CS6375.004: Machine Learning (Spring '19)

***Instructor**: Gautam Kunapuli*

***Due**: February 11 (Monday)*

Homework 1

The report component of this assignment is the **hard copy** of this homework, along with your answers to questions, and is **due at the start of class on Monday, February 11, 2019**.

The **electronic version** of this homework **must be uploaded on eLearning by 12:59pm Central Standard Time, Monday, February 11, 2019**.

All deadlines are hard and without exceptions unless permission was obtained from the instructor in advance**.

You may work in groups to discuss the problems and work through solutions together. However, you must **write up your solutions on your own**, without copying another student's work or letting another student copy your work. In your solution for each problem, you must write down the names of your partner (if any); this will not affect your grade.

Generating Synthetic Data

This assignment shows how we can extend ordinary least squares regression, which uses the hypothesis class of linear regression functions, to non-linear regression functions modeled using polynomial basis functions and radial basis functions. The function we want to fit is $y_{\text{true}} = f_{\text{true}}(x) = 6\sin(x+2) + \sin(2x+4)$. This is a **univariate function** as it has only one input variable. First, we generate synthetic input (data) x_i by sampling $n = 750$ points from a uniform distribution on the interval $[-7.5,\ 7.5]$.

In [68]:
```
# The true function
import numpy as np
def f_true(x):
  y = 6.0 * (np.sin(x + 2) + np.sin(2*x + 4))
  return y
```

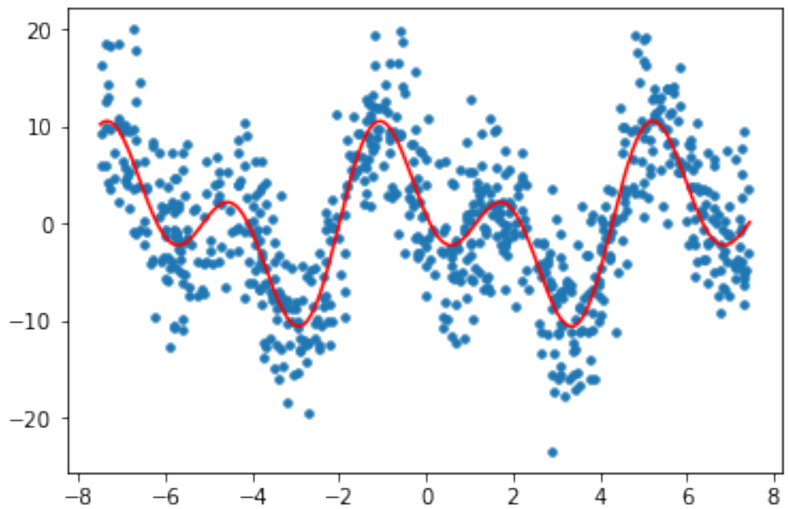
We can generate a synthetic data set, with Gaussian noise.

In [69]:
```
import numpy as np                            # For all our math needs
n = 750                                       # Number of data points
X = np.random.uniform(-7.5, 7.5, n)           # Training examples, in one d
imension
e = np.random.normal(0.0, 5.0, n)             # Random Gaussian noise
y = f_true(X) + e                             # True labels with noise
```

Now, we plot the raw data as well as the true function (without noise).

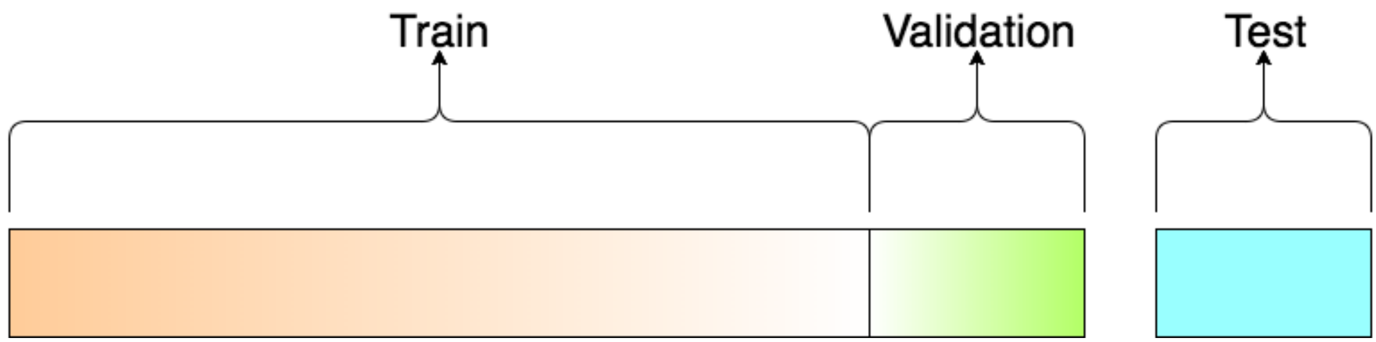
In [70]:
```
import matplotlib.pyplot as plt          # For all our plotting needs
plt.figure()

# Plot the data
plt.scatter(X, y, 12, marker='o')

# Plot the true function, which is really "unknown"
x_true = np.arange(-7.5, 7.5, 0.05)
y_true = f_true(x_true)
plt.plot(x_true, y_true, marker='None', color='r')
```

Out[70]: [<matplotlib.lines.Line2D at 0x1a1667aa20>]



Recall that we want to build a model to **generalize well on future data**, and in order to generalize well on future data, we need to pick a model that trade-off well between fit and complexity (that is, bias and variance). We randomly split the overall data set ($\mathcal{D}$) into three subsets:

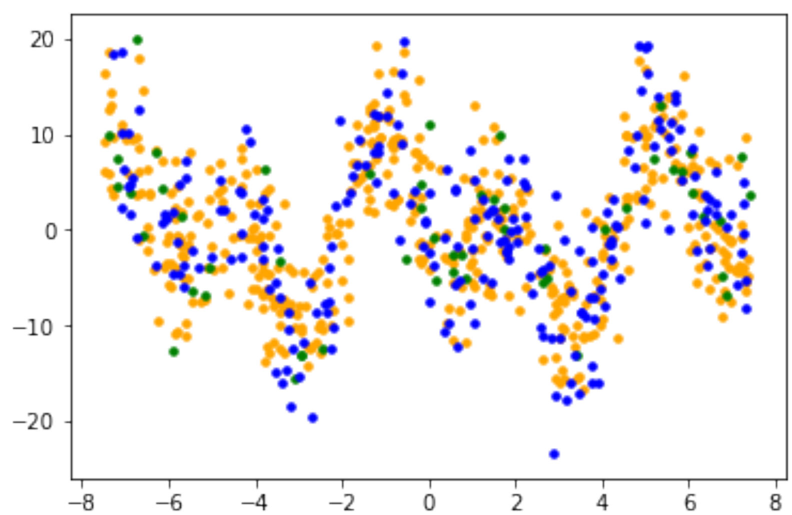
- **Training set**: $\mathcal{D}_{\text{trn}}$ consists of the actual training examples that will be used to **train the model**;
- **Validation set**: $\mathcal{D}_{\text{val}}$ consists of validation examples that will be used to **tune model hyperparameters** (such as $\lambda > 0$ in ridge regression) in order to find the best trade-off between fit and complexity (that is, the value of λ that produces the best model);
- **Test set**: $\mathcal{D}_{\text{tst}}$ consists of test examples to **estimate how the model will perform on future data**.



For this example, let us randomly partition the data into three non-intersecting sets: $\mathcal{D}_{\text{trn}} = 60\%$ of $\mathcal{D}$, $\mathcal{D}_{\text{val}} = 10\%$ of $\mathcal{D}$ and $\mathcal{D}_{\text{tst}} = 30\%$ of $\mathcal{D}$.

In [71]:
```
# scikit-learn has many tools and utilities for model selection
from sklearn.model_selection import train_test_split
tst_frac = 0.3  # Fraction of examples to sample for the test set
val_frac = 0.1  # Fraction of examples to sample for the validation se
t

# First, we use train_test_split to partition (X, y) into training and
test sets
X_trn, X_tst, y_trn, y_tst = train_test_split(X, y, test_size=tst_frac
, random_state=42)

# Next, we use train_test_split to further partition (X_trn, y_trn) in
to training and validation sets
X_trn, X_val, y_trn, y_val = train_test_split(X_trn, y_trn, test_size=
val_frac, random_state=42)

# Plot the three subsets
plt.figure()
plt.scatter(X_trn, y_trn, 12, marker='o', color='orange')
plt.scatter(X_val, y_val, 12, marker='o', color='green')
plt.scatter(X_tst, y_tst, 12, marker='o', color='blue')
```

Out[71]: <matplotlib.collections.PathCollection at 0x1a19235cc0>



1. **Regression with Polynomial Basis Functions**, 30 points.

This problem extends **ordinary least squares regression**, which uses the hypothesis class of *linear regression functions*, to *non-linear regression functions* modeled using **polynomial basis functions**. In order to learn nonlinear models using linear regression, we have to explicitly **transform the data** into a higher-dimensional space. The nonlinear hypothesis class we will consider is the set of d-degree polynomials of the form $f(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$ or **a linear combination of polynomial basis function**:

$$f(x) = [w_0, w_1, w_2 \ldots, w_d]^T \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^d \end{bmatrix}.$$

The monomials $\{1, x, x^2, \ldots, x^d\}$ are called **basis functions**, and each basis function x^k has a corresponding weight w_k associated with it, for all $k = 1, \ldots, d$. We transform each univariate data point x_i into into a multivariate (d-dimensional) data point via $\phi(x_i) \rightarrow [1, x_i, x_i^2, \ldots, x_i^d]$. When this transformation is applied to every data point, it produces the **Vandermonde matrix**:

$$\Phi = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^d \\ 1 & x_2 & x_2^2 & \ldots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{bmatrix}.$$

In [178]:

```
# X float(n, ): univariate data
# d int: degree of polynomial

import numpy as np
def polynomial_transform(X, d):
    vander=[]
    for i in range(len(X)):
        new=[]
        for j in range(d):
            new.append(X[i]**j)
        vander.append(new)
    return np.flip(np.array(vander),axis=1)
```

b. (10 points)

Complete the Python function below that takes a Vandermonde matrix Φ and the labels $\mathbf{y}$ as input and learns weights via **ordinary least squares regression**. Specifically, given a Vandermonde matrix Φ, implement the computation of $\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$. *Remember that in Python, @ performs matrix multiplication, while * performs element-wise multiplication. Alternately, [numpy.dot (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.dot.html)](https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.dot.html) also performs matrix multiplication.*

In [181]:

```
# Phi float(n, d): transformed data
# y   float(n,  ): labels
import numpy as np
def train_model(Phi, y):
  Phit=Phi.transpose()
  Inner=np.dot(Phit,Phi)
  InvInner=np.linalg.inv(Inner)
  a=np.dot(InvInner,Phit)
  return np.dot(a,y)
```

c. (5 points)

Complete the Python function below that takes a Vandermonde matrix Φ, corresponding labels $\mathbf{y}$, and a linear regression model $\mathbf{w}$ as input and evaluates the model using **mean squared error**. That is, $\epsilon_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \Phi_i)^2$.

In [182]:

```
# Phi float(n, d): transformed data
# y   float(n,  ): labels
# w   float(d,  ): linear regression model
import numpy as np
def evaluate_model(Phi, y, w):
    mul=np.dot(Phi,w)
    res=(np.subtract(y,mul))
    res=np.power(res,2)
    res=np.sum(res)
    return res/len(y)
```

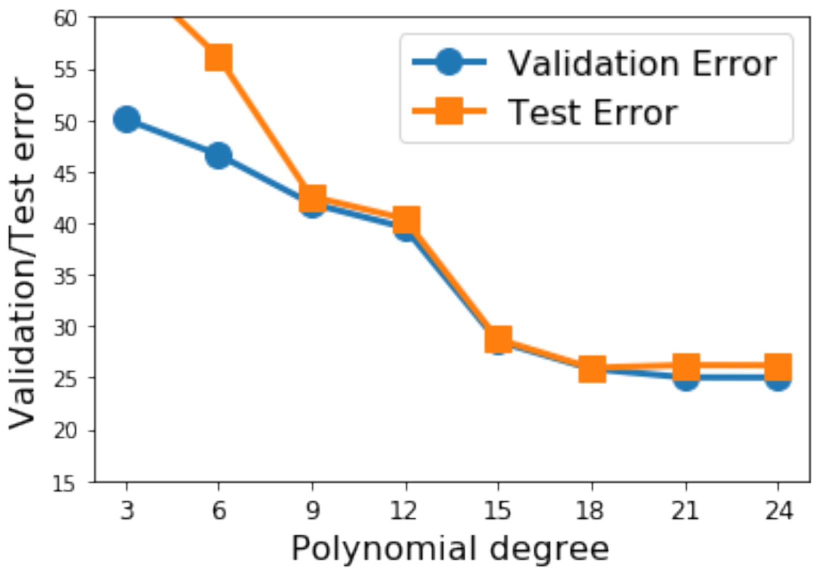
d. (5 points, Discussion)

We can explore the **effect of complexity** by varying $d = 3, 6, 9, \cdots, 24$ to steadily increase the non-linearity of the models. For each model, we train using the transformed training data (Φ, whose dimension increases) and evaluate its performance on the transformed validation data and estimate what our future accuracy will be using the test data.

From plot of d vs. validation error below, which choice of d do you expect will generalize best?

In [183]:
```
w = {}                     # Dictionary to store all the trained models
validationErr = {}   # Validation error of the models
testErr = {}           # Test error of all the models
for d in range(3, 25, 3):  # Iterate over polynomial degree
    Phi_trn = polynomial_transform(X_trn, d)                 # Transfo
rm training data into d dimensions
    w[d] = train_model(Phi_trn, y_trn)                       # Learn m
odel on training data

    Phi_val = polynomial_transform(X_val, d)                 # Transfo
rm validation data into d dimensions
    validationErr[d] = evaluate_model(Phi_val, y_val, w[d])  # Evaluat
e model on validation data

    Phi_tst = polynomial_transform(X_tst, d)           # Transform tes
t data into d dimensions
    testErr[d] = evaluate_model(Phi_tst, y_tst, w[d])  # Evaluate mode
l on test data

# Plot all the models
plt.figure()
plt.plot(validationErr.keys(), validationErr.values(), marker='o', lin
ewidth=3, markersize=12)
plt.plot(testErr.keys(), testErr.values(), marker='s', linewidth=3, ma
rkersize=12)
plt.xlabel('Polynomial degree', fontsize=16)
plt.ylabel('Validation/Test error', fontsize=16)
plt.xticks(list(validationErr.keys()), fontsize=12)
plt.legend(['Validation Error', 'Test Error'], fontsize=16)
plt.axis([2, 25, 15, 60])
```

Out[183]: [2, 25, 15, 60]

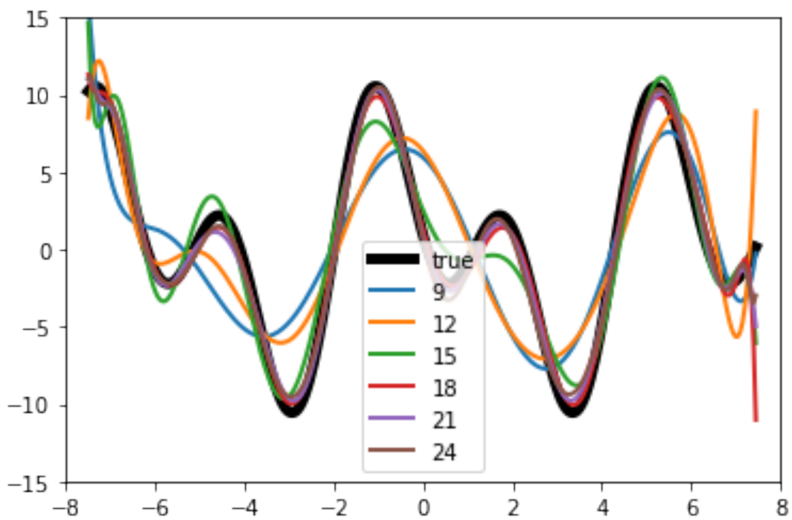


In [186]:
```
#Discussion: Choice of d=18nwill generalize the best as it gives the m
inimum validation and test error.
#validationErr - minimum 31.90(d=18)
#testErr- minimum 25.80(d=18)
```

Finally, let's visualize each learned model.

In [188]:
```
plt.figure()
plt.plot(x_true, y_true, marker='None', linewidth=5, color='k')

for d in range(9, 25, 3):
  X_d = polynomial_transform(x_true, d)
  y_d = X_d @ w[d]
  plt.plot(x_true, y_d, marker='None', linewidth=2)

plt.legend(['true'] + list(range(9, 25, 3)))
plt.axis([-8, 8, -15, 15])
```

Out[188]: [-8, 8, -15, 15]



2. **Regression with Radial Basis Functions**, 70 points

In the previous case, we considered a nonlinear extension to linear regression using a linear combination of polynomial basis functions, where each basis function was introduced as a feature $\phi(x) = x^k$. Now, we consider Gaussian radial basis functions of the form:

$$\phi(\mathbf{x}) = e^{-\gamma\,(x-\mu)^2}\,,$$

whose shape is defined by its center μ and its width $\gamma > 0$. In the case of polynomial basis regression, the user's choice of the dimension d determined the transformation and the model. For radial basis regression, we have to contend with deciding how many radial basis functions we should have, and what their center and width parameters should be. For simplicity, let's assume that $\gamma = 0.1$ is fixed. Instead of trying to identify the number of radial basis functions or their centers, we can treat **each data point as the center of a radial basis function**, which means that the model will be:

$$f(x) = [w_0,\ w_1,\ w_2 \ldots, w_n]^T \begin{bmatrix} e^{-\gamma\,(x-x_1)^2} \\ e^{-\gamma\,(x-x_2)^2} \\ e^{-\gamma\,(x-x_2)^2} \\ \ldots \\ e^{-\gamma\,(x-x_n)^2} \end{bmatrix}.$$

This transformation uses radial basis functions centered around data points $e^{-\gamma\,(x-x_i)^2}$ and each basis function has a corresponding weight w_i associated with it, for all $i = 1, \ldots, n$. We transform each univariate data point x_i into into a multivariate (n-dimensional) data point via $\phi(x_j) \rightarrow [\ldots, e^{-\gamma\,(x_j-x_i)^2}, \ldots]$. When this transformation is applied to every data point, it produces the **radial-basis kernel**:

$$\Phi = \begin{bmatrix} 1 & e^{-\gamma\,(x_1-x_2)^2} & e^{-\gamma\,(x_1-x_3)^2} & \ldots & e^{-\gamma\,(x_1-x_n)^2} \\ e^{-\gamma\,(x_2-x_1)^2} & 1 & e^{-\gamma\,(x_2-x_3)^2} & \ldots & e^{-\gamma\,(x_2-x_n)^2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e^{-\gamma\,(x_n-x_1)^2} & e^{-\gamma\,(x_n-x_2)^2} & e^{-\gamma\,(x_n-x_3)^2} & \cdots & 1 \end{bmatrix}.$$

a. (15 points)

Complete the Python function below that takes univariate data as input and computes a radial-basis kernel. This transforms one-dimensional data into n-dimensional data in terms of Gaussian radial-basis functions centered at each data point and allows us to model nonlinear (kernel) regression.

In [189]:
```
# X float(n, ): univariate data
# B float(n, ): basis functions
# gamma float : standard deviation / scaling of radial basis kernel
import numpy as np
def radial_basis_transform(X, B, gamma=0.1):
    res=[]
    for i in range(len(X)):
        new=[]
        for j in range(len(B)):
            new.append(-gamma*((X[i]-B[j])**2))
        res.append(new)
    return np.exp(res)
```

b. (15 points)

Complete the Python function below that takes a radial-basis kernel matrix Φ, the labels $\mathbf{y}$, and a regularization parameter $\lambda > 0$ as input and learns weights via **ridge regression**. Specifically, given a radial-basis kernel matrix Φ, implement the computation of $\mathbf{w} = \left(\Phi^T\Phi + \lambda I_n\right)^{-1} \Phi^T \mathbf{y}$.

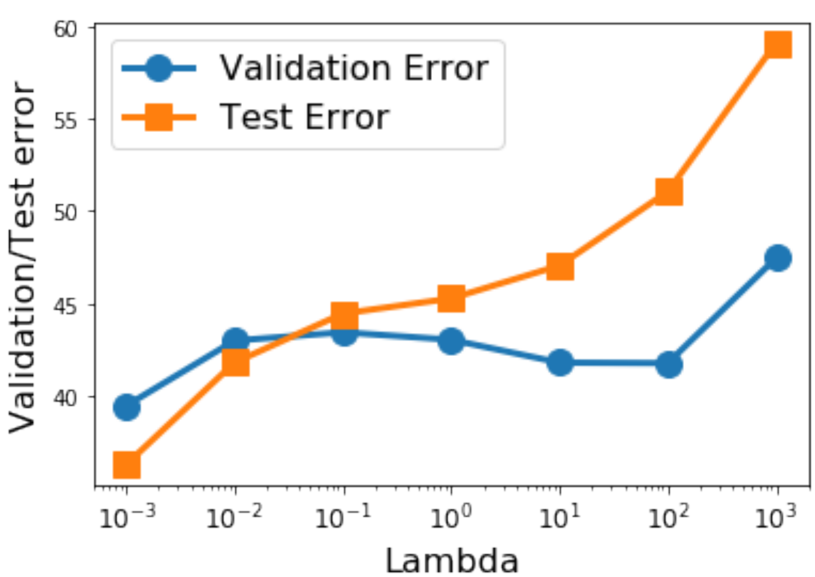
In [190]:
```
# Phi float(n, d): transformed data
# y   float(n,  ): labels
# lam float      : regularization parameter
import numpy as np
def train_ridge_model(Phi, y, lam):
    a1=np.dot(Phi.transpose(),Phi)
    a2=(lam*np.identity(len(Phi.transpose())))
    a3=np.add(a1,a2)
    res=np.linalg.inv(a3)
    return np.dot(res,np.dot(Phi.transpose(),y))
```

c. (30 points)

As before, we can explore the tradeoff between **fit and complexity** by varying $\lambda \in [10^{-3}, 10^{-2} \cdots, 1, \cdots 10^{3}]$. For each model, train using the transformed training data (Φ) and evaluate its performance on the transformed validation and test data. Plot two curves: (i) λ vs. validation error and (ii) λ vs. test error, as above.

What are some ideal values of λ?

In [191]:
```
w = {}                  # Dictionary to store all the trained models
validationErr = {}      # Validation error of the models
testErr = {}            # Test error of all the models
lam=10**-3
while lam<=10**3:  # Iterate over polynomial degree
    Phi_trn = radial_basis_transform(X_trn,X_trn)                     # Tr
ansform training data into d dimensions
    w[lam] = train_ridge_model(Phi_trn, y_trn,lam)
# Learn model on training data

    Phi_val = radial_basis_transform(X_val,X_trn)                     # Tr
ansform validation data into d dimensions
    validationErr[lam] = evaluate_model(Phi_val, y_val, w[lam])  # Eva
luate model on validation data

    Phi_tst = radial_basis_transform(X_tst, X_trn)               # Transfo
rm test data into d dimensions
    testErr[lam] = evaluate_model(Phi_tst, y_tst, w[lam])  # Evaluate
model on test data
    lam=lam*10
# Plot all the models
plt.figure()
plt.plot(validationErr.keys(), validationErr.values(), marker='o', lin
ewidth=3, markersize=12)
plt.plot(testErr.keys(), testErr.values(), marker='s', linewidth=3, ma
rkersize=12)
plt.xlabel('Lambda', fontsize=16)
plt.ylabel('Validation/Test error', fontsize=16)
plt.xticks(list(validationErr.keys()), fontsize=12)
plt.legend(['Validation Error', 'Test Error'], fontsize=16)
plt.xscale("log")
#plt.axis([2, 25, 15, 60])
```



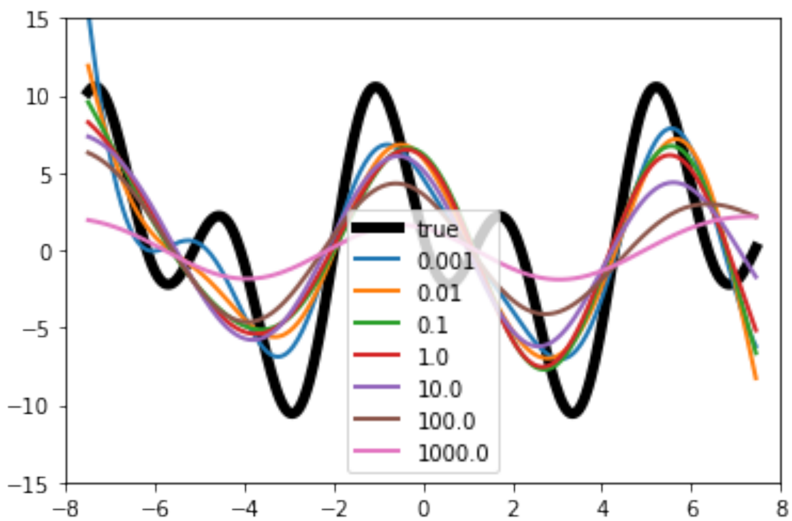
d. (10 points, Discussion)

Plot the learned models as well as the true model similar to the polynomial basis case above. How does the linearity of the model change with λ?

In [192]:

```
plt.figure()
plt.plot(x_true, y_true, marker='None', linewidth=5, color='k')
lam=10**-3
this_list=list()
while lam<=10**3:
  this_list.append(lam)
  X_lam = radial_basis_transform(x_true, X_trn)
  y_lam = X_lam @ w[lam]
  #print(w[lam])
  plt.plot(x_true, y_lam, marker='None', linewidth=2)
  lam=lam*10
plt.legend(['true'] + this_list)
plt.axis([-8, 8, -15, 15])
```

Out[192]: [-8, 8, -15, 15]



In [193]:

```
#As lambda increases, linearity of model increases and vice versa.
#minimum error is on lambda=10^-3
#validationErr- minimum 46.160 on lambda=10^-3
#testErr- minimum 37.50 on lambda=10^-3
```