Freie Universität Berlin

Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 7

Boyan Hristov, Nedeltscho Petrov

3. Dezember 2017

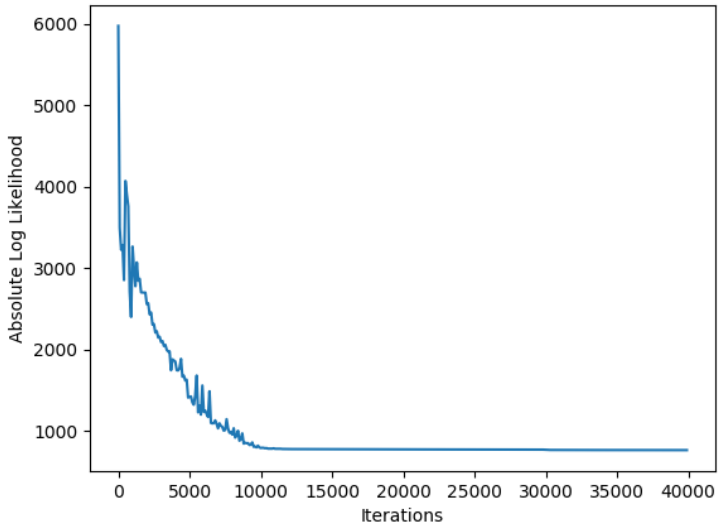
Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework7

Logistische Regression

Ausgabe des Programms und damit Score

```
Score: 0.9294245385450597
[[541  42]
 [ 23 315]]
Score from sklearn: 0.9370249728555917
```

Absolute Wert von Log Likelihood während der Iteration



Wie kann man eine Klasse bevorzugen

Wir können den ersten Koeffizient in dem Gewichtsvektor etwas ändern. Falls wir es höher setzen, werden wir die positive Klasse bevorzugen und umgekehrt.

Erklärung des Codes

Transformation

Wir normalisieren immer die Daten, in dem Wir je Feature duch den Varianz aller solchen Features teilen. Diese transform Methode wird vor dem Fitting und auch vor je Vorhersage gemacht. Weiter fügen wir hier auch die Spalte mit 1. ein.

```
    def transform(self, X, y):

        # feature_means = X.sum(0) / len(X[0])
        # X_t = X.T
        # variance = [((X_t[i] - feature_means[i])**2).sum() for i in range(len(X_t))]

        # numpy does it more effectively and normalized
        self.transformation_vector = np.var(X, axis=0)
        X = X / self.transformation_vector

        ones = np.ones((len(X), 1), dtype=np.float64)
        X = np.append(ones, X, axis=1)

        return X, y
```

Fit Methode

In der Fit Methode versuchen wir, die Likelihood zu maximieren. Das machen wir anhand der Gradientensteigerung der Log-likelihood. Um das zu approximieren, nutzt man ein learning rate, da man von den Gradientensteigerung nur erfährt, in welcher Richtung unsere Lösung sich befindet. Die Implementierung benutzt für die Berechnung der Steigerung die in der Vorlesung besprochene Formeln, deswegen ist hier die Lernrate interessanter.

Da wir mit einer statischen Lernrate etwas schlechtere Ergbebnisse (um circa 2etwas angepasst). Laut die Methode, muss man jedes mal, wenn man ein schlechteres Ergebniss bekommt, die Lernrate halbieren und das vorherige Gewichtsvektor nehmen. Sonst darf man die lernrate um 5aber 3

```
def fit(self, X, y, iterations, plot):
        features_len = len(X[0])
        self.beta = np.zeros(features_len, dtype=np.float64)
        log_error_over_time = []

        last_log_error = float('inf')
        current_learning_rate = self.learn_rate

        for i in range(iterations):
            weighted = X.dot(self.beta)
            probabilities = LogisticRegression.sigmoid(weighted)
            directions = y - probabilities
            gradient = X.T.dot(directions)

            self.learn_rate = self.learn_rate / 2
            self.beta = self.beta + current_learning_rate*gradient

            if i % 100 == 0:
                current_log_error = abs(self.get_log_likelihood(X, y))

                if current_log_error < last_log_error:
                    current_learning_rate += current_learning_rate * 0.0003  # increase by
    0.03%
                else:
                    self.beta = last_beta
                    current_learning_rate -= current_learning_rate * 0.03 # decrease by 3%

                last_log_error = current_log_error
                last_beta = np.copy(self.beta)
```

