Pattern Recognition

Project 2: Least Squares Regression
Nearest Neighbor Classifiers

Group members:
Alina Arunova
Mohammad Ali Ghasemi
Aditya Kela
Aleksandr Korovin
Marina Mircheska
Sattar Rahimbeyli

Task 2.1: least squares regression for missing value prediction

Task

- load the file whDataquot;.dat, remove the outliers and collect the remaining height and weight data in two vectors x and y respectively
- use the method of least squares to fit polynomial models to the data

$$y(x) = \sum_{j=0}^{d} w_j x^j$$

- fit models for $d \in \{1, 5, 10\}$ and plot the results
- use each of your resulting models to predict a weight value for the outliers
- possible solutions:
  - **numpy.polynomial.polynomial.polyfit(***x***,** *y***,** *deg***,** *rcond=None***,** *full=False***,** *w=None***)** – numerical problems
  - **numpy.polyfit**(*x***,** *y***,** *deg***,** *rcond=None***,** *full=False***,** *w=None***,** *cov=False*)
  - **numpy.linalg.lstsq**(*a***,** *b***,** *rcond=-1*) - R squared is worse than polyfit

Task 2.1: least squares regression for missing value prediction

Solution

Preparing data

```
# read data as 2D array of data type 'object'
data = np.loadtxt('whData.dat',dtype=np.object,comments='#',delimiter=None)

# read height and weight data into 2D array (i.e. into a matrix)
data = data[:,0:2].astype(np.float)

# remove the outliers
data_train = data[data[:,0] > 0]

# prediction data (outliers)
data_pred = data[data[:,0] ==-1]

# create height vector for prediction data
predict_x = np.copy(data_pred[:,1])

# create weight vector for train data
y = np.copy(data_train[:,0])

# create height vector for train data
x = np.copy(data_train[:,1])

d = [1, 5, 10]
```

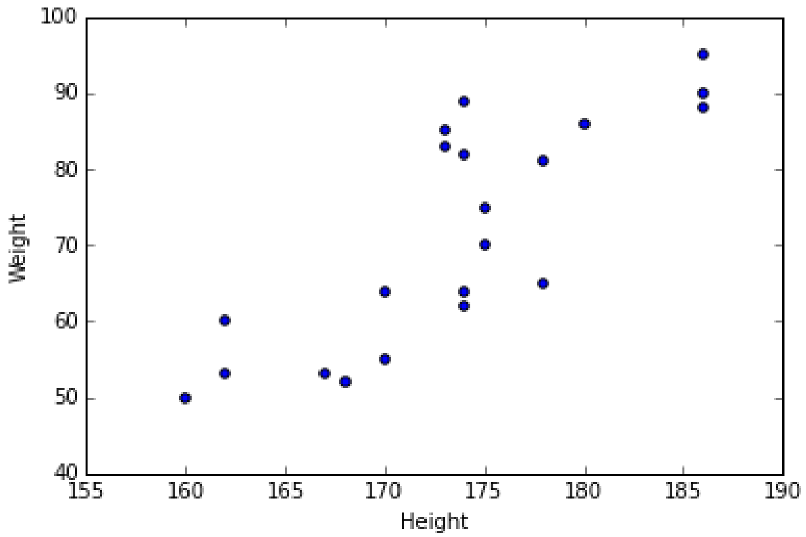
Task 2.1: least squares regression for missing value prediction

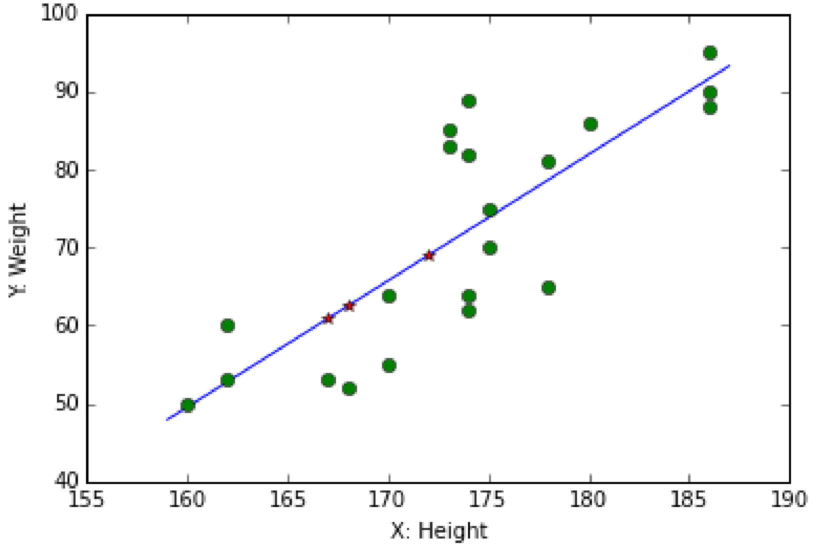
Solution: code

```
for i in d:
    predicator = np.poly1d(np.polyfit(x,y,i)) #

    predict_y = predicator(predict_x)
    for n in range(len(predict_y)):
        print 'X = ', predict_x[n], '| Y = ',predict_y[n]
    print "This is our polynomial function: \n", predicator

    xs = np.linspace(x.min() - 1, x.max() + 1, 1000)
    ys = np.dot(np.vander(xs, i + 1), predicator)
    plt.plot(xs, ys, '-')
    plt.plot(x, y, 'o')
    plt.plot(predict_x,predict_y,'*')
    plt.ylabel("Y: Weight")
    plt.xlabel("X: Height")
    plt.show()
    res = np.array([y[k] - predicator(x[k]) for k in range(len(x))])
    ssres = np.sum(res**2)
    sstot =np.sum((y - np.mean(y))**2)
    r2 = 1-ssres/sstot
    print 'R-squared = ', r2
```

