Machine Learning & Data Mining
Caltech CS/CNS/EE 155
Homework 1

Changhao Xu
UID: 2103530
January 13th, 2020

---

1 Basics

Question A : Hypothesis set is a set of hypothesis/functions that approximates an unknown target function.

Question B: Hypothesis set of linear model is in the form of f(x|w, b) = $w^T x + b$.

Question C: Overfitting occurs when the model function fits too closely to a limited set of data points, which results in a large out-of-sample error while in-sample error is small.

Question D: Validation & Regularization.

Question E: Training data is the data set that is used for training a model. Testing data is the data set that is used for evaluating the trained model's performance. We should never change the model based on test data because that will make test data become training data, and cannot indicate out-of-sample performance any more.

Question F: i.i.d., i.e. Data point is independent and identically distributed.

Question G: Input space X can be a 'bag of words' feature vector extracted from the email contents, and Y can be the result whether this email is spam or not (let +1 when the email is spam and -1 when the email is not spam).

Question H : K-fold cross-validation procedure: 1) Split data into k equal partitions, 2) Train on (k-1) partitions and test one 1 partition, 3) Repeat the process for k times. This procedure allows re-using training data as test data and using all data as validation.

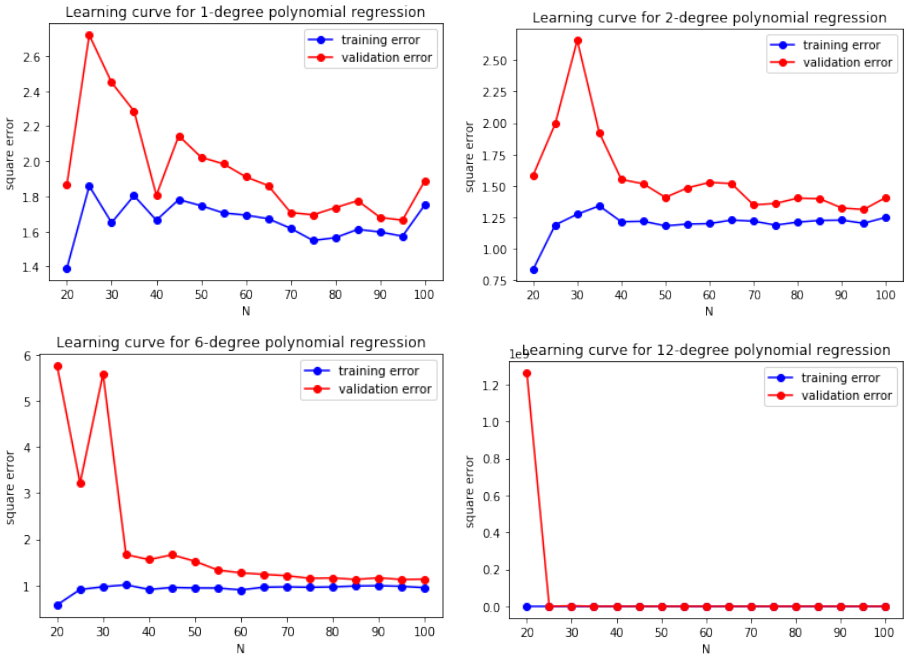
2 Bias-Variance Tradeoff

Question A:

$$\begin{aligned}
\mathbb{E}_s\left[E_{out}(f_s)\right] &= \mathbb{E}_s\left[\mathbb{E}_x\left[(f_s(x)-y(x))^2\right]\right] \\
&= \mathbb{E}_x\left[\mathbb{E}_s\left[(f_s(x)-y(x))^2\right]\right] \\
&= \mathbb{E}_x\left[\mathbb{E}_s\left[(f_s(x)-F(x)+F(x)-y(x))^2\right]\right] \\
&= \mathbb{E}_x\left[\mathbb{E}_s\left[(f_s(x)-F(x))^2\right]\right] + \mathbb{E}_x\left[\mathbb{E}_s\left[(F(x)-y(x))^2\right]\right] + 2\mathbb{E}_x\left[\mathbb{E}_s\left[(f_s(x)-F(x))(F(x)-y(x))\right]\right] \\
&= \mathbb{E}_x\left[Var(x)\right] + \mathbb{E}_x\left[Bias(x)\right] + 2\mathbb{E}_x\left[(F(x)-y(x))\mathbb{E}_s\left[(f_s(x)-F(x))\right]\right] \\
&= \mathbb{E}_x\left[Var(x)\right] + \mathbb{E}_x\left[Bias(x)\right] + 0 \\
&= \mathbb{E}_x\left[Bias(x)+Var(x)\right]
\end{aligned}$$

Question B:

Changhao Xu
UID: 2103530



Question C: 1st-degree polynomial has the highest bias, since it has the highest square error for both training and validation on average, which is a sign of underfitting.

Question D: 12th-degree polynomial has the highest variance, since it has the biggest validation error and small training error initially when data set is small, which is a sign of overfitting.

Question E: The model will not improve much with additional training points due to limited model complexity. As shown in the learning curve, the slope of both training error and validation error becomes small after reaching certain number of data points.

Question F: Training error is the error on the data set that has been used by the model. Since the model has used these data to minimize error of loss function, while validation data has not been used by the model, training error is generally lower than validation error.

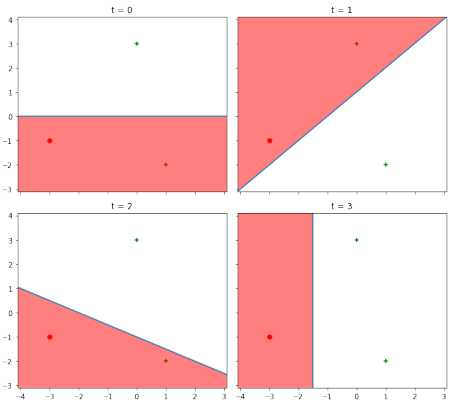
Question G: 6th-degree polynomial is expected to perform best on unseen data, since it has the lowest validation error and optimal bias-variance tradeoff, which is close to out-of-sample performance.

3 The Perceptron

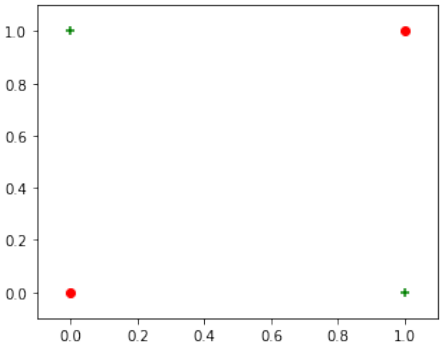
Question A: Code output as follows:

Changhao Xu
UID: 2103530

```
t = 0, b = 0, w = [0. 1.], [x1, x2, Y] = [array([ 1, -2]), 1]
t = 1, b = 1, w = [ 1. -1.], [x1, x2, Y] = [array([0, 3]), 1]
t = 2, b = 2, w = [1. 2.], [x1, x2, Y] = [array([ 1, -2]), 1]
t = 3, b = 3, w = [2. 0.], [x1, x2, Y] = []

final w = [2. 0.], final b = 3.0
```



Question B: In a 2D data set, 4 points are linearly inseparable:



In a 3D data set, 5 points are linearly inseparable: suppose any 3 points form a plane, and let these 3 points be +1; let one point at one side of the plane be -1, and one point at the other side of the plane be -1. The resulting feature is linear inseparable.
In a N-dimensional data set, (N+2) points will be linearly inseparable.

Question C: When data set is not linearly seperable, PLA will keep finding a hyperplane that classifies all data points correctly, which doesn't exist, so PLA will not converge.

4 Stochastic Gradient Descent

Question A: We can simply let w_0 =b, and x_0 = 1, so that **w** = (w_0, w_1, w_2, ..., w_d), **x** = (1, x_1, x_2, ..., x_d).

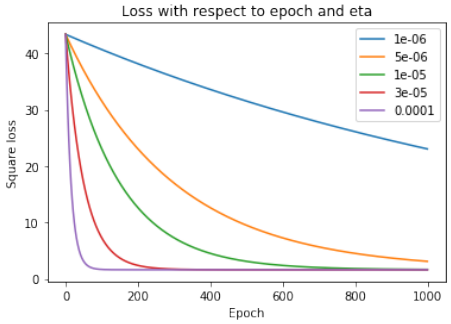
Question B:

$$\partial_w \sum_{i=1}^{N} (y_i - \mathbf{w}^T x_i)^2 = \sum_{i=1}^{N} -2x_i(y_i - \mathbf{w}^T x_i)$$

Question C: Please see Jupyter Notebook.

Question D: SGD converges at the same minimum point regardless of varying starting point, though the converging rate varies between different starting points. Different datasets have different converging rate, but they all converge towards a minimum point.

Question E: As η increases, SGD converges faster, but when η becomes very big, SGD cannot converge

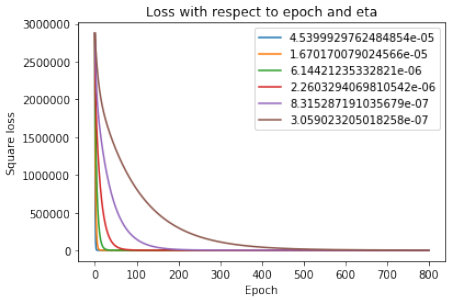


any more.

Changhao Xu
UID: 2103530

Question F: Final weights **w** = [-0.22717707 -5.94209998 3.94391318 -11.72382875 8.78568403]

Question G: As η increases, SGD converges faster. (When η becomes very big, SGD cannot converge



any more. But here η overall is small.)

Question H: Final weights **w** = [-0.31644251 -5.99157048 4.01509955 -11.93325972 8.99061096]. The result is not the same, but quite similar with SGD.

Question I: SGD may not give the exact same answer as closed form solution, but it is very efficient to get an approximate solution, especially when dataset is large in reality.

