project-02-demo

January 7, 2016

1 Project 2

1.1 B-IT Pattern Recognition

Presented on 07-Jan-2016 by:

- Abdullah Abdullah
- Can Güney Aksakallı
- Kang Cifong
- Umut Hatipoğlu

In [1]:
```
import numpy as np
import scipy.spatial as ss
import matplotlib.pyplot as plt
%matplotlib inline

import timeit

import pattrex.plotting_mpl as plt_rex
import pattrex.preprocessing as pre_rex
import pattrex.fitting as fit_rex
import pattrex.kdTreeCK as kd_rex
import pattrex.KNearestNeighbor as knn_rex


from pattrex.demo_helper import read_whdata
```

In [2]:
```
# Read data
ws, hs, gs = read_whdata()
HW = np.vstack((hs, ws)).astype(np.float)

# removing outliers
HW_new, neg_idx = pre_rex.only_all_positive(HW, True, return_neg_idx=True)

# unknown
hu = np.array([h for i, h in enumerate(hs) if i in neg_idx])

hn = HW_new[0, :]
wn = HW_new[1, :]
```

```
In [3]: qx = np.random.uniform(3, 5, 60)
        qx = np.hstack((qx, np.random.uniform(-5, -0.5, 40)))
        qy = np.abs(qx) + np.random.uniform(0, 1.0, 100)
        qx = np.hstack((qx, np.array([1, 2, 3, 5, 12])))
        qy = np.hstack((qy, np.array([11, 3.5, 2.0, 10, 21])))

        qu = np.array([1.1, 2.2, -3.3, -1.1])
```

1.2 Task 2.1

1.2.1 Ordinary Least Squares Regression

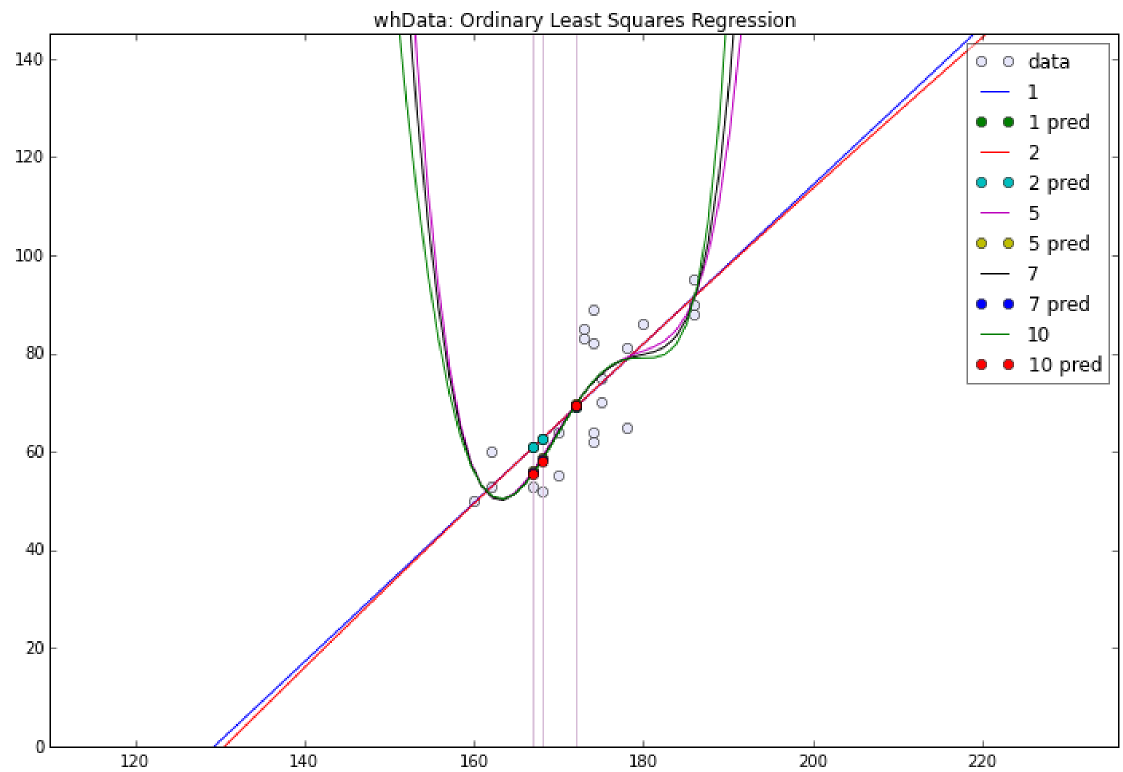
- We are assuming that Weight is a function of Height
- We are assuming that the function is an nth degree polynomial
- We use `numpy.linalg.lstsq(...)` to do the job for us
  - If doing *manually*, one can use `numpy.linalg.pinv(...)` to calculate the Moore-Penrose pseudo-inverse

```
In [4]: def demo_1(x, y, u, degrees, title, padding=50):

            # fit polynomial of degrees 1...10
            results = []
            for degree in range(1, max(degrees)+1):
                results.append(
                    fit_rex.fit_polynomial_nplstsq(x, y, degree
                                                   , x_pad=padding
                                                   , X_unknown=u))

            # # pretty print and plot results for [1, 5, 10] ###########
            coeffs = []
            coeffs_string = "{}: "

            preds = []
            preds_string = "{}: "

            fig = plt.figure(figsize=(12, 8))
            axs = fig.add_subplot(111)

            XY_ = np.vstack((x, y))
            xmin, ymin = XY_.min(axis=1)
            xmax, ymax = XY_.max(axis=1)

            xlim = [xmin-padding, xmax+padding]
            ylim = [ymin-padding, ymax+padding]

            # plot data
            plt_rex.plot2d(XY_, colwise_data=True,
                           hatch='o', color='lavender',
                           x_lim=xlim, y_lim=ylim,
                           show=False, axs=axs, plotlabel="data",
                           title=title)

            # plot unkown vertical lines
```

```
            for h in u:
                axs.axvline(x=h, color='thistle')

            for degree in degrees:
                res = results[degree-1]

                coeffs.append(res[0])

                preds.append(res[2][1])
                preds_string += "{:4.3} |"

                line = res[1]
                pred = res[2]

                plt_rex.plot2d(np.vstack(line), colwise_data=True,
                              hatch='-',
                              show=False, axs=axs,
                              plotlabel=str(degree))
                plt_rex.plot2d(np.vstack(pred), colwise_data=True,
                              hatch='o',
                              show=False, axs=axs,
                              plotlabel=str(degree)+" pred")

            print("Coefficients")
            print("\n\n".join("{}".format(c) for c in coeffs))
            print()
            print("Predictions " + str(degrees))
            print("\n".join(preds_string.format(*p) for p in zip(u, *preds)))
```

```
In [5]: demo_1(hn, wn, hu, [1, 2, 5, 7, 10]
             , "whData: Ordinary Least Squares Regression")
```

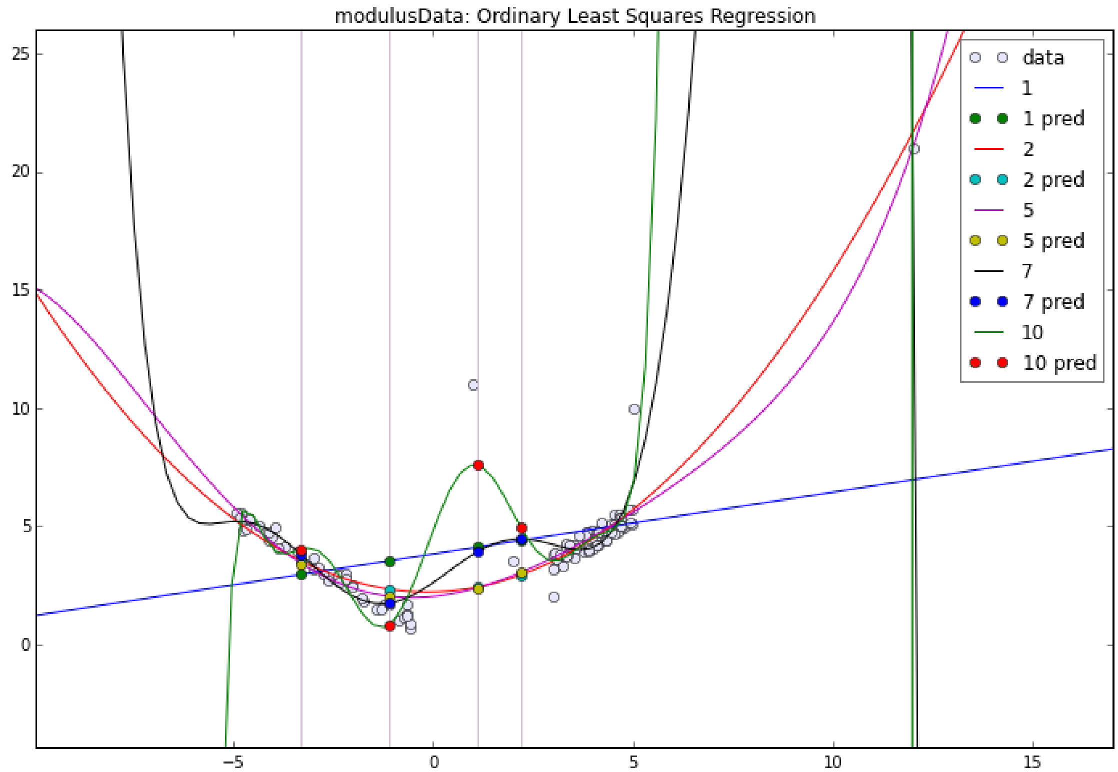
```
Coefficients
[-209.32503027    1.61805916]

[ -2.42736271e+02   2.00365675e+00  -1.11060527e-03]

[  1.45372950e+02   5.00648642e+03  -1.15279840e+02   9.94128981e-01
  -3.80603682e-03   5.45891635e-06]

[  1.30258110e-06   9.54928046e-05   5.52169250e-03   1.90340817e-01
  -4.38414360e-03   3.78208524e-05  -1.44853576e-07   2.07836521e-10]

[  3.39741629e-19   1.17900058e-11  -9.63490518e-15   2.89518900e-13
   2.13966507e-11   1.23047796e-09   4.24710787e-08  -9.78013259e-10
   8.43625378e-12  -3.23109114e-14   4.63624155e-17]

Predictions [1, 2, 5, 7, 10]
168.0: 62.5 |62.5 |58.8 |58.4 |58.0 |
172.0: 69.0 |69.0 |69.7 |69.5 |69.5 |
167.0: 60.9 |60.9 |56.1 |55.7 |55.5 |
```



