Freie Universität Berlin

Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 3

Boyan Hristov, Nedeltscho Petrov

7. November 2017

Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework3

Klassifikation mit Gauss Verteilung

Score (Output des Programs) bzw. Analyse

Das ist die Ausgabe des Programs und damit auch das Score

```
Score 3 vs 5: 0.923312883436
Score 3 vs 7: 0.929712460064
Score 3 vs 8: 0.843373493976
Score 5 vs 7: 0.931596091205
Score 5 vs 8: 0.874233128834
Score 7 vs 8: 0.837060702875
Score 3 vs 5 vs 7 vs 8: 0.838810641628
Score all: 0.840558046836
Score all with train data: 0.969551501852
```

Leider haben wir sehr komisches Score bei der letzten Hausaufgabe bekommen und können das jetzige anhand von selbsterstellte Daten von linearen Regression vergleichen. Mit den Daten aus der Musterlösung, die im Tutorium vorgestellt wurde, wurden wir sagen, dass die jetzige Lösung mit Gauss Verteilung nicht besser (sogar schlechter) ist, als diese mit linearen Regression (da war die Unterscheidung zwischen 5 und 7 mit $94+\%$ Genauigkeit).

Wenn man alle Ziffer vergleicht, bekommt man ein Score von 84.0558046836% Genauigkeit mit den Testdaten und 96.9551501852% Genauigkeit mit den Traindaten. D.h. wir lernen immer noch ganz viel äuswendig"bzw. es gibt relativ viel överfitting".

Das Programm funktioniert ungefähr so schnell wie bei der Klassifikation mit linearen Regression. Die Implementierung war aber schwieriger und kostet mehr Speicher damit es performant ist (Zentren, Konvergenzmatritzen, pseudo-invertierte Konvergenzmatritzen und Determinanten). Bei der Implementierung mit linearen Regression musste man nur die "Hat"Matritzen speichern.

Es ist aber leichter für mehrere Klassen zu implementieren. Man muss nicht separat die Ergebnissen vom mehreren Klassifikatoren vergleichen um zwischen mehrere Klassen zu unterscheiden. Man kann direkt

die Wahrscheinlichkeit, dass ein Punkt zu einer Klasse gehört berechnen, und dann direkt die Klasse mit der Höhsten Wahrscheinlichkeit wählen.

Erklärung interessanteren Teilen des Codes

Die Fit Methode ist diesmal nicht besonders interessant, da wir dort nur die Zentren berechnen (Alle Punkte aufsummieren und durch ihre Anzahl teilen) und die Konvergenzmatrixberechnungsfunktion aufrufen. Diese wäre aber interessant, da wir die vektorisiert haben und die Hauptimplementierung ausgelagert haben.

Die Summe und die Teilung durch der Anzahl kann man hier auch sehen. Sonst haben wir numpy.vectorize benutzt, um die Berechnung der einzelnen Summaren zu beschleunigen.

```
self.covariance_matrix[label] = np.vectorize(GaussianClassifier.covariance_for_point,
    signature='(m),(n)->(m,m)')(
                points_per_label[label], self.centers[label]).sum(axis=0) / len(
    points_per_label[label])
```

Wir nutzen hier die Formel aus der Vorlesung. Das einzigste Unterschied hier ist, dass wir den ersten Vektor transponieren und nicht der zweiten. Das ist so, da wir am Ende eine Matrix bekommen wollen und Numpy interpretiert eine 1-dimensionale Liste als eine Matritze der Form 1xn und nicht mx1 (Vektor) als wir diese betrachten wollen und umgekehrt für das zweite.

```
def covariance_for_point(point, center):
        return np.matrix(point - center, dtype=np.float64).T.dot(np.matrix(point - center,
    dtype=np.float64))
```

Bei der Berechnung der Wahrscheinlichkeit haben wir auch die Formel aus der Vorlesung benutzt. Wichtig wäre hier, dass es zu eine Determinante $= 0$ kommen könnte, wenn wir linear abhängige Features haben. Das ist oft der Fall bei manchen Ziffern, da wir die Pixels als Features betrachten und diese oft abhängig sind. Z.b wenn eine Ziffer ein Horizontales Strich hat, dann falls ein Pixel grauer ist, sind mit größem Wahrscheinlichkeit auch diese daneben (links und rechts) auch grauer.

