Freie Universität Berlin

Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 4

Boyan Hristov, Nedeltscho Petrov

15. November 2017

Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework4

Fischer Klassifikation

Score (Output des Programs) bzw. Analyse

Das ist die Ausgabe des Programs und damit auch das Score

```
Best score for seed=879: 0.930510314875
Worst score for seed530: 0.880564603692
Score: 0.9305103148751357
Score with linear regression: 0.9055374592833876
```

Wie wir hier sehen, ist das Score auch sehr davon abhängig, wie man die Daten in Test und Train Sets splitet. Es ist deswegen so, weil einige Emails sehr untypische spam Emails bzw nicht spam Emails sind. Deswegen ist unser Score zwischen 88

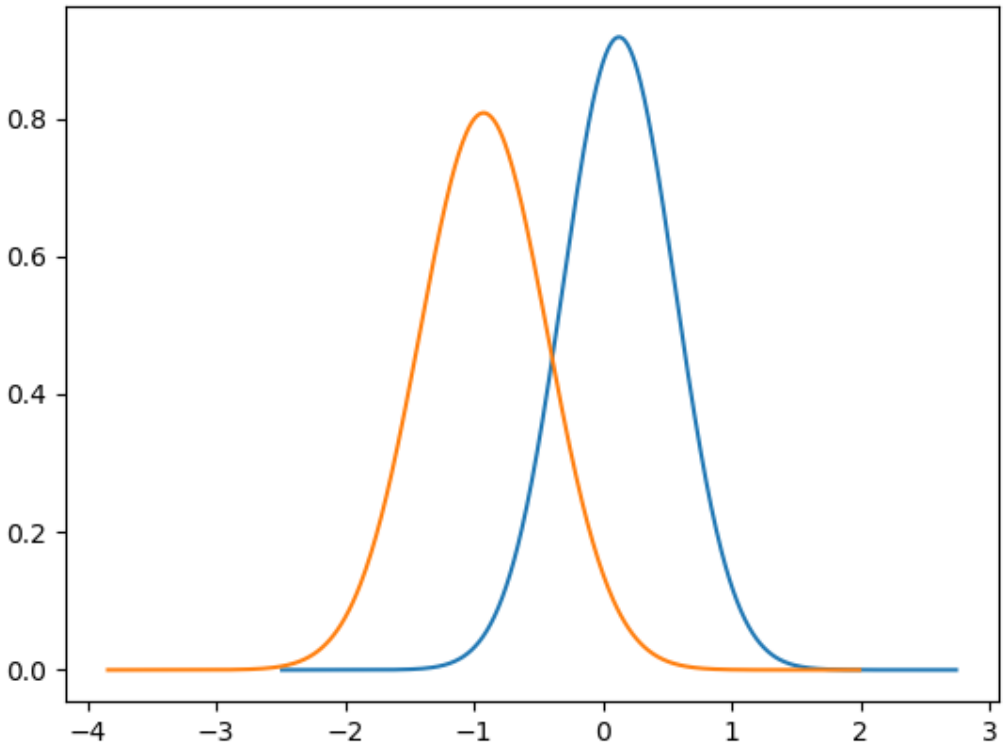
Bei linearen Regression haben wir etwas schlechteres Ergebniss mit dem Datensatz bekommen (lineare Regression von scikit-learn wurde benutzt). Das ist deswegen so, wie in der Vorlesung erklärt, weil die Kovarianzmatritzen in dem Fall für Fischer Klassifizierung besser geeignet sind. Anders gesagt, die Streuung der Daten von je Klasse liegen teilweise othogonal zu einander.

Aus performance-sichten haben wir nicht erkennbar bessere Ergebnisse bekommen (eine weitere lineare Methode, ist schnell genug).

Der Datensatz wurde im Jahre 1999 veröffentlicht, die Ergebnisse von 1998 zeigen circa 7Da wir aber deutlich später das entwickelt haben, denken wir nicht, dass unsere Ergebnisse ausreichend sind. Wie schon auf der Webseite des Datensatzes erleutert wurde, sind falsch als Spam erkannte Emails ganz schädlich und 7ist in dem Fall schon viel.

Plot

Auf dem Plot haben wir die Dichtefunktion, projiziert auf dem Fischer-Vektor berechnet. Wie man sieht, überlappen schon die Klassen sich einigermaßen. Man könnte also auch bessere Ergebnisse bekommen, könnte aber auch viel schlimmer sein.



