fridljandd_assignment1_problem1_2

February 5, 2023

Problem 1

In a classical algorithm the instructions are specified by the programmer before run time. In a machine learning frame work, the programmer provides data the machine learning algorithm uses to train and determine most of the parameters. Based on the learned parameters, the output for new input is calculated.

Problem 2

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random

random.seed(10)
```

[ ]:
```
import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)
```

[ ]:
```
from ps1_functions import problem2_evaluate_function_on_random_noise,␣
 ↪problem2_fit_polynomial, problem3_knn_classifier
```

1 Problem 1

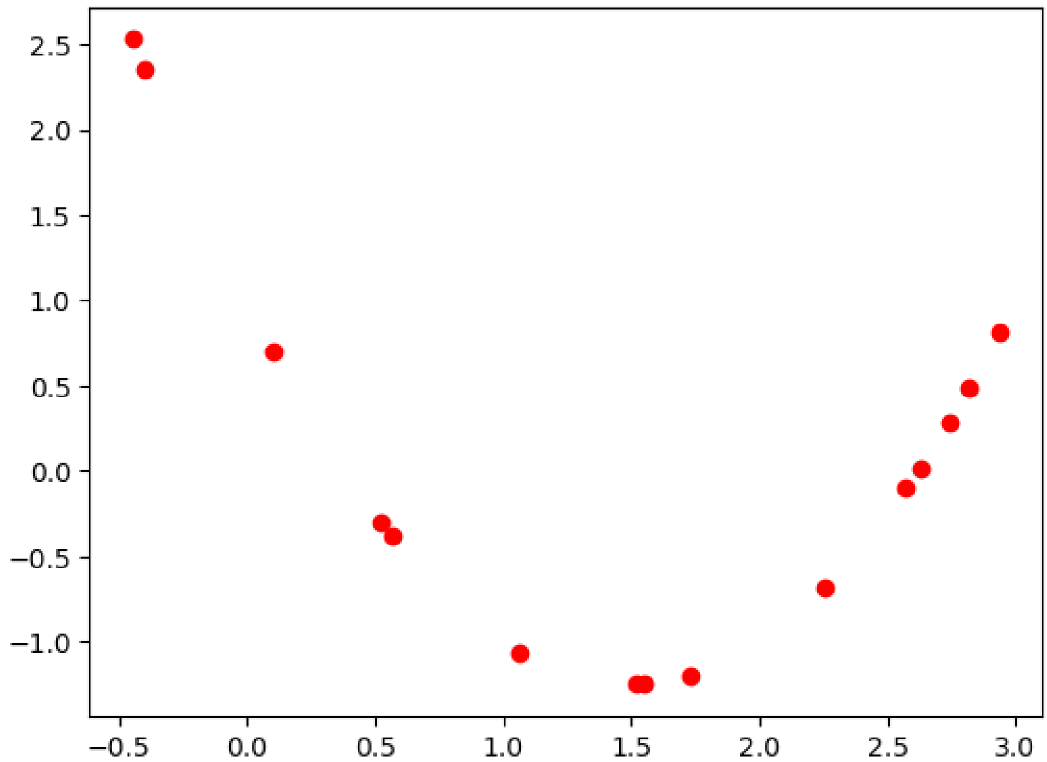
N = 15; 100 and sigma = 0; 0:05; 0:2 generate problem2_evaluate_function_on_random_noise with

1.1 Problem 1a

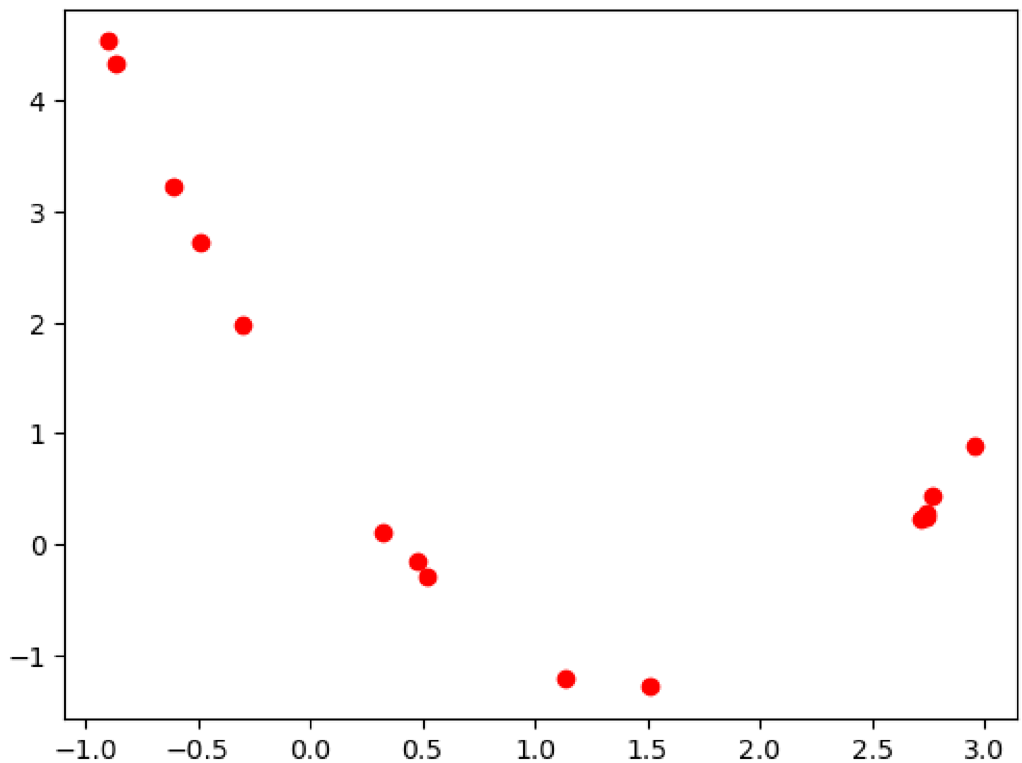
[ ]:
```
#data_15_0 = problem2_evaluate_function_on_random_noise(15, 0)
#data_15_005 = problem2_evaluate_function_on_random_noise(15, 0.05)
#data_15_02 = problem2_evaluate_function_on_random_noise(15, 0.2)
#data_100_0 = problem2_evaluate_function_on_random_noise(100, 0)
#data_100_005 = problem2_evaluate_function_on_random_noise(100, 0.05)
#data_100_02 = problem2_evaluate_function_on_random_noise(100, 0.2)
```

[ ]:
```
for n_sample in [15, 100]:
    for noise in [0, 0.05, 0.2]:
        data = problem2_evaluate_function_on_random_noise(n_sample, noise)
        print("n_sample: ", n_sample, "noise: ", noise)
        plt.plot(data[0], data[1], 'ro')
        plt.show()
```

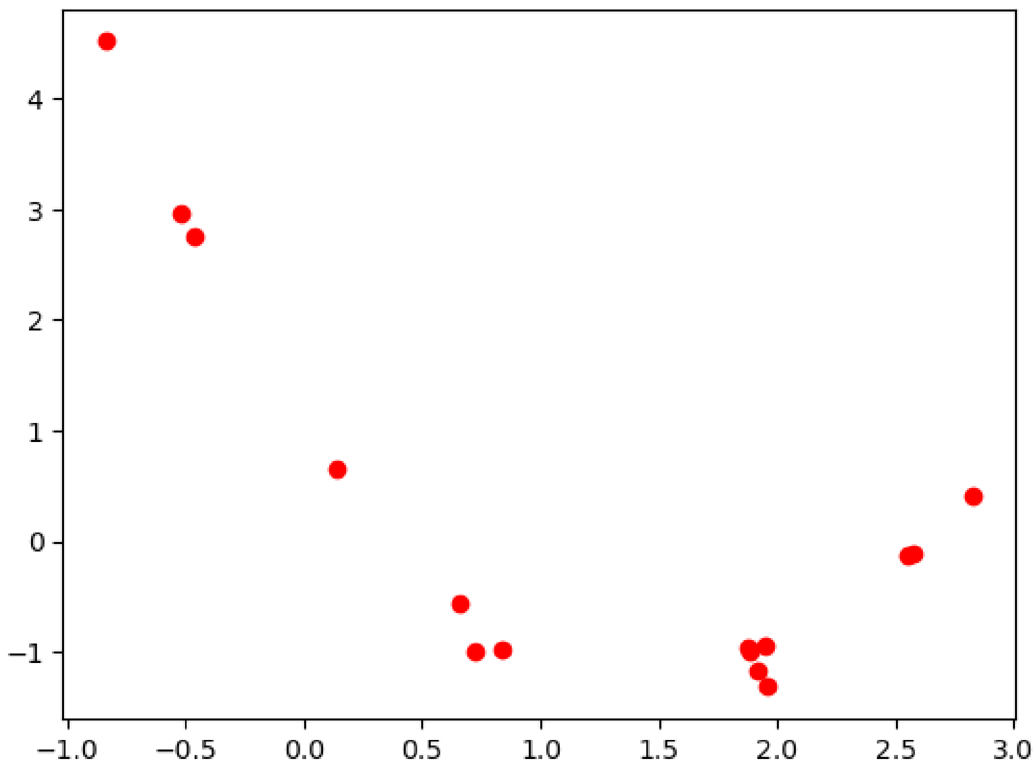
```
n_sample:  15 noise:  0
```



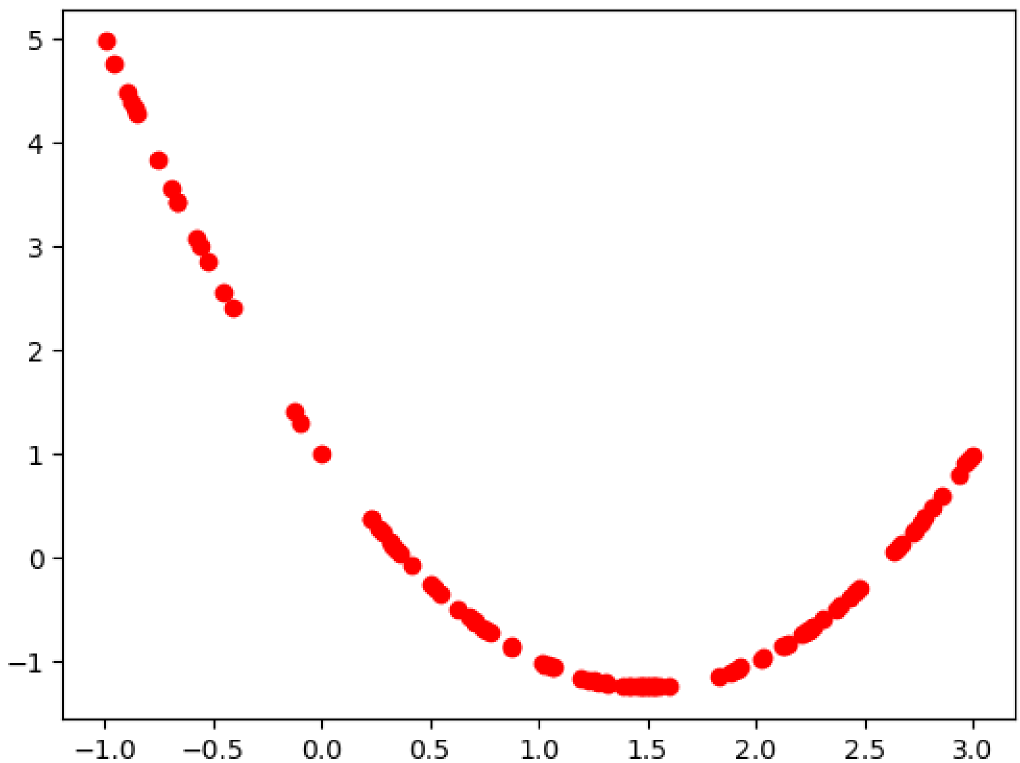

```
n_sample:  15 noise:  0.05
```



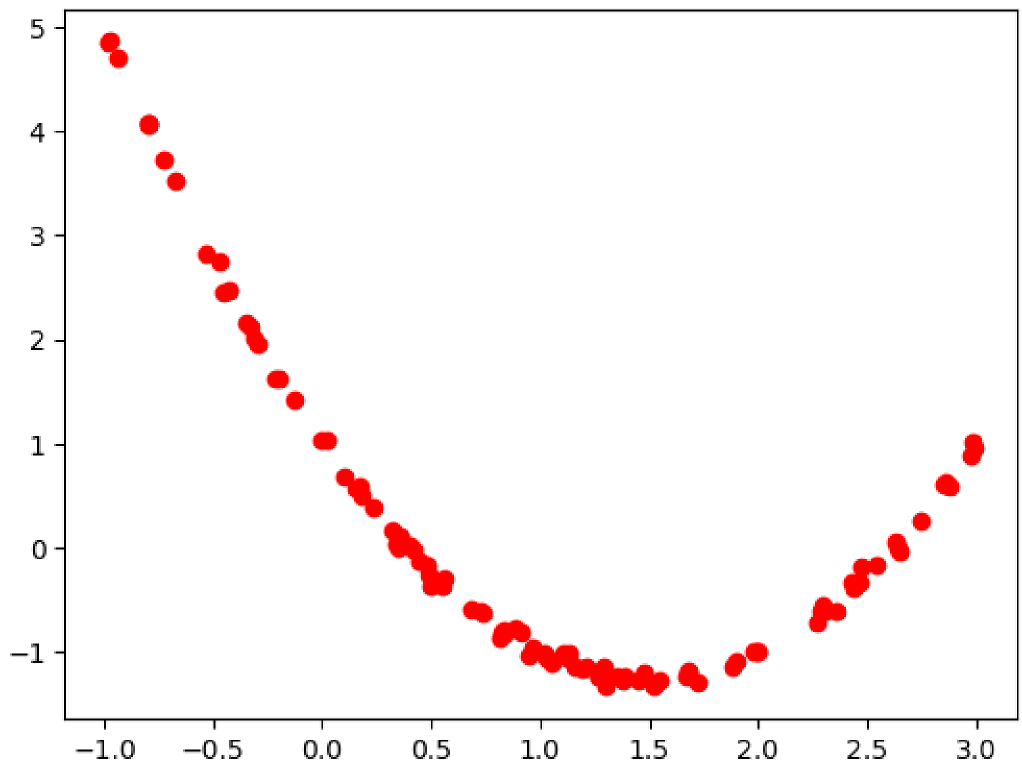

```
n_sample:  15 noise:  0.2
```



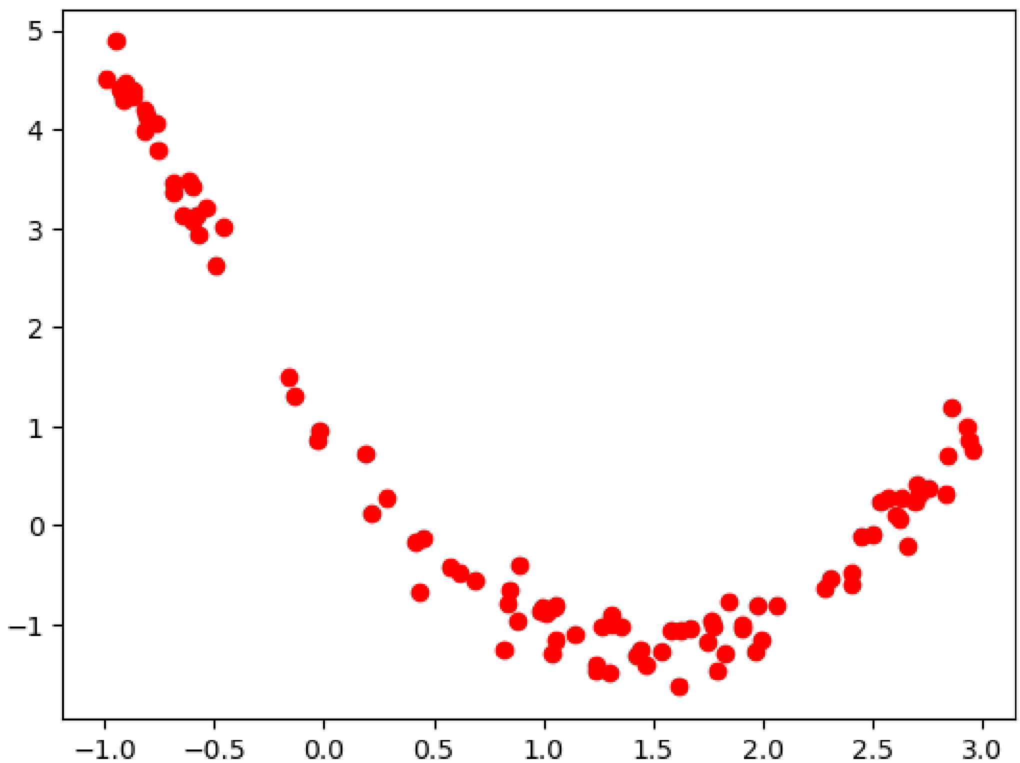

```
n_sample:  100 noise:  0
```




```
n_sample:  100 noise:  0.05
```




```
n_sample:  100 noise:  0.2
```



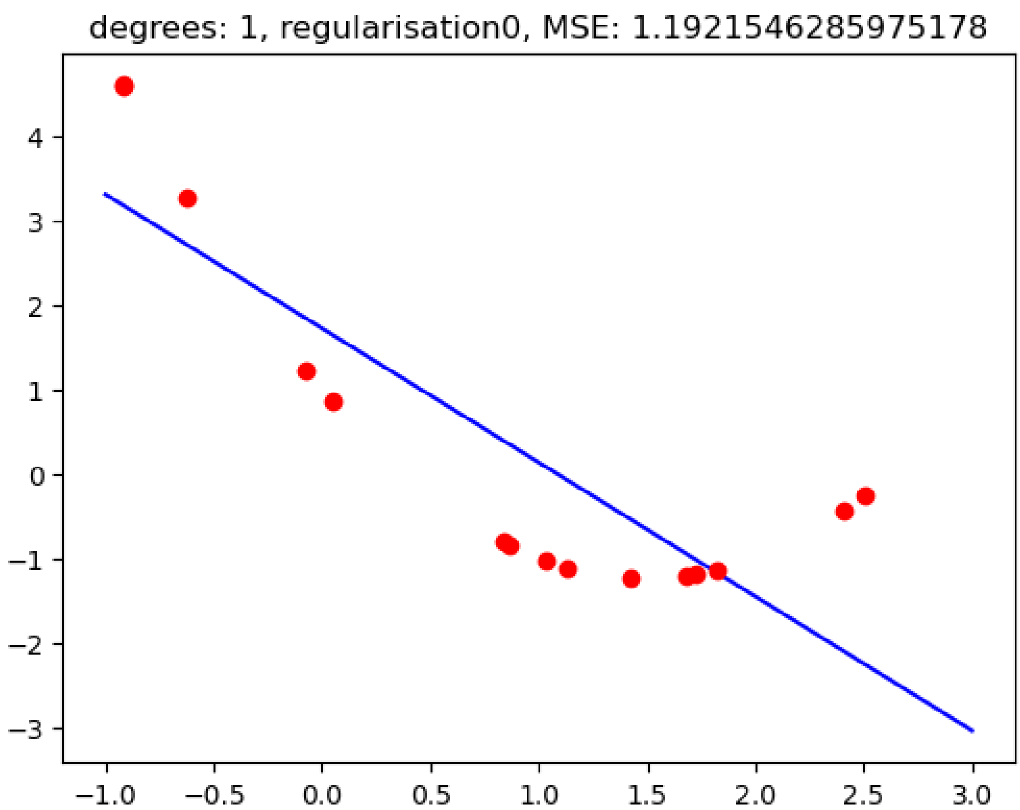
1.2 1b

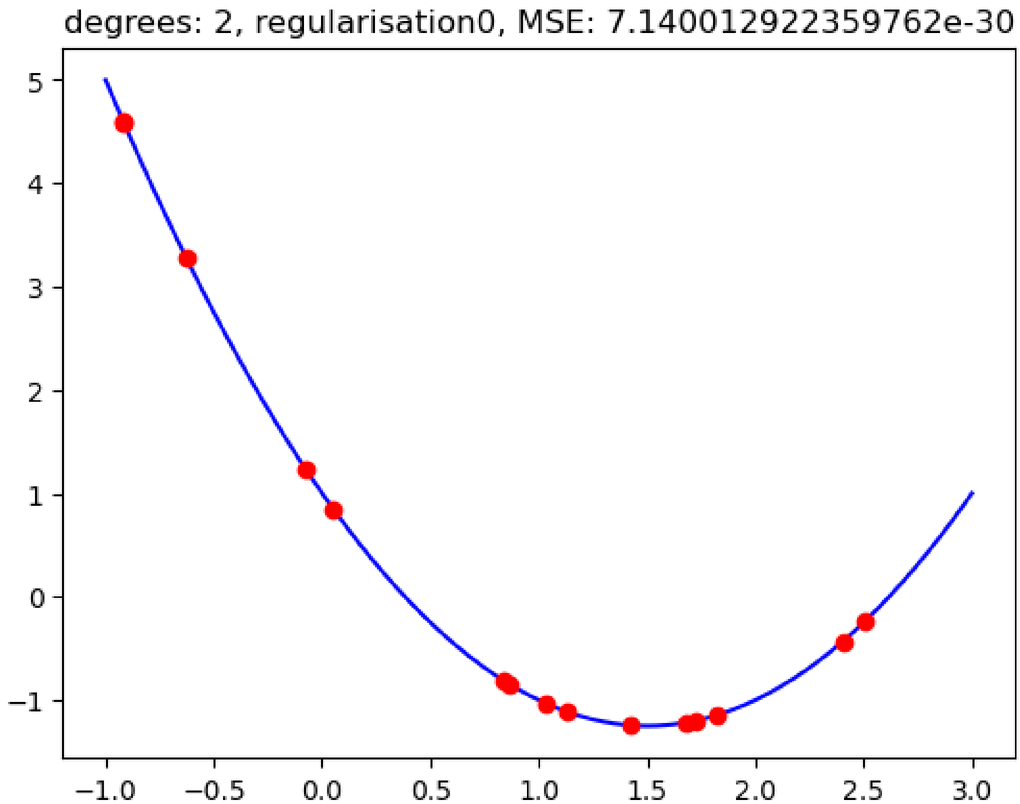
[ ]:
```
def plot_fitted_polynomial(data_x, data_y, degree, regularisation=0):

    coeffs = problem2_fit_polynomial(data_x, data_y, degree, regularisation)
    #plot polynomial with weights w on top of data
    plot_x = np.linspace(-1, 3, 100)
    plot_y = np.array([sum([w_i * x_i ** n for n, w_i in enumerate(coeffs)])␣
 ↪for x_i in plot_x])
    plt.plot(plot_x, plot_y, 'b-')