Predict Methode

Die predict Methode ist ganz simpel. Deswegen wäre hier auch interessant die get_probability Methode, die haben wir aber 1 zu 1 aus der Vorlesung genommen.

```
    def predict_single(self, x):
        return 1 if self.get_probability(x, 1) > 0.5 else 0

    def get_probability(self, x, y):
        return 1 / (1 + math.exp((-y * self.beta).T.dot(x)))
```

Analyse booleschen Funktionen

Output des Programs

```
beta and vs or: [ 0.11601896  0.53124725  0.53124725 -1.12376802]
transformed X and vs or: [[ 1.  0.  0.  0.]
 [ 1.  0.  4.  0.]
 [ 1.  4.  0.  0.]
 [ 1.  4.  4.  4.]
 [ 1.  0.  0.  0.]
 [ 1.  0.  4.  4.]
 [ 1.  4.  0.  4.]
 [ 1.  4.  4.  4.]]
Predicted: [1, 1, 1, 0, 1, 0, 0, 0]; Expected: [ 1.  1.  1.  1.  0.  0.  0.  0.]
beta and vs xor: [ 0.0795093   0.09251593  0.09251593 -0.29707532]
transformed X and vs xor: [[ 1.          0.          0.          0.        ]
 [ 1.          0.          4.          0.        ]
 [ 1.          4.          0.          0.        ]
 [ 1.          4.          4.          4.26666667]
 [ 1.          0.          0.          0.        ]
 [ 1.          0.          4.          4.26666667]
 [ 1.          4.          0.          4.26666667]
 [ 1.          4.          4.          0.        ]]
Predicted: [1, 1, 1, 0, 1, 0, 0, 1]; Expected: [ 1.  1.  1.  1.  0.  0.  0.  0.]
beta or vs xor: [-0.18989504 -0.115224   -0.115224    0.26844163]
transformed X or vs xor: [[ 1.          0.          0.          0.        ]
 [ 1.          0.          4.          4.26666667]
 [ 1.          4.          0.          4.26666667]
 [ 1.          4.          4.          4.26666667]
 [ 1.          0.          0.          0.        ]
 [ 1.          0.          4.          4.26666667]
 [ 1.          4.          0.          4.26666667]
 [ 1.          4.          4.          0.        ]]
Predicted: [0, 1, 1, 1, 0, 1, 1, 0]; Expected: [ 1.  1.  1.  1.  0.  0.  0.  0.]
```

Analyse

Bei AND vs OR sehen wir, dass das Endergebniss die entschedentste Rolle hat mit Koeffizient von -1.124. Das klingt auch logisch, da die AND Funktion seltener ein Wert von 1 zurückgibt als ein Wert von 0. $1 - \frac{1}{1+e^{-(0.11601896-1.124*4)}} \approx 98.7\%$, dass es keine AND ist, wenn die Funktion 1 zurückgibt. Die zwei Eingabewerte haben je ein Einfluss von $1 - \frac{1}{1+e^{-(0.11601896+0.53124725*4)}} \approx 9.6\%$, dass es eine OR Funktion ist.

Wir können sehen, dass unsere Theorie stimmt, da das Algorithmus bei allen Eingaben mit Endergebniss 1 eine OR Funktion vorhergesehen haben. Das kann man auch ausrechen: $1-\frac{1}{1+e^{-(0.11601896+2*4*0.53124725-1.124*4)}} \approx 53.2\%$ dass es eine AND Funktion ist.

