**1.Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.**

import csv

a = []

with open('enjoysport.csv', 'r') as csvfile:

for row in csv.reader(csvfile):

a.append(row)

print(a)

num\_attribute = len(a[0]) - 1

print("\n The initial hypothesis is : ")

hypothesis = ['0'] \* num\_attribute

print(hypothesis)

print("\n The total number of training instances are : ", len(a))

for i in range(0, len(a)):

if a[i][num\_attribute] == 'yes':

for j in range(0, num\_attribute):

if hypothesis[j] == '0' or hypothesis[j] == a[i][j]:

hypothesis[j] = a[i][j]

else:

hypothesis[j] = '?'

print("\n The hypothesis for the training instance {} is :\n".format(i + 1), hypothesis)

print("\n The Maximally specific hypothesis for the training instance is ")

print(hypothesis)

**2.For a given set of training data examples stored in a .CSV file, implement and**

**demonstrate the Candidate-Elimination algorithm to output a description of the set of all**

**hypotheses consistent with the training examples.**

import numpy as np

import pandas as pd

data = pd.read\_csv('enjoysport.csv')

concepts = np.array(data.iloc[:, 0:-1])

print(concepts)

target = np.array(data.iloc[:, -1])

print(target)

def learn(concepts, target):

specific\_h = concepts[0].copy()

print("initialization of specific\_h and general\_h")

print(specific\_h)

general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]

print(general\_h)

for i, h in enumerate(concepts):

print("For Loop Starts")

if target[i] == "yes":

print("If instance is Positive ")

for x in range(len(specific\_h)):

if h[x] != specific\_h[x]:

specific\_h[x] = '?'

general\_h[x][x] = '?'

if target[i] == "no":

print("If instance is Negative ")

for x in range(len(specific\_h)):

if h[x] != specific\_h[x]:

general\_h[x][x] = specific\_h[x]

else:

general\_h[x][x] = '?'

print(" steps of Candidate Elimination Algorithm", i + 1)

print(specific\_h)

print(general\_h)

print("\n")

print("\n")

indices = [i for i, val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]

for i in indices:

general\_h.remove(['?', '?', '?', '?', '?', '?'])

return specific\_h, general\_h

s\_final, g\_final = learn(concepts, target)

print("Final Specific\_h:", s\_final, sep="\n")

print("Final General\_h:", g\_final, sep="\n")

**3.Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.**

import math

import csv

def load\_csv(filename):

lines = csv.reader(open(filename, "r"));

dataset = list(lines)

headers = dataset.pop(0)

return dataset, headers

class Node:

def \_\_init\_\_(self, attribute):

self.attribute = attribute

self.children = []

self.answer = ""

def subtables(data, col, delete):

dic = {}

coldata = [row[col] for row in data]

attr = list(set(coldata))

counts = [0] \* len(attr)

r = len(data)

c = len(data[0])

for x in range(len(attr)):

for y in range(r):

if data[y][col] == attr[x]:

counts[x] += 1

for x in range(len(attr)):

dic[attr[x]] = [[0 for i in range(c)] for j in range(counts[x])]

pos = 0

for y in range(r):

if data[y][col] == attr[x]:

if delete:

del data[y][col]

dic[attr[x]][pos] = data[y]

pos += 1

return attr, dic

def entropy(S):

attr = list(set(S))

if len(attr) == 1:

return 0

counts = [0, 0]

for i in range(2):

counts[i] = sum([1 for x in S if attr[i] == x]) / (len(S) \* 1.0)

sums = 0

for cnt in counts:

sums += -1 \* cnt \* math.log(cnt, 2)

return sums

def compute\_gain(data, col):

attr, dic = subtables(data, col, delete=False)

total\_size = len(data)

entropies = [0] \* len(attr)

ratio = [0] \* len(attr)

total\_entropy = entropy([row[-1] for row in data])

for x in range(len(attr)):

