Statistical Machine Learning 2018
Assignment 1 - Codelisting
Deadline: 7th of October 2018

Christoph Schmidl
s4226887
c.schmidl@student.ru.nl

Mark Beijer
s4354834
mbeijer@science.ru.nl

October 7, 2018

Exercise 1 - weight 5

1.1

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
# Return evenly spaced numbers over a specified interval.

# Exercise 1.1

def f(x):
  return 1 + np.sin(6*(x-2))

def noisy_function(func, func_argument):
  noise = np.random.normal(0, 0.3)
  return noise + func(func_argument)

def generate_data(amount_of_datapoints):
    return [noisy_function(f, x) for x in np.linspace(0, 1, amount_of_datapoints)]

def plot_data(dataset):
    # plot the dataset
    plt.scatter(np.linspace(0, 1, 10), dataset, label='noisy observations')
    # plot the actual function
    X = np.linspace(0, 1, 100) # the higher the num value, the smoother the function plot
    gets
    y = [f(x) for x in X]
    plt.plot(X, y, color='green', label='True function')
    # plt.xlim(xmin=0)
    plt.ylabel('y')
    plt.xlabel('x')
    plt.legend(loc='upper right')
    # fancy caption. Not needed if latex is doing the job later on
    #txt="I need the caption to be present a little below X-axis"
    #plt.figtext(0.5, -0.05, txt, wrap=True, horizontalalignment='center', fontsize=12)
    plt.savefig('exercise_1_1.png')
    plt.show()

# Generating data set D_10 of 10 noisy observations
training_set = generate_data(10)

# Generating data set T of 100 noisy observations
test_set = generate_data(100)

# Plotting the function and observations in D_10
plot_data(training_set)
```

1.2

```python
# Exercise 1.2

def SSE(observations, targets):
    """ Calculate the sum-of-squares error. """
    return 0.5 * np.sum((observations - targets)**2)


def pol_cur_fit(data, polynomial_order):
    """ Return weights for an optimal polynomial curve fit. """

    polynomial_order = polynomial_order + 1

    observations = data[0, :] # Get me the first row, D_N
    targets = data[1, :] # Get me the second row, M

    # observation matrix
    A = np.zeros((polynomial_order, polynomial_order)) # Create matrix
    for i in range(polynomial_order):
        for j in range(polynomial_order):
            A[i, j] = np.sum(observations ** (i+j))

    # target vector
    B = np.zeros(polynomial_order)
    for i in range(polynomial_order):
        B[i] = np.sum(targets * observations**i)

    # numpy.linalg.solve(a, b)
    # Solve a linear matrix equation, or system of linear scalar equations.
    # Computes the  exact  solution, x, of the well-determined, i.e., full rank, linear
    matrix equation ax = b.

    # Here's where the magic happens. Solve the linear system.
    weights = np.linalg.solve(A, B)
    return weights
```

1.3

```python
# Exercise 1.3

def evaluate_polynomial(point, weights):
    """ Evaluate a polynomial. """
    return np.polyval(list(reversed(weights)), point)

def RMSE(observations, targets):
    """ Calculate the root-mean-squared error. """
    error = SSE(observations, targets)
    return np.sqrt(2 * error / len(observations))


def evaluate_and_plot_curve_fitting(training_set, test_set, min_polynomial_order = 0,
    max_polynomial_order = 10):
    """Evaluate the RMSE based on different polynomial orders"""

    errors_train = []
    errors_test = []

    X = np.linspace(0, 1, 100)
    y = [f(x) for x in X]

    for m in range(min_polynomial_order, max_polynomial_order):
        w = pol_cur_fit(training_data, m)
        fitted_curve = evaluate_polynomial(X, w)

        print("Evaluated polynomial: {}".format(fitted_curve))

        rmse_train = RMSE(evaluate_polynomial(training_set[0, :], w), training_set[1, :])
        rmse_test = RMSE(evaluate_polynomial(test_set[0, :], w), test_set[1, :])
        errors_train.append(rmse_train)
        errors_test.append(rmse_test)

        plt.figure()
        plt.plot(X, fitted_curve, 'b', label='Fitted Curve')
        plt.plot(X, y, 'r', label='True function')
        plt.scatter(training_set[0, :], training_set[1, :], c='g', label='Noisy observations
    ')
        plt.title('M=%d: train RMSE=%.2f, test RMSE=%.2f' % (m, rmse_train, rmse_test))
```

```
        plt.xlabel("x")
        plt.ylabel("y")
        plt.legend()
        plt.savefig('curvefit_m%d_n_%d.png' % (m, training_set.shape[1]))
        plt.show()

    plt.figure()
    x_range = np.arange(min_polynomial_order, max_polynomial_order)

    plt.plot(x_range, errors_test, 'r', label='Test')
    plt.plot(x_range, errors_train, 'b', label='Training')
    #plt.ylim([0, 0.4])
    plt.legend()
    plt.xlabel("Polynomial order")
    plt.ylabel("RMSE")
    plt.savefig("rmse_polynomial_order.png")
    plt.show()

    print(errors_train)
    print(errors_test)