Question J: Stopping condition: store the losses after each epoch. When the relative loss decrease becomes very small (e.g. $<$0.0001), stop the epoch.

Question K: SGD is smoothier compared with PLA. SGD uses gradient descent to guide weight modification, while perceptron simply goes from one point to another and may change weight abruptly.

2_notebook

January 15, 2020

1 Problem 2

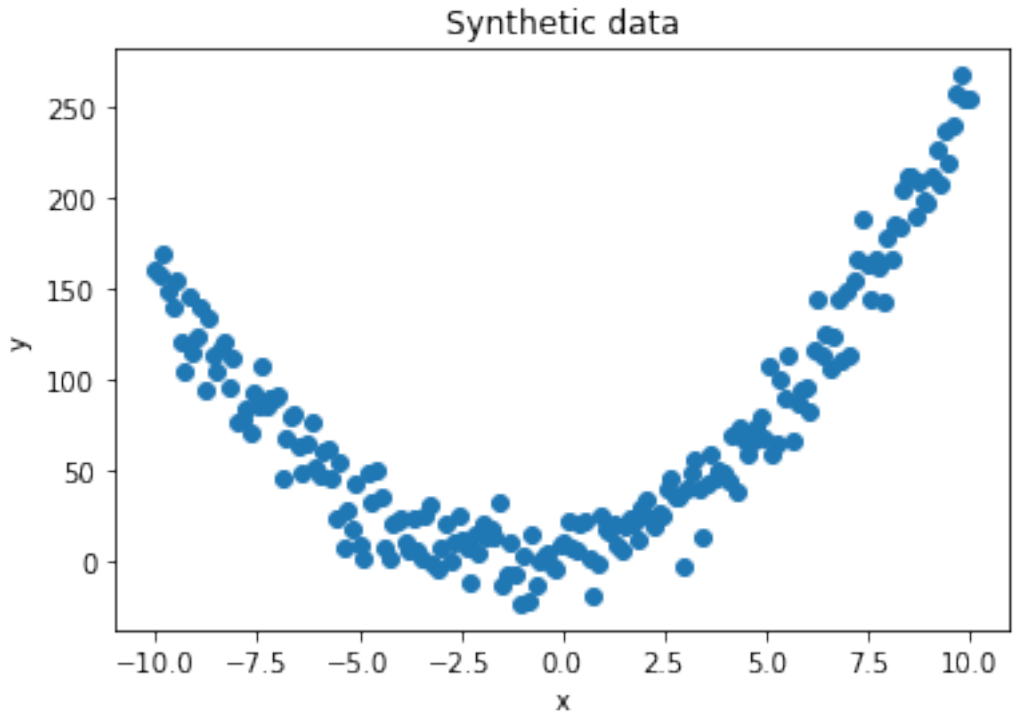
[1]:
```
# Setup:

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold

import warnings
warnings.filterwarnings("ignore")
```

1.1 Example code using the polyfit and kfold functions

Note: This section is not part of the homework problem, but provides some potentially-helpful example code regarding the usage of numpy.polyfit, numpy.polyval, and sklearn.model_selection.KFold. First, let's generate some synthetic data: a quadratic function plus some Gaussian noise.

[2]:
```
# Coefficients of the quadratic function, y(x) = ax^2 + bx + c:
a = 2
b = 5
c = 7

N = 200        # Number of data points
x = np.linspace(-10, 10, num = N)                # x ranges from -10 to 10
# y is the quadratic function of x specified by a, b, and c, plus noise
y = a*x**2 + b*x + c + 15* np.random.randn(N)

# Plot the data:
plt.figure()
plt.plot(x, y, marker = 'o', linewidth = 0)

plt.xlabel('x')
plt.ylabel('y')
plt.title('Synthetic data')
```

```
plt.show()
```

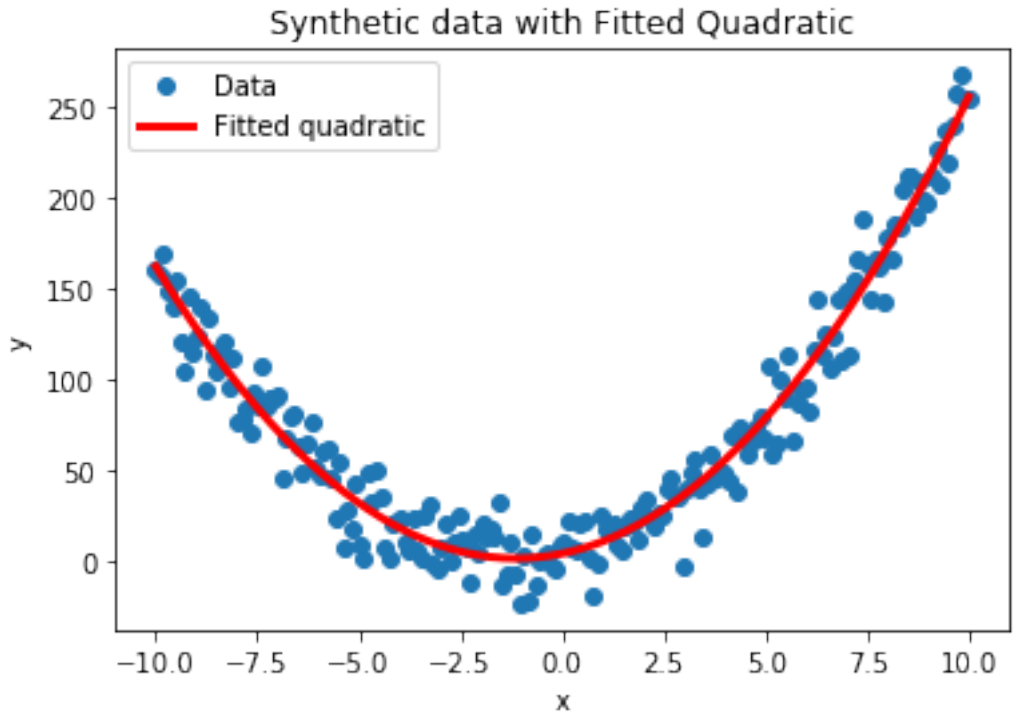


Next, we'll use the numpy.polyfit function to fit a quadratic polynomial to this data. We can evaluate the resulting polynomial at arbitrary points.

[3]:
```
# Fit a degree-2 polynomial to the data:
degree = 2
coefficients = np.polyfit(x, y, degree)

# Print out the resulting quadratic function:
print('We fit the following quadratic function: f(x) = %f*x^2 + %f*x + %f' %  \
      (coefficients[0], coefficients[1], coefficients[2]))

# Evaluate the fitted polynomial at x = 4:
x_test = 4
f_eval = np.polyval(coefficients, x_test)
print('\nf(%i) = %f' % (x_test, f_eval))

# Let's visualize our fitted quadratic:
plt.figure()

plt.plot(x, y, marker = 'o', linewidth = 0)
plt.plot(x, np.polyval(coefficients, x), color = 'red', linewidth = 3)

plt.legend(['Data', 'Fitted quadratic'], loc = 'best')
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.title('Synthetic data with Fitted Quadratic')

plt.show()
```

```
We fit the following quadratic function: f(x) = 2.044687*x^2 + 4.657820*x +
4.671919

f(4) = 56.018187
```



Finally, assume that we'd like to perform 10-fold cross validation with this dataset. Let's divide it into training and test sets, and print out the test sets. To limit the amount of text that we are printing out, we'll modify the dataset to make it smaller.

[4]:
```
# Coefficients of the quadratic function, y = ax^2 + bx + c:
a = 2
b = 5
c = 7

N = 80        # Number of points--fewer this time!
x = np.linspace(-10, 10, num = N)                # x ranges from -10 to 10
# y is the quadratic function of x specified by a, b, and c, plus noise
y = a*x**2 + b*x + c + 15* np.random.randn(N)
```

```python
# Initialize kfold cross-validation object with 10 folds:
num_folds = 10
kf = KFold(n_splits=num_folds)

# Iterate through cross-validation folds:
i = 1
for train_index, test_index in kf.split(x):

    # Print out train indices:
    print('Fold ', i, ' of ', num_folds, ' train indices:', train_index)
    # Print out test indices:
    print('Fold ', i, ' of ', num_folds, ' test indices:', test_index)

    # Training and testing data points for this fold:
    x_train, x_test = x[train_index], x[test_index]
    y_train, y_test = y[train_index], y[test_index]

    i += 1
```

```
Fold  1  of  10  train indices: [ 8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  1  of  10  test indices: [0 1 2 3 4 5 6 7]
Fold  2  of  10  train indices: [ 0  1  2  3  4  5  6  7 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  2  of  10  test indices: [ 8  9 10 11 12 13 14 15]
Fold  3  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
24 25 26 27 28 29 30 31
 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  3  of  10  test indices: [16 17 18 19 20 21 22 23]
Fold  4  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  4  of  10  test indices: [24 25 26 27 28 29 30 31]
Fold  5  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  5  of  10  test indices: [32 33 34 35 36 37 38 39]
Fold  6  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 48 49 50 51 52 53 54 55
```

```
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  6  of  10  test indices: [40 41 42 43 44 45 46 47]
Fold  7  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  7  of  10  test indices: [48 49 50 51 52 53 54 55]
Fold  8  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
Fold  8  of  10  test indices: [56 57 58 59 60 61 62 63]
Fold  9  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 72 73 74 75 76 77 78 79]
Fold  9  of  10  test indices: [64 65 66 67 68 69 70 71]
Fold  10  of  10  train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71]
Fold  10  of  10  test indices: [72 73 74 75 76 77 78 79]
```

1.2 Loading the Data for Problem 2

This code loads the data from bv_data.csv using the load_data helper function. Note that data[:, 0] is an array of all the x values in the data and data[:, 1] is an array of the corresponding y values.