    #plot  on top
    plt.plot(data_x, data_y, 'ro')

    #plot polynomial with weights w on top of data
    predicted_y = np.array([sum([w_i * x_i ** n for n, w_i in␣
 ↪enumerate(coeffs)]) for x_i in data_x])
    #MSE between predicted_y and data_y
    mse = np.mean((predicted_y - data_y) ** 2)

    #add mse to plot
```

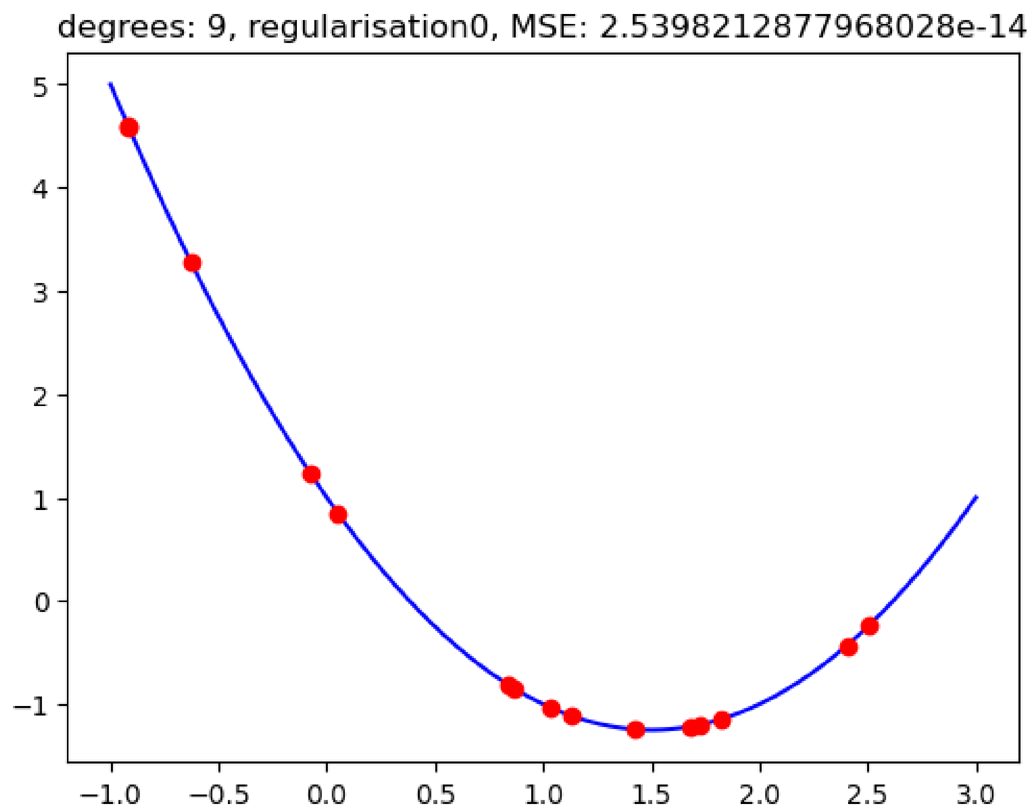
```
    plt.title("degrees: "+ str(degree)+", regularisation"+ str(regularisation)␣
↪+", MSE: " + str(mse))

    #print("MSE: ", mse)
    return mse, coeffs
```

[ ]:
```
#create empty list
list_performance = list()

for n_sample in [15, 100]:
    for noise in [0, 0.05, 0.2]:
        data = problem2_evaluate_function_on_random_noise(n_sample, noise)
        print("n_sample: ", n_sample, "noise: ", noise)
        for degree in [1,2,9]:
            mse, coeffs = plot_fitted_polynomial(data[0], data[1], degree)
            #add mse, coeffs tupel to list
            list_performance.append((n_sample, noise, degree, mse, coeffs))
            plt.show()
```

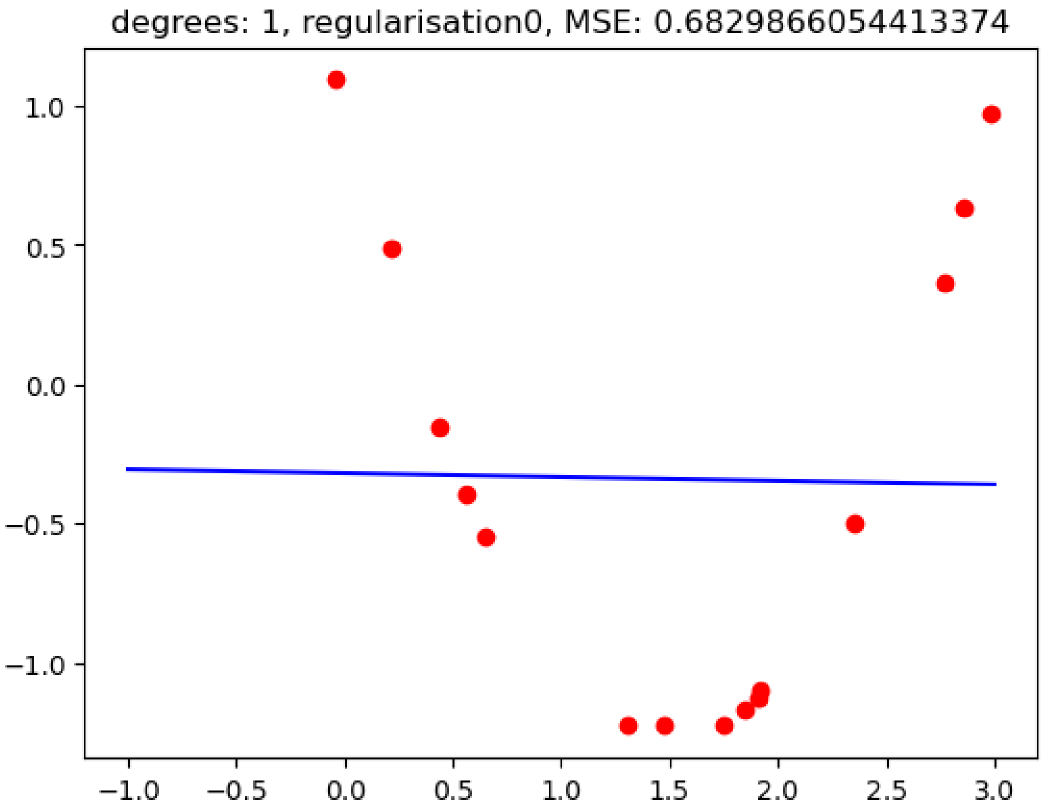
```
n_sample:  15 noise:  0
```

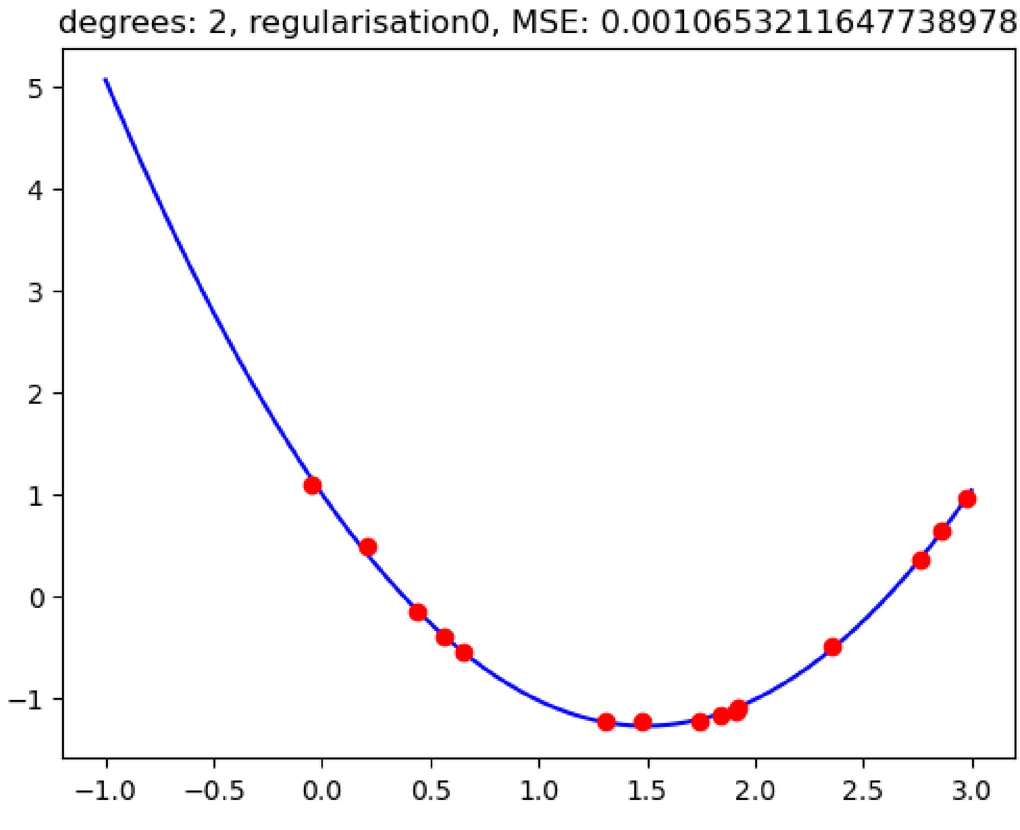


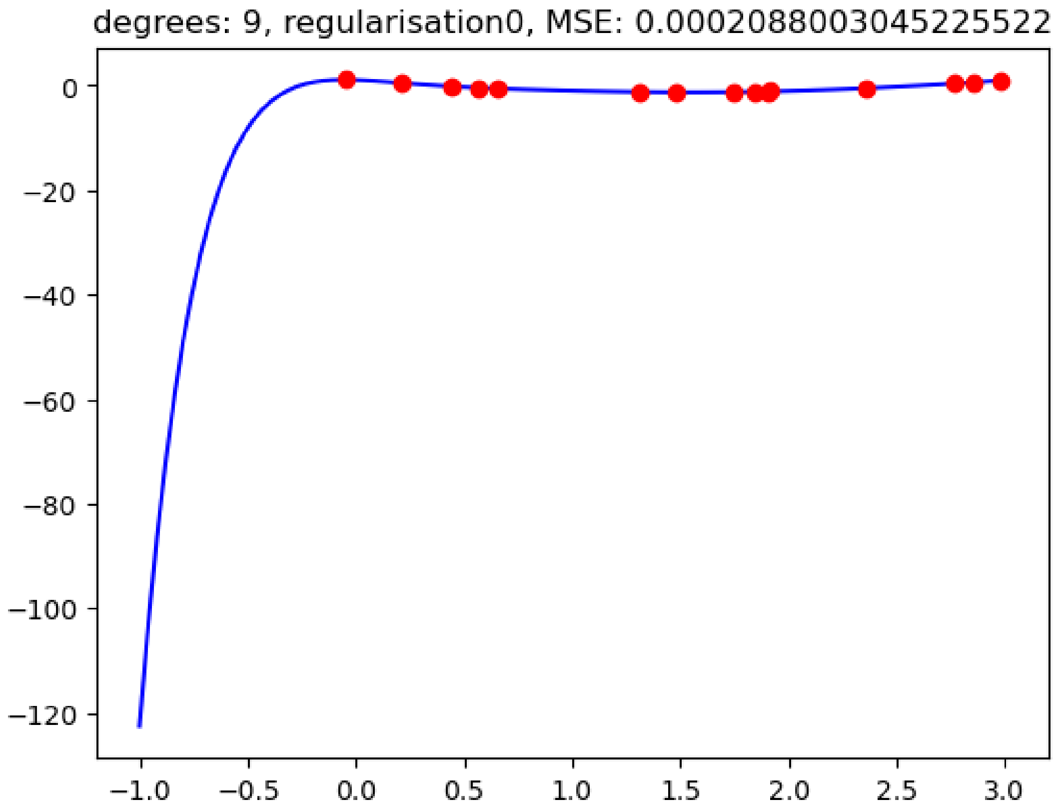

degrees: 2, regularisation0, MSE: 7.140012922359762e-30
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



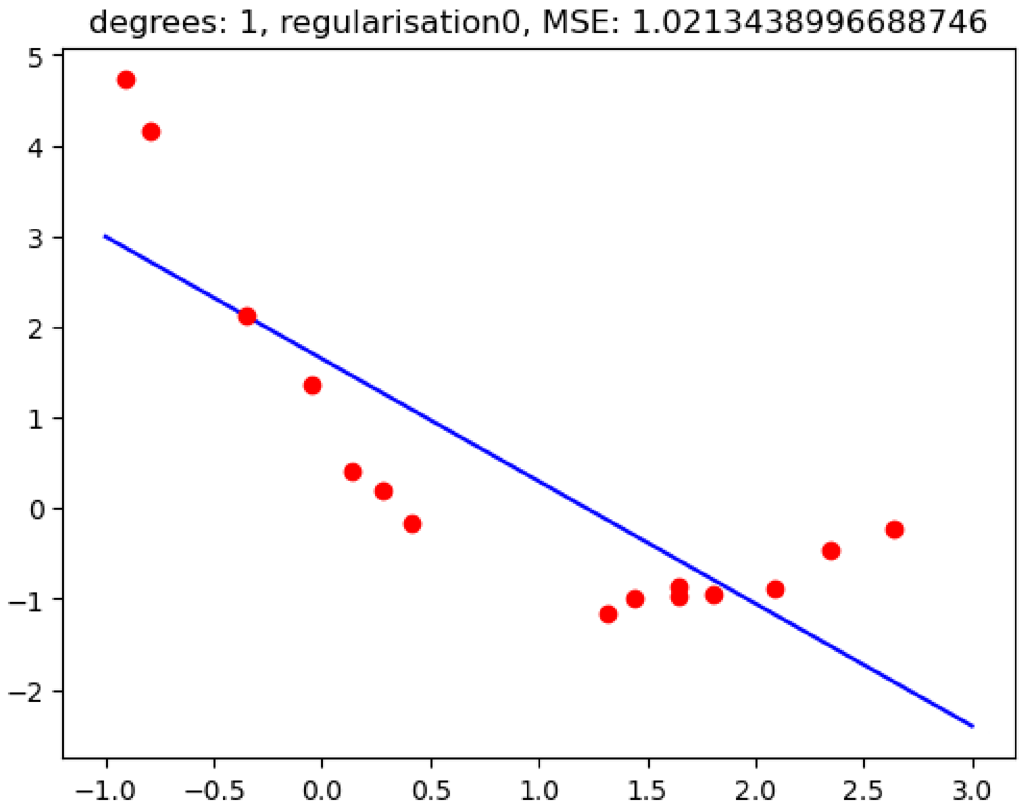