Damit wir die 0 Determinante umgehen (wegen Division mit 0), haben wir $2 * \pi * ||X||$ ausgelagert und benutzen später das Minimum von die und eine kleine Delta.

```
def get_possibility_for_class(self, point_class, point):
        two_pi_det = 2 * math.pi * self.covariance_matrix_det[point_class]
        left_side = 1 / max(0.2, math.sqrt(two_pi_det))
        right_side = math.e**(-0.5 * (point - self.centers[point_class]).T.
                              dot(self.covariance_matrix_pinv[point_class]).dot(point -
    self.centers[point_class]))

        return left_side * right_side
```

Die Voraussage machen wir, in dem wir die Wahrscheinlichkeit für je Klasse berechnen und dann diese mit höhsten nehmen.

```
possibilities = list(map(lambda x: self.get_possibility_for_class(x, point), self.classes))
        winning_index = possibilities.index(max(possibilities))
        return self.classes[winning_index]
```

Vollständiges Code

```
from Classifier import Classifier
from Parser import *
import numpy as np
import math


class GaussianClassifier(Classifier):
    @staticmethod
    def covariance_for_point(point, center):
        # calculate covariance for a single point (well not really, but a single summar
    thingy)
        # idea of this method is to easily vectorize it with np.vectorize

        # implementation specific:
        # in the formula, the first vector should be transposed and the second not
        # this is only done, because we need to receive a matrix at the end
        # with the way numpy handles single vectors, we actually need to transpose the
    first one
        # and not the second in order to do that what we are used to in math
        # (numpy treats a 1-dimensional array as 1xn matrix and not as nx1 as we want to)
        return np.matrix(point - center, dtype=np.float64).T.dot(np.matrix(point - center,
    dtype=np.float64))

    def __init__(self, train_data, classes = [x for  x in range(10)]):
        """
        :param classes: list of classes the classifier should train itself to distinguish
                        (e.g [3,5] for 3 vs 5 classifier) default is all digits
        :param trainData:
        :param trainLabels:
        :param testData:
        :param testLabels:
        """

        self.centers = {}
        self.covariance_matrix = {}
        self.covariance_matrix_det = {}
        self.covariance_matrix_pinv = {}
        (train_labels, train_points) = get_labels_and_points_from_data(train_data, classes)
        self.classes = classes
        self.fit(train_labels, train_points)

    def fit(self, train_labels, train_points):
        assert(len(train_labels) == len(train_points))
        points_per_label = {}

        # sort points in a dictionary, separated by classes
        # eg {3: [first 256 dimension vector, second 256 dimensional vector, etc.], 5: ...
    , ...}
        for idx, point in enumerate(train_points):
            current_label = train_labels[idx]
            if current_label not in points_per_label:
                points_per_label[current_label] = [point]
            else:
                points_per_label[current_label].append(point)

        # then for each class, find the centroid and the covariance matrix
        # for optimization reasons, we also save the inverse of the covariance matrix and
    it's determinant
        for label in points_per_label:
            # average of all points from the current class (with axis 0, so row-wise
    average)
            self.centers[label] = np.array(points_per_label[label], dtype=np.float64).mean
    (0)
            # calculate covariance matrix using vectorization (see covariance_for_point
    static method)
```

```
            # using the formula 1/n*(sum_i((point-center)(point-center)T))
            self.covariance_matrix[label] = np.vectorize(GaussianClassifier.
    covariance_for_point, signature='(m),(n)->(m,m)')(
                points_per_label[label], self.centers[label]).sum(axis=0) / len(
    points_per_label[label])
            # also calculate and save determinant and pseudo-inverse of matrix for
    performance reasons
            self.covariance_matrix_det[label] = np.linalg.det(self.covariance_matrix[label
    ])
            self.covariance_matrix_pinv[label] = np.linalg.pinv(self.covariance_matrix[
    label])

