Assignment 1

We explored linear models the last lecture. We will strengthen this understanding by implementing linear and logistic regression models as part of the assignment.

Section I - Linear Regression

We will implement a linear regression model to fit a curve to some data. Since the data is nonlinear, we will implement polynomial regression and use ridge regression to implement the best possible fit.

1. Load Data and Visualize

Let us load a dataset of points (x,y). As a first step, let's import the required libraries followed by the dataset.

In [1]:

```
import numpy as np
from datasets import ridge_reg_data

# Libraries for evaluating the solution
import pytest
import numpy.testing as npt
import random
random.seed(1)
np.random.seed(1)

train_X, train_Y, test_X, test_Y = ridge_reg_data() # Pre-defined function for loading the dataset
train_Y = train_Y.reshape(-1,1) # reshaping from (m,) -> (m,1)
test_Y = test_Y.reshape(-1,1)
print('train_X.shape is ', train_X.shape)
print('train_Y.shape is ', train_Y.shape)
print('test_X.shape is ', test_X.shape)
print('test_Y.shape is ', test_Y.shape)
```

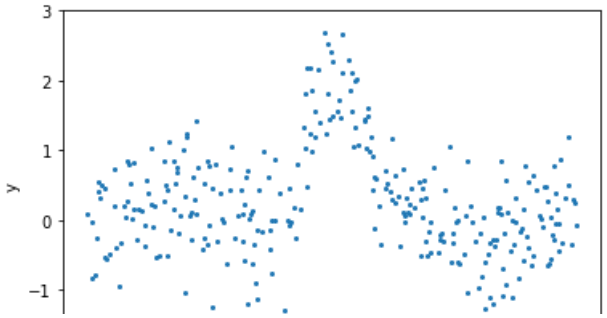
```
train_X.shape is  (300, 1)
train_Y.shape is  (300, 1)
test_X.shape is  (200, 1)
test_Y.shape is  (200, 1)
```

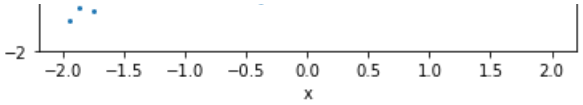
Visualize Data

The dataset is split into train and test sets. The train set consists of 300 samples and the test set consists of 200 samples. We will use scatter plot to visualize the relationship between the 'x' and 'y'. Lets visualize the data using the scatter plot from matplotlib.

In [3]:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.scatter(train_X,train_Y,marker='o',s=4)
plt.ylim(-2, 3)
plt.xlabel('x')
plt.ylabel('y');
```





Linear Regression - Polynomial Transformation

Using the train data we hope to learn a relationship mapping x to y. We can evaluate this mapping using the test data. Linear regression will try to fit a straight line (linear relation) mapping x to y. However, we observe the x and y do not have a linear relationship. A straight line will not be a good fit. We need a non-linear mapping (curve) between x and y.

We discussed in the lecture that nonlinear regression can be achieved by transforming the scalar x to a high dimension sample and performing linear regression with the transformed data. We can transform x into a d dimensional vector ($d \geq 2$) in order to perform nonlinear regression. For example, $d = 5$ transforms x into a $(d+1)$ dimension vector $[1,x,x^2,x^3,x^4,x^5]^\top$, where x^k is x raised to k. In vectorized notation, the dataset X is transformed to $\Phi(X)$ of dimension $m \times (d+1)$, where m is the number of samples.

Every scalar x is converted into a $(d+1)$ dimension vector, $[1,x_1,x_2,x_3,\ldots,x_d]^\top$. We can now perform linear regression in $(d+1)$ dimensions. \begin{equation*} y = \Phi(x)\boldsymbol{\theta} = \theta_0 + x_1\theta_1 + ... + x_{d-1}\theta_{d-1} + x_d\theta_d\\ \end{equation*} In the above equation, y is the target variable, $\boldsymbol{\theta} = [\theta_{0},.., \theta_{d}]^\top$ are the parameters/weights of the model, $\Phi(x) = [1,x_{1},.., x_{d}]$ is the transformed data point in the row vector format, where x_k is the k^{th} component.

In the vectorized notation, the linear regression for m samples is written as $\hat{Y} = \Phi(X)\boldsymbol{\theta}$, where $\Phi(X)$ has the data points as row vectors and is of dimensions $m \times (d+1)$,
\begin{align} \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)}\\ \vdots\\ \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} 1 & x^{(1)}_1 & x^{(1)}_2 & \ldots & x^{(1)}_d\\ 1 & x^{(2)}_1 & x^{(2)}_2 & \ldots & x^{(2)}_d\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ 1 & x^{(m)}_1 & x^{(m)}_2 & \ldots & x^{(m)}_d\\ \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1\\ \vdots \\ \theta_d\\ \end{bmatrix} \end{align}

X - is the Design matrix of dimension $m \times (d+1) $, where m is the number of samples and d is the degree of the polynomial that we are trying to fit. The first column of 1's in the design matrix will account for the bias , resulting in $d+1$ dimensions

Y - Vector of the prediction labels of dimension $m \times 1 $. Lets implement a function to achieve this transformation.

In [4]:

```
def poly_transform(X,d):
    '''
    Function to transform scalar values into (d+1)-dimension vectors.
    Each scalar value x is transformed a vector [1,x,x^2,x^3, ... x^d].

    Inputs:
        X: vector of m scalar inputs od shape (m, 1) where each row is a scalar input x
        d: number of dimensions

    Outputs:
        Phi: Transformed matrix of shape (m, (d+1))
    '''
    Phi = np.ones((X.shape[0],1))
    for i in range(1,d+1):
        col = np.power(X,i)
        Phi = np.hstack([Phi,col])
    return Phi
```

Linear Regression - Objective Function (5 Points)

Let us define the objective function that will be optimized by the linear regression model. \begin{equation*} L\big(\Phi(X),Y,\theta\big) = \big(Y-\Phi(X)\boldsymbol{\theta}\big)^\top \big(Y-\Phi(X)\boldsymbol{\theta}\big) \end{equation*}

Here, $\Phi(X)$ is the design matrix of dimensions (m \times (d+1)) and Y is the m dimension vector of labels. θ is the $(d+1)$ dimension vector of weight parameters.

Hint: You may want to use numpy.dot

In [17]:

```
def lin_reg_obj(Y,Phi,theta):
    '''
    Objective function to estimate loss for the linear regression model.
```

```
    Inputs:
        Phi: Design matrix of dimensions (m, (d+1))
        Y: ground truth labels of dimensions (m, 1)
        theta: Parameters of linear regression of dimensions ((d+1),1)

    outputs:
        loss: scalar loss
    '''
    # your code here
    Yhat = np.dot(Phi, theta)
    diff = Y - Yhat
    loss = np.dot(np.transpose(diff),diff)

    return loss
```

In [18]:

```
# Contains hidden tests

random.seed(1)
np.random.seed(1)
m1 = 10;
d1 = 5;
X_t = np.random.randn(m1,1)
Y_t = np.random.randn(m1,1)
theta_t = np.random.randn((d1+1),1)
PHI_t = poly_transform(X_t,d1)
loss_est = lin_reg_obj(Y_t,PHI_t,theta_t)
```

Linear Regression - Closed Form Solution (10 Points)

Let us define a closed form solution to the objective function. Feel free to revisit the lecture to review the topic. Closed form solution is given by,
\begin{equation*} \theta = \big(\Phi(X)^\top\Phi(X)\big)^{-1}\Phi(X)^\top Y \end{equation*}

Here $\Phi(X)$ is the $(m \times (d+1))$ dimension design matrix obtained using *poly_transform* function defined earlier and Y are the ground truth labels of dimensions $(m \times 1)$.

Hint: You may want to use numpy.linalg.inv and numpy.dot.

In [40]:

```
#Closed form solution
def lin_reg_fit(Phi_X,Y):
    '''
    A function to estimate the linear regression model parameters using the closed form solution.
    Inputs:
        Phi_X: Design matrix of dimensions (m, (d+1))
        Y: ground truth labels of dimensions (m, 1)

    Outputs:
        theta: Parameters of linear regression of dimensions ((d+1),1)
    '''
    # your code here
    first = np.linalg.inv(np.dot(np.transpose(Phi_X), Phi_X))
    second = np.dot(first, np.transpose(Phi_X))
    theta = np.dot(second, Y)

    return theta
```

In [24]:

```
# Contains hidden tests

random.seed(1)
np.random.seed(1)
m1 = 10;
d1 = 5;
X_t = np.random.randn(m1,1)
Y_t = np.random.randn(m1,1)
PHI_t = poly_transform(X_t,d1)
theta_est = lin_reg_fit(PHI_t,Y_t)
```

Metrics for Evaluation (10 points)

We will evaluate the goodness of our linear regression model using root mean square error. This compares the difference between the estimate Y-labels and the groundth truth Y-labels. The smaller the RMSE value, better is the fit.

1. RMSE (Root Mean Squared Error) \begin{equation*} \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_pred^{(i)} - y^{(i)})^{2} } \end{equation*}

Hint: You may want to use:
numpy.sqrt, numpy.sum or numpy.dot.

In [41]:

```
def get_rmse(Y_pred,Y):
    '''
    function to evaluate the goodness of the linear regression model.

    Inputs:
        Y_pred: estimated labels of dimensions (m, 1)
        Y: ground truth labels of dimensions (m, 1)

    Outputs:
        rmse: root means square error
    '''
    diff = Y_pred-Y
    s = np.dot(np.transpose(diff),diff)
    rmse = np.sqrt(s/Y.shape[0])

    return rmse
```

In [42]:

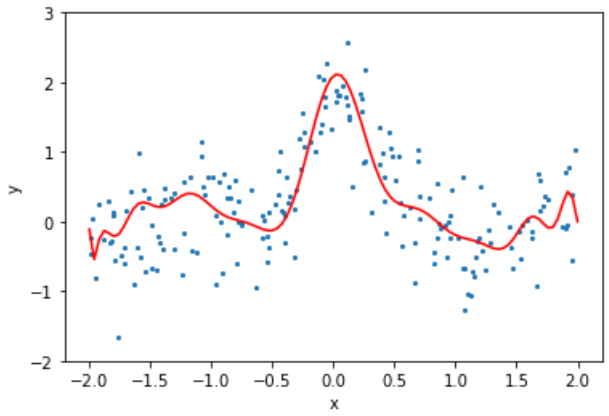
```
# Contains hidden tests

random.seed(1)
np.random.seed(1)
m1 = 50
Y_Pred_t = np.random.randn(m1,1)
Y_t = np.random.randn(m1,1)
rmse_est = get_rmse(Y_Pred_t,Y_t)
```

Let's visualize the nonlinear regression fit and the RMSE evaluation error on the test data

In [43]:

```
d = 20
Phi_X_tr = poly_transform(train_X,d)
theta = lin_reg_fit(Phi_X_tr,train_Y)
#Estimate the prediction on the train data
Y_Pred_tr = np.dot(Phi_X_tr,theta)
rmse = get_rmse(Y_Pred_tr,train_Y)
print('Train RMSE = ', rmse)

#Perform the same transform on the test data
Phi_X_ts = poly_transform(test_X,d)
#Estimate the prediction on the test data
Y_Pred_ts = np.dot(Phi_X_ts,theta)
#Evaluate the goodness of the fit
rmse = get_rmse(Y_Pred_ts,test_Y)
print('Test RMSE = ', rmse)

import matplotlib.pyplot as plt
%matplotlib inline
plt.scatter(test_X,test_Y,marker='o',s=4)
# Sampling more points to plot a smooth curve
px = np.linspace(-2,2,100).reshape(-1,1)
PX = poly_transform(px,d)
py = np.dot(PX,theta)
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(-2, 3)
plt.plot(px,py,color='red');
```

```
Train RMSE =  [[0.51363406]]
Test RMSE =  [[0.50376918]]
```



2. Ridge Regression

The degree of the polynomial regression is $d=10$. Even though the curve appears to be smooth, it may be fitting to the noise. We will use Ridge Regression to get a smoother fit and avoid-overfitting. Recall the ridge regression objective form:
\begin{equation*} L\big(\Phi(X),Y,\theta, \lambda\big) = \big(Y-\Phi(X)\boldsymbol{\theta}\big)^\top \big(Y-\Phi(X)\boldsymbol{\theta}\big) + \lambda^2\boldsymbol{\theta}^\top\boldsymbol{\theta} \end{equation*}

where, $\lambda\geq 0$ is the regularization parameter. Larger the value of λ, the more smooth the curve. The closed form solution to the objective is give by:
\begin{equation*} \theta = \big(\Phi(X)^\top\Phi(X) + \lambda^2I_d \big)^{-1}\Phi(X)^\top Y \end{equation*}

Here, I_d is the identity matrix of dimensions $((d+1) \times (d+1))$, $\Phi(X)$ is the $(m \times (d+1))$ dimension design matrix obtained using *poly_transform* function defined earlier and Y are the ground truth labels of dimensions $(m \times 1)$.

Ridge Regression Closed Form Solution (5 points)

Similar to Linear regression, lets implement the closed form solution to ridge regression.

In [46]:

```
def ridge_reg_fit(Phi_X,Y,lamb_d):
    '''
    A function to estimate the ridge regression model parameters using the closed form solution.
    Inputs:
        Phi_X: Design matrix of dimensions (m, (d+1))
        Y: ground truth labels of dimensions (m, 1)
        lamb_d: regularization parameter

    Outputs:
        theta: Parameters of linear regression of dimensions ((d+1),1)
    '''
    #Step 1: get the dimension dplus1 using Phi_X to create the identity matrix $I_d$
    I_d=np.identity(Phi_X.shape[1])
    #Step 2: Estimate the closed form solution similar to *linear_reg_fit* but now includethe lamb
_d**2*I_d term
    # your code here
    ridge_regression=lamb_d**2*I_d
    step01=np.dot(np.transpose(Phi_X), Phi_X)
    step02=np.linalg.inv(step01+ridge_regression)
    step03=np.dot(step02, np.transpose(Phi_X))
    theta=np.dot(step03, Y)
    return theta
```

In [47]:

```
# Contains hidden tests

random.seed(1)
np.random.seed(1)
```

```
m1 = 10;
d1 = 5;
lamb_d_t = 0.1
X_t = np.random.randn(m1,1)
Y_t = np.random.randn(m1,1)
PHI_t = poly_transform(X_t,d1)
theta_est = ridge_reg_fit(PHI_t,Y_t,lamb_d_t)
```

Cross Validation to Estimate (λ)

In order to avoid overfitting when using a high degree polynomial, we have used **ridge regression**. We now need to estimate the optimal value of λ using **cross-validation**.

We will obtain a generic value of λ using the entire training dataset to validate. We will employ the method of **k-fold cross validation**, where we split the training data into k non-overlapping random subsets. In every cycle, for a given value of λ, $(k-1)$ subsets are used for training the ridge regression model and the remaining subset is used for evaluating the goodness of the fit. We estimate the average goodness of the fit across all the subsets and select the $lambda$ that results in the best fit.

It is easier to shuffle the index and slice the training into required number of segments, than processing the complete dataset. The below function **k_val_ind$()$** returns a 2D list of indices by spliting the datapoints into 'k_fold' sets

Refer the following documentation for splitting and shuffling:

- https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.shuffle.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.split.html

In [48]:

```
def k_val_ind(index,k_fold,seed=1):
    '''
    Function to split the data into k folds for cross validation. Returns the indices of the data
points
    belonging to every split.

    Inputs:
        index: all the indices of the training
        k_fold: number of folds to split the data into

    Outputs:
        k_set: list of arrays with indices
    '''
    np.random.seed(seed)
    np.random.shuffle(index) # Shuffle the indices
    k_set = np.split(index,k_fold) # Split the indices into 'k_fold'
    return k_set
```

K- Fold Cross Validation (10 Points)

Let's now implement k-fold cross validation.

In [64]:

```
def k_fold_cv(k_fold,train_X,train_Y,lamb_d,d):
    '''
    Function to implement k-fold cross validation.
    Inputs:
        k_fold: number of validation subsests
        train_X: training data of dimensions (m, 1)
        train_Y: ground truth training labels
        lamb_d: ridge regularization lambda parameter
        d: polynomial degree

    Outputs:
        rmse_list: list of root mean square errors (RMSE) for k_folds
    '''
    index = np.arange(train_X.shape[0]) # indices of the training data
```

```
    index = np.arange(train_X.shape[0]) # Indices of the training data
    k_set = k_val_ind(index,k_fold) # pre-defined function to shuffle and split indices

    Phi_X = poly_transform(train_X, d) #transform all the data to (m,(d+1)) dimensions
    rmse_list = []
    for i in range(k_fold):
        ind = np.zeros(train_X.shape[0], dtype=bool) # binary mask
        ind[k_set[i]] = True # validation portion is indicated
        t_Phi_X = Phi_X[~ind]
        t_Y=train_Y[~ind]
        v_Phi_X = Phi_X[ind]
        v_Y=train_Y[ind]
        #Note: Eg. train_X[ind] -> validation set, train_X[~ind] -> training set
        # Write your answer inside the 'for' loop
        # Note: Phi_X[~ind,:] is training subset and Phi_X[ind,:] is validation subset. Similary
for the train and validation labels.
        # Step 1: Estimate the theta parameter using ridge_reg_fit with the training subset,
training labels and lamb_d
        t_theta = ridge_reg_fit(t_Phi_X, t_Y,lamb_d)
        # Step 2: Estimate the prediction Y_pred over the validation as a dot product over
Phi_X[ind,:] and theta
        v_Y_pred = np.dot(v_Phi_X, t_theta)
        # Step 3: use 'get_rmse' function to determine rmse using Y_pred and train_Y[ind]
        rmse = get_rmse(v_Y_pred, v_Y)
        # your code here

        rmse_list.append(rmse)
    return rmse_list
```

In [65]:

```
# Contains hidden tests

np.random.seed(1)
m1 = 20;
d1 = 5;
k_fold_t = 5 # number of portions to split the training data
lamb_d_t = 0.1
X_t = np.random.randn(m1,1)
Y_t = np.random.randn(m1,1)

rmse_list_est = k_fold_cv(k_fold_t,X_t,Y_t,lamb_d_t,d1)
```

Let us select the value of λ that provides the lowest error based on RMSE returned by the 'k_fold_cv' function.

In this example, we will choose the best value of λ among 6 values.

In [66]:

```
k_fold = 5
l_range = [0,1e-3,1e-2,1e-1,1,10] # The set of lamb_d parameters used for validation.
th = float('inf')
for lamb_d in l_range:
    print('lambda:'+str(lamb_d))
    rmse = k_fold_cv(k_fold,train_X,train_Y,lamb_d,d)
    print("RMSE: ",rmse)
    print("**************")
    mean_rmse = np.mean(rmse)
    if mean_rmse<th:
        th = mean_rmse
        l_best = lamb_d

print("Best value for the regularization parameter(lamb_d):",l_best)
```

```
lambda:0
RMSE:  [array([[0.90055518]]), array([[0.59950635]]), array([[0.4889937]]), array([[0.57349943]]),
array([[0.57782947]])]
*************
lambda:0.001
RMSE:  [array([[0.92547771]]), array([[0.60188953]]), array([[0.48867704]]),
array([[0.57084667]]), array([[0.57846187]])]
*************
lambda:0.01
RMSE:  [array([[1.04590449]]), array([[0.62518222]]), array([[0.49331378]]),
```

```
array([[0.55706474]]), array([[0.5899622]])]
*************
lambda:0.1
RMSE:  [array([[0.82614745]]), array([[0.64652459]]), array([[0.49033082]]),
array([[0.56609503]]), array([[0.59456687]])]
*************
lambda:1
RMSE:  [array([[0.67996651]]), array([[0.68866935]]), array([[0.56473576]]),
array([[0.63930749]]), array([[0.64703293]])]
*************
lambda:10
RMSE:  [array([[0.7335261]]), array([[0.69930692]]), array([[0.75561325]]), array([[0.79926087]]),
array([[0.8199075]])]
*************
Best value for the regularization parameter(lamb_d): 0.1
```

Evaluation on the Test Set (10 Points)

As discussed in previous section, we will present the final evaluation of the model based on the test set.

In [67]:

```
lamb_d = l_best

# Step 1: Create Phi_X using 'poly_transform(.)' on the train_X and d=20
Phi_X = poly_transform(train_X, 20)
# Step 2: Estimate theta using ridge_reg_fit(.) with Phi_X, train_Y and the best lambda
theta = ridge_reg_fit(Phi_X, train_Y, lamb_d)
# Step 3: Create Phi_X_test using 'poly_transform(.)' on the test_X and d=20
Phi_X_test = poly_transform(test_X, 20)
# Step 4: Estimate the Y_Pred for the test data using Phi_X_test and theta
Y_Pred = np.dot(Phi_X_test, theta)
# Step 5: Estimate rmse using get_rmse(.) on the Y_Pred and test_Y
rmse = get_rmse(Y_Pred, test_Y)
# your code here


print("RMSE on test set is "+str(rmse))
```

```
RMSE on test set is [[0.49850101]]
```

In [68]:

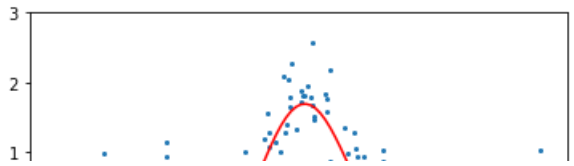
```
# Contains hidden tests checking for rmse < 0.5
```

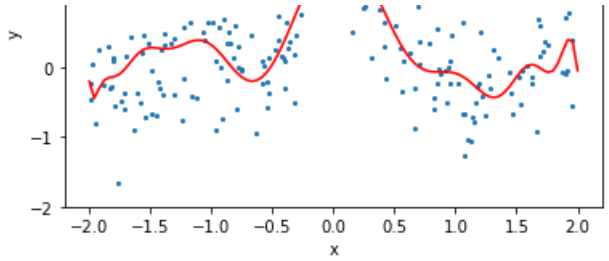
Let's visualize the model's prediction on the test data set.

In [69]:

```
print('Test RMSE = ', rmse)

%matplotlib inline
plt.scatter(test_X,test_Y,marker='o',s=4)
# Sampling more points to plot a smooth curve
px = np.linspace(-2,2,100).reshape(-1,1)
PX = poly_transform(px,d)
py = np.dot(PX,theta)
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(-2, 3)
plt.plot(px,py,color='red');
```

```
Test RMSE =  [[0.49850101]]
```





You have completed linear ridge regression and estimated the best value for the regularization parameter λ using k-fold cross validation.

Section II - Logistic Regression

Machine learning is used in medicine for assisting doctors with crucial decision-making based on dignostic data. In this assignment we will be designing a logistic regression model (single layer neural network) to predict if a subject is diabetic or not. The model will classify the subjects into two groups diabetic (Class 1) or non-diabetic (Class 0) - a binary classification model.

We will be using the 'Pima Indians Diabetes dataset' to train our model which contains different clinical parameters (features) for multiple subjects along with the label (diabetic or not-diabetic). Each subject is represented by 8 features (Pregnancies, Glucose, Blood-Pressure, SkinThickness, Insulin, BMI, Diabetes-Pedigree-Function, Age) and the 'Outcome' which is the class label. The dataset contains the results from 768 subjects.

We will be spliting the dataset into train and test data. We will train our model on the train data and predict the categories on the test data.

In [1]:

```
#importing a few libraries
import numpy as np
from datasets import pima_data
import sys
import matplotlib.pyplot as plt
import numpy.testing as npt
```

1. Load Data, Visualize and Normalize

Let us load the training and test data.

In [2]:

```
train_X,train_Y,test_X,test_Y  = pima_data()

print('train_X.shape = ', train_X.shape)
print('train_Y.shape = ', train_Y.shape)
print('test_X.shape = ', test_X.shape)
print('test_Y.shape = ', test_Y.shape)

# Lets examine the data
print('\nFew Train data examples')
print(train_X[:5, :])
print('\nFew Train data labels')
print(train_Y[:5])
```

```
train_X.shape =  (500, 8)
train_Y.shape =  (500,)
test_X.shape =  (268, 8)
test_Y.shape =  (268,)

Few Train data examples
[[6.000e+00 1.480e+02 7.200e+01 3.500e+01 0.000e+00 3.360e+01 6.270e-01
  5.000e+01]
 [1.000e+00 8.500e+01 6.600e+01 2.900e+01 0.000e+00 2.660e+01 3.510e-01
  3.100e+01]
 [8.000e+00 1.830e+02 6.400e+01 0.000e+00 0.000e+00 2.330e+01 6.720e-01
  3.200e+01]
 [1.000e+00 8.900e+01 6.600e+01 2.300e+01 9.400e+01 2.810e+01 1.670e-01
```

```
  2.100e+01]
 [0.000e+00 1.370e+02 4.000e+01 3.500e+01 1.680e+02 4.310e+01 2.288e+00
  3.300e+01]]

Few Train data labels
[1. 0. 1. 0. 1.]
```

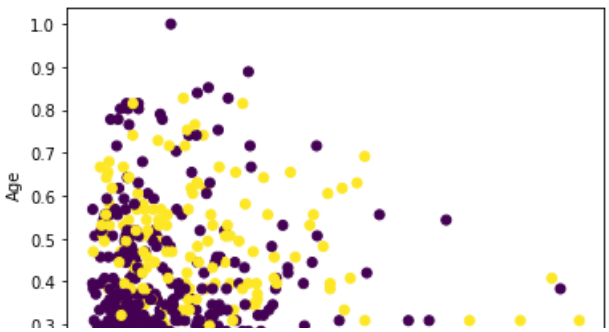
In [3]:

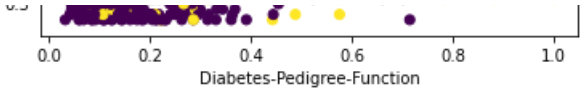
```
# We notice the data is not normalized. Lets do a simple normalization scaling to data between 0 a
nd 1
# Normalized data is easier to train using large learning rates
train_X = np.nan_to_num(train_X/train_X.max(axis=0))
test_X = np.nan_to_num(test_X/test_X.max(axis=0))

#Lets reshape the data so it matches our notation from the lecture.
#train_X should be (d, m) and train_Y should (1,m) similarly for test_X and test_Y
train_X = train_X.T
train_Y= train_Y.reshape(1,-1)

test_X = test_X.T
test_Y= test_Y.reshape(1,-1)
print('train_X.shape = ', train_X.shape)
print('train_Y.shape = ', train_Y.shape)
print('test_X.shape = ', test_X.shape)
print('test_Y.shape = ', test_Y.shape)

# Lets examine the data and verify it is normalized
print('\nFew Train data examples')
print(train_X[:, :5])
print('\nFew Train data labels')
print(train_Y[0,:5])
```

```
train_X.shape =  (8, 500)
train_Y.shape =  (1, 500)
test_X.shape =  (8, 268)
test_Y.shape =  (1, 268)

Few Train data examples
[[0.35294118 0.05882353 0.47058824 0.05882353 0.        ]
 [0.74371859 0.42713568 0.91959799 0.44723618 0.68844221]
 [0.59016393 0.54098361 0.52459016 0.54098361 0.32786885]
 [0.35353535 0.29292929 0.         0.23232323 0.35353535]
 [0.         0.         0.         0.11111111 0.19858156]
 [0.50074516 0.39642325 0.34724292 0.41877794 0.64232489]
 [0.25909091 0.14504132 0.27768595 0.06900826 0.94545455]
 [0.61728395 0.38271605 0.39506173 0.25925926 0.40740741]]

Few Train data labels
[1. 0. 1. 0. 1.]
```

In [4]:

```
#There are 8 features for each of the data points. Lets plot the data using a couple of features
fig, ax = plt.subplots()
plt.scatter(train_X[6,:],train_X[7,:], c=train_Y[0])
plt.xlabel('Diabetes-Pedigree-Function')
plt.ylabel('Age')
plt.show();
# We have plotted train_X[6,:],train_X[7,:].
# Feel free to insert your own cells to plot and visualize different variable pairs.
```





2. Quick Review of the Steps Involved in Logistic Regression Using Gradient Descent.

1. Training data X is of dimensions $(d \times m)$ where d is number of features and m is number of samples. Training labels Y is of dimensions $(1 \times m)$.
2. Initilaize logistic regression model parameters w and b where w is of dimensions $(d, 1)$ and b is a scalar. w is initialized to small random values and b is set to zero
3. Calculate Z using X and intial parameter values (w , b)
$$\begin{equation*} Z= w^\top X + b \end{equation*}$$
4. Apply the sigmoid activation to estimate A on Z,
$$\begin{equation*} A=\frac{1}{1+\text{exp}(-Z)} \end{equation*}$$
5. Calculate the loss $L()$ between predicted probabilities A and groundtruth labels Y,
$$\begin{equation*} loss = logistic\_loss(A,Y) \end{equation*}$$
6. Calculate gradient dZ (or $\frac{dL}{dZ}$),
$$\begin{equation*} dZ = (A - Y) \end{equation*}$$
7. Calculate gradients $\frac{dL}{dw}$ represented by dw, $\frac{dL}{db}$ represented by db
$$\begin{equation*} dw,db = grad\_fn(X ,dZ) \end{equation*}$$
8. Adjust the model parameters using the gradients. Here α is the learning rate.
$$\begin{equation*} w := w - \alpha.dw\\ b := b - \alpha.db \end{equation*}$$
9. Loop until the loss converges or for a fixed number of epochs. We will first define the functions **logistic_loss()** and **grad_fn()** along with other functions below.

Review

Intialize Parameters (5 Points)

we will initialize the model parameters. The weights will be initialized with small random values and bias as 0. While the bias will be a scalar, the dimension of weight vector will be $(d \times 1)$, where d is the number of features.

Hint:np.random.randn can be used here to create a vector of random integers of desired shape.

In [5]:

```
def initialize(d, seed=1):
    '''
    Function to initialize the parameters for the logisitic regression model

    Inputs:
        d: number of features for every data point
        seed: random generator seed for reproducing the results

    Outputs:
        w: weight vector of dimensions (d, 1)
        b: scalar bias value
    '''
    np.random.seed(seed)

    # NOTE: initialize w to be a (d,1) column vector instead of (d,) vector
    # Hint: initialize w to a random vector with small values. For example,
0.01*np.random.randn(.) can be used.
    #       and initialize b to scalar 0
    # your code here
    w = 0.01*np.random.randn(d,1)
    b = 0

    return w,b
```

In [ ]:

```
# Contains hidden tests
```

Sigmoid Function (5 Points)

Let's now implement Sigmoid activation function.

$$\sigma \left(z\right) = \frac{1}{1+\text{exp}(-z)}$$

where z is in the input variable. Hint: numpy.exp can be used for defining the exponential function.

In [9]:

```
def sigmoid(z):
    # your code here
    A = 1/(1+np.exp(-z))
    return A
```

In [44]:

```
# Contains hidden tests

np.random.seed(1)
d = 2
m1 = 5
X_t = np.random.randn(d,m1)
```

Logistic Loss Function (5 Points)

We will define the objective function that will be used later for determining the loss between the model prediction and groundtruth labels. We will use vectors A (activation output of the logistic neuron) and Y (groundtruth labels) for defining the loss.

$$L(A,Y) = -\frac{1}{m}\sum_{i =1}^{m}y^{(i)}\text{log} a^{(i)} + (1-y^{(i)}) \text{log}(1 - a^{(i)})$$

where m is the number of input datapoints and is used for averaging the total loss. Hint: numpy.sum and numpy.log.

In [91]:

```
def logistic_loss(A,Y):
    '''
    Function to calculate the logistic loss given the predictions and the targets.

    Inputs:
        A: Estimated prediction values, A is of dimension (1, m)
        Y: groundtruth labels, Y is of dimension (1, m)

    Outputs:
        loss: logistic loss
    '''
    m = A.shape[1]
    # your code here
    first = np.dot(np.transpose(Y), np.log(A))
    second = np.dot(np.transpose(1-Y), np.log(1-A))
    loss = -(np.sum(first+second))/m

    return loss
```

In [92]:

```
# Contains hidden tests

np.random.seed(1)
d = 2
m1 = 10
X_t = np.random.randn(d,m1)
Y_t = np.random.rand(1,m1)
Y_t[Y_t>0.5] = 1
Y_t[Y_t<=0.5] = 0
```

Gradient Function (5 Points)

Let us define the gradient function for calculating the gradients ($\frac{dl }{dw} \frac{dl }{db}$). We will use it during gradient descent

Let us define the gradient function for calculating the gradients ($\frac{dL}{dw},\frac{dL}{db}$). We will use it during gradient descent.

The gradients can be calculated as,

$$\begin{equation*} dw = \frac{1}{m}X(A- Y)^{T}\\ db = \frac{1}{m}\sum_{i =1}^{m} (a^{(i)} - y^{(i)}) \end{equation*}$$

Instead of $(A-Y)$, we will use dZ (or $\frac{dL}{dZ}$) since,

$$\begin{equation*} dZ = \left(A- Y\right) \end{equation*}$$

Make sure the gradients are of correct dimensions. Refer to lecture for more information.

Hint: numpy.dot and numpy.sum. Check use of 'keepdims' parameter.

In [93]:

```
def grad_fn(X,dZ):
    '''
    Function to calculate the gradients of weights (dw) and biases (db) w.r.t the objective functi
on L.

    Inputs:
        X: training data of dimensions (d, m)
        dZ: gradient dL/dZ where L is the logistic loss and Z = w^T*X+b is the input to the sigmoi
d activation function
            dZ is of dimensions (1, m)

    outputs:
        dw: gradient dL/dw - gradient of the weight w.r.t. the logistic loss. It is of dimensions
(d,1)
        db: gradient dL/db - gradient of the bias w.r.t. the logistic loss. It is a scalar
    '''
    m = X.shape[1]
    # your code here

    dw = (np.dot(X, np.transpose(dZ)))/m
    db = np.sum(dZ, axis=1, keepdims=True)/m

    return dw,db
```

In [94]:

```
# Contains hidden tests

np.random.seed(1)
d = 2
m1 = 10
X_t = np.random.randn(d,m1)
Y_t = np.random.rand(1,m1)
Y_t[Y_t>0.5] = 1
Y_t[Y_t<=0.5] = 0
```

Training the Model (10 Points)

We will now implement the steps for gradient descent discussed earlier.

In [95]:

```
def model_fit(w,b,X,Y,alpha,n_epochs,log=False):
    '''
    Function to fit a logistic model with the parameters w,b to the training data with labels X an
d Y.

    Inputs:
        w: weight vector of dimensions (d, 1)
        b: scalar bias value
        X: training data of dimensions (d, m)
        Y: training data labels of dimensions (1, m)
        alpha: learning rate
        n_epochs: number of epochs to train the model

    Outputs:
        params: a dictionary to hold parameters w and b
        losses: a list train loss at every epoch
    '''
    losses=[]
```

```
    for epoch in range(n_epochs):

        # Implement the steps in the logistic regression using the functions defined earlier.
        # For each iteration of the for loop
        # Step 1: Calculate output Z = w.T*X + b
        Z = np.dot(np.transpose(w),X) + b
        # Step 2: Apply sigmoid activation: A = sigmoid(Z)
        A = sigmoid(Z)
        # Step 3: Calculate loss = logistic_loss(.) between predicted values A and groundtruth lab
els Y
        loss = logistic_loss(A, Y)
        # Step 4: Estimate gradient dZ = A-Y
        dZ = A-Y
        # Step 5: Estimate gradients dw and db using grad_fn(.).
        dw, db = grad_fn(X, dZ)
        # Step 6: Update parameters w and b using gradients dw, db and learning rate
        #         w = w - alpha * dw
        #         b = b - alpha * db
        w = w - alpha * dw
        b = b - alpha * db

        # your code here

        if epoch%100 == 0:
            losses.append(loss)
            if log == True:
                print("After %i iterations, Loss = %f"%(epoch,loss))
    params ={"w":w,"b":b}

    return params,losses
```

In [96]:

```
# Contains hidden tests

np.random.seed(1)
d = 2
m1 = 10
X_t = np.random.randn(d,m1)
Y_t = np.random.rand(1,m1)
Y_t[Y_t>0.5] = 1
Y_t[Y_t<=0.5] = 0
```

Model Prediction (10 Points)

Once we have the optimal values of model parameters (w,b), we can determine the accuracy of the model on the test data.

\begin{equation*} Z = w^{T}X + b\\ A=\sigma\left(Z\right) \end{equation*}

In [97]:

```
def model_predict(params,X,Y=np.array([]),pred_threshold=0.5):
    '''
    Function to calculate category predictions on given data and returns the accuracy of the predi
ctions.
    Inputs:
        params: a dictionary to hold parameters w and b
        X: training data of dimensions (d, m)
        Y: training data labels of dimensions (1, m). If not provided, the function merely makes p
redictions on X

    outputs:
        Y_Pred: Predicted class labels for X. Has dimensions (1, m)
        acc: accuracy of prediction over X if Y is provided else, 0
        loss: loss of prediction over X if Y is provided else, Inf
    '''
    w = params['w']
    b = params['b']
    m = X.shape[1]

    # Calculate Z using X, w and b
    Z = np.dot(np.transpose(w), X) + b
    # Calculate A using the sigmoid - A is the set of (1,m) probabilities
    A = sigmoid(Z)
```

```
    # Calculate the prediction labels Y_Pred of size (1,m) using A and pred_threshold
    # When A>pred_threshold Y_Pred is 1 else 0
    Y_Pred = np.copy(A)
    isGreaterThanThreshold = Y_Pred>pred_threshold
    Y_Pred[isGreaterThanThreshold]=1
    Y_Pred[~isGreaterThanThreshold]=0
    # your code here

    acc = 0
    loss = float('inf')
    if Y.size!=0:
        loss = logistic_loss(A,Y)
        acc = np.mean(Y_Pred==Y)
    return Y_Pred, acc, loss
```

In [98]:

```
# Contains hidden tests

np.random.seed(1)
d = 2
m1 = 10

# Test standard
X_t = np.random.randn(d,m1)
Y_t = np.random.rand(1,m1)
Y_t[Y_t>0.5] = 1
Y_t[Y_t<=0.5] = 0
```

3. Putting it All Together (10 Points)

We will train our logistic regression model using the data we have loaded and test our predictions on diabetes classification.

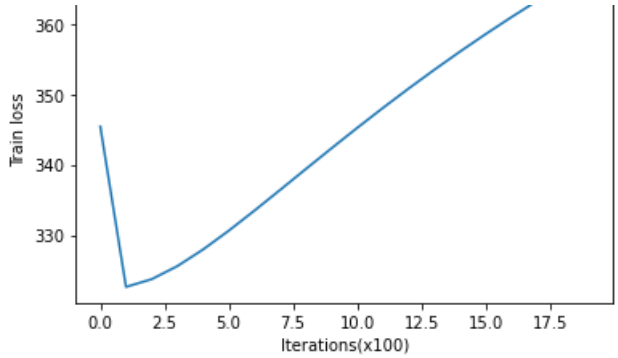
In [99]:

```
#We can use a decently large learning rate becasue the features have been normalized
#When features are not normalized, larger learning rates may cause the learning to oscillate
#and go out of bounds leading to 'nan' errors
#Feel free to adjust the learning rate alpha and the n_epochs to vary the test accuracy
#You should be able to get test accuracy > 70%
#You can go up to 75% to 80% test accuracies as well

alpha = 0.2
n_epochs = 2000

# Write code to initialize parameters w and b with initialize(.) (use train_X to get feature dimen
sions d)
w,b = initialize(train_X.shape[0])
# Use model_fit(.) to estimate the updated 'params' of the logistic regression model and calculate
how the 'losses' varies
# Use variables 'params' and 'losses' to store the outputs of model_fit(.)
params,losses = model_fit(w,b,train_X,train_Y,alpha,n_epochs)
# your code here
```

In [100]:

```
Y_Pred_tr, acc_tr, loss_tr = model_predict(params,train_X,train_Y)
Y_Pred_ts, acc_ts, loss_ts = model_predict(params,test_X,test_Y)
print("Train Accuracy of the model:",acc_tr)
print("Test Accuracy of the model:",acc_ts)
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(losses)
plt.xlabel('Iterations(x100)')
plt.ylabel('Train loss');
```

```
Train Accuracy of the model: 0.778
Test Accuracy of the model: 0.753731343283582
```



In [101]:

```
# Contains hidden tests testing accuracy of test to be greater than 0.7 with the above parameter s
ettings
```

Congratulations on completing this week's assignment - building a single leayer neural network for binary classification. In the following weeks, we will learn to build and train a multilayer neural network for multi category classification.