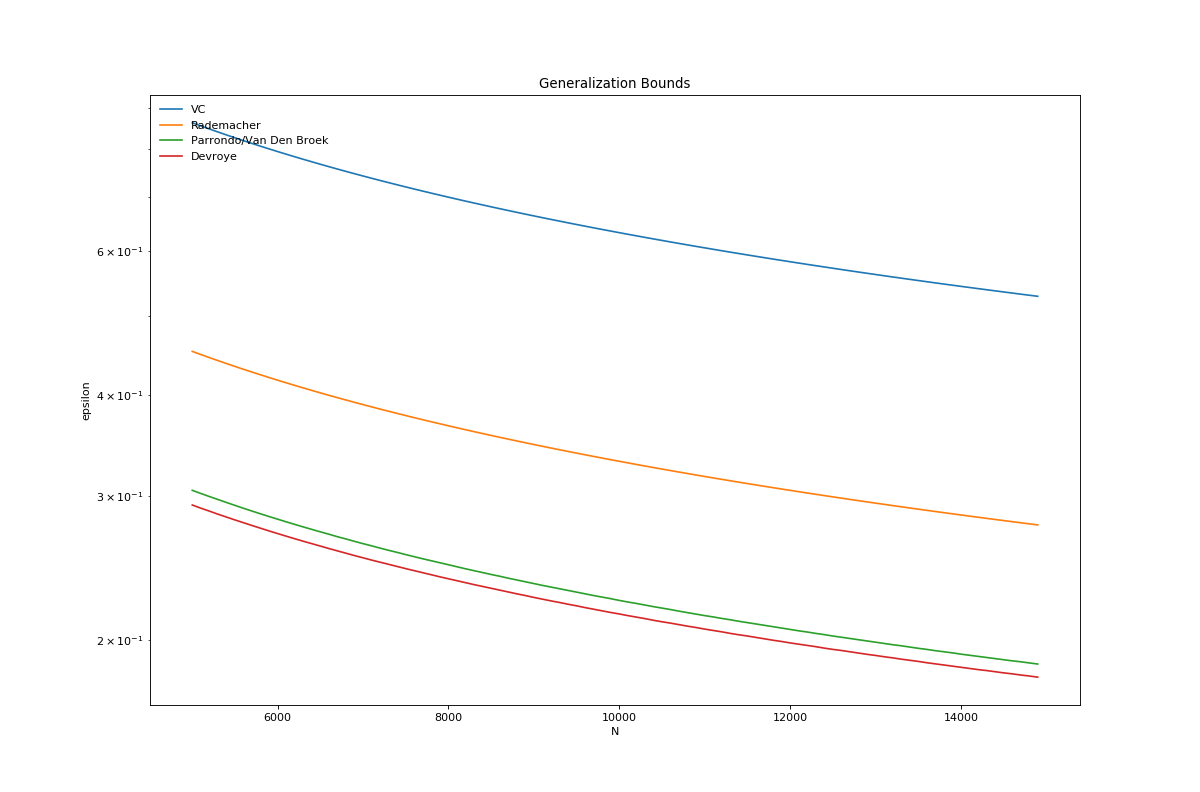
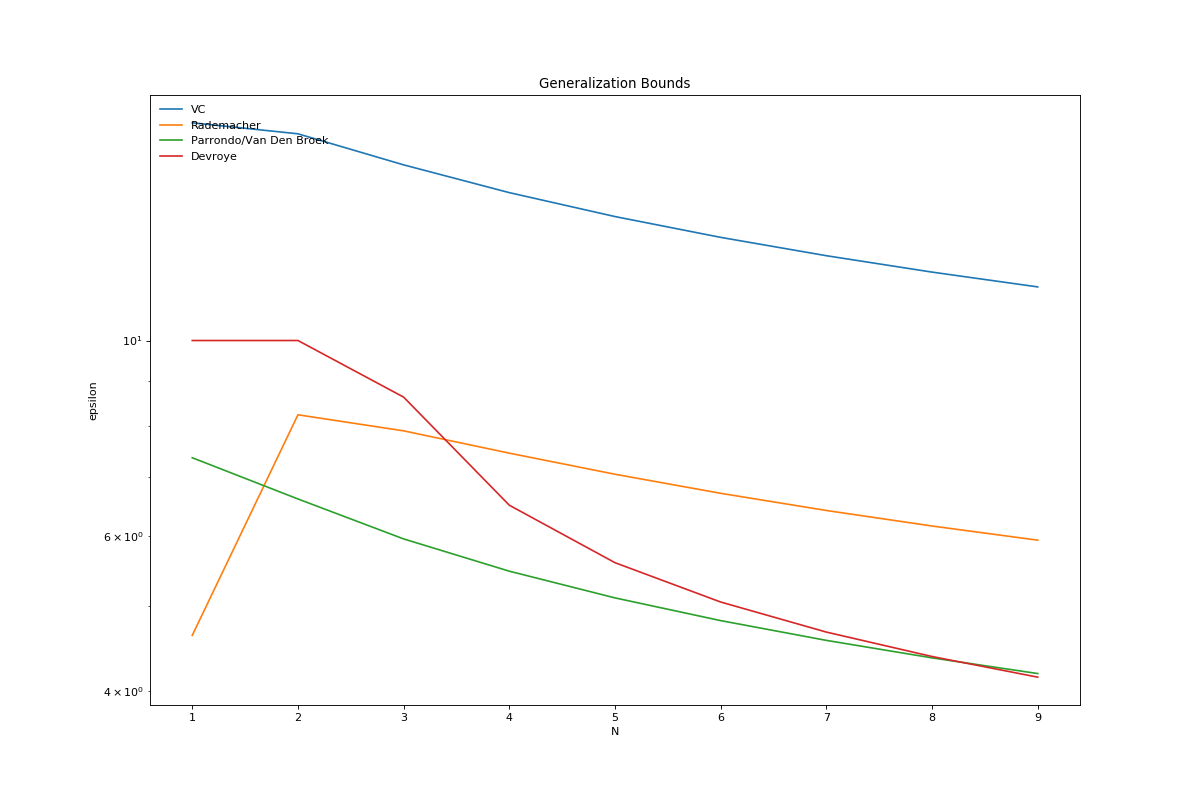
**Q2 and Q3 are based on the following codes:**





