

**Sayyed Mohsen Vazirizade**

**23398312**

[**smvazirizade@email.arizona.edu**](mailto:smvazirizade@email.arizona.edu)

**INFO 521**

**Introduction to Machine Learning**

Assignment 3

# Problem 1

Part A

The following script is a function and a for loop for the calculation process of pmf while

# -\*- coding: utf-8 -\*-

"""

Created on Sun Oct 1 12:30:35 2017

@author: smvaz

"""

#improting requeired lib

import math

def POISSON(Lambda, y):

#define a function to calculate poisson dist

pmf= math.exp(-Lambda) \* Lambda\*\*y / math.factorial(y)

return pmf

# a for loop for calculation of 4=<y=<9

Lambda=3

pmf=0

for y in range(4,10):

pmf=POISSON(Lambda,y)+pmf

print('pmf for 4=<y=<9 and Lambda=%s is %s' %(Lambda, pmf))

And the output of this script is

pmf for 4=<y=<9 and Lambda=3 is 0.35166562308765326

Part B

This part is just complement of the earlier part. So the answer is just 1- 0.35166562308765326= 0.6483343769123467

# Problem 2

If a=-4 and b=10

Using these values:

To solve it using approximate sampling method, I modified approx..expected.value.py as follow:

# approx\_expected\_value\_sin.py

# Port of approx\_expected\_value.m

# From A First Course in Machine Learning, Chapter 2.

# Simon Rogers, 01/11/11 [simon.rogers@glasgow.ac.uk]

# Approximating expected values via sampling

import numpy

import matplotlib.pyplot as plt

# Turn off annoying matplotlib warning

import warnings

warnings.filterwarnings("ignore", ".\*GUI is implemented.\*")

print('mohsen')

# We are trying to estimate the expected value of

# $f(y) = 60-0.1\*x-0.5\*x\*\*3+0.05\*x\*\*4$

##

# ... where

# $p(y)=U(-4,10)$

##

# ... which is given by:

# $\int f(y) p(y) dy$

##

# The analytic result is = 44.860...

# (NOTE: this just gives you the analytic result -- you should be able to derive it!)

# First, let's plot the function, shading the area under the curve for x=[-4,10]

# The following plot\_fn helps us do this.

# Information about font to use when plotting the function

font = {'family' : 'serif',

'color' : 'darkred',

'weight' : 'normal',

'size' : 12,

}

def plot\_fn(fn, a, b, fn\_name, resolution=100):

x = numpy.append(numpy.array([a]),

numpy.append(numpy.linspace(a, b, resolution), [b]))

# for example it makes it as [0 0 1 2 3 4 5 6 7 8 9 10 10]

# x = numpy.linspace(x\_begin, x\_end, resolution)

y = fn(x)

y[0] = 0 # the first

y[-1] = 0 # the last

plt.figure()

plt.fill(x, y, 'b', alpha=1)

# plt.fill\_between(x, y, y2=0, color='b', alpha=0.3)

fname, x\_tpos, y\_tpos = fn\_name()

plt.text(x\_tpos, y\_tpos, fname, fontdict=font)

plt.title('Area under function')

plt.xlabel('$y$')

plt.ylabel('$f(y)$')

x\_range = b - a

plt.xlim(a-(x\_range\*0.1), b+(x\_range\*0.1))

# Define the function y that we are going to plot

def y\_fn(x):

c=60-0.1\*x-0.5\*x\*\*3+0.05\*x\*\*4

return c

def y\_fn\_name():

"""

Helper for displaying the name of the fn in the plot

Returns the parameters for plotting the y function name,

used by approx\_expected\_value

:return: fname, x\_tpos, y\_tpos, year

"""

fname = r'$f(y) = 60-0.1\*y-0.5\*y\*\*3+0.05\*y\*\*4$' # latex format of fn name

x\_tpos = -3 # x position for plotting text of name

y\_tpos = 100 # y position for plotting text of name

return fname, x\_tpos, y\_tpos

# Plot the function!

plot\_fn(y\_fn, -4, 10, y\_fn\_name)