Task 2.1: least squares regression for missing value prediction

Solution: plots

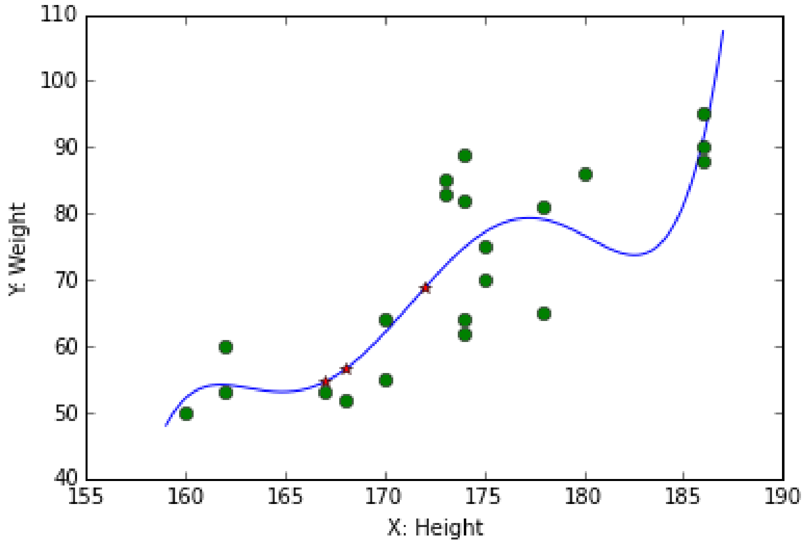




d = 1

X = 168.0 | Y = 62.51
X = 172.0 | Y = 68.98
X = 167.0 | Y = 60.89

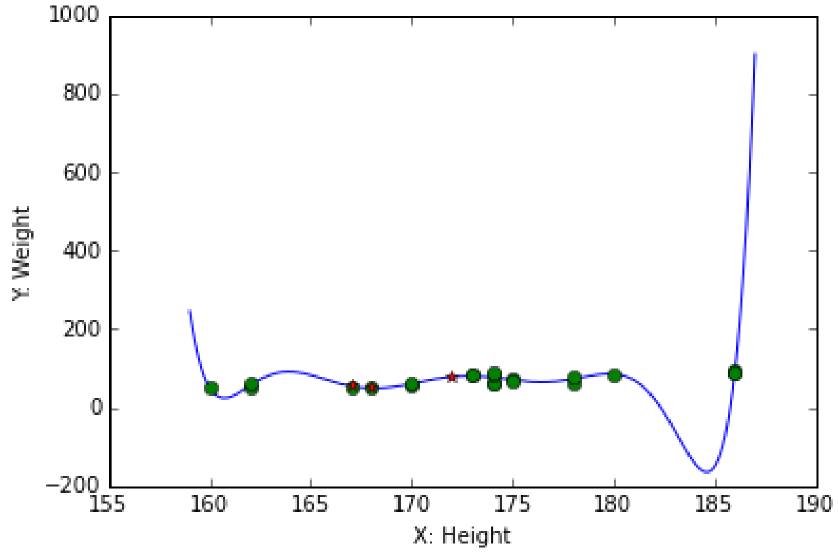
R-squared = 0.657



d = 5

X = 168.0 | Y = 56.49
X = 172.0 | Y = 68.77
X = 167.0 | Y = 54.64

R-squared = 0.7



d = 10

X = 168.0 | Y = 48.99
X = 172.0 | Y = 78.49
X = 167.0 | Y = 54.45

R-squared = 0.81

Task 2.2: conditional expectation for missing value prediction

Bivariate Distribution:

$$\mathcal{N}(h, w) = \frac{1}{2\pi\,\sigma_h\,\sigma_w\sqrt{1-\rho^2}}\, e^{-\frac{1}{2(1-\rho^2)}\left[\frac{(h-\mu_h)^2}{\sigma_h^2} + \frac{(w-\mu_w)^2}{\sigma_w^2} - 2\rho\frac{(h-\mu_h)(w-\mu_w)}{\sigma_h\,\sigma_w}\right]}$$

Conditional expectation for $\boldsymbol{w}$ given that $\boldsymbol{h} = \boldsymbol{h_0}$ is then

$$\mathbb{E}[w \mid h = h_0] = \int w\,\mathcal{N}(w \mid \mu_{w|h=h_0}, \sigma^2_{w|h=h_0})\, dw \;=\; \mu_w + \rho\,\frac{\sigma_w}{\sigma_h}(h_0 - \mu_h)$$

Task 2.2: conditional expectation for missing value prediction

Solution: code

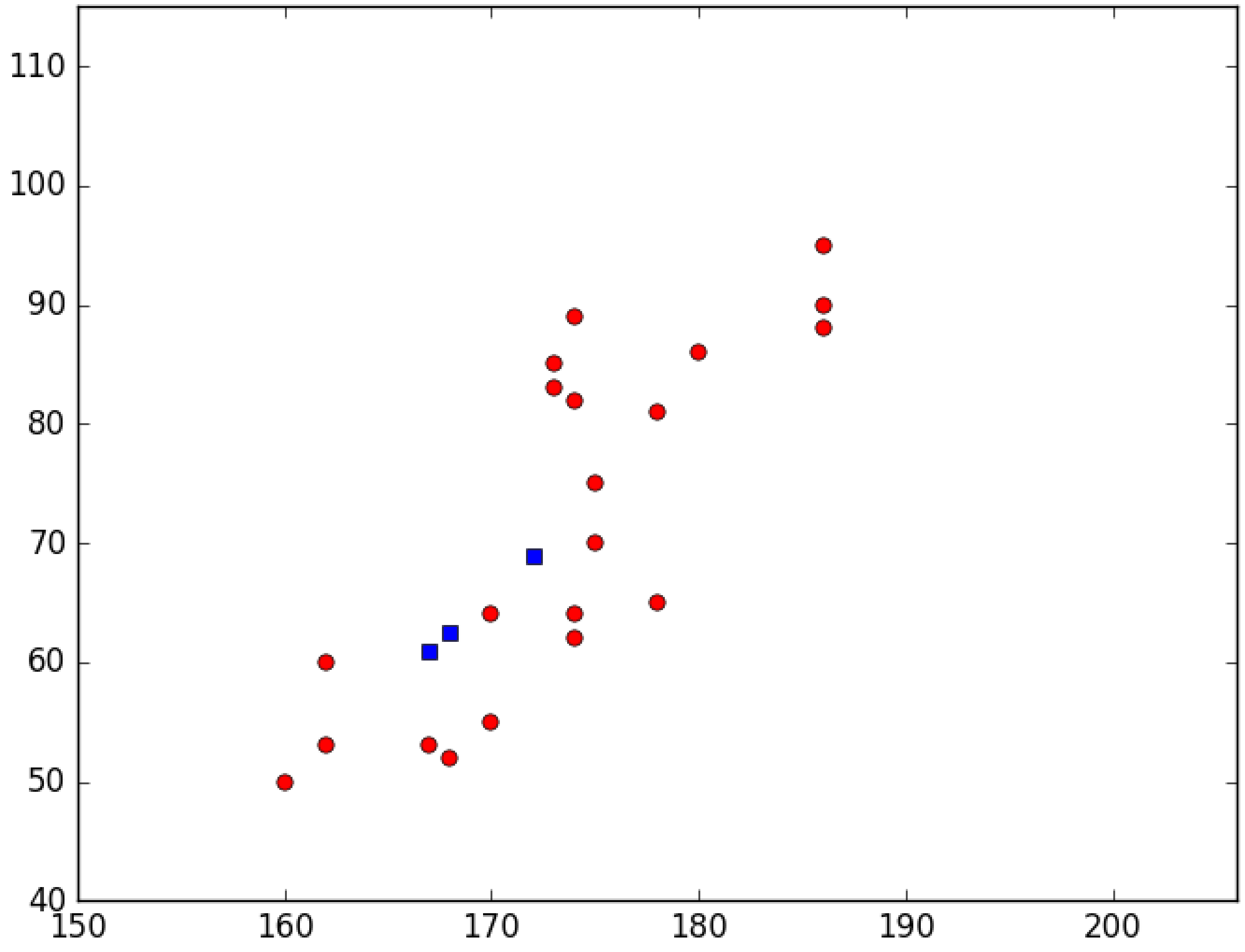
```
meanofweight = np.round(np.mean(y),2)
meanofheight = np.round(np.mean(x),2)
devweight = np.round(np.std(y),2)
devheight = np.round(np.std(x),2)
corrcoef = np.corrcoef(x,y)[0,1]
z = 0
predict_y =np.zeros([len(predict_x)])
while (z!=len(predict_x)):
    h0 = predict_x[z]
    predicted = meanofweight+corrcoef *(h0 - meanofheight) *devweight/devheight
    print predicted
    predict_y[z] = predicted
    z= z + 1
```

