CS4487 - Machine Learning

Lecture 5b - Supervised Learning - Regression

Dr. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

Outline

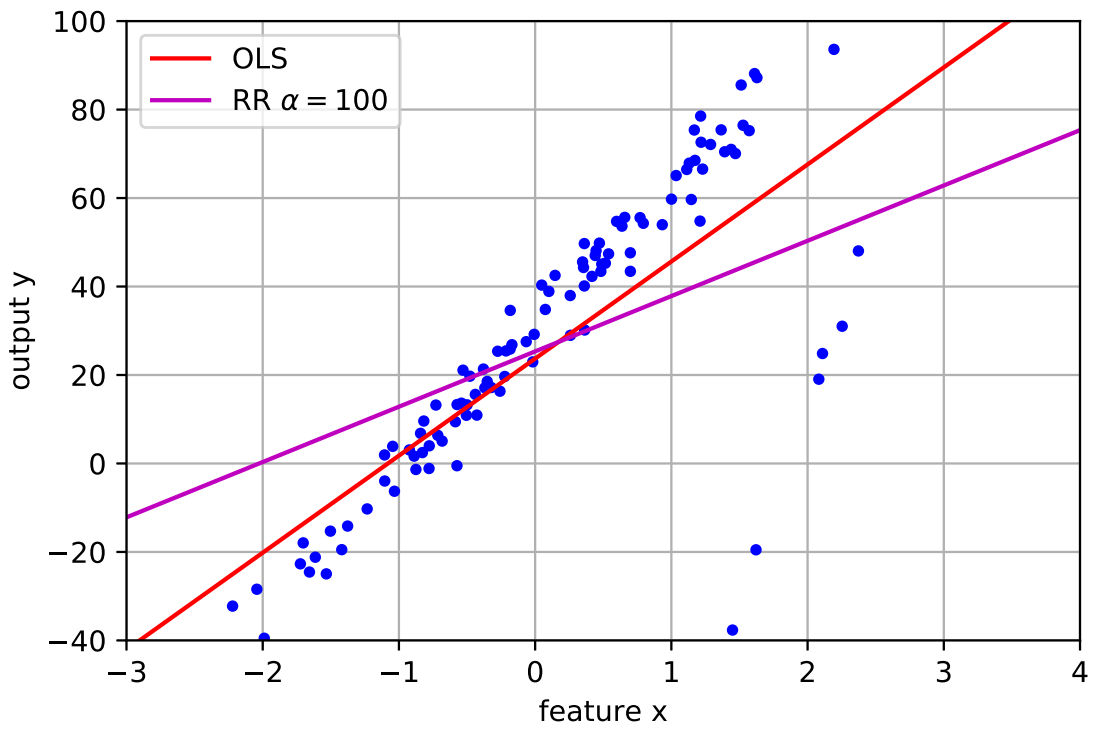
1. Linear Regression
2. Selecting Features
3. **Removing Outliers**
4. Non-linear regression

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

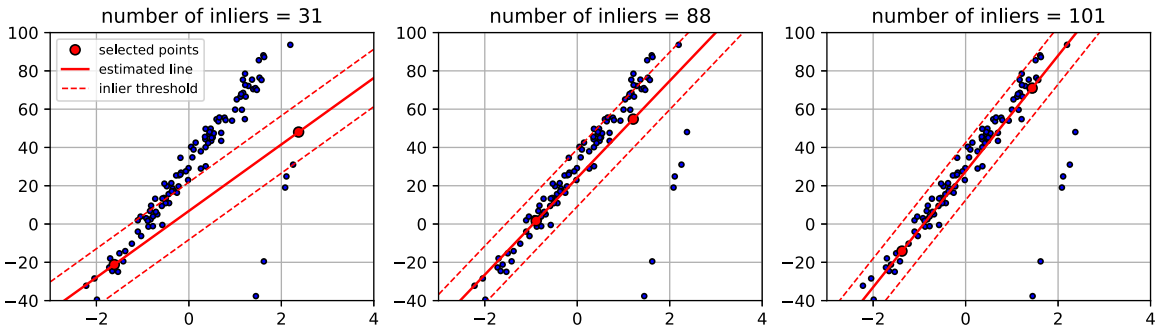
- **RAN**dom **SA**mple **C**onsensus
  - attempt to robustly fit a regression model in the presence of corrupted data (outliers).
  - works with any regression model.
- **Idea:**
  - split the data into inliers (good data) and outliers (bad data).
  - learn the model only from the inliers

Random sampling

- Repeat many times...
  - randomly sample a subset of points from the data. Typically just enough to learn the regression 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.
- Finally, use the largest consensus set to learn the final model.

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

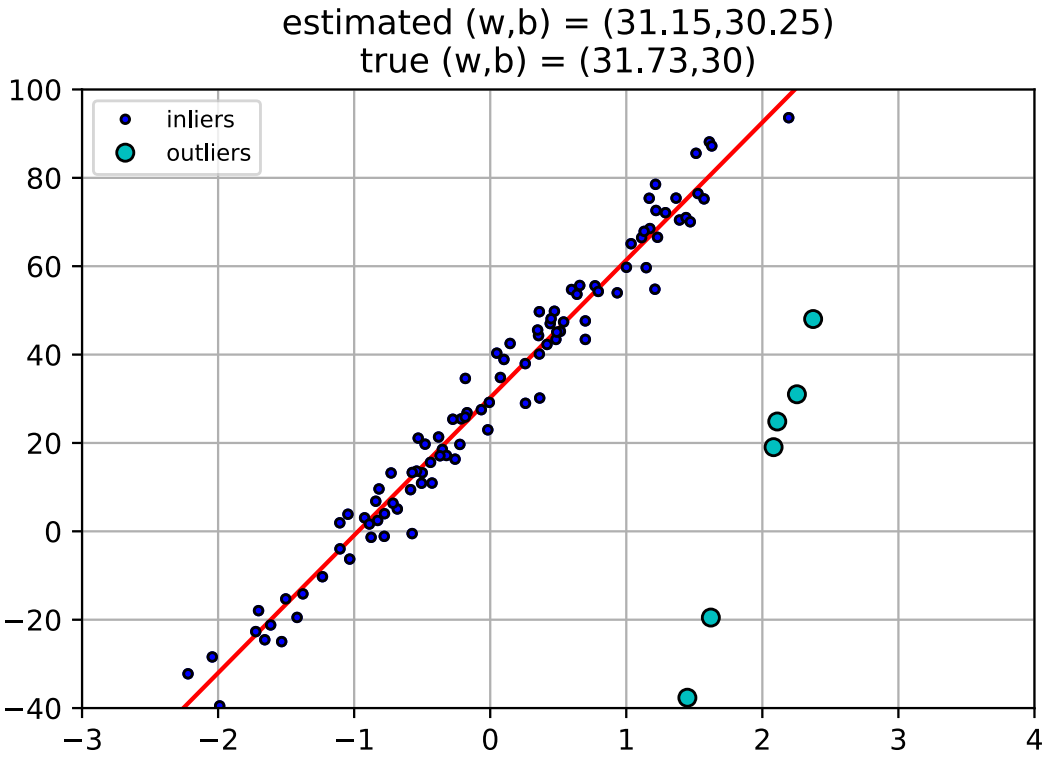
Out[5]:



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)

plt.figure()
plot_regr_trans_1d(rlin, axbox, outlinX, outlinY)
plt.plot(outlinX[inlier_mask], outlinY[inlier_mask], 'b.', label='inliers', mark
eredgecolor='k')
plt.plot(outlinX[outlier_mask], outlinY[outlier_mask], 'co', label='outliers', m
arkeredgecolor='k')
leg = plt.legend(fontsize=8, loc='upper left')
plt.title('estimated (w,b) = (%0.4g,%0.4g)\ntrue (w,b) = (%0.4g,%0.4g)' %
          (rlin.estimator_.coef_, rlin.estimator_.intercept_, lincoefs, linbias)
);
```



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.

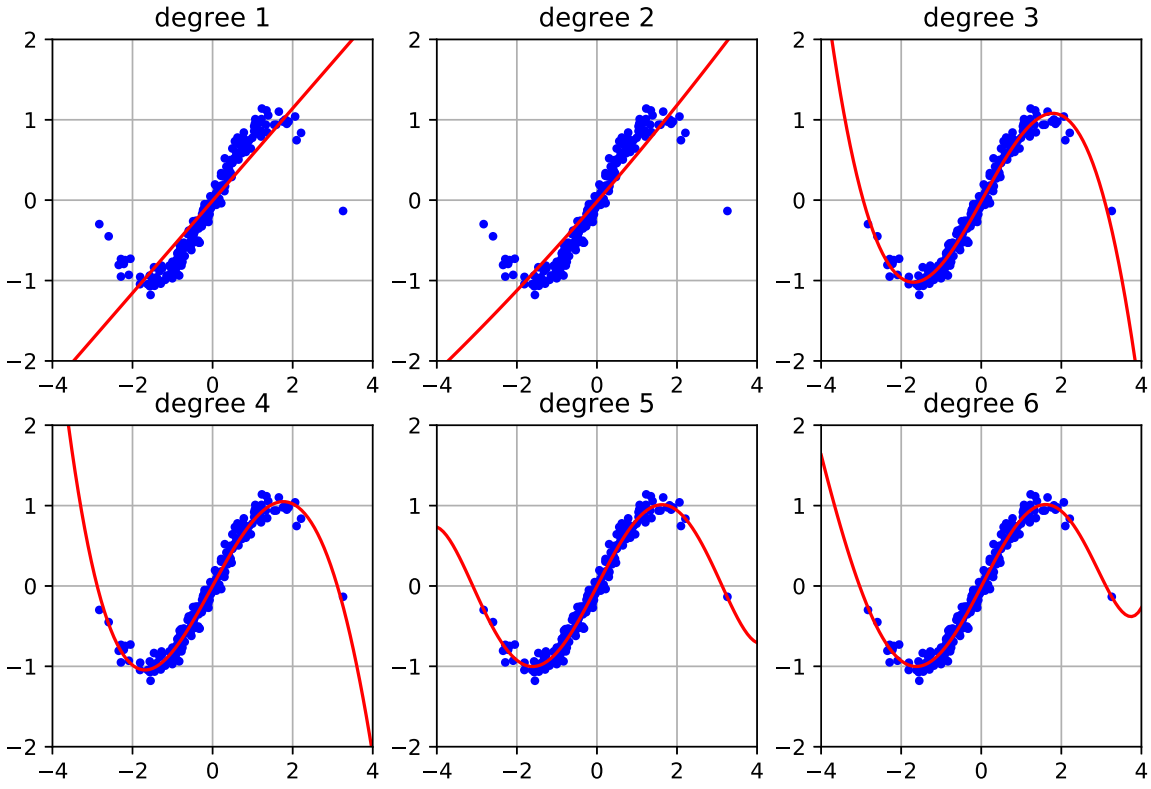
Polynomial regression

- 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 [8]:
```
# example data
polyX = random.normal(size=200)
polyY = sin(polyX) + 0.1*random.normal(size=200)
polyX = polyX[:,newaxis]

plt.figure(figsize=(9,6))
axbox = [-4, 4, -2, 2]

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 = linear_model.LinearRegression()
    plin.fit(polyXf, polyY)

    # make plot
    plt.subplot(2,3,d)
    plot_regr_trans_1d(plin, axbox, polyX, polyY, polyfeats.transform)
    plt.title("degree " + str(d))
```



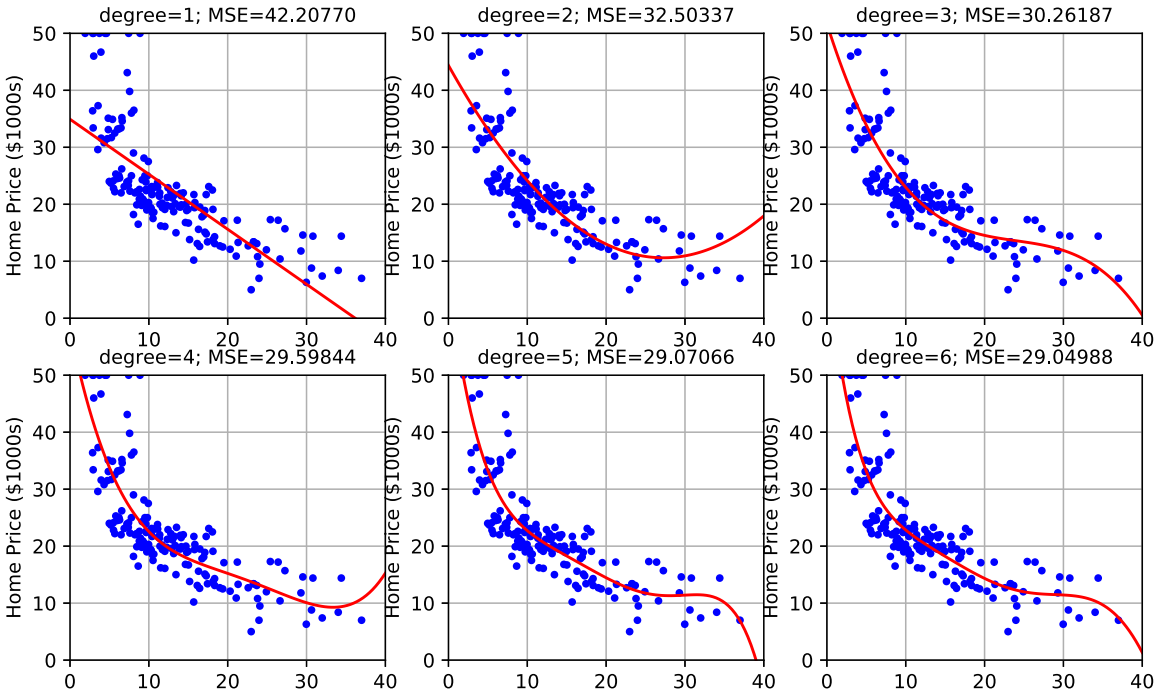
Example: Boston data

- 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 [10]:
```
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 [12]:
```
pfig
```

Out[12]:



Select degree using Cross-Validation

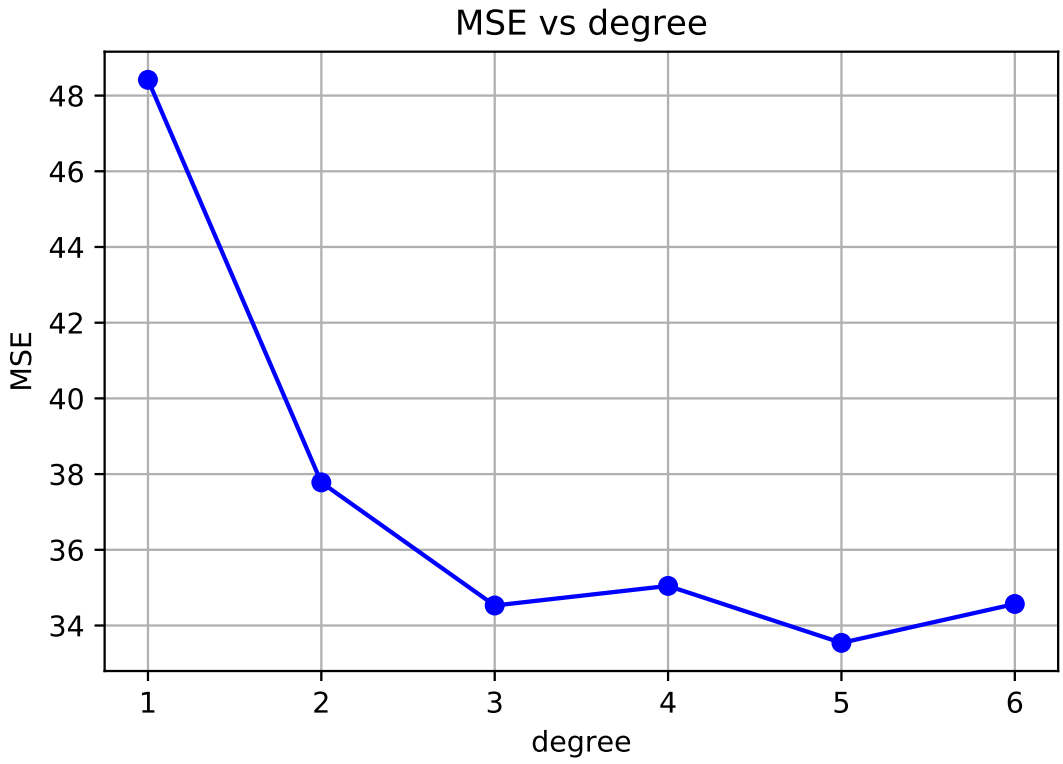
- Minimizing the MSE on the training set will overfit
  - More complex function always has lower MSE on training set
- Use cross-validation to select the proper model
  - the parameters we want to change are in feature transformation step
  - use `pipeline` to merge all steps into one object for easier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

In [13]:
```
# make the pipeline
#  each entry is a tuple with the name and transformer (implements fit, transfor
m)
#  the last entry should be a model (implements fit)
polylin = pipeline.Pipeline([
        ('polyfeats', preprocessing.PolynomialFeatures(degree=1)),
        ('linreg',    linear_model.LinearRegression())
    ])
```

In [14]:
```
# 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-validdation search
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 [16]:
```
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');
```



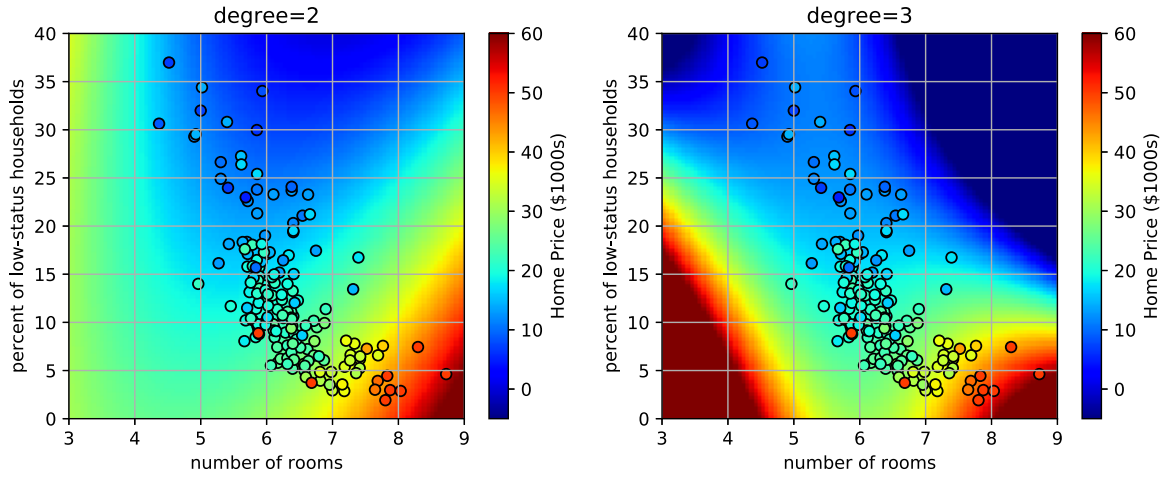
Polynomial features: 2D Example

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

In [18]:
```
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 [20]:
```
pfig
```

Out[20]:



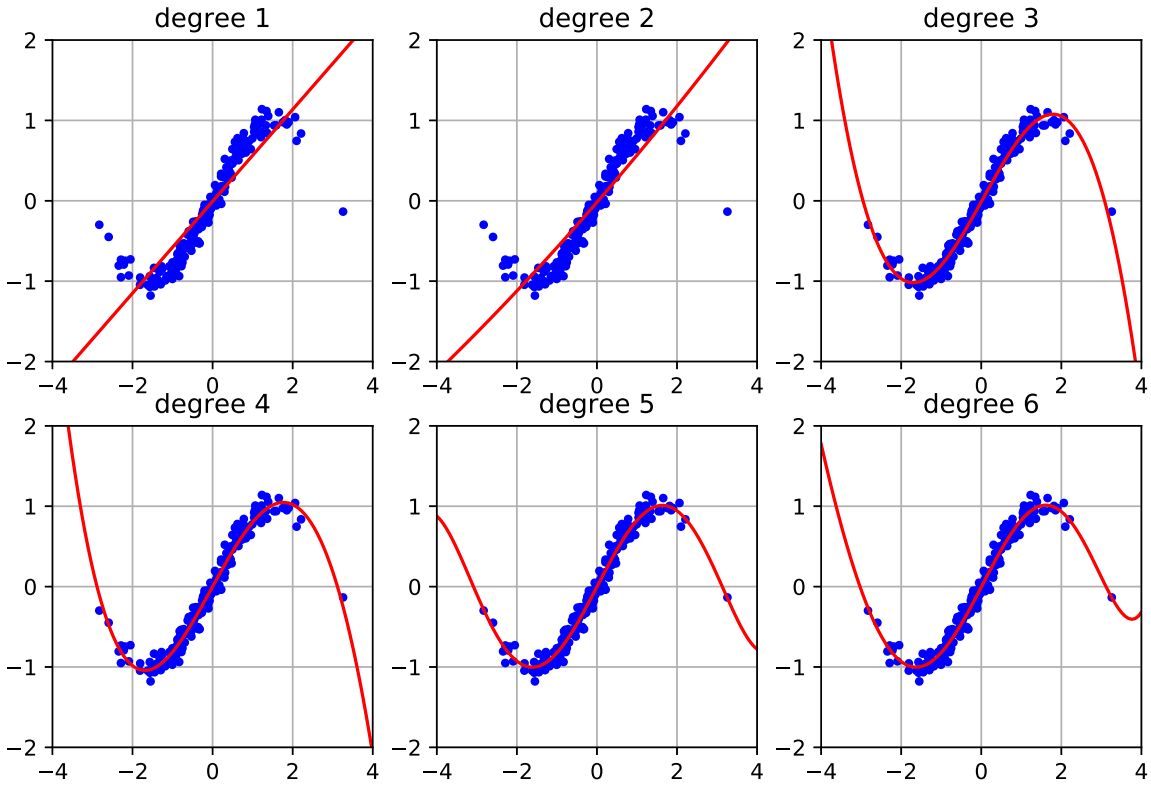
Kernel Ridge Regression

- Apply *kernel trick* to ridge regression
  - turn linear regression into non-linear regression
- Closed form solution:
  - for an input point $\mathbf{x}_*$,
    - prediction: $y_* = \mathbf{k}_*(\mathbf{K} + \alpha I)^{-1}\mathbf{y}$
      - $\mathbf{K}$ - the kernel matrix ($N \times N$)
      - $\mathbf{k}_*$ - 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.

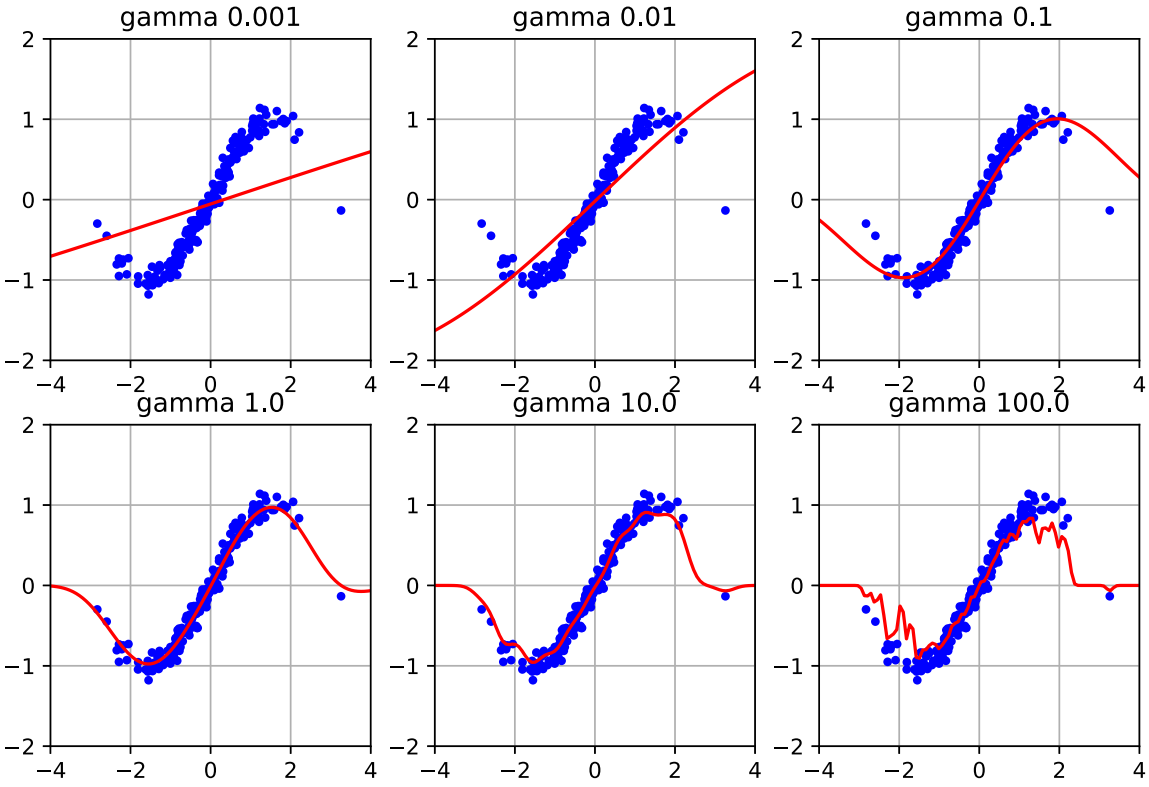
In [21]:
```
plt.figure(figsize=(9,6))
axbox = [-4, 4, -2, 2]

for d in [1,2,3,4,5,6]:

    # fit the parameters
    krr = kernel_ridge.KernelRidge(alpha=1, kernel='poly', degree=d)
    krr.fit(polyX, polyY)

    # plot the function
    plt.subplot(2,3,d)
    plot_regr_trans_1d(krr, axbox, polyX, polyY)
    plt.title("degree " + str(d))
```



Example: RBF kernel

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

In [22]:
```
plt.figure(figsize=(9,6))

axbox = [-4, 4, -2, 2]

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

    # plot the function
    plt.subplot(2,3,i+1)
    plot_regr_trans_1d(krr, axbox, polyX, polyY)
    plt.title("gamma " + str(g))
```



Boston Data: Cross-validation

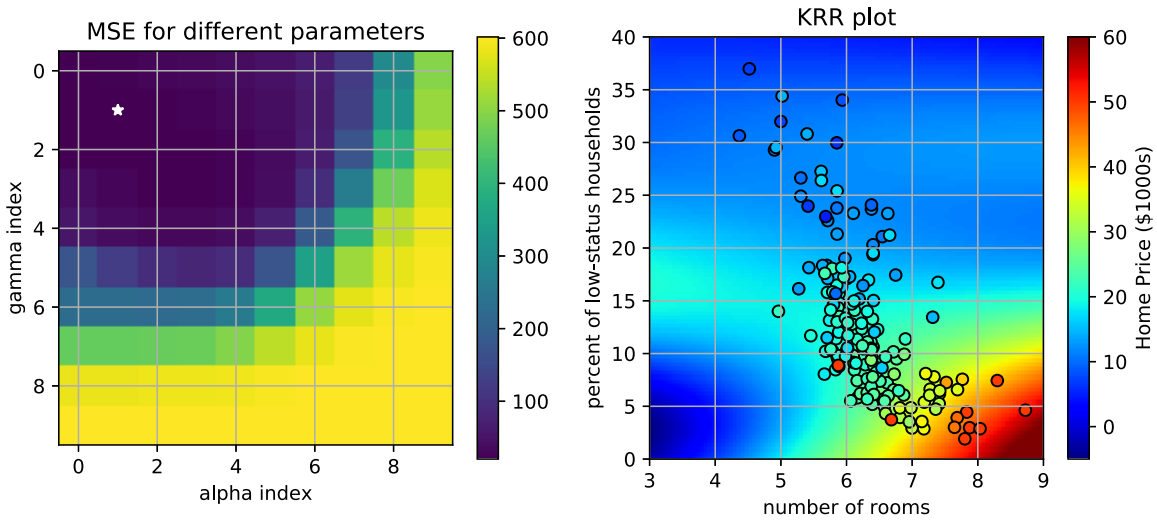
- RBF kernel
  - cross-validation to select α and γ.

In [23]:
```
# 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
-20.406861548069113
{'gamma': 0.004641588833612777, 'alpha': 0.004641588833612777}

[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:    2.4s finished
```

In [25]:
```
kfig
```

Out[25]:

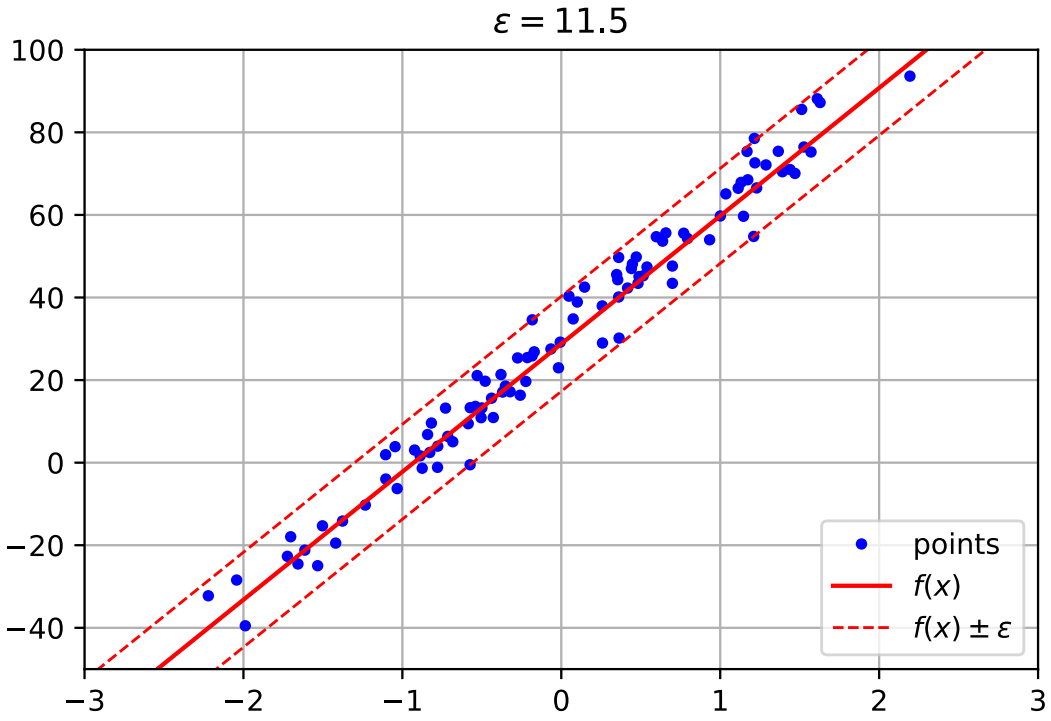


Support Vector Regression (SVR)

- 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 [28]:
```
svrfig
```

Out[28]:



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

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

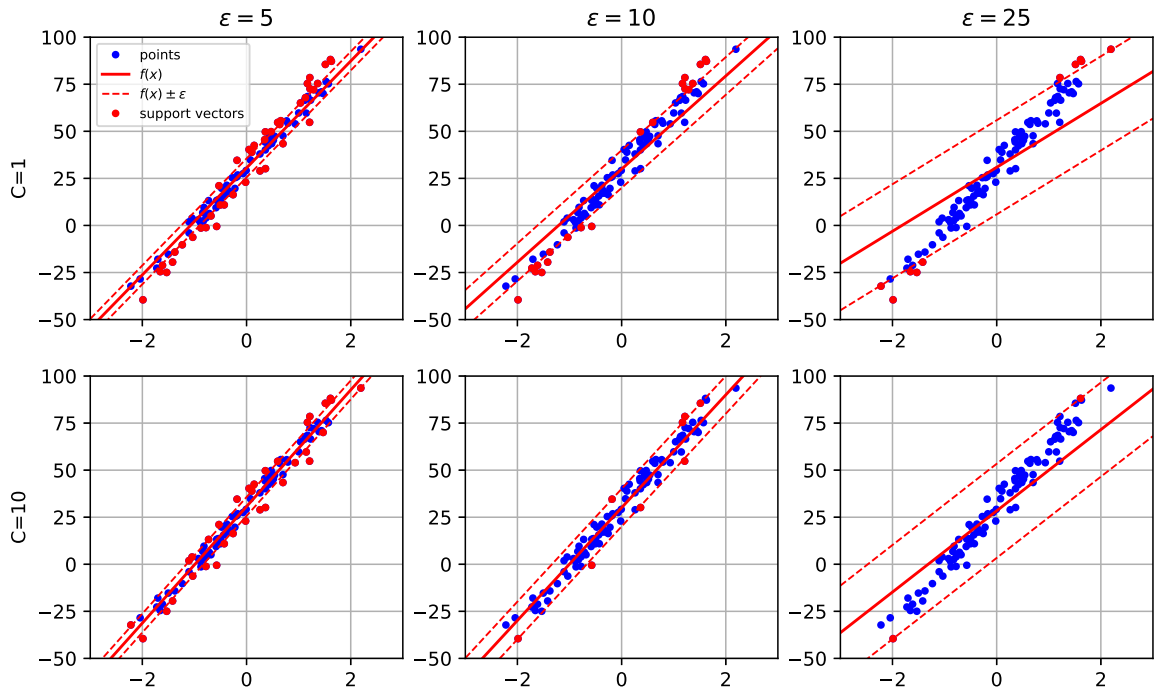
- epsilon-insensitive error:
  - $|z|_\epsilon = \begin{cases} 0, & |z| \leq \epsilon \\ |z| - \epsilon, & |z| > \epsilon \end{cases}$
- Similar to SVM classifier, the points on the band will be the *support vectors* that define the function.

Different tube widths

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

In [30]:
```
svrfig
```

Out[30]:

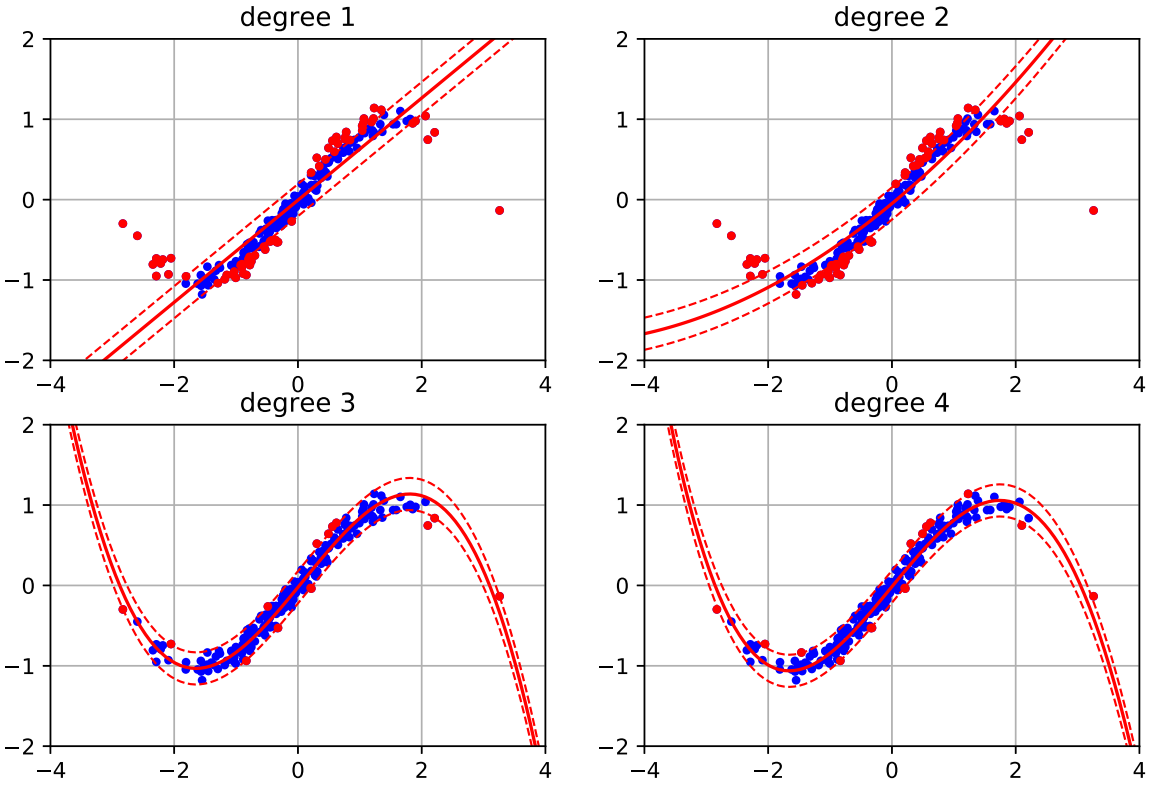


Kernel SVR

- Support vector regression can also be kernelized similar to SVM
  - turn linear regression to non-linear regression
- Polynomial Kernel:

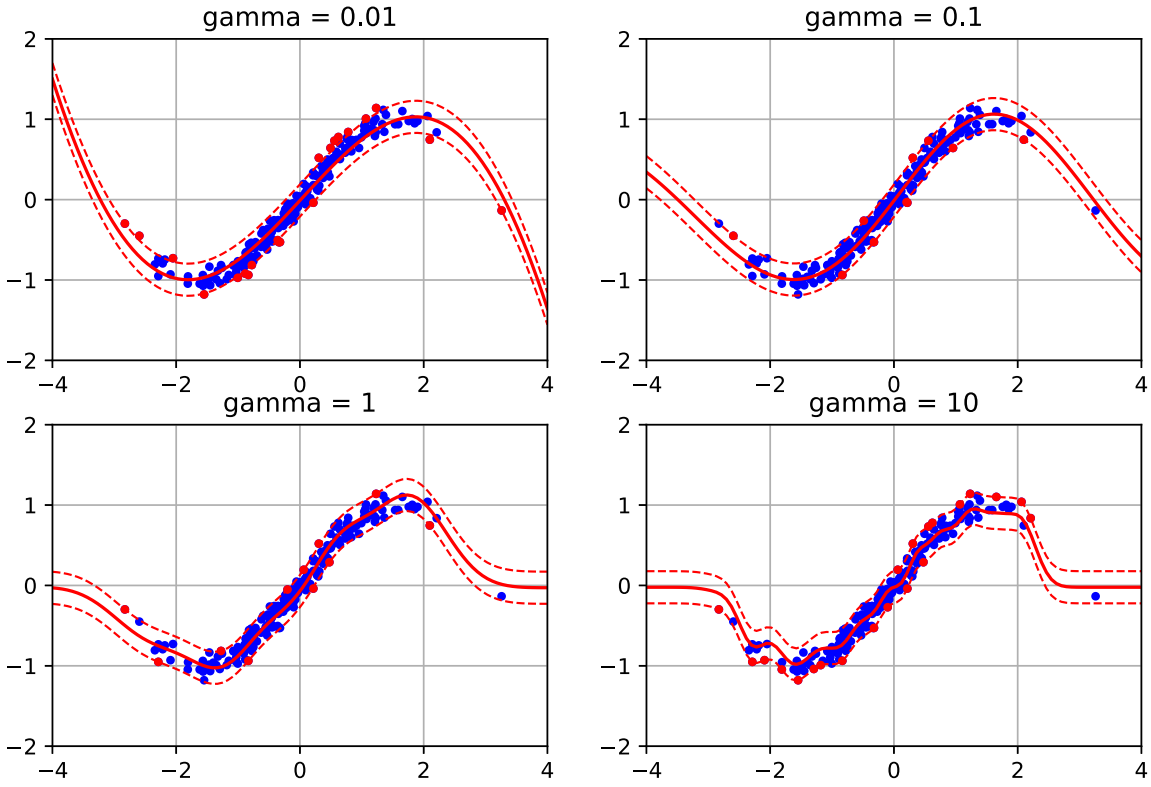
In [31]:

```
plt.figure(figsize=(9,6))

axbox = [-4, 4, -2, 2]
epsilon = 0.2

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

    plt.subplot(2,2,d)
    plot_svr_1d(svr, axbox, polyX, polyY, showsv=True)
    plt.title("degree " + str(d))
```



SVR with RBF kernel

In [32]:
```
plt.figure(figsize=(9,6))

axbox = [-4, 4, -2, 2]
epsilon = 0.2

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

    plt.subplot(2,2,i+1)
    plot_svr_1d(svr, axbox, polyX, polyY, showsv=True)
    plt.title("gamma = " + str(g))
```



Boston Data

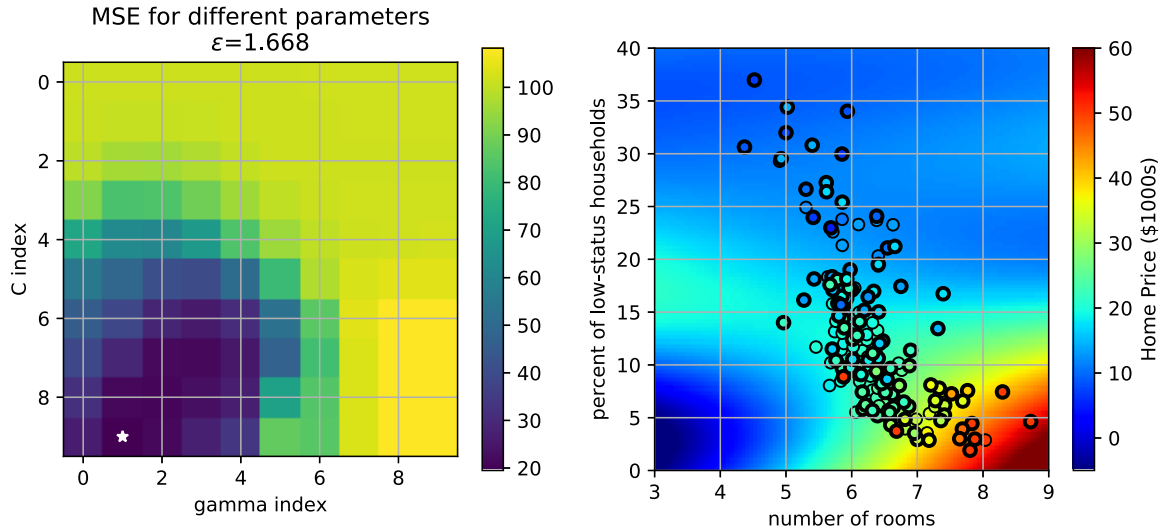
- Cross-validation to select 3 parameters
  - C, γ, ϵ

In [33]:
```
# 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)]: Done 952 tasks      | elapsed:    2.2s

-19.47732071915685
{'epsilon': 1.6681005372000592, 'C': 1000.0, 'gamma': 0.004641588833612777}

[Parallel(n_jobs=-1)]: Done 5000 out of 5000 | elapsed:   11.9s finished
```

In [35]:
```
kfig
```

Out[35]:



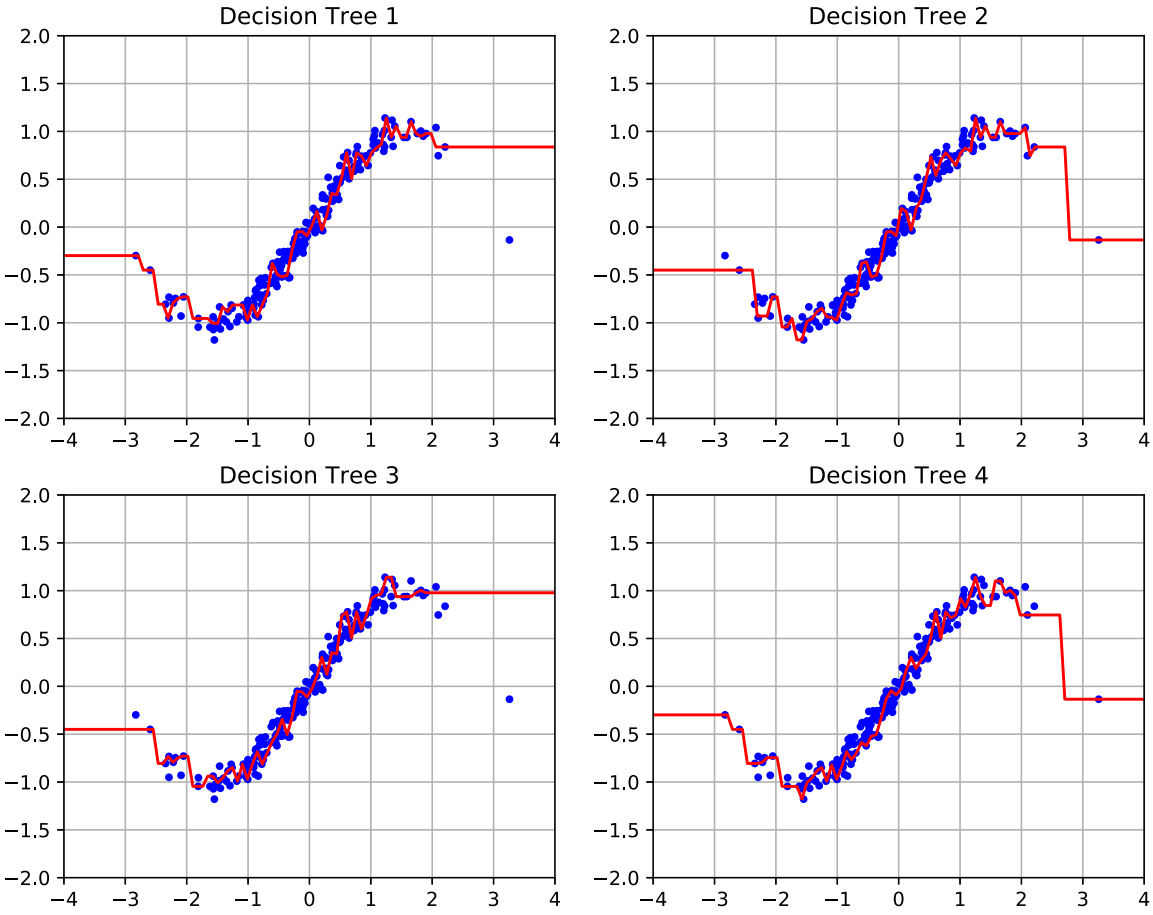
Random Forest Regression

- 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

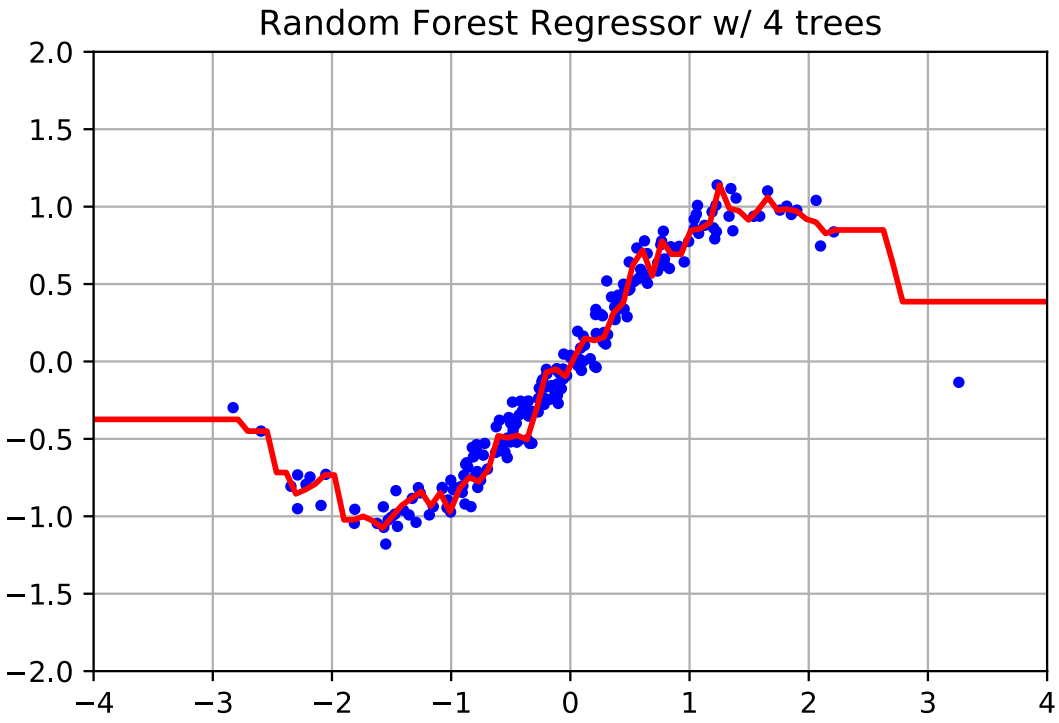
- Four decision trees
  - the regressed function has "steps" because of the decision tree has a constant prediction for ranges of feature values.

In [37]:
```
axbox = [-4, 4, -2, 2]
rf = ensemble.RandomForestRegressor(n_estimators=4, random_state=4487, n_jobs=-1
)
rf.fit(polyX, polyY)

plt.figure(figsize=(10,8))
for i in range(4):
    plt.subplot(2,2,i+1)
    plot_regr_trans_1d(rf.estimators_[i], axbox, polyX, polyY)
    plt.title('Decision Tree ' + str(i+1))
```



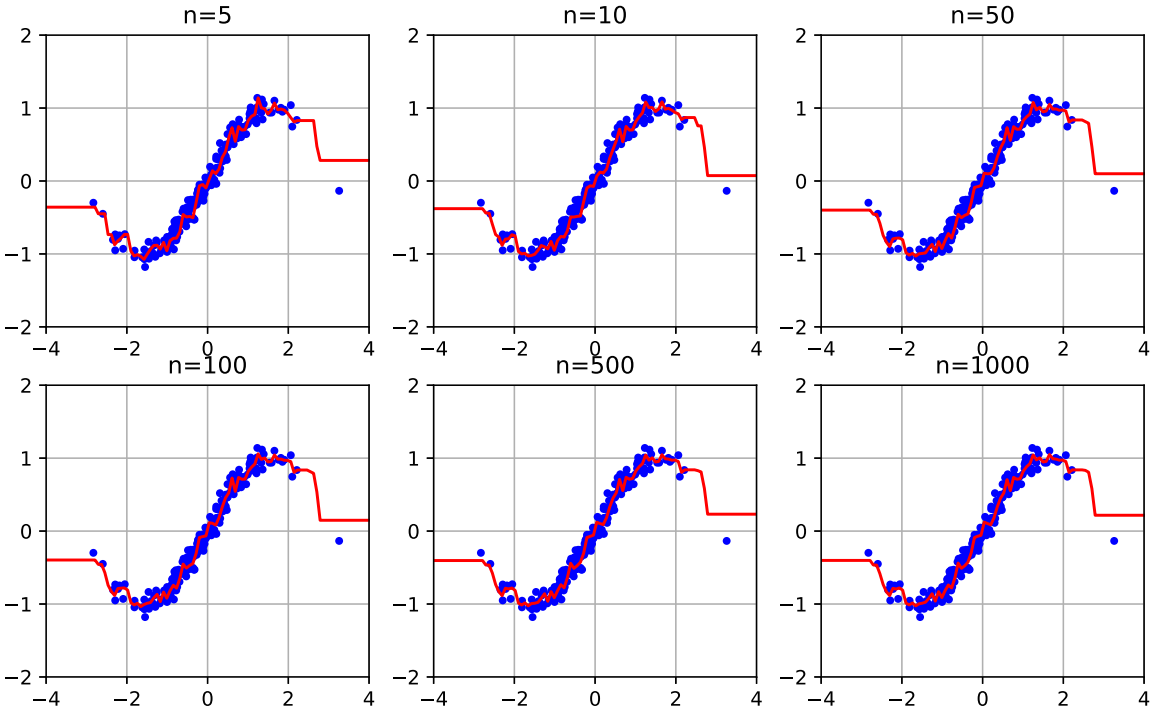
In [38]:
```
# the aggregated function
plt.figure()
plot_rf_1d(rf, axbox, polyX, polyY)
plt.title('Random Forest Regressor w/ 4 trees')
```

Out[38]:
```
Text(0.5,1,'Random Forest Regressor w/ 4 trees')
```



- Using more trees...