# Add the linear spacing of y values to training and test set
full_training_set = np.vstack((np.linspace(0, 1, 10), training_set))
full_test_set = np.vstack((np.linspace(0, 1, 100), test_set))

evaluate_and_plot_curve_fitting(full_training_set, full_test_set)
```

1.4

```
# Generating data set D_40 of 40 noisy observations
training_set_40 = generate_data(40)
full_training_set_40 = np.vstack((np.linspace(0, 1, 40), training_set_40))

evaluate_and_plot_curve_fitting(full_training_set_40, full_test_set)
```

1.5

```
def pol_cur_fit_with_regularization(data, polynomial_order, regularizer = 0.00):
    """ Return weights for an optimal polynomial curve fit. """

    observations = data[0, :] # Get me the first row, D_N
    targets = data[1, :] # Get me the second row, M

    # observation matrix
    A = np.zeros((polynomial_order, polynomial_order)) # Create matrix
    for i in range(polynomial_order):
        for j in range(polynomial_order):
            A[i, j] = np.sum(observations ** (i+j))

    # Regularization
    # Multiply the diagonal matrix of order m with regularization term and add it to A
    A = A + ( regularizer * np.identity(polynomial_order) )

    # target vector
    B = np.zeros(polynomial_order)
    for i in range(polynomial_order):
        B[i] = np.sum(targets * observations**i)

    # numpy.linalg.solve(a, b)
    # Solve a linear matrix equation, or system of linear scalar equations.
    # Computes the exact solution, x, of the well-determined, i.e., full rank, linear matrix equation ax = b.

    # Here's where the magic happens. Solve the linear system.
    weights = np.linalg.solve(A, B)
    return weights


def evaluate_and_plot_curve_fitting_with_reg(training_set,
                                             test_set,
                                             reg = 0.1):
    """Evaluate the RMSE based on different polynomial orders"""

    errors_train = []
    errors_test = []
```

```

    regularizer_range = np.arange(-40, -20)
    exp_regularizer_range = np.exp(regularizer_range) # perform e^x because Bishop uses ln lambda

    for regularizer_value in exp_regularizer_range:
        w = pol_cur_fit_with_regularization(training_set, 9, regularizer_value) # fix polynomial order to 9
        rmse_train = RMSE(evaluate_polynomial(training_set[0, :], w), training_data[1, :])
        rmse_test = RMSE(evaluate_polynomial(test_set[0, :], w), test_set[1, :])
        errors_train.append(rmse_train)
        errors_test.append(rmse_test)

    plt.figure()

    plt.plot(regularizer_range, errors_test, 'r', label='Test')
    plt.plot(regularizer_range, errors_train, 'b', label='Training')
    #plt.ylim([0, 0.4])
    plt.legend()
    plt.xlabel(r'$\ln {\lambda}$')
    plt.ylabel("RMSE")
    plt.savefig("rmse_polynomial_order_reg.png")
    plt.show()


weights = w = pol_cur_fit(training_data, 9)
weights_regularized = pol_cur_fit_with_regularization(training_data, 9, 0.1)

print(weights)
print(weights_regularized)

evaluate_and_plot_curve_fitting_with_reg(full_training_set, full_test_set)
```

Exercise 2 - weight 2.5

```
#%%Imorts
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
from matplotlib import cm
import random as rand

#%%Functions

def H(x,y):
    return 100*(y-x**2)**2+(1-x)**2

def NabH(x,y): #Gives the nabla for a given point (x,y).
    return np.array([400*x**3-400*x*y+2*x-2,200*y-200*x**2])

def NabHvec(X):
    return NabH(X[0],X[1])

def Distance(X,Y): #Calculates the distance between two (two-dimensional) vectors.
    return np.sqrt((X[0]-Y[0])**2+(X[1]-Y[1])**2)

def TooFar(X): #If the distance from the origin is larger then 10^10 this functions returns true. This because if we wander that far from the origin the next steps will lead us to values to high to calculate.
    return Distance(X,[0,0]) > 10**10

def Test(eta,StepTimes,TestTimes,xmin,xmax,ymin,ymax,MinDistance,Best=[1,1]):
    Result = 0
    for i in range(TestTimes):
        RandBegin = np.array([rand.uniform(xmin,xmax),rand.uniform(ymin,ymax)])
        for j in range(StepTimes):
            NewPoint = RandBegin - eta * NabHvec(RandBegin)
            RandBegin = NewPoint
            if(TooFar(RandBegin)):
                break
        if(Distance(RandBegin,Best) < MinDistance):
            Result += 1
    return Result

def save2D(X,FileName): #Saves a 2d array X to a file with FileName name.
    File = open(FileName,'w')
```