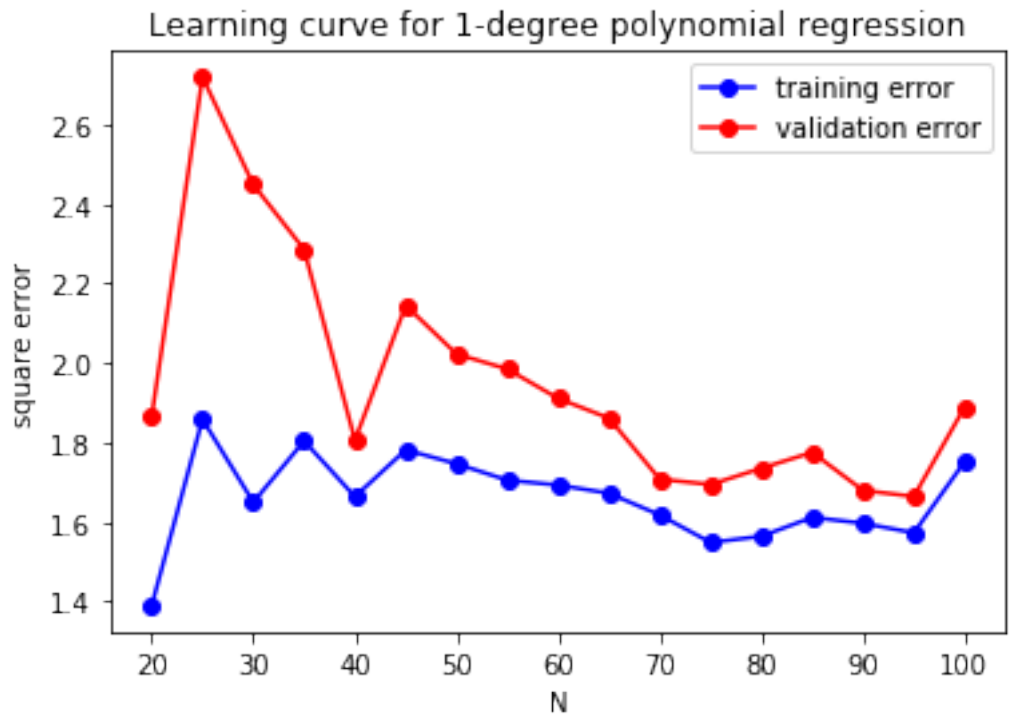
[15]:
```
def load_data(filename):
    """
    Function loads data stored in the file filename and returns it as a numpy␣
 ↪ndarray.
    Input:
        filename: given as a string.
    Output:
        Data contained in the file, returned as a numpy ndarray
    """
    return np.loadtxt(filename, skiprows=1, delimiter=',')
```

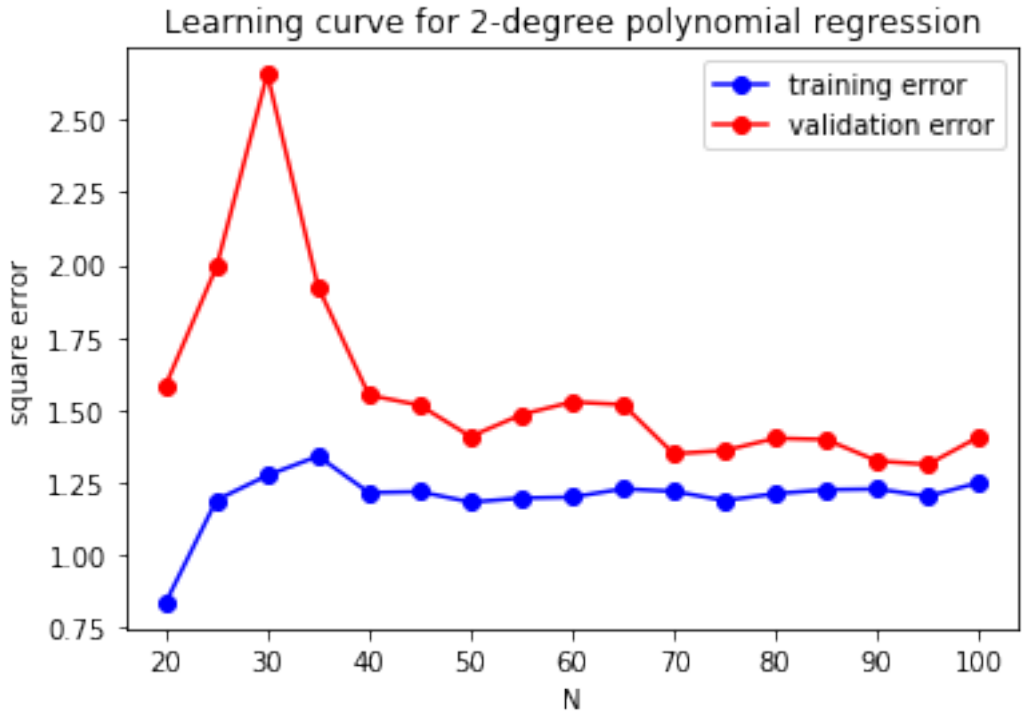
[18]:
```
data = load_data('data/bv_data.csv')
x = data[:, 0]
y = data[:, 1]
```

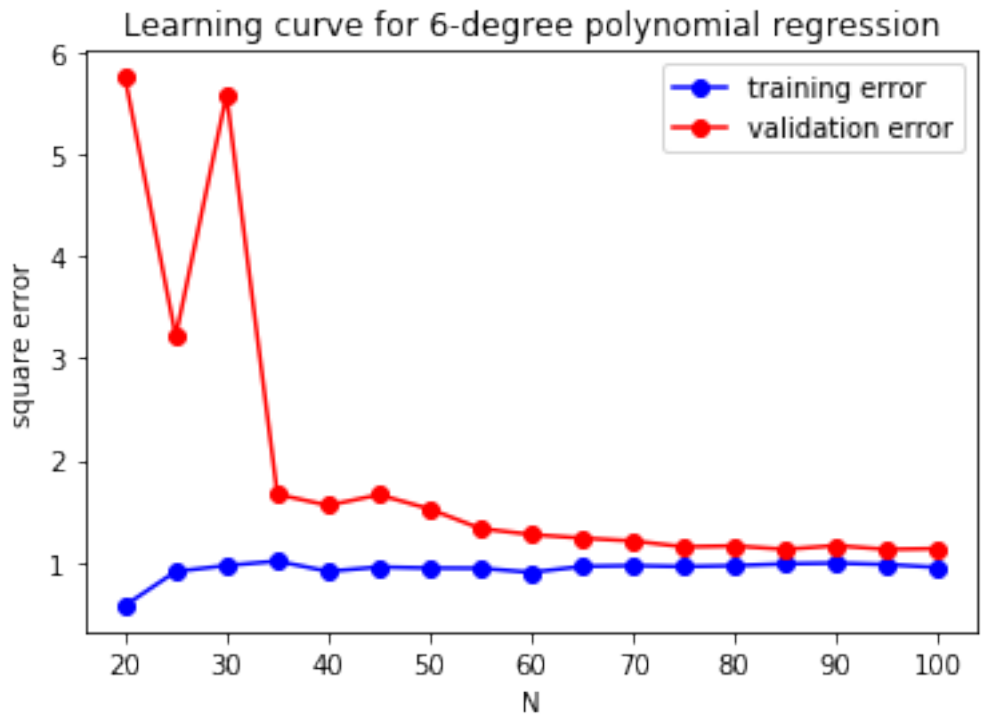
Write your code below for solving problem 2 part B:

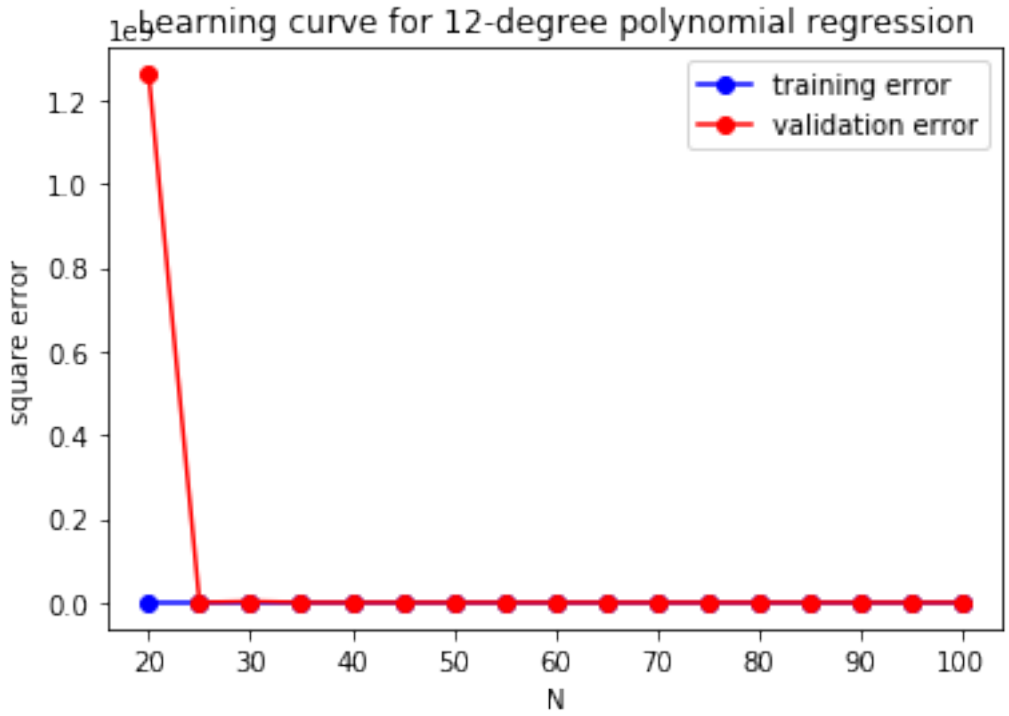
[33]:
```
def crossvalidation(N, degree):
    # Initialize kfold cross-validation object with 5 folds:
    num_folds = 5
    kf = KFold(n_splits=num_folds)
    x_new, y_new = x[:N], y[:N]
```

```
    err_train = 0
    err_test = 0
    for train_index, test_index in kf.split(x_new): # Iterate through␣
 ↪cross-validation folds
        # Training and testing data points for this fold:
        x_train, x_test = x_new[train_index], x_new[test_index]
        y_train, y_test = y_new[train_index], y_new[test_index]
        coefficients = np.polyfit(x_train, y_train, degree)
        y_train_predict = np.polyval(coefficients, x_train)
        y_test_predict = np.polyval(coefficients, x_test)
        err_train += np.mean((y_train_predict - y_train) ** 2)
        err_test += np.mean((y_test_predict - y_test) ** 2)

    avg_err_train = err_train/num_folds
    avg_err_test = err_test/num_folds
    return avg_err_train, avg_err_test


degrees = [1,2,6,12]
N = np.arange(20,105,5)

for degree in degrees:
    Err_train = np.array([])
    Err_test = np.array([])
    for n in N:
        err_train, err_test = crossvalidation(n, degree)
        Err_train = np.append(Err_train, err_train)
        Err_test = np.append(Err_test, err_test)
    plt.plot(N, Err_train, color = 'b',marker = 'o', label = "training error")
    plt.plot(N, Err_test, color = 'r',marker = 'o', label = "validation error")
    plt.legend(['training error', 'validation error'], loc = 'best')
    plt.xlabel('N')
    plt.ylabel('square error')
    plt.title('Learning curve for %i-degree polynomial regression' %degree)

    plt.show()
```