Output

```
62.5089084933
68.9811451306
60.890849334
```

Task 2.2: conditional expectation for missing value prediction

Plot



Task 2.3: Bayesian regression for missing value prediction

We assume that the points are generated by the according to: $y_i = \sum_{j=0}^{d} w_j x_i^j + \epsilon_i$

We therefore want to find the 'best' *w*. We do this by choosing *w* such that:

$$\begin{aligned} \boldsymbol{w}_{MAP} &= \operatorname*{argmax}_{\boldsymbol{w}} p(\boldsymbol{w} \mid D) \\ &= \tfrac{1}{\sigma^2}\left(\tfrac{1}{\sigma^2}\boldsymbol{X}^T\boldsymbol{X} + \tfrac{1}{\sigma_0^2}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \\ &= \left(\boldsymbol{X}^T\boldsymbol{X} + \tfrac{\sigma^2}{\sigma_0^2}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} \end{aligned}$$

Where:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ & & & \vdots & \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{bmatrix}$$

Task 2.3: Bayesian regression for missing value prediction

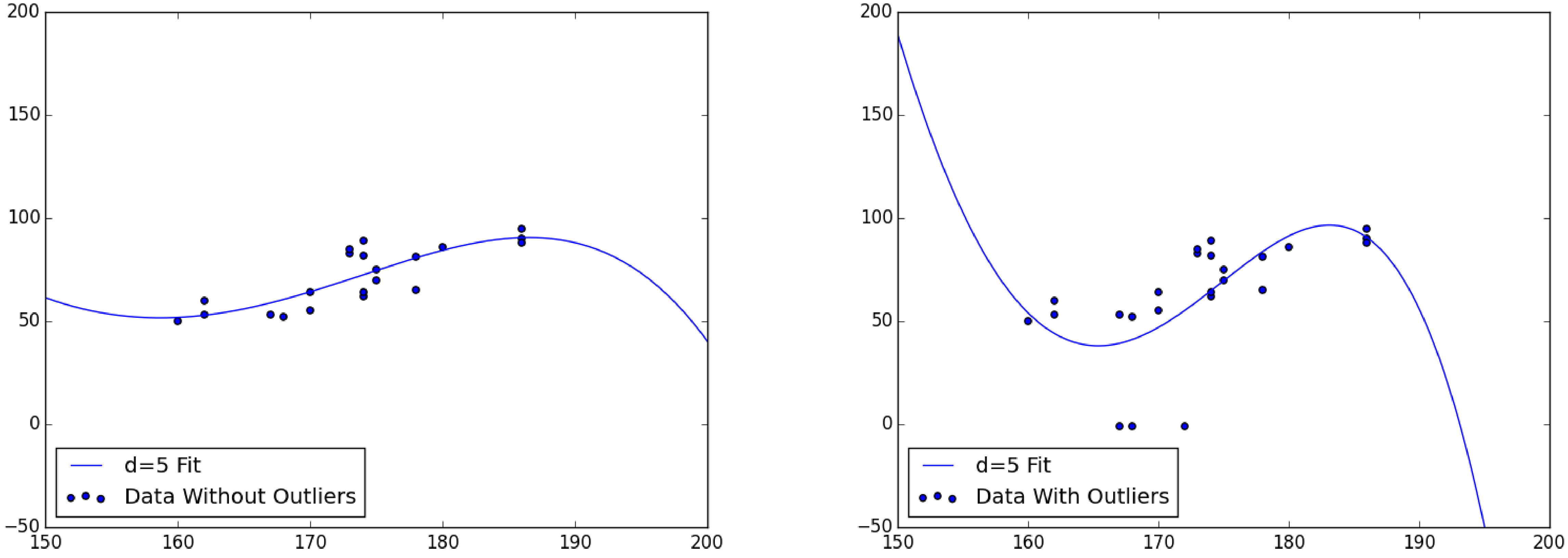
$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ & & & \vdots & \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{bmatrix}$$

```python
def xMatrix(l,d):
    xMatrix=np.zeros((len(l),d))
    for i in range(len(l)):
        for j in range(d):
            xMatrix[i][j]=l[i]**j
    return xMatrix
```