# Now we'll approximate the area under the curve using sampling...

# Sample 5000 uniformly random values in [-4..10]

numpy.random.seed(seed=1)

ys = numpy.random.uniform(low=-4.0, high=10.0, size=5000)

# compute the expectation of y, where y is the function that squares its input

ey2 = numpy.mean(60-0.1\*ys-0.5\*ys\*\*3+0.05\*ys\*\*4)

print('\nSample-based approximation for 5000 samples: {:f}'.format(ey2))

# Store the evolution of the approximation, every 100 samples

sample\_sizes = numpy.arange(100, ys.shape[0], 100)

ey2\_evol = numpy.zeros((sample\_sizes.shape[0])) # storage for the evolving estimate...

# the following computes the mean of the sequence up to i, as i iterates

# through the sequence, storing the mean in ey2\_evol:

for i in range(sample\_sizes.shape[0]):

ey2\_evol[i] = numpy.mean(60-0.1\*ys[0:sample\_sizes[i]]-0.5\*ys[0:sample\_sizes[i]]\*\*3+0.05\*ys[0:sample\_sizes[i]]\*\*4)

print('\nSample-based approximation for {:4d} sample: {:f}'.format(sample\_sizes[i],ey2\_evol[i]))

# Create plot of evolution of the approximation

plt.figure()

# plot the curve of the estimation of the expected value of f(x)=y^2

plt.plot(sample\_sizes, ey2\_evol)

# The true, analytic result of the expected value of f(y)=y^2 where y ~ U(0,1): $\frac{1}{3}$

# plot the analytic expected result as a red line:

plt.plot(numpy.array([sample\_sizes[0], sample\_sizes[-1]]),

numpy.array([44.860, 44.860]), color='r')

plt.xlabel('Sample size')

plt.ylabel('Approximation of expectation')

plt.title('Approximation of expectation of $f(y) = 60-0.1\*x-0.5\*x\*\*3+0.05\*x\*\*4$')

plt.pause(.1) # required on some systems so that rendering can happen

plt.show() # keeps the plot open

And the output of this script is:

Sample-based approximation for 5000 samples: 44.734913

Sample-based approximation for 100 sample: 45.936236

Sample-based approximation for 200 sample: 46.197343

Sample-based approximation for 300 sample: 44.705475

Sample-based approximation for 400 sample: 44.270098

Sample-based approximation for 500 sample: 44.478565

Sample-based approximation for 600 sample: 44.801953

Sample-based approximation for 700 sample: 44.545896

Sample-based approximation for 800 sample: 44.349374

Sample-based approximation for 900 sample: 44.459208

Sample-based approximation for 1000 sample: 44.869757

Sample-based approximation for 1100 sample: 44.739597

Sample-based approximation for 1200 sample: 44.523223

Sample-based approximation for 1300 sample: 44.598721

Sample-based approximation for 1400 sample: 44.568219

Sample-based approximation for 1500 sample: 44.788658

Sample-based approximation for 1600 sample: 44.703174

Sample-based approximation for 1700 sample: 44.527192

Sample-based approximation for 1800 sample: 44.416092

Sample-based approximation for 1900 sample: 44.323732

Sample-based approximation for 2000 sample: 44.279820

Sample-based approximation for 2100 sample: 44.404413

Sample-based approximation for 2200 sample: 44.492738

Sample-based approximation for 2300 sample: 44.349648

Sample-based approximation for 2400 sample: 44.491154

Sample-based approximation for 2500 sample: 44.576945

Sample-based approximation for 2600 sample: 44.719268

Sample-based approximation for 2700 sample: 44.674108

Sample-based approximation for 2800 sample: 44.634310

Sample-based approximation for 2900 sample: 44.700456

Sample-based approximation for 3000 sample: 44.777454

Sample-based approximation for 3100 sample: 44.710476

Sample-based approximation for 3200 sample: 44.741601

Sample-based approximation for 3300 sample: 44.756282

Sample-based approximation for 3400 sample: 44.714038

Sample-based approximation for 3500 sample: 44.661867

Sample-based approximation for 3600 sample: 44.721462

Sample-based approximation for 3700 sample: 44.790040

Sample-based approximation for 3800 sample: 44.776965

Sample-based approximation for 3900 sample: 44.560913

Sample-based approximation for 4000 sample: 44.581013

Sample-based approximation for 4100 sample: 44.533646

Sample-based approximation for 4200 sample: 44.640211

Sample-based approximation for 4300 sample: 44.652262

Sample-based approximation for 4400 sample: 44.748157

Sample-based approximation for 4500 sample: 44.928357

Sample-based approximation for 4600 sample: 44.913713

Sample-based approximation for 4700 sample: 44.879668

Sample-based approximation for 4800 sample: 44.770774

Sample-based approximation for 4900 sample: 44.791253

The figures below show the area under function and comparison between true answer and influence of number of samples on approximate sampling method, respectively. As we can see, it is obvious the more sampling we use, the more precise is the answer.

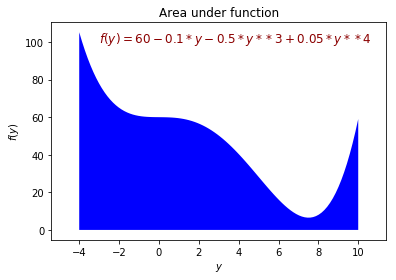


Figure 1area under function

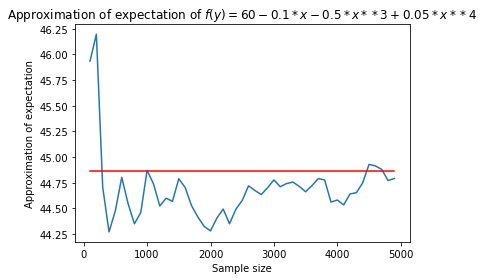


Figure 2comparison between true answer and influence of number of samples on approximate sampling method

# Problem 3

As it shown,

# Problem 4

In x1, x2, …, xn each one can be either 0 or 1. Furthermore, the order is important, that is 1,0, …,0 is different from 0,1, …,0. In other words, the former is permutation and the latter is combination. The Bernoulli distribution is a discrete distribution having two possible outcomes; however, the Binomial distribution gives the discrete probability distribution of obtaining exactly n successes out of N. To put it differently, Bernoulli is just for 1 event; nevertheless, Binomial is a couple of events.

# Problem 5

Because two functions are not complete, calculate prediction variance and calculate cov w, I added these lines to the script.

def calculate\_prediction\_variance(x\_new, X, w, t):

"""

Calculates the variance for the prediction at x\_new

:param X: Design matrix: matrix of N observations

:param w: vector of parameters

:param t: vector of N target responses

:param x\_new: new observation

:return: the predictive variance around x\_new

"""

### Insert your code here to calculate the predictive variance ###

Size1=t.shape

Sigma2=(numpy.transpose(t)@t-numpy.transpose(t)@X@w)/Size1

predictive\_variance = Sigma2\*numpy.transpose(x\_new)@numpy.linalg.inv((numpy.transpose(X)@X))@x\_new

return predictive\_variance

def calculate\_cov\_w(X, w, t):

"""

Calculates the covariance of w

:param X: Design matrix: matrix of N observations

:param w: vector of parameters

:param t: vector of N target responses

:return: the matrix covariance of w

"""

### Insert your code here to calculate the estimated covariance of w ###

Sigma2=(numpy.transpose(t)@t-numpy.transpose(t)@X@w)/t.shape[0]

covw = Sigma2\*numpy.linalg.inv((numpy.transpose(X)@X))

print('CovW={}'.format(covw))

return covw

After completing two incomplete scripts, calculate prediction variance and calculate cov w, we can run the script and it generates the following figures. Figure 3 illustrates all the data that are used to calculate w. Actually the spots which their x dimension were between -2 and 2 are removed, therefore, their absence in the mentioned figure is obvious. It should be mentioned these data are generated by using a polynomial relationship between x and y and adding noise to y.