Learning curve for 1-degree polynomial regression
training error
validation error
square error
N
2.6
2.4
2.2
2.0
1.8
1.6
1.4
20
30
40
50
60
70
80
90
100


Learning curve for 2-degree polynomial regression
training error
validation error
square error
N
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
20
30
40
50
60
70
80
90
100


Learning curve for 6-degree polynomial regression
training error
validation error
square error
N
1
2
3
4
5
6
20
30
40
50
60
70
80
90
100


Learning curve for 12-degree polynomial regression
1e9
training error
validation error
square error
N
0.0
0.2
0.4
0.6
0.8
1.0
1.2
20
30
40
50
60
70
80
90
100

3_notebook

January 15, 2020

1 Problem 3

Use this notebook to write your code for problem 3 by filling in the sections marked `# TODO` and running all cells.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import itertools

from perceptron_helper import (
    predict,
    plot_data,
    boundary,
    plot_perceptron,
)

%matplotlib inline
```

1.1 Implementation of Perceptron

First, we will implement the perceptron algorithm. Fill in the `update_perceptron()` function so that it finds a single misclassified point and updates the weights and bias accordingly. If no point exists, the weights and bias should not change.

Hint: You can use the `predict()` helper method, which labels a point 1 or -1 depending on the weights and bias.

[2]:
```
def update_perceptron(X, Y, w, b):
    """
    This method updates a perceptron model. Takes in the previous weights
    and returns weights after an update, which could be nothing.

Inputs:
        X: A (N, D) shaped numpy array containing N points of D dimensions.
        Y: A (N, ) shaped numpy array containing the N labels (one for each␣
 ↪point in X).
        w: A (D, ) shaped numpy array containing the initial weight vector of D␣
 ↪dimensions.
        b: A float containing the bias term.
```

```
    Output:
        next_w: A (D, ) shaped numpy array containing the next weight vector
                after updating on a single misclassified point, if one exists.
        next_b: The next float bias term after updating on a single
                misclassified point, if one exists.
    """
    next_w, next_b = np.copy(w), np.copy(b)

    #=============================================
    # TODO: Implement update rule for perceptron.
    #=============================================

    for i in range(len(X)):
        point = []
        if predict(X[i], w, b) != Y[i]:
            next_w = next_w + np.dot(X[i], Y[i])
            next_b = next_b + Y[i]
            point.append(X[i])
            point.append(Y[i])
            break
    return next_w, next_b, point
```

Next you will fill in the `run_perceptron()` method. The method performs single updates on a misclassified point until convergence, or max_iter updates are made. The function will return the final weights and bias. You should use the `update_perceptron()` method you implemented above.

[3]:
```
def run_perceptron(X, Y, w, b, max_iter):
    """
    This method runs the perceptron learning algorithm. Takes in initial␣
↪weights
    and runs max_iter update iterations. Returns final weights and bias.

    Inputs:
        X: A (N, D) shaped numpy array containing N points of D dimensions.
        Y: A (N, ) shaped numpy array containing the N labels (one for each␣
↪point in X).
        w: A (D, ) shaped numpy array containing the initial weight vector of D␣
↪dimensions.
        b: A float containing the initial bias term.
        max_iter: An int for the maximum number of updates evaluated.

    Output:
        w: A (D, ) shaped numpy array containing the final weight vector.
        b: The final float bias term.
    """
```

```
    iteration = 0
    convergence = False

    while (iteration < max_iter) and (convergence == False):
        w_new, b_new, point = update_perceptron(X, Y, w, b)
        print("t = %d, b = %d, w = %s, [x1, x2, Y] = %s" % (iteration, b, w,␣
↪point))
        if (w_new == w).all() and (b_new == b):
            convergence = True
        iteration += 1
        w = w_new
        b = b_new

    return w, b
```

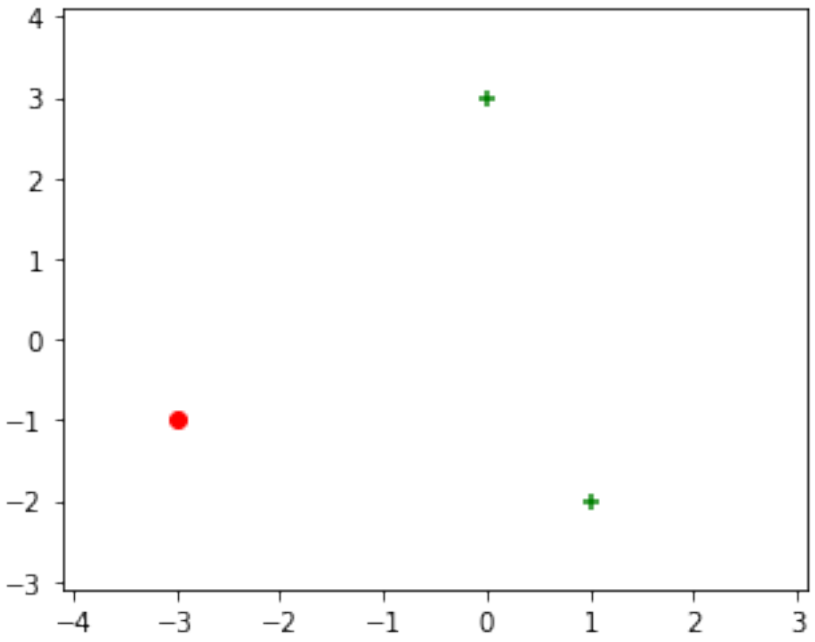
2 Problem 3A

2.1 Visualizing a Toy Dataset

We will begin by training our perceptron on a toy dataset of 3 points. The green points are labelled +1 and the red points are labelled -1. We use the helper function `plot_data()` to do so.

[4]:
```
X = np.array([[ -3, -1], [0, 3], [1, -2]])
Y = np.array([ -1, 1, 1])
```

[5]:
```
fig = plt.figure(figsize=(5,4))
ax = fig.gca(); ax.set_xlim(-4.1, 3.1); ax.set_ylim(-3.1, 4.1)
plot_data(X, Y, ax)
```



2.2 Running the Perceptron

Next, we will run the perceptron learning algorithm on this dataset. Update the code to show the weights and bias at each timestep and the misclassified point used in each update. You may change the `update_perceptron()` method to do this, but be sure to update the starter code as well to reflect those changes.

Run the below code, and fill in the corresponding table in the set.

[6]:
```
# Initialize weights and bias.
weights = np.array([0.0, 1.0])
bias = 0.0

weights, bias = run_perceptron(X, Y, weights, bias, 16)

print()
print ("final w = %s, final b = %.1f" % (weights, bias))
```

```
t = 0, b = 0, w = [0. 1.], [x1, x2, Y] = [array([ 1, -2]), 1]
t = 1, b = 1, w = [ 1. -1.], [x1, x2, Y] = [array([0, 3]), 1]
t = 2, b = 2, w = [1. 2.], [x1, x2, Y] = [array([ 1, -2]), 1]
t = 3, b = 3, w = [2. 0.], [x1, x2, Y] = []

final w = [2. 0.], final b = 3.0
```

2.3 Visualizating the Perceptron

Getting all that information in table form isn't very informative. Let us visualize what the decision boundaries are at each timestep instead.

The helper functions `boundary()` and `plot_perceptron()` plot a decision boundary given a perceptron weights and bias. Note that the equation for the decision boundary is given by:

$$w_1 x_1 + w_2 x_2 + b = 0.$$

Using some algebra, we can obtain x_2 from x_1 to plot the boundary as a line.

$$x_2 = \frac{-w_1 x_2 - b}{w_2}.$$

Below is a redefinition of the `run_perceptron()` method to visualize the points and decision boundaries at each timestep instead of printing. Fill in the method using your previous `run_perceptron()` method, and the above helper methods.

Hint: The axs element is a list of Axes, which are used as subplots for each timestep. You can do the following:

```
ax = axs[i]
```

to get the plot correponding to $t = i$. You can then use ax.set_title() to title each subplot. You will want to use the `plot_data()` and `plot_perceptron()` helper methods.

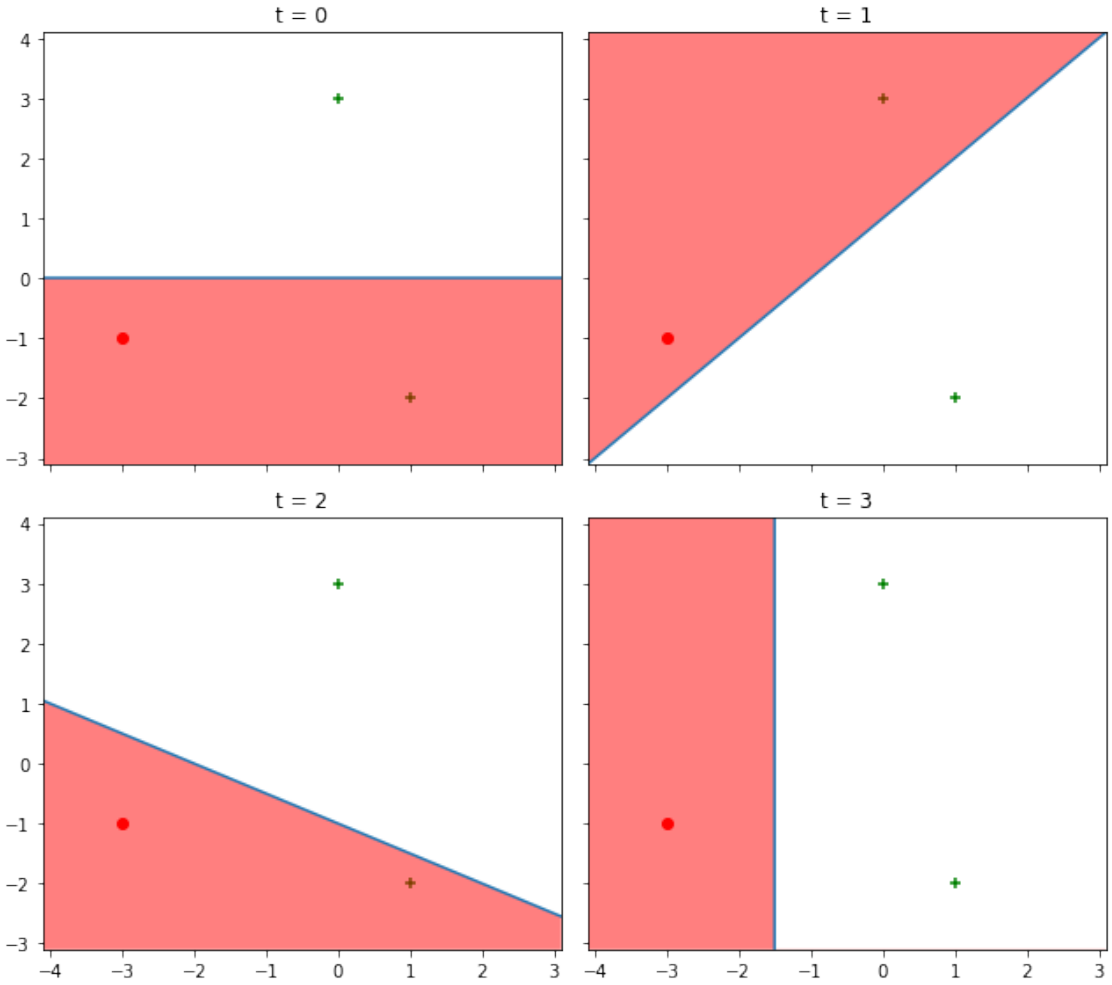
[7]:
```
def run_perceptron(X, Y, w, b, axs, max_iter):
    """
    This method runs the perceptron learning algorithm. Takes in initial␣
↪weights
    and runs max_iter update iterations. Returns final weights and bias.

    Inputs:
        X: A (N, D) shaped numpy array containing N points of D dimensions.
        Y: A (N, ) shaped numpy array containing the N labels (one for each␣
↪point in X).
        w: A (D, ) shaped numpy array containing the initial weight vector of D␣
↪dimensions.
        b: A float containing the initial bias term.
        axs: A list of Axes that contain suplots for each timestep.
        max_iter: An int for the maximum number of updates evaluated.

    Output:
        The final weight and bias vectors.
    """

    iteration = 0
    convergence = False

    while (iteration < max_iter) and (convergence == False):
        ax = axs[iteration]
        ax.set_title("t = %d" % iteration)
        plot_data(X, Y, ax)
        plot_perceptron(w, b, ax)

        w_new, b_new, point = update_perceptron(X, Y, w, b)
        print("t = %d, b = %d, w = %s, [x1, x2, Y] = %s" % (iteration, b, w,␣
↪point))
        if (w_new == w).all() and (b_new == b):
            convergence = True
        iteration += 1
        w = w_new
        b = b_new

    return w, b
```

Run the below code to get a visualization of the perceptron algorithm. The red region are areas the perceptron thinks are negative examples.

[8]:
```
# Initialize weights and bias.
weights = np.array([0.0, 1.0])
bias = 0.0
```

```
# Note: there are 4 subplots and max_iter=4 below. Plot BEFORE each timestep␣
 ↪update.
f, ax_arr = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(9,8))
axs = list(itertools.chain.from_iterable(ax_arr))
for ax in axs:
    ax.set_xlim(-4.1, 3.1); ax.set_ylim(-3.1, 4.1)

run_perceptron(X, Y, weights, bias, axs, 4)

f.tight_layout()
```

```
t = 0, b = 0, w = [0. 1.], [x1, x2, Y] = [array([ 1, -2]), 1]
t = 1, b = 1, w = [ 1. -1.], [x1, x2, Y] = [array([0, 3]), 1]
t = 2, b = 2, w = [1. 2.], [x1, x2, Y] = [array([ 1, -2]), 1]
t = 3, b = 3, w = [2. 0.], [x1, x2, Y] = []
```



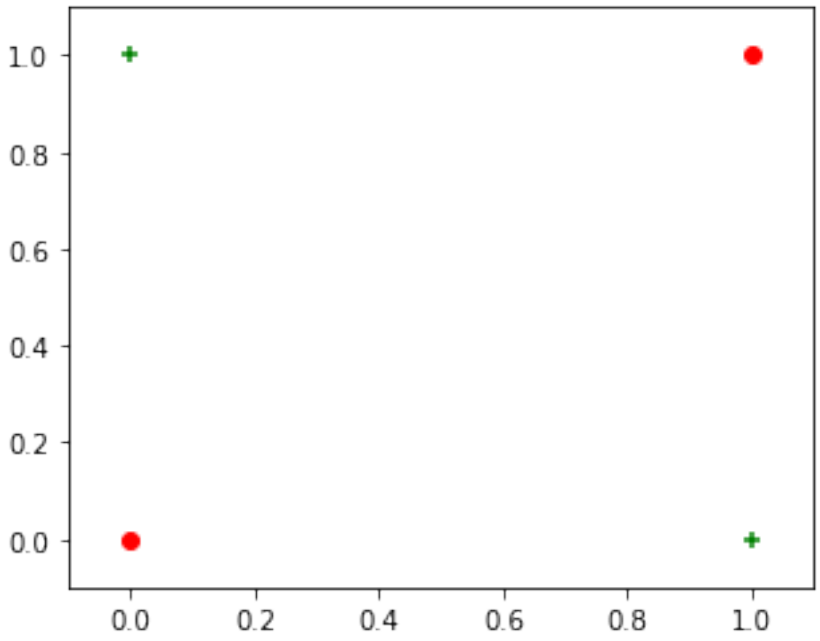
3 Problem 3C

3.1 Visualize a Non-linearly Separable Dataset.

We will now work on a dataset that cannot be linearly separated, namely one that is generated by the XOR function.

[9]:
```
X = np.array([[0, 1], [1, 0], [0, 0], [1, 1]])
Y = np.array([1, 1, -1, -1])
```

[10]:
```
fig = plt.figure(figsize=(5,4))
ax = fig.gca(); ax.set_xlim(-0.1, 1.1); ax.set_ylim(-0.1, 1.1)
plot_data(X, Y, ax)
```

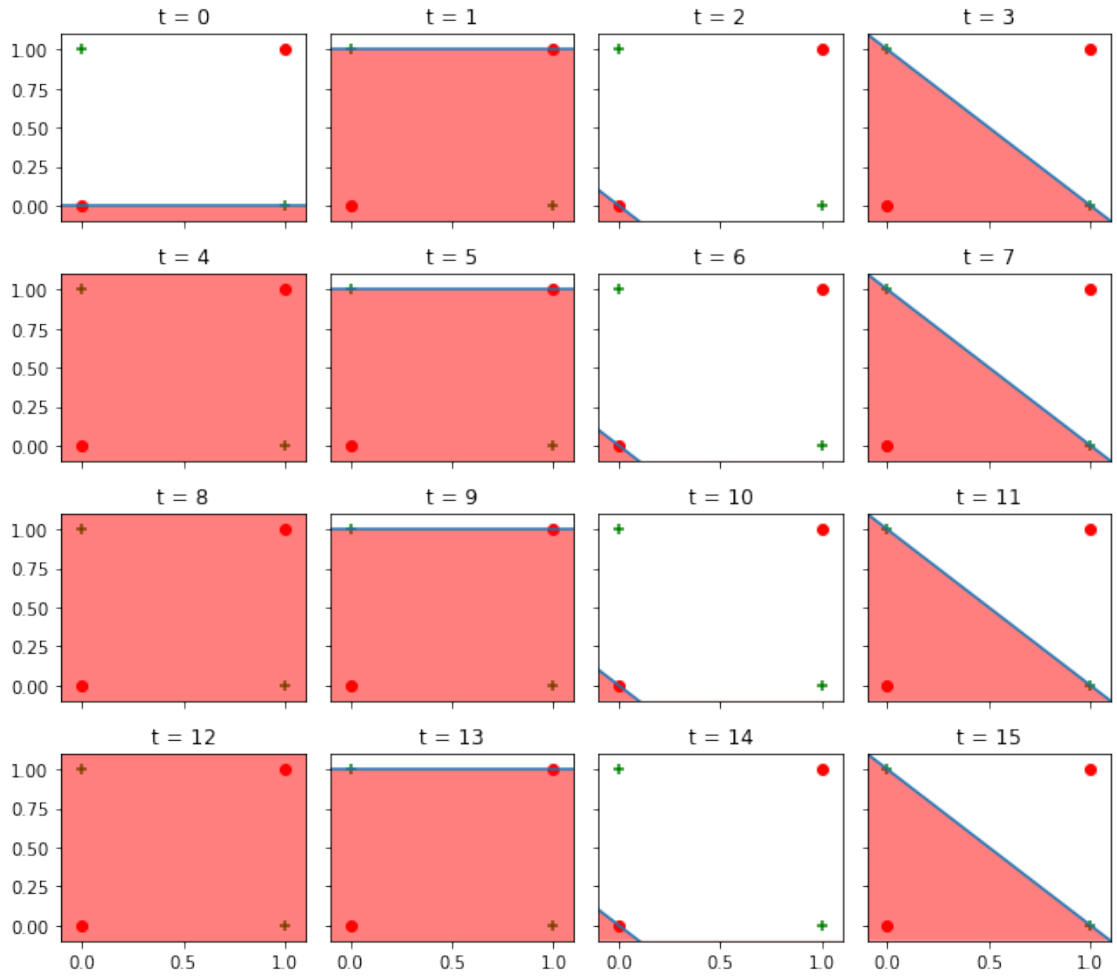


We will now run the perceptron algorithm on this dataset. We will limit the total timesteps this time, but you should see a pattern in the updates. Run the below code.

[11]:
```
# Initialize weights and bias.
weights = np.array([0.0, 1.0])
bias = 0.0