```
n_sample:  15 noise:  0.05
```

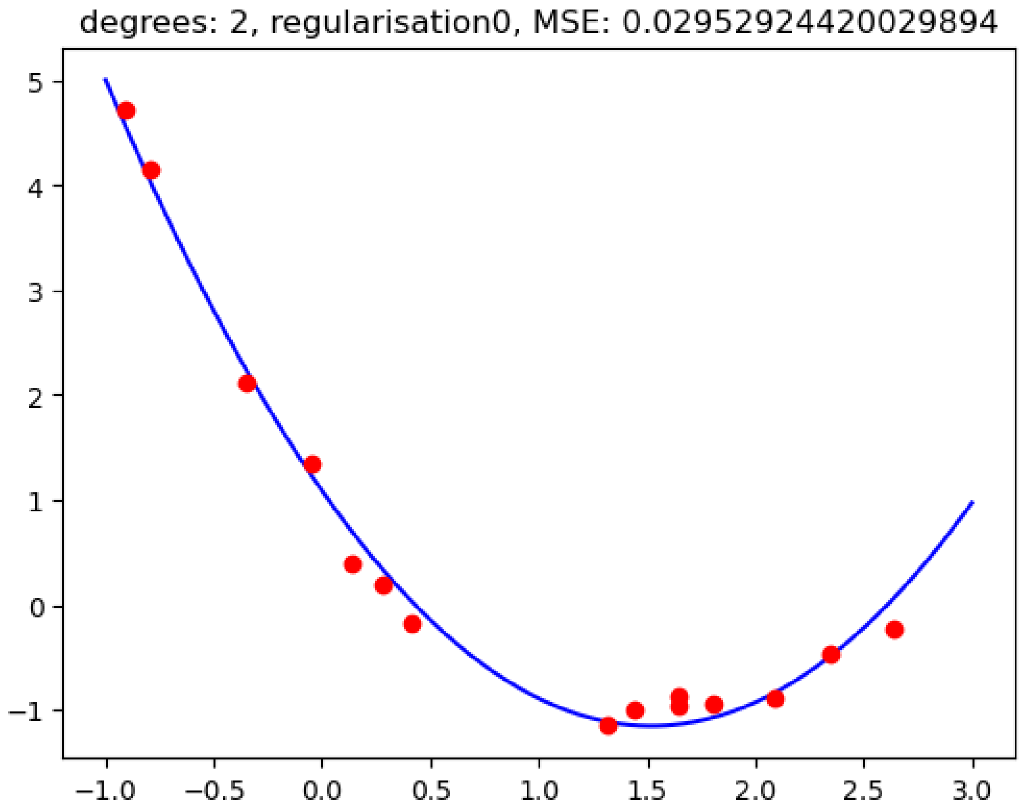

degrees: 1, regularisation0, MSE: 0.6829866054413374
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0


degrees: 2, regularisation0, MSE: 0.0010653211647738978
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

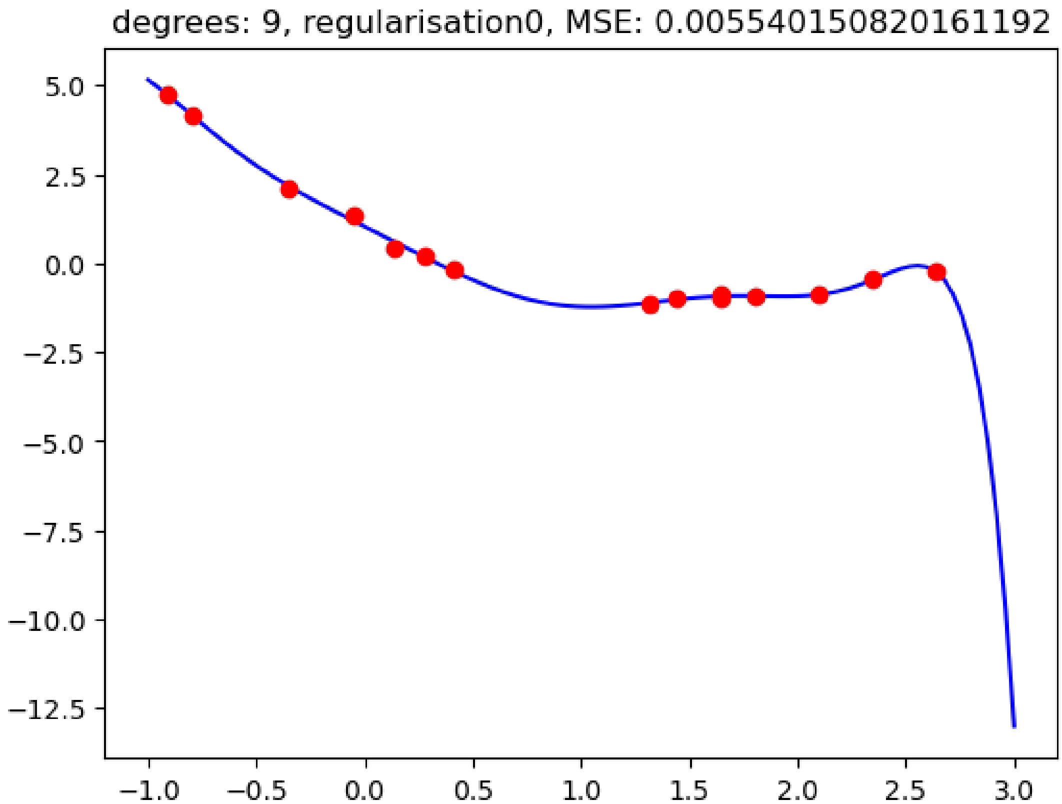


n_sample: 15 noise: 0.2

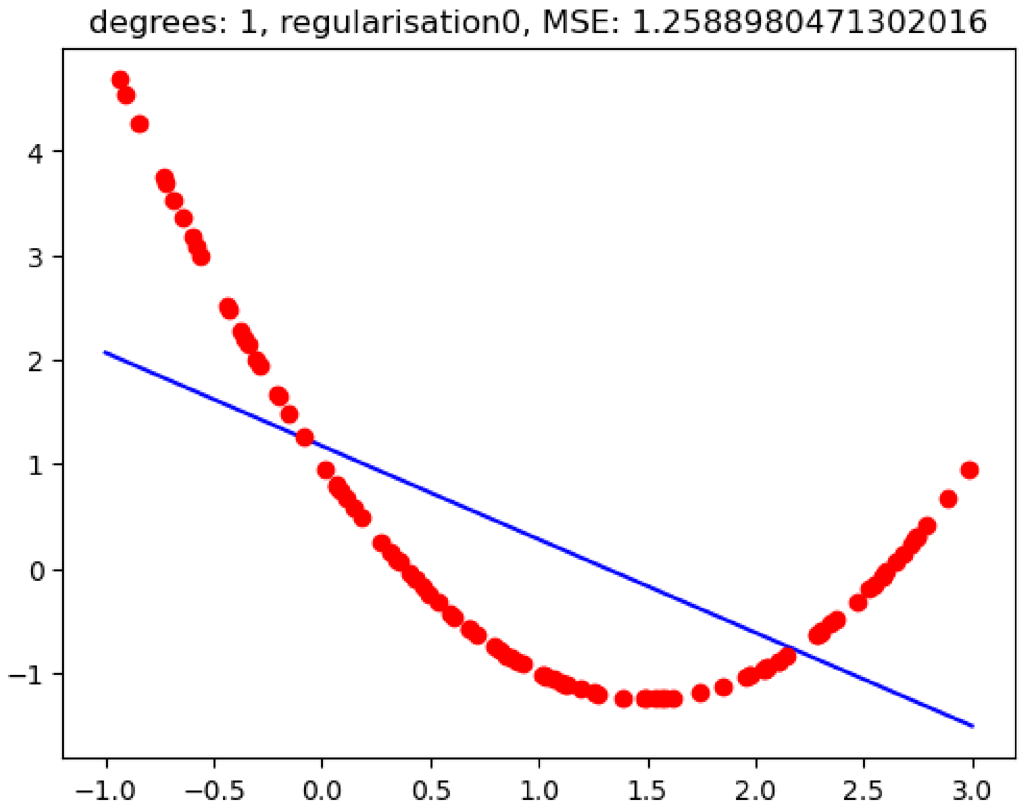

degrees: 1, regularisation0, MSE: 1.0213438996688746
5
4
3
2
1
0
−1
−2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

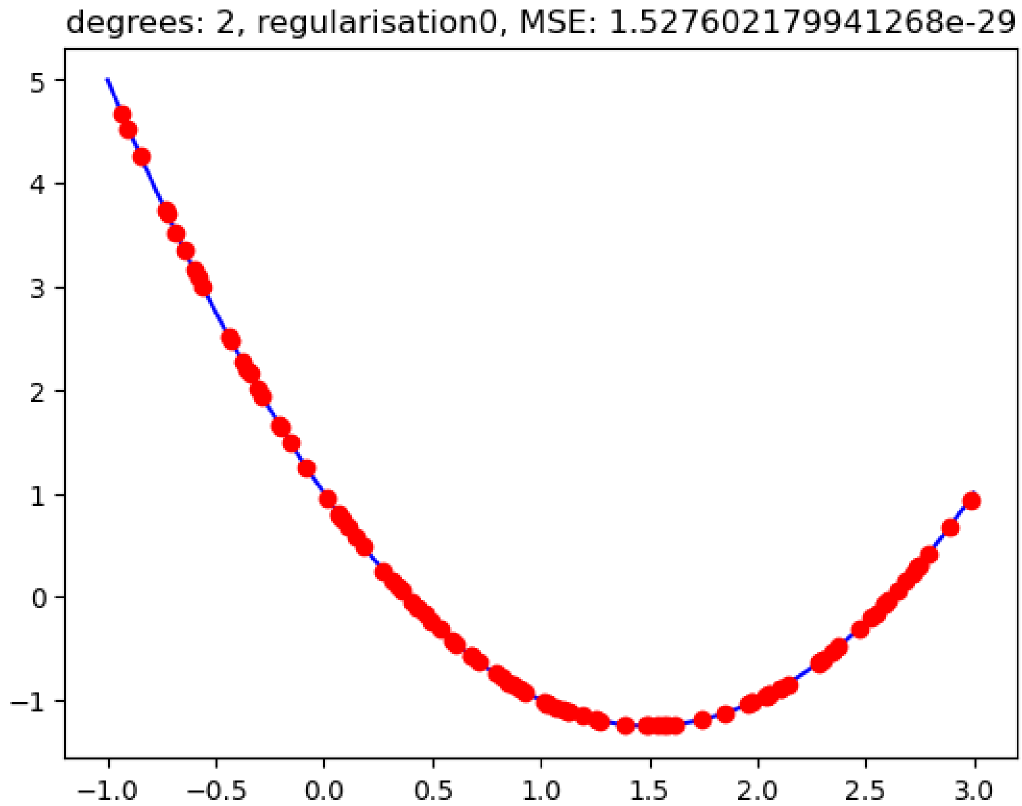


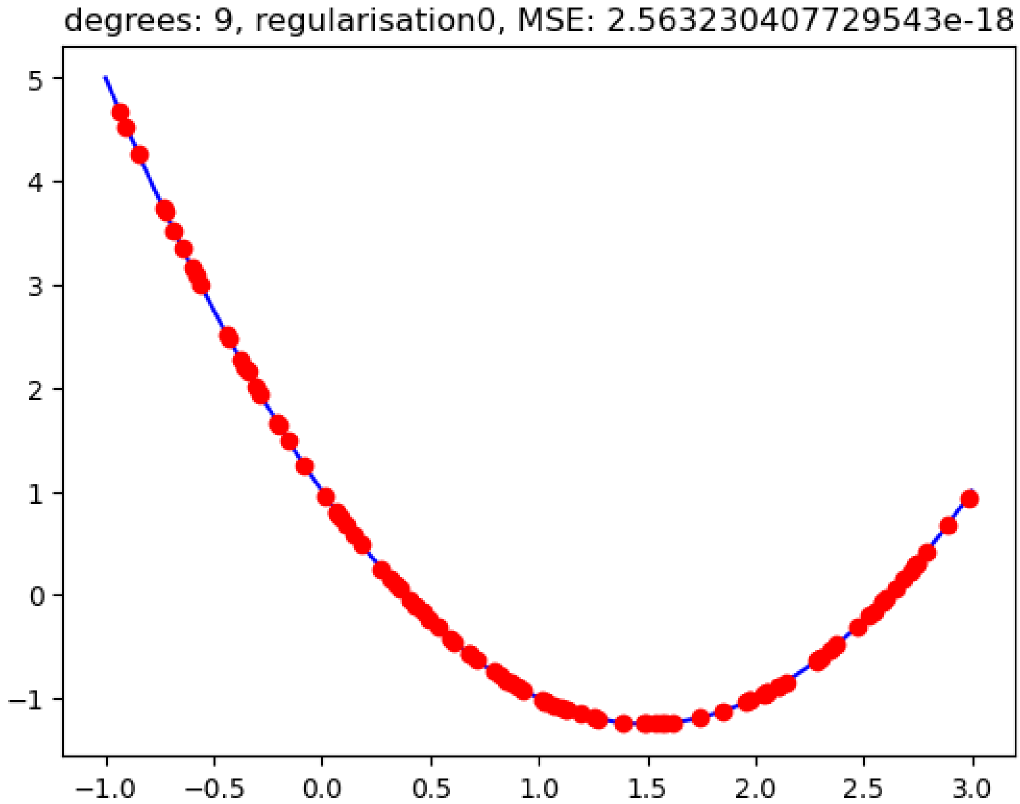
degrees: 2, regularisation0, MSE: 0.02952924420029894
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



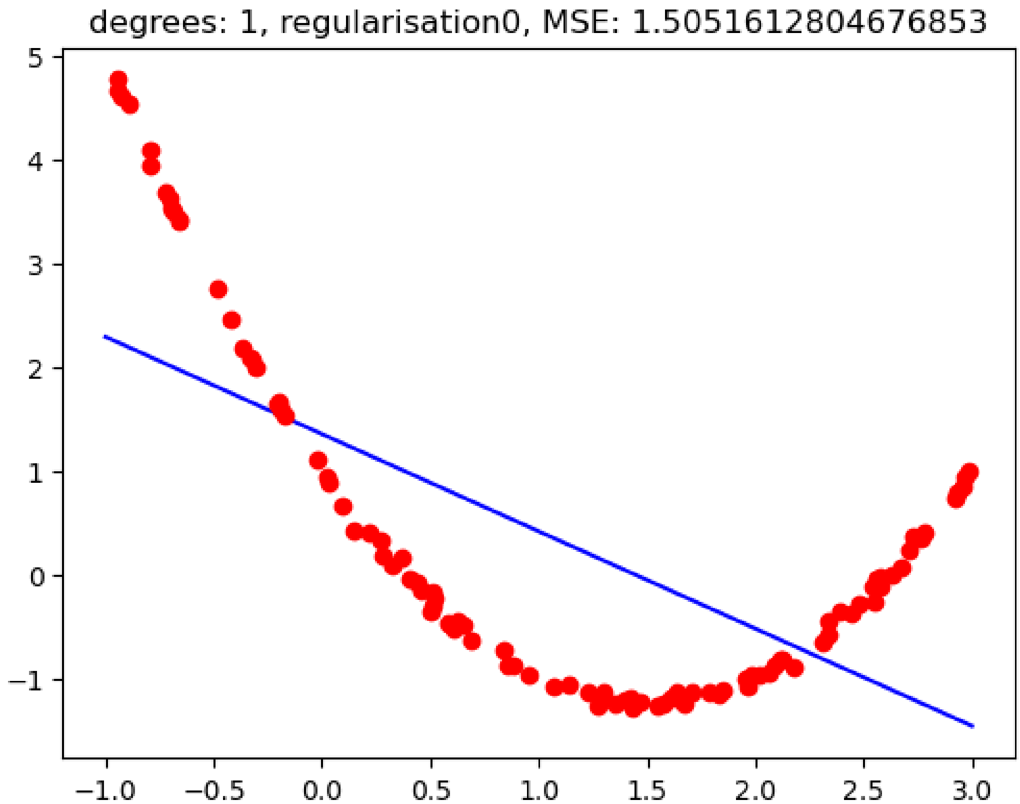

```
n_sample:  100 noise:  0
```

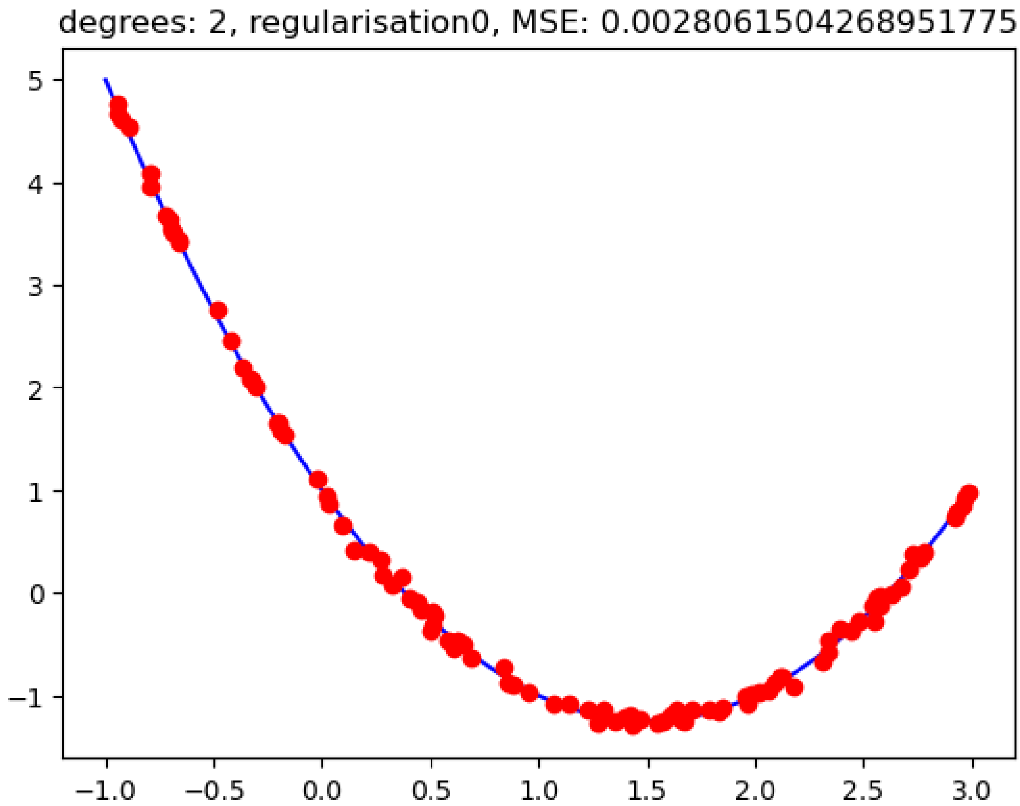

degrees: 1, regularisation0, MSE: 1.2588980471302016
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

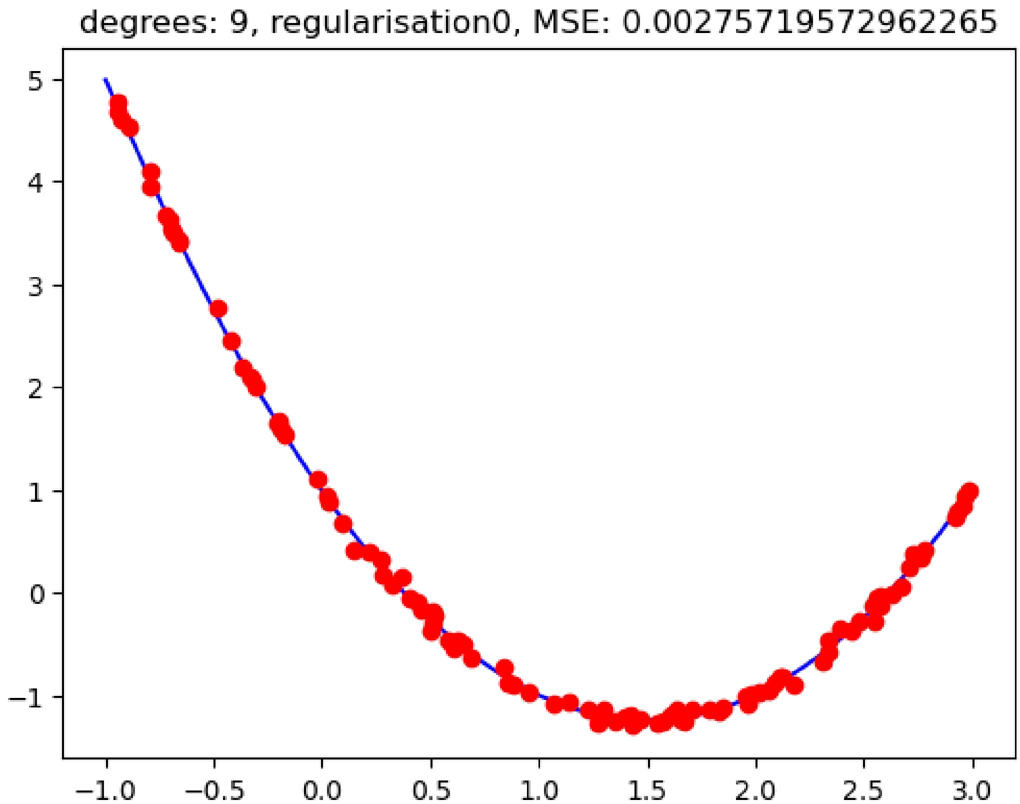

degrees: 2, regularisation0, MSE: 1.527602179941268e-29
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



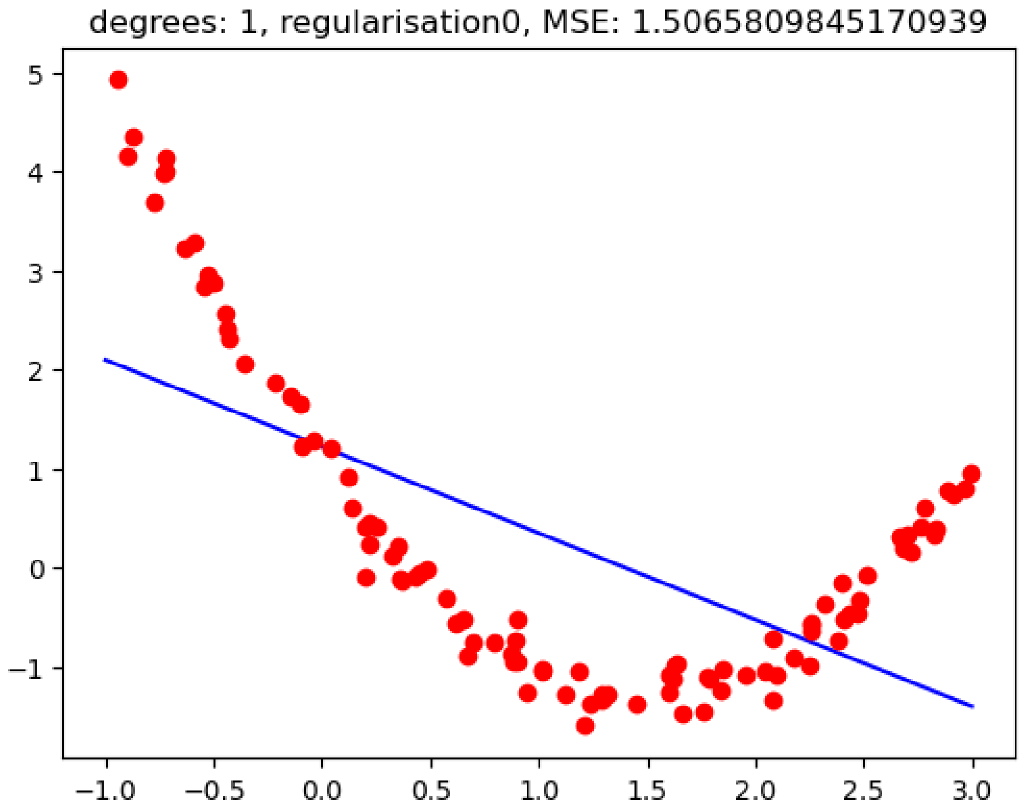

```
n_sample:  100 noise:  0.05
```

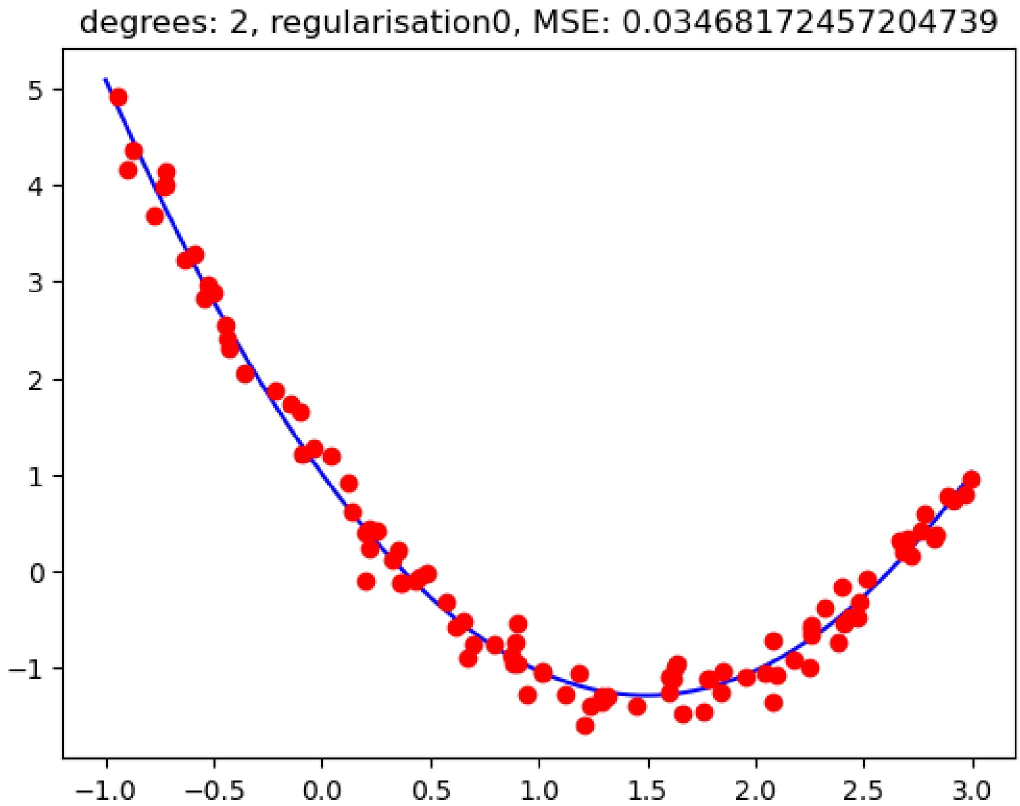

degrees: 1, regularisation0, MSE: 1.5051612804676853
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

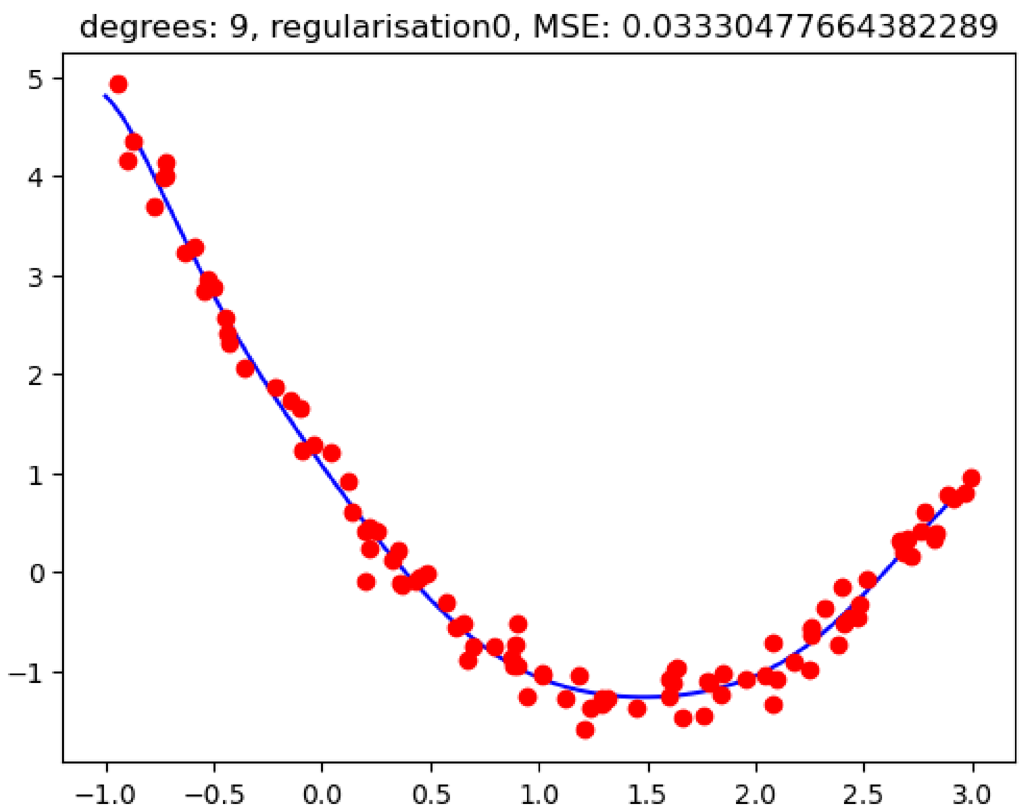

degrees: 2, regularisation0, MSE: 0.0028061504268951775
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0