Details zur Implementierung

In der Fit Methode haben wir beide Kovarianzmatritzen berechnet, als auch die Medians. Anhand der Formel von der Vorlesung haben wir dann unser Fischer Vektor berechnet. Nachdem, haben wir den Mittelpunkt beider Zentren auf dem Fischer-Vektor projiziert, damit wir dem später für die Klassifizierung benutzen können.

```
    def fit(self, X_train, y_train):
        X_a, X_b = FisherClassifier.split_in_classes(X_train, y_train)
        cov_mat_a = np.cov(X_a, rowvar=False, bias=True)
        cov_mat_b = np.cov(X_b, rowvar=False, bias=True)
        center_a = np.array(X_a, dtype=np.float64).mean(0)
        center_b = np.array(X_b, dtype=np.float64).mean(0)

        alpha = np.linalg.pinv(cov_mat_a + cov_mat_b).dot(center_a - center_b)
        alpha_normalized = alpha / np.linalg.norm(alpha)
        self.alpha = alpha_normalized

        # to determine whether a point belongs to class a or class b we need a threshold
        # on the 1 dimensional space. This one is the projected point between the 2 centers
```

```
        self.threshold = self.project_point((center_a + center_b) / 2)

        self.plot_probability_distribution(center_a, center_b, X_a, X_b)
```

Die eigentliche Klassifizierung ist ganz simpel. Wir projizieren das neue Punkt auf dem Fischer-Vektor und schauen, ob es vor oder nach unser in Fit definierten Threshold ist.

```
    def predict_single(self, x):
        # project x into alpha (AKA Fisher's vector)
        projected = self.project_point(x)
        return projected < self.threshold
```

Die ganze Plot Funktionalität ist nicht so interessant, außer 2 relevante Teile. Wir haben die folgende Funktion benutzt, um die Dichtefunktion zu berechnen (da bei Fischer Normalverteilung eine Voraussetzung ist)

```
@staticmethod
    def get_density_function(center, covariance):
        return lambda x: math.e ** (
            (-1/2) * ((x - center) / covariance) ** 2
        ) / (covariance * math.sqrt((2*math.pi)))
```

Dabei brauchen wir aber die Kovarianz. Die wird von alle projizierten Punkten so berechnet.

```
@staticmethod
    def get_covariance_for_projected(points, center):
        vectorized_sq_distances_sum = np.vectorize(lambda x, m: (x - m)**2)
        square_distances_sum = np.sum(vectorized_sq_distances_sum(points, center))
        return math.sqrt(square_distances_sum / len(points))
```

