CS5489 - Machine Learning

Lecture 5b - Supervised Learning - Regression

Dr. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

Outline

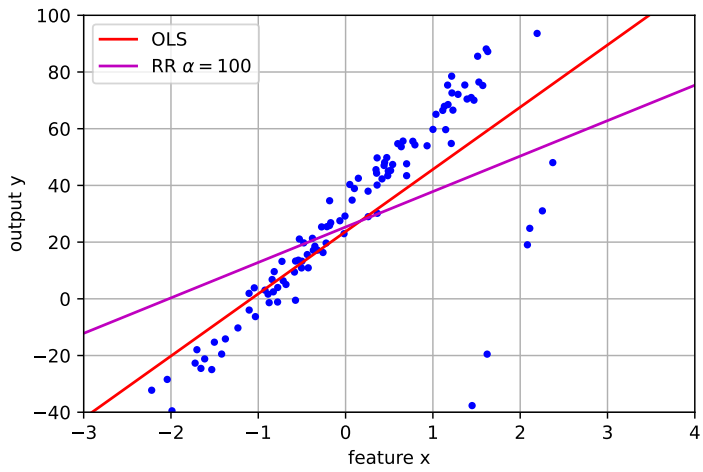
1. Linear Regression
2. Selecting Features
3. **Removing Outliers**
4. Non-linear regression

Outliers

- Too many outliers in the data can affect the squared-error term.
  - regression function will try to reduce the large prediction error for outliers, at the expense of worse prediction for other points

In [3]:
```
outfig
```

Out[3]:



RANSAC

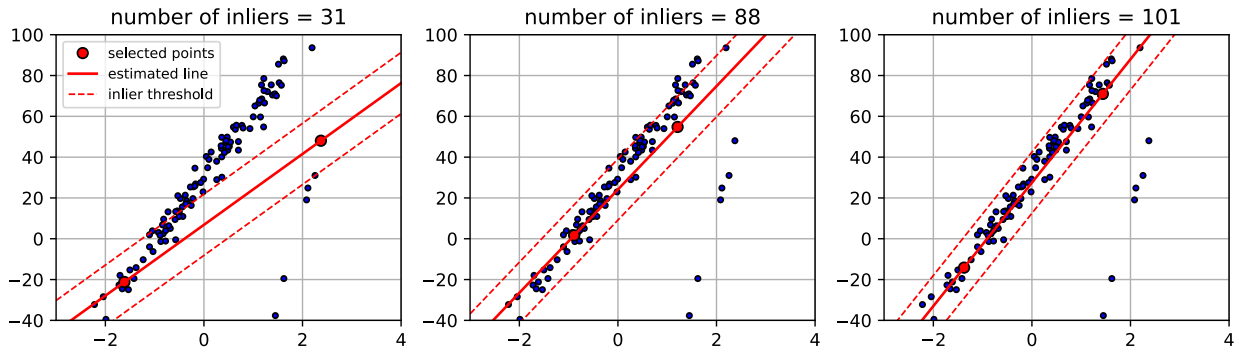
- **RAN**dom **SA**mple **C**onsensus
  - attempt to robustly fit a regression model in the presence of corrupted data (outliers).
  - works with any regression model.
- **Idea:**
  - split the data into inliers (good data) and outliers (bad data).
  - learn the model only from the inliers

Random sampling

- Repeat many times with random subset of points (usually just enough to fit the model)
  - fit a model to the subset.
  - classify all data as inlier or outlier by calculating the residuals (prediction errors) and comparing to a threshold. The set of inliers is called the *consensus set*.
    - save the model with the highest number of inliers.
- Use the largest consensus set to learn the final model.

In [5]:
```
ransacfig
```

Out[5]:



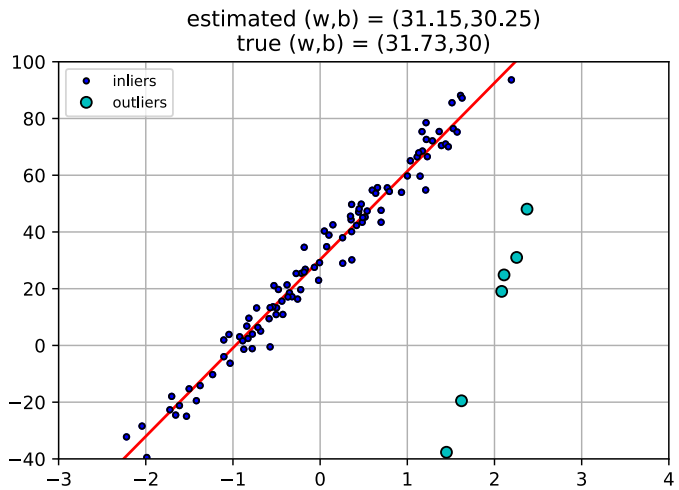
RANSAC

- More iterations increases the probability of finding the correct function.
  - higher probability to select a subset of points contains all inliers.
- Threshold typically set as the median absolute deviation of y.

In [7]:
```
# use RANSAC model (defaults to linear regression)
rlin = linear_model.RANSACRegressor(random_state=1234)
rlin.fit(outlinX, outlinY)

inlier_mask = rlin.inlier_mask_
outlier_mask = logical_not(inlier_mask)
```

In [9]:
```
rfig
```

Out[9]:



Non-linear regression

- So far we have only considered linear regression: $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$
- Similar to classifcation, we can do non-linear regression by forming a feature vector of $\mathbf{x}$ and then performing linear regression on the feature vector.

Polynomial regression

- p-th order Polynomial function
  - $f(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \cdots + w_p x^p$
- Collect the terms into a vector
  - $f(x) = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_p \end{bmatrix} * \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^p \end{bmatrix} = \mathbf{w}^T\phi(x)$
  - weight vector $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_p \end{bmatrix}$; polynomial feature vector: $\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^p \end{bmatrix}$
- Now it's a linear function, so we can use the same linear regression!

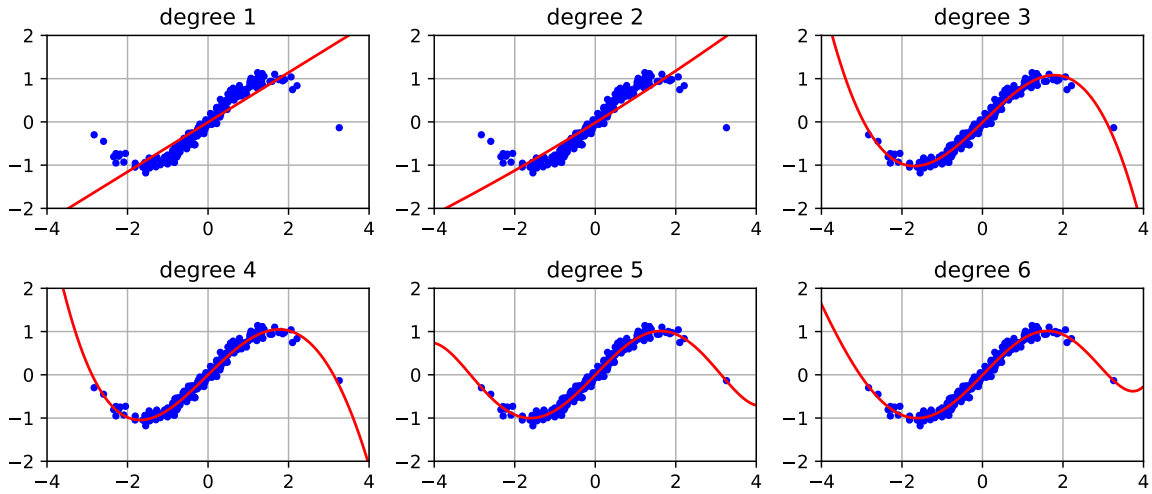
Example

- 1st to 6th order polynomials

In [10]:
```
# example data
polyX = random.normal(size=200)
polyY = sin(polyX) + 0.1*random.normal(size=200)
polyX = polyX[:,newaxis]

plin = {}
for d in [1,2,3,4,5,6]:
    # extract polynomial features with degree d
    polyfeats = preprocessing.PolynomialFeatures(degree=d)
    polyXf = polyfeats.fit_transform(polyX)

    # fit the parameters
    plin[d] = linear_model.LinearRegression()
    plin[d].fit(polyXf, polyY)
```

In [12]:
```
polyfig
```

Out[12]:



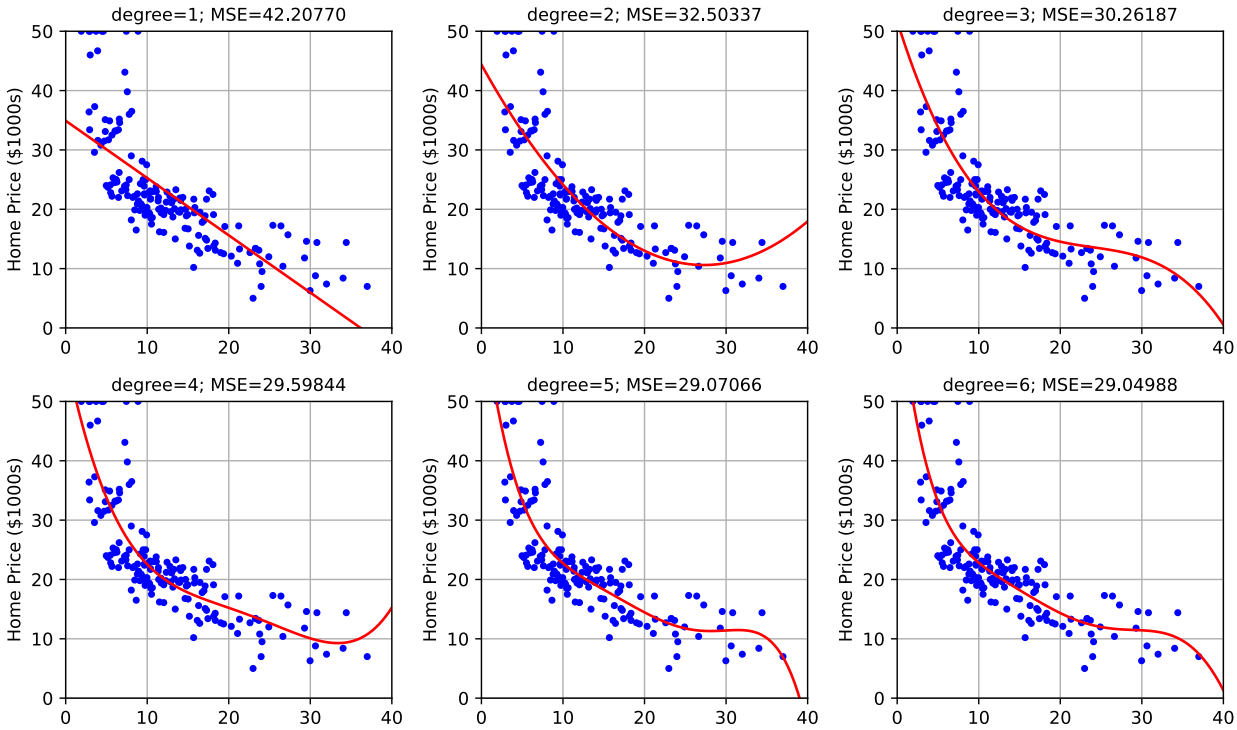
Example: Boston data

- Using "percentage of lower-status" feature
- Increasing polynomial degree d will decrease MSE of training data
  - more complicated model always fits data better
  - (but it could overfit)

In [14]:
```
polyfeats = {}
plin = {}
MSE = {}
for d in [1,2,3,4,5,6]:
    # extract polynomial features with degree d
    polyfeats[d] = preprocessing.PolynomialFeatures(degree=d)
    bostonXf = polyfeats[d].fit_transform(bostonX)

    # fit the parameters
    plin[d] = linear_model.LinearRegression()
    plin[d].fit(bostonXf, bostonY)

    # calculate mean-square error on training set
    MSE[d] = metrics.mean_squared_error(bostonY, plin[d].predict(bostonXf))
```

In [16]:
```
pfig
```

Out[16]:



Select degree using Cross-Validation

- Minimizing the MSE on the training set will overfit
  - More complex function always has lower MSE on training set
- Use cross-validation to select the proper model
  - the parameters we want to change are in feature transformation step
  - Use `pipeline` to merge all steps into one object for easier cross-validation

- `pipeline` object
  - pass an array of stages
  - each entry is a tuple with the stage name and transformer (implements `.fit`, `.transform`)
  - the last entry should be a model (implements `.fit`)

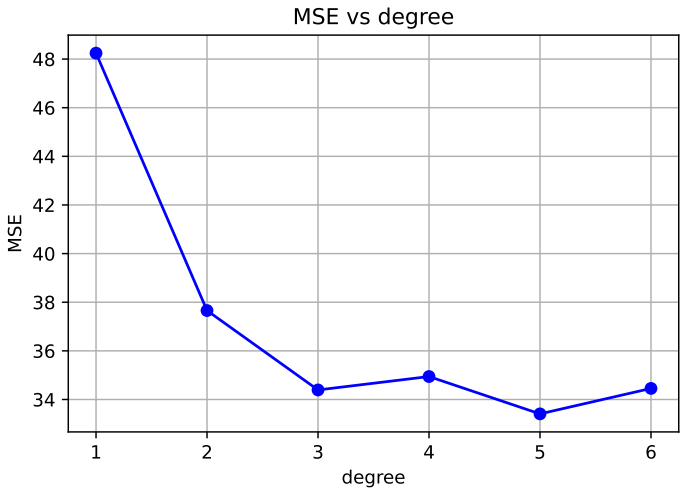
In [17]:
```
# make the pipeline
polylin = pipeline.Pipeline([
        ('polyfeats', preprocessing.PolynomialFeatures(degree=1)),
        ('linreg',    linear_model.LinearRegression())
    ])
```

In [18]:
```
# set the parameters for grid search
# the parameters in each stage are named: <stage>__<parameter>
paramgrid = {
    "polyfeats__degree": array([1, 2, 3, 4, 5, 6]),
}

# do the cross-validddation search - use -MSE as the score for maximizing
plincv = model_selection.GridSearchCV(polylin, paramgrid, cv=5, n_jobs=-1,
                                      scoring='neg_mean_squared_error')
plincv.fit(bostonX, bostonY)

print(plincv.best_params_)
```

```
{'polyfeats__degree': 5}
```

In [20]:
```
avgscores,pnames,bestind = extract_grid_scores(plincv, paramgrid)
plt.figure()
plt.plot(paramgrid['polyfeats__degree'], -avgscores, 'bo-')
```

```
plt.xlabel('degree'); plt.ylabel('MSE'); plt.grid(True);
plt.title('MSE vs degree');
```



Polynomial features: 2D Example

- 2D feature vectors:
  - $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T$
- degree 2 polynomial transformation:

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 & x_1 x_2 & x_2^2 \end{bmatrix}^T$$

- degree 3 polynomial transformation:

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^3 & x_1^2 x_2 & x_1 x_2^2 & x_3^3 \end{bmatrix}^T$$

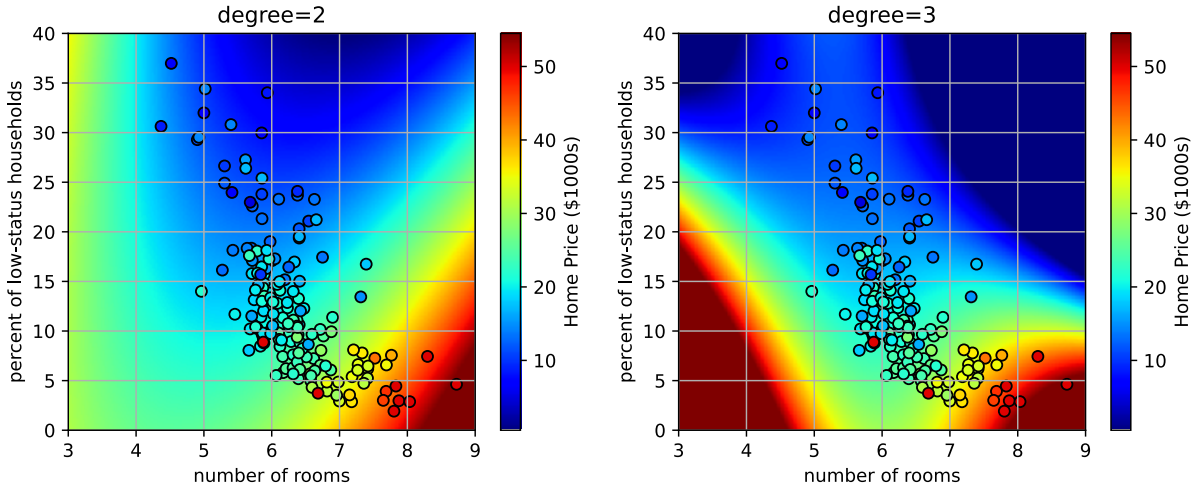
In [23]:

```
plin = {}
polyfeats = {}
for i,d in enumerate([2,3]):
    # get polynomial features
    polyfeats[d] = preprocessing.PolynomialFeatures(degree=d)
    bostonXf = polyfeats[d].fit_transform(bostonX)

    # learn with both dimensions
    plin[d] = linear_model.LinearRegression()
    plin[d].fit(bostonXf, bostonY)

    # calculate MSE
    MSE = metrics.mean_squared_error(bostonY, plin[d].predict(bostonXf))
```

In [25]:

```
pfig
```

Out[25]:



Kernel Ridge Regression

- Apply *kernel trick* to ridge regression
  - turn linear regression into non-linear regression
  - use kernel $k(x, x')$
- Closed form solution:
  - for an input point $\mathbf{x}_*$,
    - prediction: $y_* = \mathbf{k}_*^T(\mathbf{K} + \alpha I)^{-1}\mathbf{y}$
    - $\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}$ is the kernel matrix $(N \times N)$
    - $\mathbf{k}_* = [k(\mathbf{x}_1, \mathbf{x}_*), \cdots, k(\mathbf{x}_N, \mathbf{x}_*)]^T$ is vector containing the kernel values between $\mathbf{x}_*$ and all training points $\mathbf{x}_i$.

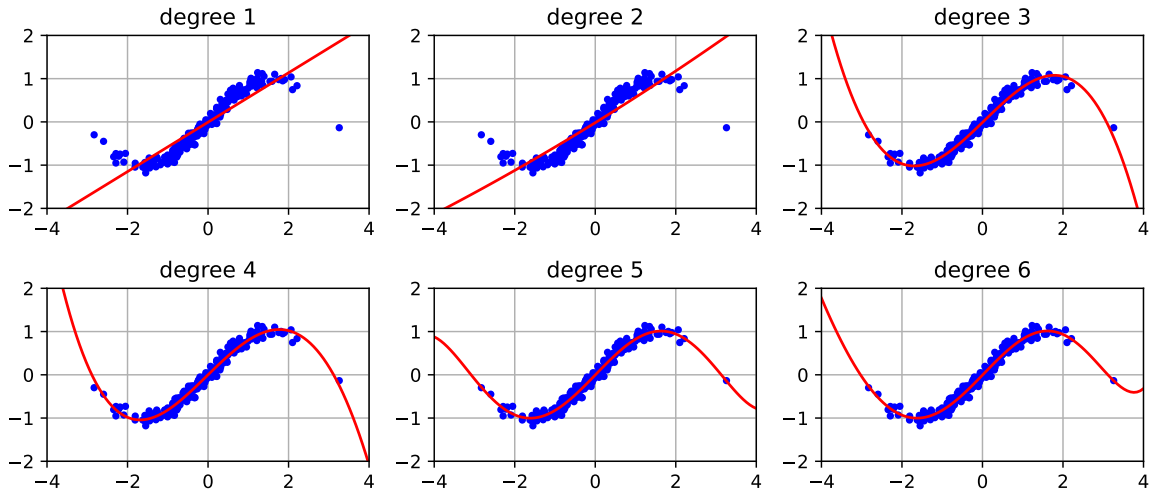
Example: Polynomial Kernel

- Note: it's the same as using polynomial features and linear regression!
  - Using the kernel, we don't need to explicitly calculate the polynomial features.
  - But, we do need to calculate the kernel function between all pairs of training points.