```
n_sample:  100 noise:  0.2
```


degrees: 1, regularisation0, MSE: 1.5065809845170939
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0


degrees: 2, regularisation0, MSE: 0.03468172457204739
5
4
3
2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0



Qualitatively assess: degree 1 always underfits. best degree is 2, degree 9 overfits. This makes sense, because the actual underlying model has degree 2. The degree 9 polynomial is too flexible and will overfit the data.

[ ]:
```
result = pd.DataFrame(list_performance, columns=["n_sample", "noise", "degree",␣
↪"mse", "coeffs"])
#sort by mse
#result.sort_values(by="mse", inplace=True)
result
```

[ ]:

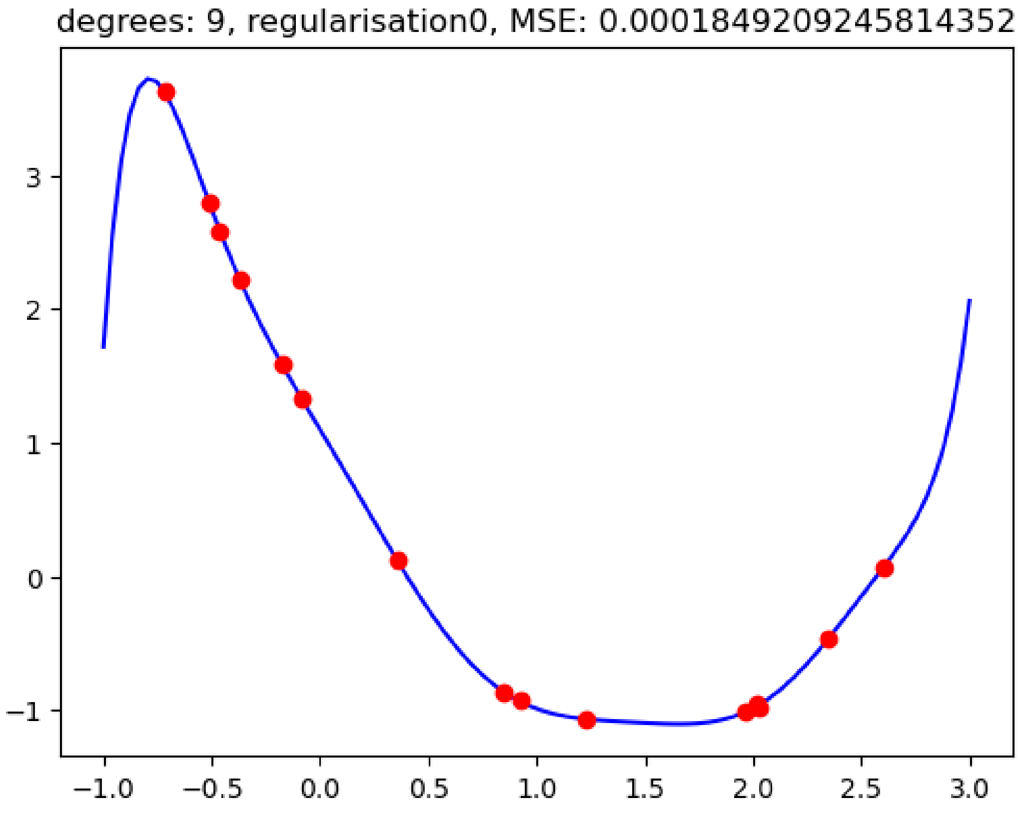
	n_sample	noise	degree	mse	\
0	15	0.00	1	1.192155e+00	
1	15	0.00	2	7.140013e-30	
2	15	0.00	9	2.539821e-14	
3	15	0.05	1	6.829866e-01	
4	15	0.05	2	1.065321e-03	
5	15	0.05	9	2.088003e-04	
6	15	0.20	1	1.021344e+00	
7	15	0.20	2	2.952924e-02	
8	15	0.20	9	5.540151e-03	
9	100	0.00	1	1.258898e+00	

```
10        100   0.00        2  1.527602e-29
11        100   0.00        9  2.563230e-18
12        100   0.05        1  1.505161e+00
13        100   0.05        2  2.806150e-03
14        100   0.05        9  2.757196e-03
15        100   0.20        1  1.506581e+00
16        100   0.20        2  3.468172e-02
17        100   0.20        9  3.330478e-02

                                                  coeffs
0           [1.7190668313955149, -1.5864713282492393]
1    [1.0000000000000002, -2.9999999999999956, 0.99…
2    [1.0000001615100165, -3.000001627484405, 1.00…
3        [-0.31879452536163566, -0.013456372197002936]
4    [1.0048075895904534, -3.0430776180856225, 1.01…
5    [1.0519282625294812, -1.4505518092867469, -10…
6              [1.647219150899124, -1.348992055523905]
7    [1.0872197162032258, -2.949028704636852, 0.970…
8    [1.0312260764802694, -3.0967695029469073, 0.02…
9           [1.1728566747216282, -0.8934739427149433]
10   [0.9999999999999981, -2.9999999999999925, 0.99…
11   [0.9999999990252593, -3.000000002053155, 1.000…
12          [1.3513554813542958, -0.9376435563290872]
13   [0.9980268313429574, -2.994828513173294, 0.999…
14   [0.9857152317254674, -2.9988329978256907, 1.04…
15           [1.220854504145304, -0.8748387604551577]
16   [1.0052326305805872, -3.059301661229381, 1.021…
17   [1.0696966974137632, -3.0370809010022275, 0.53…
```

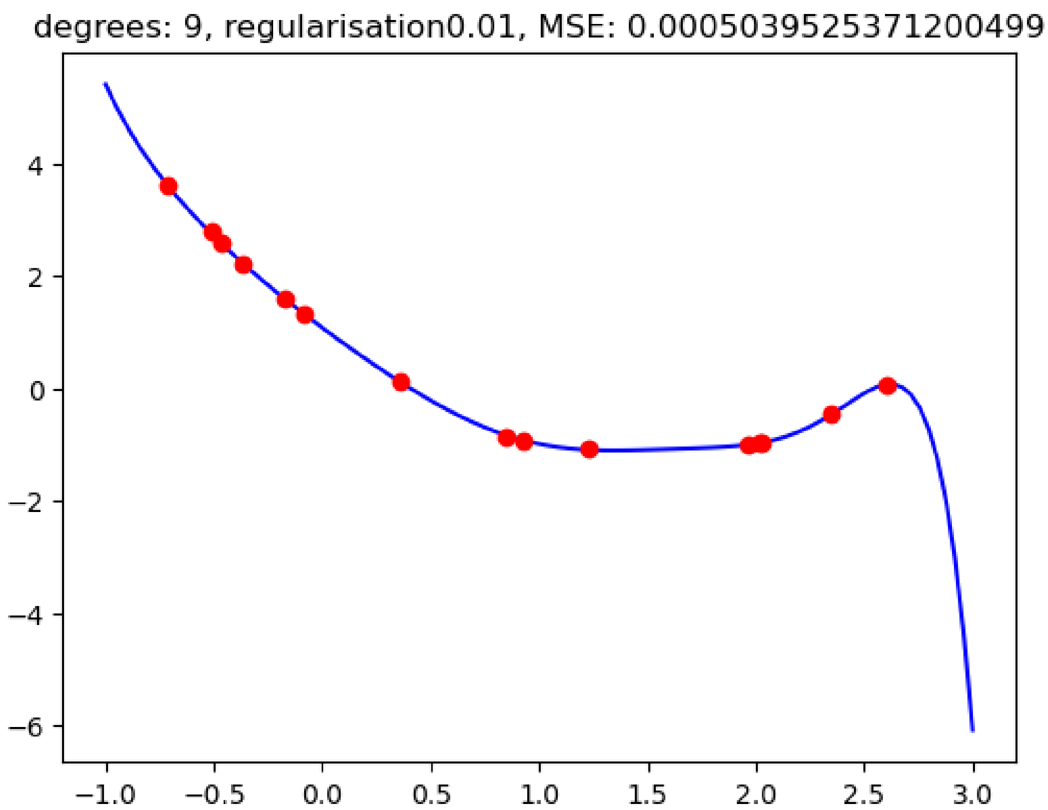
2c

```
[ ]: #create empty list
list_performance = list()

for n_sample in [15, 100]:
    for noise in [0.05]:
        data = problem2_evaluate_function_on_random_noise(n_sample, noise)
        print("n_sample: ", n_sample, "noise: ", noise)
        for degree in [9]:
            for regularisation in [0, 0.01, 0.1, 1, 10, 100, 1000]:
                print("regularisation: ", regularisation)
                mse, coeffs = plot_fitted_polynomial(data[0], data[1], degree,␣
 ↪regularisation)
                #add mse, coeffs tupel to list
                list_performance.append((n_sample, noise, degree,␣
 ↪regularisation, mse, coeffs))
                plt.show()
```

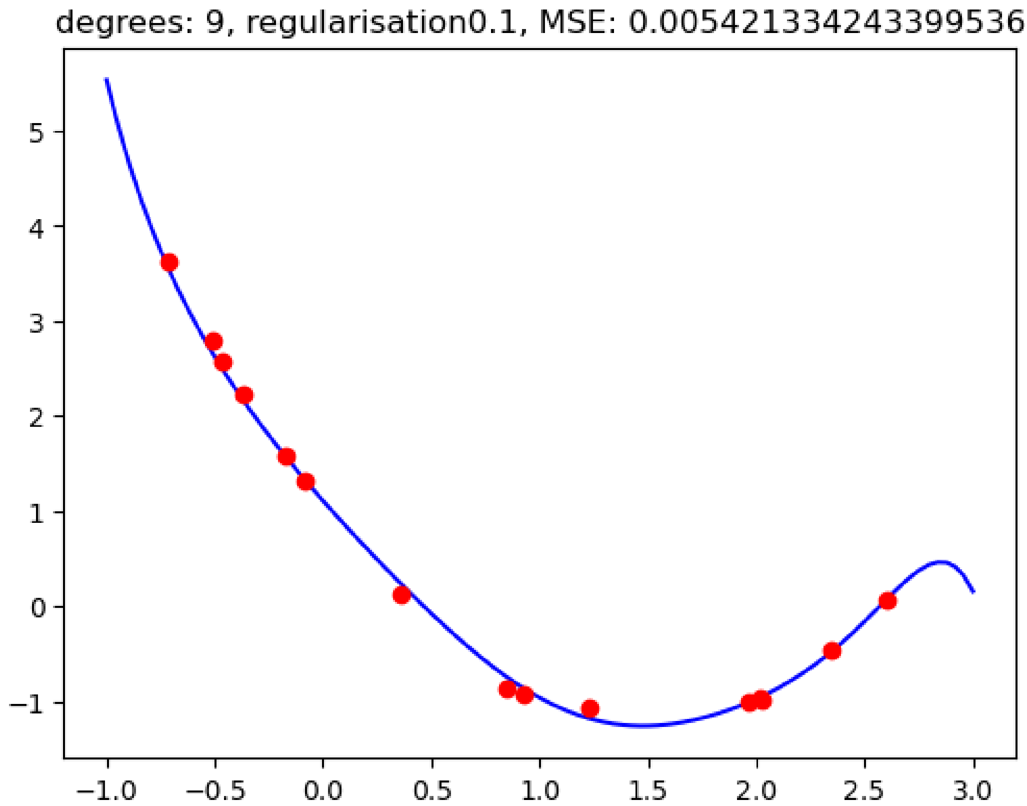
```
n_sample:  15 noise:  0.05
regularisation:  0
```



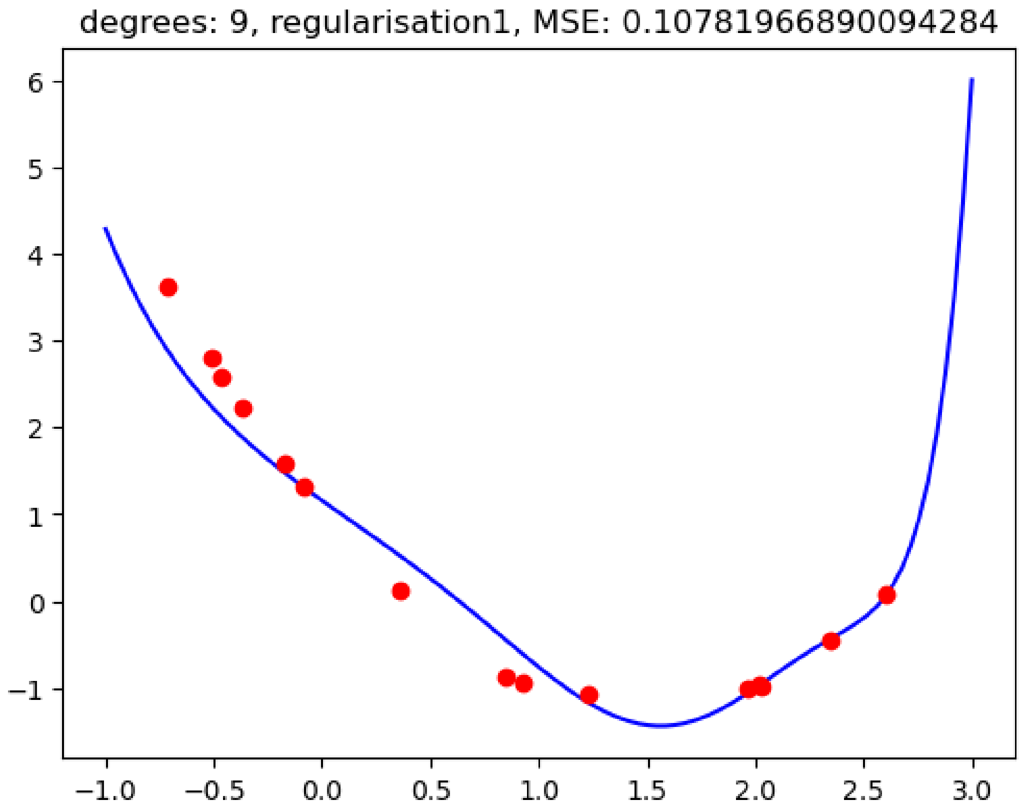