# Note: there are 16 subplots and max_iter=16. Plot BEFORE each timestep update.
f, ax_arr = plt.subplots(4, 4, sharex=True, sharey=True, figsize=(9,8))
axs = list(itertools.chain.from_iterable(ax_arr))
for ax in axs:
    ax.set_xlim(-0.1, 1.1); ax.set_ylim(-0.1, 1.1)

run_perceptron(X, Y, weights, bias, axs, 16)
```

```
f.tight_layout()
```

```
t = 0, b = 0, w = [0. 1.], [x1, x2, Y] = [array([0, 0]), -1]
t = 1, b = -1, w = [0. 1.], [x1, x2, Y] = [array([1, 0]), 1]
t = 2, b = 0, w = [1. 1.], [x1, x2, Y] = [array([0, 0]), -1]
t = 3, b = -1, w = [1. 1.], [x1, x2, Y] = [array([1, 1]), -1]
t = 4, b = -2, w = [0. 0.], [x1, x2, Y] = [array([0, 1]), 1]
t = 5, b = -1, w = [0. 1.], [x1, x2, Y] = [array([1, 0]), 1]
t = 6, b = 0, w = [1. 1.], [x1, x2, Y] = [array([0, 0]), -1]
t = 7, b = -1, w = [1. 1.], [x1, x2, Y] = [array([1, 1]), -1]
t = 8, b = -2, w = [0. 0.], [x1, x2, Y] = [array([0, 1]), 1]
t = 9, b = -1, w = [0. 1.], [x1, x2, Y] = [array([1, 0]), 1]
t = 10, b = 0, w = [1. 1.], [x1, x2, Y] = [array([0, 0]), -1]
t = 11, b = -1, w = [1. 1.], [x1, x2, Y] = [array([1, 1]), -1]
t = 12, b = -2, w = [0. 0.], [x1, x2, Y] = [array([0, 1]), 1]
t = 13, b = -1, w = [0. 1.], [x1, x2, Y] = [array([1, 0]), 1]
t = 14, b = 0, w = [1. 1.], [x1, x2, Y] = [array([0, 0]), -1]
t = 15, b = -1, w = [1. 1.], [x1, x2, Y] = [array([1, 1]), -1]
```



4_notebook_part1

January 15, 2020

1 Problem 4, Parts C-E: Stochastic Gradient Descent Visualization

In this Jupyter notebook, we visualize how SGD works. This visualization corresponds to parts C-E of question 4 in set 1.

Use this notebook to write your code for problem 4 parts C-E by filling in the sections marked `# TODO` and running all cells.

[1]:
```
# Setup.

import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML

from sgd_helper import (
    generate_dataset1,
    generate_dataset2,
    plot_dataset,
    plot_loss_function,
    animate_convergence,
    animate_sgd_suite
)
```

1.1 Problem 4C: Implementation of SGD

Fill in the loss, gradient, and SGD functions according to the guidelines given in the problem statement in order to perform SGD.

[2]:
```
def loss(X, Y, w):
    '''
    Calculate the squared loss function.

    Inputs:
        X: A (N, D) shaped numpy array containing the data points.
        Y: A (N, ) shaped numpy array containing the (float) labels of the data␣
 ↪points.
        w: A (D, ) shaped numpy array containing the weight vector.

    Outputs:
```

```
        The loss evaluated with respect to X, Y, and w.
    '''

    sq_loss = np.sum((Y - np.dot(X, w)) ** 2)
    return sq_loss

    pass

def gradient(x, y, w):
    '''
    Calculate the gradient of the loss function with respect to
    a single point (x, y), and using weight vector w.

    Inputs:
        x: A (D, ) shaped numpy array containing a single data point.
        y: The float label for the data point.
        w: A (D, ) shaped numpy array containing the weight vector.

    Output:
        The gradient of the loss with respect to x, y, and w.
    '''

    sq_gradient = -2 * x * (y - np.dot(w, x))
    return sq_gradient

    pass

def SGD(X, Y, w_start, eta, N_epochs):
    '''
    Perform SGD using dataset (X, Y), initial weight vector w_start,
    learning rate eta, and N_epochs epochs.

    Inputs:
        X: A (N, D) shaped numpy array containing the data points.
        Y: A (N, ) shaped numpy array containing the (float) labels of the data␣
 ↪points.
        w_start:  A (D, ) shaped numpy array containing the weight vector␣
 ↪initialization.
        eta: The step size.
        N_epochs: The number of epochs (iterations) to run SGD.

    Outputs:
        W: A (N_epochs, D) shaped array containing the weight vectors from all␣
 ↪iterations.
        losses: A (N_epochs, ) shaped array containing the losses from all␣
 ↪iterations.
    '''
```

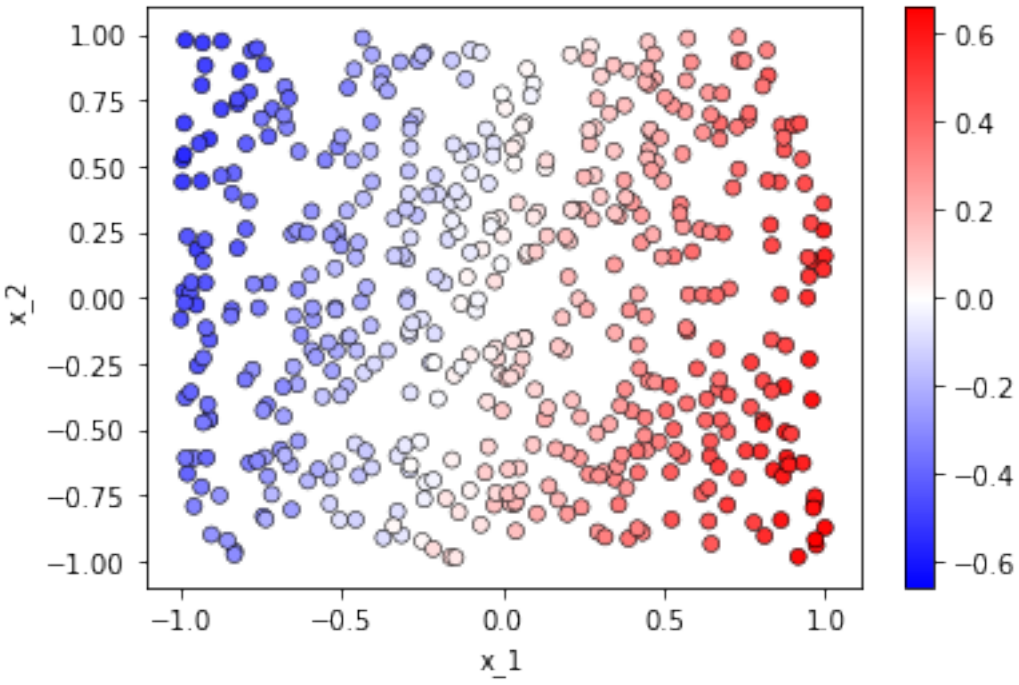
```
    w = w_start
    losses = np.array([])
    W = np.array([])

    for i in range(N_epochs):
        index = np.random.permutation(len(Y))
        W = np.append(W, w)
        losses = np.append(losses, loss(X, Y, w))
        for j in index:
            w = w - eta * gradient(X[j], Y[j], w)