1.2.2 for some noisy data for $y = mod(x)$

```
In [6]: demo_1(qx, qy, qu, [1, 2, 5, 7, 10]
               , "modulusData: Ordinary Least Squares Regression"
               , padding=5)
```

```
Coefficients
[ 3.81448481  0.26177253]

[ 2.21066934  0.03797136  0.13196377]

[  2.01791830e+00   1.71706932e-01   1.54870230e-01  -9.56041981e-03
  -3.12282467e-04   7.08850508e-05]

[  2.60248403e+00   1.17549562e+00   1.91738204e-01  -1.67679169e-01
  -9.57249546e-03   6.47605169e-03   2.90816285e-04  -5.62320636e-05]

[  4.80327744e+00   4.61197166e+00  -6.36256661e-01  -1.45574145e+00
   1.34855931e-01   1.60432446e-01  -1.31138913e-02  -7.38866864e-03
   5.99914376e-04   1.21107070e-04  -9.99848698e-06]

Predictions [1, 2, 5, 7, 10]
1.1:  4.1 |2.41 |2.38 | 3.9 |7.59 |
2.2: 4.39 |2.93 |3.04 |4.46 |4.92 |
-3.3: 2.95 |3.52 |3.42 |3.78 |4.02 |
-1.1: 3.53 |2.33 |2.03 |1.74 |0.829 |
```



More, later with Task 2.3

1.3 Task 2.2

1.3.1 Conditional Expectation from a Bivariate Gaussain

- We used the equations from the slides for fitting a Bivariate Gaussian

$$
\begin{aligned}
\mathbf{E}\left[w \mid h = h_0\right] &= \int w \mathcal{N}\left(w \mid \mu_{w|h=h_0}, \sigma^2_{w|h=h_0}\right) dw \\
&= \mu_{w|h=h_0}
\end{aligned}
$$

where

$$
\begin{aligned}
\mu_{w|h=h_0} &= \mu_w + \rho \frac{\sigma_w}{\sigma_h}(h_0 - \mu_h) \\
\sigma^2_{w|h=h_0} &= \sigma^2_w (1 - \rho^2) \\
\rho &= \frac{cov(h, w)}{\sigma_h \sigma_w}
\end{aligned}
$$

- We used `numpy.cov(...)` to calculate the covariance
  - by default, the normalization is by `N-1`.
  - for MLE estimate, use `ddof=0`
  - we used the default, but standard-deviation is still biased

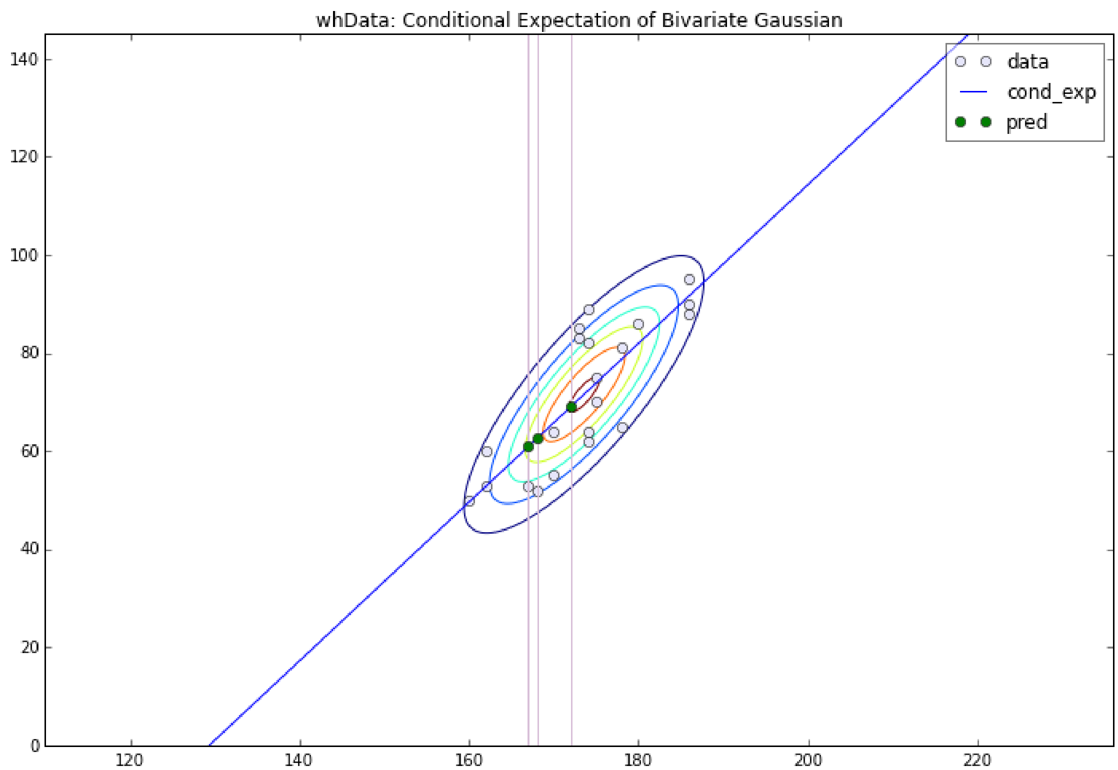
- for the correlation coefficient ρ, one can also use numpy.corrcoef(...)

```
In [7]: def demo_2(x, y, u, title, ddof=None, padding=50):
            XY = np.vstack((x, y))
            res = fit_rex.fit_multivariate_normal_dist(XY, ddof=ddof
                                                        , padding=padding
                                                        , get_pdf=True
                                                        , X_unknown=u
                                                        , X_unknown_dim=0)
            coeff = res[0]
            line = res[1]
            pred = res[2]
            xypdf = res[3]

            preds = []
            preds_string = "{}: "

            preds.append(pred[1])
            preds_string += "{:4.3} |"

            fig = plt.figure(figsize=(12, 8))
            axs = fig.add_subplot(111)

            xmin, ymin = XY.min(axis=1)
            xmax, ymax = XY.max(axis=1)

            xlim = [xmin-padding, xmax+padding]
            ylim = [ymin-padding, ymax+padding]

            # plot data
            plt_rex.plot2d(XY, colwise_data=True,
                           hatch='o', color='lavender',
                           x_lim=xlim, y_lim=ylim,
                           show=False, axs=axs, plotlabel="data",
                          title=title)

            # plot unkown vertical lines
            for h in u:
                axs.axvline(x=h, color='thistle')

            plt_rex.plot2d(np.vstack(line), colwise_data=True,
                          hatch='-',
                          show=False, axs=axs,
                          plotlabel="cond_exp")

            plt_rex.plot2d(np.vstack(pred), colwise_data=True,
                          hatch='o',
                          show=False, axs=axs,
                          plotlabel="pred")

            axs.contour(*xypdf)

            print("Coefficients")
            print("\n".join("{} {}".format(*c) for c
```

```
                    in zip(["mean\n", "covariance\n", "corr\n"], coeff)))
        print()
        print("Predictions")
        print("\n".join(preds_string.format(*p) for p in zip(u, *preds)))
```

```
In [8]: demo_2(hn, wn, hu
               , "whData: Conditional Expectation of Bivariate Gaussian")
```

```
Coefficients
mean
 [ 173.57142857   71.52380952]
covariance
 [[  55.05714286   89.08571429]
 [  89.08571429  219.46190476]]
corr
 0.8104414767618634

Predictions
168.0: 62.5 |
172.0: 69.0 |
167.0: 60.9 |
```