```
regularisation:  0.01
```



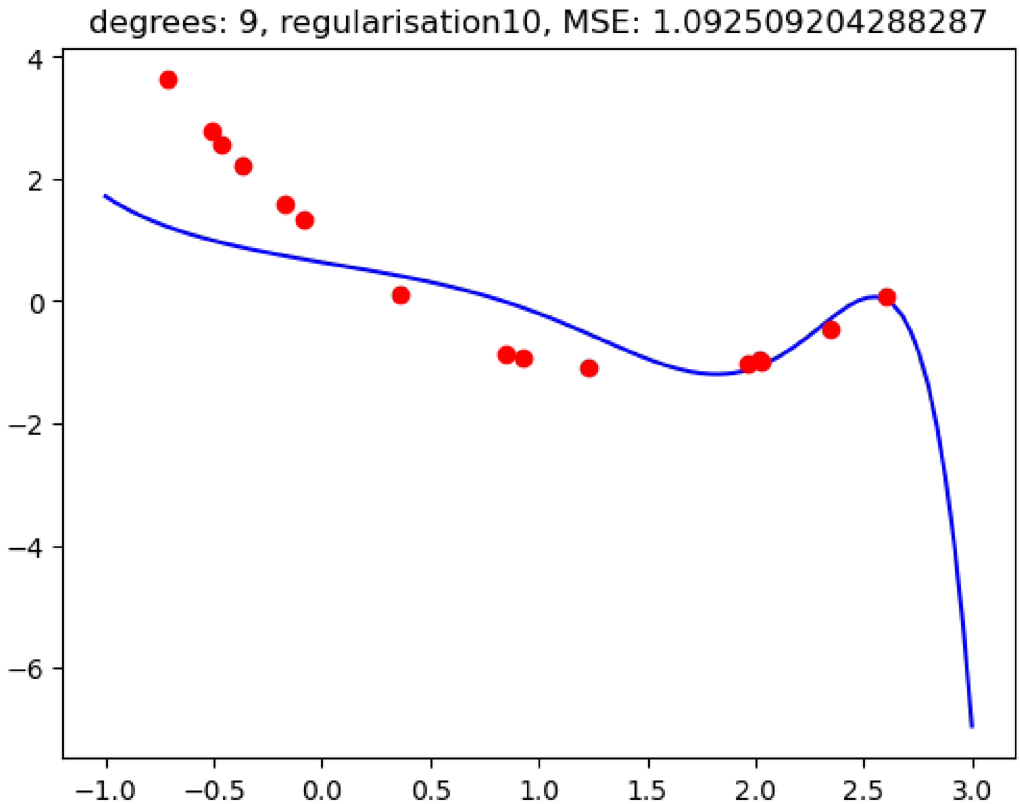

```
regularisation:  0.1
```



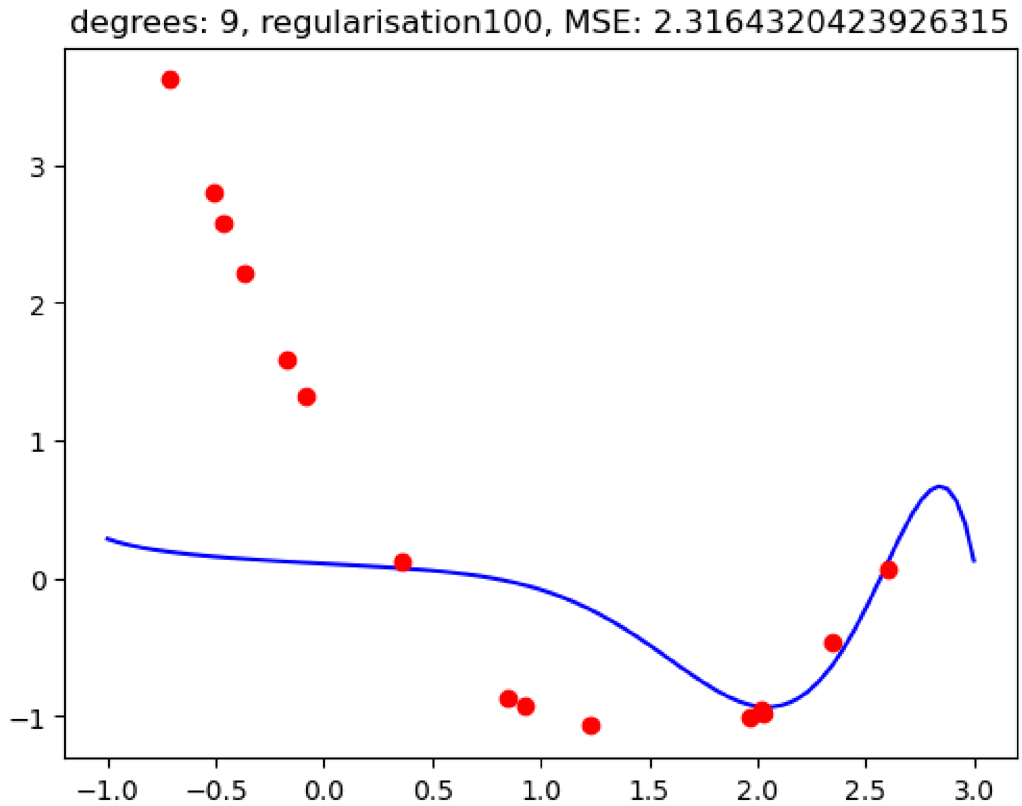

```
regularisation:  1
```



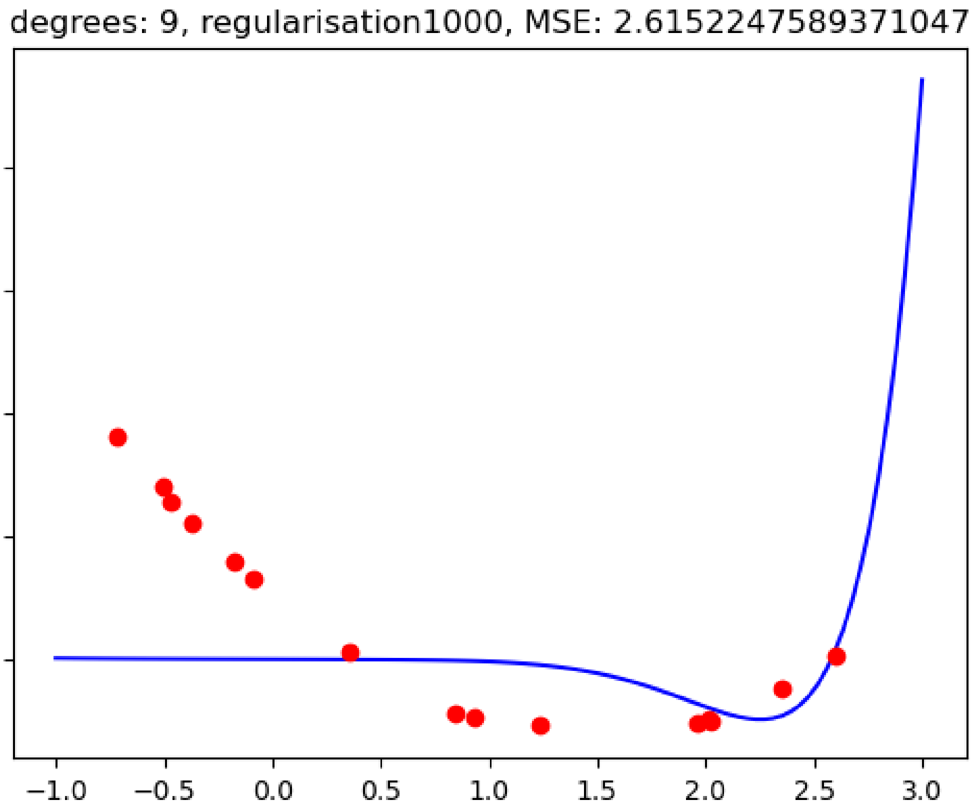

```
regularisation:  10
```



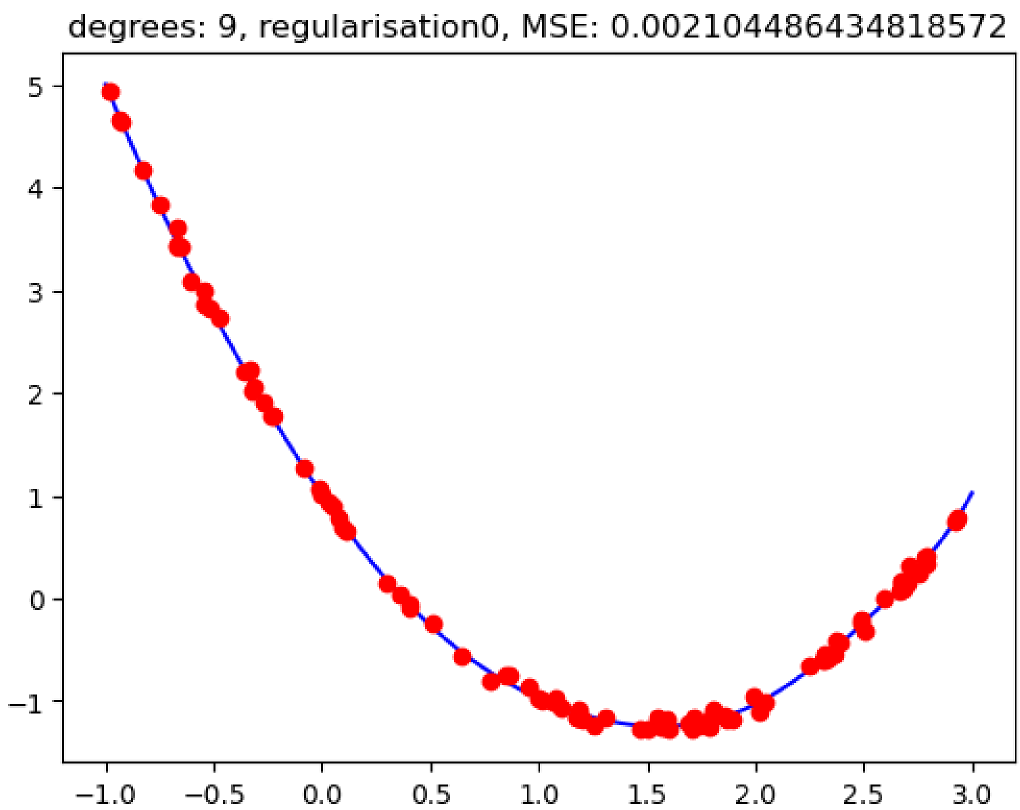

```
regularisation:  100
```



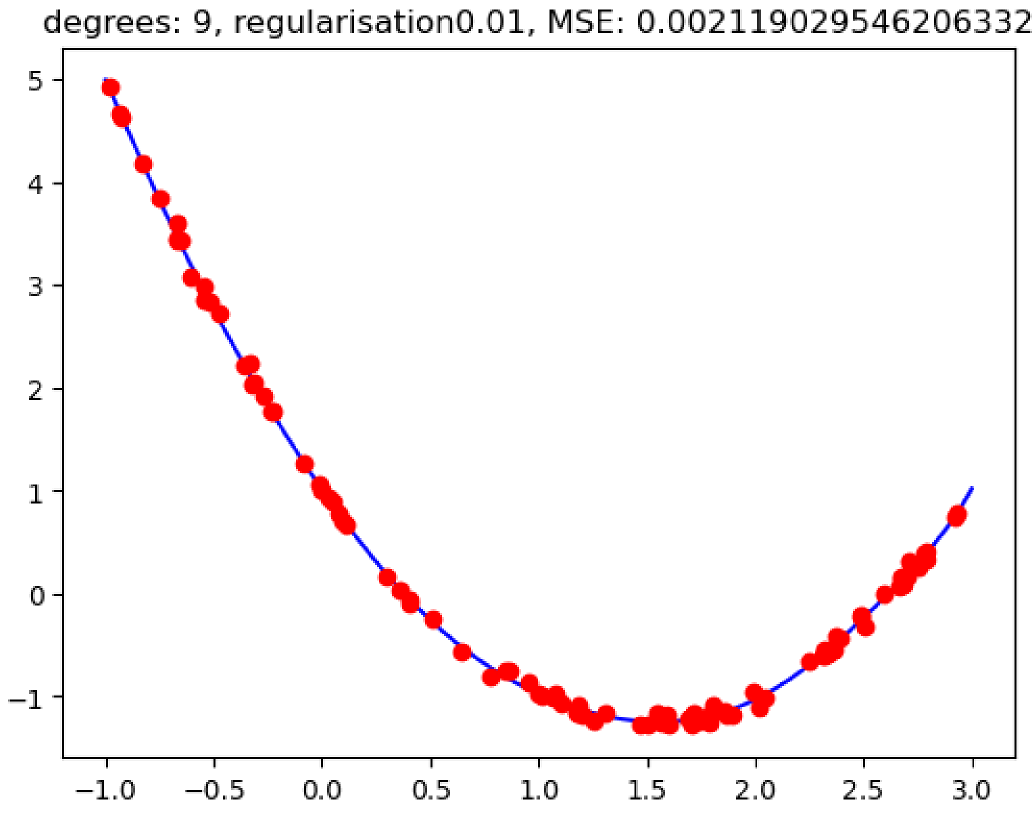

```
regularisation:  1000
```



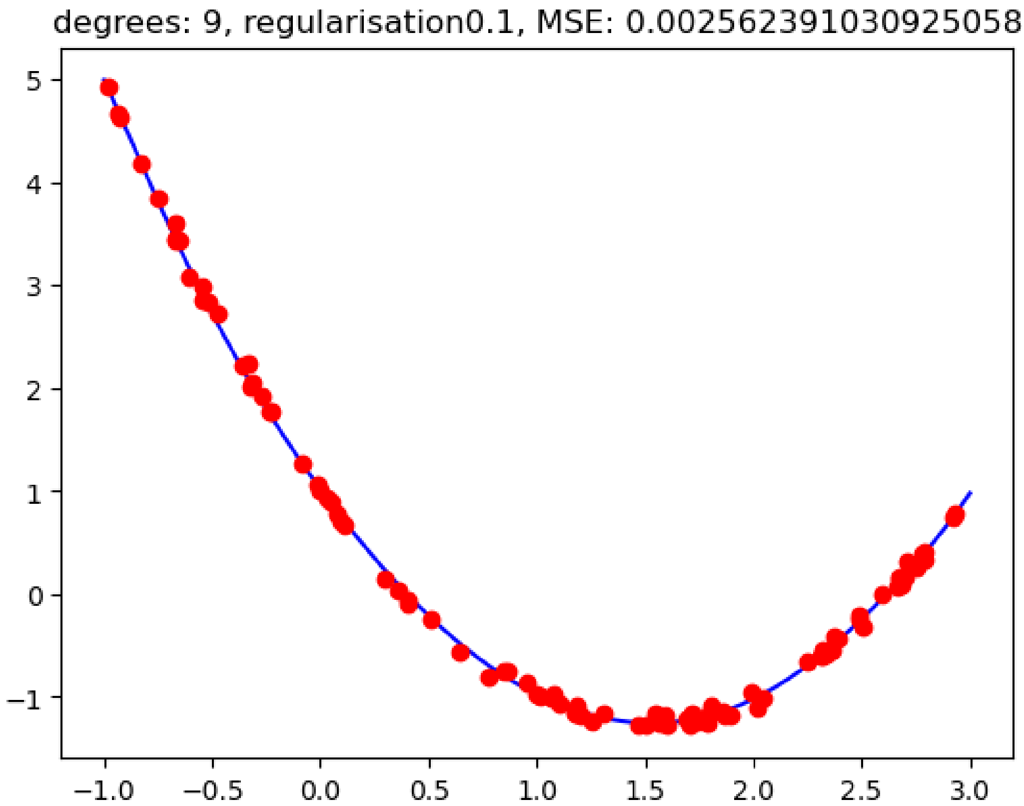

```
n_sample:  100 noise:  0.05
regularisation:  0
```



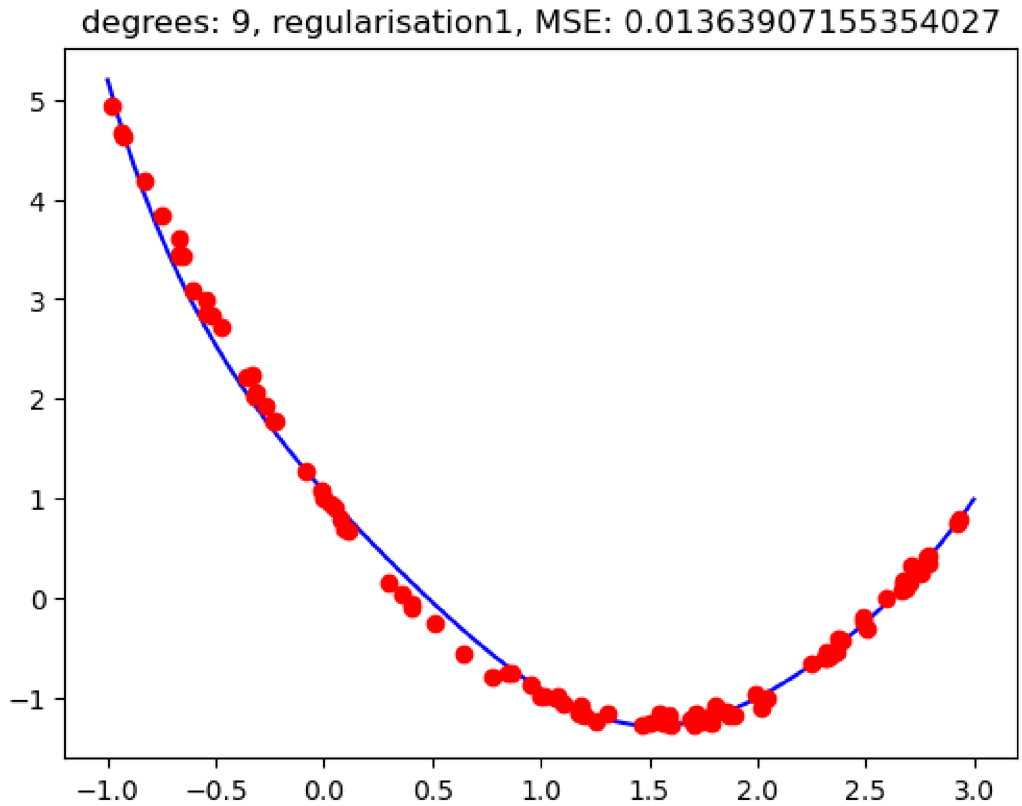
`regularisation:  0.01`



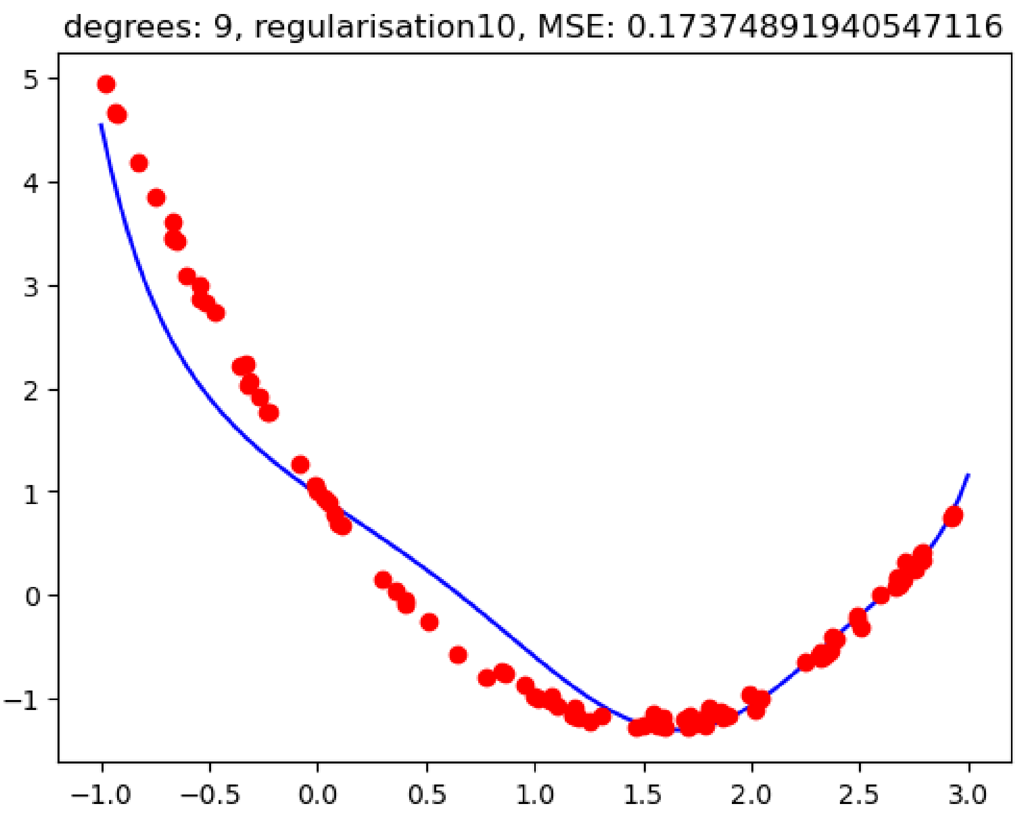

```
regularisation:  0.1
```



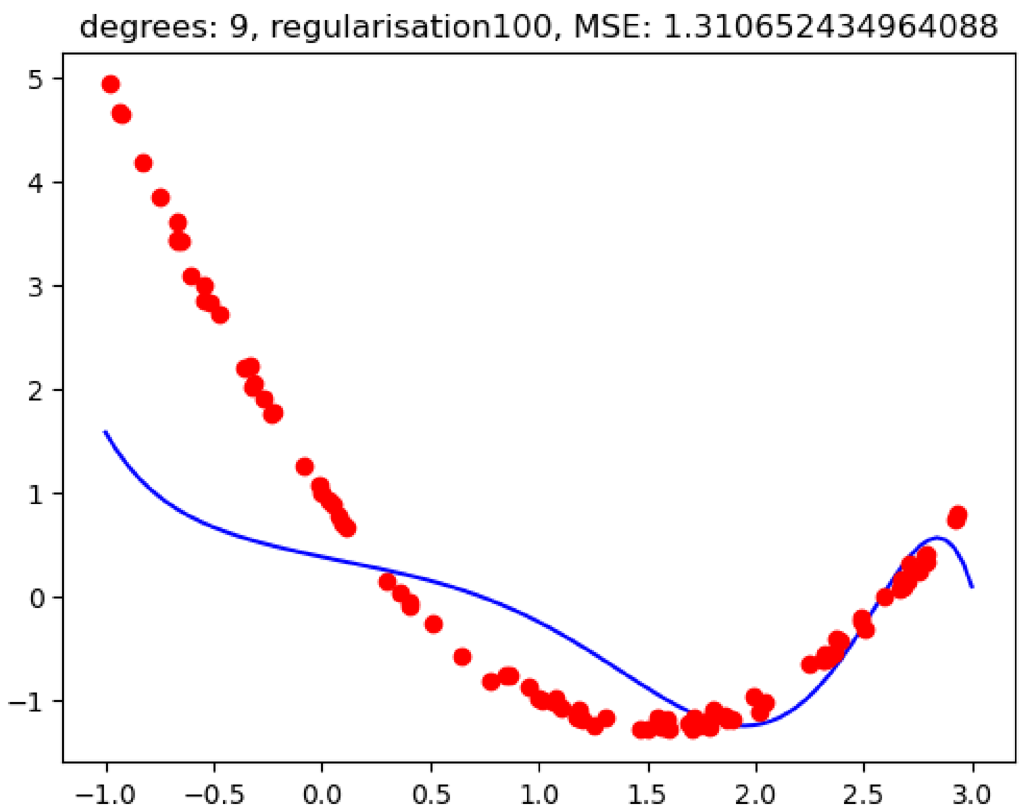

```
regularisation:  1
```



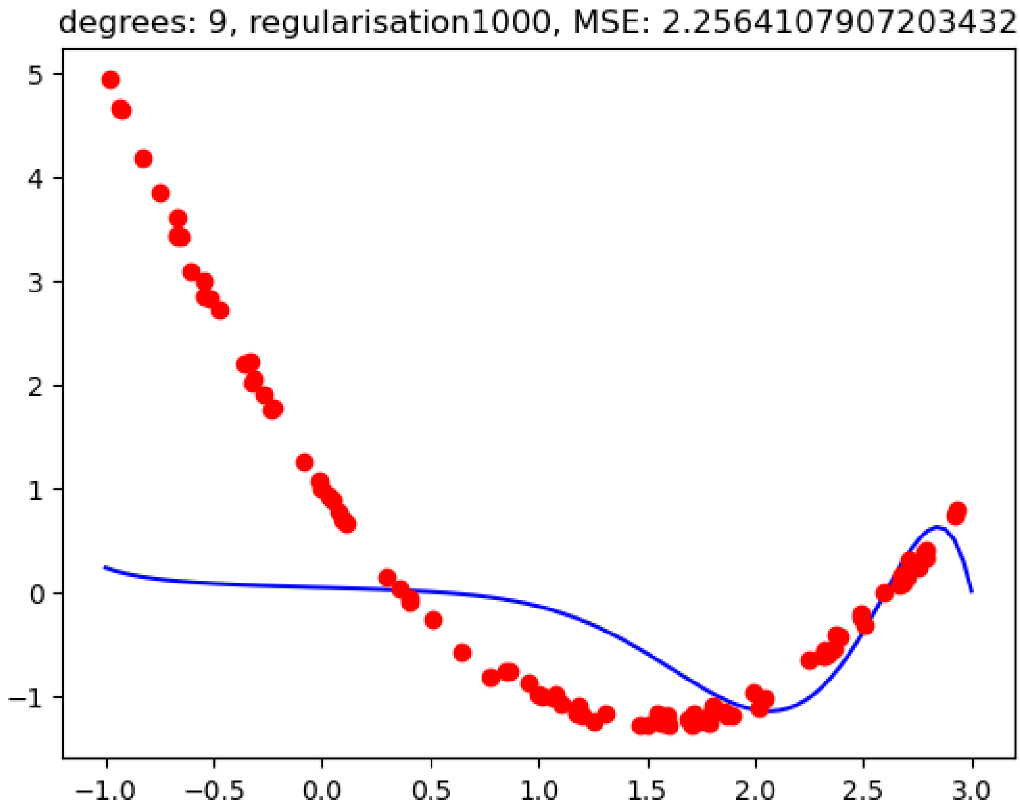

```
regularisation:  10
```




```
regularisation:  100
```




```
regularisation:  1000
```



Regularisation 1000 results in underfitting. Regularisation 1 works well. Regularisation 0 results in overfitting.

fridljandd_assignment1_problem3_4

February 5, 2023

[ ]:
```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from collections import Counter
import matplotlib.pyplot as plt
#logistic regression classifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
```

[ ]:
```
from ps1_functions import problem3_knn_classifier
```

[ ]:
```
#load prob3_data_seed.dat
data = np.genfromtxt('prob3_data_seed.dat')