import numpy as np

import matplotlib.pyplot as plt

import math

def vc\_eps(delta, N, dvc):

return math.sqrt((8.0/N)\*((dvc\*math.log(2.0\*N))-math.log(delta/4.0)))

def rademacher\_eps(delta, N, dvc):

cur\_eps = math.sqrt((2.0/N)\*(math.log(2.0\*N)+(dvc\*math.log(N)))) + math.sqrt((2.0/N) \* math.log(1.0/delta)) + (1.0/N)

return cur\_eps

eps\_range = np.arange(0.0,10, 0.0001)

def parrondo\_eps(delta, N, dvc):

last\_true = False

last\_eps = eps\_range[0]

for cur\_eps in eps\_range:

cur\_rightside = (1.0/N) \* ((2.0\*cur\_eps) + math.log(6.0/delta) + (dvc\*math.log(2.0\*N)))

cur\_rightside = math.sqrt(cur\_rightside)

cur\_true = cur\_eps <= cur\_rightside

if cur\_true == False and last\_true == True:

break

elif cur\_true == True:

last\_true = True

last\_eps = cur\_eps

return last\_eps

def devroye\_eps(delta, N, dvc):

last\_true = False

last\_eps = eps\_range[0]

for cur\_eps in eps\_range:

cur\_rightside = (1.0/(2.0\*N))\*(((4.0\*cur\_eps)\*(1.0+cur\_eps)) + math.log(4.0/delta) + ((2.0\*dvc)\*math.log(N)))

cur\_rightside = math.sqrt(cur\_rightside)

cur\_true = cur\_eps <= cur\_rightside

if cur\_true == False and last\_true == True:

break

elif cur\_true == True:

last\_true = True

last\_eps = cur\_eps

return last\_eps

def main():

vc\_func = np.vectorize(vc\_eps)

rademacher\_func = np.vectorize(rademacher\_eps)

parrondo\_func = np.vectorize(parrondo\_eps)

devroye\_func = np.vectorize(devroye\_eps)

prob2 = {}

prob2["N"] = np.arange(5000,15000, 100)

prob2["vc"] = vc\_func(0.05, prob2["N"], 50)

prob2["rad"] = rademacher\_func(0.05, prob2["N"], 50)

prob2["par"] = parrondo\_func(0.05, prob2["N"], 50)

prob2["dev"] = devroye\_func(0.05,prob2["N"],50)

prob2["plot"] = plt.figure(figsize=(15,10), dpi=80)

prob2["ax"] = prob2["plot"].add\_subplot(111)

prob2["ax"].set\_title("Generalization Bounds")

prob2["ax"].set\_xlabel("N")

prob2["ax"].set\_ylabel("epsilon")

prob2["ax"].set\_yscale("log", basey=10)

prob2["ax"].plot(prob2["N"], prob2["vc"], label="VC")

prob2["ax"].plot(prob2["N"], prob2["rad"], label="Rademacher")

prob2["ax"].plot(prob2["N"], prob2["par"], label="Parrondo/Van Den Broek")

prob2["ax"].plot(prob2["N"], prob2["dev"], label = "Devroye")

prob2["ax"].legend(loc='upper left', frameon=False)

plt.show()

if \_\_name\_\_== "\_\_main\_\_":

main()

**Q4 and Q7 are based on the following codes:**

f = @(x) sin(pi\*x);