ratio[x] = len(dic[attr[x]]) / (total\_size \* 1.0)

entropies[x] = entropy([row[-1] for row in dic[attr[x]]])

total\_entropy -= ratio[x] \* entropies[x]

return total\_entropy

def build\_tree(data, features):

lastcol = [row[-1] for row in data]

if (len(set(lastcol))) == 1:

node = Node("")

node.answer = lastcol[0]

return node

n = len(data[0]) - 1

gains = [0] \* n

for col in range(n):

gains[col] = compute\_gain(data, col)

split = gains.index(max(gains))

node = Node(features[split])

fea = features[:split] + features[split + 1:]

attr, dic = subtables(data, split, delete=True)

for x in range(len(attr)):

child = build\_tree(dic[attr[x]], fea)

node.children.append((attr[x], child))

return node

def print\_tree(node, level):

if node.answer != "":

print(" " \* level, node.answer)

return

print(" " \* level, node.attribute)

for value, n in node.children:

print(" " \* (level + 1), value)

print\_tree(n, level + 2)

def classify(node, x\_test, features):

if node.answer != "":

print(node.answer)

return

pos = features.index(node.attribute)

for value, n in node.children:

if x\_test[pos] == value:

classify(n, x\_test, features)

'''Main program'''

dataset, features = load\_csv("D:/ML\_Pracs/id3.csv")

node1 = build\_tree(dataset, features)

print("The decision tree for the dataset using ID3 algorithm is")

print\_tree(node1, 0)

testdata, features = load\_csv("D:/ML\_Pracs/id3\_test\_1.csv")

for xtest in testdata:

print("The test instance:", xtest)

print("The label for test instance:", end=" ")

classify(node1, xtest, features)

**4.Build an Artificial Neural Network by implementing the Back-propagation algorithm and test the same using appropriate data sets.**

import numpy as np  
  
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float) *# two inputs [sleep,study]*y = np.array(([92], [86], [89]), dtype=float) *# one output [Expected % in Exams]*X = X / np.amax(X, axis=0) *# maximum of X array longitudinally*y = y / 100  
  
*# Sigmoid Function*def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
  
*# Derivative of Sigmoid Function*def derivatives\_sigmoid(x):  
 return x \* (1 - x)  
  
*# Variable initialization*epoch = 5000 *# Setting training iterations*lr = 0.1 *# Setting learning rate*inputlayer\_neurons = 2 *# number of features in data set*hiddenlayer\_neurons = 3 *# number of hidden layers neurons*output\_neurons = 1 *# number of neurons at output layer  
# weight and bias initialization*wh = np.random.uniform(size=(inputlayer\_neurons, hiddenlayer\_neurons)) *# weight of the link from input node to hidden node*bh = np.random.uniform(size=(1, hiddenlayer\_neurons)) *# bias of the link from input node to hidden node*wout = np.random.uniform(size=(hiddenlayer\_neurons, output\_neurons)) *# weight of the link from hidden node to output node*bout = np.random.uniform(size=(1, output\_neurons)) *# bias of the link from hidden node to output node  
# draws a random range of numbers uniformly of dim x\*y*for i in range(epoch):  
 *# Forward Propogation* hinp1 = np.dot(X, wh)  
 hinp = hinp1 + bh  
 hlayer\_act = sigmoid(hinp)  
 outinp1 = np.dot(hlayer\_act, wout)  
 outinp = outinp1 + bout  
 output = sigmoid(outinp)  
 *# Backpropagation* EO = y - output  
 outgrad = derivatives\_sigmoid(output)  
 d\_output = EO \* outgrad  
 EH = d\_output.dot(wout.T)  
 *# how much hidden layer weights contributed to error* hiddengrad = derivatives\_sigmoid(hlayer\_act)  
 d\_hiddenlayer = EH \* hiddengrad  
*# dotproduct of nextlayererror and currentlayerop*wout += hlayer\_act.T.dot(d\_output) \* lr  
wh += X.T.dot(d\_hiddenlayer) \* lr  
print("Input: \n" + str(X))  
print("Actual Output: \n" + str(y))  
print("Predicted Output: \n", output)