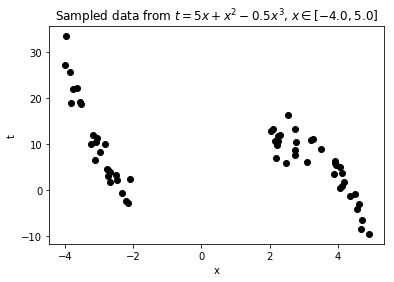


Figure 3 Data are used to calculate w

Figure 4 through Figure 7 show plot of predicted variance for model with polynomial order 1, 3, 5, and 9, respectively. As expected, because we generated that data using a 3rd-degree polynomial, the predicted variance for model in Figure 5 is the lowest. Furthermore more, in the range of x between -2 and 2 is higher than out of this span. This is because of the fact that we eliminated the data in this range, consequently, our model is less accurate in that range.

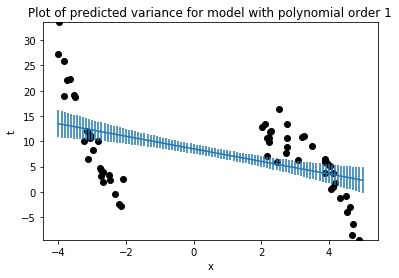


Figure 4 Plot of predicted variance for model with polynomial order 1

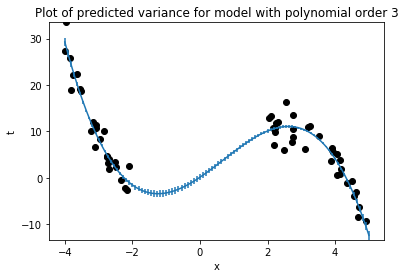


Figure 5Plot of predicted variance for model with polynomial order 3

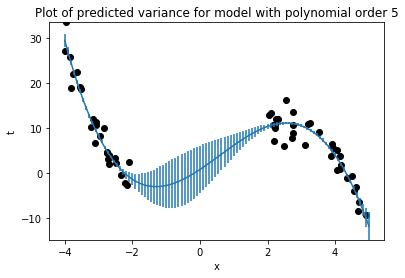


Figure 6Plot of predicted variance for model with polynomial order 5

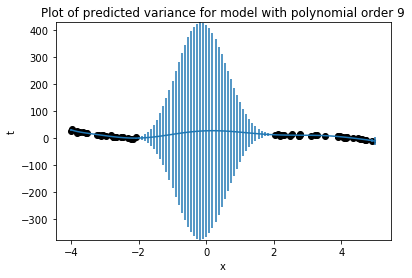


Figure 7 Plot of predicted variance for model with polynomial order 9

Figure 8 through Figure 11 provide 20 possible functions of order 1, 3, 5, 9, respectively, that pass through the scatter points.

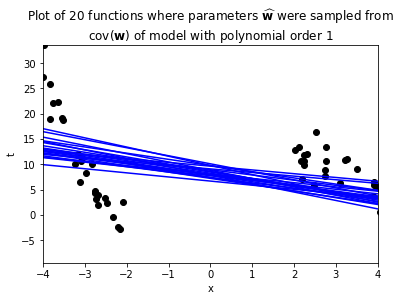


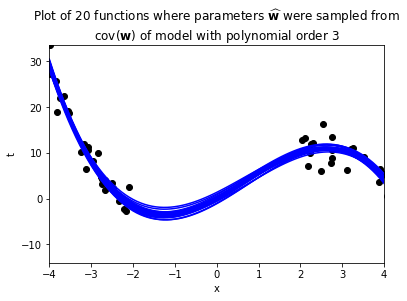
Figure 8 20 different polynomials order 1 using different w