    return W.reshape((N_epochs, len(w))), losses

    pass
```

1.2 Problem 4D: Visualization

1.2.1 Dataset

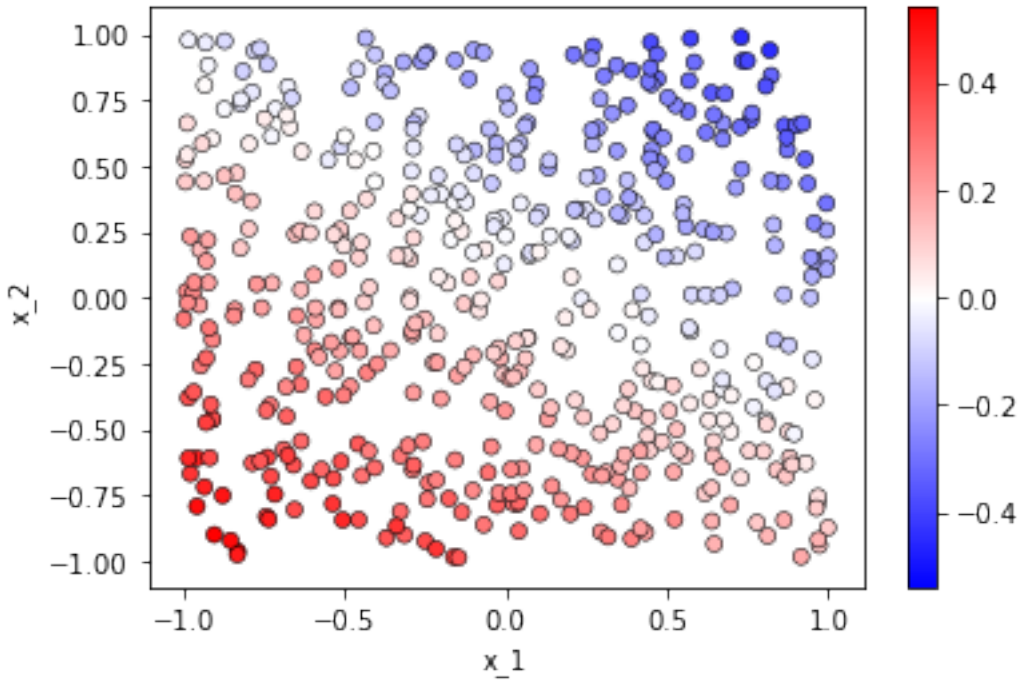
We'll start off by generating two simple 2-dimensional datasets. For simplicity we do not consider separate training and test sets.

[3]:
```
X1, Y1 = generate_dataset1()
plot_dataset(X1, Y1)
```



[3]:
```
(<Figure size 432x288 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x13a00678ac8>)
```

[4]:
```
X2, Y2 = generate_dataset2()
plot_dataset(X2, Y2)
```



[4]:
```
(<Figure size 432x288 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x13a009b51d0>)
```

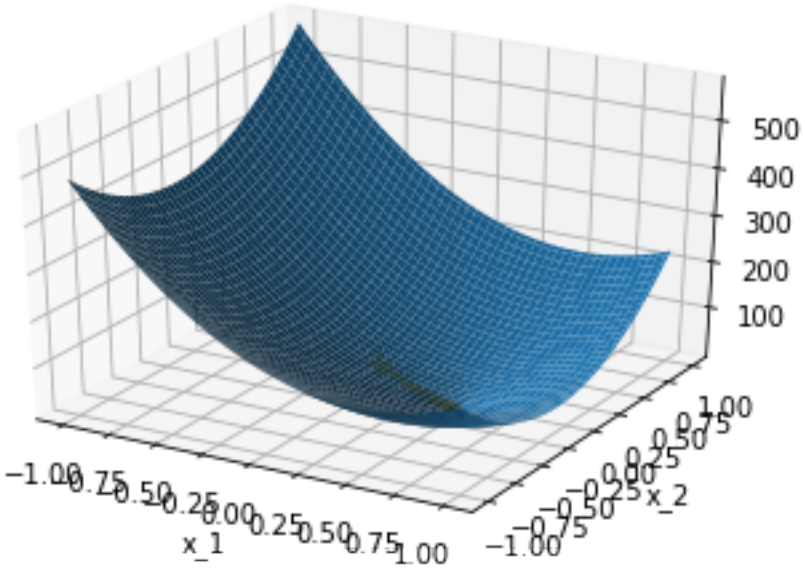
1.2.2 SGD from a single point

First, let's visualize SGD from a single starting point:

[5]:
```
# Parameters to feed the SGD.
# <FR> changes the animation speed.
params = ({'w_start': [0.01, 0.01], 'eta': 0.00001},)
N_epochs = 1000
FR = 20

# Let's animate it!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR)
HTML(anim.to_html5_video())
```

```
Performing SGD with parameters {'w_start': [0.01, 0.01], 'eta': 1e-05} ...

Animating...
```

[5]: <IPython.core.display.HTML object>

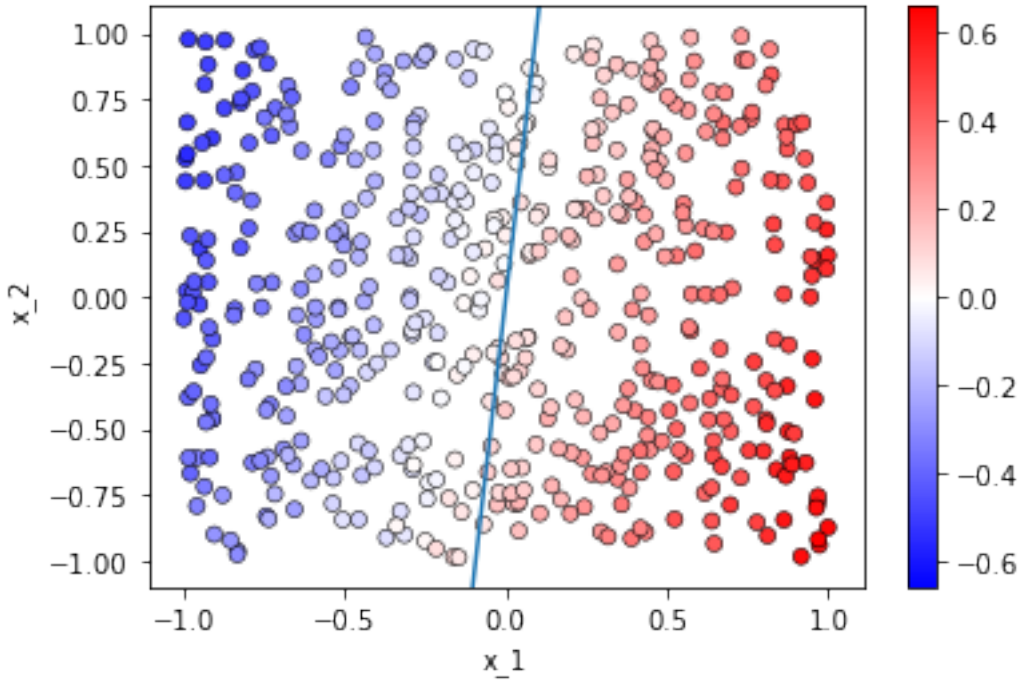


Let's view how the weights change as the algorithm converges:

[6]:
```
# Parameters to feed the SGD.
params = ({'w_start': [0.01, 0.01], 'eta': 0.00001},)
N_epochs = 1000
FR = 20

# Let's do it!
W, _ = SGD(X1, Y1, params[0]['w_start'], params[0]['eta'], N_epochs)
anim = animate_convergence(X1, Y1, W, FR)
HTML(anim.to_html5_video())
```

```
Animating...
```

[6]: <IPython.core.display.HTML object>



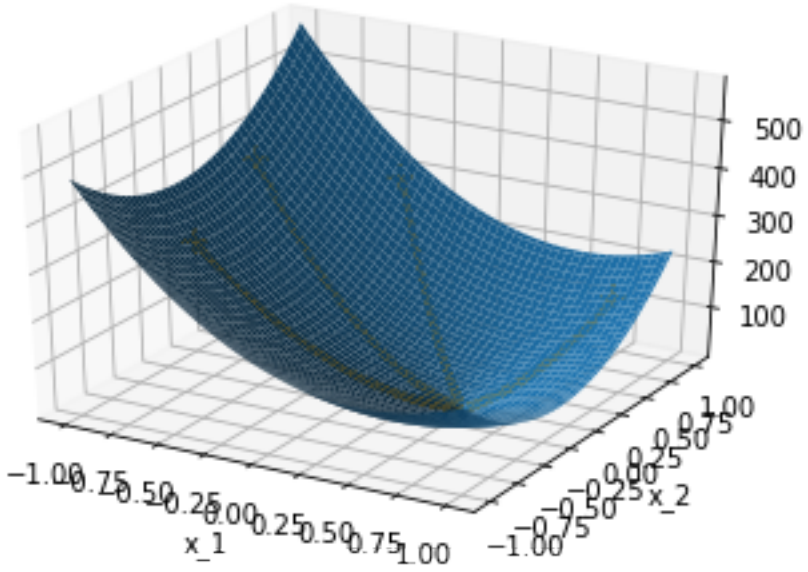
1.2.3 SGD from multiple points

Now, let's visualize SGD from multiple arbitrary starting points:

[7]:
```
# Parameters to feed the SGD.
# Here, we specify each different set of initializations as a dictionary.
params = (
    {'w_start': [-0.8, -0.3], 'eta': 0.00001},
    {'w_start': [-0.9, 0.4], 'eta': 0.00001},
    {'w_start': [-0.4, 0.9], 'eta': 0.00001},
    {'w_start': [0.8, 0.8], 'eta': 0.00001},
)
N_epochs = 1000
FR = 20

# Let's go!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR)
HTML(anim.to_html5_video())
```

```
Performing SGD with parameters {'w_start': [-0.8, -0.3], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [-0.9, 0.4], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [-0.4, 0.9], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [0.8, 0.8], 'eta': 1e-05} ...