**5.Write a pogram to implement the naïve Bayesian classifier for a sample training dataset stored as a .CSV file. Compute the accuracy of the classifier, considering few test datasets.**

import numpy as np  
  
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float) *# two inputs [sleep,study]*y = np.array(([92], [86], [89]), dtype=float) *# one output [Expected % in Exams]*X = X / np.amax(X, axis=0) *# maximum of X array longitudinally*y = y / 100  
  
*# Sigmoid Function*def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
  
*# Derivative of Sigmoid Function*def derivatives\_sigmoid(x):  
 return x \* (1 - x)  
  
*# Variable initialization*epoch = 5000 *# Setting training iterations*lr = 0.1 *# Setting learning rate*inputlayer\_neurons = 2 *# number of features in data set*hiddenlayer\_neurons = 3 *# number of hidden layers neurons*output\_neurons = 1 *# number of neurons at output layer  
# weight and bias initialization*wh = np.random.uniform(size=(inputlayer\_neurons, hiddenlayer\_neurons)) *# weight of the link from input node to hidden node*bh = np.random.uniform(size=(1, hiddenlayer\_neurons)) *# bias of the link from input node to hidden node*wout = np.random.uniform(size=(hiddenlayer\_neurons, output\_neurons)) *# weight of the link from hidden node to output node*bout = np.random.uniform(size=(1, output\_neurons)) *# bias of the link from hidden node to output node  
# draws a random range of numbers uniformly of dim x\*y*for i in range(epoch):  
 *# Forward Propogation* hinp1 = np.dot(X, wh)  
 hinp = hinp1 + bh  
 hlayer\_act = sigmoid(hinp)  
 outinp1 = np.dot(hlayer\_act, wout)  
 outinp = outinp1 + bout  
 output = sigmoid(outinp)  
 *# Backpropagation* EO = y - output  
 outgrad = derivatives\_sigmoid(output)  
 d\_output = EO \* outgrad  
 EH = d\_output.dot(wout.T)  
 *# how much hidden layer weights contributed to error* hiddengrad = derivatives\_sigmoid(hlayer\_act)  
 d\_hiddenlayer = EH \* hiddengrad  
*# dotproduct of nextlayererror and currentlayerop*wout += hlayer\_act.T.dot(d\_output) \* lr  
wh += X.T.dot(d\_hiddenlayer) \* lr  
print("Input: \n" + str(X))  
print("Actual Output: \n" + str(y))  
print("Predicted Output: \n", output)  
  
import csv  
import random  
import math  
  
def loadcsv(filename):  
 lines = csv.reader(open(filename, "r"));  
 dataset = list(lines)  
 for i in range(len(dataset)):  
 *# converting strings into numbers for processing* dataset[i] = [float(x) for x in dataset[i]]  
 return dataset  
  
def splitdataset(dataset, splitratio):  
 *# 67% training size* trainsize = int(len(dataset) \* splitratio);  
 trainset = []  
 copy = list(dataset);  
 while len(trainset) < trainsize:  
 *# generate indices for the dataset list randomly to pick ele for training data* index = random.randrange(len(copy));  
 trainset.append(copy.pop(index))  
 return [trainset, copy]  
  
def separatebyclass(dataset):  
 separated = {} *# dictionary of classes 1 and 0  
 # creates a dictionary of classes 1 and 0 where the values are  
 # the instances belonging to each class* for i in range(len(dataset)):  
 vector = dataset[i]  
 if (vector[-1] not in separated):  
 separated[vector[-1]] = []  
 separated[vector[-1]].append(vector)  
 return separated  
  
def mean(numbers):  
 return sum(numbers) / float(len(numbers))  
  