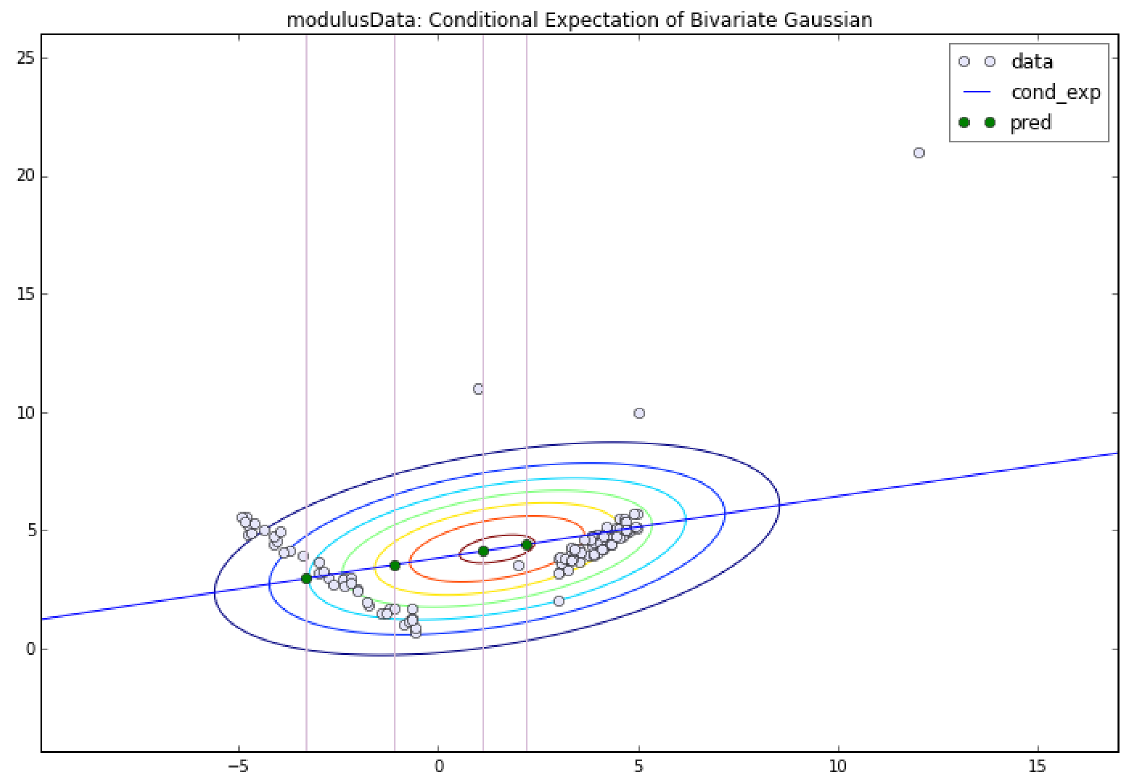


- As expected, the conditional expectation gives a straight line through the mean
- **But**, why is it not cutting through the center evenly?
  - is it because of the *still* biased standard deviations?

1.3.2 for some noisy data for $y = mod(x)$

```
In [9]: demo_2(qx, qy, qu
                , "modulusData: Conditional Expectation of Bivariate Gaussian"
                , padding=5)
```

```
Coefficients
mean
 [ 1.47593543  4.20084416]
covariance
 [[ 12.59813115   3.29784464]
 [  3.29784464   5.10714763]]
corr
 0.41113828870332314

Predictions
1.1:  4.1 |
2.2: 4.39 |
-3.3: 2.95 |
-1.1: 3.53 |
```



1.4 Task 2.3

1.4.1 Bayesian Parameter Estimation

- We tried two methods, using equations from the lecture slides

    1. Find the Maximum A Posteriori Estimate of the coefficients, and use the dot product

2. Use the expectation of the conditional probability of the weights

1.4.2 $\mathbf{w}_{MAP}$ as Regularized Least Squares

We used the equation:

$$\begin{aligned}\mathbf{w}_{MAP} &= argmax_{\mathbf{w}}\, p(\mathbf{w} \mid D) \\ &= \left(\mathbf{X}^T\mathbf{X} + \frac{\sigma^2}{{\sigma_0}^2}\mathbf{I}\right)^{-1} \mathbf{X}^T\mathbf{y}\end{aligned}$$

Which is a case of ***Regularized Least Squares***

- We used `scipy.sparse.linalg.lsmr(...)` for this
  - It is not exactly needed, since the dataset is so small
- $\sigma_0^2 = 3$ was given
- we chose $\sigma^2 = var(\mathbf{y})$
  - similar to how `sklearn.linear_model.BayesianRidge()` does
  - also because this variance will be *corrected* while calculating the parameters for the conditional probability of the weights

1.4.3 Expectation of the conditional probability

Assuming that the observed weights are normally distributed about the polynomial :

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

we used the equations from the slides:

$$\mathbb{E}[y \mid x = x_0, D] = \mu^T \mathbf{x_0}$$

where

$$\begin{aligned}\mu &= \frac{1}{\sigma^2}\mathbf{\Lambda}^{-1}\mathbf{X}^T\mathbf{y} \\ \mathbf{\Lambda} &= \frac{1}{\sigma^2}\mathbf{X}^T\mathbf{X} + \frac{1}{{\sigma_0}^2}\mathbf{I}\end{aligned}$$

Comparing results for degree = 5

```
In [10]: def demo_3(x, y, u, degree, title, padding=50):
             XY = np.vstack((x, y))

             res_man_lsmr = \
                 fit_rex.fit_polynomial_bayesian(x, y, degree
                                                 , sig2=None, sig2_0=3.0
                                                 , use_lsmr=True, use_pinv=False
                                                 , padding=padding, get_pdf=False
                                                 , X_unknown=u)


             res_man_pinv = \
```

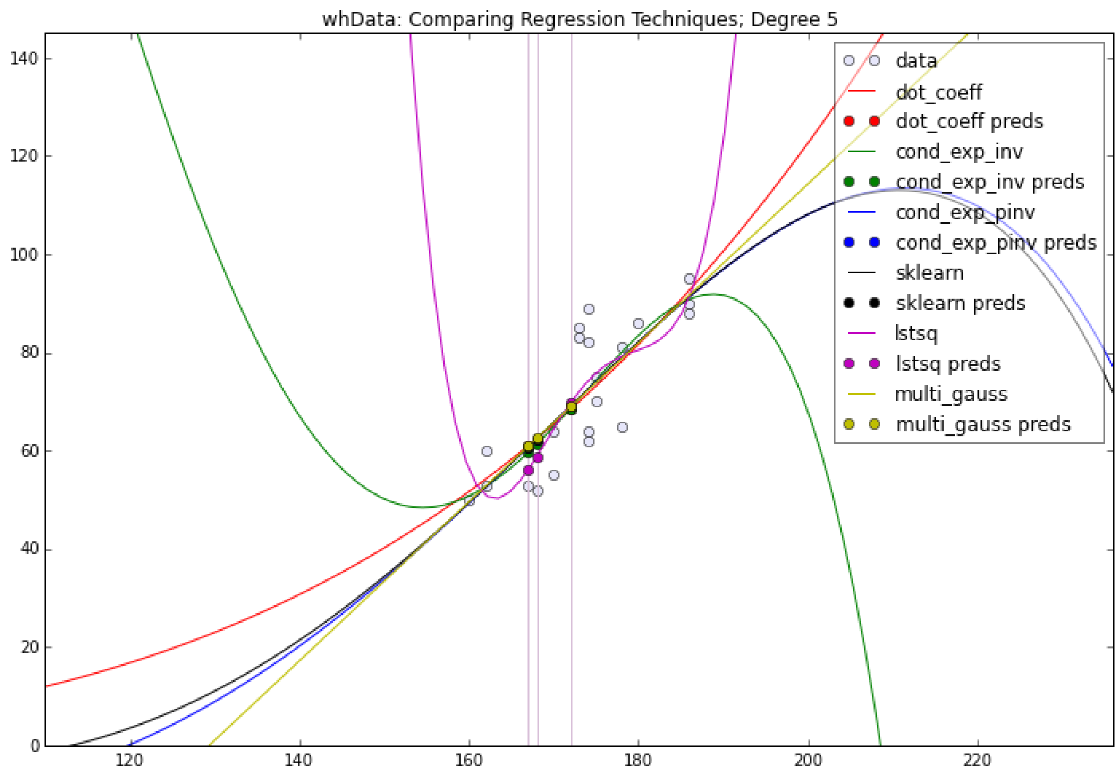
```
    fit_rex.fit_polynomial_bayesian(x, y, degree
                                    , sig2=None, sig2_0=3.0
                                    , use_lsmr=True, use_pinv=True
                                    , padding=padding, get_pdf=False
                                    , X_unknown=u)

res_skl = \
    fit_rex.fit_polynomial_bayesian_skl(x, y, degree
                                        , padding=padding
                                        , X_unknown=u)

res_lstsq = \
    fit_rex.fit_polynomial_nplstsq(x, y, degree
                                   , x_pad=padding
                                   , X_unknown=u)

res_multi_gauss = \
    fit_rex.fit_multivariate_normal_dist(XY, ddof=None
                                         , padding=padding
                                         , get_pdf=False
                                         , X_unknown=u
                                         , X_unknown_dim=0)

labels = [
    "dot_coeff"
    , "cond_exp_inv"
    , "cond_exp_pinv"
    , "sklearn"
    , "lstsq"
    , "multi_gauss"
]

coeffs = [
    res_man_lsmr[0]
    , res_skl[0]
    , res_lstsq[0]
]

lines = [
    res_man_lsmr[1][1]
    , res_man_lsmr[1][0]
    , res_man_pinv[1][0]
    , res_skl[1][0]
    , res_lstsq[1]
    , res_multi_gauss[1]
]

preds = [
    res_man_lsmr[2][1]
    , res_man_lsmr[2][0]
    , res_man_pinv[2][0]
    , res_skl[2][0]
    , res_lstsq[2]
    , res_multi_gauss[2]
```

```
]

predsw = [
    res_man_lsmr[2][1][1]
    , res_man_lsmr[2][0][1]
    , res_man_pinv[2][0][1]
    , res_skl[2][0][1]
    , res_lstsq[2][1]
    , res_multi_gauss[2][1]
]

fig = plt.figure(figsize=(12, 8))
axs = fig.add_subplot(111)

xmin, ymin = XY.min(axis=1)
xmax, ymax = XY.max(axis=1)

xlim = [xmin-padding, xmax+padding]
ylim = [ymin-padding, ymax+padding]

# plot data
plt_rex.plot2d(XY, colwise_data=True,
               hatch='o', color='lavender',
               x_lim=xlim, y_lim=ylim,
               show=False, axs=axs, plotlabel="data",
              title=title)

# plot unkown vertical lines
for h in u:
    axs.axvline(x=h, color='thistle')

colors = ['r', 'g', 'b', 'k', 'm', 'y']
for label, line, pred, c in zip(labels, lines, preds, colors):
    plt_rex.plot2d(np.vstack(line), colwise_data=True,
                   hatch=c+'-',
                   show=False, axs=axs,
                   plotlabel=label)
    plt_rex.plot2d(np.vstack(pred), colwise_data=True,
                   hatch=c+'o',
                   show=False, axs=axs,
                   plotlabel=label+" preds")


coeff_string = "{}: " + "".join("{:5.5} |"
                                for ci in range(len(coeffs)))
preds_string = "{}: " + "".join("{:4.3} |"
                                for pi in range(len(predsw)))

print("\n### DEGREE = {} ###############"\
      "###################\n".format(degree))
print("Coefficients\n" +
      str(["Bayesian inv", "sklearn", "lstsq"]))
print("\n".join(coeff_string.format(i, *c)
                for i, c in enumerate(zip(*coeffs))))
```

```
            print()
            print("Predictions\n" + str(labels))
            print("\n".join(preds_string.format(*p) for p in zip(u, *predsw)))
```

```
In [11]: demo_3(hn, wn, hu, 5
               , "whData: Comparing Regression Techniques; Degree 5")
```