In [26]:
```
krr = {}
for d in [1,2,3,4,5,6]:
    # fit the parameters
    krr[d] = kernel_ridge.KernelRidge(alpha=1, kernel='poly', degree=d)
    krr[d].fit(polyX, polyY)
```

In [28]:
```
krrfig
```

Out[28]:



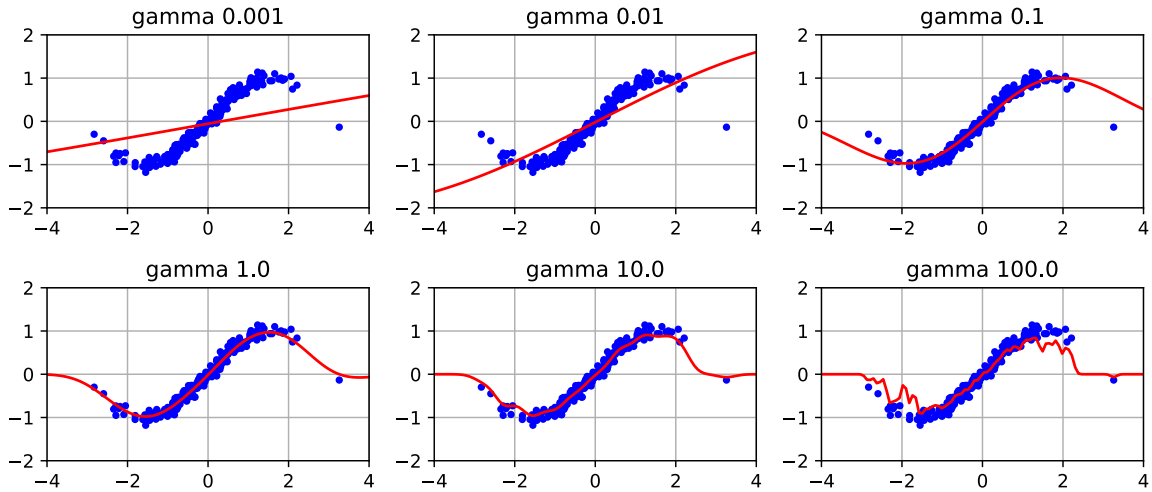
Example: RBF kernel

- gamma controls the smoothness
  - small gamma will etimate a smooth function
  - large gamma will estimate a wiggly function

In [29]:
```
krr = {}
for i,g in enumerate(logspace(-3,2,6)):
    # fit the parameters
    krr[i] = kernel_ridge.KernelRidge(alpha=1, kernel='rbf', gamma=g)
    krr[i].fit(polyX, polyY)
```

In [31]:
```
krrfig
```

Out[31]:



Boston Data: Cross-validation

- RBF kernel
  - cross-validation to select α and γ.

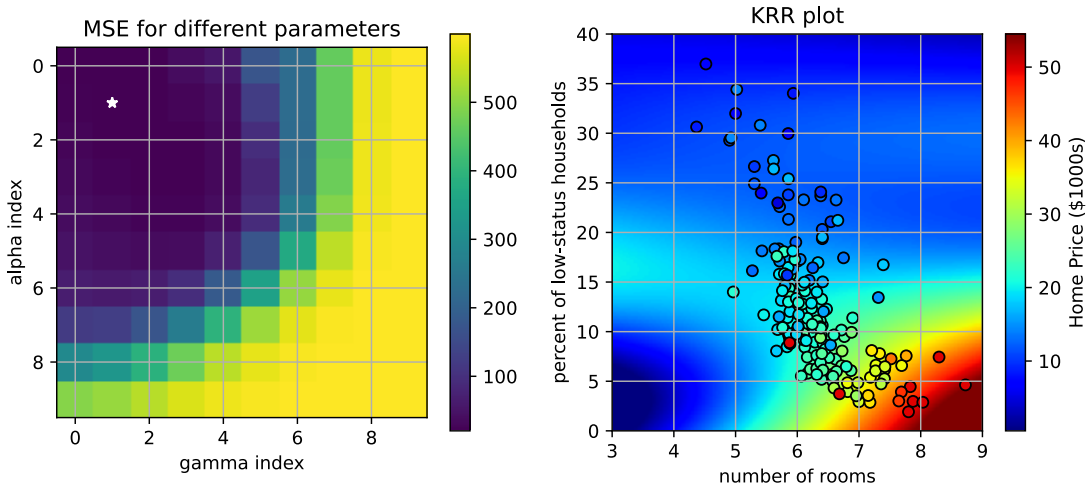
In [32]:
```
# parameters for cross-validation
paramgrid = {'alpha': logspace(-3,3,10),
             'gamma': logspace(-3,3,10)}

# do cross-validation
krrcv = model_selection.GridSearchCV(
  kernel_ridge.KernelRidge(kernel='rbf'),  # estimator
  paramgrid,                               # parameters to try
```

```
  scoring='neg_mean_squared_error',       # score function
  cv=5,                                   # number of folds
  n_jobs=-1, verbose=True)
krrcv.fit(bostonX, bostonY)

print(krrcv.best_score_)
print(krrcv.best_params_)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:    0.1s
-20.36359530141061
{'alpha': 0.004641588833612777, 'gamma': 0.004641588833612777}
[Parallel(n_jobs=-1)]: Done 440 tasks      | elapsed:    0.4s
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:    0.5s finished
```

In [34]:
```
kfig
```

Out[34]:



Gaussian Process Regression

- *Gaussian Process* is an infinite collection of r.v.s where any finite subset of r.v.s is joint Gaussian distributed.
  - infinite collection of values -> function
  - GP is prior distribution over **functions**.
- Denoted as: $f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$
  - function value: $f(\mathbf{x})$ is a distribution of f at location $\mathbf{x}$.
  - mean function: $m(\mathbf{x})$ is the mean function. Usually $m(\mathbf{x}) = c$, a constant.
  - covariance function: $\text{cov}(f(\mathbf{x}), f(\mathbf{x}')) = k(\mathbf{x}, \mathbf{x}')$
    - covariance of function values depends on inputs through the kernel k.

- For any $(x_1, \cdots, x_N)$, the distribution of function values is:

$$f_1, \cdots, f_N | \mathbf{x}_N, \cdots, \mathbf{x}_N \sim \mathcal{N}(0, \mathbf{K})$$

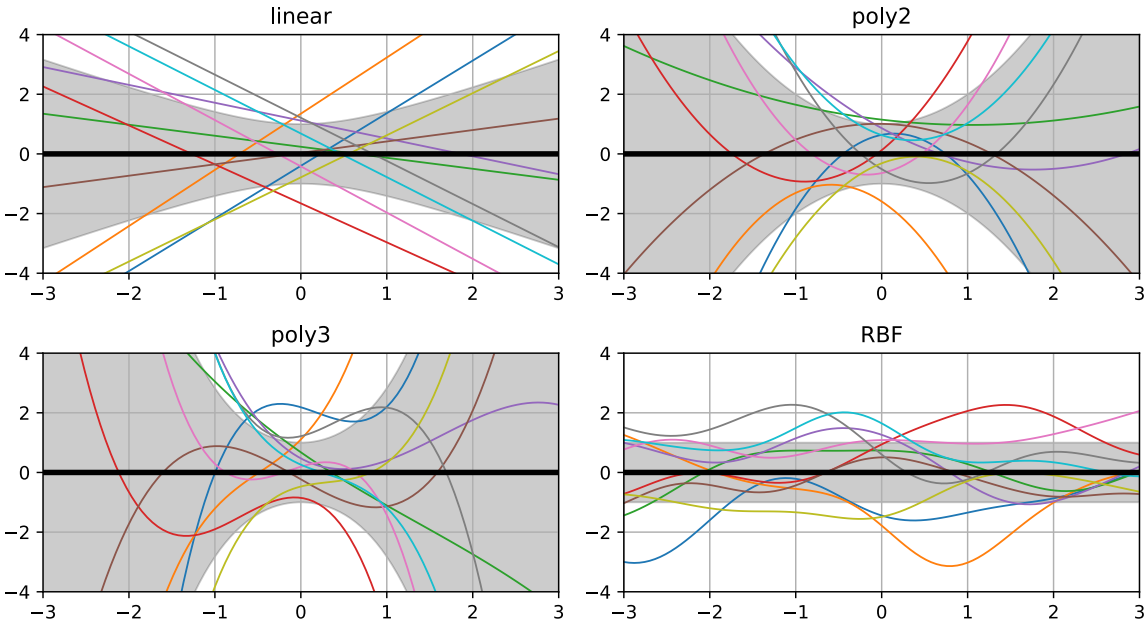
  - $\mathbf{K}$ is the kernel matrix for points $\{\mathbf{x}_N, \cdots, \mathbf{x}_N\}$

Examples of GP priors

- the kernel defines the types of functions that are regressed

In [36]:
```
pfig
```

Out[36]:



Gaussian Process Regression

- Model framework
  - observation noise: $p(\mathbf{y}|\mathbf{f}) = \mathcal{N}(\mathbf{y}|\mathbf{f}, \sigma^2\mathbf{I})$
    - equivalent to mean-squared error loss
  - function prior: $\mathbf{f} \sim \mathcal{GP}(0, k(\mathbf{x}, \mathbf{x}'))$
- Training: given dataset $\{\mathbf{X}, \mathbf{y}\}$
  - compute the posterior distribution of the function values for the observed data:
    - $p(\mathbf{f}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{f})p(\mathbf{f})}{p(\mathbf{y}|\mathbf{X})}$
- Inference: given a new point $\mathbf{x}_*$
  - $p(f_*|x_*, \mathbf{X}, \mathbf{y}) = \int p(f_*|x_*, \mathbf{f})p(\mathbf{f}|\mathbf{X}, \mathbf{y})d\mathbf{f}$
    - averages the predictions over all probable function values $\mathbf{f}$.