N = 10000; % Number of runs

% Average a

a = 0;

for i=1:N

x = unifrnd([-1 -1], [1 1]);

a\_hat = (f(x(1))\*x(1) + f(x(2))\*x(2))/(x(1)^2 + x(2)^2);

a = a + a\_hat;

end

a = a/N;

% Bias

b = 0;

for i=1:N

x = unifrnd(-1, 1);

b = b + (a\*x - f(x))^2;

end

b = b/N;

% Variance

v = 0;

N = 150;

for i=1:N

x = unifrnd([-1 -1], [1 1]);

a\_hat = (f(x(1))\*x(1) + f(x(2))\*x(2))/(x(1)^2 + x(2)^2);

t = 0;

for j=1:N

x = unifrnd(-1, 1);

t = t + (a\_hat\*x - a\*x)^2;

end

v = v + t/N;

end

v = v/N;

%%

% Target function

f = @(x) sin(pi\*x);

% Average a

a = 0;

% Number of runs

N = 20000;

% All of the given hypothesis are polynomials of degree 2

% We provide different learning algorithms for each type

fprintf('expected E\_out A: %f\n', expectedOutOfSampleError(@learnA, f, N));

fprintf('expected E\_out B: %f\n', expectedOutOfSampleError(@learnB, f, N));

fprintf('expected E\_out C: %f\n', expectedOutOfSampleError(@learnC, f, N));

fprintf('expected E\_out D: %f\n', expectedOutOfSampleError(@learnD, f, N));

fprintf('expected E\_out E: %f\n', expectedOutOfSampleError(@learnE, f, N));

function e = expectedOutOfSampleError(learn, f, N)

% Approximates the expected out-of-sample error

%

% PARAMETERS

% learn - [function] The training function

% f - [function] The target function

% N - [1x1] Number of runs

%

% RETURN

% e - [1x1] Expected out-of-sample error

a = [0 0 0];

% Determine the average hypothesis

for i=1:N

% Sample two points

x = unifrnd([-1 -1], [1 1]);

% Determine a

a\_hat = learn(x, f);

a = a + a\_hat;

end

a = a/N;

% Compute the variance

v = 0;

for i=1:floor(sqrt(N))

% Sample two points

x = unifrnd([-1 -1], [1 1]);

% Determine a

w = learn(x, f);

t = 0;

for j=1:floor(sqrt(N))

x = unifrnd(-1, 1);

bx = [x^2; x; 1];

t = t + (a\*bx - w\*bx)^2;

end

v = v + t/floor(sqrt(N));

end

v = v/floor(sqrt(N));

% Compute the squared bias

% Compute the bias

b = 0;

for i=1:N

x = unifrnd(-1, 1);

bx = [x^2; x; 1];

b = b + (a\*bx - f(x))^2;

end

b = b/N;

e = b+v;

end

function w = learnA(x, f)

% Trains the parameters for the hypothesis h(x) = c

%

% PARAMETERS

% x - [2x1] The training points

% f - [function] The target function

%

% RETURN

% w - [1x3] The weight vector

A = [1;1];

b = [f(x(1)); f(x(2))];

w\_hat = A\b;

w = [0, 0, w\_hat(1)];

end

function w = learnB(x, f)

% Trains the parameters for the hypothesis h(x) = bx

%

% PARAMETERS

% x - [2x1] The training points

% f - [function] The target function

%

% RETURN

% w - [1x3] The weight vector

A = [x(1);x(2)];

b = [f(x(1)); f(x(2))];

w\_hat = A\b;

w = [0, w\_hat(1), 0];

end

function w = learnC(x, f)

% Trains the parameters for the hypothesis h(x) = bx + c

%

% PARAMETERS

% x - [2x1] The training points

% f - [function] The target function

%

% RETURN

% w - [1x3] The weight vector

A = [x(1), 1;x(2), 1];

b = [f(x(1)); f(x(2))];

w\_hat = A\b;

w = [0, w\_hat(1), w\_hat(2)];

end

function w = learnD(x, f)

% Trains the parameters for the hypothesis h(x) = ax^2

%

% PARAMETERS

% x - [2x1] The training points

% f - [function] The target function

%

% RETURN

% w - [1x3] The weight vector

A = [x(1)^2;x(2)^2];

b = [f(x(1)); f(x(2))];

w\_hat = A\b;

w = [w\_hat(1), 0, 0];

end

function w = learnE(x, f)

% Trains the parameters for the hypothesis h(x) = ax^2 + b

%

% PARAMETERS

% x - [2x1] The training points

% f - [function] The target function

%

% RETURN

% w - [1x3] The weight vector

A = [x(1)^2, 1;x(2)^2, 1];

b = [f(x(1)); f(x(2))];

w\_hat = A\b;

w = [w\_hat(1), 0, w\_hat(2)];

end