Bei AND vs XOR sieht man wieder, dass das Endergebniss beides in der Normalisierung und auch bei dem Beta-Koeffizeint das größte Einfluss hat. Das ist so, da 1 XOR 0 = 1, 0 XOR 1 = 1, wobei AND bei den gleichen Werten 0 zurückgibt. Der Koeffizient ist aber nicht so groß wie bei AND vs OR, da 1 XOR 1 = 0 und 1 AND 1 = 1, also das Gegenbeispiel. Wir sehen aber wieder, dass das Algorithmus immer bei Endergebniss 1 eine XOR vorhersagt.

OR und XOR kann man nur anhand von dem letzten Sample von dem Datensatz unterscheiden. Also 1 OR 1 = 1 und 1 XOR 1 = 0. Andere Eingaben liefern gleiche Ausgaben. Deswegen, ein Endergebniss von 1 wurde eine OR bevorzugen (Koeffizient 0.26844163). Wir sehen auch, dass das Algorithmus nur bei diesen Eingaben bei beiden Klassen keine Fehler macht.

Vollständiges Code

LogisticRegression.py

```
from Classifier import Classifier
import numpy as np
import math
from matplotlib import pyplot as plt
import os


class LogisticRegression(Classifier):
    # def __init__(self, X_train, y_train, learn_rate=1e-3):
    def __init__(self, X_train, y_train, learn_rate=1e-3, iterations=40000, plot=False):
        self.beta = None
        self.transformation_vector = None
        self.learn_rate = learn_rate
        X, y = self.transform(X_train, y_train)
        self.fit(X, y, iterations, plot)

    @staticmethod
    def sigmoid(weighted):
        return 1 / (1 + np.exp(-weighted))

    def transform(self, X, y):

        # feature_means = X.sum(0) / len(X[0])
        # X_t = X.T
        # variance = [((X_t[i] - feature_means[i])**2).sum() for i in range(len(X_t))]

        # numpy does it more effectively and normalized
        self.transformation_vector = np.var(X, axis=0)
        X = X / self.transformation_vector

        # add ones
        ones = np.ones((len(X), 1), dtype=np.float64)
        X = np.append(ones, X, axis=1)

        return X, y

    def fit(self, X, y, iterations, plot):
        features_len = len(X[0])
        self.beta = np.zeros(features_len, dtype=np.float64)
        log_error_over_time = []

        last_log_error = float('inf')
        current_learning_rate = self.learn_rate

        for i in range(iterations):
            weighted = X.dot(self.beta)
            probabilities = LogisticRegression.sigmoid(weighted)
            directions = y - probabilities
            gradient = X.T.dot(directions)

            # Bold driver technique
            # If error was actually larger (overshooting) use previous weight vector
            # and decrease learning rate by 50%; otherwise increase learn rate by 5%

            # self.beta = self.beta + (self.learn_rate / (2**(i/5000)))*gradient
            self.learn_rate = self.learn_rate / 2
            self.beta = self.beta + current_learning_rate*gradient

            if i % 100 == 0:
                current_log_error = abs(self.get_log_likelihood(X, y))

                if current_log_error < last_log_error:
```

```
                    current_learning_rate += current_learning_rate * 0.0003  # increase by
    0.03%
                else:
                    self.beta = last_beta
                    current_learning_rate -= current_learning_rate * 0.03 # decrease by 3%

                last_log_error = current_log_error
                last_beta = np.copy(self.beta)

                if plot:
                    log_error_over_time.append(current_log_error)

        if plot:
            plt.ylabel('Absolute Log Likelihood')
            plt.xlabel('Iterations')
            plt.plot([i for i in range(0, iterations, 100)], log_error_over_time)
            plt.savefig(os.path.join(os.path.dirname(__file__), 'll_over_time.png'))

    def predict(self, X):
        X = X / self.transformation_vector
        # add ones
        ones = np.ones((len(X), 1), dtype=np.float64)
        X = np.append(ones, X, axis=1)
        return list(map(lambda x: self.predict_single(x), X))

    def get_integrated_error(self, X, y):

        data_len = len(y)
        get_integrated_error_per_data_point = np.vectorize(
            lambda idx: y[idx]*X[idx]*(1 - self.get_probability(X[idx], y[idx])),
            signature='()->(m)')
        integrated_errors = get_integrated_error_per_data_point(range(data_len))

        return integrated_errors.sum(0)

    def get_probability(self, x, y):
        try:
            return 1 / (1 + math.exp((-y * self.beta).T.dot(x)))
        except:
            return 0

    def predict_single(self, x):
        return 1 if self.get_probability(x, 1) > 0.5 else 0

    def get_log_likelihood(self, X, y):
        weighted = X.dot(self.beta)
        return np.sum(y * weighted - np.log(1 + np.exp(weighted)))
```