```python
    for i in range(len(X)):
        for j in range(len(X[0])):
            File.write(str(X[i][j]))
            if(j < (len(X[0])-1)):
                File.write('\t')
        if(i < (len(X)-1)):
            File.write('\n')
    File.close()

def save1D(X,FileName):#Saves a 1d array X to a file with FileName name.
    File = open(FileName,'w')
    for i in range(len(X)):
        File.write(str(X[i]))
        if(i < (len(X) -1)):
            File.write('\n')
    File.close()
#%%Constants

ReCalculate = False

Amount = 1000
xmin = -2
xmax = 2
ymin = -1
ymax = 3

Eta = 2*10**-3
Punt = np.array([-1,2])

LogEtasMin = -7
LogEtasMax = -2
AmountEtas = 100

StepTimesMin = 1
StepTimesMax = 300
MinDistance = 0.1
TestTimes = 100



#%% Create Data


x = np.linspace(xmin,xmax,Amount)
y = np.linspace(ymin,ymax,Amount)
x,y = np.meshgrid(x,y)
z = H(x,y)

#%%TEST


Best = np.array([1,1])

fig = plt.figure("Path over 3d surface")
ax = fig.gca(projection='3d')

plt.xlabel('x',fontsize=20,labelpad=20)
plt.ylabel('y',fontsize=20,labelpad=20)





surf = ax.plot_surface(x, y, z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=True)



fig.colorbar(surf, shrink=0.5, aspect=5)


ax.plot([1,1],[1,1],[0,2000],color='red')


for i in range(100):
    PuntNew = Punt - Eta*NabHvec(Punt)
```

```python
    X = np.linspace(Punt[0],PuntNew[0],Amount)
    Y = np.linspace(Punt[1],PuntNew[1],Amount)
    Z = H(X,Y)
    ax.plot(X, Y, Z, color='orange')
    Punt = PuntNew
plt.title('Path over 3d surface',fontsize=40)
plt.tick_params(labelsize=20)
matplotlib.rcParams.update({'font.size': 22})

#%% Num Test
"""
etas = np.linspace(1*10**-6,1*10**-2,100)
point = []
testMax = 1000


for eta in etas:
    point.append(0)
    for i in range(testMax):
        RandBegin = np.array([rand.uniform(xmin,xmax),rand.uniform(ymin,ymax)])
        for j in range(100):
            NewPoint = RandBegin - eta * NabHvec(RandBegin)
            RandBegin = NewPoint
            if(TooFar(RandBegin)):
                break
        if(Distance(RandBegin,Best) < 1):
            point[-1] += 1

plt.plot(etas,point)
"""
#%% Num Test 3d



if(ReCalculate):
    LogEtas = np.linspace(LogEtasMin,LogEtasMax,AmountEtas)
    Etas = 10**LogEtas
    StepTimes = np.linspace(StepTimesMin,StepTimesMax,StepTimesMax,dtype=int)

    Etas,StepTimes = np.meshgrid(Etas,StepTimes)
    Result = np.zeros_like(Etas)

    for i in range(len(Etas)):
        print(i)
        for j in range(len(Etas[0])):
            Result[i][j] = Test(Etas[i][j],StepTimes[i][j],TestTimes,xmin,xmax,ymin,ymax,
    MinDistance)
    save1D(LogEtas,'LogEtas.txt')
    save2D(Etas,'Etas.txt')
    save2D(StepTimes,'StepTimes.txt')
    save2D(Result,'Result.txt')
else:
    LogEtas = np.genfromtxt('LogEtas.txt')
    Etas = np.genfromtxt('Etas.txt')
    StepTimes = np.genfromtxt('StepTimes.txt')
    Result = np.genfromtxt('Result.txt')

fig2 = plt.figure("Eta vs Times vs Good")
ax2 = fig2.gca(projection='3d')
surf2 = ax2.plot_surface(LogEtas, StepTimes, Result, cmap=cm.coolwarm,
                        linewidth=0, antialiased=True)

plt.title("Amount of sucessful walks for a given $\eta$ and amount of steps.")
plt.xlabel('LogEta',labelpad=20)
plt.ylabel('Time',labelpad=20)
fig2.colorbar(surf2, shrink=0.5, aspect=5)





#%%

"""


```

```

Nab = NabHvec(RandBegin)
xDif = Nab[0]
if(xDif < 0):
    Spacex = RandBegin[0] - xmax
else:
    Spacex = RandBegin[0] - xmin
etax = Spacex/xDif
yDif = Nab[1]
if(yDif < 0):
    Spacey = RandBegin[1] - ymax
else:
    Spacey = RandBegin[1] - ymin
etay = Spacey/yDif
eta = min(etay,etax)

"""
```