Figure 920 different polynomials order 3 using different w

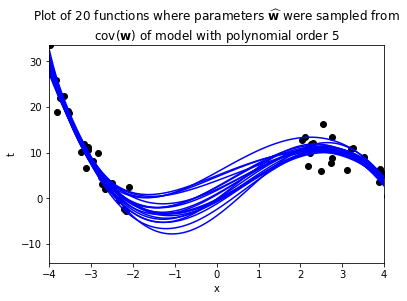


Figure 10 20 different polynomials order 5 using different w

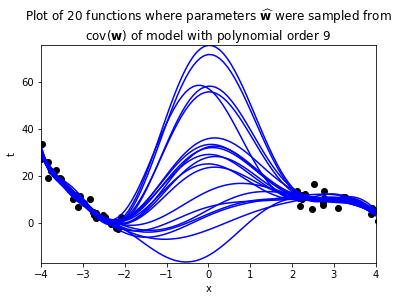


Figure 11 20 different polynomials order 9 using different w

As it can be seen, 20 various functions of polynomial order 3 are more similar and there is less scattering among them. This is the whole code used to generate these figures:

# Extension of predictive\_variance\_example.py

# Port of predictive\_variance\_example.m

# From A First Course in Machine Learning, Chapter 2.

# Simon Rogers, 01/11/11 [simon.rogers@glasgow.ac.uk]

# Predictive variance example

import numpy

import matplotlib.pyplot as plt

# Turn off annoying matplotlib warning

import warnings

warnings.filterwarnings("ignore", ".\*GUI is implemented.\*")

# --------------------------------------------------------------------------

# Globals: edit these as needed

# --------------------------------------------------------------------------

# set to True in order to automatically save the generated plots

SAVE\_FIGURES = False

# change this to where you'd like the figures saved

# (relative to your python current working directory)

FIGURE\_PATH = '../figs/'

# --------------------------------------------------------------------------

# You must complete the implementation of the following two functions

# --------------------------------------------------------------------------

def calculate\_prediction\_variance(x\_new, X, w, t):

"""

Calculates the variance for the prediction at x\_new

:param X: Design matrix: matrix of N observations

:param w: vector of parameters

:param t: vector of N target responses

:param x\_new: new observation

:return: the predictive variance around x\_new

"""

### Insert your code here to calculate the predictive variance ###

Size1=t.shape

Sigma2=(numpy.transpose(t)@t-numpy.transpose(t)@X@w)/Size1

predictive\_variance = Sigma2\*numpy.transpose(x\_new)@numpy.linalg.inv((numpy.transpose(X)@X))@x\_new

return predictive\_variance

def calculate\_cov\_w(X, w, t):

"""

Calculates the covariance of w

:param X: Design matrix: matrix of N observations

:param w: vector of parameters

:param t: vector of N target responses

:return: the matrix covariance of w

"""

### Insert your code here to calculate the estimated covariance of w ###

Sigma2=(numpy.transpose(t)@t-numpy.transpose(t)@X@w)/t.shape[0]

covw = Sigma2\*numpy.linalg.inv((numpy.transpose(X)@X))

print('CovW={}'.format(covw))

return covw

# --------------------------------------------------------------------------

# Part 5a -- Generate and Plot the data

# There is nothing you need to implement here

# --------------------------------------------------------------------------

def true\_function(x):

"""$t = 5x+x^2-0.5x^3$"""

return (5 \* x) + x\*\*2 - (0.5 \* x\*\*3)

def sample\_from\_function(N=100, noise\_var=1000, xmin=-5., xmax=5.):

""" Sample data from the true function.

N: Number of samples

Returns a noisy sample t\_sample from the function

and the true function t. """

x = numpy.random.uniform(xmin, xmax, N)

t = true\_function(x)

# add standard normal noise using numpy.random.randn

# (standard normal is a Gaussian N(0, 1.0) (i.e., mean 0, variance 1),

# so multiplying by numpy.sqrt(noise\_var) make it N(0,standard\_deviation))

t = t + numpy.random.randn(x.shape[0])\*numpy.sqrt(noise\_var)

return x, t

numpy.random.seed(seed=1)

xmin = -4.

xmax = 5.

noise\_var = 6

# sample 100 points from function

x, t = sample\_from\_function(100, noise\_var, xmin, xmax)

print(t.shape[0])

