**CS6301.010: Machine Learning for Engineers and Scientists (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 9:59am 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 [320]: # The true function
          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 [321]: 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 dimension
          X0=X
          e = np.random.normal(0.0, 5.0, n)       # Random Gaussian noise
          y = f_true(X) + e                       # True labels with noise
```
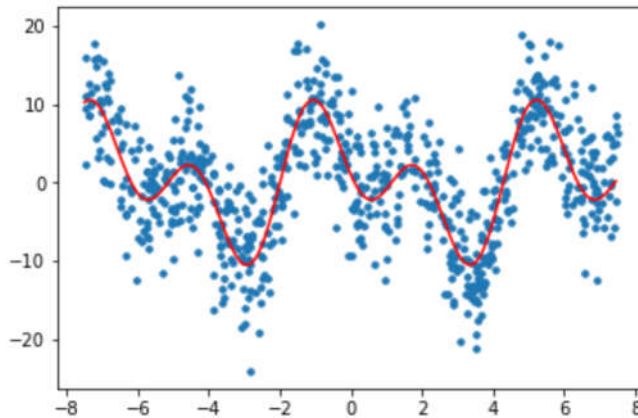
Now, we plot the raw data as well as the true function (without noise).

```
In [322]:  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[322]: [<matplotlib.lines.Line2D at 0x2d43761f9b0>]

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**: $\mathcal{D}_{trn}$ consists of the actual training examples that will be used to **train the model**;
- **Validation set**: $\mathcal{D}_{val}$ 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**: $\mathcal{D}_{tst}$ 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: $\mathcal{D}_{trn} = 60\%$ of $\mathcal{D}$, $\mathcal{D}_{val} = 10\%$ of $\mathcal{D}$ and $\mathcal{D}_{tst} = 30\%$ of $\mathcal{D}$.

```
In [323]:  # 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 set

           # First, we use train_test_split to partition (X, y) into training and test set
           s
           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) into traini
           ng 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[323]: <matplotlib.collections.PathCollection at 0x2d438a81a58>
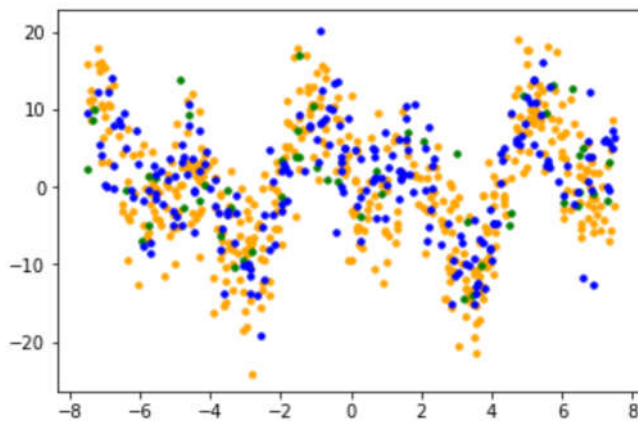
1. **Regression with Polynomial Basis Functions**, 30 points.

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**:

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

a. (10 points)

Complete the Python function below that takes univariate data as input and computes a Vandermonde matrix of dimension d. This transforms one-dimensional data into d-dimensional data in terms of the polynomial basis and allows us to model regression using a d-degree polynomial.

```
In [324]:  # X float(n, ): univariate data
           # d int: degree of polynomial
           import numpy as np
           from numpy import vander
           def polynomial_transform(X, d):
               n=len(X)
               Phi=np.zeros((n,d))
               for i in range(0,n):
                   exp=0
                   for j in range(0,d):
                       Phi[i][j]=X[i]**j
               return(Phi)
```

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 [325]: # Phi float(n, d): transformed data
          # y   float(n,  ): labels
          import numpy as np
          from numpy.linalg import inv
          def train_model(Phi, y):
              a= np.dot(np.transpose(Phi),Phi)
              b= np.dot(inv(a),np.transpose(Phi))
              w = np.dot(b,y)
              #print(w)
              return w
```

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, $\epsilon_{\mathsf{MSE}} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \mathbf{w}^T \Phi_i\right)^2$.

```
In [326]: # 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):
              mse = 0
              num=len(y)
              for i in range(num):
                  e= y[i]-np.dot((np.transpose(w)),Phi[i])
                  mse = mse + e*e
                  mse=mse/num
              #print(mse)
              return (mse)
```