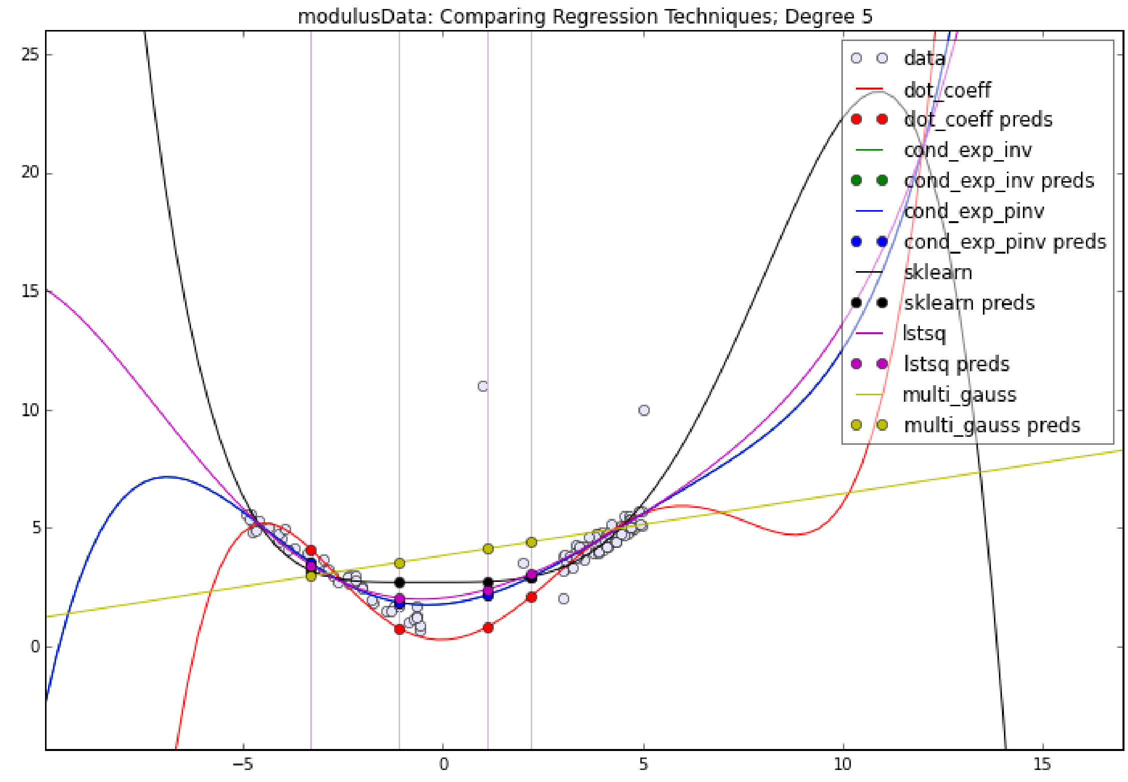
```
### DEGREE = 5 ###################################

Coefficients
['Bayesian inv', 'sklearn', 'lstsq']
0: 4.5666e-16 |15.974 |145.37 |
1: 6.4147e-14 |-8.584e-09 |5006.5 |
2: 8.4604e-12 |-1.4678e-06 |-115.28 |
3: 9.9334e-10 |-0.00012551 |0.99413 |
4: 8.7598e-08 |1.4368e-06 |-0.003806 |
5: -5.4888e-11 |-3.7582e-09 |5.4589e-06 |

Predictions
['dot_coeff', 'cond_exp_inv', 'cond_exp_pinv', 'sklearn', 'lstsq', 'multi_gauss']
168.0: 62.4 |61.3 |62.4 |62.4 |58.8 |62.5 |
172.0: 68.4 |68.6 |69.1 |69.0 |69.7 |69.0 |
167.0: 61.0 |59.6 |60.8 |60.8 |56.1 |60.9 |
```



1.4.4 for some noisy data for $y = mod(x)$

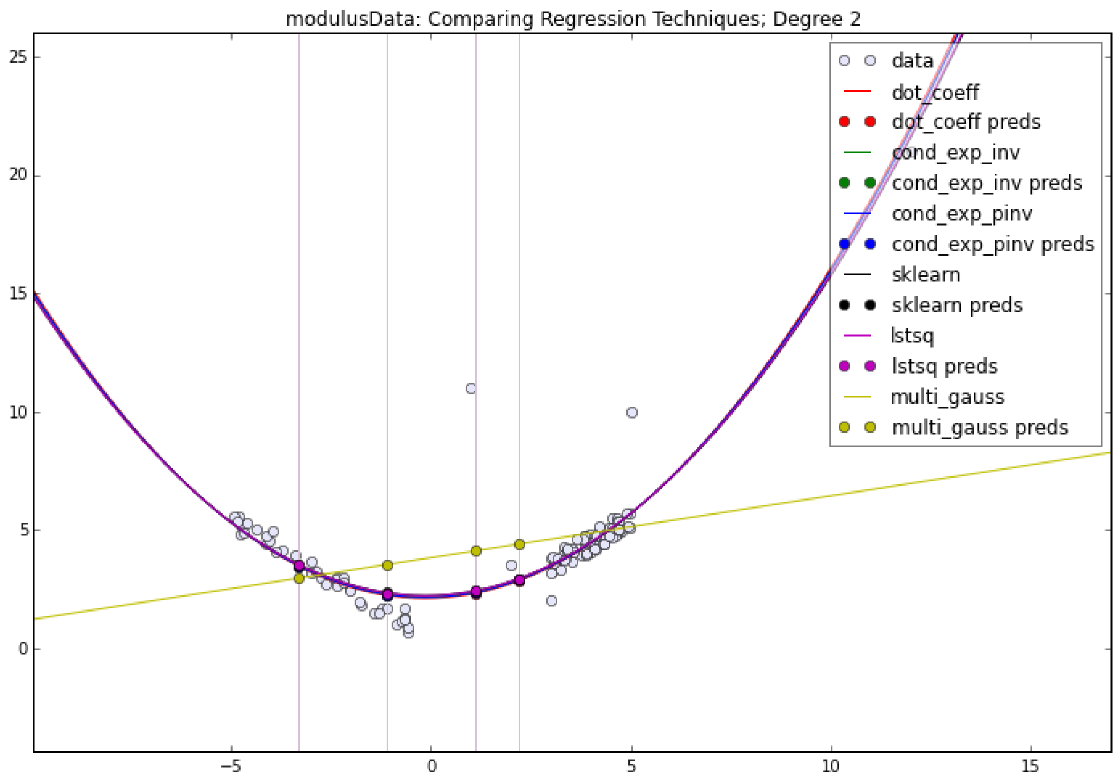
```
In [12]: demo_3(qx, qy, qu, 5
                , "modulusData: Comparing Regression Techniques; Degree 5"
```

```
                , padding=5)

### DEGREE = 5 ##################################

Coefficients
['Bayesian inv', 'sklearn', 'lstsq']
0: 0.25529 |2.6861 |2.0179 |
1: 0.036255 |0.0014729 |0.17171 |
2: 0.43747 |0.0029669 |0.15487 |
3: -0.017005 |0.009939 |-0.0095604 |
4: -0.0096037 |0.0054508 |-0.00031228 |
5: 0.00074687 |-0.00045134 |7.0885e-05 |

Predictions
['dot_coeff', 'cond_exp_inv', 'cond_exp_pinv', 'sklearn', 'lstsq', 'multi_gauss']
1.1: 0.789 |2.15 |2.15 |2.71 |2.38 | 4.1 |
2.2: 2.08 | 2.9 | 2.9 |2.91 |3.04 |4.39 |
-3.3: 4.08 |3.51 |3.51 |3.18 |3.42 |2.95 |
-1.1: 0.752 |1.84 |1.84 |2.68 |2.03 |3.53 |
```




```
In [13]: demo_3(qx, qy, qu, 2
                , "modulusData: Comparing Regression Techniques; Degree 2"
                , padding=5)

### DEGREE = 2 ##################################

Coefficients
```

```
['Bayesian inv', 'sklearn', 'lstsq']
0: 2.0969 |2.2207 |2.2107 |
1: 0.038019 |0.035571 |0.037971 |
2: 0.13585 |0.13152 |0.13196 |

Predictions
['dot_coeff', 'cond_exp_inv', 'cond_exp_pinv', 'sklearn', 'lstsq', 'multi_gauss']
1.1:  2.3 |2.35 |2.35 |2.42 |2.41 | 4.1 |
2.2: 2.84 |2.88 |2.88 |2.94 |2.93 |4.39 |
-3.3: 3.45 |3.48 |3.48 |3.54 |3.52 |2.95 |
-1.1: 2.22 |2.26 |2.26 |2.34 |2.33 |3.53 |
```