Code in FisherClassifier.py

```
from Classifier import Classifier
from Parser import get_data_set
import numpy as np
from sklearn.linear_model import LinearRegression
import math
from random import random
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal


class FisherClassifier(Classifier):
    def __init__(self, Ximport csv
import numpy as np
import os
from sklearn.model_selection import train_test_split


def parse_data():
    file_name = os.path.join(os.path.dirname(__file__), './Dataset/spambase.data')
    csv_file = open(file_name, 'rt')
    reader = csv.reader(csv_file, delimiter=',', quoting=csv.QUOTE_NONE)
    data = []

    for row in reader:
        filtered = list(filter(lambda x: x != '', row))
        data.append(list(map(lambda x: float(x), filtered)))

    return data
```

```
def get_points_and_labels_from_data(data):
    points = np.array(list(map(lambda x: x[:-1], data)), dtype=np.float64)
    labels = np.array(list(map(lambda x: int(x[-1]), data)))

    return points, labels


def get_data_set(seed):
    data = parse_data()
    X, y = get_points_and_labels_from_data(data)
    # for determined results we use a seed for random_state, so that data is always split
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size
    =0.2,
                                                        random_state=seed)

    return X_train, X_test, y_train, y_test
_train, y_train):
        self.alpha = None
        self.threshold = None
        self.fit(X_train, y_train)

    @staticmethod
    def split_in_classes(X_train, y_train):
        split_X = ([], [])

        for idx, x in enumerate(X_train):
            current_label = y_train[idx]
            split_X[current_label].append(x)

        return split_X

    @staticmethod
    def get_density_function(center, covariance):
        return lambda x: math.e ** (
            (-1/2) * ((x - center) / covariance) ** 2
        ) / (covariance * math.sqrt((2*math.pi)))

    @staticmethod
    def get_covariance_for_projected(points, center):
        vectorized_sq_distances_sum = np.vectorize(lambda x, m: (x - m)**2)
        square_distances_sum = np.sum(vectorized_sq_distances_sum(points, center))
        return math.sqrt(square_distances_sum / len(points))

    def plot_class(self, points, center):
        projected_center = self.project_point(center)
        projected_points = list(map(lambda x: self.project_point(x), points))
        covariance = FisherClassifier.get_covariance_for_projected(projected_points,
    projected_center)
        density_a = FisherClassifier.get_density_function(projected_center, covariance)
        plot_distance = 5000
        y_of_plot = [density_a(float(x) / 100) for x in range(-plot_distance, plot_distance
    )]
        x_to_plot = [float(x) / 100 for x in range(-plot_distance, plot_distance)]
        first_decent = None
        last_decent = None

        for idx, y in enumerate(y_of_plot):
            if y > 0.01 and first_decent is None:
                first_decent = idx
            elif y <= 0.01 and first_decent is not None:
                last_decent = idx
                break

        beauty_margin = (last_decent - first_decent)
        start = int(first_decent - beauty_margin/2)
        end = int(last_decent + beauty_margin/2)
```

```
        plt.plot(x_to_plot[start:end], y_of_plot[start:end])

    def plot_probability_distribution(self, center_a, center_b, points_a, points_b):
        self.plot_class(points_a, center_a)
        self.plot_class(points_b, center_b)
        plt.show()

    def project_point(self, x):
        return x.dot(self.alpha)

    def fit(self, X_train, y_train):
        X_a, X_b = FisherClassifier.split_in_classes(X_train, y_train)
        cov_mat_a = np.cov(X_a, rowvar=False, bias=True)
        cov_mat_b = np.cov(X_b, rowvar=False, bias=True)
        center_a = np.array(X_a, dtype=np.float64).mean(0)
        center_b = np.array(X_b, dtype=np.float64).mean(0)

        alpha = np.linalg.pinv(cov_mat_a + cov_mat_b).dot(center_a - center_b)
        alpha_normalized = alpha / np.linalg.norm(alpha)
        self.alpha = alpha_normalized

        # to determine whether a point belongs to class a or class b we need a threshold
        # on the 1 dimensional space. This one is the projected point between the 2 centers
        self.threshold = self.project_point((center_a + center_b) / 2)

        self.plot_probability_distribution(center_a, center_b, X_a, X_b)

    def predict(self, X):
        return list(map(lambda x: self.predict_single(x), X))

    def predict_single(self, x):
        # project x into alpha (AKA Fisher's vector)
        projected = self.project_point(x)
        return projected < self.threshold


# max_score = 0
# min_score = 100
# best_seed = 0
# worst_seed = 0

# for i in range(1000):
#     X_train, X_test, y_train, y_test = get_data_set(i)
#     classifier = FisherClassifier(X_train, y_train)
#     score = classifier.score(X_test, y_test)
#     if score > max_score:
#         max_score = score
#         best_seed = i
#     if score < min_score:
#         min_score = score
#         worst_seed = i
#
# print('Best score for seed={}: {}'.format(best_seed, max_score))
# print('Worst score for seed{}: {}'.format(worst_seed, min_score))

X_train, X_test, y_train, y_test = get_data_set(879)
classifier = FisherClassifier(X_train, y_train)
score = classifier.score(X_test, y_test)
print('Score: {}'.format(classifier.score(X_test, y_test)))

lm = LinearRegression()
y_train_modified = list(map(lambda x: 1 if x == 1 else -1, y_train))
lm.fit(X_train, y_train_modified)
prediction = np.array(list(map(lambda x: 1 if x > 0 else 0, lm.predict(X_test))), dtype=np.
    float64)
score = np.mean(prediction == np.array(y_test, dtype=np.float64))
print('Score with linear regression: {}'.format(score))
```

Code in Parser.py

```
import csv
import numpy as np
import os
from sklearn.model_selection import train_test_split


def parse_data():
    file_name = os.path.join(os.path.dirname(__file__), './Dataset/spambase.data')
    csv_file = open(file_name, 'rt')
    reader = csv.reader(csv_file, delimiter=',', quoting=csv.QUOTE_NONE)
    data = []

    for row in reader:
        filtered = list(filter(lambda x: x != '', row))
        data.append(list(map(lambda x: float(x), filtered)))

    return data


def get_points_and_labels_from_data(data):
    points = np.array(list(map(lambda x: x[:-1], data)), dtype=np.float64)
    labels = np.array(list(map(lambda x: int(x[-1]), data)))

    return points, labels


def get_data_set(seed):
    data = parse_data()
    X, y = get_points_and_labels_from_data(data)
    # for determined results we use a seed for random_state, so that data is always split
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size
    =0.2,
                                                        random_state=seed)

    return X_train, X_test, y_train, y_test
```