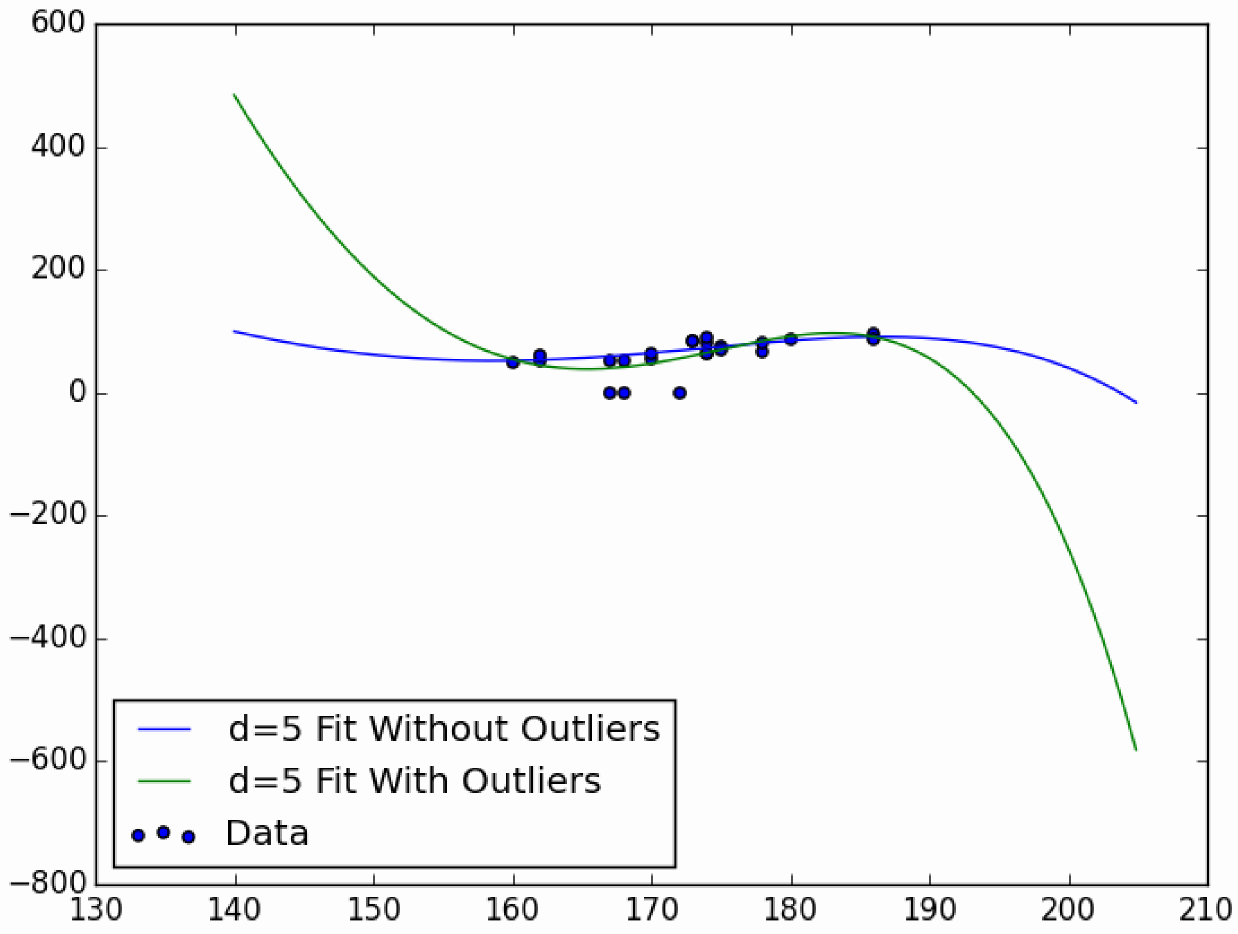
$$\left(\boldsymbol{X}^T\boldsymbol{X} + \frac{\sigma^2}{\sigma_0^2}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

```python
def MAP(x,y,d,sigma0square):
    X=xMatrix(x,d)
    return
np.matmul(np.linalg.inv(np.matmul(np.transpose(X),X)+(np.var(y)/sigma0square)*np.identity(d)),np.matmul(np.transpose(X),y))
```

Task 2.3: Bayesian regression for missing value prediction



Task 2.3: Bayesian regression for missing value prediction



Task 2.4: nearest neighbor classifier

Task:

- Implement a function that realizes an n-nearest neighbor classifier
- Determine the overall run time for computing the 1-nearest neighbor
- Determine the recognition accuracy for different $\mathrm{n} \in \{1, 3, 5\}$

Given:

- Test data (data2-test.dat)
- Train data (data2-train.dat)
- $\mathrm{n} \in \{1, 3, 5\}$

Task 2.4: nearest neighbor classifier

Solution

Non-parametric method for classification

Given a training data $\mathrm{X} = \{(x_i,\ y_i)\}_{i=1}^{n}$ and a new sample q

Algorithm:

- Select k entries in database (training data) closest to the new sample q (Euclidean distance)

$$\|x_i - q\|^2 = \sum_{h=1}^{m} (X_{hi} - q_h)^2$$

- Find the most common (Majority vote) classification and give it to q

Task 2.4: nearest neighbor classifier

Solution: code

K-Nearest neighbors function:

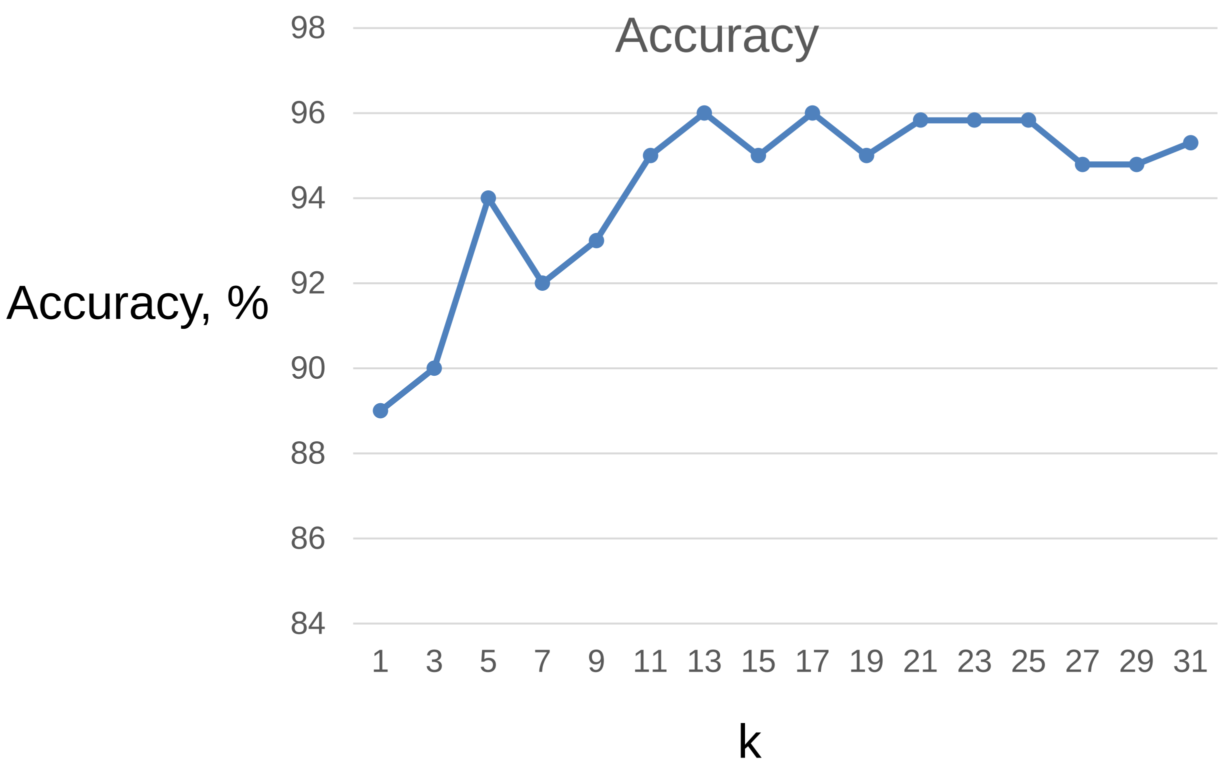
```
def k_nearest_neighbors(data, query_point, k):
    #calculate distance vector and use argsort - to sort the indices.
    sorted_inds = np.argsort(np.sum((data - query_point)**2, axis=1))
    #return first k of sorted indices
    return sorted_inds[:k]
```