1.4.5 Comparing the approaches

- **Ordinary Least Squares**
  - has tendency to overfit with complexity of model
- **conditional expectation of Bivariate Gaussian**
  - Assumes the parameters of the model are fixed, and the samples represent them
  - worked fine for `whData.dat` (maybe), but is not a right model for ***every*** problem

1.4.6 Comparing the approaches

- w_{MAP}

- This is regularized least squares
- still has tendency to overfit the observations when complexity of the model is increased
  * even though it worked quite fine for `whData.dat` (maybe)

- **Bayesian Regression**
  - using `numpy.linalg.inv(...)` or `numpy.linalg.pinv(...)` for Λ^{-1} may or may not coincide with the the results of `sklearn`
    * `sklearn` assumes the priors for σ^2 and ${\sigma_0}^2$ to be gamma distributions
    * `sklearn` results cannot be assumed to be correct, if our model is wrong
  - which one is correct?

$$\mu = \frac{1}{\sigma^2}\mathbf{\Lambda}^{-1}\mathbf{X}^T\mathbf{y}$$

1.5 Task 2.4

1.5.1 Nearest Neighbor Classifier

1.5.2 Objectives

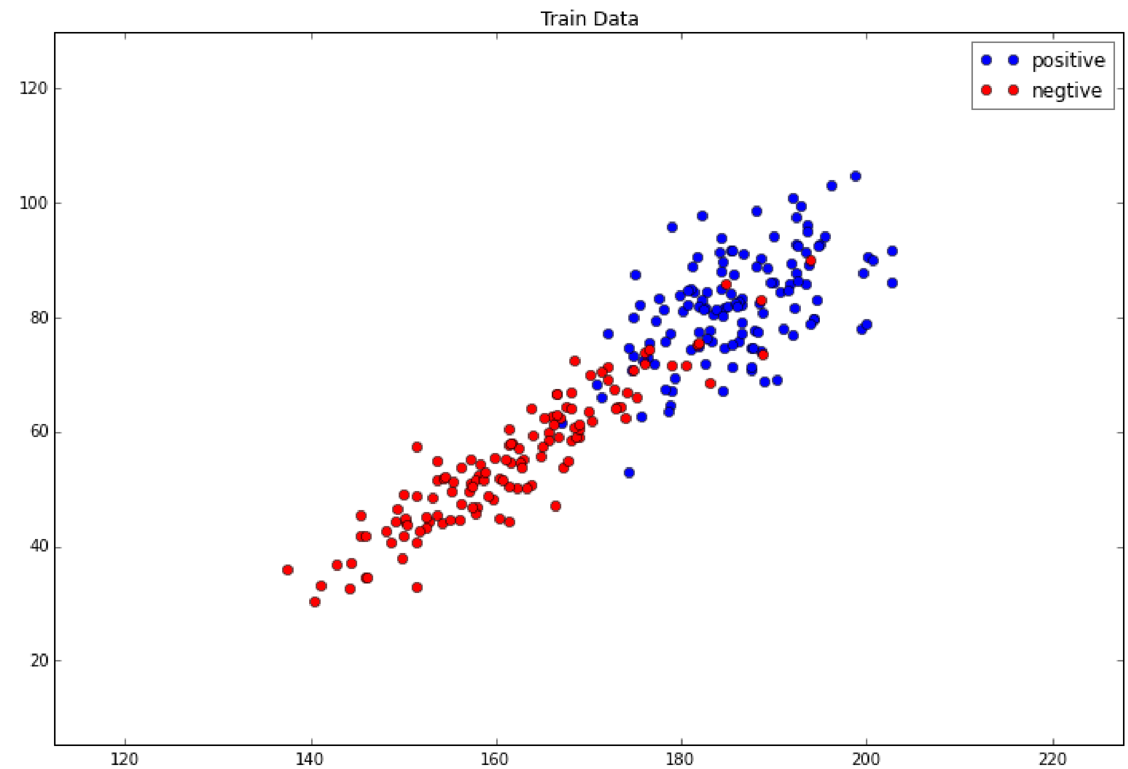
1. Determine the recognition accuracy (percentage of correctly classified data points) of KNN classifier on `K=1,3,5`
2. Determine the overall run time for computing the 1-nearest neighbor of every data in `data2-test.dat`.

```
In [14]: def demo_4_read():
             dt = np.dtype([('x', np.float), ('y', np.float), ('lable', np.float)])  # g is byte-string

             data = np.loadtxt('data/data2-train.dat', dtype=dt, comments='#', delimiter=None)

             x = np.array([d[0] for d in data])#x
             y = np.array([d[1] for d in data])#y
             lable = np.array([d[2] for d in data])#label

             X = np.vstack((x, y, lable))  # data is going to be column-wise
             X.shape

             # split
             X_pos, X_neg = pre_rex.split_data(X, True, 2, [1.0, -1.0])

             print("\n ### TRAIN DATA ####################################\n")
             print("Positive :", X_pos.shape[1], "; Negative :", X_neg.shape[1])

             return (x, y, X, X_pos, X_neg)


         def demo_4_plot(x, y, X, X_pos, X_neg):
             # plotting
             fig = plt.figure(figsize=(12, 8))
             axs = fig.add_subplot(111)

             # limits for the axes
             X_ = np.vstack((x, y))  # only the measurements; data is col-wise
```

```
    xmin, ymin = X_.min(axis=1)
    xmax, ymax = X_.max(axis=1)

    xlim = [xmin-25, xmax+25]  # purely for looks
    ylim = [ymin-25, ymax+25]

    plt_rex.plot2d(X_pos, colwise_data=True, hatch='bo', x_lim=xlim, y_lim=ylim,
                   show=False, axs=axs, set_aspect_equal=False, plotlabel="positive")
    plt_rex.plot2d(X_neg, colwise_data=True, hatch='ro', x_lim=xlim,
                    y_lim=ylim, show=False, axs=axs, set_aspect_equal=False,
                    plotlabel="negtive", title="Train Data")
```

In [15]:
```
res = demo_4_read()
demo_4_plot(*res)
```

```
### TRAIN DATA ####################################

Positive : 128 ; Negative : 128
```