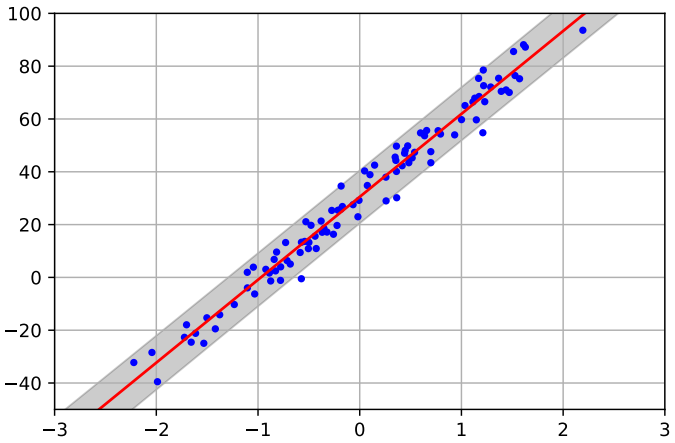
GP Prediction

- All distributions are Gaussian, so there is a closed-form solution.
- The predictive distribution is Gaussian:
  - $p(f_*|x_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(f_*|\mu_*, \sigma_*^2)$
    - mean of prediction: $\mu_* = \mathbf{k}_*^T(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{y}$
    - variance of predictions: $\sigma_*^2 = k_{**} - \mathbf{k}_*^T(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{k}_*$
      - where $k_{**} = k(\mathbf{x}_*, \mathbf{x}_*)$.
  - The uncertainty of the prediction is measured with its variance.
    - (higher values means more uncertain).

- GPR with a linear kernel is equivalent to Bayesian linear regression
  - `DotProduct()` - linear kernel: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}' + \alpha_1$
  - `WhiteKernel()` - observation noise (σ^2): $k(\mathbf{x}, \mathbf{x}') = \sigma^2\delta(\mathbf{x} - \mathbf{x}')$
  - gray area shows 2 standard deviations around the mean (95% confidence region)

In [38]:
```
from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
```

```
k = DotProduct() + WhiteKernel()
gpr = gaussian_process.GaussianProcessRegressor(kernel=k,
                            random_state=5489, normalize_y=True)
# normalize_y: normalize Y to mean 0, var 1 (GPR will be better behaved)
gpr.fit(linX, linY)
plot_regr_trans_1d(gpr, laxbox, linX, linY)
```



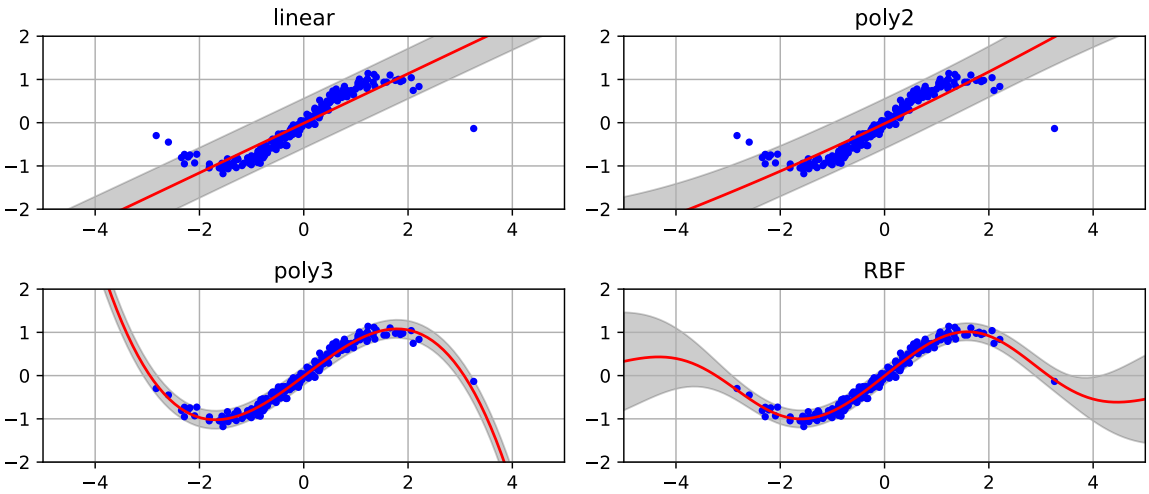
Non-linear regression using kernels

- kernels functions allow non-linear regression
  - `DotProduct()` - linear: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}' + \alpha_1$
  - `DotProduct()**2` - 2nd order polynomial: $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}' + \alpha_1)^2$
  - `DotProduct()**3` - 3rd order polynomial: $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}' + \alpha_1)^3$
  - `RBF()` - Radial-basis function: $k(\mathbf{x}, \mathbf{x}') = \exp(-\frac{1}{2\alpha_2^2}||\mathbf{x} - \mathbf{x}'||^2)$
- Applying the kernel trick to Bayesian linear regression will yield GPR

In [40]:
```
from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel, RBF
kernels = [  DotProduct()    + WhiteKernel(),
             DotProduct()**2 + WhiteKernel(),
             DotProduct()**3 + WhiteKernel(),
             RBF()           + WhiteKernel()   ]
gpr = {}
for i,k in enumerate(kernels):
    gpr[i] = gaussian_process.GaussianProcessRegressor(kernel=k, random_state=0, normalize_y=True)
    gpr[i].fit(polyX, polyY)
```

In [42]:
```
gprfig
```

Out[42]:



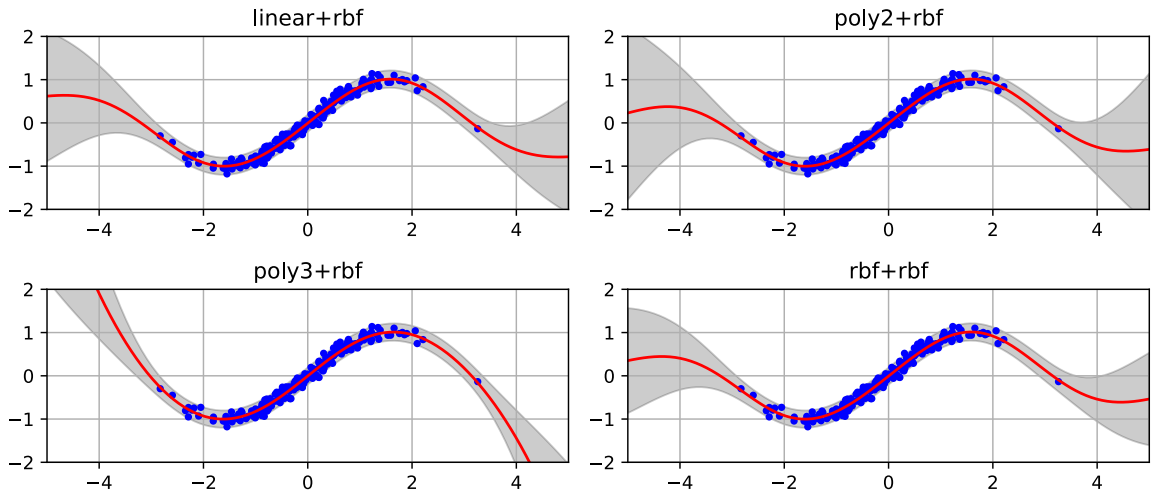
- kernels can be summed, multiplied, and exponentiated to make new kernels

- e.g., `RBF() + DotProduct()**2 + WhiteKernel()`
- regressed function is a sum of quadratic and RBF functions

In [43]:
```
kernels = [  DotProduct()    + RBF() + WhiteKernel(),
             DotProduct()**2 + RBF() + WhiteKernel(),
             DotProduct()**3 + RBF() + WhiteKernel(),
             RBF(length_scale=0.01) + RBF() + WhiteKernel()]
gpr = {}
for i,k in enumerate(kernels):
    gpr[i] = gaussian_process.GaussianProcessRegressor(kernel=k, random_state=0, normalize_y=True)
    gpr[i].fit(polyX, polyY)
```

In [45]:
```
gprfig
```

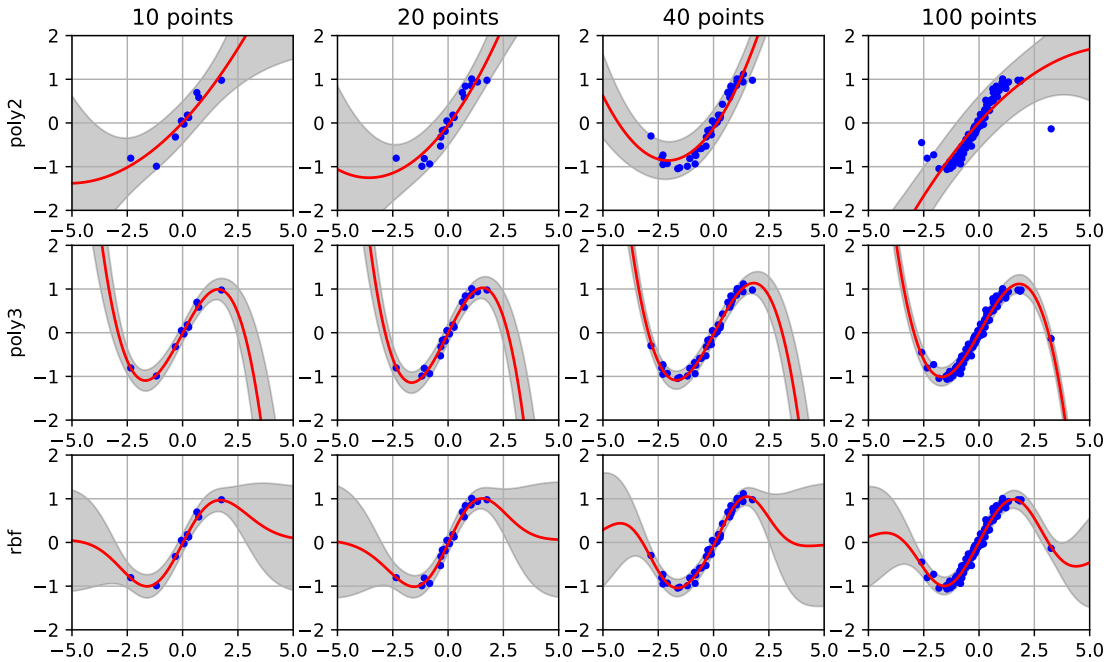
Out[45]:



- As a Bayesian method, GPR handles lack of data well
  - the uncertainty (stddev of the prediction) increases when less data is available.