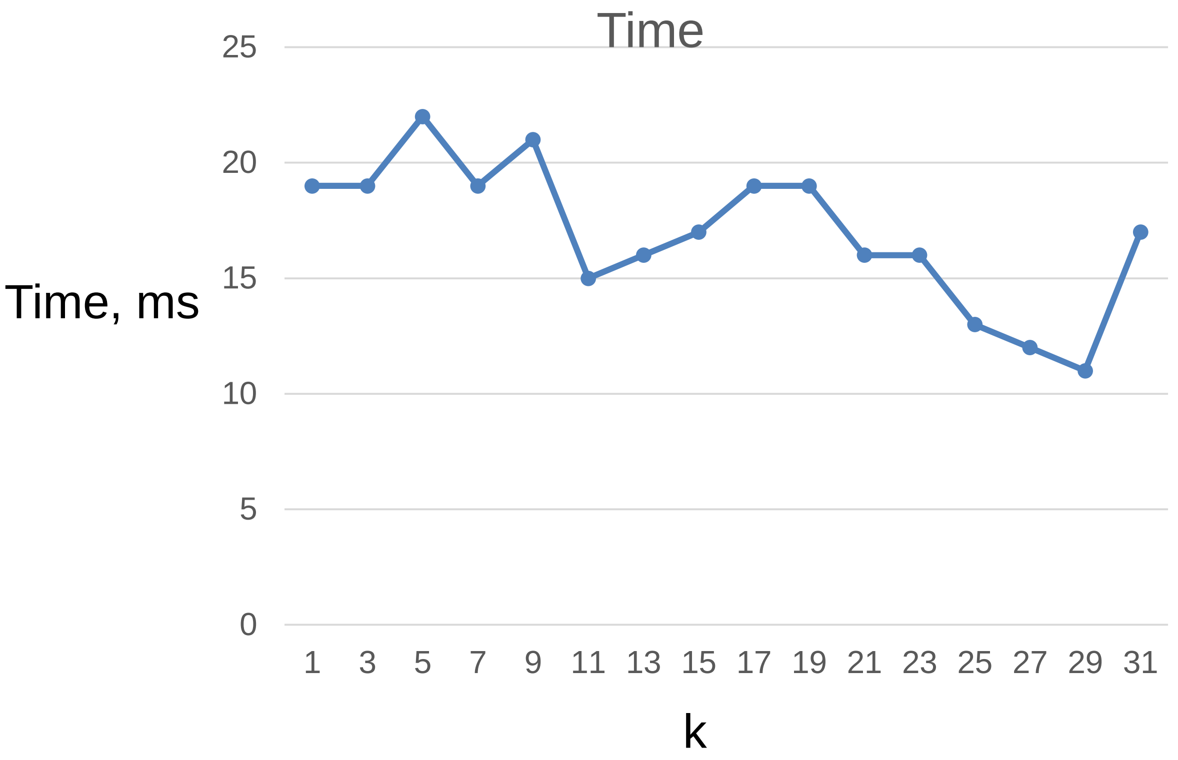
Majority vote function:

```
def classify(neighbors):
    freqresult = itemfreq(neighbors)
    sorted_freq = freqresult[freqresult[:, 1].argsort()[::-1]]
    return sorted_freq[0, 0]
```

Task 2.4: nearest neighbor classifier

Solution: results





n = 1

argsort	argmin
23 ms	14 ms

Task 2.5: computing a kD-tree

Solution

- we create a kD-tree class with various parameters to tune tree performance

```
class kDTree():

    def __init__(self, X, split_dim, split_dim_style, split_point_style, depth, rectangle):
        '''
        Creates a node of a kD-tree
        :param X: the data passed to the node
        :param split_dim: the splitting dimension passed to the node
            0 for x axis
            1 for y axis
        :param split_dim_style: determine splitting dimension style
            0 for round robin
            1 for higher variance dimension
        :param split_point_style: determite splitting point style
            0 for splitting at midpoint
            1 for splitting at median
        :param depth: the depth of the current node in the tree
        :param rectangle: the rectangle for the current node
        '''
....
```