X = data[:,0:6]
Y = data[:,7]
print(data)
```

```
[[15.26   14.84    0.871  … 2.221   5.22    1.    ]
 [14.88   14.57    0.8811 … 1.018   4.956   1.    ]
 [14.29   14.09    0.905  … 2.699   4.825   1.    ]
 …
 [13.2    13.66    0.8883 … 8.315   5.056   3.    ]
 [11.84   13.21    0.8521 … 3.598   5.044   3.    ]
 [12.3    13.34    0.8684 … 5.637   5.063   3.    ]]
```

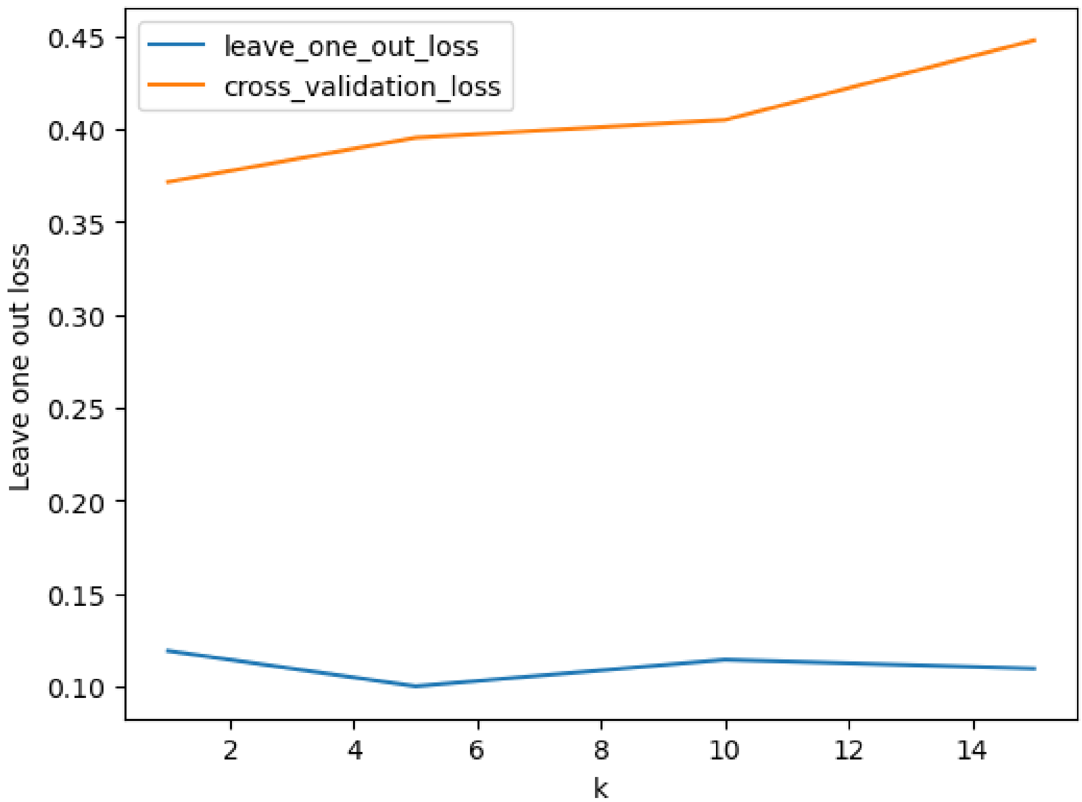
[ ]:
```
#min-max normalization of data columns
min = np.min(X, axis=0)
max = np.max(X, axis=0)
X = (X - min) / (max - min)
```

1; 5; 10; 15

[ ]:
```
def cross_validation(X, Y, k, folds = 5):
    """
    Leave one out cross validation for KNN classifier
    :param X: input data
    :param Y: class labels
```

```
    :param k: number of nearest neighbors
    :param folds: number of folds
    :return: accuracy
    """
    loss = list()
    X_folds = np.array_split(X, folds)
    Y_folds = np.array_split(Y, folds)

    for i in range(folds):
        hold_out = [j for j in range(X.shape[0]) if j != i]

        #combine hold_out from X_folds and Y_folds
        X_hold_out_train = np.concatenate(X_folds[:(i-1)] + X_folds[(i+1):],␣
 ↪axis=0)
        Y_hold_out_train = np.concatenate(Y_folds[:(i-1)] + Y_folds[(i+1):],␣
 ↪axis=0)

    # X_hold_out_train = [X_folds[j] for j in hold_out]
        #X_hold_out_train = np.vstack(X_hold_out_train)
        #Y_hold_out_train = np.vstack(Y_folds[j] for j in hold_out)
        X_leave_out_test = X_folds[i]
        Y_leave_out_test  = Y_folds[i].flatten()

        Y_predicted = problem3_knn_classifier(X_hold_out_train,␣
 ↪Y_hold_out_train, X_leave_out_test, k).flatten()

        loss_i = np.mean(Y_predicted != Y_leave_out_test)
        #print('Leave out: ', leave_out, 'Loss: ', loss_i)
        loss.append(loss_i)

    #average of loss
    return np.mean(loss)
```

[ ]:
```
ks = [1, 5, 10, 15]
cross_validation_loss = [cross_validation(X, Y, k, folds = 5) for k in ks]
#
leave_one_out_loss = [cross_validation(X, Y, k, folds = X.shape[0]) for k in ks]
#plot leave_one_out_loss and cross_validation_loss on same plot
plt.plot(ks, leave_one_out_loss, label='leave_one_out_loss')
plt.plot(ks, cross_validation_loss, label='cross_validation_loss')
plt.legend()
plt.xlabel('k')
plt.ylabel('Leave one out loss')
plt.show()
```



Problem 2 c

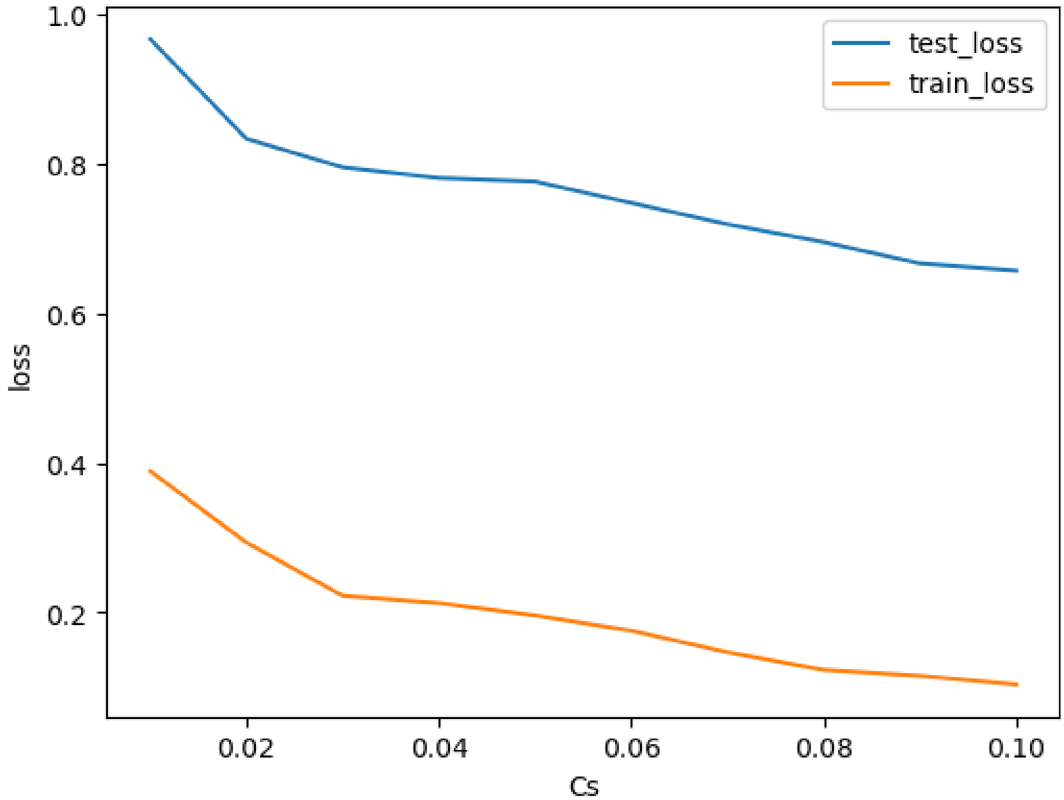
```
[ ]: def cross_validation_general(X, Y, classifier, cv = 5):
         """
         Leave one out cross validation for KNN classifier
         :param X: input data
         :param Y: class labels
         :param k: number of nearest neighbors
         :param cv: number of folds
         :return: accuracy
         """
         test_loss = list()
         train_loss = list()

         X_folds = np.array_split(X, cv)
         Y_folds = np.array_split(Y, cv)

         for i in range(cv):
             hold_out = [j for j in range(X.shape[0]) if j != i]

             #combine hold_out from X_folds and Y_folds
```

```
        X_hold_out_train = np.concatenate(X_folds[:(i-1)] + X_folds[(i+1):],␣
↪axis=0)
        Y_hold_out_train = np.concatenate(Y_folds[:(i-1)] + Y_folds[(i+1):],␣
↪axis=0)

        X_leave_out_test = X_folds[i]
        Y_leave_out_test  = Y_folds[i].flatten()

        classifier.fit(X_hold_out_train, Y_hold_out_train)

        Y_predicted_test = classifier.predict(X_leave_out_test).flatten()
        Y_predicted_train = classifier.predict(X_hold_out_train).flatten()

        test_loss_i = np.mean(Y_predicted_test != Y_leave_out_test)
        train_loss_i = np.mean(Y_predicted_train != Y_hold_out_train)

        #print('Leave out: ', leave_out, 'Loss: ', loss_i)
        test_loss.append(test_loss_i)
        train_loss.append(train_loss_i)

    #average of loss
    test_loss = np.mean(test_loss)
    train_loss = np.mean(train_loss)
    #print('Test loss: ', test_loss, 'Train loss: ', train_loss)

    return test_loss, train_loss
```

[ ]:
```
cross_validation_general(X, Y, classifier = LogisticRegression())
```

[ ]: (0.4619047619047619, 0.07281746031746031)

[ ]:
```
#sequence from 0.01 to 0.1 with 10 steps
Cs = np.linspace(0.01, 0.1, 10)

cross_validation_loss = [cross_validation_general(X, Y, classifier =␣
↪SVC(kernel="linear", C = C_i)) for C_i in Cs]
cross_validation_loss = np.vstack(cross_validation_loss)

plt.plot(Cs, cross_validation_loss[:,0], label='test_loss')
plt.plot(Cs, cross_validation_loss[:,1], label='train_loss')
plt.legend()
plt.xlabel('Cs')
plt.ylabel('loss')
plt.show()
```



Problem 4

Problem 4 a

Consider a single perceptron. Let σ be the activation function of the perceptron i.e. $\sigma(x) = 1(x > 0)$. Let w denote the weights and b the bias. Then the output of the perceptron for an input x is $\sigma(wx + b)$. Rescaling the weights and bias by $c > 0$ is

$$\sigma(cwx + cb) = \sigma(c(wx + b)) = 1(c(wx + b) > 0) = 1(wx + b > 0) = \sigma(cwx + cb).$$

We used $c > 0$. Since this holds true for every perceptron in a perceptron network, rescaling does not behave the behaviour.

Problem 4 b

The sigmoid function is

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Then

$$\sigma(c(wx + b)) = \frac{1}{1 + e^{-c(wx+b)}} = \frac{1}{1 + (e^{-(wx+b)})^c}.$$

We see that for $wx + b \neq 0$ we have $\lim_{c\to\infty} \sigma(c(wx + b)) = 1(wx + b > 0)$, which is exactly the behavior of a perceptron. For $wx + b \neq 0$ we have $\sigma(c(wx + b)) = 0.5$ for all c.

Problem 4.3

```
[ ]: #sigmoid function
     def perceptron(x):
         return np.where(x > 0, 1, 0)

     def sigmoid(x):
         return 1 / (1 + np.exp(-x))
```

translate

```
[ ]: W_1 = np.array([[0.6, 0.5, -0.6], [-0.7, 0.4, 0.8]])
     W_2 = np.array([[1, 1]])

     b_1 = np.array([-0.4, -0.5])
     b_2 = np.array([-0.5])
```

```
[ ]: def simple_neural_network(input, W_1, W_2, b_1, b_2, activation_function =␣
      ↪sigmoid):
         #print('input: ', input)
         Z_1 = np.dot(W_1, input) + b_1
         #print('Z_1: ', Z_1)
         A_1 = activation_function(Z_1)
         #print('A_1: ', A_1)

         Z_2 = np.dot(W_2, A_1) + b_2
         #print('Z_2: ', Z_2)

         A_2 = activation_function(Z_2)
         #print('A_2: ', A_2)

         return A_2
```

```
[ ]: #matrix
     X0 = np.array([0,0,0]).T
     X1 = np.array([1,0,0]).T
     X2 = np.array([0,1,0]).T
     X3 = np.array([0,0,1]).T
     X4 = np.array([1,1,0]).T
     X5 = np.array([1,0,1]).T
     X6 = np.array([0,1,1]).T
     X7 = np.array([1,1,1]).T

     Xs = [X0, X1, X2, X3, X4, X5, X6, X7]
```

output of perceptron

problem 4.3

[ ]:
```
output  = [simple_neural_network(X_i, W_1 = W_1, W_2 = W_2, b_1 = b_1, b_2 =␣
 ↪b_2, activation_function = perceptron) for X_i in Xs]
for i in range(len(Xs)):
    print('X: ', Xs[i], 'output: ', output[i])
```

```
X:  [0 0 0] output:  [0]
X:  [1 0 0] output:  [1]
X:  [0 1 0] output:  [1]
X:  [0 0 1] output:  [1]
X:  [1 1 0] output:  [1]
X:  [1 0 1] output:  [0]
X:  [0 1 1] output:  [1]
X:  [1 1 1] output:  [1]
```

output of sigmoid nn, problem 4.4

[ ]:
```
output  = [simple_neural_network(X_i, W_1 = W_1, W_2 = W_2, b_1 = b_1, b_2 =␣
 ↪b_2, activation_function = sigmoid) for X_i in Xs]
for i in range(len(Xs)):
    print('X: ', Xs[i], 'output: ', output[i])
```

```
X:  [0 0 0] output:  [0.569265]
X:  [1 0 0] output:  [0.56986717]
X:  [0 1 0] output:  [0.62245933]
X:  [0 0 1] output:  [0.58501229]
X:  [1 1 0] output:  [0.61732588]
X:  [1 0 1] output:  [0.57508402]
X:  [0 1 1] output:  [0.63314399]
X:  [1 1 1] output:  [0.62831133]
```

Problem 4.5

list two-digit binary numbers as two-dimensional binary vectors

[ ]:
```
X0= np.array([0,0]).T
X1= np.array([1,0]).T
X2= np.array([0,1]).T
X3= np.array([1,1]).T
Xs = [X0, X1, X2, X3]
```

single digit addition as neural network

[ ]:
```
def single_digit_binary_addition(input):
    W_1 = np.array([[1,0,2], [0,1,2]]).T
    W_2 = np.array([[0,0,1],[1, 1,-2]])
```

```
    b_1 = np.array([0,0,-3])
    b_2 = np.array([0,0])
    return simple_neural_network(input, W_1 = W_1, W_2 = W_2, b_1 = b_1, b_2 =␣
 ↪b_2, activation_function = perceptron)
```

[ ]:
```
[print(X_i[0]," + ", X_i[1]," = ", single_digit_binary_addition(X_i)) for X_i␣
 ↪in Xs];
```

```
0  +  0  =  [0 0]
1  +  0  =  [0 1]
0  +  1  =  [0 1]
1  +  1  =  [1 0]
```

concatenate `single_digit_binary_addition`multiple times

[ ]:
```
def two_digit_binary_addition(binary_number_1, binary_number_2):
    #first digit
    N1 = single_digit_binary_addition(np.array([binary_number_1[1-0],␣
 ↪binary_number_2[1-0]]))
    D0 = N1[1-0]
    #second digit
    N2 = single_digit_binary_addition(np.array([binary_number_1[1-1],␣
 ↪binary_number_2[1-1]]))
    N3 = single_digit_binary_addition(np.array([N2[1-0], N1[1-1]]))
    D1 = N3[1-0]
    #third digit
    N4 = single_digit_binary_addition(np.array([N2[1-1], N3[1-1]]))
    D2 = N4[1-0]

    sum_result = np.array([D2, D1, D0])
    return sum_result
```

[ ]:
```
[[print(binary_number_1," + ", binary_number_2," = ",␣
 ↪two_digit_binary_addition(binary_number_1, binary_number_2)) for␣
 ↪binary_number_1 in Xs] for binary_number_2 in Xs];
```

```
[0 0]  +  [0 0]  =  [0 0 0]
[1 0]  +  [0 0]  =  [0 1 0]
[0 1]  +  [0 0]  =  [0 0 1]
[1 1]  +  [0 0]  =  [0 1 1]
[0 0]  +  [1 0]  =  [0 1 0]
[1 0]  +  [1 0]  =  [1 0 0]
[0 1]  +  [1 0]  =  [0 1 1]
[1 1]  +  [1 0]  =  [1 0 1]
[0 0]  +  [0 1]  =  [0 0 1]
[1 0]  +  [0 1]  =  [0 1 1]
[0 1]  +  [0 1]  =  [0 1 0]
[1 1]  +  [0 1]  =  [1 0 0]
```

```
[0 0]   +   [1 1]   =   [0 1 1]
[1 0]   +   [1 1]   =   [1 0 1]
[0 1]   +   [1 1]   =   [1 0 0]
[1 1]   +   [1 1]   =   [1 1 0]
```

fridljandd_assignment1_problem5

February 5, 2023

[ ]:
```
import torch
import torch.nn as nn  # neural network modules
import torch.nn.functional as F  # activation functions
import torch.optim as optim  # optimizer
from torch.autograd import Variable # add gradients to tensors
from torch.nn import Parameter # model parameter functionality
import torchvision.datasets as datasets

from sklearn.metrics import confusion_matrix
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

[ ]:
```
from prob5_fcnn import train, plot_accuracies_v_epoch
```

```
train data shape: torch.Size([1000, 784])
train label shape: torch.Size([1000])
test data shape: torch.Size([2000, 784])
test label shape: torch.Size([2000])
```

[ ]:
```
# parameters
learning_rate = 0.01 # Ha ha! This means it will learn really quickly, right?
#TODO Daniel increase epochs
num_epochs = 150 # Training for a long time to see overfitting
batch_size = 128
n_hidden_1 = 500

# TODO 5.2: Defining loss functions
loss_functions = {
    "CE": torch.nn.CrossEntropyLoss(),
    "MSE": torch.nn.MSELoss(),
    "L1": torch.nn.L1Loss()
}
loss_functions_label = "CE"

#regularization
p = 0.05
exp_reg = 2
```

```
lambda_reg = 0#.01 #0.001

activation_functions = {
    "sigmoid": nn.Sigmoid(),
    "relu": nn.ReLU(),
    "tanh": nn.Tanh()
}
activation_functions_label = "sigmoid"

# network parameters
num_input = 784  # MNIST data input (img shape: 28*28)
num_classes = 10  # MNIST total classes (0-9 digits)
```

Problem 5.1 best run

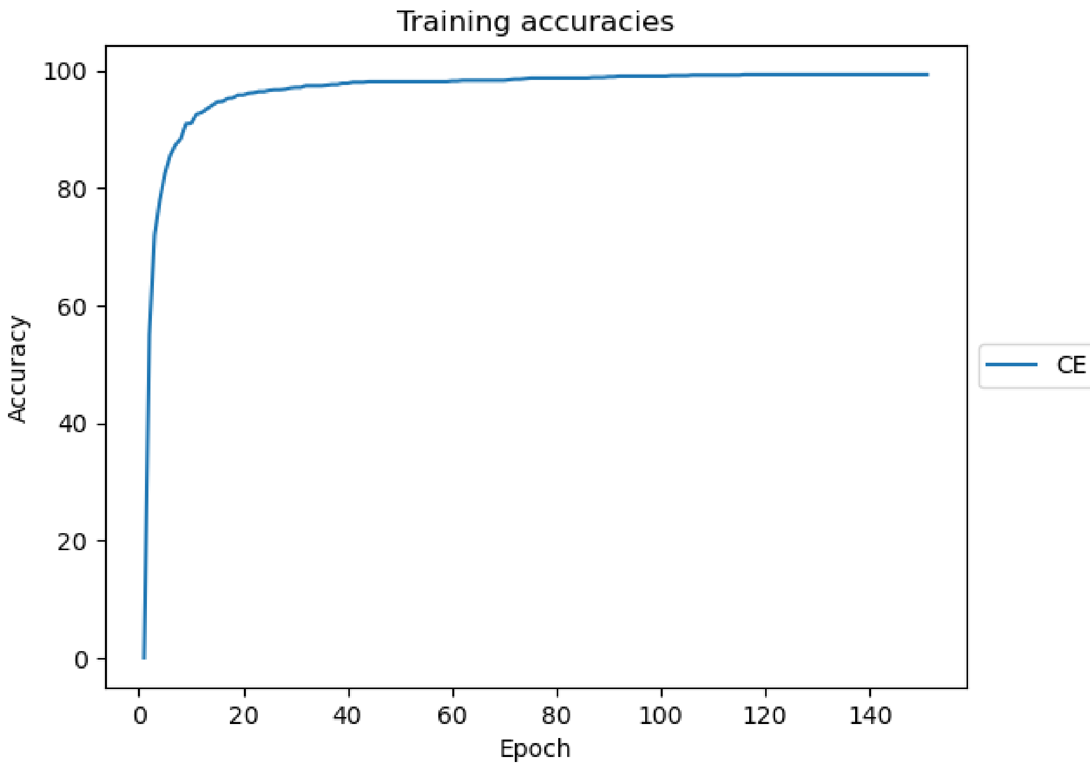
Print hyper parameters and accuracy generated with tensor board

[ ]:
```
#load hparams_table.csv with first line as header
hparams_table = np.genfromtxt('result_files/hparams_table.csv', delimiter=',',
 ↪dtype=None, encoding=None, names=True)
pd.DataFrame(hparams_table)
```