In [47]:
```
sfig
```

Out[47]:

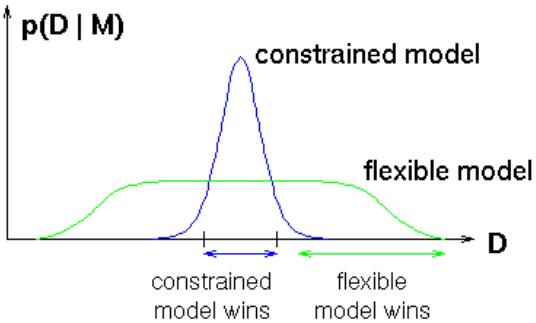


Estimation of Kernel Hyperparameters

- the hyperparameters of the kernel ($\alpha_1, \alpha_2, \sigma^2$, etc.) are estimated by maximizing the marginal likelihood:
  - marginal likelihood (aka model evidience)
    - $p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{X}, \mathbf{f}) p(\mathbf{f}) d\mathbf{f}$
    - averages over all probable functions
  - Use iterative methods to maximize
    - $\alpha^* = \text{argmax}_\alpha \log p(\mathbf{y}|\mathbf{X})$
- Advantages:
  - typically more efficient than grid-search when there are many kernel hyperparameters.
  - principled approach to model selection
- Disadvantage:
  - difficult optimization problem, possibly many local maximum

Intuition of MML

- Consider the space of datasets D.
  - A *constrained (simple)* model can only represent a few datasets.
    - the likelihood of data $p(D|M)$ for those datasets should be large, since it integrates to 1
  - A *flexible* (complex)* model can represent many datasets.
    - the likelihood of data $p(D|M)$ for those datasets should be small.
  - For a given D, by choosing the model with highest $p(D|M)$, we select the least complex model that fits the data.



Example on Boston

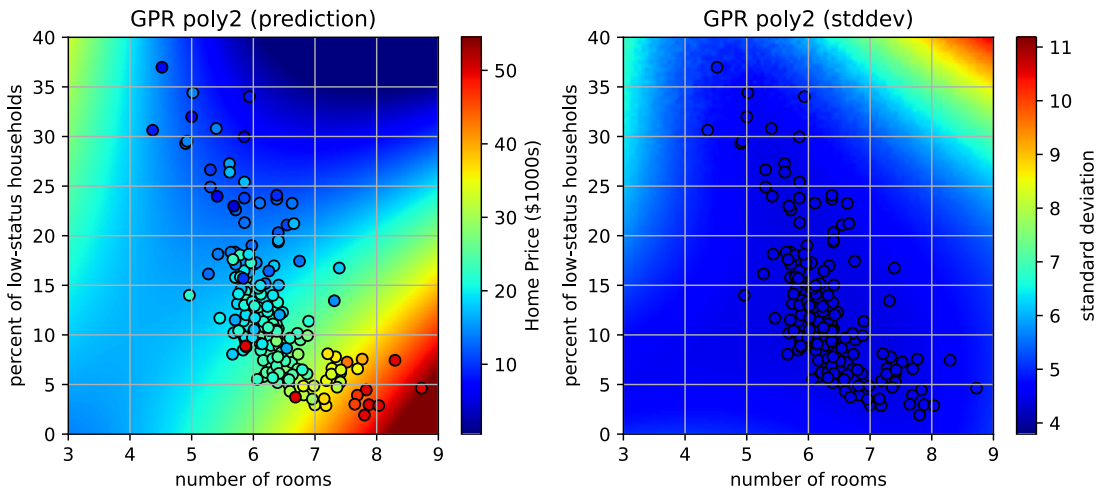
- try 2nd-order polynomial and RBF kernels.

In [48]:
```
kernels = [DotProduct()**2 + WhiteKernel(),
           RBF()           + WhiteKernel()]
kernelnames = ['poly2', 'RBF']

bostonYs  = bostonY/std(bostonY) # rescale outputs

gpr = [None]*2
for i,k in enumerate(kernels):
    gpr[i] = gaussian_process.GaussianProcessRegressor(
        kernel=k, random_state=5489,
        n_restarts_optimizer=5, normalize_y=True)
    gpr[i].fit(bostonX, bostonY)
```

- 2nd order polynomial kernel
  - stddev of prediction shows when the model is not confident

In [50]:
```
gprfig[0]
```

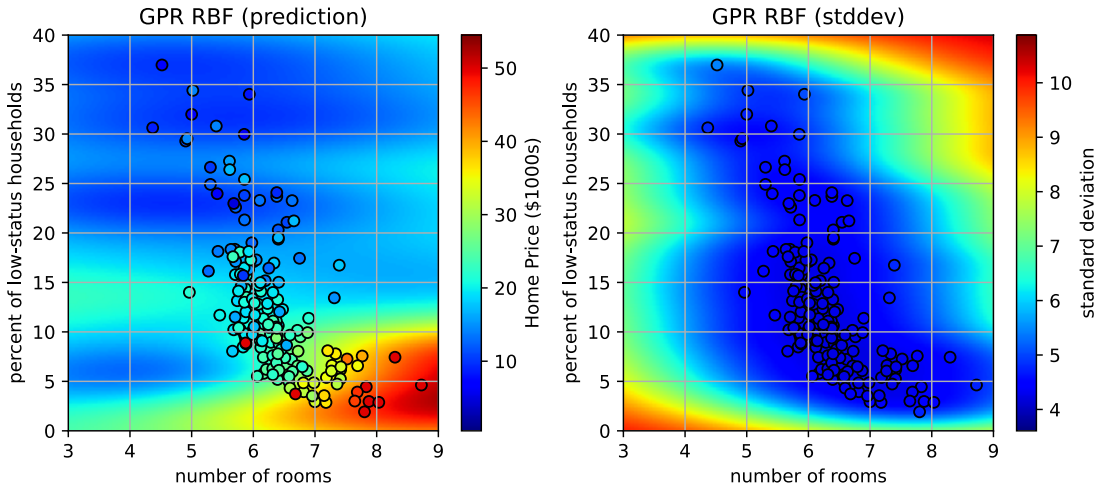
Out[50]:



- RBF kernel
  - stddev of prediction shows when the model is not confident
  - Since the RBF kernel has finite extent, it is not confident where it doesn't see data.

In [51]:
```
gprfig[1]
```

Out[51]:

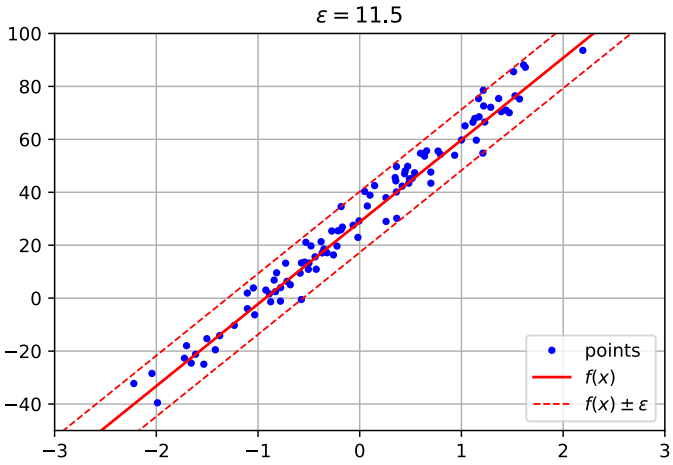


Support Vector Regression (SVR)

- Borrow ideas from classification
  - Suppose we form a "band" of width ϵ around the function:
    - if a point is inside, then it is "correctly" predicted
    - if a point is outside, then it is incorrectly predicted

In [54]:
```
svrfig
```

Out[54]:



- Allow some points to be outside the "tube".
  - penalty of point outside tube is controlled by C parameter.
- SVR objective function:

$$\min_{\mathbf{w},b} \sum_{i=1}^{N} \left| y_i - (\mathbf{w}^T \mathbf{x}_i + b) \right|_\epsilon + \frac{1}{C} ||\mathbf{w}||^2$$

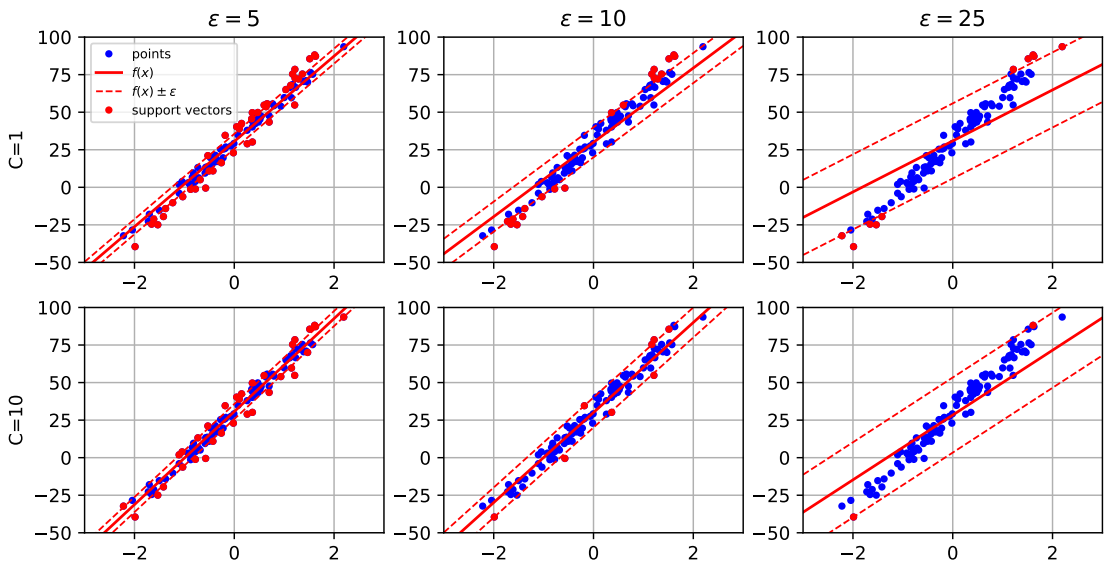
  - find the least complex function with most points inside the tube.
- epsilon-insensitive error:
  - $|z|_\epsilon = \begin{cases} 0, & |z| \leq \epsilon \\ |z| - \epsilon, & |z| > \epsilon \end{cases}$
- Similar to SVM classifier, the points on the band will be the *support vectors* that define the function.