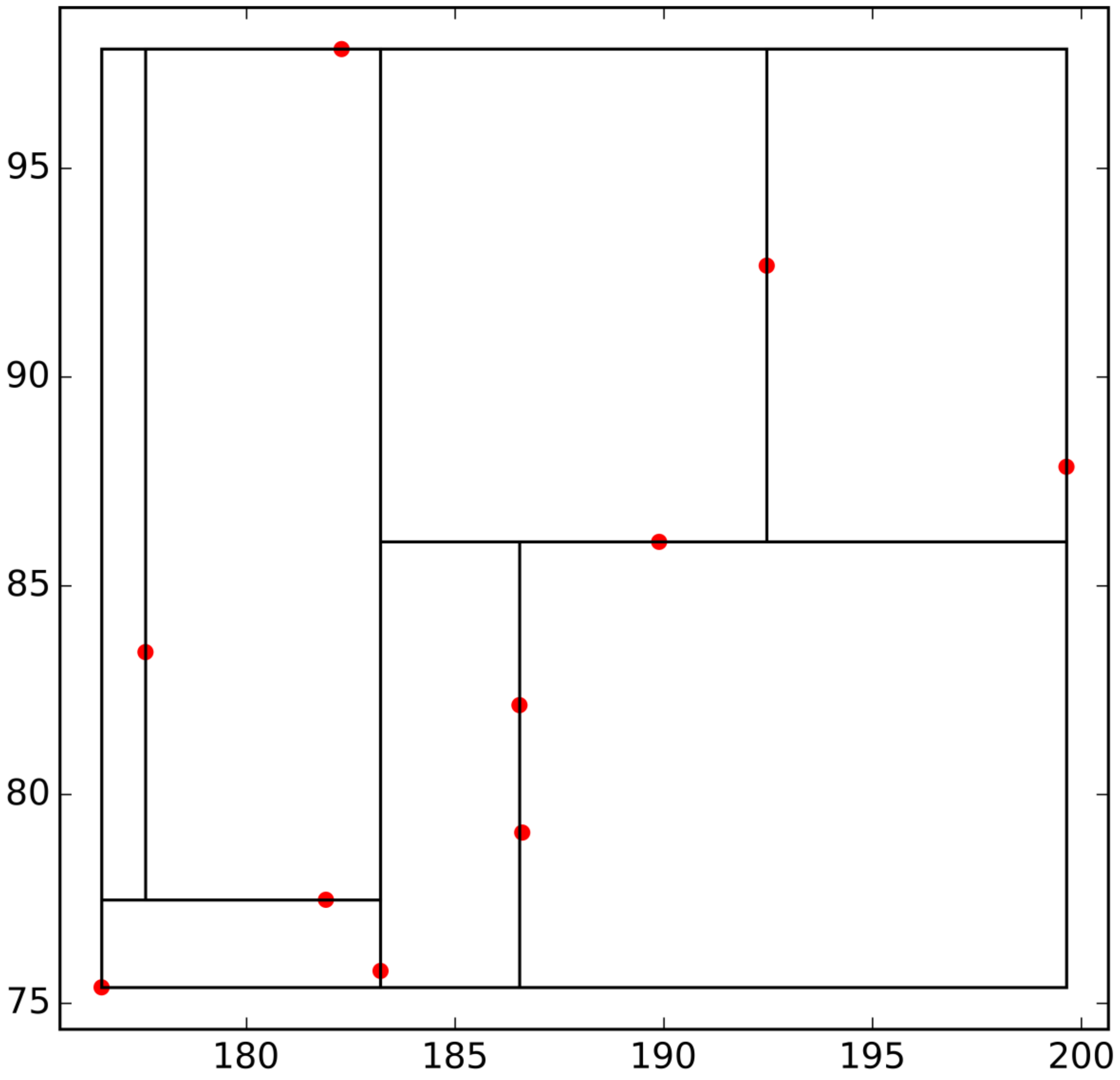
```
# adjust the node's splitting dimension according to the chosen splitting dimension style
if (self.split_dim_style):
    # split along the dimension with higher variance
    self.split_dim = 0 if (np.var(self.X[:,0]) > np.var(self.X[:,1])) else 1
    new_dim = self.split_dim
else:
    # split in round robin fashion
    self.split_dim = split_dim
    new_dim = 1 - self.split_dim

# choose splitting point according to the chosen splitting point style
if (self.split_point_style):
    # split according to median of data
    if ((self.data.size % 2) == 0):
        self.split_point = self.data[self.data.size/2 - 1]
    else:
        self.split_point = self.data[self.data.size/2]
else:
    # split according to midpoint of data
    n = np.argmax(self.data > np.mean(self.data))
    self.split_point = self.data[n-1]
```

```
# if the splitted data parts are not empty, add the appropriate node in the kD-tree
if (left_data.size > 0):
    self.left = kDTree(left_data, new_dim, self.split_dim_style, self.split_point_style, depth+1, rect_left)
if (right_data.size > 0):
    self.right = kDTree(right_data, new_dim, self.split_dim_style, self.split_point_style, depth+1, rect_right)
```

Task 2.5: computing a kD-tree

Plots

- we plot the kD-tree by computing rectangles for each node
- we start with a rectangle that envelopes all the training data points
- each new node inherits its parent’s rectangle
- on each split we update one corner coordinate of the node’s rectangle
- example: tree of ten points where we select splitting dimensions in round robin fashion and we split at the median



Task 2.5: computing a kD-tree

Solution

- we plot four different kinds of kD-trees of the data in data2-train.dat
- following ideas were combined:

  - selecting the splitting dimension:

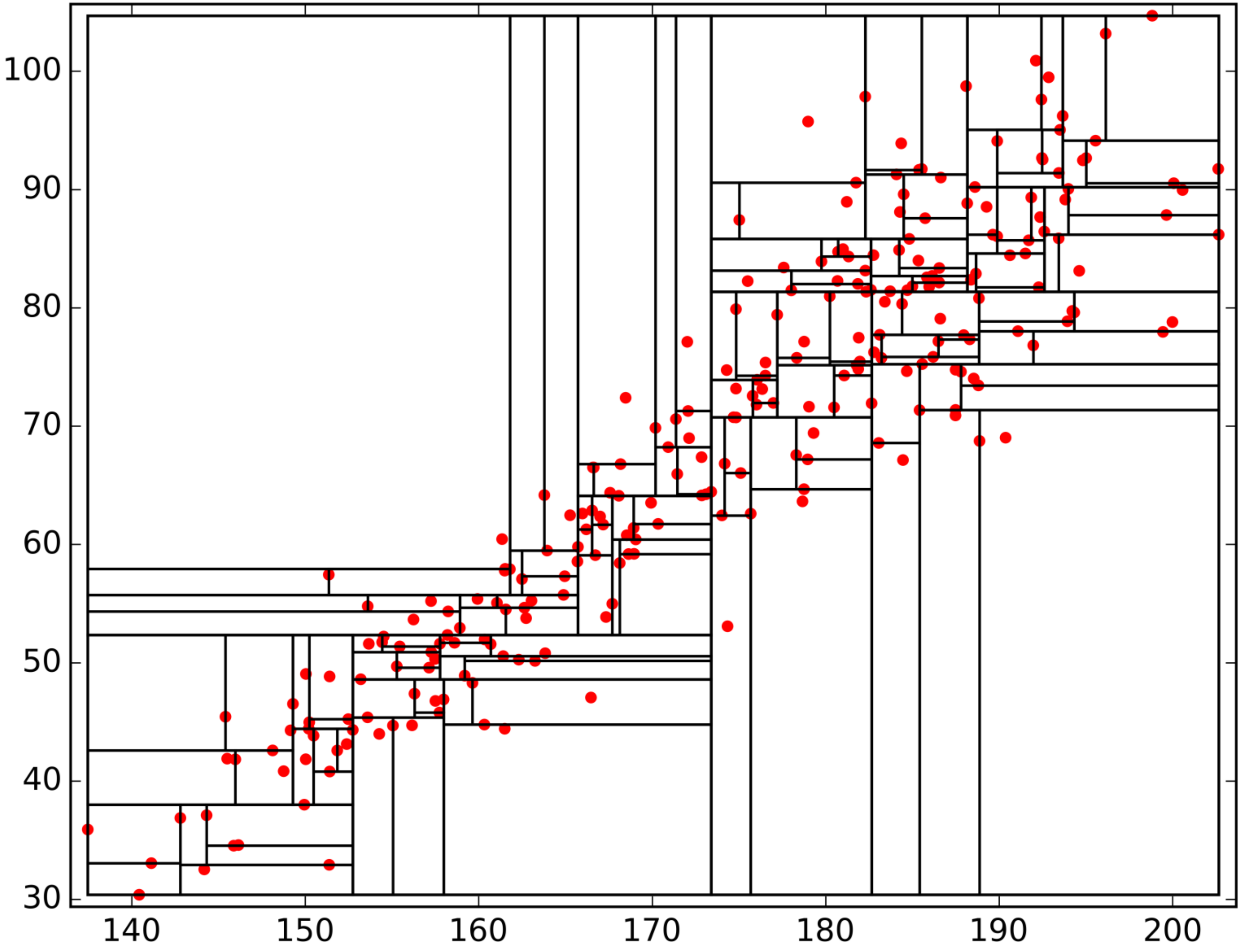
    1. round robin fashion (alternating between x and y starting with x)
    2. select the dimension with higher variance (variance computed on data of the subtree)