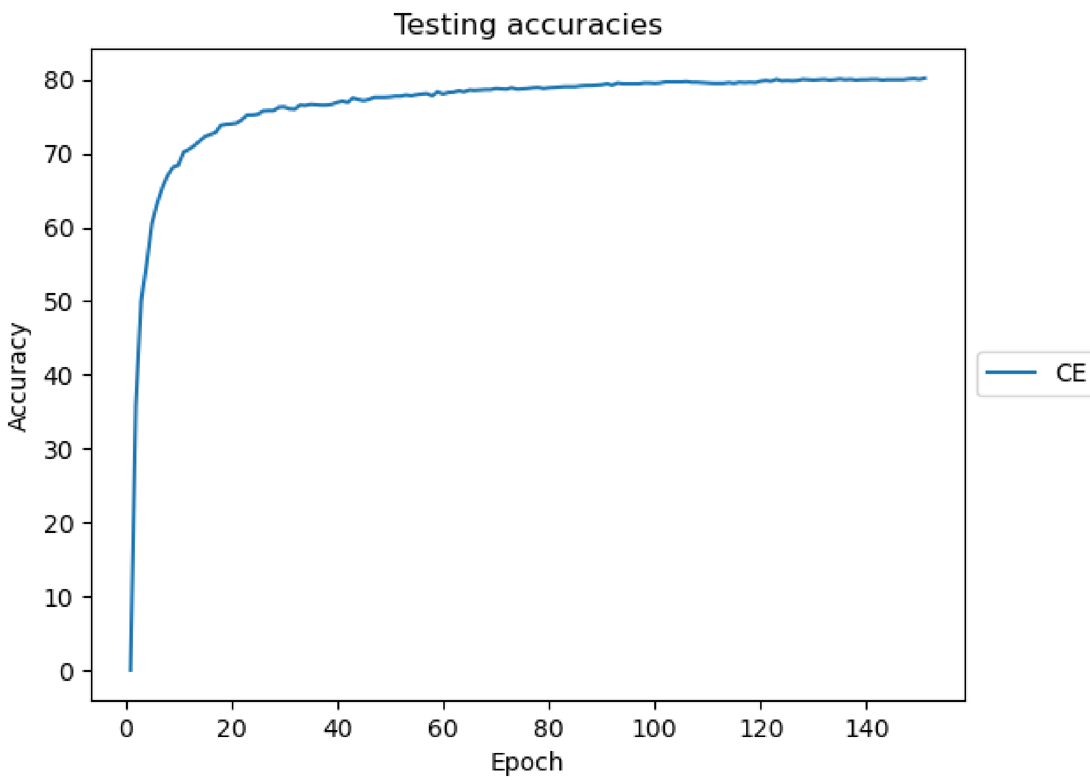
[ ]:
```
    learning_rate  num_epochs  n_hidden_1 loss_functions_label  \
0           0.100      1000.0        64.0                   CE
1           0.100       100.0        64.0                   CE
2           0.010       100.0        64.0                   CE
3           0.010       100.0      1000.0                   CE
4           0.001       100.0        64.0                   CE
5           0.100       100.0        64.0                   CE
6           0.100       100.0        64.0                  MSE
7           0.200       100.0        64.0                  MSE
8           0.200       100.0        64.0                   CE
9           0.100       100.0        64.0                   CE
10          0.100       100.0        64.0                   CE

   activation_functions_label  train_accuracy  test_accuracy
0                     sigmoid       99.699997      81.449997
1                     sigmoid       96.300003      80.550003
2                     sigmoid       95.599998      75.250000
3                     sigmoid       99.699997      81.199997
4                     sigmoid       77.599998      53.849998
5                     sigmoid       96.300003      80.550003
6                     sigmoid       96.900002      75.050003
7                     sigmoid       96.800003      77.750000
8                     sigmoid       86.000000      68.949997
9                     sigmoid       96.300003      80.550003
10                    sigmoid       96.300003      80.550003
```

[ ]:
```
metric_array, model_mse = train(loss_functions_label= "CE")
```

[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array, "CE", ax=ax)
plt.show()
```



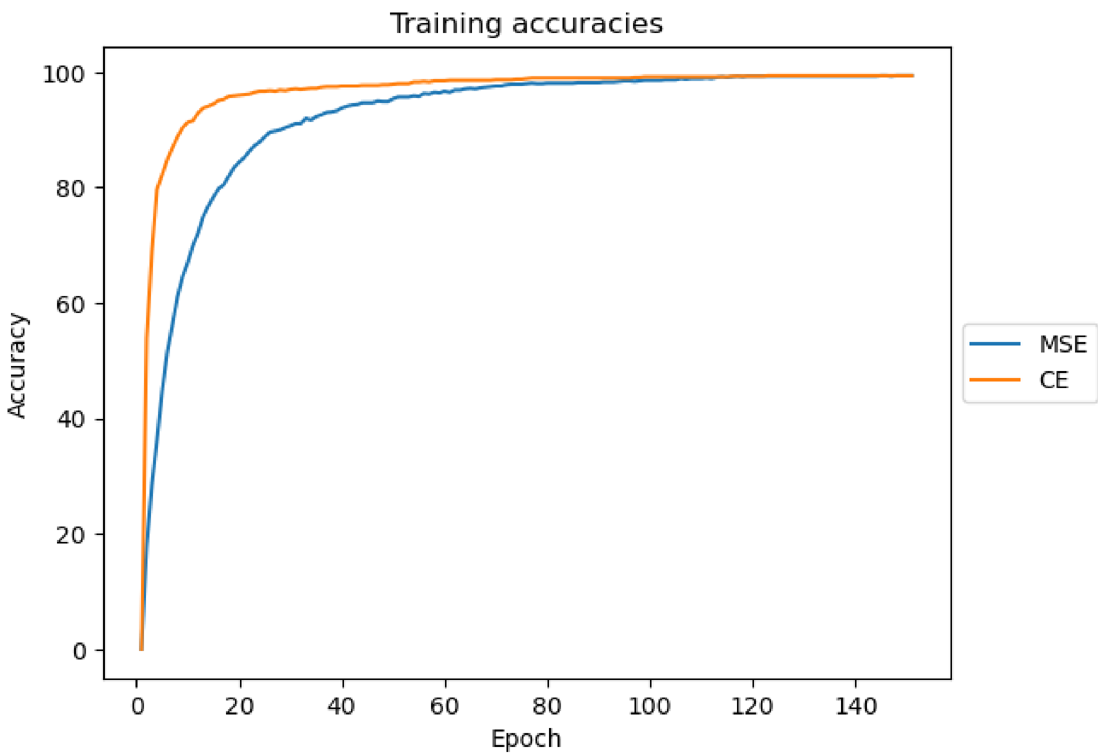
[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array, "CE", ax=ax, plot_training = False)
plt.show()
```



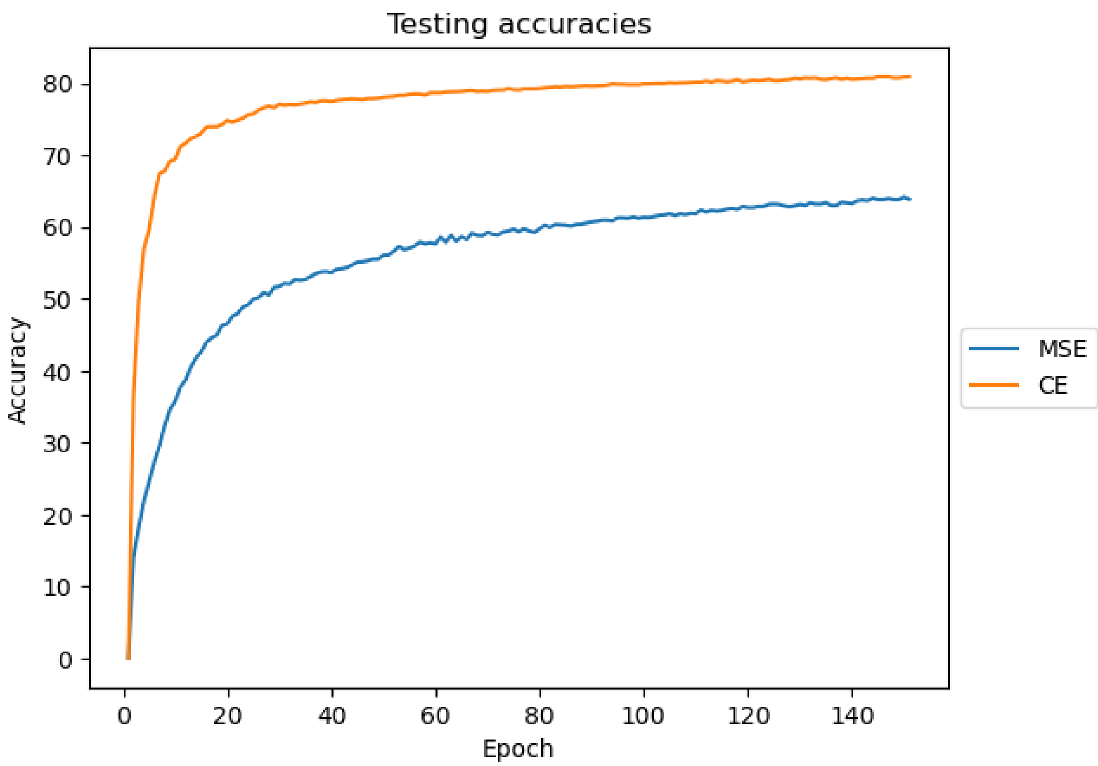
Problem 5.2

```
[ ]: metric_array_mse, model_mse = train(loss_functions_label= "MSE")
     metric_array_ce, model_ce = train(loss_functions_label= "CE")
```

```
[ ]: fig, ax = plt.subplots()
     plot_accuracies_v_epoch(metric_array_mse, "MSE", ax=ax)
     plot_accuracies_v_epoch(metric_array_ce, "CE", ax=ax)
     plt.show()
```



[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array_mse, "MSE", plot_training=False, ax=ax)
plot_accuracies_v_epoch(metric_array_ce, "CE", plot_training=False, ax=ax)
plt.show()
```

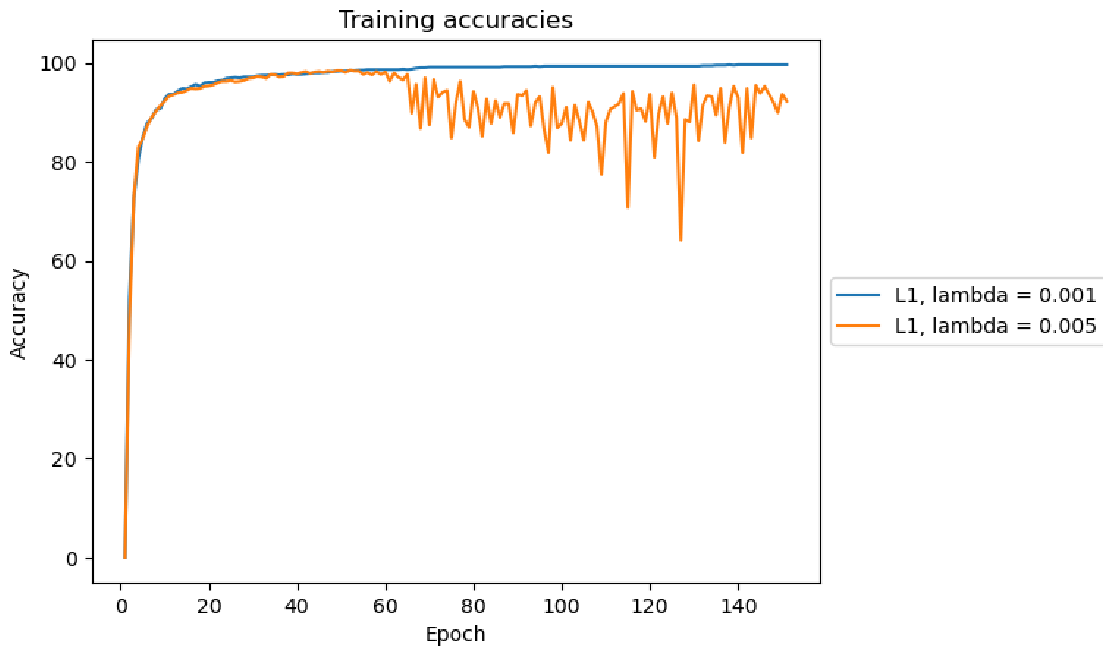


CE converges faster and has the highest test accuracy. When the CE cost function and the sigmoid activation are combined, the learning rate depends on the input error rate. Learning happens quickly. For MSE on the other hand, the learning is slow and it plateaus in the beginning.

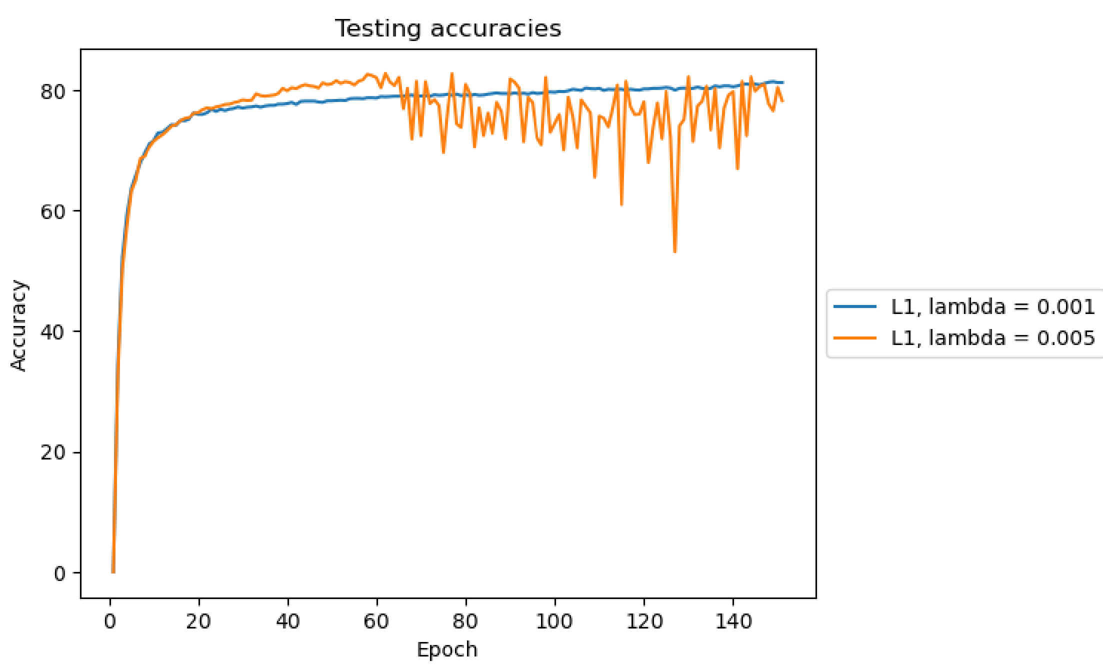
L1 regularisation

[ ]:
```
metric_array1, model1 = train(exp_reg = 1, lambda_reg= 0.001)
metric_array2, model2 = train(exp_reg = 1, lambda_reg= 0.005)
```

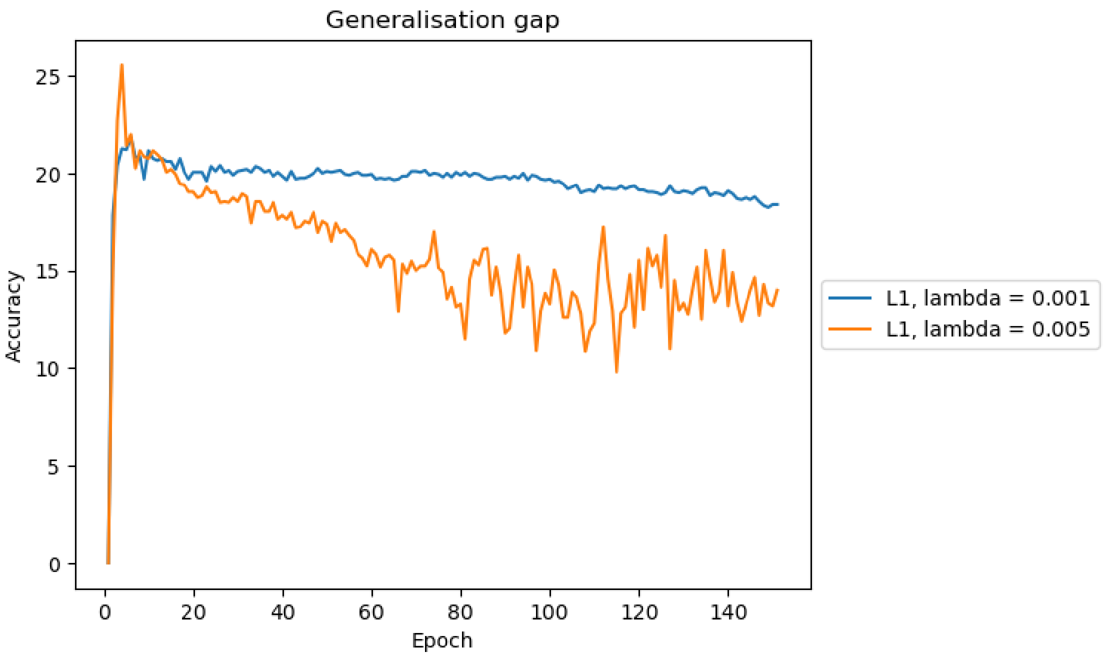
[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array1, "L1, lambda = 0.001", ax=ax)
plot_accuracies_v_epoch(metric_array2, "L1, lambda = 0.005", ax=ax)
plt.show()
```




```
[ ]: fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array1, "L1, lambda = 0.001",␣
 ↪plot_training=False, ax=ax)
plot_accuracies_v_epoch(metric_array2, "L1, lambda = 0.005",␣
 ↪plot_training=False, ax=ax)
plt.show()
```



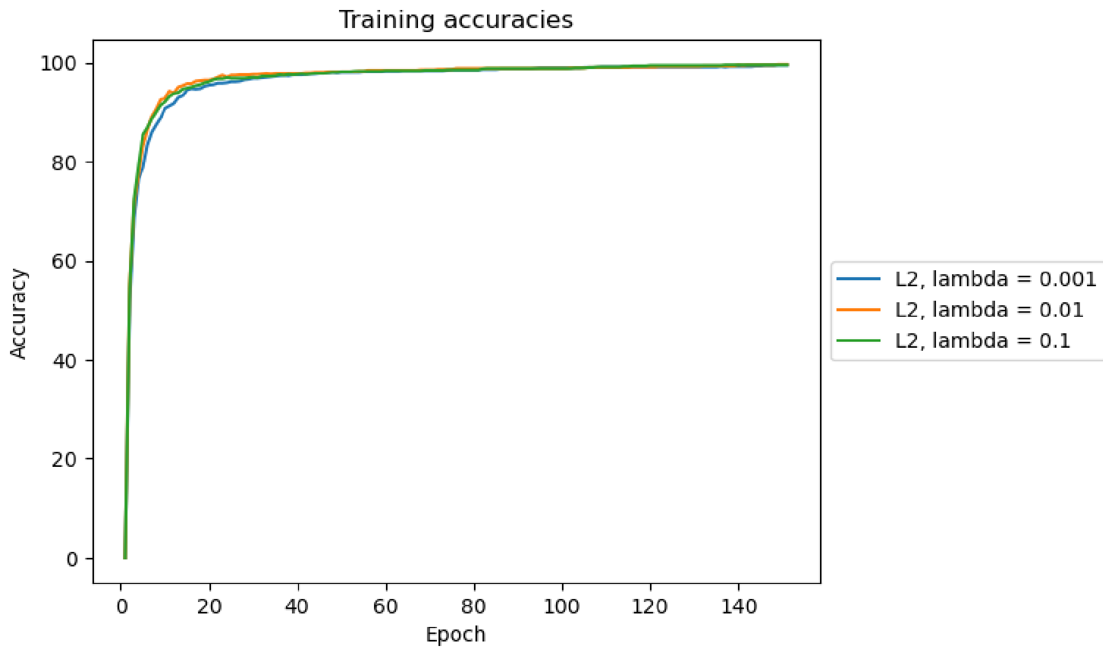
[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array1, "L1, lambda = 0.001",␣
↪generalisation_gap=True, ax=ax)
plot_accuracies_v_epoch(metric_array2, "L1, lambda = 0.005",␣
↪generalisation_gap=True, ax=ax)
plt.show()
```



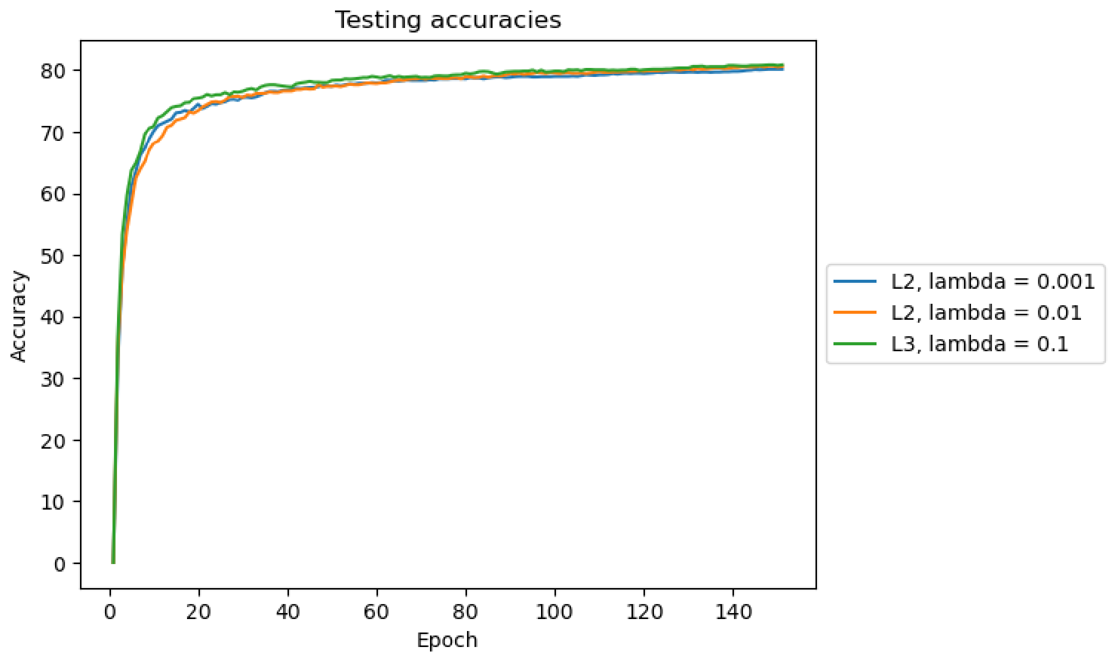
L2 regularisation

[ ]:
```
metric_array4, model1 = train(exp_reg = 2, lambda_reg= 0.001)
metric_array5, model2 = train(exp_reg = 2, lambda_reg= 0.01)
metric_array6, model3 = train(exp_reg = 2, lambda_reg= 0.1)
```

[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array4, "L2, lambda = 0.001", ax=ax)
plot_accuracies_v_epoch(metric_array5, "L2, lambda = 0.01", ax=ax)
plot_accuracies_v_epoch(metric_array6, "L2, lambda = 0.1", ax=ax)
plt.show()
```