# Chop out some x data:

xmin\_remove = -2 #

xmax\_remove = 2 #

# the following line expresses a boolean function over the values in x;

# this produces a list of the indices of list x for which the test

# was not met; these indices are then deleted from x and t.

pos = ((x >= xmin\_remove) & (x <= xmax\_remove)).nonzero()

x = numpy.delete(x, pos, 0)

t = numpy.delete(t, pos, 0)

# Plot just the sampled data

plt.figure(0)

plt.scatter(numpy.asarray(x), numpy.asarray(t), color='k', edgecolor='k')

plt.xlabel('x')

plt.ylabel('t')

plt.title('Sampled data from {0}, $x \in [{1},{2}]$'

.format(true\_function.\_\_doc\_\_, xmin, xmax))

print(true\_function.\_\_doc\_\_)

plt.pause(.1) # required on some systems so that rendering can happen

print('mmmmmmmmmmmmmmmmmmmmmmmm')

if SAVE\_FIGURES:

plt.savefig(FIGURE\_PATH + 'data.pdf', fmt='pdf')

# Fit models of various orders

orders = [1, 3, 5, 9]

# Make a set of 100 evenly-spaced x values between xmin and xmax

testx = numpy.linspace(xmin, xmax, 100)

# --------------------------------------------------------------------------

# Part 5b -- Error bar plots

# There is nothing you need to implement here;

# However, you need to complete the implementation of function

# calculate\_prediction\_variance, above

# --------------------------------------------------------------------------

# Generate plots of predicted variance (error bars) for various model orders

for i in orders:

# create input representation for given model polynomial order

X = numpy.zeros(shape=(x.shape[0], i + 1))

testX = numpy.zeros(shape=(testx.shape[0], i + 1))

for k in range(i + 1):

X[:, k] = numpy.power(x, k)

testX[:, k] = numpy.power(testx, k)

# fit model parameters

w = numpy.dot(numpy.linalg.inv(numpy.dot(X.T, X)), numpy.dot(X.T, t))

# calculate predictions

prediction\_t = numpy.dot(testX, w)

# get variance, each x\_new at a time

prediction\_t\_variance = numpy.zeros(testX.shape[0])

for j in range(testX.shape[0]):

x\_new = testX[j, :] # get the x\_new observation from row i of testX

var = calculate\_prediction\_variance(x\_new, X, w, t) # calculate prediction variance

prediction\_t\_variance[j] = var

# Plot the data and predictions

plt.figure()

plt.scatter(x, t, color='k', edgecolor='k')

plt.xlabel('x')

plt.ylabel('t')

plt.errorbar(testx, prediction\_t, prediction\_t\_variance)

# find ylim plot bounds automagically...

min\_model = min(prediction\_t - prediction\_t\_variance)

max\_model = max(prediction\_t + prediction\_t\_variance)

min\_testvar = min(min(t), min\_model)

max\_testvar = max(max(t), max\_model)

plt.ylim(min\_testvar, max\_testvar)

ti = 'Plot of predicted variance for model with polynomial order {:g}'.format(i)

plt.title(ti)

plt.pause(.1) # required on some systems so that rendering can happen

if SAVE\_FIGURES:

filename = 'error-{0}.pdf'.format(i)

plt.savefig(FIGURE\_PATH + filename, fmt='pdf')

# --------------------------------------------------------------------------

# Part 5c -- Plot functions by sampling from cov(\hat{w})

# There is nothing you need to implement here.

# However, you need to complete the implementation of function

# calculate\_cov\_w, above

# --------------------------------------------------------------------------

# Generate plots of functions whose parameters are sampled based on cov(\hat{w})

num\_function\_samples = 20

for i in orders:

# create input representation for given model polynomial order

X = numpy.zeros(shape=(x.shape[0], i+1))

testX = numpy.zeros(shape=(testx.shape[0], i+1))

for k in range(i + 1):

X[:, k] = numpy.power(x, k)

testX[:, k] = numpy.power(testx, k)

# fit model parameters

w = numpy.dot(numpy.linalg.inv(numpy.dot(X.T, X)), numpy.dot(X.T, t))