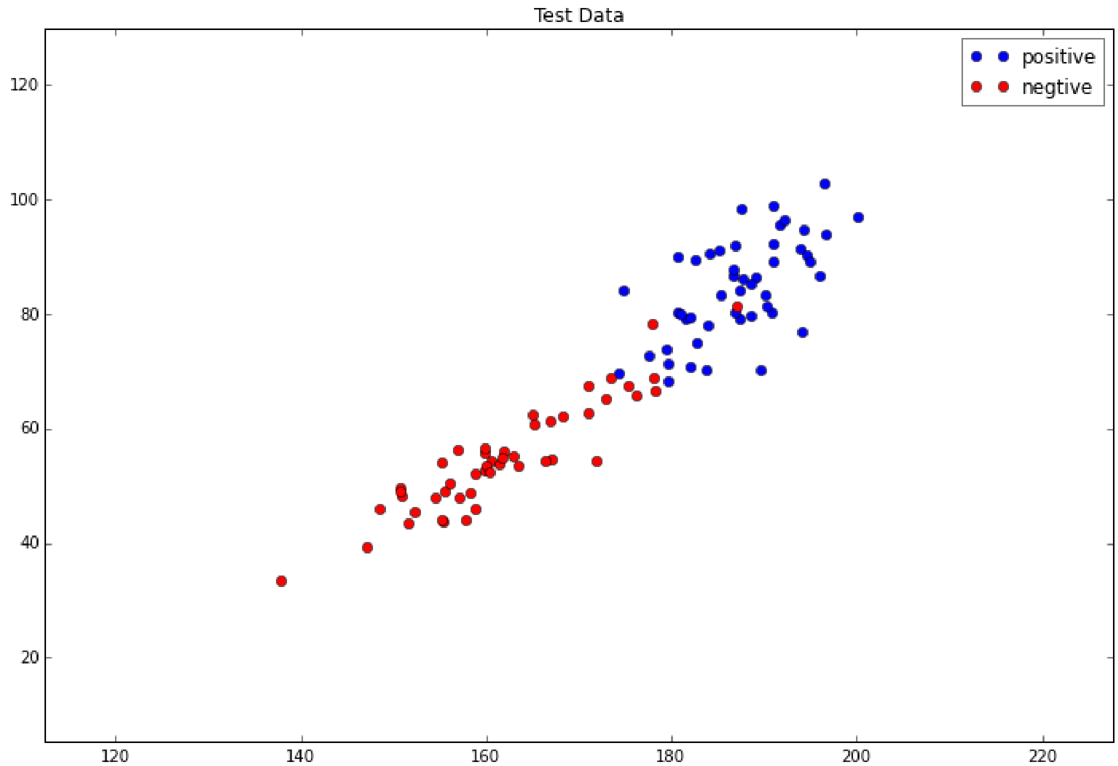


In [16]:
```
def demo_4_read_test():
    dt = np.dtype([('x', np.float), ('y', np.float), ('lable', np.float)])  # g is byte-string

    data = np.loadtxt('data/data2-test.dat', dtype=dt, comments='#', delimiter=None)

    x_t = np.array([d[0] for d in data])#x
    y_t = np.array([d[1] for d in data])#y
    lable_t = np.array([d[2] for d in data])#label
```

```
    X_t = np.vstack((x_t, y_t, lable_t))  # data is going to be column-wise
    X_t.shape

    # split
    X_t_pos, X_t_neg = pre_rex.split_data(X_t, True, 2, [1.0, -1.0])

    print("\n ### TEST DATA ###################################\n")
    print("Positive :", X_t_pos.shape[1], "; Negative :", X_t_neg.shape[1])

    return (x_t, y_t, X_t, X_t_pos, X_t_neg)


def demo_4_plot_test(x, y, x_t, y_t, X_t, X_t_pos, X_t_neg):
    # plotting
    fig = plt.figure(figsize=(12, 8))
    axs = fig.add_subplot(111)

    # limits for the axes
    X_t_ = np.vstack((x, y))  # only the measurements; data is col-wise
    xmin, ymin = X_t_.min(axis=1)
    xmax, ymax = X_t_.max(axis=1)

    xlim = [xmin-25, xmax+25]  # purely for looks
    ylim = [ymin-25, ymax+25]

    plt_rex.plot2d(X_t_pos, colwise_data=True, hatch='bo', x_lim=xlim, y_lim=ylim,
                  show=False, axs=axs, set_aspect_equal=False, plotlabel="positive")
    plt_rex.plot2d(X_t_neg, colwise_data=True, hatch='ro', x_lim=xlim,
                   y_lim=ylim, show=False, axs=axs, set_aspect_equal=False,
                   plotlabel="negtive", title="Test Data")
```

In [17]:
```
res_test = demo_4_read_test()
demo_4_plot_test(res[0], res[1], *res_test)
```

```
### TEST DATA ###################################

Positive : 48 ; Negative : 48
```



1.5.3 Use the nearest neighbor from Train Data to predict

- First write a test function to test 3 differnet implementations
  - **Method 0**: Run the nearest neighbor k times, for small k
  - **Method 1**: Use the method from Pr's paper
  - **Method 2**: Adoption based on Pr's paper, instead of all members, we sort partially

In [18]:
```
def k_nearest_neighbors_smallk(X, q, k):
    inds=nearest_neighbor_method3(X, q)
    a_inds = np.array(inds)
    X=np.delete(X, inds, axis=1)
    for i in range(k-1):
        inds=nearest_neighbor_method2(X, q)
        a_inds=np.append(a_inds,inds)#remember to assign a pointer to new array. the return va
        if i!=k-1:
            X=np.delete(X, inds, axis=1)
    return a_inds


def k_nearest_neighbors(X, q, k):
    X = X.T
    sorted_inds = np.argsort(np.sum((X - q)**2, axis=1))
    return sorted_inds[:k]
```

```
def k_nearest_neighbors_par(X, q, k):
    X=X.T
    sorted_inds = np.argpartition(np.sum((X - q)**2, axis=1), k-1)
    return sorted_inds[:k]


def recoAccurKNN(train, test, k, method):
    hit=0
    for i in range(test.shape[1]):
        if method==0:
            inds = knn_rex.k_nearest_neighbors_smallk(train[0:2,:],test[0:2,i],k)
        elif method==1:
            inds = knn_rex.k_nearest_neighbors(train[0:2,:],test[0:2,i],k)
        elif method==2:
            inds = knn_rex.k_nearest_neighbors_par(train[0:2,:],test[0:2,i],k)
        if np.multiply(np.sum(train[2,inds]),test[2,i])>0:#if the KNN votes is the same sign a
            hit+=1
    return (hit/test.shape[1])
```

```
In [19]: def demo_4_1(X, X_t):
             i=1
             while i <=5:
                 for j in range(3):
                     start = timeit.default_timer()
             #          for k in range(2):
                     reco = recoAccurKNN(X,X_t,i,j)
                     stop = timeit.default_timer()
                     print("Accuracy on k =",i,", method =",j,":","{0:.4f}".format(reco)," Time:",stop-
                 i=i+2
```

Experiment 2.4.1

- Run the Accuracy Test on k = 1,3,5
- Test with 3 methods
- Record the time

```
In [20]: demo_4_1(res[2], res_test[2])
```

```
Accuracy on k = 1 , method = 0 : 0.8854  Time: 0.011000161990523338
Accuracy on k = 1 , method = 1 : 0.8854  Time: 0.007752555015031248
Accuracy on k = 1 , method = 2 : 0.8854  Time: 0.0033649879624135792
Accuracy on k = 3 , method = 0 : 0.8958  Time: 0.017618791025597602
Accuracy on k = 3 , method = 1 : 0.8958  Time: 0.003665815049316734
Accuracy on k = 3 , method = 2 : 0.8958  Time: 0.003170748008415103
Accuracy on k = 5 , method = 0 : 0.9375  Time: 0.027580601978115737
Accuracy on k = 5 , method = 1 : 0.9375  Time: 0.004693805996794254
Accuracy on k = 5 , method = 2 : 0.9375  Time: 0.0038653210503980517
```

Discussion 2.4.1

- In which we can see that method 2 and 3 are nearly good.
- Then we run this test with higher K on method 2 and 3.
- And to get a stable result we repeat 10 times on both method

Experiment 2.4.2

- We run the experiments 100 times on method 2 and 3
- Test with small K up to 25

In [21]:
```
def demo_4_2(X, X_t):
    i=1
    repeat = 100
    while i <=25:
        for j in range(1,3):
            start = timeit.default_timer()
            for k in range(repeat):
                reco = recoAccurKNN(X,X_t,i,j)
            stop = timeit.default_timer()
            print("Accuracy on k =",i,", method =",j,":","{0:.4f}".format(reco)," Time:",stop-s
        i=i+2
```

In [22]:
```
demo_4_2(res[2], res_test[2])
```

```
Accuracy on k = 1 , method = 1 : 0.8854  Time: 0.4786799040157348
Accuracy on k = 1 , method = 2 : 0.8854  Time: 0.4091633160132915
Accuracy on k = 3 , method = 1 : 0.8958  Time: 0.42411490698577836
Accuracy on k = 3 , method = 2 : 0.8958  Time: 0.37382687197532505
Accuracy on k = 5 , method = 1 : 0.9375  Time: 0.4385979769867845
Accuracy on k = 5 , method = 2 : 0.9375  Time: 0.3839304230059497
Accuracy on k = 7 , method = 1 : 0.9167  Time: 0.4322182150208391
Accuracy on k = 7 , method = 2 : 0.9167  Time: 0.39254300604807213
Accuracy on k = 9 , method = 1 : 0.9271  Time: 0.4319147280184552
Accuracy on k = 9 , method = 2 : 0.9271  Time: 0.3983517350279726
Accuracy on k = 11 , method = 1 : 0.9479  Time: 0.4075413689715788
Accuracy on k = 11 , method = 2 : 0.9479  Time: 0.3911596069810912
Accuracy on k = 13 , method = 1 : 0.9583  Time: 0.40552423702320084
Accuracy on k = 13 , method = 2 : 0.9583  Time: 0.38304321200121194
Accuracy on k = 15 , method = 1 : 0.9479  Time: 0.4085477130138315
Accuracy on k = 15 , method = 2 : 0.9479  Time: 0.37633531901519746
Accuracy on k = 17 , method = 1 : 0.9583  Time: 0.4034845369751565
Accuracy on k = 17 , method = 2 : 0.9583  Time: 0.36949757498223335
Accuracy on k = 19 , method = 1 : 0.9479  Time: 0.40806648100260645
Accuracy on k = 19 , method = 2 : 0.9479  Time: 0.3729934219736606
Accuracy on k = 21 , method = 1 : 0.9583  Time: 0.4069304330041632
Accuracy on k = 21 , method = 2 : 0.9583  Time: 0.3856181330047548
Accuracy on k = 23 , method = 1 : 0.9583  Time: 0.4144851809833199
Accuracy on k = 23 , method = 2 : 0.9583  Time: 0.36859023600118235
Accuracy on k = 25 , method = 1 : 0.9583  Time: 0.4107820210047066
Accuracy on k = 25 , method = 2 : 0.9583  Time: 0.375577119004447
```