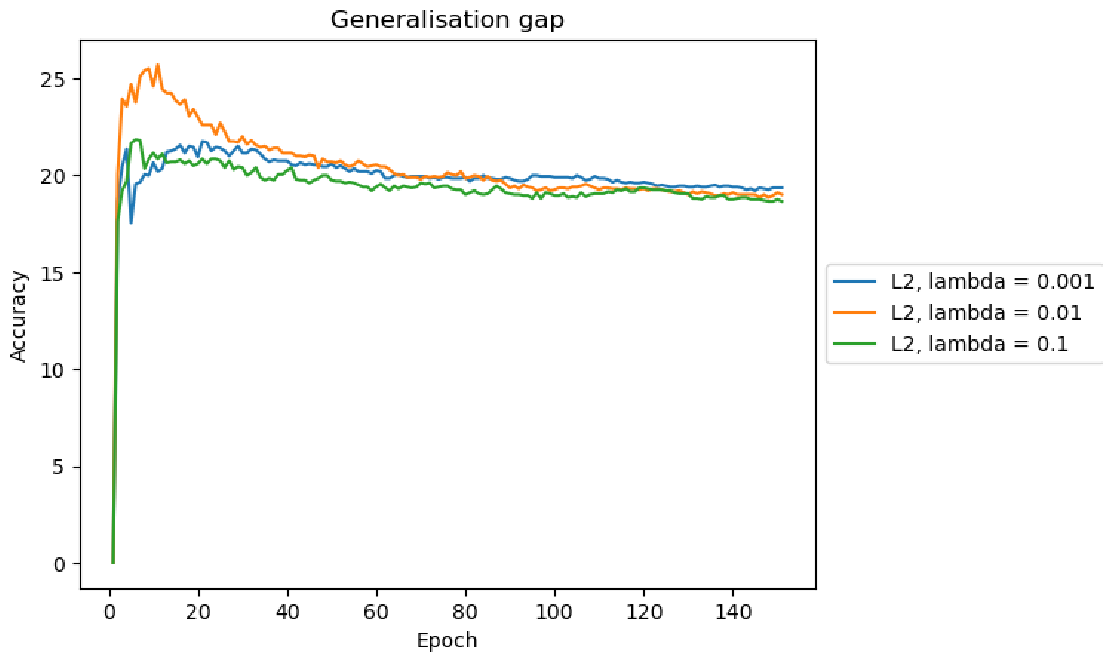


[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array4, "L2, lambda = 0.001", ax=ax,␣
 ↪plot_training=False)
plot_accuracies_v_epoch(metric_array5, "L2, lambda = 0.01", ax=ax,␣
 ↪plot_training=False)
plot_accuracies_v_epoch(metric_array6, "L3, lambda = 0.1", ax=ax,␣
 ↪plot_training=False)
plt.show()
```



[ ]:

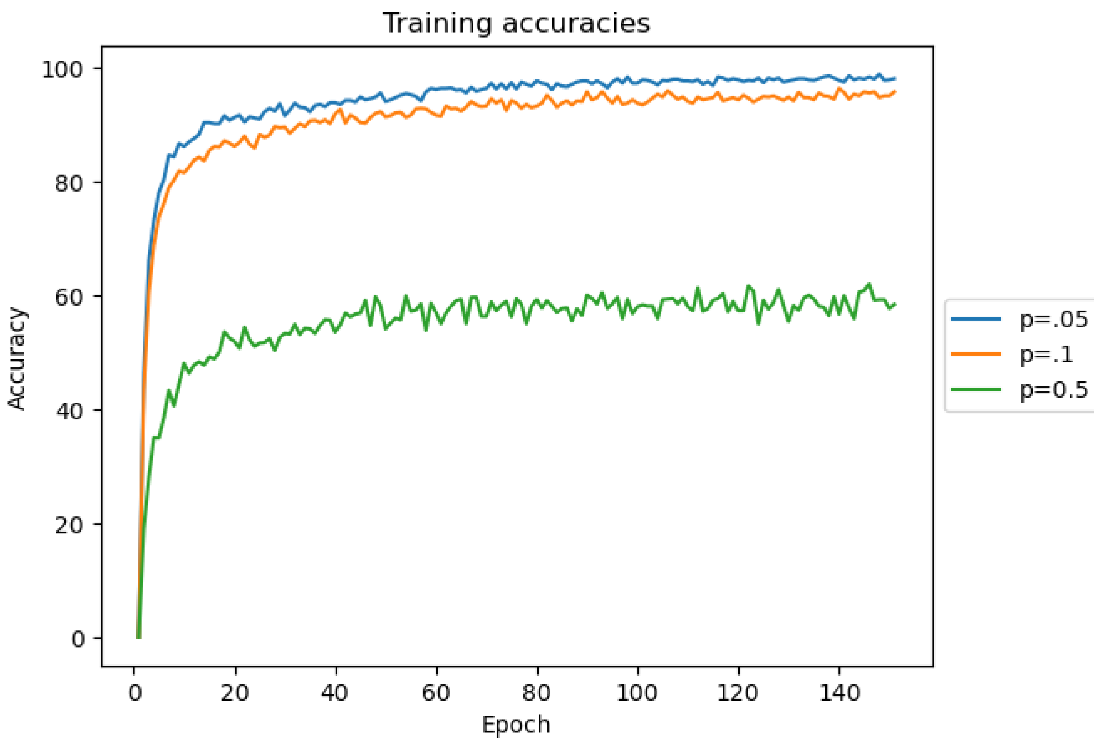
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array4, "L2, lambda = 0.001", ax=ax,␣
 ↪generalisation_gap=True)
plot_accuracies_v_epoch(metric_array5, "L2, lambda = 0.01", ax=ax,␣
 ↪generalisation_gap=True)
plot_accuracies_v_epoch(metric_array6, "L2, lambda = 0.1", ax=ax,␣
 ↪generalisation_gap=True)
plt.show()
```



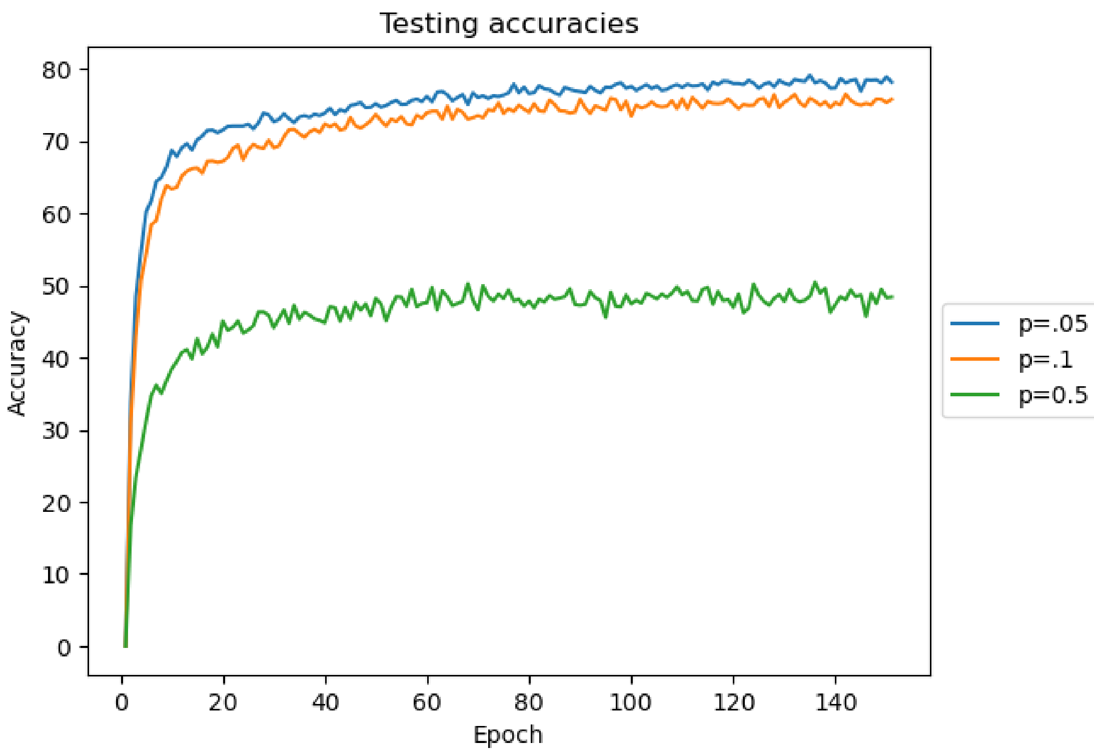
dropout

```
[ ]: metric_array7, model1 = train(p = 0.05)
     metric_array8, model2 = train(p = 0.1)
     metric_array9, model3 = train(p = 0.5)
```

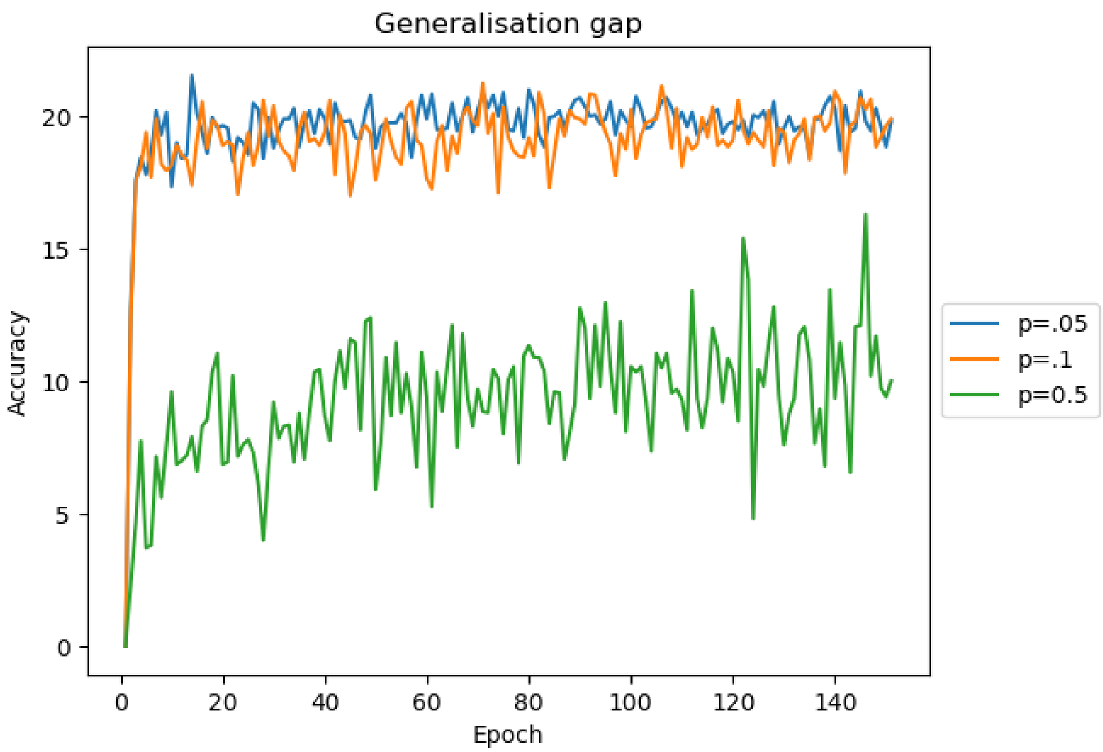
```
[ ]: fig, ax = plt.subplots()
     plot_accuracies_v_epoch(metric_array7, "p=.05", ax=ax)
     plot_accuracies_v_epoch(metric_array8, "p=.1", ax=ax)
     plot_accuracies_v_epoch(metric_array9, "p=0.5", ax=ax)
     plt.show()
```



[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array7, "p=.05", ax=ax, plot_training=False)
plot_accuracies_v_epoch(metric_array8, "p=.1", ax=ax, plot_training=False)
plot_accuracies_v_epoch(metric_array9, "p=0.5", ax=ax, plot_training=False)
plt.show()
```

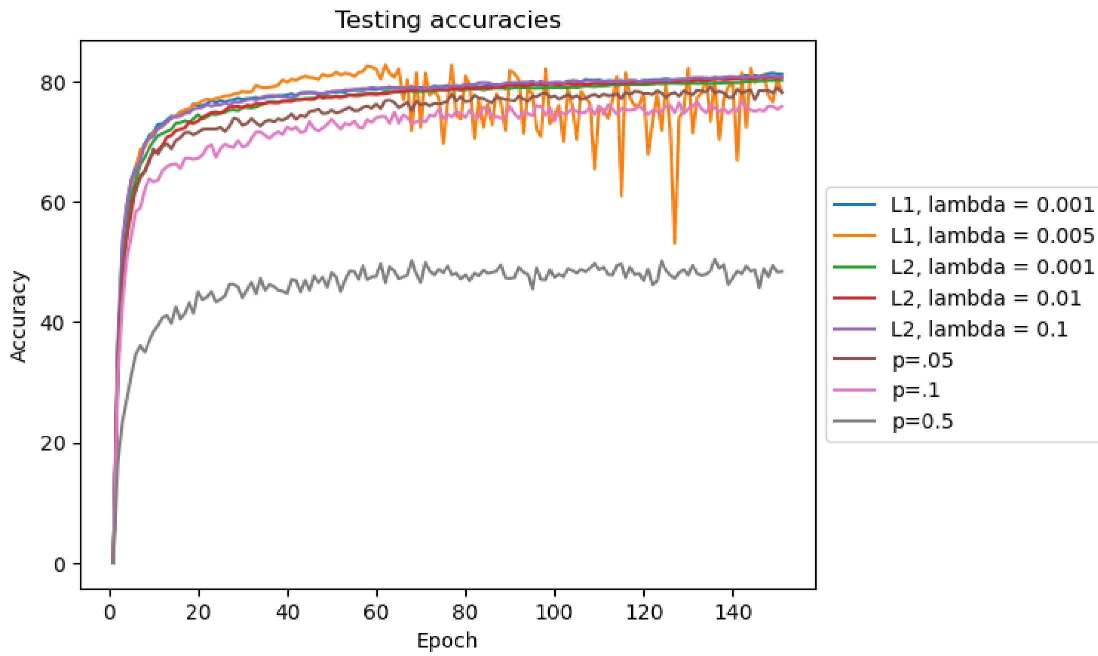


[ ]:
```
fig, ax = plt.subplots()
plot_accuracies_v_epoch(metric_array7, "p=.05", ax=ax, generalisation_gap=True)
plot_accuracies_v_epoch(metric_array8, "p=.1", ax=ax, generalisation_gap=True)
plot_accuracies_v_epoch(metric_array9, "p=0.5", ax=ax, generalisation_gap=True)
plt.show()
```



collected figure

```
[ ]: fig, ax = plt.subplots()
#L1
plot_accuracies_v_epoch(metric_array1, "L1, lambda = 0.001", ax=ax,␣
  ↪plot_training=False)
plot_accuracies_v_epoch(metric_array2, "L1, lambda = 0.005", ax=ax,␣
  ↪plot_training=False)
#L2
plot_accuracies_v_epoch(metric_array4, "L2, lambda = 0.001", ax=ax,␣
  ↪plot_training=False)
plot_accuracies_v_epoch(metric_array5, "L2, lambda = 0.01", ax=ax,␣
  ↪plot_training=False)
plot_accuracies_v_epoch(metric_array6, "L2, lambda = 0.1", ax=ax,␣
  ↪plot_training=False)
#p
plot_accuracies_v_epoch(metric_array7, "p=.05", ax=ax, plot_training=False)
plot_accuracies_v_epoch(metric_array8, "p=.1", ax=ax, plot_training=False)
plot_accuracies_v_epoch(metric_array9, "p=0.5", ax=ax, plot_training=False)
plt.show()
```



The results are sensitive to the parameters. The best regularisation is L1 with lambda = 0.001

Problem 5.4

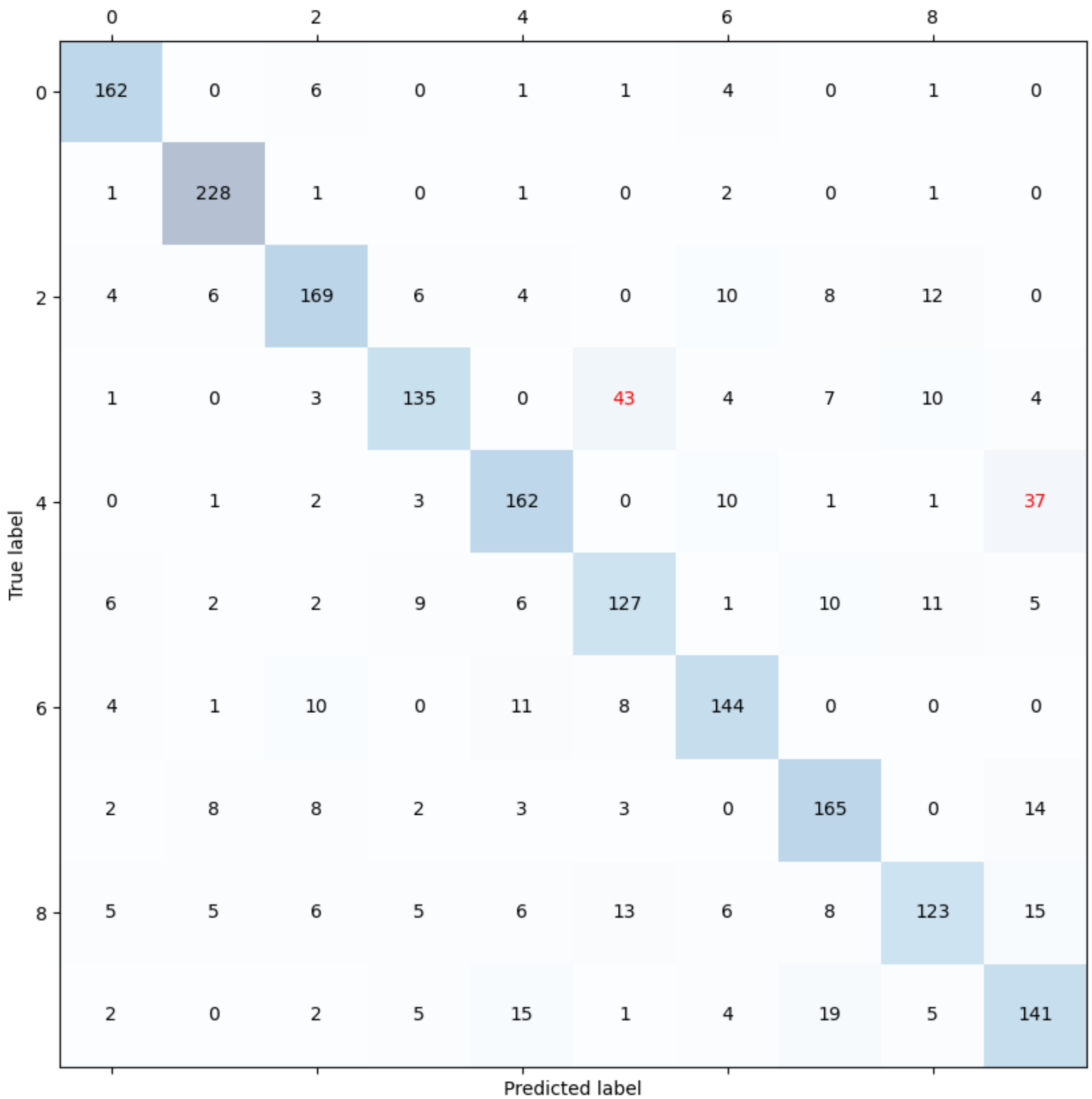
confusion matrix of missclassified digits

[ ]:
```
# Download the MNIST dataset
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=None)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=None)

# Separate into data and labels
# Reducing training dataset to 1000 points and test dataset to 2000 points in␣
 ↪order to create an overfitting model on
# which to study regularization later

# training data
train_data = mnist_trainset.data.to(dtype=torch.float32)[:1000]
train_data = train_data.reshape(-1, 784)
train_labels = mnist_trainset.targets.to(dtype=torch.long)[:1000]

print(f"train data shape: {train_data.size()}")
print(f"train label shape: {train_labels.size()}")

# testing data
```

```
test_data = mnist_testset.data.to(dtype=torch.float32)[:2000]
test_data = test_data.reshape(-1, 784)
test_labels = mnist_testset.targets.to(dtype=torch.long)[:2000]

print(f"test data shape: {test_data.size()}")
print(f"test label shape: {test_labels.size()}")

# Load into torch datasets
train_dataset = torch.utils.data.TensorDataset(train_data, train_labels)
test_dataset = torch.utils.data.TensorDataset(test_data, test_labels)
```

```
train data shape: torch.Size([1000, 784])
train label shape: torch.Size([1000])
test data shape: torch.Size([2000, 784])
test label shape: torch.Size([2000])
```

[ ]:
```
test_label_predicted = model1(test_data)
# get max
test_label_predicted = torch.argmax(test_label_predicted, dim=1)
confusion_matrix_output = confusion_matrix(test_labels, test_label_predicted)

#plot confusion matrix
fig, ax = plt.subplots(figsize=(10,10))
ax.matshow(confusion_matrix_output, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confusion_matrix_output.shape[0]):
    for j in range(confusion_matrix_output.shape[1]):
        #if confusion_matrix_output[i, j] > 15, print in red
        if confusion_matrix_output[i, j] > 20 and i != j:
            ax.text(x=j, y=i, s=confusion_matrix_output[i, j], va='center',␣
 ↪ha='center', color='red')
        else:
            ax.text(x=j, y=i, s=confusion_matrix_output[i, j], va='center',␣
 ↪ha='center')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()
```



mistaken digits are colored in red. E.g. the 3 is often mistaken for a 5.