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 [329]: 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)                    # Transform traini
          ng data into d dimensions
              w[d] = train_model(Phi_trn, y_trn)                          # Learn model on t
          raining data

              Phi_val = polynomial_transform(X_val, d)                    # Transform valida
          tion data into d dimensions
              validationErr[d] = evaluate_model(Phi_val, y_val, w[d])  # Evaluate model o
          n validation data

              Phi_tst = polynomial_transform(X_tst, d)                   # Transform test data in
          to d dimensions
              testErr[d] = evaluate_model(Phi_tst, y_tst, w[d])  # Evaluate model on test
          data
          #print(validationErr.values())
          #print(testErr.values())
          # Plot all the models
          plt.figure()
          plt.plot(validationErr.keys(), validationErr.values(), marker='o', linewidth=3,
          markersize=12)
          plt.plot(testErr.keys(), testErr.values(), marker='s', linewidth=3, markersize=
          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, 0, 2.5])
```
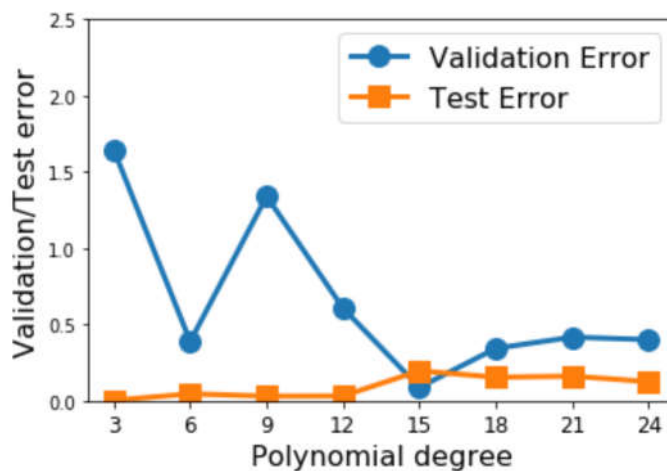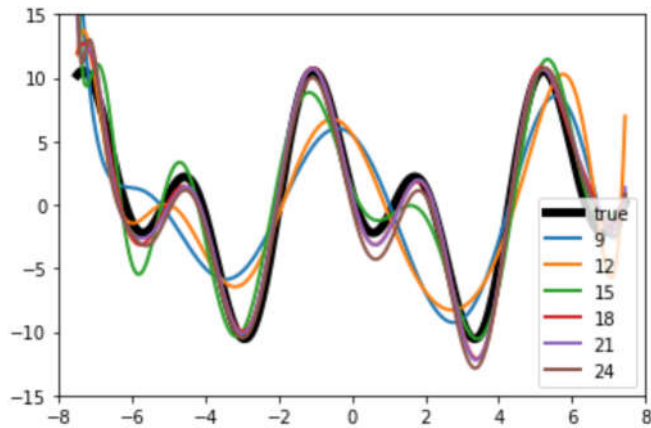
Out[329]: [2, 25, 0, 2.5]

Finally, let's visualize each learned model.

```
In [330]: 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[330]: [-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**:

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

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.

```python
# 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=X0, gamma=0.1):
    n=len(X)
    k=len(B)
    rbkernel=np.zeros((n,k))
    for i in range(0,n):
        for j in range(0,k):
            rbkernel[i][j]=np.exp(-gamma*np.square((X[i]-B[j])))
    return(rbkernel)
```
In [331]:

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 $\mathbf{w}\,=\,\left(\Phi^T\Phi + \lambda I_n\right)^{-1}\Phi^T\mathbf{y}$.

```python
In [332]: # Phi float(n, d): transformed data
          # y   float(n,  ): labels
          # lam float       : regularization parameter
          import numpy as np
          from numpy.linalg import inv
          def train_ridge_model(Phi, y, lam):
              #print(np.shape(Phi))
              a= np.dot(np.transpose(Phi),Phi)
              b =np.identity(len(np.transpose(Phi)))
              c= np.dot(inv(a+lam*b),np.transpose(Phi))
              w= np.dot(c,y)
              #print(np.shape(w))
              return w
```