# Sample functions with parameters w sampled from a Gaussian with

# $\mu = \hat{\mathbf{w}}$

# $\Sigma = cov(w)$

# determine cov(w)

covw = calculate\_cov\_w(X, w, t) # calculate the covariance of w

# The following samples num\_function\_samples of w from

# multivariate Gaussian (normal) with covaraince covw

wsamp = numpy.random.multivariate\_normal(w, covw, num\_function\_samples)

# Calculate means for each function

prediction\_t = numpy.dot(testX, wsamp.T)

# Plot the data and functions

plt.figure()

plt.scatter(x, t, color='k', edgecolor='k')

plt.xlabel('x')

plt.ylabel('t')

plt.plot(testx, prediction\_t, color='b')

# find reasonable ylim bounds

plt.xlim(xmin\_remove-2, xmax\_remove+2) # (-2,4) # (-3, 3)

min\_model = min(prediction\_t.flatten())

max\_model = max(prediction\_t.flatten())

min\_testvar = min(min(t), min\_model)

max\_testvar = max(max(t), max\_model)

plt.ylim(min\_testvar, max\_testvar) # (-400,400)

ti = 'Plot of {0} functions where parameters '\

.format(num\_function\_samples, i) + \

r'$\widehat{\bf w}$ were sampled from' + '\n' + r'cov($\bf w$)' + \

' of model with polynomial order {1}' \

.format(num\_function\_samples, i)

plt.title(ti)

plt.pause(.1) # required on some systems so that rendering can happen

if SAVE\_FIGURES:

filename = 'sampled-fns-{0}.pdf'.format(i)

plt.savefig(FIGURE\_PATH + filename, fmt='pdf')

plt.show()

# Problem 6

In this problem, first we generate 25 sample points, and then use the function y=(5 \* x) + x\*\*2 - (0.5 \* x\*\*3) to calculate y. Then, we use different polynomials (1, 3, 5, and 9) to pass curves through the mentioned points. Afterward, we added polluted y with 20 different noise and repeat the previous step for each of them. Figure 12 through Figure 15 show these polynomials. As it can be seen, the figure pertinent to polynomials of order 3 have the least scattering. The script is on the following. The blue lines show polynomial interpolation when we are using polluted data, red line is the theoretical line, and the green one is the interpolation when we are using unpolluted data

# -\*- coding: utf-8 -\*-

"""

Created on Mon Oct 2 14:30:26 2017

@author: smvaz

"""

import numpy

import matplotlib.pyplot as plt

# Turn off annoying matplotlib warning

import warnings

warnings.filterwarnings("ignore", ".\*GUI is implemented.\*")

# define required procedure

#1

def true\_function(x):

"""$t = 5x+x^2-0.5x^3$"""

return (5 \* x) + x\*\*2 - (0.5 \* x\*\*3)

#2

def sample\_from\_function(N=100, noise\_var=1000, xmin=-5., xmax=5.):

""" Sample data from the true function.

N: Number of samples

Returns a noisy sample t\_sample from the function

and the true function t. """

x = numpy.random.uniform(xmin, xmax, N)

t = true\_function(x)

# add standard normal noise using numpy.random.randn

# (standard normal is a Gaussian N(0, 1.0) (i.e., mean 0, variance 1),

# so multiplying by numpy.sqrt(noise\_var) make it N(0,standard\_deviation))

t = t + numpy.random.randn(x.shape[0])\*numpy.sqrt(noise\_var)

return x, t

#3

#This one is for generating data without noise

def sample\_from\_function\_main(N=100, xmin=-5., xmax=5.):

""" Sample data from the true function.

N: Number of samples

Returns a noisy sample t\_sample from the function

and the true function t. """

x = numpy.random.uniform(xmin, xmax, N)

t = true\_function(x)

return x, t

#range of random numbers

xmin = -4.

xmax = 5.