  - selecting the splitting point:

    1. midpoint of the data
    2. median of the data

Task 2.5: computing a kD-tree

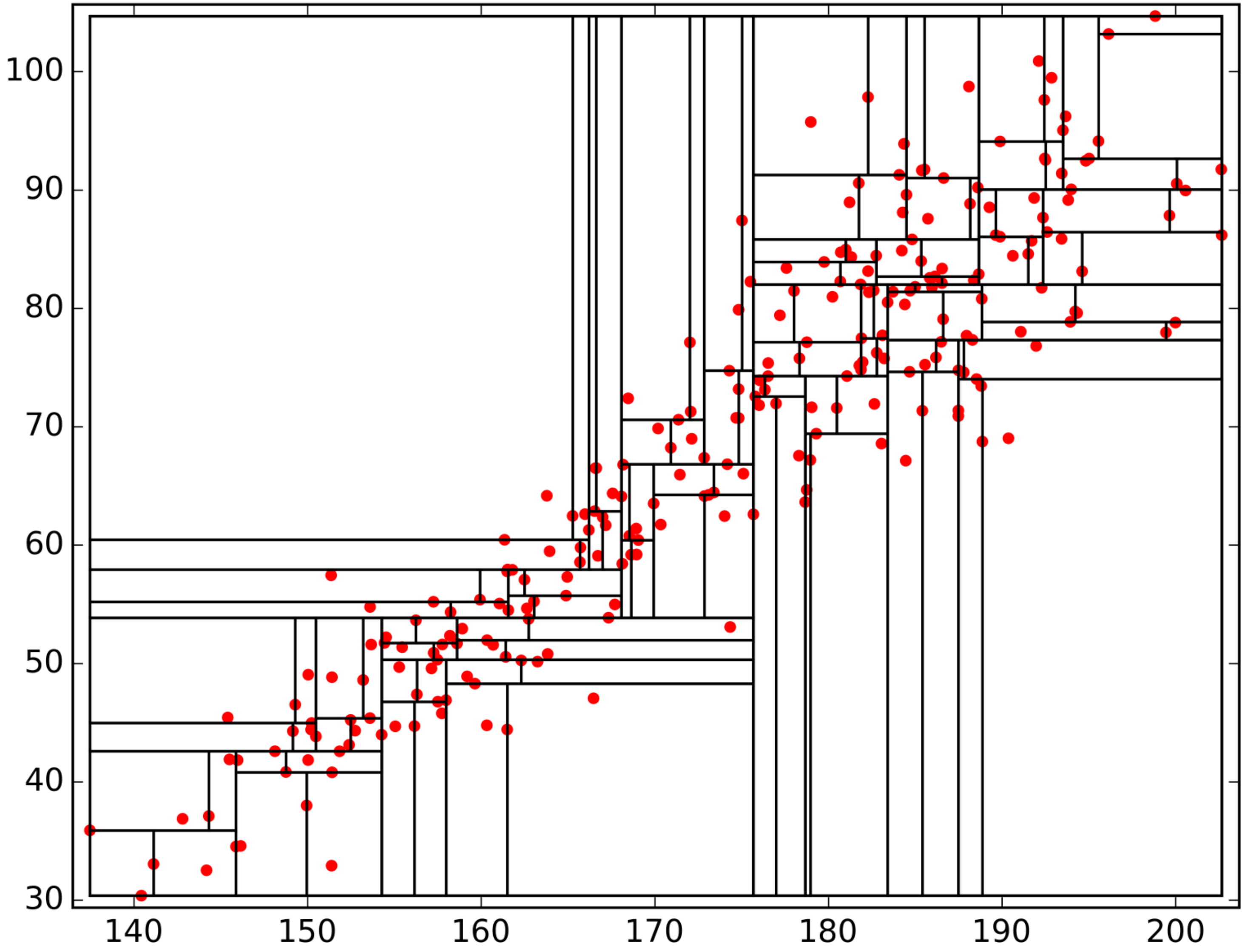
Plots



Tree A: splitting dimension - round robin; splitting point - midpoint

Task 2.5: computing a kD-tree

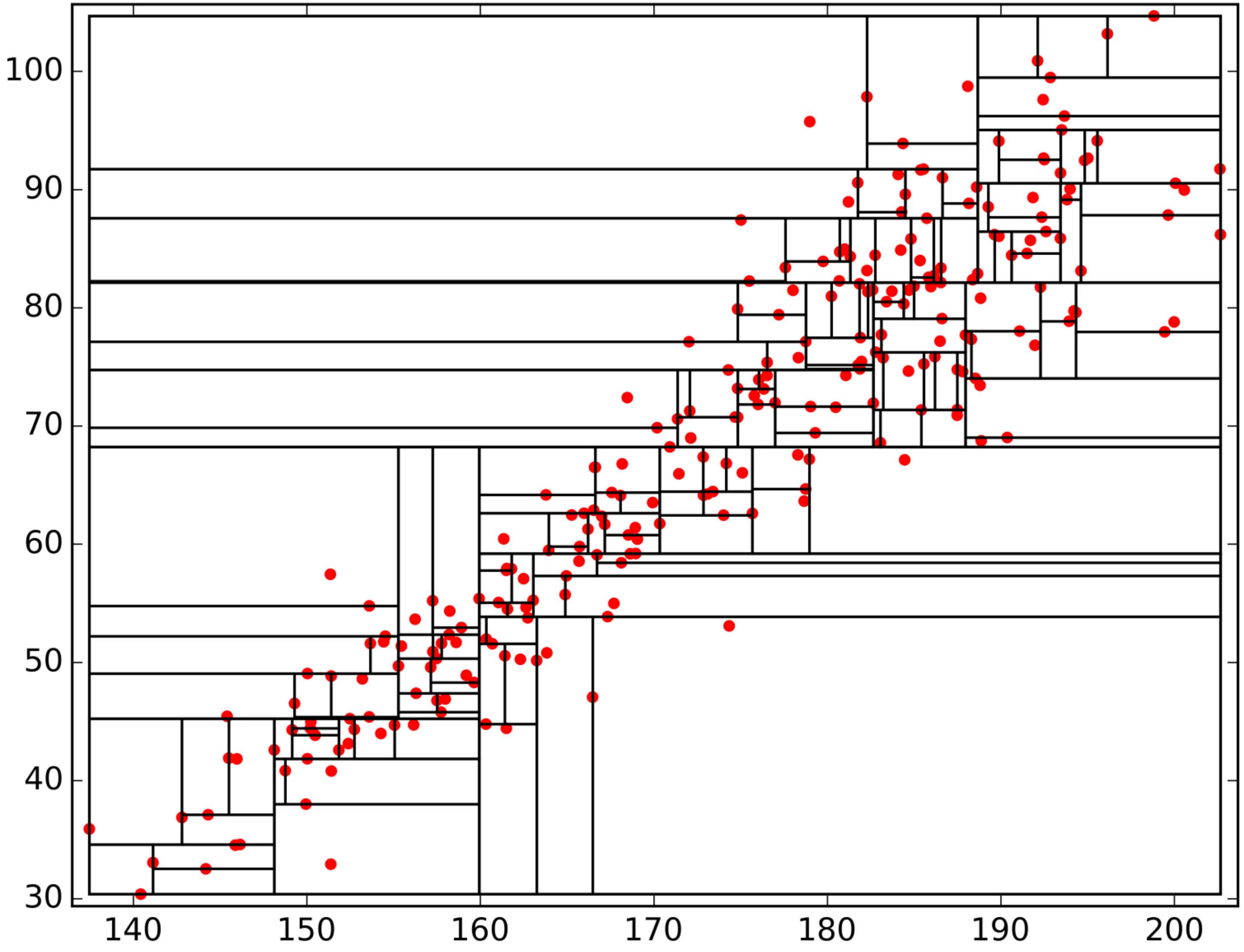
Plots



Tree B: splitting dimension - round robin; splitting point - median

Task 2.5: computing a kD-tree

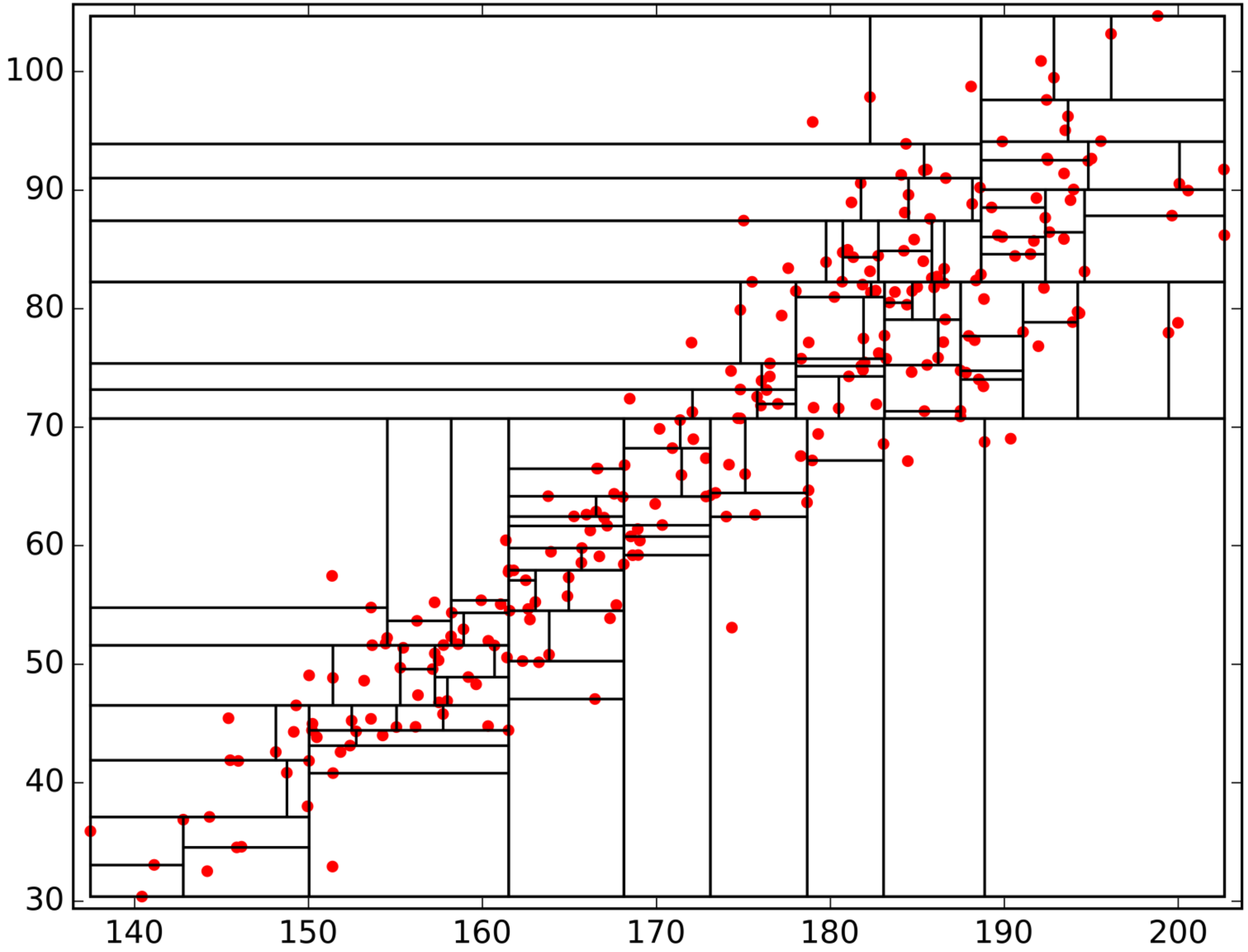
Plots



Tree C: splitting dimension – with highest variance; splitting point - midpoint

Task 2.5: computing a kD-tree

Plots



Tree D: splitting dimension – with highest variance; splitting point - median

Task 2.5: computing a kD-tree

Observations

- Overall runtime for computing 1-nearest neighbor of all points in data2-test.dat
- Tree A: 0.0073 seconds; max depth 10
- Tree B: 0.0070 seconds; balanced, depth 7 for all nodes
- Tree C: 0.008 seconds; max depth 8, average depth 7.5
- Tree D: 0.0076 balanced, balanced, depth 7 for all nodes

- Choosing the midpoint as a splitting point yields “faster” trees

  - queries are faster since the splitting plane is arithmetically better centered
- Choosing the median as a splitting point yields balanced trees

  - depth is less, but that doesn’t speed the tree up – this is because the median point splits the data arithmetically (with respect to the Euclidean distance) not so well