def stdev(numbers):  
 avg = mean(numbers)  
 variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)  
 return math.sqrt(variance)  
  
def summarize(dataset): *# creates a dictionary of classes* summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)];  
 del summaries[-1] *# excluding labels +ve or -ve* return summaries  
  
def summarizebyclass(dataset):  
 separated = separatebyclass(dataset);  
 *# print(separated)* summaries = {}  
 for classvalue, instances in separated.items():  
 *# for key,value in dic.items()  
 # summaries is a dic of tuples(mean,std) for each class value* summaries[classvalue] = summarize(instances) *# summarize is used to cal to mean and std* return summaries  
  
def calculateprobability(x, mean, stdev):  
 exponent = math.exp(-(math.pow(x - mean, 2) / (2 \* math.pow(stdev, 2))))  
 return (1 / (math.sqrt(2 \* math.pi) \* stdev)) \* exponent  
  
def calculateclassprobabilities(summaries, inputvector):  
 probabilities = {} *# probabilities contains the all prob of all class of test data* for classvalue, classsummaries in summaries.items(): *# class and attribute information as mean and sd* probabilities[classvalue] = 1  
 for i in range(len(classsummaries)):  
 mean, stdev = classsummaries[i] *# take mean and sd of every attribute for class 0 and 1 seperaely* x = inputvector[i] *# testvector's first attribute* probabilities[classvalue] \*= calculateprobability(x, mean, stdev); *# use normal dist* return probabilities  
  
def predict(summaries, inputvector): *# training and test data is passed* probabilities = calculateclassprobabilities(summaries, inputvector)  
 bestLabel, bestProb = None, -1  
 for classvalue, probability in probabilities.items(): *# assigns that class which has he highest prob* if bestLabel is None or probability > bestProb:  
 bestProb = probability  
 bestLabel = classvalue  
 return bestLabel  
  
def getpredictions(summaries, testset):  
 predictions = []  
 for i in range(len(testset)):  
 result = predict(summaries, testset[i])  
 predictions.append(result)  
 return predictions  
  
def getaccuracy(testset, predictions):  
 correct = 0  
 for i in range(len(testset)):  
 if testset[i][-1] == predictions[i]:  
 correct += 1  
 return (correct / float(len(testset))) \* 100.0  
  
def main():  
 filename = 'C:/Users/hp/Downloads/naivedata.csv'  
 splitratio = 0.67  
 dataset = loadcsv(filename);  
 trainingset, testset = splitdataset(dataset, splitratio)  
 print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset), len(trainingset), len(testset)))  
 *# prepare model* summaries = summarizebyclass(trainingset); *# print(summaries)  
 # test model* predictions = getpredictions(summaries, testset) *# find the predictions of test data with the training data* accuracy = getaccuracy(testset, predictions)  
 print('Accuracy of the classifier is : {0}%'.format(accuracy))  
  
main()

**6.Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set.**

import pandas as pd  
  
msg = pd.read\_csv('C:/Users/hp/Downloads/naivetext.csv',names=['message','label'])  
print('The dimensions of the dataset', msg.shape)  
msg['labelnum'] = msg.label.map({'pos': 1, 'neg': 0})  
X = msg.message  
y = msg.labelnum  
print(X)  
print(y)  
*# splitting the dataset into train and test data*from sklearn.model\_selection import train\_test\_split  
  
xtrain, xtest, ytrain, ytest = train\_test\_split(X, y)  
print('\n The total number of Training Data :', ytrain.shape)  
print('\n The total number of Test Data :', ytest.shape)  
*# output of the words or Tokens in the text documents*from sklearn.feature\_extraction.text import CountVectorizer  
  
count\_vect = CountVectorizer()  
xtrain\_dtm = count\_vect.fit\_transform(xtrain)  
xtest\_dtm = count\_vect.transform(xtest)  
print('\n The words or Tokens in the text documents \n')  
print(count\_vect.get\_feature\_names\_out())  
df = pd.DataFrame(xtrain\_dtm.toarray(), columns=count\_vect.get\_feature\_names\_out())  
*# Training Naive Bayes (NB) classifier on training data.*from sklearn.naive\_bayes import MultinomialNB  
  