Discussion 2.4.2

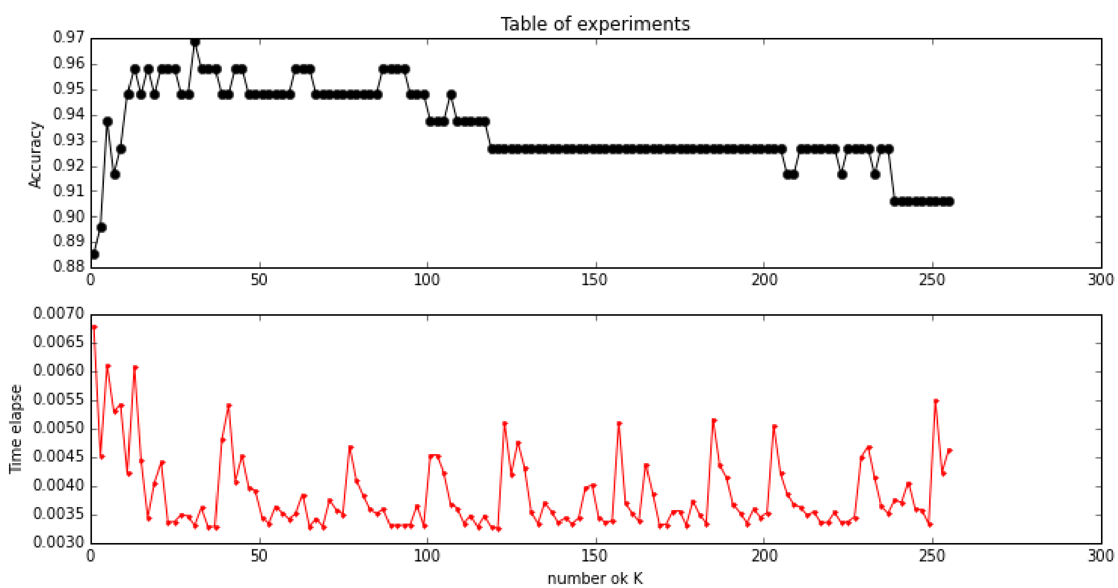
- On average, when K is small, Partial sort is better than sort.
- Observed that accuracy stablized at around 95%

Experiment 2.4.3

- Use method 3 to run k up to the limit

```
In [23]: def demo_4_3(X, X_t):
             i=int(1)
             recoList=[]
             kList=[]
             tList=[]
             while i <=255:
                 start = timeit.default_timer()
                 reco = recoAccurKNN(X,X_t,i,2)
                 stop = timeit.default_timer()
                 recoList.append(reco)
                 kList.append(i)
                 tList.append(stop-start)
                 i=int(i+2)

             fig = plt.figure(figsize=(12, 6))
             kArr=np.asarray(kList)
             recoArr=np.asarray(recoList)
             tArr=np.asarray(tList)

             plt.subplot(2, 1, 1)
             plt.plot(kArr, recoArr, 'ko-')
             plt.title('Table of experiments')
             plt.ylabel('Accuracy')

             plt.subplot(2, 1, 2)
             plt.plot(kArr, tArr, 'r.-')
             plt.xlabel('number ok K')
             plt.ylabel('Time elapse')

             plt.show()
```

```
In [24]: demo_4_3(res[2], res_test[2])
```

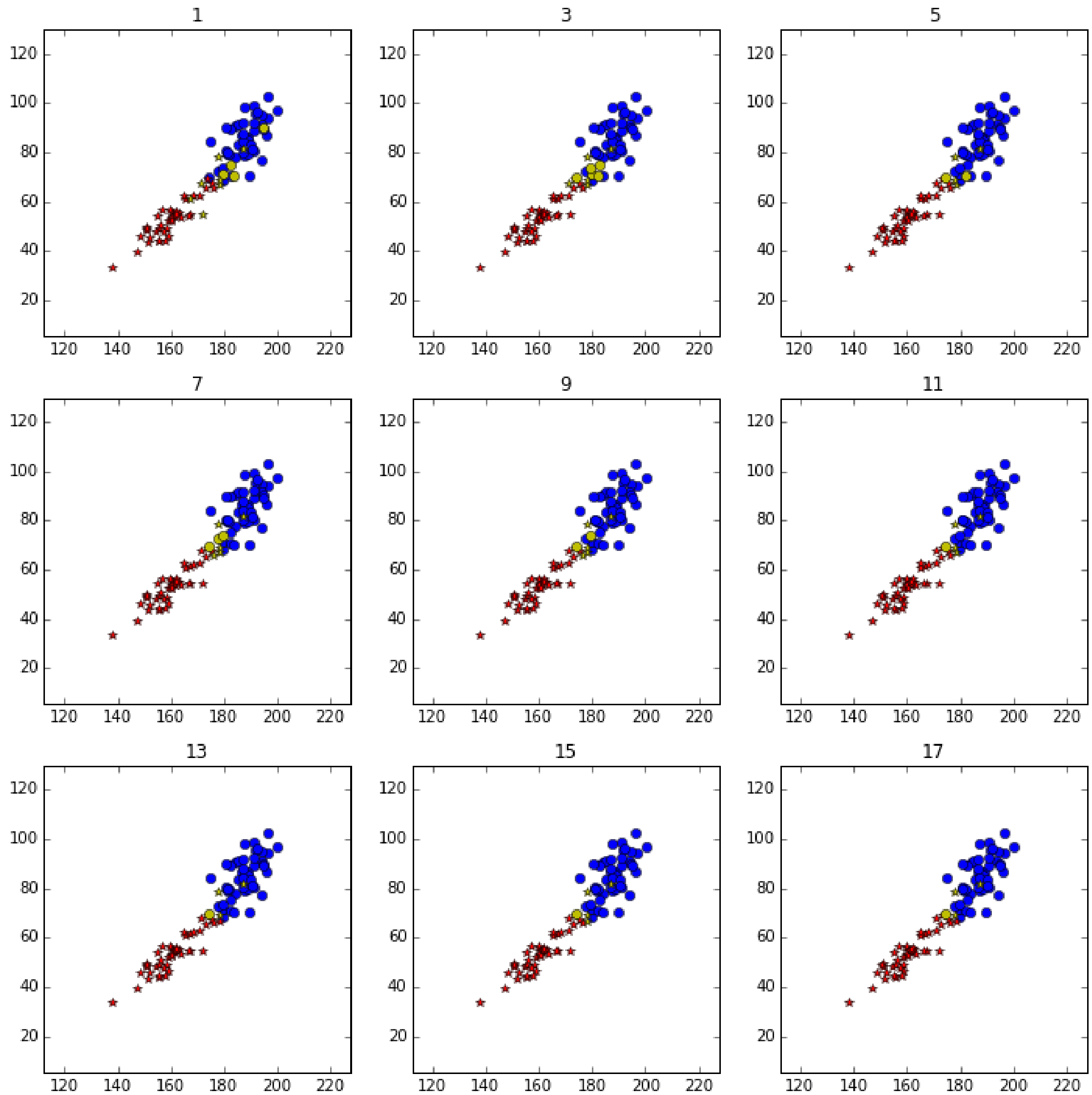


```
In [25]: def preparePlotKNN(train, test, k):
             ret = np.copy(test)
             for i in range(test.shape[1]):
                 inds = knn_rex.k_nearest_neighbors_par(train[0:2,:],test[0:2,i],k)
                 if np.multiply(np.sum(train[2,inds]),test[2,i])< 0:#if the KNN votes is not the same s
                     if(test[2,i]<0):
                         ret[2,i]=-2 #-2, -1 is judged as 1
                     else:
                         ret[2,i]=2
             return ret


         def demo_4_plots2gether(X_t, x, y, X, X_pos, X_neg):
             # plotting
             fig = plt.figure(figsize=(12, 12))

             # limits for the axes
             X_ = np.vstack((x, y))  # only the measurements; data is col-wise
             xmin, ymin = X_.min(axis=1)
             xmax, ymax = X_.max(axis=1)
             xlim = [xmin-25, xmax+25]  # purely for looks
             ylim = [ymin-25, ymax+25]

             k_num = 9 # 1,3,5
             KNNPlotList=[]
             axsList=[]
             for i in range(k_num):
                 KNNPlotList.append(preparePlotKNN(X,X_t,i*2+1))
                 axsList.append(fig.add_subplot(3,3,i+1))
                 Temp = pre_rex.split_data(KNNPlotList[i], True, 2, [1.0, -1.0, 2.0, -2.0])
                 plt_rex.plot2d(Temp[0], colwise_data=True, hatch='bo', x_lim=xlim, y_lim=ylim,
                         show=False, axs=axsList[i], set_aspect_equal=False)
                 plt_rex.plot2d(Temp[1], colwise_data=True, hatch='r*', x_lim=xlim, y_lim=ylim,
                         show=False, axs=axsList[i], set_aspect_equal=False)
                 plt_rex.plot2d(Temp[2], colwise_data=True, hatch='yo', x_lim=xlim, y_lim=ylim,
                         show=False, axs=axsList[i], set_aspect_equal=False)
                 plt_rex.plot2d(Temp[3], colwise_data=True, hatch='y*', x_lim=xlim, y_lim=ylim,
                         show=False, axs=axsList[i], set_aspect_equal=False, title=2*i+1)
```