LogisticRegressionDemo.py

```
from sklearn.linear_model import LogisticRegression as LRSKL
import numpy as np
from Parser import get_data_set
from LogisticRegression import LogisticRegression

X_train, X_test, y_train, y_test = get_data_set(1)
lr = LogisticRegression(X_train, y_train, plot=True)
score = lr.score(X_test, y_test)
print('Score: {}'.format(score))
print(lr.confusion_matrix(X_test, y_test))

sklearn_lr = LRSKL()
sklearn_lr.fit(X_train, y_train)
predictions = sklearn_lr.predict(X_test)
```

```
score_sklearn = np.mean(predictions == y_test)
print('Score from sklearn: {}'.format(score_sklearn))
```

bool_func_analysis.py

```
import os
import numpy as np
import pandas as pd
from LogisticRegression import LogisticRegression
from Parser import extract_classes_from_data_set


file_name = os.path.join(os.path.dirname(__file__), './Dataset/boolfunc.data')
data = np.array(pd.read_csv(file_name, header=None).as_matrix())
X = np.array(data[:,:-1], dtype=np.float64)
y = data[:,-1]

and_or_X, and_or_y = extract_classes_from_data_set(X, y, ['and', 'or'])
and_or_y = np.array([1 if x == 'and' else 0 for x in and_or_y], dtype=np.float64)
lr_and_or = LogisticRegression(and_or_X, and_or_y, iterations=1000)
print('beta and vs or: {}'.format(lr_and_or.beta))
print('transformed X and vs or: {}'.format(lr_and_or.transform(and_or_X, and_or_y)[0]))
print('Predicted: {}; Expected: {}'.format(lr_and_or.predict(and_or_X), and_or_y))

and_xor_X, and_xor_y = extract_classes_from_data_set(X, y, ['and', 'xor'])
and_xor_y = np.array([1 if x == 'and' else 0 for x in and_xor_y], dtype=np.float64)
lr_and_xor = LogisticRegression(and_xor_X, and_xor_y, iterations=1000)
print('beta and vs xor: {}'.format(lr_and_xor.beta))
print('transformed X and vs xor: {}'.format(lr_and_xor.transform(and_xor_X, and_xor_y)[0]))
print('Predicted: {}; Expected: {}'.format(lr_and_xor.predict(and_xor_X), and_xor_y))

or_xor_X, or_xor_y = extract_classes_from_data_set(X, y, ['or', 'xor'])
or_xor_y = np.array([1 if x == 'or' else 0 for x in or_xor_y], dtype=np.float64)
lr_or_xor = LogisticRegression(or_xor_X, or_xor_y, iterations=1000)
print('beta or vs xor: {}'.format(lr_or_xor.beta))
print('transformed X or vs xor: {}'.format(lr_or_xor.transform(or_xor_X, or_xor_y)[0]))
print('Predicted: {}; Expected: {}'.format(lr_or_xor.predict(or_xor_X), or_xor_y))
```

Parser.py

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


def parse_data(name):
    file_name = os.path.join(os.path.dirname(__file__), './Dataset/{}.data'.format(name))
    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 extract_classes_from_data_set(X, y, classes):
    is_from_classes = np.vectorize(lambda y: y in classes)
    filter_arr = is_from_classes(y)
    return X[filter_arr], y[filter_arr]


def get_data_set(seed):
    data = parse_data('spambase')
    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
```