clf = MultinomialNB().fit(xtrain\_dtm, ytrain)  
predicted = clf.predict(xtest\_dtm)  
*# printing accuracy, Confusion matrix, Precision and Recall*from sklearn import metrics  
  
print('\n Accuracy of the classifier is', metrics.accuracy\_score(ytest, predicted))  
print('\n Confusion matrix')  
print(metrics.confusion\_matrix(ytest, predicted))  
print('\n The value of Precision', metrics.precision\_score(ytest, predicted))  
print('\n The value of Recall', metrics.recall\_score(ytest, predicted))

**7.Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set. You can use Java/Python ML library classes/API.**

import numpy as np

import pandas as pd

import csv

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianNetwork

from pgmpy.inference import VariableElimination

heartDisease = pd.read\_csv('D:/ML\_Pracs/heart.csv')

heartDisease = heartDisease.replace('?',np.nan)

print('Sample instances from the dataset are given below')

print(heartDisease.head())

print('\n Attributes and datatypes')

print(heartDisease.dtypes)

model = BayesianNetwork([('age','heartdisease'),('sex','heartdisease'),('exang','heartdisease'),('cp','heartdisease'),('heartdisease','restecg'),('heartdisease','chol')])

print('\nLearning CPD using Maximum likelihood estimators')

model.fit(heartDisease,estimator=MaximumLikelihoodEstimator)

print('\n Inferencing with Bayesian Network:')

HeartDiseasetest\_infer = VariableElimination(model)

print('\n 1. Probability of HeartDisease given evidence= restecg')

q1=HeartDiseasetest\_infer.query(variables=['heartdisease'],evidence={'restecg':1})

print(q1)

print('\n 2. Probability of HeartDisease given evidence= cp ')

q2=HeartDiseasetest\_infer.query(variables=['heartdisease'],evidence={'cp':2})

print(q2)

**8.Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program**.

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.cluster import KMeans

import sklearn.metrics as sm

import pandas as pd

import numpy as np

iris = datasets.load\_iris()

X = pd.DataFrame(iris.data)

X.columns = ['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width']

y = pd.DataFrame(iris.target)

y.columns = ['Targets']

model = KMeans(n\_clusters=3)

model.fit(X)

plt.figure(figsize=(14,7))

colormap = np.array(['red', 'lime', 'black'])

# Plot the Original Classifications

plt.subplot(1, 2, 1)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y.Targets], s=40)

plt.title('Real Classification')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

# Plot the Models Classifications

plt.subplot(1, 2, 2)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[model.labels\_], s=40)

plt.title('K Mean Classification')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

print('The accuracy score of K-Mean: ',sm.accuracy\_score(y, model.labels\_))

print('The Confusion matrix of K-Mean: ',sm.confusion\_matrix(y, model.labels\_))

from sklearn import preprocessing

scaler = preprocessing.StandardScaler()

scaler.fit(X)

xsa = scaler.transform(X)

xs = pd.DataFrame(xsa, columns = X.columns)

#xs.sample(5)

from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n\_components=3)

gmm.fit(xs)

y\_gmm = gmm.predict(xs)

#y\_cluster\_gmm

plt.subplot(2, 2, 3)

plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[y\_gmm], s=40)

plt.title('GMM Classification')

plt.xlabel('Petal Length')

plt.ylabel('Petal Width')

print('The accuracy score of EM: ',sm.accuracy\_score(y, y\_gmm))

print('The Confusion matrix of EM: ',sm.confusion\_matrix(y, y\_gmm))

plt.show()