1.5.4 Plot together

- We plot the prediction result and the real result together
- The Blue circle (0) represents data with label 1
- The Red star (*) represents data label -1
- The Yellow circle (0) represents data with label 1, with prediction fail
- The Yellow star (*) represents data label -1, with prediction fail

```
In [26]: demo_4_plots2gether(res_test[2], *res)
```



1.5.5 2. Determin the Run Time on Nearest Neighbor

First define the test function

- We have 4 methods
  - Näive approch
  - From Prof's paper, use the norm
  - From Prof's paper
  - From Prof's paper

```
In [27]: def nearest_neighbor_method0(X,q):
             m, n = X.shape
             sqr = np.square(np.subtract(X.T,q))# (X-q)^2
             _sum = np.add(sqr[:,0],sqr[:,1]) #sum up the x and y
             return np.argmin(_sum) # retun the argmin
```

```
def nearest_neighbor_method1(X, q):
    m, n = X.shape
    minindx = 0
    mindist = np.inf
    for i in range(n):
        dist = la.norm(X[:,i] - q)
        if dist <= mindist:
            mindist = dist
            minindx = i
    return minindx

def nearest_neighbor_method2(X, q):
    m, n = X.shape
    return np.argmin(np.sum((X-q.reshape(m,1))**2, axis=0))

def nearest_neighbor_method3(X, q):
    X = X.T
    return np.argmin(np.sum((X - q)**2, axis=1))
```

In [28]:
```
def test1NNtime(data, method):
#     start = timeit.default_timer()
    for i in range(data.shape[1]):
        data_=np.delete(data, i, axis=1)##Delete itself
        if method==0:
            start = timeit.default_timer()
            inds = knn_rex.nearest_neighbor_method1(data_[0:2,:],data[0:2,i])
            stop = timeit.default_timer()
        elif method==1:
            start = timeit.default_timer()
            inds = knn_rex.nearest_neighbor_method1(data_[0:2,:],data[0:2,i])
            stop = timeit.default_timer()
        elif method==2:
            start = timeit.default_timer()
            inds = knn_rex.nearest_neighbor_method2(data_[0:2,:],data[0:2,i])
            stop = timeit.default_timer()
        elif method==3:
            start = timeit.default_timer()
            inds = knn_rex.nearest_neighbor_method3(data_[0:2,:],data[0:2,i])
            stop = timeit.default_timer()
#     stop = timeit.default_timer()
    return (stop-start)


def demo_4_test1NNtime(X_t):
    for i in range(4):
        print("Time for method ",i, " :","{0:.15f}".format(test1NNtime(X_t,i)))
```

In [29]:
```
demo_4_test1NNtime(res_test[2])
```

```
Time for method  0  : 0.001013032975607
Time for method  1  : 0.001025281031616
Time for method  2  : 0.000018177030142
Time for method  3  : 0.000031941977795
```

1.6 Task 2.5

1.6.1 Computing KDTree

1.6.2 1 : Construct and plot k = 2 KDTree for `data2-train.dat`

- 2 Variants on selecting slicing dimension

  1. alternate between the x and the y dimension
  2. split the data along the dimension of higher variance

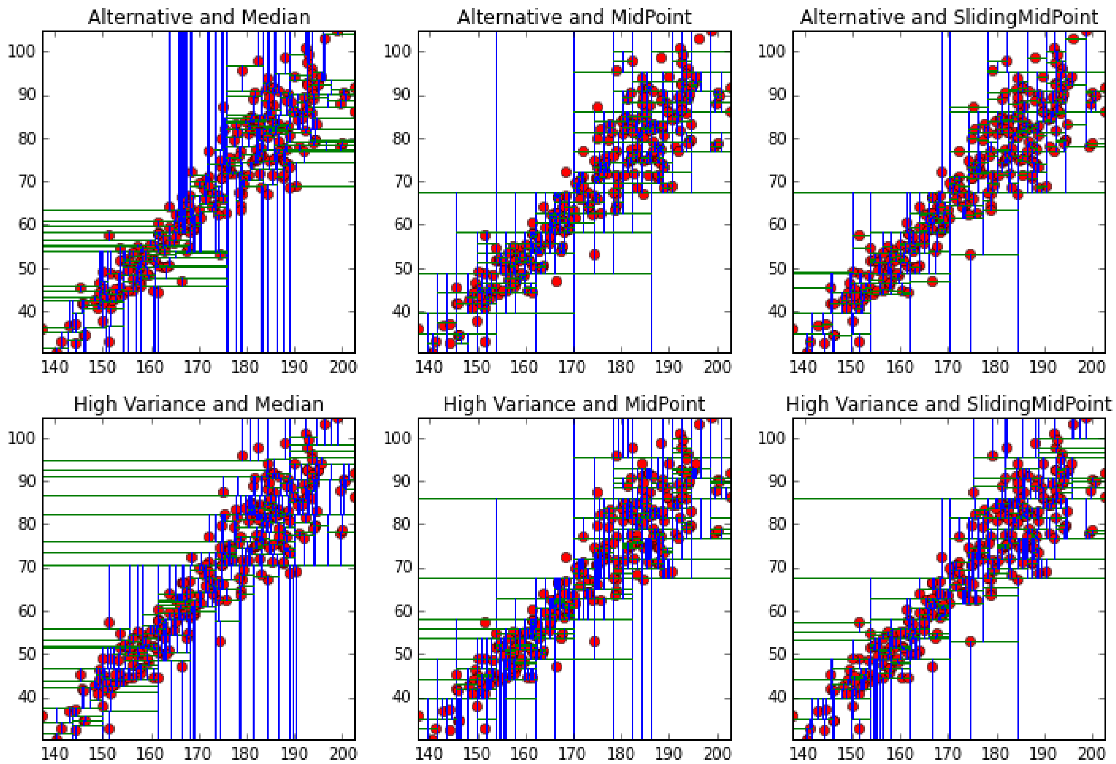
- 3 Variants on computing sliping points

  1. split at the median of the data
  2. split at the midpoint of the data
  3. split at the midpoint of the data with sliding (S. Maneewongvatana and David M. Mount, 1999)

In [30]:
```
def demo_5_read():
    dt = np.dtype([('x', np.float), ('y', np.float), ('lable', np.float)])  # g is byte-string

    data = np.loadtxt('data/data2-train.dat', dtype=dt, comments='#', delimiter=None)

    x = np.array([d[0] for d in data])#x
    y = np.array([d[1] for d in data])#y
    lable = np.array([d[2] for d in data])#label

    X = np.vstack((x, y, lable))  # data is going to be column-wise
    X.shape
    X_pos, X_neg = pre_rex.split_data(X, True, 2, [1.0, -1.0])
    print("Positive :", X_pos.shape[1], "; Negative :", X_neg.shape[1])

    return(x, y, X, X_pos, X_neg)

def demo_5_construct(x, y, X, X_pos, X_neg):
    TreeList=[]
    dim, splt = 2,3
    for i in range(dim):
        for j in range(splt):
            #print("dim,splt",dim, splt)
            TreeList.append(kd_rex.KDTree(X[0:2,:].T,0,i,j))

    kd_rex.KDTreePlot2D(x,y,TreeList,dim,splt)
```

In [31]:
```
res = demo_5_read()
demo_5_construct(*res)
```

```
Positive : 128 ; Negative : 128
```




```
In [32]: def demo_5_read_test():
             #data2-test
             dt = np.dtype([('x', np.float), ('y', np.float), ('lable', np.float)])  # g is byte-string
             data = np.loadtxt('data/data2-test.dat', dtype=dt, comments='#', delimiter=None)

             x_t = np.array([d[0] for d in data])#x
             y_t = np.array([d[1] for d in data])#y
             lable_t = np.array([d[2] for d in data])#label

             X_t = np.vstack((x_t, y_t, lable_t))  # data is going to be column-wise
             X_t.shape

             # split
             X_t_pos, X_t_neg = pre_rex.split_data(X_t, True, 2, [1.0, -1.0])
             print("Positive :", X_t_pos.shape[1], "; Negative :", X_t_neg.shape[1])

             return (x_t, y_t, X_t, X_t_pos, X_t_neg)

         def demo_5_time(X, x_t, y_t, X_t, X_t_pos, X_t_neg):
             tree = ss.KDTree(X[0:2,:].T,4)
             k=1

             start = timeit.default_timer()
             tree.query(X_t[0:2,:].T,k)
             end = timeit.default_timer()

             print("Overall run time:", end-start, "seconds")
```

1.6.3 2: Determine overall run time

For computing the 1-nearest neighbor of every data in `data2-test.dat`

```
In [33]: res_test = demo_5_read_test()
         print()
         demo_5_time(res[2], *res_test)
```

```
Positive : 48 ; Negative : 48

Overall run time: 0.02661741100018844 seconds
```

1.7 References

- Lecture Slides
- `docs.scipy.org`
- C.Bauckhage, "NumPy / SciPy Recipes for Data Science: Regularized Least Squares Optimization", researchgate.net
- C.Bauckhage, "NumPy / SciPy Recipes for Data Science: Computing Nearest Neighbors", researchgate.net
- S. Maneewongvatana and David M. Mount, 1999
- It's okay to be skinny, if your friends are fat: Sliding Midpoint

---

2 Thanks