Different tube widths

- The points on/outside the tube are the *support vectors*.

In [56]:
```
svrfig
```

Out[56]:



Kernel SVR

- Support vector regression can also be kernelized similar to SVM

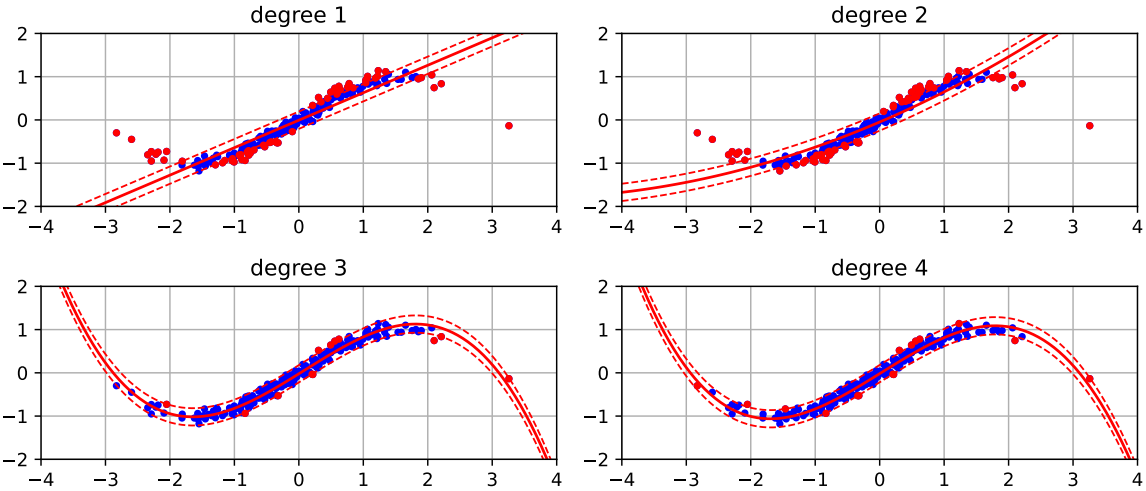
- turn linear regression to non-linear regression

- Polynomial Kernel:

In [57]:
```
epsilon = 0.2
svr = {}
for d in [1,2,3,4]:
    # fit the parameters (poly SVR)
    svr[d] = svm.SVR(C=1000, kernel='poly', coef0=0.1, degree=d, epsilon=epsilon)
    svr[d].fit(polyX, polyY)
```

In [59]:
```
svrfig
```

Out[59]:

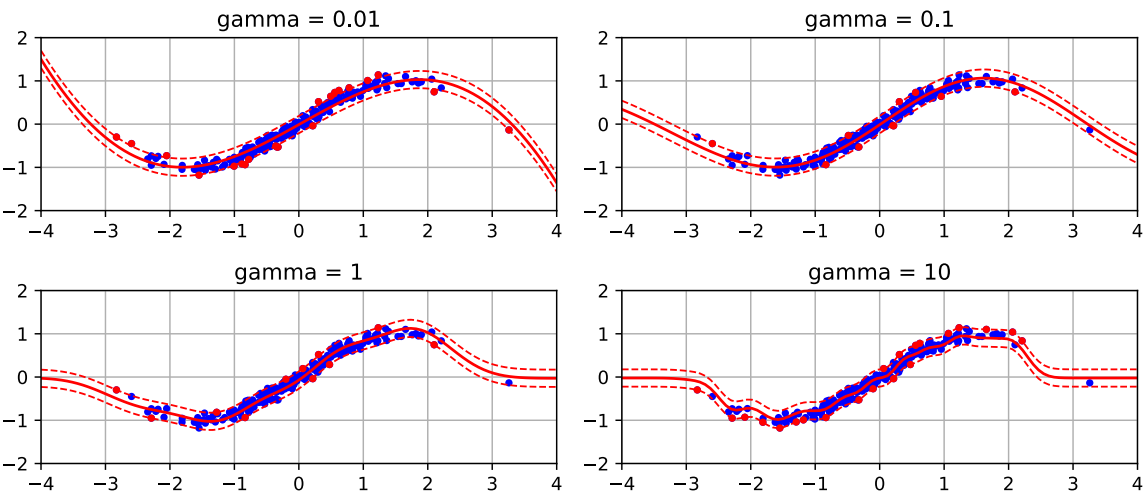


SVR with RBF kernel

In [60]:
```
epsilon = 0.2
svr = {}
for i,g in enumerate([0.01, 0.1, 1, 10]):
    # fit the parameters: SVR with RBF
    svr[i] = svm.SVR(C=1000, kernel='rbf', gamma=g, epsilon=epsilon)
    svr[i].fit(polyX, polyY)
```

In [62]:
```
svrfig
```

Out[62]:



Boston Data

- Cross-validation to select 3 parameters
  - C, γ, ϵ

In [63]:
```
# parameters for cross-validation
paramgrid = {'C':       logspace(-3,3,10),
             'gamma':   logspace(-3,3,10),
             'epsilon': logspace(-2,2,10)}

# do cross-validation
svrcv = model_selection.GridSearchCV(
    svm.SVR(kernel='rbf'),  # estimator
    paramgrid,                      # parameters to try
    scoring='neg_mean_squared_error',  # score function
    cv=5,
    n_jobs=-1, verbose=1)               # show progress
svrcv.fit(bostonX, bostonY)

print(svrcv.best_score_)
print(svrcv.best_params_)
```

```
Fitting 5 folds for each of 1000 candidates, totalling 5000 fits
```

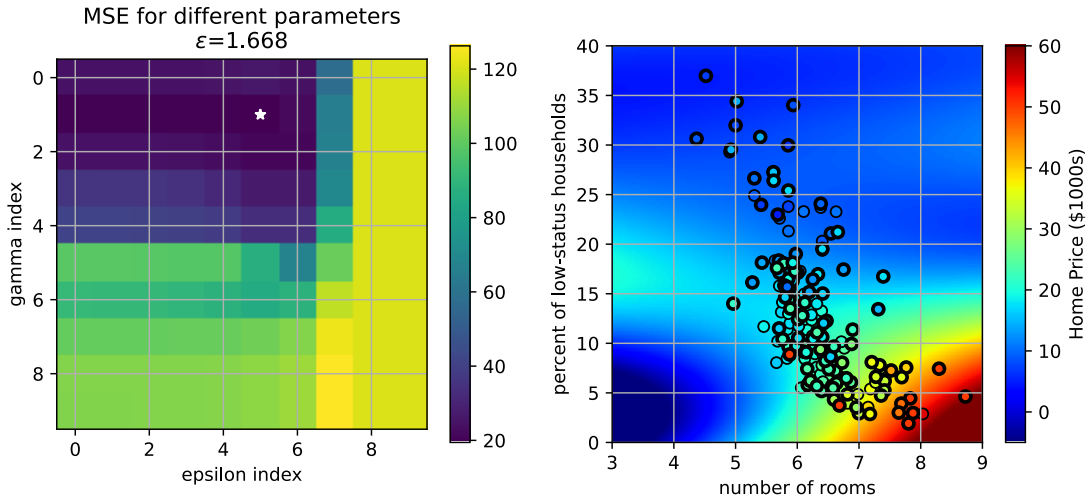
```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks       | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 3560 tasks      | elapsed:    1.9s
```

```
-19.443081073165292
{'C': 1000.0, 'epsilon': 1.6681005372000592, 'gamma': 0.004641588833612777}
```

```
[Parallel(n_jobs=-1)]: Done 5000 out of 5000 | elapsed:    2.7s finished
```

In [65]:
```
kfig
```

Out[65]:



Random Forest Regression

- Similar to Random Forest Classifier
  - Average predictions over many Decision Trees
    - Each decision tree sees a random sampling of the Training set
    - Each split in the decision tree uses a random subset of features
    - Leaf node of tree contains the predicted value.