**9.Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem**

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn import datasets

iris=datasets.load\_iris()

x = iris.data

y = iris.target

print ('sepal-length', 'sepal-width', 'petal-length', 'petal-width')

print(x)

print('class: 0-Iris-Setosa, 1- Iris-Versicolour, 2- Iris-Virginica')

print(y)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.3)

#To Training the model and Nearest nighbors K=5

classifier = KNeighborsClassifier(n\_neighbors=5)

classifier.fit(x\_train, y\_train)

#To make predictions on our test data

y\_pred=classifier.predict(x\_test)

print('Confusion Matrix')

print(confusion\_matrix(y\_test,y\_pred))

print('Accuracy Metrics')

print(classification\_report(y\_test,y\_pred))

**10.Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.**

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

def kernel(point, xmat, k):

m, n = np.shape(xmat)

weights = np.mat(np.eye((m)))

for j in range(m):

diff = point - X[j]

weights[j, j] = np.exp(diff \* diff.T / (-2.0 \* k \*\* 2))

return weights

def localWeight(point, xmat, ymat, k):

wei = kernel(point, xmat, k)

W = (X.T \* (wei \* X)).I \* (X.T \* (wei \* ymat.T))

return W

def localWeightRegression(xmat, ymat, k):

m, n = np.shape(xmat)

ypred = np.zeros(m)

for i in range(m):

ypred[i] = xmat[i] \* localWeight(xmat[i], xmat, ymat, k)

return ypred

data = pd.read\_csv('D:/ML\_Pracs/tips.csv')

bill = np.array(data.total\_bill)

tip = np.array(data.tip)

mbill = np.mat(bill)

mtip = np.mat(tip)

m = np.shape(mbill)[1]

one = np.mat(np.ones(m))

X = np.hstack((one.T, mbill.T))

ypred = localWeightRegression(X, mtip, 0.5)

SortIndex = X[:, 1].argsort(0)

xsort = X[SortIndex][:, 0]

fig = plt.figure()

ax = fig.add\_subplot(1, 1, 1)

ax.scatter(bill, tip, color='green')

ax.plot(xsort[:, 1], ypred[SortIndex], color='red', linewidth=5)

plt.xlabel('Total bill')

plt.ylabel('Tip')

plt.show();

**code**

import numpy as np

from bokeh.plotting import figure, show, output\_notebook

from bokeh.layouts import gridplot

from bokeh.io import push\_notebook

def local\_regression(x0, X, Y, tau): # add bias term

x0 = np.r\_[1, x0] # Add one to avoid the loss in information

X = np.c\_[np.ones(len(X)), X]

xw = X.T \* radial\_kernel(x0, X, tau) # XTranspose \* W

beta = np.linalg.pinv(xw @ X) @ xw @ Y # @ Matrix Multiplication or Dot Product

return x0 @ beta # @ Matrix Multiplication or Dot Product for prediction

def radial\_kernel(x0, X, tau):

return np.exp(np.sum((X - x0) \*\* 2, axis=1) / (-2 \* tau \* tau))

n = 1000

X = np.linspace(-3, 3, num=n)

print("The Data Set ( 10 Samples) X :\n", X[1:10])

Y = np.log(np.abs(X \*\* 2 - 1) + .5)

print("The Fitting Curve Data Set (10 Samples) Y :\n", Y[1:10])

X += np.random.normal(scale=.1, size=n)

print("Normalised (10 Samples) X :\n", X[1:10])

domain = np.linspace(-3, 3, num=300)

print(" Xo Domain Space(10 Samples) :\n", domain[1:10])

def plot\_lwr(tau):

prediction = [local\_regression(x0, X, Y, tau) for x0 in domain]

plot = figure(width=400, height=400)

plot.title.text = 'tau=%g' % tau

plot.scatter(X, Y, alpha=.3)

plot.line(domain, prediction, line\_width=2, color='red')

return plot

show(gridplot([

[plot\_lwr(10.), plot\_lwr(1.)],

[plot\_lwr(0.1), plot\_lwr(0.01)]]))