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

    def predict_single(self, point):
        possibilities = list(map(lambda x: self.get_possibility_for_class(x, point), self.
    classes))
        winning_index = possibilities.index(max(possibilities))

        return self.classes[winning_index]

    def get_possibility_for_class(self, point_class, point):
        # using the formula from the lecture, calculate the probability for a point with
    coordinates to
        # be part of a class

        # only important thing here is that 2*pi*det(covariance_matrix) can be zero
        # (in case the covariance_matrix doesn't have a full rank (when we have identical
    values for some features
        # this can often be the case because of the white pixels at the edges)),
        # so we use
        # np.nextafter to replace any zeros with a reaaaaly small float (because of
    DivideByZero exceptions...)

        two_pi_det = 2 * math.pi * self.covariance_matrix_det[point_class]
        left_side = 1 / max(0.2, math.sqrt(two_pi_det))
        right_side = math.e**(-0.5 * (point - self.centers[point_class]).T.
                              dot(self.covariance_matrix_pinv[point_class]).dot(point -
    self.centers[point_class]))

        return left_side * right_side

train_data = parse_data_file('./Dataset/train')
test_data = parse_data_file('./Dataset/test')

three_vs_five = GaussianClassifier(train_data, [3,5])
(three_vs_five_test_labels, three_vs_five_test_data) = get_labels_and_points_from_data(
    test_data, [3,5])
print("Score 3 vs 5: {}%".format(three_vs_five.score(three_vs_five_test_data,
    three_vs_five_test_labels)))

three_vs_seven = GaussianClassifier(train_data, [3,7])
(three_vs_seven_test_labels, three_vs_seven_test_data) = get_labels_and_points_from_data(
    test_data, [3,7])
print("Score 3 vs 7: {}%".format(three_vs_seven.score(three_vs_seven_test_data,
    three_vs_seven_test_labels)))

three_vs_eight = GaussianClassifier(train_data, [3,8])
(three_vs_eight_test_labels, three_vs_eight_test_data) = get_labels_and_points_from_data(
    test_data, [3,8])
print("Score 3 vs 8: {}%".format(three_vs_eight.score(three_vs_eight_test_data,
    three_vs_eight_test_labels)))

five_vs_seven = GaussianClassifier(train_data, [5,7])
(five_vs_seven_test_labels, five_vs_seven_test_data) = get_labels_and_points_from_data(
    test_data, [5,7])
```

```
print("Score 5 vs 7: {}%".format(five_vs_seven.score(five_vs_seven_test_data,
    five_vs_seven_test_labels)))

five_vs_eight = GaussianClassifier(train_data, [5,8])
(five_vs_eight_test_labels, five_vs_eight_test_data) = get_labels_and_points_from_data(
    test_data, [5,8])
print("Score 5 vs 8: {}%".format(five_vs_eight.score(five_vs_eight_test_data,
    five_vs_eight_test_labels)))

seven_vs_eight = GaussianClassifier(train_data, [7,8])
(seven_vs_eight_test_labels, seven_vs_eight_test_data) = get_labels_and_points_from_data(
    test_data, [7,8])
print("Score 7 vs 8: {}%".format(seven_vs_eight.score(seven_vs_eight_test_data,
    seven_vs_eight_test_labels)))

combined = GaussianClassifier(train_data, [3, 5, 7, 8])
(combined_test_labels, combined_test_data) = get_labels_and_points_from_data(test_data,
    [3,5,7,8])
print("Score 3 vs 5 vs 7 vs 8: {}%".format(combined.score(combined_test_data,
    combined_test_labels)))

all_digits = [x for x in range(10)]
all_classifier = GaussianClassifier(train_data, all_digits)
(all_test_labels, all_test_data) = get_labels_and_points_from_data(test_data, all_digits)
print("Score all: {}%".format(all_classifier.score(all_test_data, all_test_labels)))

all_digits = [x for x in range(10)]
all_classifier = GaussianClassifier(train_data, all_digits)
(all_train_labels, all_train_data) = get_labels_and_points_from_data(train_data, all_digits
    )
print("Score all: {}".format(all_classifier.score(all_train_labels, all_train_data)))
```

Und das simple Parser

```
import csv
import numpy as np


def parse_data_file(file_name):
    file = open(file_name, 'rt')
    reader = csv.reader(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_labels_and_points_from_data(data, classes):
    data = list(filter(lambda x: int(x[0]) in classes, data))
    labels = np.array(list(map(lambda x: int(x[0]), data)))
    points = np.array(list(map(lambda x: x[1:], data)), dtype=np.float64)

    return labels, points
```