2.],

                    [2., 0.,  0.],

                    [0., 1., -1.]])

x\_scaled = preprocessing.scale(x\_train)

print("Printing x\_scaled:")

print(x\_scaled)

# print(x\_scaled.mean(axis=0))

# print(x\_scaled.std(axis=0))

# The preprocessing module further provides a utility class StandardScaler

# that implements the Transformer API to compute the mean and standard deviation

# on a training set so as to be able to later reapply the same transformation

# on the testing set.

scaler = preprocessing.StandardScaler().fit(x\_train)

print("Printing scaler.mean\_:")

print(scaler.mean\_)

print("Printing scaler.scale\_:")

print(scaler.scale\_)

print("Printing scaler.transform(x\_train):")

print(scaler.transform(x\_train))

x\_test = [[-1., 1., 0.]]

print("Printing scaler.transform(x\_test):")

print(scaler.transform(x\_test))

""" Scalling features to a range """

""" MinMaxScaler """

# An alternative standardization is scaling features to lie between a given minimum

# and maximum value, often between zero and one, or so that the maximum absolute value

# of each feature is scaled to unit size.

min\_max\_scaler = preprocessing.MinMaxScaler()

x\_train\_minmax = min\_max\_scaler.fit\_transform(x\_train)

print("Printing x\_train\_minmax:")

print(x\_train\_minmax)

x\_test = np.array([[-3., -1., 4.]])

x\_test\_minmax = min\_max\_scaler.transform(x\_test)

print("Printing x\_test\_minmax:")

print(x\_test\_minmax)

""" MaxAbsScaler """

max\_abs\_scaler = preprocessing.MaxAbsScaler()

x\_train\_maxabs = max\_abs\_scaler.fit\_transform(x\_train)

print("Printing x\_train\_maxabs:")

print(x\_train\_maxabs)

x\_test\_maxabs = max\_abs\_scaler.transform(x\_test)

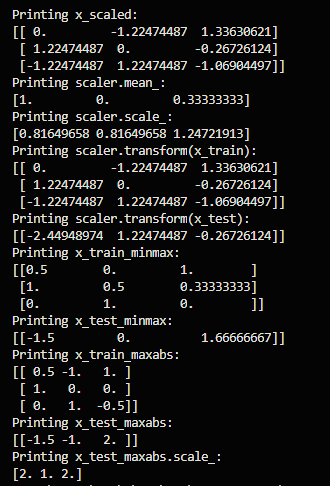
print("Printing x\_test\_maxabs:")

print(x\_test\_maxabs)

print("Printing x\_test\_maxabs.scale\_:")

print(max\_abs\_scaler.scale\_)

**Output:**



Work cited:

Akanksha, Rai. “Multiclass Classification Using Scikit-Learn.” GeeksforGeeks, 28 Oct. 2019, <https://www.geeksforgeeks.org/multiclass-classification-using-scikit-learn/>.

“An Introduction to Machine Learning.” ScikitLearn, 1 Nov. 2019, <https://scikit-learn.org/stable/tutorial/basic/tutorial.html#type-casting>