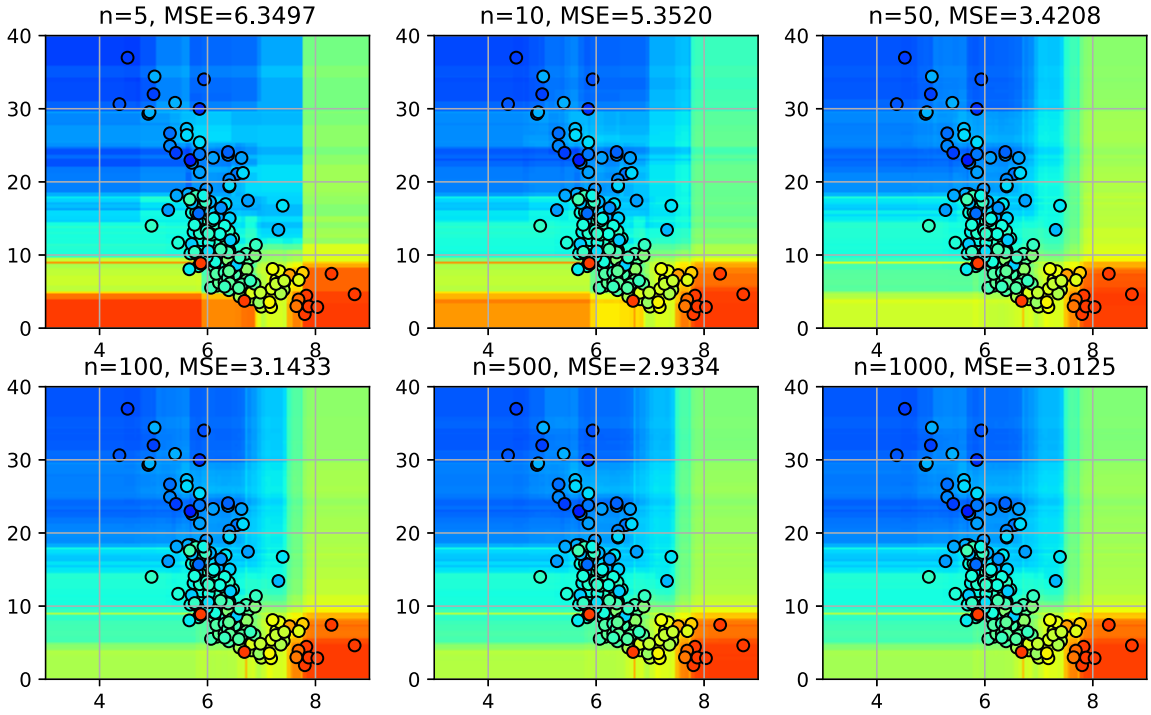
In [39]:
```
plt.figure(figsize=(10,6))
for i,n in enumerate([5, 10, 50, 100, 500, 1000]):
    plt.subplot(2,3,i+1)
    rf = ensemble.RandomForestRegressor(n_estimators=n, random_state=4487, n_jobs=-1)
    rf.fit(polyX, polyY)

    plot_regr_trans_1d(rf, axbox, polyX, polyY)
    plt.title('n='+str(n))
```



Boston data

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

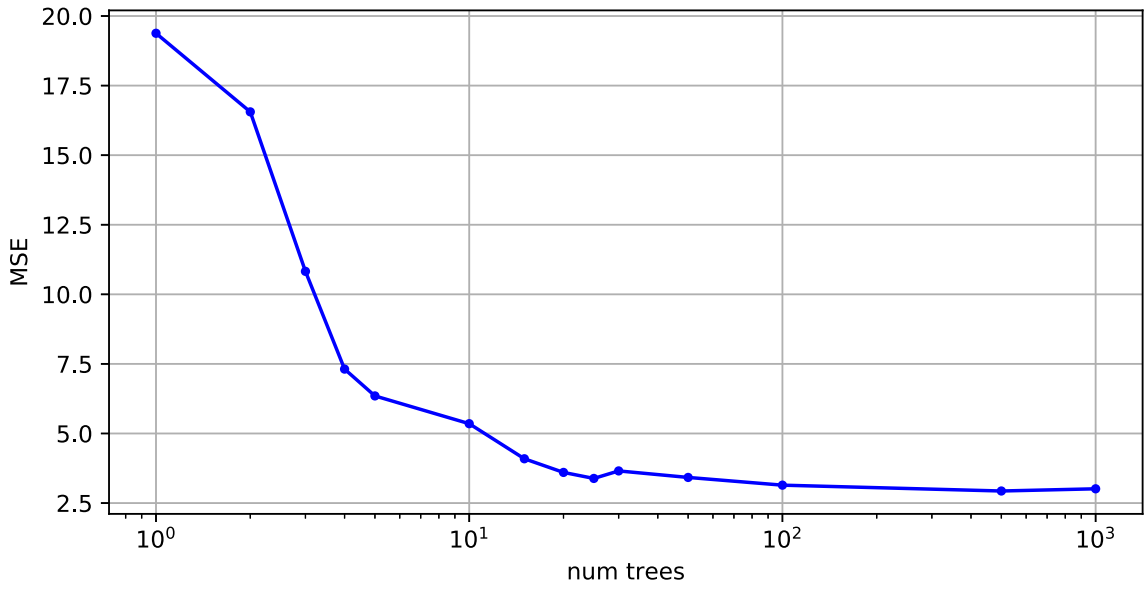
In [40]:
```
plt.figure(figsize=(10,6))
for i,n in enumerate([5, 10, 50, 100, 500, 1000]):
    plt.subplot(2,3,i+1)
    rf = ensemble.RandomForestRegressor(n_estimators=n, random_state=4487, n_jobs=-1)
    rf.fit(bostonX, bostonY)
    MSE = metrics.mean_squared_error(bostonY, rf.predict(bostonX))

    plot_regr_trans_2d(rf, bostonaxbox2, bostonX, bostonY)
    plt.title('n={}, MSE={:.4f}'.format(n, MSE))
```



- plot of MSE versus number of trees

In [42]:
```
mfig
```

Out[42]:



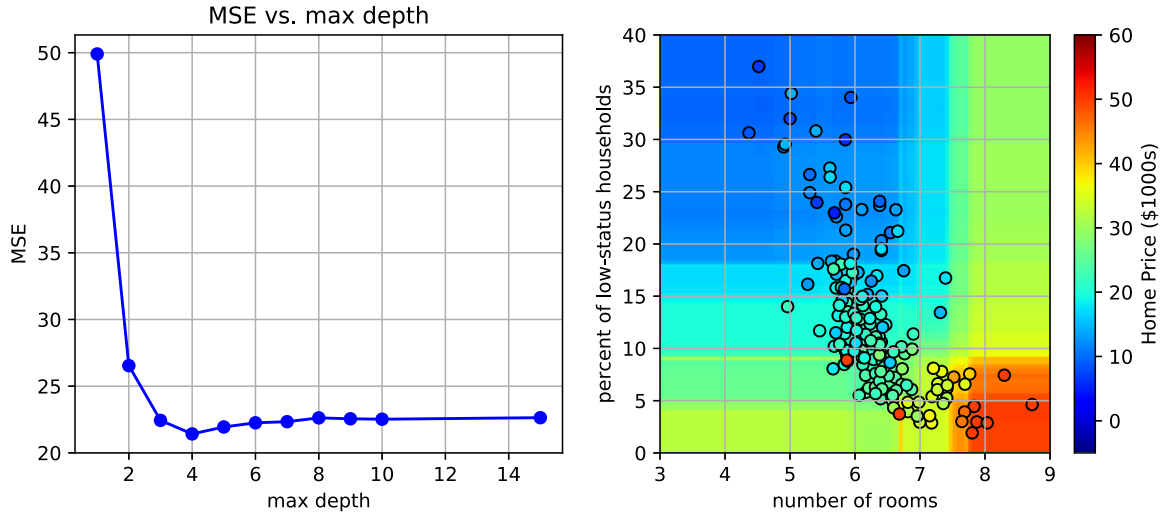
- Use cross-validation to select the tree depth

In [43]:
```
# 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),  # esti
mator
    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)]: Done  42 tasks      | elapsed:    9.6s
[Parallel(n_jobs=-1)]: Done  55 out of  55 | elapsed:   11.5s finished

-21.419077779026953
{'max_depth': 4}
```

In [44]:

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

plt.figure(figsize=(10,4))
# show scores
plt.subplot(1,2,1)
plt.plot(paramgrid['max_depth'], -avgscores, 'bo-')
plt.xlabel('max depth'); plt.ylabel('MSE')
plt.title('MSE vs. max depth')
plt.grid(True)

# show regression function
plt.subplot(1,2,2)
plot_regr_trans_2d(rfcv, bostonaxbox2, bostonX, bostonY)
cbar = plt.colorbar()
cbar.set_label('Home Price ($1000s)')
plt.xlabel('number of rooms'); plt.ylabel('percent of low-status households');
```



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 $\|\|w\|\|^2$ regularization term.	- closed-form solution; - shrinkage to prevent overfitting.	- sensitive to outliers.
LASSO	linear	minimize squared error with $\sum_{j=1}^{d} \|w_j\|$ regularization term.	- feature selection (by forcing weights to 0)	- 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.
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.

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.