#noise variance

noise\_var = 6

#creating data for test and drawing figures

testx = numpy.linspace(xmin, xmax, 100)

xmin\_remove=-2

xmax\_remove=2

#polynomial orders

orders = [1, 3, 5, 9]

#number of different curves we want to draw+ the main one

numberofcurves=20+1

#number of sample point we want to use to calculate w

numberofsamples=25

for i in orders:

#seed number

numpy.random.seed(seed=2)

w = numpy.zeros(shape=(i + 1,numberofcurves))

prediction\_t= numpy.zeros(shape=(testx.shape[0],numberofcurves))

for j in range(0,numberofcurves):

if j==0:

#without noise

x, t = sample\_from\_function\_main(numberofsamples, xmin, xmax)

#with noise

else:

x, t = sample\_from\_function(numberofsamples, noise\_var, xmin, xmax)

#print(x)

#print(t.shape[0])

#print('j=',j)

# create input representation for given model polynomial order

X = numpy.zeros(shape=(x.shape[0], i + 1))

testX = numpy.zeros(shape=(testx.shape[0], i + 1))

for k in range(i + 1):

X[:, k] = numpy.power(x, k)

testX[:, k] = numpy.power(testx, k)

# fit model parameters

w[:,j] = numpy.dot(numpy.linalg.inv(numpy.dot(X.T, X)), numpy.dot(X.T, t))

#print('w=', w)

# calculate predictions

prediction\_t[:,j]= numpy.dot(testX, w[:,j])

#print('prediction\_t=', prediction\_t)

# Plot the data and functions

#plt.plot(testx.T, prediction\_t[:,j], color='k', edgecolor='k')

plt.plot(testx, prediction\_t, color='b')

plt.plot(testx, prediction\_t[:,0], color='r',linewidth=3)

plt.xlabel('x')

plt.ylabel('t')

# find reasonable ylim bounds

plt.xlim(xmin\_remove-2, xmax\_remove+2) # (-2,4) # (-3, 3)

min\_model = min(prediction\_t.flatten())

max\_model = max(prediction\_t.flatten())

min\_testvar = min(min(t), min\_model)

max\_testvar = max(max(t), max\_model)

plt.ylim(min\_testvar, max\_testvar) # (-400,400)

ti = 'Plot of {0} functions '\

.format(j) + \

' of model with polynomial order {1}' \

.format(num\_function\_samples, i)

plt.title(ti)

plt.pause(.1) # required on some systems so that rendering can happen

plt.show()

plt.show()

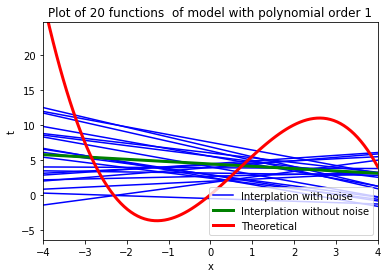


Figure 12 Different polynomials of order 1 generated by different sample points and

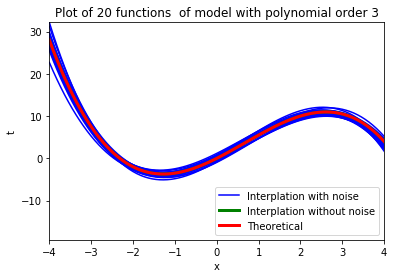


Figure 13Different polynomials of order 3 generated by different sample points and

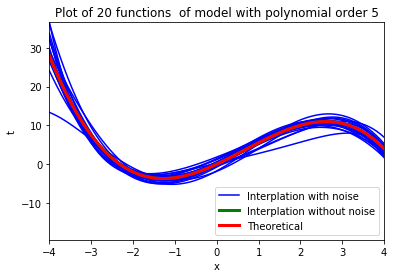


Figure 14Different polynomials of order 5 generated by different sample points and

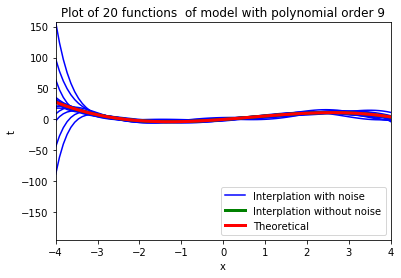


Figure 15Different polynomials of order 9 generated by different sample points and

# Problem 7

The Fisher information is calculated as follow: