Fall 2020 CS 4641\7641 A: Machine Learning Homework 3

Instructor: Dr. Mahdi Roozbahani

Deadline: October 28th, Wednesday, AOE

- No unapproved extension of the deadline is allowed. Late submission will lead to 0 credit.
- Discussion is encouraged on Piazza as part of the Q/A. However, all assignments should be done individually.

Instructions for the assignment

- In this assignment, we have programming and writing questions.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You could directly type the Latex equations in the markdown cell.
- Typing with Latex\markdown is required for all the written questions. Handwritten answers would not be accepted.
- If a question requires a picture, you could use this syntax $"< imgsrc =""\ style =" width : 300px;" / >"$ to include them within your ipython notebook.
- Questions marked with **[P]** are programming only and should be submitted to the autograder. Questions marked with **[W]** may required that you code a small function or generate plots, but should **NOT** be submitted to the autograder. It should be submitted on the writing portion of the assignment on gradescope
- The outline of the assignment is as follows:
  - Q1 [30 pts] > Image compression with SVD **[W]** 1.2 and 1.3 | **[P]** items 1.1
  - Q2 [15 pts] > Understanding PCA **[W]** items 2.2 | **[P]** 2.1
  - Q3 [55+(20 bonus for undergrads)]> Regression and regularization **[W]** items 3.2, 3.3, 3.4 and 3.5 | **[P]** items 3.1
  - Q4 [20 pts] > Naive Bayes classification. **[W]** items 4.1 | **[P]** items 4.2
  - Q5 [15 pts] > Noise in PCA and Linear Regression. **[W]** items 5.1, 5.2 and 5.3
  - Q6 [Bonus for all][30 pts] > Manifold learning with Isomap **[W]** items 6.2 | **[P]** items 6.1
  - Q7 [No Points] > Feature Selection. **[P]** items 7

Using the autograder

- You will find two assignments on Gradescope that correspond to HW3: "HW3 - Programming" and "HW3 - Non-programming".
- You will submit your code for the autograder on "HW3 - Programming" in the following format:
  - imgcompression.py
  - pca.py
  - regression.py
  - nb.py
  - isomap.py
- All you will have to do is to copy your implementations of the classes "ImgCompression", "PCA", "Regression", "NaiveBayes", "Isomap" onto the respective files. We provided you different .py files and we added libraries in those files please DO NOT remove those lines and add your code after those lines. Note that these are the only allowed libraries that you can use for the homework.
- You are allowed to make as many submissions until the deadline as you like. Additionally, note that the autograder tests each function separately, therefore it can serve as a useful tool to help you debug your code if you are not sure of what part of your implementation might have an issue.
- **For the "HW3 - Non-programming" part, you will download your jupyter notbook as HTML, print it as a PDF from your browser and submit it on Gradescope. To download the notebook as html, click on "File" on the top left corner of this page and select "Download as > HTML". The non-programming part corresponds to Q1.2 - 1.3, Q2.2, Q3.2 - 3.5, Q4.1, Q5 and Q6.2. For questions that include images include both your response and the generated images in your submission**

In [1]:
```
# HELPER CELL, DO NOT MODIFY
# This is cell which sets up some of the modules you might need
# Please do not change the cell or import any additional packages.

import numpy as np
import json
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.feature_extraction import text
from sklearn.datasets import load_boston, load_diabetes, load_digits, load_bre
ast_cancer, load_iris, load_wine
from sklearn.linear_model import Ridge, LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, accuracy_score
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import floyd_warshall
import warnings

warnings.filterwarnings('ignore')

%matplotlib inline
```

1. Image compression with SVD [30 pts] **[P]** **[W]**

Load images data and plot

In [2]:
```
# HELPER CELL, DO NOT MODIFY
# Load Image
image = plt.imread("hw3_image_2.jpg")/255.
#plot image
fig = plt.figure(figsize=(10,10))
plt.imshow(image)
```

Out[2]: <matplotlib.image.AxesImage at 0x177a369d710>

In [3]:
```
# HELPER CELL, DO NOT MODIFY
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

fig = plt.figure(figsize=(10, 10))
# plot several images
plt.imshow(rgb2gray(image), cmap=plt.cm.bone)
```

Out[3]: <matplotlib.image.AxesImage at 0x177a36fb358>

1.1 Image compression [20pts] **[P]**

SVD is a dimensionality reduction technique that allows us to compress images by throwing away the least important information.

Higher singular values capture greater variance and thus capture greater information from the corresponding singular vector. To perform image compression, apply SVD on each matrix and get rid of the small singular values to compress the image. The loss of information through this process is negligible and the difference between the images can hardly be spotted. For example, the variance captured by the first component

$$\frac{\sigma_1}{\sum_{i=1}^{n} \sigma_i}$$

where σ_i is the i^{th} singular value. You need to finish the following functions to do SVD and then reconstruct the image by components.

Hint 1: http://timbaumann.info/svd-image-compression-demo/ (http://timbaumann.info/svd-image-compression-demo/) is an useful article on image compression.

In [84]:
```
from matplotlib import pyplot as plt
import numpy as np


class ImgCompression(object):
    def __init__(self):
        pass

    def svd(self, X): # [5pts]
        """
        Do SVD. You could use numpy SVD.
        Your function should be able to handle black and white
        images (N*D arrays) as well as color images (N*D*3 arrays)
        In the image compression, we assume that each colum of the image is a
 feature. Image is the matrix X.
        Args:
            X: N * D array corresponding to an image (N * D * 3 if color imag
e)
        Return:
            U: N * N (*3 for color images)
            S: min(N, D) * 1 (* 3 for color images)
            V: D * D (* 3 for color images)
        """
        color = (len(X.shape) == 3)
        if not color:
            X = np.expand_dims(X, 2)
        U = []
        S = []
        V = []
        for i in range(X.shape[2]):
            u, s, v = np.linalg.svd(X[:, :, i])
            U.append(u)
            S.append(s)
            V.append(v)
        U = np.stack(U, 2)
        S = np.stack(S, 1)
        V = np.stack(V, 2)
        if not color:
            U = U[:, :, 0]
            V = V[:, :, 0]
            S = S[:, 0]
        S = S[:min(X.shape[0], X.shape[1])]
        return (U, S, V)



    def rebuild_svd(self, U, S, V, k): # [5pts]
        """
        Rebuild SVD by k componments.
        Args:
            U: N*N (*3 for color images)
            S: min(N, D)*1 (*3 for color images)
            V: D*D (*3 for color images)
            k: int corresponding to number of components
        Return:
            Xrebuild: N*D array of reconstructed image (N*D*3 if color image)
```

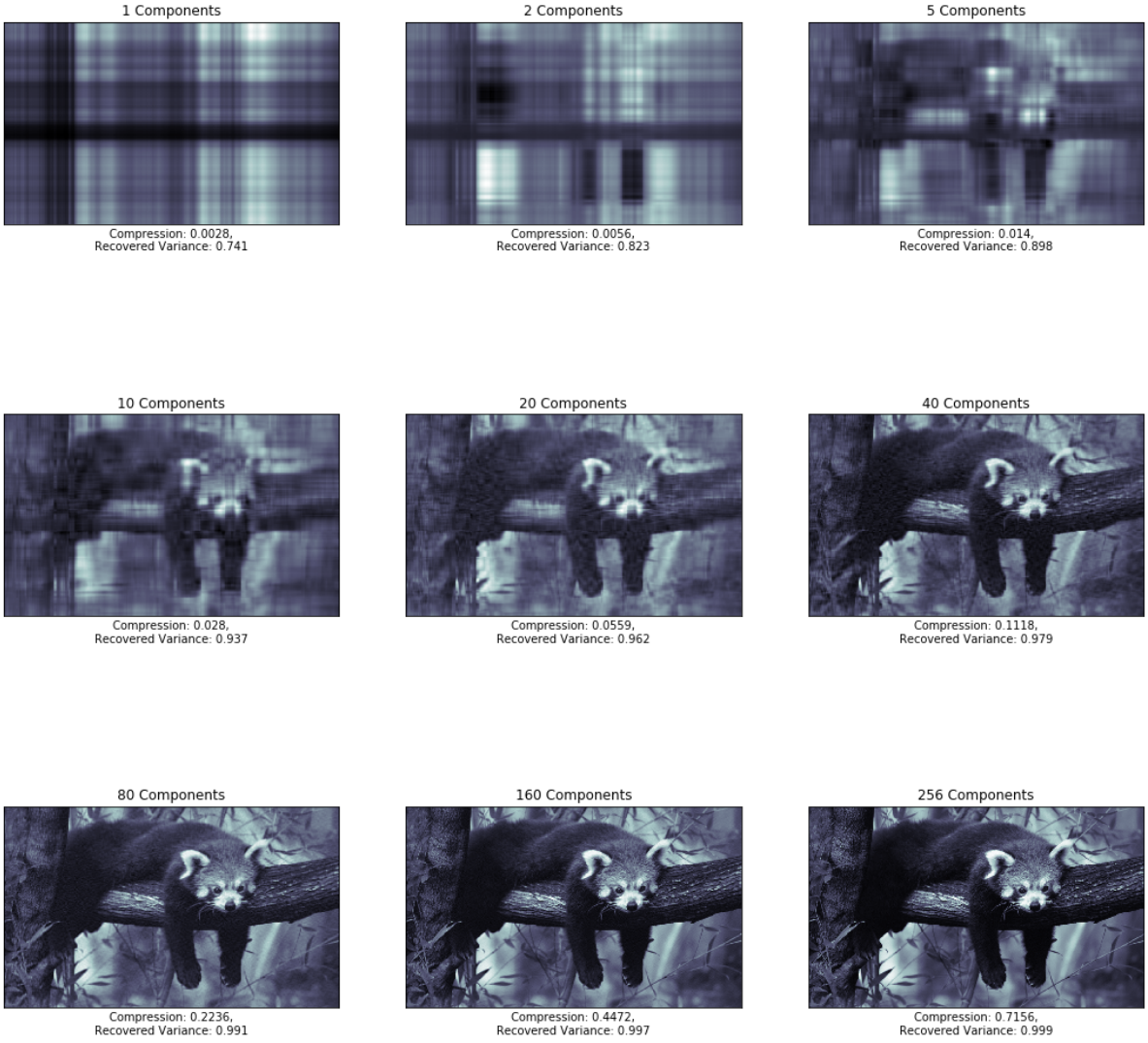
```
        Hint: numpy.matmul may be helpful for reconstructing color images
        """
        color = (len(U.shape) == 3)
        if not color:
            U = np.expand_dims(U, 2)
            S = np.expand_dims(S, 1)
            V = np.expand_dims(V, 2)
        X = []
        for i in range(U.shape[2]):
            u = U[:, :k, i]
            s = S[:k, i]
            v = V[:k, :, i]
            s_mat = np.zeros((k, k))
            for j in range(k):
                s_mat[j, j] = s[j]
            x = u @ s_mat @ v
            X.append(x)
        X = np.stack(X, 2)
        if not color:
            X = X[:, :, 0]
        return X

    def compression_ratio(self, X, k): # [5pts]
        """
        Compute compression of an image: (num stored values in original)/(num
 stored values in compressed)
        Args:
            X: N * D array corresponding to an image (N * D * 3 if color imag
e)
            k: int corresponding to number of components
        Return:
            compression_ratio: float of proportion of storage used by compress
ed image
        """
        return (k * X.shape[0] + k * X.shape[1] + k) / (X.shape[1] * X.shape[0
])

    def recovered_variance_proportion(self, S, k): # [5pts]
        """
        Compute the proportion of the variance in the original matrix recovere
d by a rank-k approximation

        Args:
           S: min(N, D)*1 (*3 for color images) of singular values for the ima
ge
           k: int, rank of approximation
        Return:
           recovered_var: int (array of 3 ints for color image) corresponding
 to proportion of recovered variance
        """
        color = (len(S.shape) == 2)
        if not color:
            S = np.expand_dims(S, 1)
        var = []
        for i in range(S.shape[1]):
            var.append(np.sum(S[:k, i] ** 2) / np.sum(S[:, i] ** 2))
```

```
        if not color:
            return var[0]
        return var
```

1.2 Black and white [5 pts] **[W]**

Use your implementation to generate a set of images compressed to different degrees. Include the images in your non-programming submission to the assignment.

In [5]:
```
# HELPER CELL, DO NOT MODIFY
imcompression = ImgCompression()
bw_image = rgb2gray(image)
U, S, V = imcompression.svd(bw_image)
component_num = [1,2,5,10,20,40,80,160,256]

fig = plt.figure(figsize=(18, 18))

# plot several images
i=0
for k in component_num:
    img_rebuild = imcompression.rebuild_svd(U, S, V, k)
    c = np.around(imcompression.compression_ratio(bw_image, k), 4)
    r = np.around(imcompression.recovered_variance_proportion(S, k), 3)
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(img_rebuild, cmap=plt.cm.bone)
    ax.set_title(f"{k} Components")
    ax.set_xlabel(f"Compression: {c},\nRecovered Variance: {r}")
    i = i+1
```



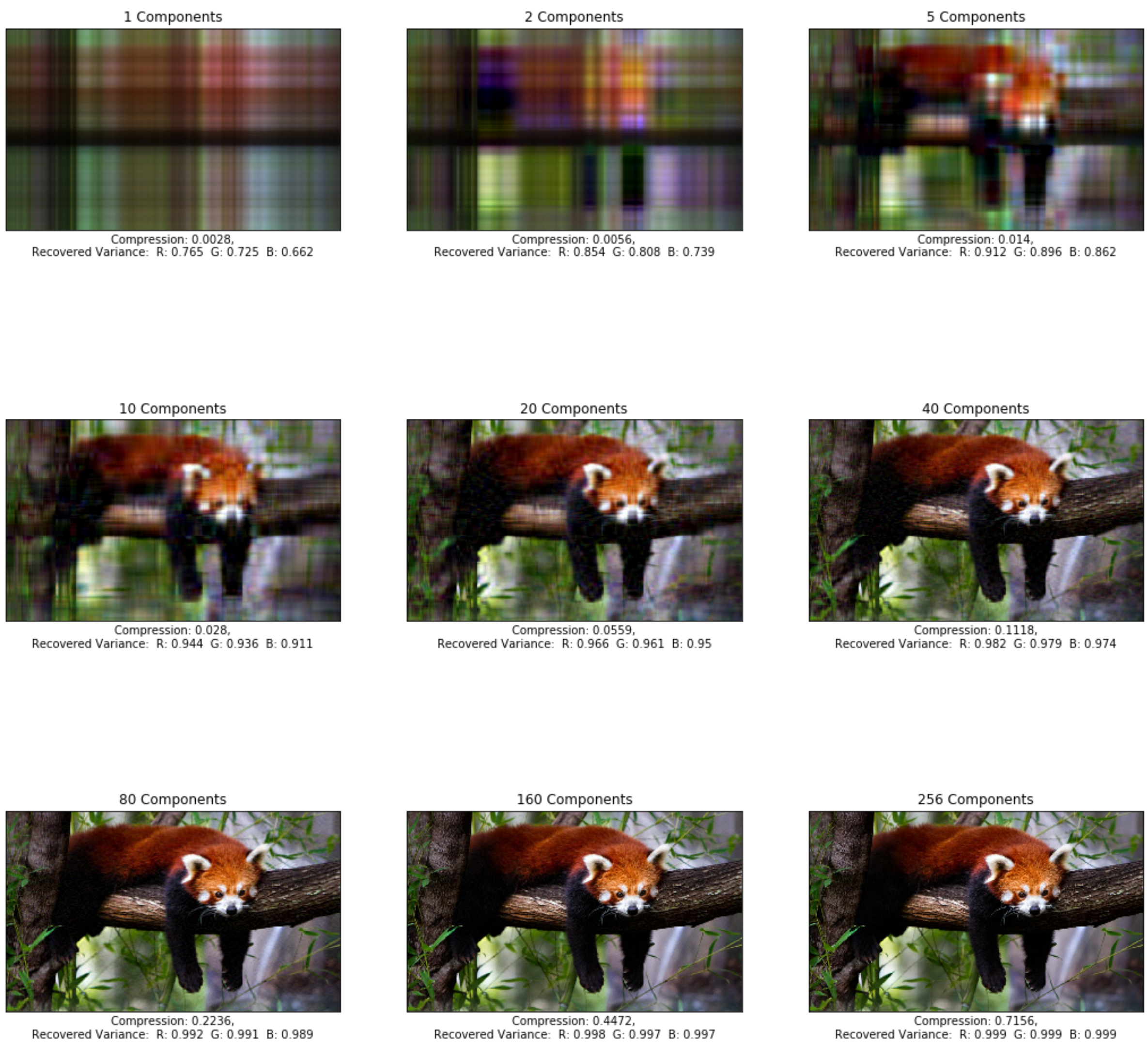
1.3 Color image [5 pts] **[W]**

Use your implementation to generate a set of images compressed to different degrees. Include the images in your non-programming submission to the assignment.

Note: You might get warning "Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)." This warning is acceptable since while rebuilding some of the pixels may go above 1.0. You should see similar image to original even with such clipping.

In [6]:
```
# HELPER CELL, DO NOT MODIFY
imcompression = ImgCompression()
U, S, V = imcompression.svd(image)

# component_num = [1,2,5,10,20,40,80,160,256]
component_num = [1,2,5,10,20,40,80,160,256]

fig = plt.figure(figsize=(18, 18))

# plot several images
i=0
for k in component_num:
    img_rebuild = imcompression.rebuild_svd(U, S, V, k)
    c = np.around(imcompression.compression_ratio(image, k), 4)
    r = np.around(imcompression.recovered_variance_proportion(S, k), 3)
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(img_rebuild)
    ax.set_title(f"{k} Components")
    ax.set_xlabel(f"Compression: {np.around(c,4)},\nRecovered Variance:  R: {r
[0]}  G: {r[1]}  B: {r[2]}")
    i = i+1
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
```



2 Understanding PCA [15 pts] **[P]** | **[W]**

2.1 Implementation [10 pts] **[P]**

Principal Component Analysis (https://en.wikipedia.org/wiki/Principal_component_analysis) (PCA) is another dimensionality reduction technique that reduces dimensions by eliminating small variance eigenvalues and their vectors. With PCA, we center the data first by subtracting the mean. Each singular value tells us how much of the variance of a matrix (e.g. image) is captured in each component. In this problem, we will investigate how PCA can be used to improve features for regression and classification tasks and how the data itself affects the behavior of PCA.

Implement PCA in the below cell.

Assume a dataset is composed of N datapoints, each of which has D features with D < N. The dimension of our data would be D. It is possible, however, that many of these dimensions contain redundant information. Each feature explains part of the variance in our dataset. Some features may explain more variance than others.

In the following cell complete the PCA class by completing functions fit, transform and transform_rv.

In [7]:
```
import numpy as np


class PCA(object):

    def __init__(self):
        self.U = None
        self.S = None
        self.V = None

    def fit(self, X):
        """
        Decompose dataset into principal components.
        You may use your SVD function from the previous part in your implement
ation or numpy.linalg.svd function.

        Don't return anything. You can directly set self.U, self.S and self.V
 declared in __init__ with
        corresponding values from PCA.

        Args:
            X: N*D array corresponding to a dataset
        Return:
            None
        """
        u,s,v = np.linalg.svd(X - np.mean(X, 0), full_matrices = False)
        self.U = u
        self.S = s
        self.V = v
    def transform(self, data, K=2):
        """
        Transform data to reduce the number of features such that final data h
as given number of columns

        Utilize self.U, self.S and self.V that were set in fit() method.

        Args:
            data: N*D array corresponding to a dataset
            K: Int value for number of columns to be kept
        Return:
            X_new: N*K array corresponding to data obtained by applying PCA on
data
        """
        X = data
        return X @ self.V.T[:, :K]

    def transform_rv(self, data, retained_variance=0.99):
        """
        Transform data to reduce the number of features such that a given vari
ance is retained

        Utilize self.U, self.S and self.V that were set in fit() method.

        Args:
            data: N*D array corresponding to a dataset
            retained_variance: Float value for amount of variance to be retain
```

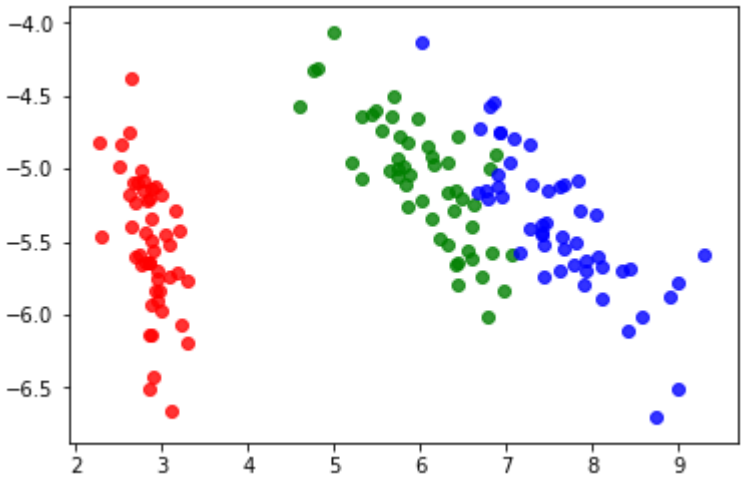
```
ed
        Return:
            X_new: N*K array corresponding to data obtained by applying PCA on
data
        """

        target = np.sum(self.S) * retained_variance
        cum = np.cumsum(self.S)
        K = np.searchsorted(cum, target, side = 'left')
        return self.transform(data, K)

    def get_V(self):
        """ Getter function for value of V """

        return self.V
```

2.2 Visualize [5 pts] **[W]**

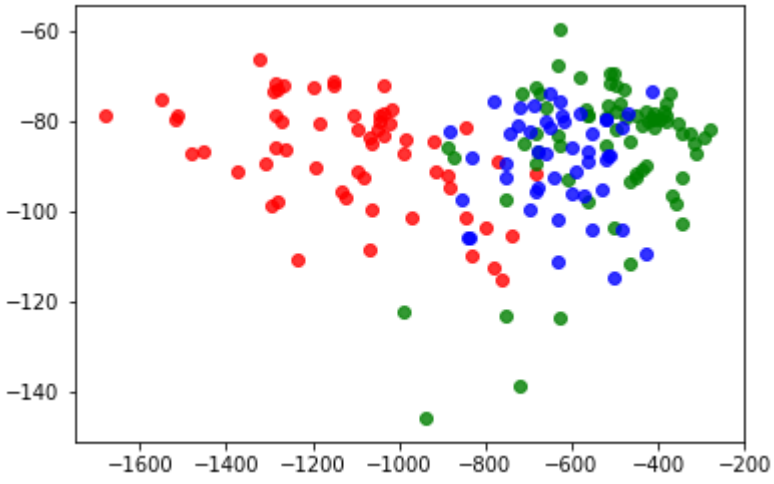
PCA is used to transform multivariate data tables into smaller sets so as to observe the hidden trends and variations in the data. Here you will visualize two datasets (iris and wine) using PCA. Use the above implementation of PCA and reduce the datasets such that they contain only two features. Make 2-D scatter plots of the data points using these features. Make sure to differentiate the data points according to their true labels. The datasets have already been loaded for you. In addition, return the retained variance obtained from the reduced features.

In [8]:
```
import matplotlib.pyplot as plt
def visualize(X,y): # 5 pts
    """
    Args:
        xtrain: NxD numpy array, where N is number of instances and D is the d
imensionality of each instance
        ytrain: numpy array (N,), the true labels

    Return:
        None
    """
    pca = PCA()
    pca.fit(X)
    Xtrans = pca.transform(X, 2)

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    colors = ("red", "green", "blue")
    for i in range(3):
        ax.scatter(Xtrans[y == i, 0], Xtrans[y == i, 1], alpha=0.8, c=colors[i
])

    plt.show()
```

In [9]:
```
# HELPER CELL, DO NOT MODIFY
#Use PCA for visualization of iris and wine data
iris_data = load_iris(return_X_y=True)

X = iris_data[0]
y = iris_data[1]

visualize(X, y)
```



In [10]:
```
# HELPER CELL, DO NOT MODIFY
wine_data = load_wine(return_X_y=True)

X = wine_data[0]
y = wine_data[1]

visualize(X, y)
```



Now you will use PCA on an actual real-world dataset. We will use your implementation of PCA function to reduce the dataset with 99% retained variance and use it to obtain the reduced features. On the reduced dataset, we will use logistic or linear regression and compare results between PCA and non-PCA datasets. Run the following cells to see how PCA works on regression and classification tasks.

In [11]:
```
# HELPER CELL, DO NOT MODIFY
#load the dataset
iris = load_iris()

X = iris.data
y = iris.target

print("data shape before PCA ",X.shape)

pca = PCA()
pca.fit(X)

X_pca = pca.transform_rv(X)

print("data shape with PCA ",X_pca.shape)
```

```
data shape before PCA  (150, 4)
data shape with PCA  (150, 3)
```

In [12]:
```
# HELPER CELL, DO NOT MODIFY
# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3,
                                                    stratify=y,
                                                    random_state=42)

# Use logistic regression to predict classes for test set
clf = LogisticRegression()
clf.fit(X_train, y_train)
preds = clf.predict_proba(X_test)
print('Accuracy: {:.5f}'.format(accuracy_score(y_test,
                                               preds.argmax(axis=1))))
```

```
Accuracy: 0.91111
```

In [14]:
```
# HELPER CELL, DO NOT MODIFY
# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=.3,
                                                    stratify=y,
                                                    random_state=42)

# Use logistic regression to predict classes for test set
clf = LogisticRegression()
clf.fit(X_train, y_train)
preds = clf.predict_proba(X_test)
print('Accuracy: {:.5f}'.format(accuracy_score(y_test,
                                               preds.argmax(axis=1))))
```

```
Accuracy: 0.91111
```

In [15]:
```
# HELPER CELL, DO NOT MODIFY
def apply_regression(X_train, y_train, X_test):
    ridge = Ridge()
    weight = ridge.fit(X_train, y_train)
    y_pred = ridge.predict(X_test)

    return y_pred
```

In [16]:
```
# HELPER CELL, DO NOT MODIFY
#load the dataset
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target

print(X.shape, y.shape)

pca = PCA()
pca.fit(X)

X_pca = pca.transform_rv(X)
print("data shape with PCA ",X_pca.shape)
```

```
(442, 10) (442,)
data shape with PCA  (442, 9)
```

In [17]:
```
# HELPER CELL, DO NOT MODIFY
# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random
_state=42)

#Ridge regression without PCA
y_pred = apply_regression(X_train, y_train, X_test)

# calculate RMSE
rmse_score = np.sqrt(mean_squared_error(y_pred, y_test))
print("rmse score without PCA",rmse_score)
```

```
rmse score without PCA 55.79391924562032
```

In [18]:
```
# HELPER CELL, DO NOT MODIFY
#Ridge regression with PCA
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=.3, ra
ndom_state=42)

#use Ridge Regression for getting predicted labels
y_pred = apply_regression(X_train,y_train,X_test)

#calculate RMSE
rmse_score = np.sqrt(mean_squared_error(y_pred, y_test))
print("rmse score with PCA",rmse_score)
```

```
rmse score with PCA 55.79690542965134
```

For both the tasks above we see an improvement in performance by reducing our dataset with PCA.

Feel free to add other datasets in cell below and play around with what kind of improvement you get with using PCA. There are no points for playing around with other datasets.

In [ ]:
```
######## YOUR CODE BELOW ########

#################################
```

3 Polynomial regression and regularization [55 pts + 20 pts bonus for CS 4641] **[W]** | **[P]**

3.1 Regression and regularization implementations [30 pts + 20 pts bonus for CS 4641] **[P]**

We have three methods to fit linear and ridge regression models: 1) close form; 2) gradient descent (GD); 3) Stochastic gradient descent (SGD). For undergraduate students, you are required to implement the closed form for linear regression and for ridge regression, the others 4 methods are bonus parts. For graduate students, you are required to implement all of them. We use the term weight in the following code. Weights and parameters (θ) have the same meaning here. We used parameters (θ) in the lecture slides.

In [19]:
```
import numpy as np


class Regression(object):

    def __init__(self):
        pass

    def rmse(self, pred, label):  # [5pts]
        '''
        This is the root mean square error.
        Args:
            pred: numpy array of length N * 1, the prediction of labels
            label: numpy array of length N * 1, the ground truth of labels
        Return:
            a float value
        '''
        diff = (pred - label) ** 2
        return np.sqrt(np.mean(diff))

    def construct_polynomial_feats(self, x, degree):  # [5pts]
        """
        Args:
            x: numpy array of length N, the 1-D observations
            degree: the max polynomial degree
        Return:
            feat: numpy array of shape Nx(degree+1), remember to include
            the bias term. feat is in the format of:
            [[1.0, x1, x1^2, x1^3, ....,],
             [1.0, x2, x2^2, x2^3, ....,],
             ......
            ]
        """
        vandermonde = np.zeros((x.shape[0], degree + 1))
        vandermonde[:, 0] = 1
        for i in range(degree):
            vandermonde[:, i + 1] = x * vandermonde[:, i]
        return vandermonde

    def predict(self, xtest, weight):  # [5pts]
        """
        Args:
            xtest: NxD numpy array, where N is number
                   of instances and D is the dimensionality of each
                   instance
            weight: Dx1 numpy array, the weights of linear regression model
        Return:
            prediction: Nx1 numpy array, the predicted labels
        """
        return xtest @ weight

    # =================
    # LINEAR REGRESSION
    # Hints: in the fit function, use close form solution of the linear regres
sion to get weights.
    # For inverse, you can use numpy linear algebra function
```

```
    # For the predict, you need to use linear combination of data points and t
heir weights (y = theta0*1+theta1*X1+...)

    def linear_fit_closed(self, xtrain, ytrain):  # [5pts]
        """
        Args:
            xtrain: N x D numpy array, where N is number of instances and D is
the dimensionality of each instance
            ytrain: N x 1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
        return np.linalg.pinv(xtrain) @ ytrain

    def linear_fit_GD(self, xtrain, ytrain, epochs=5, learning_rate=0.001):  #
[5pts]
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytrain: Nx1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
        weight = np.zeros((xtrain.shape[1], 1)) # D x 1
        for _ in range(epochs):
            weight += learning_rate * xtrain.T @ (ytrain - (xtrain @ weight))
/ xtrain.shape[0]
        return weight

    def linear_fit_SGD(self, xtrain, ytrain, epochs=100, learning_rate=0.001):
# [5pts]
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytrain: Nx1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
        weight = np.zeros((xtrain.shape[1], 1)) # D x 1
        for i in range(epochs):
            weight += np.expand_dims(learning_rate * xtrain[i % xtrain.shape[0
], :] * (ytrain[i % xtrain.shape[0]] - (xtrain[i % xtrain.shape[0], :] @ weigh
t)), 1)
        return weight

    # ================
    # RIDGE REGRESSION

    def ridge_fit_closed(self, xtrain, ytrain, c_lambda):  # [5pts]
        """
        Args:
            xtrain: N x D numpy array, where N is number of instances and D is
the dimensionality of each instance
```

```
            ytrain: N x 1 numpy array, the true labels
            c_lambda: floating number
        Return:
            weight: Dx1 numpy array, the weights of ridge regression model
        """
        return np.linalg.inv(xtrain.T @ xtrain + c_lambda * np.eye(xtrain.shap
e[1])) @ xtrain.T @ ytrain

    def ridge_fit_GD(self, xtrain, ytrain, c_lambda, epochs=500, learning_rate
=1e-7):  # [5pts]
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytrain: Nx1 numpy array, the true labels
            c_lambda: floating number
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
        weight = np.zeros((xtrain.shape[1], 1)) # D x 1
        for _ in range(epochs):
            weight += learning_rate * (xtrain.T @ (ytrain - (xtrain @ weight))
- 2 * c_lambda * weight) / xtrain.shape[0]
        return weight

    def ridge_fit_SGD(self, xtrain, ytrain, c_lambda, epochs=100, learning_rat
e=0.001):  # [5pts]
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytrain: Nx1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
        weight = np.zeros((xtrain.shape[1], 1)) # D x 1
        for i in range(epochs):
            weight += learning_rate *  ((np.expand_dims(xtrain[i % xtrain.shap
e[0], :] * (ytrain[i  % xtrain.shape[0]] - (xtrain[i  % xtrain.shape[0], :] @
weight)), 1)) - (2 * c_lambda * weight / xtrain.shape[0]))
        return weight

    def ridge_cross_validation(self, X, y, kfold=10, c_lambda=100):  # [8 pts]
        """
        Args:
            X : NxD numpy array, where N is the number of instances and D is t
he dimensionality of each instance
            y : Nx1 numpy array, true labels
            kfold: Number of folds you should take while implementing cross va
lidation.
            c_lambda: Value of regularization constant
        Returns:
            meanErrors: Float average rmse error
        Hint: np.concatenate might be helpful.
        Look at 3.5 to see how this function is being used.
```

```
        # For cross validation, use 10-fold method and only use it for your tr
aining data (you already have the train_indices to get training data).
        # For the training data, split them in 10 folds which means that use 1
0 percent of training data for test and 90 percent for training.
        """
        trainx = [[] for k in range(kfold)]
        testx = [[] for k in range(kfold)]
        trainy = [[] for k in range(kfold)]
        testy = [[] for k in range(kfold)]
        for i in range(X.shape[0]):
            for k in range(kfold):
                if i % kfold != k:
                    trainx[k].append(X[i])
                    trainy[k].append(y[i])
                else:
                    testx[k].append(X[i])
                    testy[k].append(y[i])
        trainx = [np.stack(d, 0) for d in trainx]
        testx = [np.stack(d, 0) for d in testx]
        trainy = [np.stack(d, 0) for d in trainy]
        testy = [np.stack(d, 0) for d in testy]
        error = np.zeros((kfold))
        for k in range(kfold):
            weight = self.ridge_fit_closed(trainx[k], trainy[k], c_lambda)
            error[k] = self.rmse(self.predict(testx[k], weight), testy[k])
        return np.mean(error)
```

3.2 About RMSE [3 pts] **[W]**

Do you know whether this RMSE is good or not? If you don't know, we could normalize our labels between 0 and 1. After normalization, what does it mean when RMSE = 1?

Hint: think of the way that you can enforce your RMSE = 1. Note that you can not change the actual labels to make RMSE = 1.

Whether an RMSE of 1 is good or not depends on the standard deviation of the data. If we normalized and the standard deviation of the data is 1 and the RMSE is 1, our model is not really that powerful. This is because the standard deviation represents the minimum RMSE of a model that just guesses the mean data value. If we get an RMSE of 1 on unormalized data with a standard deviation of 6, we can say that our model is quite powerful

3.3 Testing: general functions and linear regression [5 pts] **[W]**

Let's first construct a dataset for polynomial regression.

In this case, we construct the polynomial features up to degree 5.. Each data sample consists of two features $[a, b]$. We compute the polynomial features of both a and b in order to yield the vectors $[1, a, a^2, a^3, \ldots a^{degree}]$ and $[1, b, b^2, b^3, \ldots, b^{degree}]$. We train our model with the cartesian product of these polynomial features. The cartesian product generates a new feature vector consisting of all polynomial combinations of the features with degree less than or equal to the specified degree.

For example, if degree = 2, we will have the polynomial features $[1, a, a^2]$ and $[1, b, b^2]$ for the datapoint $[a, b]$. The cartesian product of these two vectors will be $[1, a, b, ab, a^2, b^2]$. We do not generate a^3 and b^3 since their degree is greater than 2 (specified degree).

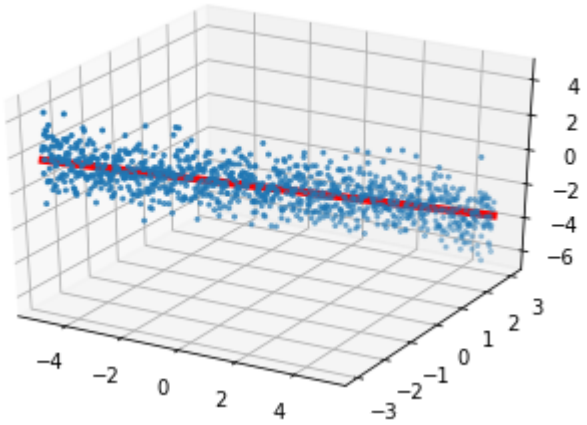
In [20]:
```
#helper, do not need to change
POLY_DEGREE = 5
NUM_OBS = 1000

rng = np.random.RandomState(seed=4)

true_weight = -rng.rand((POLY_DEGREE)**2+2, 1)
true_weight[2:, :] = 0
x_all1 = np.linspace(-5, 5, NUM_OBS)
x_all2 = np.linspace(-3, 3, NUM_OBS)
x_all = np.stack((x_all1,x_all2), axis=1)

reg = Regression()
x_all_feat1 = reg.construct_polynomial_feats(x_all[:,0], POLY_DEGREE)
x_all_feat2 = reg.construct_polynomial_feats(x_all[:,1], POLY_DEGREE)

x_cart_flat = []
for i in range(x_all_feat1.shape[0]):
    x1 = x_all_feat1[i]
    x2 = x_all_feat2[i]
    x1_end = x1[-1]
    x2_end = x2[-1]
    x1 = x1[:-1]
    x2 = x2[:-1]
    x3 = np.asarray([[m*n for m in x1] for n in x2])

    x3_flat = np.reshape(x3, (x3.shape[0]**2))
    x3_flat = list(x3_flat)
    x3_flat.append(x1_end)
    x3_flat.append(x2_end)
    x3_flat = np.asarray(x3_flat)
    x_cart_flat.append(x3_flat)

x_cart_flat = np.asarray(x_cart_flat)
x_all_feat = np.copy(x_cart_flat)

y_all = np.dot(x_cart_flat, true_weight) + rng.randn(x_all_feat.shape[0], 1) #
in the second term, we add noise to data
print(x_all_feat.shape, y_all.shape)

# Note that here we try to produce y_all as our training data
#plot_curve(x_all, y_all) # Data with noise that we are going to predict
#plot_curve(x_all, np.dot(x_cart_flat, true_weight), curve_type='-', color
='r', lw=4) # the groundtruth information

indices = rng.permutation(NUM_OBS)
```

```
(1000, 27) (1000, 1)
```

In [21]:
```
# HELPER CELL, DO NOT MODIFY
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

p = np.reshape(np.dot(x_cart_flat, true_weight), (1000,))
print(x_all[:,0].shape, x_all[:,1].shape,p.shape)
ax.plot(x_all[:,0], x_all[:,1], p, c="red",linewidth=4)
ax.scatter(x_all[:,0], x_all[:,1], y_all,s=4)
```

```
(1000,) (1000,) (1000,)
```

Out[21]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x177a9684320>
```



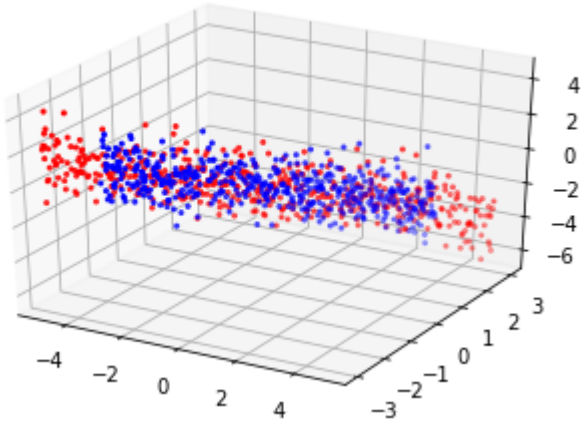
In the figure above, the red curve is the true fuction we want to learn, while the blue dots are the noisy observations. The observations are generated by $Y = X\theta + \sigma$, where σ~N(0,1) are i.i.d. generated noise.

Now let's split the data into two parts, namely the training set and test set. The red dots are for training, while the blue dots are for testing.

In [22]:
```
# HELPER CELL, DO NOT MODIFY
train_indices = indices[:NUM_OBS//2]
test_indices = indices[NUM_OBS//2:]

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

xtrain = x_all[train_indices]
ytrain = y_all[train_indices]
xtest = x_all[test_indices]
ytest = y_all[test_indices]

print(xtrain.shape, xtest.shape, y_all.shape)
ax.scatter(xtrain[:,0], xtrain[:,1], ytrain, c='r',s=4)
ax.scatter(xtest[:,1], xtest[:,1], ytest, c='b',s=4)
```

```
(500, 2) (500, 2) (1000, 1)
```

Out[22]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x177a97732b0>
```



Now let's first train using the entire training set, and see how we performs on the test set and how the learned function look like.

In [23]:
```
# HELPER CELL, DO NOT MODIFY
weight = reg.linear_fit_closed(x_all_feat[train_indices], y_all[train_indices
])
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 0.9589
```

In [24]:
```
# HELPER CELL, DO NOT MODIFY
weight = reg.linear_fit_GD(x_all_feat[train_indices], y_all[train_indices], ep
ochs=500000, learning_rate=1e-9)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 1.2971
```

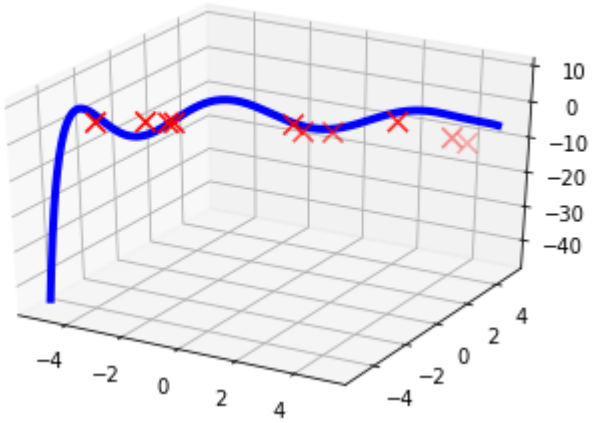
And what if we just use the first 10 observations to train?

In [25]:
```
# HELPER CELL, DO NOT MODIFY
sub_train = train_indices[:10]
weight = reg.linear_fit_closed(x_all_feat[sub_train], y_all[sub_train])
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 5.2039
```

Did you see a worse performance? Let's take a closer look at what we have learned.

In [26]:
```
# HELPER CELL, DO NOT MODIFY
y_pred = reg.predict(x_all_feat, weight)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,0]
y_pred = np.reshape(y_pred, (1000,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
```



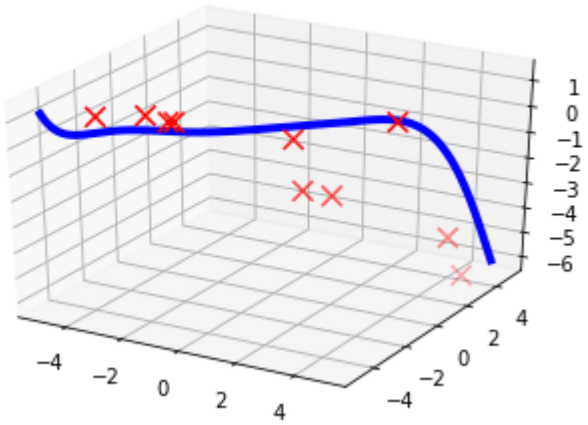
3.4.1 Testing: ridge regression [5 pts] **[W]**

Now let's try ridge regression. Similarly, undergraduate students need to implement the closed form, and graduate students need to implement all the three methods. We will call the prediction function from linear regression part.

Again, let's see what we have learned. You only need to run the cell corresponding to your specific implementation.

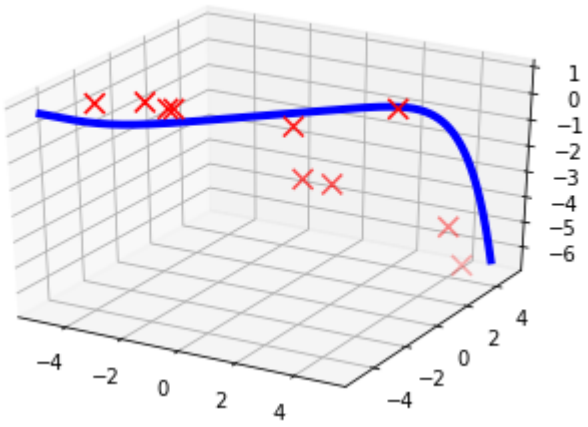
In [27]:
```
# HELPER CELL, DO NOT MODIFY
sub_train = train_indices[:10]
print(x_all_feat[sub_train].shape)
print(y_all[sub_train].shape)
weight = reg.ridge_fit_closed(x_all_feat[sub_train], y_all[sub_train], c_lambd
a=1000)

y_pred = reg.predict(x_all_feat, weight)

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,0]
y_pred = np.reshape(y_pred, (1000,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
```

```
(10, 27)
(10, 1)
test rmse: 1.6014
```



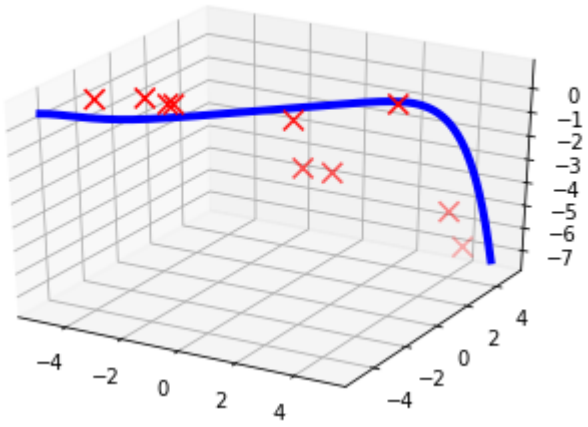
In [28]:
```
# HELPER CELL, DO NOT MODIFY
sub_train = train_indices[:10]
weight = reg.ridge_fit_GD(x_all_feat[sub_train], y_all[sub_train], c_lambda=10
00, learning_rate=1e-9)

y_pred = reg.predict(x_all_feat, weight)

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,0]
y_pred = np.reshape(y_pred, (1000,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
```

```
test rmse: 1.6795
```



In [29]:
```
# HELPER CELL, DO NOT MODIFY
sub_train = train_indices[:10]
weight = reg.ridge_fit_SGD(x_all_feat[sub_train], y_all[sub_train], c_lambda=1
000, learning_rate=1e-9)

y_pred = reg.predict(x_all_feat, weight)

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,0]
y_pred = np.reshape(y_pred, (1000,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
```

```
test rmse: 1.6893
```



3.4.2 Lasso and Ridge Regression [5 pts] **[W]**

We train two linear regression models with different regularizations- one with lasso regularization and the other with ridge regularization. Let w_1 be the final weight vector for the model with lasso regularization and let w_2 be the final weight vector for the model with ridge regularization. How do w_1 and w_2 differ in terms of sparsity? For ridge regression, how do the weights change with change in lambda?

Lasso will have a bunch of zero weights while ridge regularization will have many small nonzero weights. Lasso is thus more sparse. For ridge regression, as lambda approaches zero, the weights slowly decay towards zero but never reach zero.

3.5 Cross validation [7 pts] **[W]**

Let's use Cross Validation to find the best value for c_lambda in ridge regression.

In [30]:
```
# We provided 6 possible values for lambda, and you will use them in cross val
idation.
# For cross validation, use 10-fold method and only use it for your training d
ata (you already have the train_indices to get training data).
# For the training data, split them in 10 folds which means that use 10 percen
t of training data for test and 90 percent for training.
# At the end for each lambda, you have caluclated 10 rmse and get the mean val
ue of that.
# That's it. Pick up the lambda with the lowest mean value of rmse.
# Hint: np.concatenate is your friend.
best_lambda = None
best_error = None
kfold = 10
lambda_list = [0.1, 1, 5, 10, 100, 1000]

for lm in lambda_list:
    err = reg.ridge_cross_validation(x_all_feat[train_indices], y_all[train_in
dices], kfold, lm)
    print('lambda: %.2f' % lm, 'error: %.6f'% err)
    if best_error is None or err < best_error:
        best_error = err
        best_lambda = lm

print('best_lambda: %.2f' % best_lambda)
weight = reg.ridge_fit_closed(x_all_feat[train_indices], y_all[train_indices],
c_lambda=10)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
lambda: 0.10 error: 1.020500
lambda: 1.00 error: 1.020452
lambda: 5.00 error: 1.020513
lambda: 10.00 error: 1.021073
lambda: 100.00 error: 1.045348
lambda: 1000.00 error: 1.128672
best_lambda: 1.00
test rmse: 0.9588
```

4. Naive Bayes Classification [20pts]

In Bayesian classification, we're interested in finding the probability of a label given some observed feature vector $x = [x_1, .., x_d]$, which we can write as $P(y \mid x_1, .., x_d)$. Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(y \mid x_1, .., x_d) = \frac{P(x_1, .., x_d \mid y)P(y)}{P(x_1, .., x_d)}$$

The main assumption in Naive Bayes is that, given the label, the observed features are conditionally independent i.e.

$$P(x_1, .., x_d \mid y) = P(x_1 \mid y) \times .. \times P(x_d \mid y)$$

Therefore, we can rewrite Bayes rule as

$$P(y \mid x_1, .., x_d) = \frac{P(x_1 \mid y) \times .. \times P(x_d \mid y)P(y)}{P(x_1, .., x_d)}$$

Training Naive Bayes

One way to train a Naive Bayes classifier is done using frequentist approach to calculate probability, which is simply going over the training data and calculating the frequency of different observations in the training set given different labels. For example,

$$P(x_1 = i \mid y = j) = \frac{P(x_1 = i, y = j)}{P(y = j)} = \frac{\text{Number of times in training data } x_1 = i \text{ and } y = j}{\text{Total number of times in training data } y = j}$$

Testing Naive Bayes

During the testing phase, we try to estimate the probability of a label given an observed feature vector. We combine the probabilities computed from training data to estimate the probability of a given label. For example, if we are trying to decide between two labels y_1 and y_2, then we compute the ratio of the posterior probabilities for each label:

$$\frac{P(y_1 \mid x_1, .., x_d)}{P(y_2 \mid x_1, .., x_d)} = \frac{P(x_1, .., x_d \mid y_1)}{P(x_1, .., x_d \mid y_2)} \frac{P(y_1)}{P(y_2)} = \frac{P(x_1 \mid y_1) \times .. \times P(x_d \mid y_1)P(y_1)}{P(x_1 \mid y_2) \times .. \times P(x_d \mid y_2)P(y_2)}$$

All we need now is to compute $P(x_1|y_i), .., P(x_d \mid y_i)$ and $P(y_i)$ for each label by pluging in the numbers we got during training. The label with the higher posterior probabilities is the one that is selected.

4.1 Bayes in Advertisements [5pts] **[W]**

An advertising agency want to analyze the advertisements for a product. They want to target people from all age groups. They show 5 advertisement videos to 200 people. The following is the data on how many people from each group liked which videos.

Age Group	Total	Video 1	Video 2	Video 3	Video 4	Video 5
16 - 35	100	20	30	60	15	90
36 - 55	60	15	50	30	20	40
Above 55	40	35	30	10	10	5
Total	200	70	110	100	45	135

A new consumer is shown the videos and he likes videos 2, 3 and 5. Which age group is he most likely to belong to?

Note: You can assume that each person provides opinion about each video independently i.e. Person 1 liking Video 1 has no effect on their assesment of Video 2.

First we test if Person 1 is 16-35, we have $P(y_1|x_2,x_3,x_5) = P(x_2|y_1)P(x_3|y_1)P(x_5|y_1)P(y_1) = 0.3 * 0.6 * 0.9 * 0.5 = 0.081$. Second we test if Person 1 is 36-55, we have $P(y_2|x_2,x_3,x_5) = P(x_2|y_2)P(x_3|y_2)P(x_5|y_2)P(y_2) = (5/6)(0.5)(2/3)(60/200) = 0.0833$. Third we test if Person 1 is above 55, we have $P(y_3|x_2,x_3,x_5) = P(x_2|y_3)P(x_3|y_3)P(x_5|y_3)P(y_3) = 0.75 * 0.25 * 0.125 * 0.2 = 0.0046875$. Therefore, Person 1 is most likely 36 - 55.

4.2 The Federalist Papers [15pts] **[P]**

The Federalist Papers (https://en.wikipedia.org/wiki/The_Federalist_Papers) were a series of essays written in 1787–1788 meant to persuade the citizens of the State of New York to ratify the Constitution and which were published anonymously under the pseudonym “Publius”. In later years the authors were revealed as Alexander Hamilton, John Jay, and James Madison. However, there is some disagreement as to who wrote which essays. Hamilton wrote a list of which essays he had authored only days before being killed in a duel with then Vice President Aaron Burr. Madison wrote his own list many years later, which is in conflict with Hamilton’s list on 12 of the essays. Since by this point the two (who were once close friends) had become bitter rivals, historians have long been unsure as to the reliability of both lists. We will try to settle this dispute using a simple Naive Bayes classifier.

The code which is provided loads the documents and builds a “bag of words” representation (https://en.wikipedia.org/wiki/Bag-of-words_model) of each document. Your task is to complete the missing portions of the code and to determine your best guess as to who wrote each of the 12 disputed essays. (Hint: H and M are the labels that stand for Hamilton and Madison, while the label D stands for disputed for the papers we are trying to label in our data. Our job here is to define whether D essays belong to H or M using Naive Bayes. Note that the label D for disputed, is completely unrelated to the feature dimension D which is an integer).

_priors_ratio function calculates the ratio of class probabilities of document belonging to Hamilton as compared to Madison. We do this based on word counts rather than document counts.

_likelihood_ratio function calculates the ratio of word probablities given the author it belonged to.

Note 1: In _likelihood_ratio() add one to each word count so as to avoid issues with zero word count. This is known as Add-1 smoothing. It is a type of additive smoothing.

In [31]:
```
import numpy as np
import json
from sklearn.feature_extraction import text


class NaiveBayes(object):

    def __init__(self):
        # load Documents
        x = open('fedpapers_split.txt').read()
        papers = json.loads(x)

        # split Documents
        papersH = papers[0]  # papers by Hamilton
        papersM = papers[1]  # papers by Madison
        papersD = papers[2]  # disputed papers

        # Number of Documents for H, M and D
        nH = len(papersH)
        nM = len(papersM)
        nD = len(papersD)

        '''To ignore certain common words in English that might skew your mode
l, we add them to the stop words
        list below. You may want to experiment by choosing your own list of st
op words, but be sure to keep
        'HAMILTON' and 'MADISON' in this list at a minimum, as their names app
ear in the text of the papers
        and leaving them in could lead to unpredictable results '''

        stop_words = text.ENGLISH_STOP_WORDS.union({'HAMILTON', 'MADISON'})
        # stop_words = {'HAMILTON','MADISON'}
        # Form bag of words model using words used at least 10 times
        vectorizer = text.CountVectorizer(stop_words=stop_words, min_df=10)
        X = vectorizer.fit_transform(papersH + papersM + papersD).toarray()

        '''To visualize the full list of words remaining after filtering out s
top words and words used less
        than min_df times uncomment the following line'''
        # print(vectorizer.vocabulary_)

        # split word counts into separate matrices
        self.XH, self.XM, self.XD = X[:nH, :], X[nH:nH + nM, :], X[nH + nM:,
:]

    def _likelihood_ratio(self, XH, XM):  # [5pts]
        '''
        Args:
            XH: nH x D where nH is the number of documents that we have for Ha
milton,
                while D is the number of features (we use the word count as th
e feature)
            XM: nM x D where nM is the number of documents that we have for Ma
dison,
                while D is the number of features (we use the word count as th
e feature)
```

```
        Return:
            fratio: 1 x D vector of the likelihood ratio of different words (H
amilton/Madison)
        '''
        pH = (np.sum(XH, 0) + 1) / (np.sum(XH) + XH.shape[1])
        pM = (np.sum(XM, 0) + 1) / (np.sum(XM) + XM.shape[1])
        return np.expand_dims(pH / pM, 0)

    def _priors_ratio(self, XH, XM):  # [5pts]
        '''
        Args:
            XH: nH x D where nH is the number of documents that we have for Ha
milton,
                while D is the number of features (we use the word count as th
e feature)
            XM: nM x D where nM is the number of documents that we have for Ma
dison,
                while D is the number of features (we use the word count as th
e feature)
        Return:
            pr: prior ratio of (Hamilton/Madison)
        '''
        return np.sum(XH) / np.sum(XM)

    def classify_disputed(self, fratio, pratio, XD):  # [5pts]
        '''
        Args:
            fratio: 1 x D vector of ratio of likelihoods of different words
            pratio: 1 x 1 number
            XD: 12 x D bag of words representation of the 12 disputed document
s (D = 1307 which are the number of features for each document)
        Return:
             1 x 12 list, each entry is H to indicate Hamilton or M to indicat
e Madison for the corrsponding document
        '''
        p = np.ones((12)) * pratio
        for i in range(fratio.shape[1]):
            for j in range(XD.shape[0]):
                if XD[j, i] != 0:
                    p[j] = np.nan_to_num((p[j] * (fratio[0, i] ** XD[j, i])))
        return ['H' if p[i] >= 1 else 'M' for i in range(12)]
```

In [32]:
```
# HELPER CELL, DO NOT MODIFY
NB = NaiveBayes()
fratio = NB._likelihood_ratio(NB.XH, NB.XM)
pratio = NB._priors_ratio(NB.XH, NB.XM)
resolved = NB.classify_disputed(fratio, pratio, NB.XD)

print(resolved)
```

```
['M', 'M', 'M', 'M', 'H', 'H', 'M', 'H', 'H', 'M', 'M', 'M']
```

5 Noise in PCA and Linear Regression (15 Pts) **[W]**

Both PCA and least squares regression can be viewed as algorithms for inferring (linear) relationships among data variables. In this part of the assignment, you will develop some intuition for the differences between these two approaches, and an understanding of the settings that are better suited to using PCA or better suited to using the least squares fit.

The high level bit is that PCA is useful when there is a set of latent (hidden/underlying) variables, and all the coordinates of your data are linear combinations (plus noise) of those variables. The least squares fit is useful when you have direct access to the independent variables, so any noisy coordinates are linear combinations (plus noise) of known variables.

5.1 Slope Functions (5 Pts) **[W]**

In the following cell:

1. For this function assume that X is the first feature and Y is the second feature for the data. Write a function pca_slope that takes a vector X of xi's and a vector Y of yi's and returns the slope of the first component of the PCA.
2. Write a function linear_regression_slope that takes X and y and returns the slope of the least squares fit. (Hint: since X is one dimensional, this takes a particularly simple form1 $((X - \overline{X}) \cdot (Y - \overline{Y}))/\|X - \overline{X}\|_2^2$, where X is the mean value of X.)

In later subparts, we consider the case where our data consists of noisy measurements of x and y. For each part, we will evaluate the quality of the relationship recovered by PCA, and that recovered by standard least squares regression.

As a reminder, least squares regression minimizes the squared error of the dependent variable from its prediction. Namely, given (x_i, y_i) pairs, least squares returns the line $l(x)$ that minimizes $\sum_i (y_i - l(x_i))^2$.

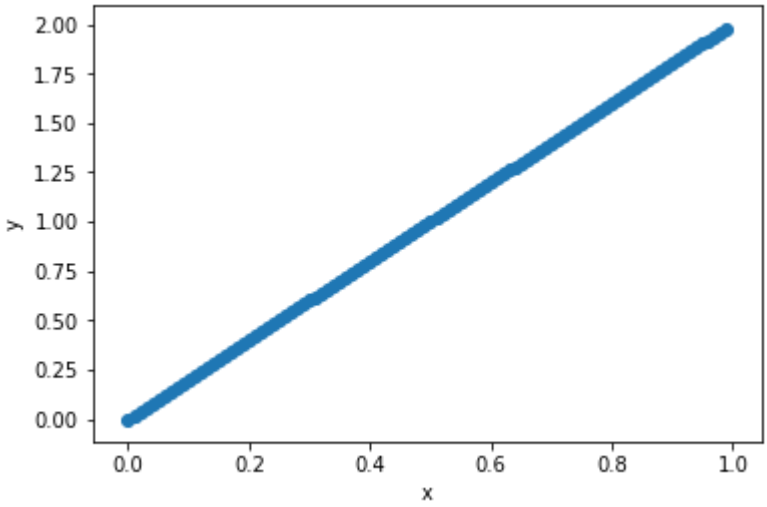
Note 1: You should use the PCA and Linear Regression implementations from Q2 and Q3 in this question. Do not use any kind of regularization for Linear Regression.

In [33]:
```
def pca_slope(x, y):
    """
    Calculates the slope of the first principal component given by PCA

    Args:
        x: (N,) vector of feature x
        y: (N,) vector of feature y
    Return:
        slope: slope of the first principal component
    """

    data = np.stack([x, y], 1)
    pca = PCA()
    pca.fit(data)
    v = pca.get_V()[0, :]
    return v[1] / v[0]


def lr_slope(X, y):
    """
    Calculates the slope of the best fit as given by Linear Regression

    For this function don't use any regularization

    Args:
        X: N*1 array corresponding to a dataset
        y: N*1 array of labels y
    Return:
        slope: slope of the best fit
    """
    reg = Regression()
    w = reg.linear_fit_closed(X, y)
    return w
```

We will consider a simple example with two variables, x and y, where the true relationship between the variables is y = 2x. Our goal is to recover this relationship—namely, recover the coefficient “2”. We set X = [0, .01, .02, .03, . . . , 1] and y = 2x. Make sure both functions return 2.

In [34]:
```
# HELPER CELL, DO NOT MODIFY
x = np.arange(0, 1, 0.01)
y = 2 * np.arange(0, 1, 0.01)

print("Slope of first principal component", pca_slope(x, y))

print("Slope of best linear fit", lr_slope(x[:, None], y))

plt.scatter(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

```
Slope of first principal component 2.0
Slope of best linear fit [2.]
```

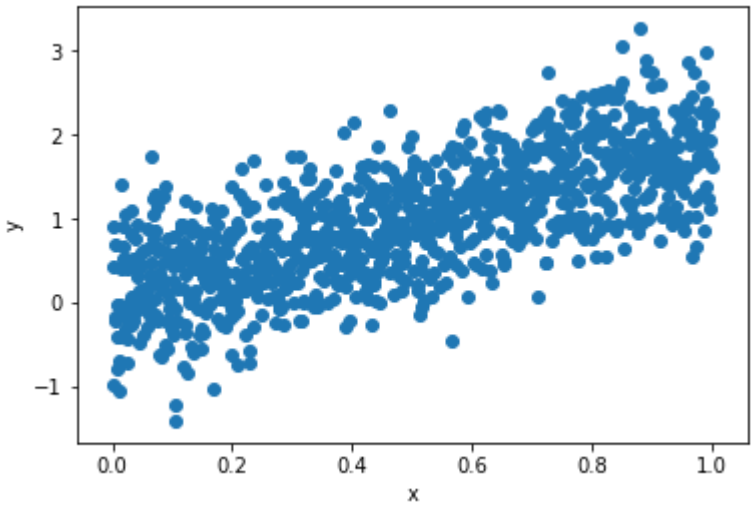


5.2 Analysis Setup (5 Pts) **[W]**

Error in y

In this subpart, we consider the setting where our data consists of the actual values of x, and noisy estimates of y. Run the following cell to see how the data looks when there is error in y.

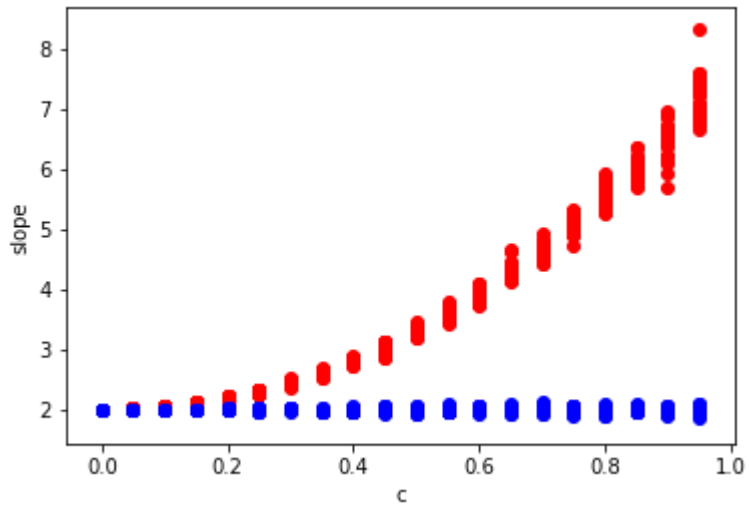
In [36]:
```
# HELPER CELL, DO NOT MODIFY
base = np.arange(0.001, 1, 0.001)
c = 0.5
X = base
y = 2 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

plt.scatter(X, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



In the subsequent cell implement the following:

1. Fix $X = [x_1, x_2, \ldots, x_{1000}] = [.001, .002, .003, \ldots, 1]$.
2. For a given noise level c, set $\hat{y}_i \sim 2x_i + \mathcal{N}(0, c) = 2i/1000 + \mathcal{N}(0, c)$, and $\hat{Y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{1000}]$
3. Make a scatter plot with c on the horizontal axis, and the output of pca-slope and ls-slope on the vertical axis.
4. For each c in $[0, 0.05, 0.1, \ldots, .95, 1.0]$, take a sample $\hat{Y}$, plot the output of pca-recover as a red dot, and the output of ls-recover as a blue dot. Repeat 30 times. You should end up with a plot of 1260 dots, in 21 columns of 60, half red and half blue.

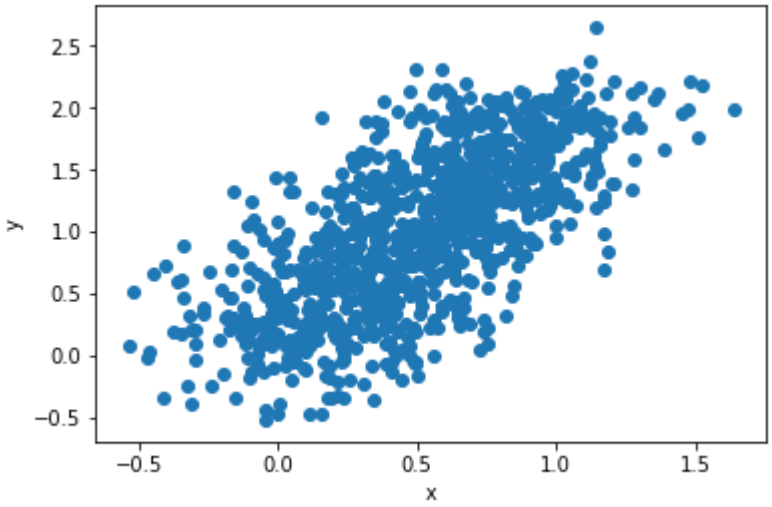
In [37]:
```
pca_slope_values = []
linreg_slope_values = []
c_values = []

for i in range(30):
    for c in np.arange(0, 1, 0.05):
        ####### YOUR CODE BELOW #######
        X = base
        y = 2 * base + np.random.normal(loc=[0], scale=c, size=base.shape)
        pca_slope_values.append(pca_slope(X, y))
        linreg_slope_values.append(lr_slope(X.reshape(999, 1), y))
        ###############################

        c_values.append(c)

plt.scatter(c_values, pca_slope_values, c='r')
plt.scatter(c_values, linreg_slope_values, c='b')
plt.xlabel("c")
plt.ylabel("slope")
plt.show()
```



Error in x and y

We now examine the case where our data consists of noisy estimates of both x and y. Run the following cell to see how the data looks when there is error in both.

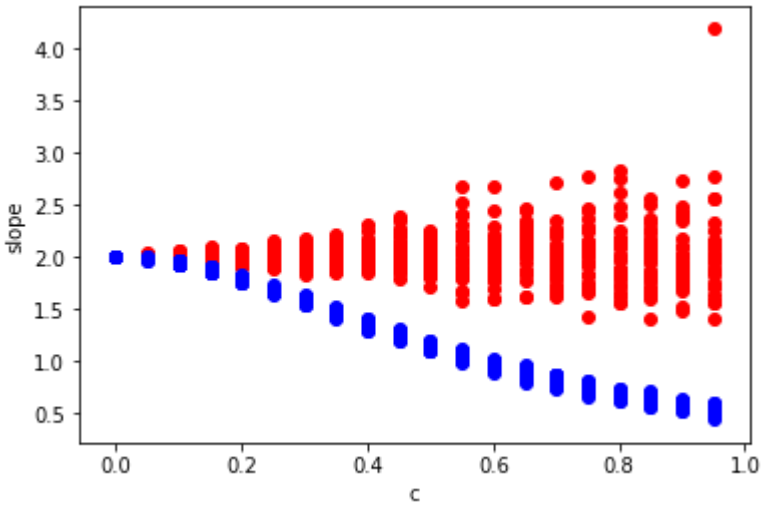
In [38]:
```
# HELPER CELL, DO NOT MODIFY
base = np.arange(0.001, 1, 0.001)
c = 0.5
X = base + np.random.normal(loc=[0], scale=c, size=base.shape) * 0.5
y = 2 * base + np.random.normal(loc=[0], scale=c, size=base.shape) * 0.5

plt.scatter(X, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



In the subsequent cell implement the following:

1. For a given noise level c, let $\hat{x}_i \sim x_i + \mathcal{N}(0, c) = i/1000 + \mathcal{N}(0, c)$, and $\hat{X} = [\hat{x}_1, \hat{x}_2, \ldots \hat{x}_{1000}]$
2. For the same noise level c, set $\hat{y}_i \sim 2x_i + \mathcal{N}(0, c) = 2i/1000 + \mathcal{N}(0, c)$, and $\hat{Y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_1000]$
3. Make a scatter plot with c on the horizontal axis, and the output of pca-slope and ls-slope on the vertical axis. For each c in $[0, 0.05, 0.1, \ldots, .95, 1.0]$, take a sample of both $\hat{X}$ and $\hat{Y}$, plot the output of pca-recover as a red dot, and the output of ls-recover as a blue dot. Repeat 30 times. You should end up with a plot of 1260 dots, in 21 columns of 60, half red and half blue.

In [39]:
```
pca_slope_values = []
linreg_slope_values = []
c_values = []

for i in range(30):
    for c in np.arange(0, 1, 0.05):
        ####### YOUR CODE BELOW #######
        X = base + np.random.normal(loc=[0], scale=c, size=base.shape)
        y = 2 * base + np.random.normal(loc=[0], scale=c, size=base.shape)
        pca_slope_values.append(pca_slope(X, y))
        linreg_slope_values.append(lr_slope(X.reshape(999, 1), y))
        ###############################

        c_values.append(c)

plt.scatter(c_values, pca_slope_values, c='r')
plt.scatter(c_values, linreg_slope_values, c='b')
plt.xlabel("c")
plt.ylabel("slope")
plt.show()
```



5.3. Analysis (5 Pts) **[W]**

Based on your observations from previous subsections answer the following questions about the two cases (error in X and error in both X and Y) in 2-3 lines.

Note:

1. The closer the value of slope to actual slope ("2" here) the better the algorithm is performing.
2. You don't need to provide a proof for this question.

Questions:

1. Which case does PCA perform worse in? Why does PCA perform worse in this case?
2. Why does PCA perform better in the other case?
3. Which case does Linear Regression perform well? Why does Linear Regression perform well in this case?

1. PCA performs worse in the case where there is only error in y. This is because PCA is minimizing the two dimensional distance from the point (x, y) to the line i.e. the principal component. PCA expects both variables to have some amount of error. When only one variable has error, PCA guesses that both variables are partially wrong. Because only one variable was faulty, PCA makes a large error.
2. PCA performs better when the error is in both x and y. In this case, PCA guesses that both dimensions are off and is able to compensate well.
3. Linear regression performs better when there is only error in y. Linear regression assumes that all of the x values are correct and the y values may be off. We optimize the squared one dimensional distance in y. Because linear regression implicitly guesses that all of x values are correct it is more accurate in this case.

6 Manifold learning [Bonus for everyone][30 pts] **[W]**|**[P]**

While PCA is wonderful tool for dimensionality reduction it does not work very well when dealing with non-linear relationships between features. Manifold learning is a class of algorithms that can be used to reduce dimensions in complex high-dimensional datasets. While a number of methods have been proposed to perform this type of operation, we will focus on Isomap. Isomap has been shown to be sensitive to data noise amongst other issues, however it has been shown to perform reasonably well for real world data. The algorithm consists of two main steps: first computing a manifold distance matrix, followed by classical mutidimensional scaling. You will be creating your implementation of Isomap. In order to do so, you must read the original paper "A Global Geometric Framework for Nonlinear Dimensionality Reduction" (http://web.mit.edu/cocosci/Papers/sci_reprint.pdf) by Tenenbaum et al. (2000), which outlines the method. You are also encouraged to read this general survey of manifold learning (https://cseweb.ucsd.edu/~lcayton/resexam.pdf) by Cayton (2005), where the original algorithm is further explained in a more detailed yet simplified fashion.

6.1 Implementation [23 pts] **[P]**

6.1.1 pairwise distance [3pts] **[P]**

In this section, you are asked to implement pairwise_dist function.

Given $X \in \mathbb{R}^{NxD}$ and $Y \in \mathbb{R}^{MxD}$, obtain the pairwise distance matrix $dist \in \mathbb{R}^{NxM}$ using the euclidean distance metric, where $dist_{i,j} = ||X_i - Y_j||_2$.

DO NOT USE FOR LOOP in your implementation -- they are slow and will make your code too slow to pass our grader. Use array broadcasting instead.

6.1.2 manifold distance matrix [10pts] **[P]**

In this section, you need to implement manifold_distance_matrix function.

Given $X \in \mathbb{R}^{NxD}$ and the number of the clusters K, compute the distance matrix $dist \in \mathbb{R}^{NxM}$, where the values obey the following equations:

$$dist_{ij} = \begin{cases} ||X_i - Y_j||_2, & j \in Neighbour(i), \\ Shortest\ Path\ Distance, & j \notin Neighbour(i). \end{cases}$$

Hint: For doing this part, you can partly utilize the scipy toolbox. After creating your k-nearest weighted neighbors adjacency matrix, you can convert it to a sparse graph object csr_matrix (https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html) and utilize the pre-built Floyd-Warshall algorithm (https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.floyd_warshall.html) to compute the manifold distance matrix.

6.1.3 multidimensional scaling [10pts] **[P]**

In this section, you need to accomplish the multimensional_scaling part.

Given the computed distance matrix $dist_{ij}$ and the size of the new reduced feature space d, you need to return the X embedding of the new feature space.

Closed Form of the Gram Matrix with Centering:

We can now succinctly state the closed matrix form of B by making use of the centering matrix:

$$\boxed{B = \frac{-1}{2} C_n D^2 C_n}$$

Note: C_n is the centering matrix with $C_n = I_n - \frac{1}{n}\mathbf{1}\mathbf{1}^\mathbf{T}$ and from the original distance matrix we have D^2 = matrix with entries d_{ij}^2

Find eigenvalues and eigenvectors of matrix B Since the gram matrix B is a real symmetric, positive definite matrix, we know that it will have real eigenvalues and we can use the following eigendecomposition of B in order to find an expression for our output configuration:

$$\begin{aligned} B &= V \Lambda V^T \\ &= (\Lambda^{\frac{1}{2}} V^T)^T (\Lambda^{\frac{1}{2}} V^T) \\ &= X X^T \qquad\qquad (\text{from original def. of B}) \end{aligned}$$

and therefore we have

$$X = V\sqrt{\Lambda}$$

where the eigenvalues are given by diagonal matrix $\mathbf{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_n)$ and the eigenvectors are given by the following matrix with the columns set as the eigenvectors $V = (v_1, \ldots, v_n)^T$

Find coordinates of output configuration We can now define a k-dimensional configuration by choosing the largest k eigenvalues and the corresponding eigenvectors from k columns of V:

$$\boxed{X_k = V_k \sqrt{\Lambda_k}}$$

where Λ_k is the k x k diagonal submatrix of Λ and V_k is the n x k submatrix of V.

In the cell below implement the code for section 5.1

In [227]:
```
import numpy as np
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import floyd_warshall
from numpy import linalg as LA


class Isomap(object):
    def __init__(self):  # No need to implement
        pass

    def pairwise_dist(self, x, y):  # [3 pts]
        """
        Args:
            x: N x D numpy array
            y: M x D numpy array
        Return:
                dist: N x M array, where dist2[i, j] is the euclidean distance
between
                x[i, :] and y[j, :]
                """


        x2 = np.sum(x**2, 1)
        y2 = np.sum(y**2, 1)
        xy = x @ y.T
        d2 = -2 * xy + y2 + x2[:, np.newaxis]
        d2[d2 < 0] = 0
        return np.sqrt(d2)

    def manifold_distance_matrix(self, x, K):  # [10 pts]
        """
        Args:
            x: N x D numpy array
        Return:
            dist_matrix: N x N numpy array, where dist_matrix[i, j] is the euc
lidean distance between points if j is in the neighborhood N(i)
            or comp_adj = shortest path distance if j is not in the neighborho
od N(i).
        Hint: After creating your k-nearest weighted neighbors adjacency matri
x, you can convert it to a sparse graph
        object csr_matrix (https://docs.scipy.org/doc/scipy/reference/generate
d/scipy.sparse.csr_matrix.html) and utilize
        the pre-built Floyd-Warshall algorithm (https://docs.scipy.org/doc/sci
py/reference/generated/scipy.sparse.csgraph.floyd_warshall.html)
        to compute the manifold distance matrix.
        """
        N, D = x.shape
        d2 = self.pairwise_dist(x, x)
        a = np.tile(np.arange(N).reshape(N, 1), (1, N - K - 1)).flatten()
        b = np.argsort(d2,axis =  1)[:,K+1:].flatten()
        d2[a, b] = 0
        sparse = csr_matrix(d2)
        dist_matrix = floyd_warshall(csgraph=sparse, directed=False, return_pr
edecessors=False)
        return dist_matrix

    def multidimensional_scaling(self, dist_matrix, d):  # [10 pts]
```

```
        """
        Args:
            dist_matrix: N x N numpy array, the manifold distance matrix
            d: integer, size of the new reduced feature space
        Return:
            S: N x d numpy array, X embedding into new feature space.
        """
        N, _ = dist_matrix.shape
        C = np.identity(N) - (1 / N)
        B = -0.5 * C @ (dist_matrix ** 2) @ C
        w, v = np.linalg.eigh(B)
        print(w.shape, v.shape)
        idx = np.argsort(w)[::-1] [:d]
        w = w[idx]
        print(w)
        v = v[:, idx]
        return v * np.sqrt(w)

    # you do not need to change this
    def __call__(self, data, K, d):
        dist_matrix = self.manifold_distance_matrix(data, K)
        return self.multidimensional_scaling(dist_matrix, d)
```

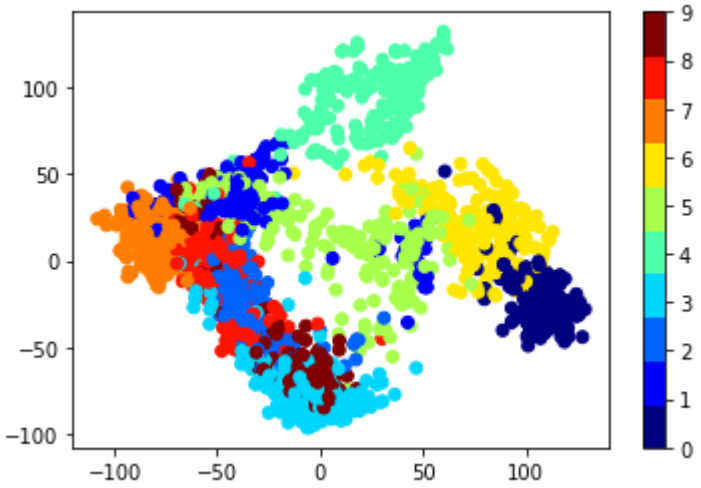
6.2 Examples for different datasets [7pts] **[W]**

Apply your implementation of Isomap for some of the datasets (e.g. MNIST and Iris). Discuss how the results compare to PCA.

In [229]:
```
# HELPER CELL, DO NOT MODIFY
# example MNIST data
mnist = load_digits()
proj = Isomap()(mnist.data, 10, 2)
print(proj)
plt.scatter(proj[:, 0], proj[:, 1], c=mnist.target, cmap=plt.cm.get_cmap('jet'
, 10))
plt.colorbar(ticks=range(10))
```

```
(1797,) (1797, 1797)
[5934446.71462086 4389750.46620551]
[[ 99.35526636 -30.67607893]
 [-28.46288812  47.14388111]
 [-33.87534434   2.91040941]
 ...
 [-41.64788515  -0.83178905]
 [-31.00155885 -53.0313556 ]
 [-21.10003475 -28.37218827]]
```

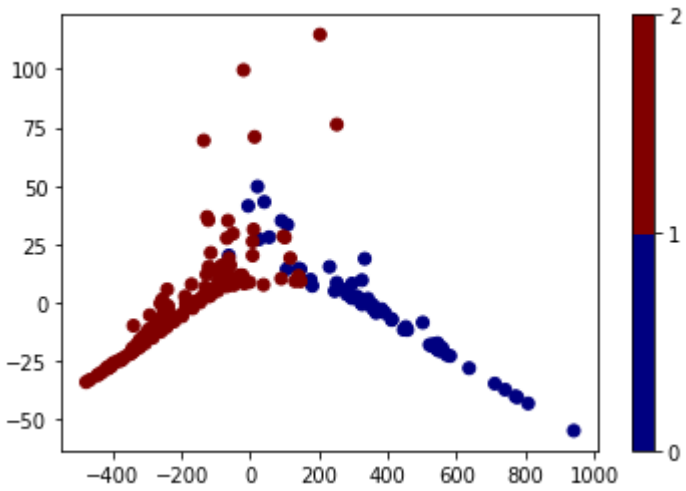
Out[229]: <matplotlib.colorbar.Colorbar at 0x177ae25a0b8>



In [236]:
```
# HELPER CELL, DO NOT MODIFY
# example MNIST data
wine = load_wine()
proj = Isomap()(wine.data, 10, 2)
plt.scatter(proj[:, 0], proj[:, 1], c=wine.target, cmap=plt.cm.get_cmap('jet',
np.max(wine.target)))
plt.colorbar(ticks=range(10))
```

```
(178,) (178, 178)
[18359778.37458121   100733.52876616]
```

Out[236]: <matplotlib.colorbar.Colorbar at 0x177b28f63c8>



Isomap gets much better separation of nonlinear clusters than pca. This is especially evident on nonlinear data list mnist.

7 Feature Selection Implementation [No Points] **[P]**

Note: This is a fun question for you to learn about Feature Reduction. No points will be awarded for it. If you have time please go over it. It would be beneficial for your project.

Implement a method to find the final list of significant features due to forward selection and backward elimination.

Forward Selection:

In forward selection, we start with a null model, start fitting the model with one individual feature at a time, and select the feature with the minimum p-value. We continue to do this until we have a set of features where one feature's p-value is less than the confidence level.

Steps to implement it:

- 1: Choose a significance level (given to you).
- 2: Fit all possible simple regression models by considering one feature at a time.
- 3: Select the feature with the lowest p-value.
- 4: Fit all possible models with one extra feature added to the previously selected feature(s).
- 5: Select the feature with the minimum p-value again. if p_value < significance, go to Step 4. Otherwise, terminate.

Backward Elimination:

In backward elimination, we start with a full model, and then remove the insignificant feature with the highest p-value (that is greater than the significance level). We continue to do this until we have a final set of significant features.

Steps to implement it:

- 1: Choose a significance level (given to you).
- 2: Fit a full model including all the features.
- 3: Select the feature with the highest p-value. If (p_value > significance level), go to Step 4, otherwise terminate.
- 4: Remove the feature under consideration.
- 5: Fit a model without this feature. Repeat entire process from Step 3 onwards.

TIP 1: The p-value is known as the observed significance value for a test hypothesis. It tests all the assumptions about how the data was generated in the model, not just the target hypothesis it was supposed to test. Some more information about p-values can be found here: https://towardsdatascience.com/what-is-a-p-value-b9e6c207247f (https://towardsdatascience.com/what-is-a-p-value-b9e6c207247f)

TIP 2: For this function, you will have to install statsmodels if not installed already. Run 'pip install statsmodels' in command line/terminal. statsmodels is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration. You will have to use this library to choose a regression model to fit your data against. Some more information about this module can be found here: https://www.statsmodels.org/stable/index.html (https://www.statsmodels.org/stable/index.html)

In [ ]:
```
import pandas as pd
import statsmodels.api as sm

class FeatureReduction(object):
    def __init__(self):
        pass

    @staticmethod
    def forward_selection(data, target, significance_level=0.05):
        '''
        Args:
            data: data frame that contains the feature matrix
            target: target feature to search to generate significant features
            significance_level: the probability of the event occuring by chanc
e
        Return:
            forward_list: list containing significant features
        '''

        raise NotImplementedError

    @staticmethod
    def backward_elimination(data, target, significance_level = 0.05):
        '''
        Args:
            data: data frame that contains the feature matrix
            target: target feature to search to generate significant features
            significance_level: the probability of the event occuring by chanc
e
        Return:
            backward_list: list containing significant features
        '''

        raise NotImplementedError
```

In [ ]:
```
# HELPER CELL, DO NOT MODIFY
boston = load_boston()
bos = pd.DataFrame(boston.data, columns = boston.feature_names)
bos['Price'] = boston.target
X = bos.drop("Price", 1)       # feature matrix
y = bos['Price']               # target feature
featurereduction = FeatureReduction()
#Run the functions to make sure two lists are generated, one for each method
print("Features selected by forward selection:", featurereduction.forward_sele
ction(X, y))
print("Features selected by backward selection:", featurereduction.backward_el
imination(X, y))
```