Example

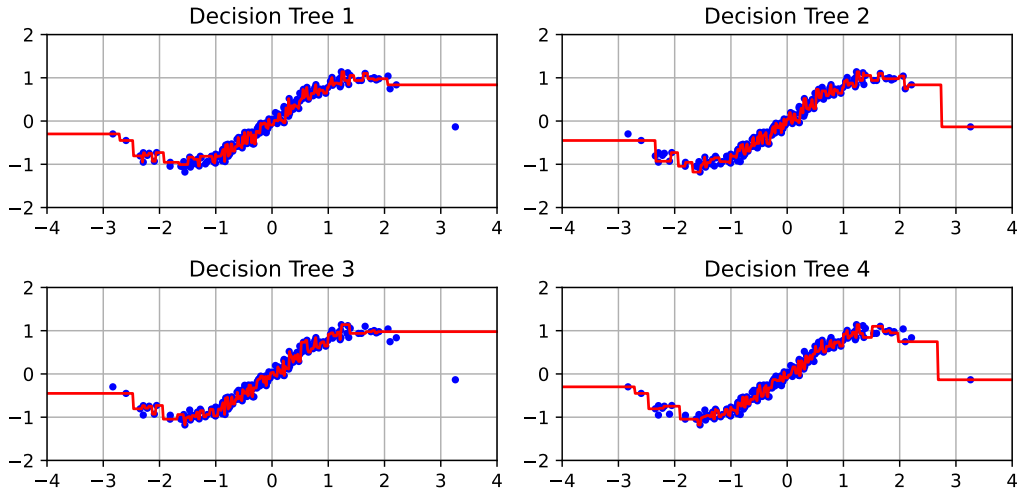
- Four decision trees
  - the regressed function has "steps" because of the decision tree has a constant prediction for ranges of feature values.

In [69]:
```
rf = ensemble.RandomForestRegressor(n_estimators=4, random_state=4487, n_jobs=-1)
```

```
rf.fit(polyX, polyY);
```

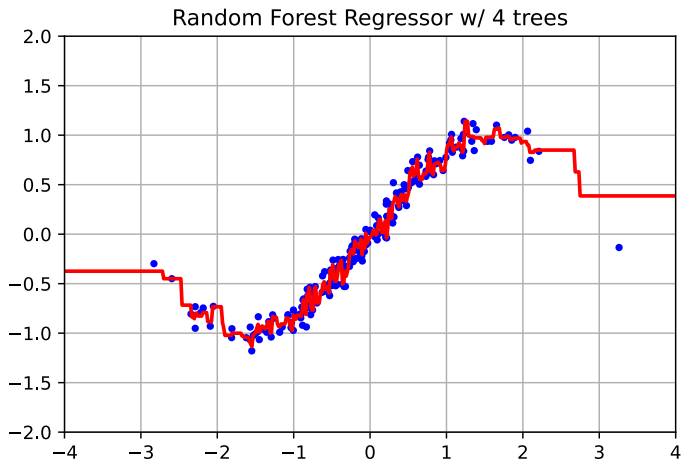
In [76]:
```
rffig
```

Out[76]:



In [77]:
```
# the aggregated function
plt.figure()
plot_rf_1d(rf, naxbox, polyX, polyY, numx=500)
plt.title('Random Forest Regressor w/ 4 trees')
```

Out[77]:
```
Text(0.5, 1.0, 'Random Forest Regressor w/ 4 trees')
```

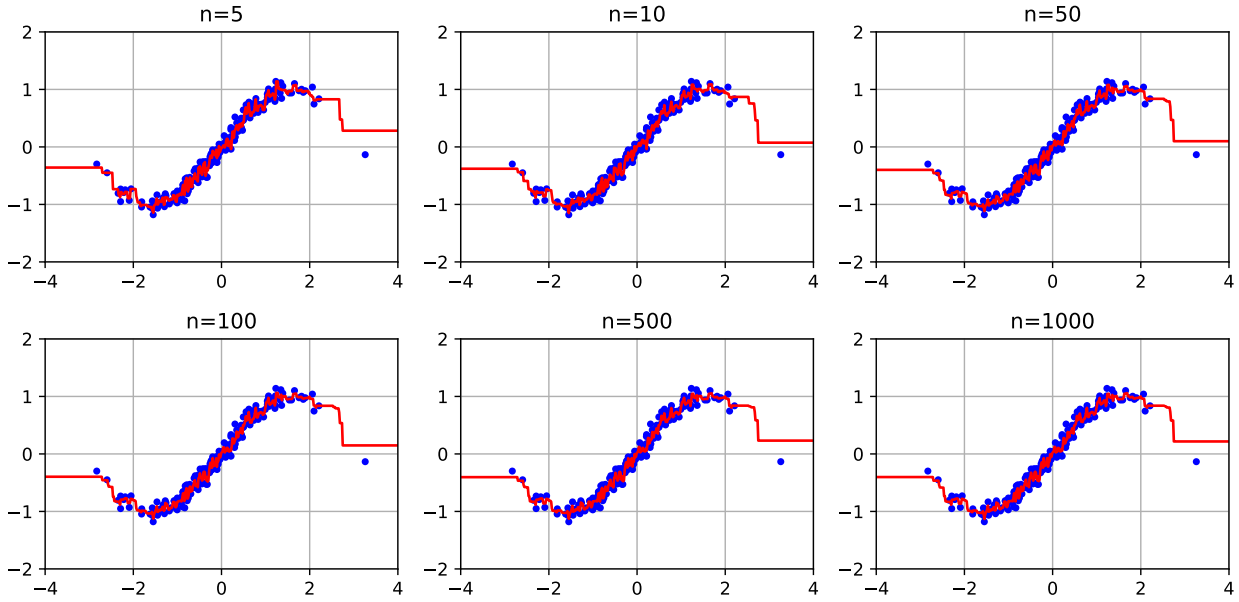


- Using more trees...

In [78]:
```
rf = {}
for i,n in enumerate([5, 10, 50, 100, 500, 1000]):
    rf[i] = ensemble.RandomForestRegressor(n_estimators=n, random_state=4487, n_jobs=-1)
    rf[i].fit(polyX, polyY)
```

In [80]:
```
rffig
```

Out[80]:



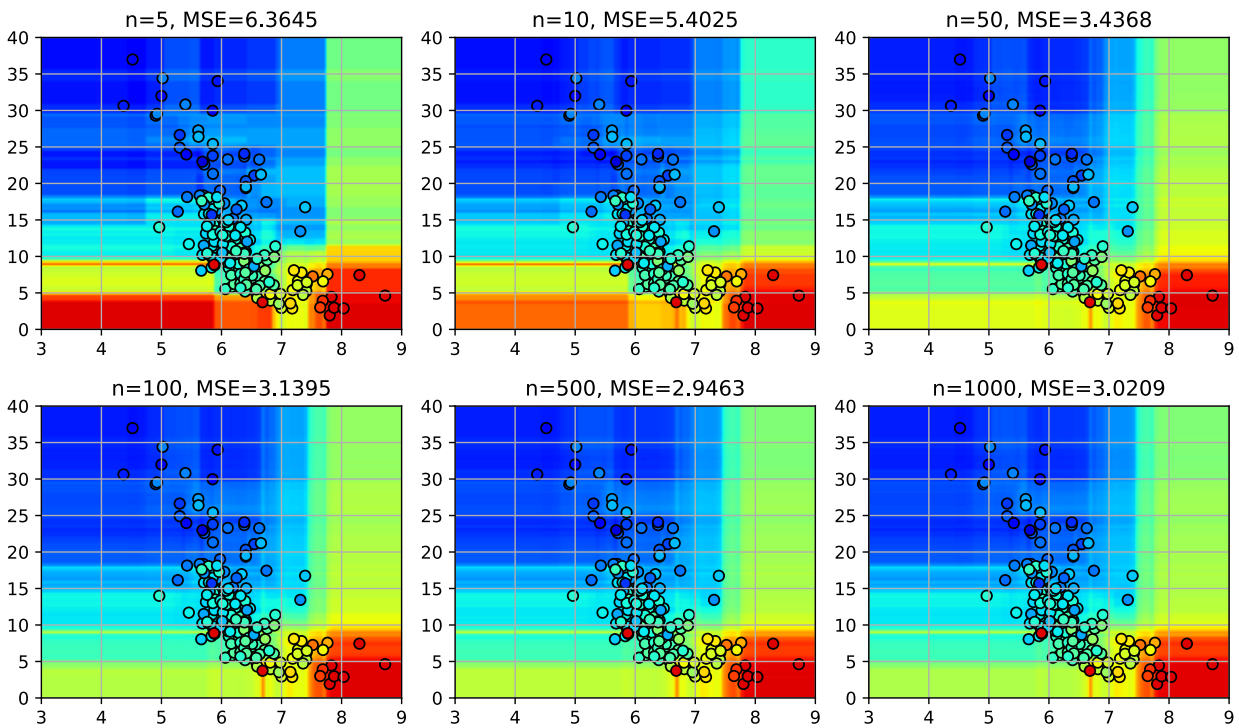
Boston data

- The regressed function looks "blocky"
  - looks more reasonable for areas without any data

In [81]:
```
rf = {}; MSE = {}
for i,n in enumerate([5, 10, 50, 100, 500, 1000]):
    rf[i] = ensemble.RandomForestRegressor(n_estimators=n, random_state=4487, n_jobs=-1)
    rf[i].fit(bostonX, bostonY)
    MSE[i] = metrics.mean_squared_error(bostonY, rf[i].predict(bostonX))
```

In [83]:
```
rffig
```

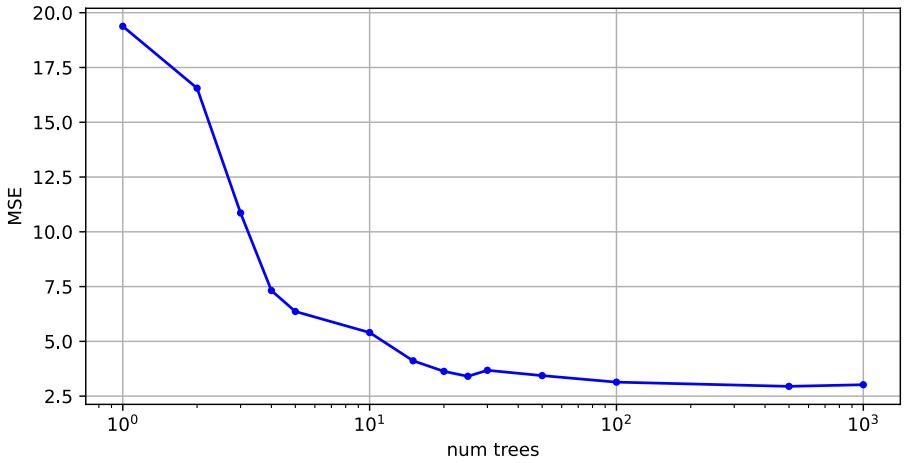
Out[83]:



- plot of MSE versus number of trees

In [85]:
```
mfig
```

Out[85]:



- Use cross-validation to select the tree depth

In [86]:

```
# parameters for cross-validation
paramgrid = {'max_depth': array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15]),
             }

# do cross-validation
rfcv = model_selection.GridSearchCV(
    ensemble.RandomForestRegressor(n_estimators=100, random_state=4487),  # estimator
    paramgrid,                    # parameters to try
    scoring='neg_mean_squared_error',  # score function
    cv=5,
    n_jobs=-1, verbose=True
)
rfcv.fit(bostonX, bostonY)

print(rfcv.best_score_)
print(rfcv.best_params_)
```

```
Fitting 5 folds for each of 11 candidates, totalling 55 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:    2.9s
-21.375250174361355
{'max_depth': 4}
[Parallel(n_jobs=-1)]: Done  55 out of  55 | elapsed:    3.4s finished
```

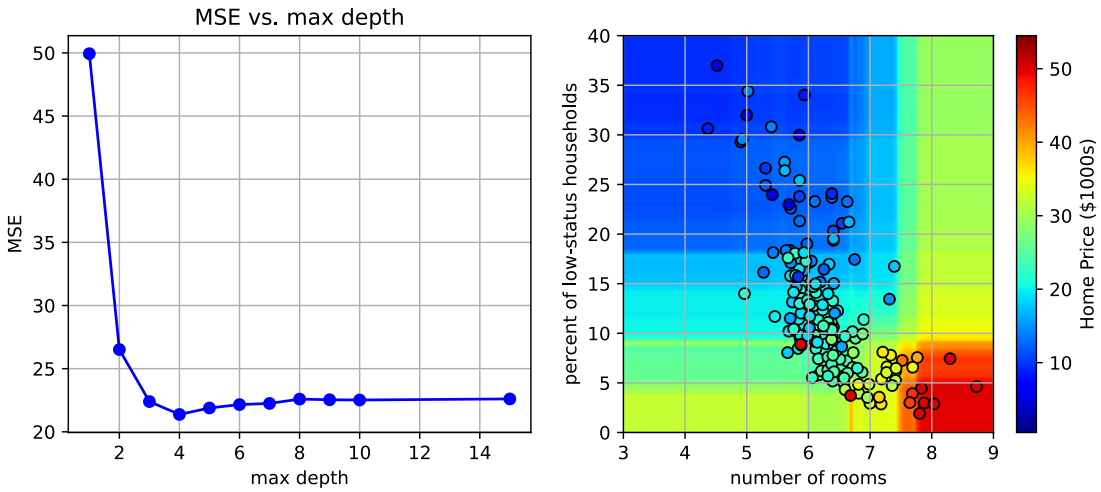
In [87]:

```
(avgscores, pnames, bestind) = extract_grid_scores(rfcv, paramgrid)
```

In [89]:

```
rffig
```

Out[89]:



XGBoost Regression

- similar to XGBoost classification
- Need to change the objective function:
  - `reg:squarederror`: regression with squared loss (Gaussian noise assumption).
  - `reg:gamma`: regression with Gamma noise assumption (non-negative values).
  - others...see documentation.

In [90]:
```
# setup dictionary of distributions for each parameter
paramsampler = {
    "colsample_bytree": stats.uniform(0.7, 0.3),  # default=1
    "gamma":            stats.uniform(0, 0.5),    # default=0
    "max_depth":        stats.randint(2, 6),      # default=6
    "subsample":        stats.uniform(0.6, 0.4),  # default=1
    "learning_rate":    stats.uniform(.001,1),    # default=1 (could also use loguniform)
    "n_estimators":     stats.randint(10, 1000),
}

# the XGB regressor using squared error loss
xr = xgb.XGBRegressor(objective="reg:squarederror", random_state=4487)

# cross-validation via random search
# n_iter = number of parameter combinations to try
xgbrcv = model_selection.RandomizedSearchCV(xr, param_distributions=paramsampler,
                           scoring='neg_mean_squared_error',  # score function
                           random_state=4487, n_iter=200, cv=5,
                           verbose=1, n_jobs=6)

xgbrcv.fit(bostonX, bostonY)
print("best params:", xgbrcv.best_params_)
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
```

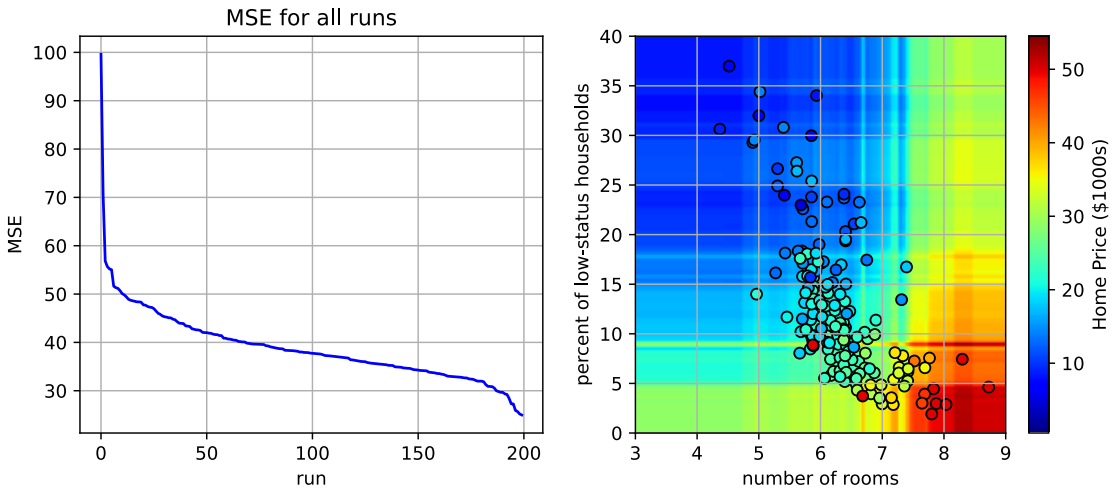
```
[Parallel(n_jobs=6)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  40 tasks      | elapsed:    1.7s
```

```
best params: {'colsample_bytree': 0.920579040898168, 'gamma': 0.06049811280910017,
'learning_rate': 0.05088009671570204, 'max_depth': 3, 'n_estimators': 191, 'subsamp
le': 0.7332861672680347}
```

```
[Parallel(n_jobs=6)]: Done 1000 out of 1000 | elapsed:    8.9s finished
/Users/abc/opt/anaconda3/lib/python3.7/site-packages/xgboost/core.py:613: UserWarni
ng: Use subset (sliced data) of np.ndarray is not recommended because it will gener
ate extra copies and increase memory consumption
  warnings.warn("Use subset (sliced data) of np.ndarray is not recommended " +
```

In [92]:
```
xfig
```

Out[92]:



Regression Summary

- **Goal:** predict output $y \in \mathbb{R}$ from input $\mathbf{x} \in \mathbb{R}^d$.
  - i.e., learn the function $y = f(\mathbf{x})$.

Name	Function	Training	Advantages	Disadvantages
Ordinary Least Squares	linear	minimize square error between observation and predicted output.	- closed-form solution.	- sensitive to outliers and overfitting.
ridge regression	linear	minimize squared error with $\|\|w\|\|^2$ (L2-norm) regularization term.	- closed-form solution; - shrinkage to prevent overfitting.	- sensitive to outliers.
LASSO	linear	minimize squared error with $\sum_{j=1}^{d} \|w_j\|$ (L1-norm) regularization term.	- feature selection (by forcing weights to 0)	- sensitive to outliers.
OMP	linear	minimize squared error with constraint on number of non-zero weights (L0-norm).	- feature selection	- difficult optimization problem, sensitive to outliers.
RANSAC	same as the base model	randomly sample subset of training data and fit model; keep model with most inliers.	- ignores outliers.	- requires enough iterations to find good consensus set.
kernel ridge regression	non-linear (kernel function)	apply "kernel trick" to ridge regression.	- non-linear regression. - Closed-form solution.	- requires calculating kernel matrix $O(N^2)$. - cross-validation to select hyperparameters.
Gaussian process regression	non-linear (kernel function)	- compute posterior distribution - estimate hyperparameters via MML.	- non-linear regression - Closed-form solution. -works well with small datasets - hyperparameter estimation	- requires calculating kernel matrix $O(N^2)$.
kernel support vector regression	non-linear (kernel function)	minimize squared error, insensitive to epsilon-error.	- non-linear regression. - faster predictions than kernel ridge regression.	- requires calculating kernel matrix $O(N^2)$. - iterative solution (slow). - cross-validation to select hyperparameters.
random forest regression	non-linear (ensemble)	aggregate predictions from decision trees.	- non-linear regression. - fast predictions.	- predicts step-wise function. - cannot learn a completely smooth function.
XGBoost regression	non-linear (ensemble)	aggregaate predictions from decision trees.	- non-linear regression. - fast predictions.	- predicts step-wise function. - cannot learn a completely smooth function.

Other Things

- *Feature normalization*
  - feature normalization is typically required for regression methods with regularization.
  - makes ordering of weights more interpretable (LASSO, RR).

- *Output transformations*
  - sometimes the output values y have a large dynamic range (e.g., 10^{-1} to 10^{5}).
    - large output values will have large error, which will dominate the training error.
    - in this case, it is better to transform the output values using the logarithm function.
      - $\hat{y} = \log_{10}(y)$, for example, see the tutorial.
  - Gaussian process regression assumes y values are zero-mean and unit variance.
    - need to normalize to make it well behaved.