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?

```python
# Phi float(n, d): transformed data
# y   float(n,  ): labels
# lam float       : regularization parameter
import numpy as np
from numpy.linalg import inv
def train_ridge_model(Phi, y, lam):
    #print(np.shape(Phi))
    a= np.dot(np.transpose(Phi),Phi)
    b =np.identity(len(np.transpose(Phi)))
    c= np.dot(inv(a+lam*b),np.transpose(Phi))
    w= np.dot(c,y)
    #print(np.shape(w))
    return w
```

```python
In [335]: w = {}                     # Dictionary to store all the trained models
          validationErr = {}         # Validation error of the models
          testErr = {}               # Test error of all the models

          for i in range(-5,4):   # Iterate over polynomial degree
              lam = 10**i
              Phi_trn = radial_basis_transform(X_trn,X0)    # Transform training data into
          d dimensions
              #print(Phi_trn.shape)
              w[lam] = train_ridge_model(Phi_trn, y_trn, lam) # Learn model on training d
          ata
              #print(w[lam])
              Phi_val = radial_basis_transform(X_val,X0)    # Transform validation data in
          to d dimensions
              #print(Phi_val.shape)
              validationErr[lam] = evaluate_model(Phi_val, y_val, w[lam])  # Evaluate mod
          el on validation data

              Phi_tst = radial_basis_transform(X_tst,X0)    # Transform test data into d d
          imensions
              testErr[lam] = evaluate_model(Phi_tst, y_tst, w[lam])  # Evaluate model on
          test data
          #print (validationErr.values())
          #print (validationErr.keys())
          #print (testErr.values())
          #print (testErr.keys())

          #Plot all the models
          plt.figure()
          plt.semilogx(validationErr.keys(),validationErr.values(), marker='o', linewidth
          =3, markersize=12)
          plt.semilogx(testErr.keys(),testErr.values(), marker='s', linewidth=3, markersi
          ze=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.axis([0.00001,1000, 0, 3])
```
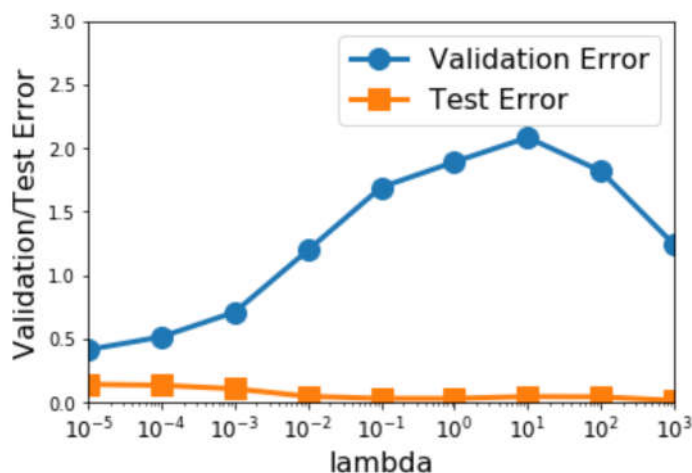
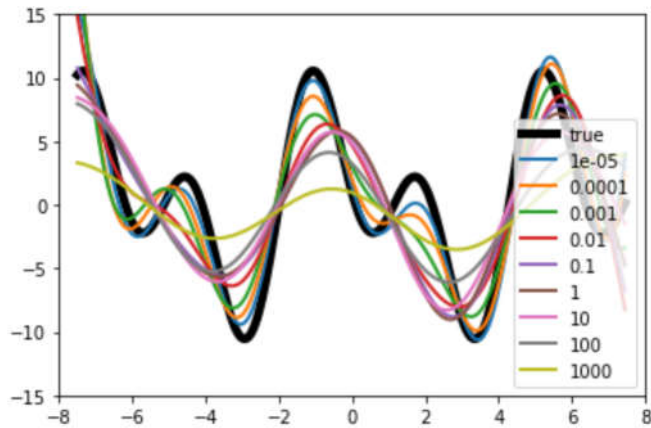Out[335]: [1e-05, 1000, 0, 3]

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

          for lam in [10**i for i in range(-5,4)]:
              X_d = radial_basis_transform(x_true,X0)
              y_d = X_d@w[lam]
              plt.plot(x_true, y_d, marker='None', linewidth=2)

          plt.legend(['true'] + [0.00001,0.0001,0.001, 0.01, 0.1, 1, 10, 100, 1000])
          plt.axis([-8, 8, -15, 15])
```

Out[334]: [-8, 8, -15, 15]


```
In [ ]:
```