Animating...
```

[7]: <IPython.core.display.HTML object>

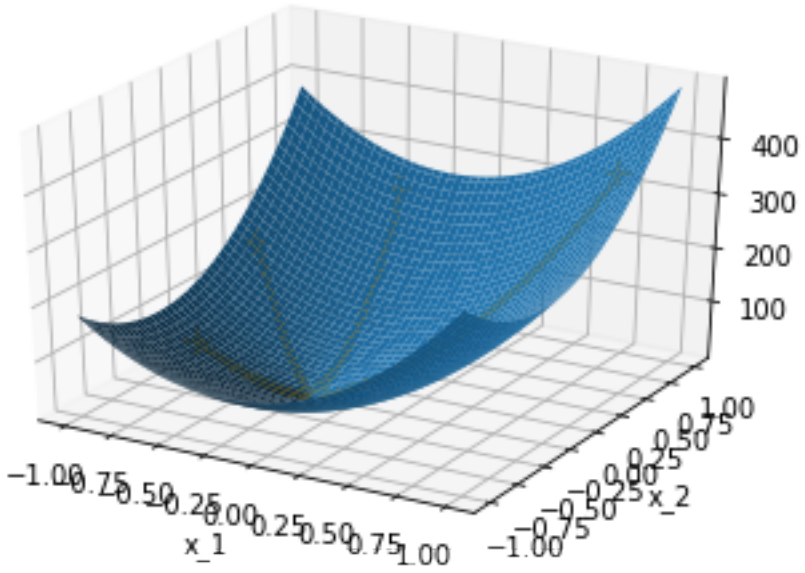


Let's do the same thing but with a different dataset:

[8]:
```
# Parameters to feed the SGD.
params = (
    {'w_start': [-0.8, -0.3], 'eta': 0.00001},
    {'w_start': [-0.9, 0.4], 'eta': 0.00001},
    {'w_start': [-0.4, 0.9], 'eta': 0.00001},
    {'w_start': [0.8, 0.8], 'eta': 0.00001},
)
N_epochs = 1000
FR = 20

# Animate!
anim = animate_sgd_suite(SGD, loss, X2, Y2, params, N_epochs, FR)
HTML(anim.to_html5_video())
```

```
Performing SGD with parameters {'w_start': [-0.8, -0.3], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [-0.9, 0.4], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [-0.4, 0.9], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [0.8, 0.8], 'eta': 1e-05} ...

Animating...
```

[8]: <IPython.core.display.HTML object>



1.3 Problem 4E: SGD with different step sizes

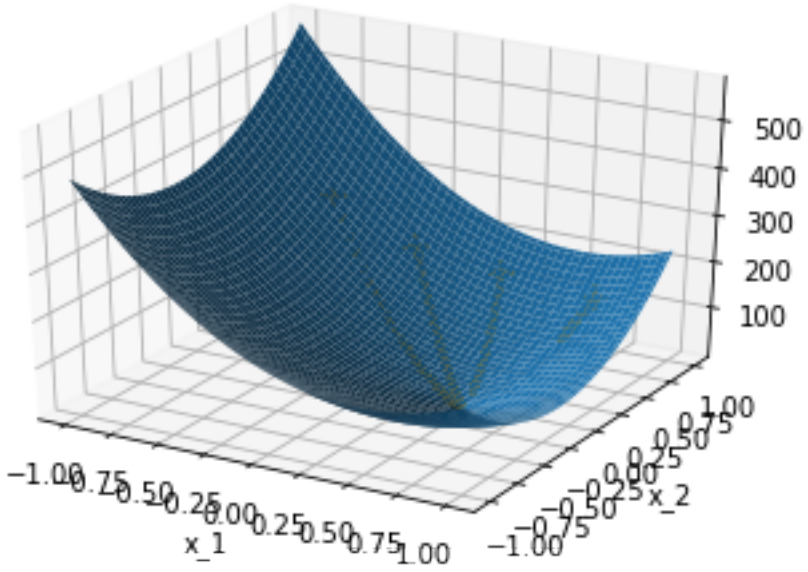
Now, let's visualize SGD with different step sizes (eta):

(For ease of visualization: the trajectories are ordered from left to right by increasing eta value. Also, note that we use smaller values of N_epochs and FR here for easier visualization.)

[9]:
```
# Parameters to feed the SGD.
params = (
    {'w_start': [0.7, 0.8], 'eta': 0.00001},
    { 'w_start': [0.2, 0.8], 'eta': 0.00005},
    {'w_start': [-0.2, 0.7], 'eta': 0.0001},
    {'w_start': [-0.6, 0.6], 'eta': 0.0002},
)
N_epochs = 100
FR = 2

# Go!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)
HTML(anim.to_html5_video())
```

```
Performing SGD with parameters {'w_start': [0.7, 0.8], 'eta': 1e-05} ...
Performing SGD with parameters {'w_start': [0.2, 0.8], 'eta': 5e-05} ...
Performing SGD with parameters {'w_start': [-0.2, 0.7], 'eta': 0.0001} ...
Performing SGD with parameters {'w_start': [-0.6, 0.6], 'eta': 0.0002} ...

Animating...
```

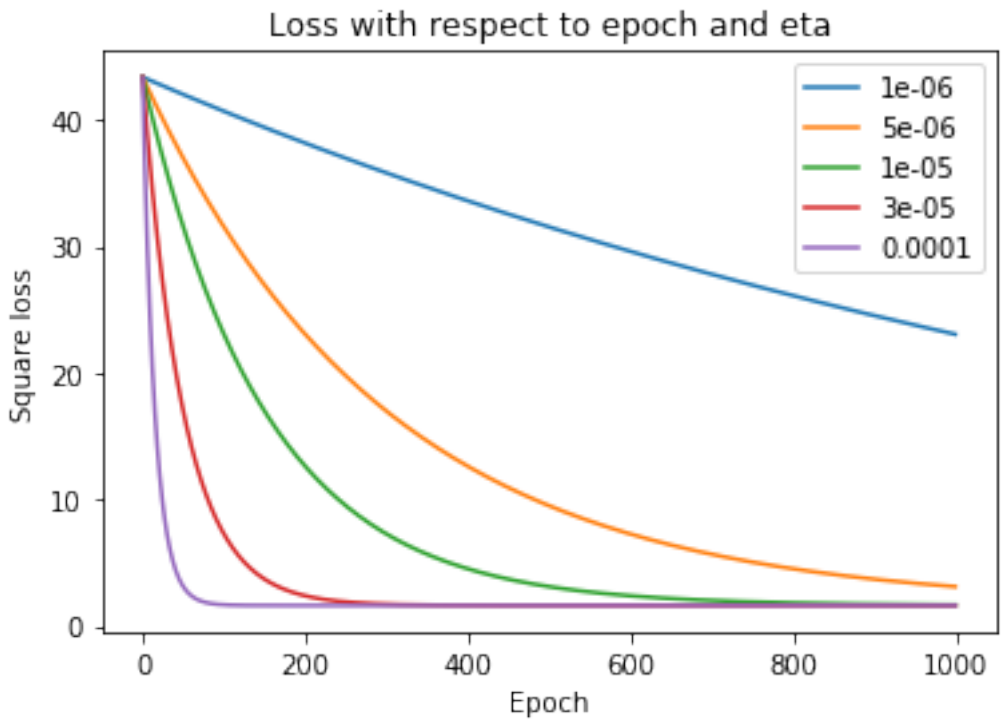
[9]: <IPython.core.display.HTML object>



1.3.1 Plotting SGD Convergence

Let's visualize the difference in convergence rates a different way. Plot the loss with respect to epoch (iteration) number for each value of eta on the same graph.

[10]:
```
'''Plotting SGD convergence'''

eta_vals = [1e-6, 5e-6, 1e-5, 3e-5, 1e-4]
w_start = [0.01, 0.01]
N_epochs = 1000

for eta in eta_vals:
    W, losses = SGD(X1, Y1, w_start, eta, N_epochs)
    plt.plot(losses, label=str(eta))

plt.title('Loss with respect to epoch and eta')
plt.xlabel('Epoch')
plt.ylabel('Square loss')
plt.legend(loc = "best")
plt.show()
```



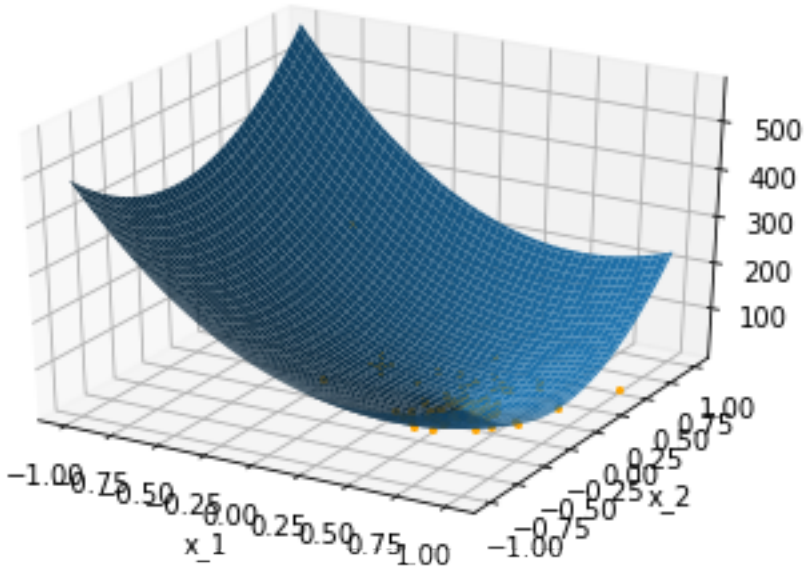
Clearly, a big step size results in fast convergence! Why don't we just set eta to a really big value, then? Say, eta=1?

(Again, note that the FR is lower for this animation.)

[11]:
```
# Parameters to feed the SGD.
params = ({'w_start': [0.01, 0.01], 'eta': 1},)
N_epochs = 100
FR = 2

# Voila!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)
HTML(anim.to_html5_video())
```

```
Performing SGD with parameters {'w_start': [0.01, 0.01], 'eta': 1} ...

Animating...
```

[11]:
```
<IPython.core.display.HTML object>
```



Just for fun, let's try eta=10 as well. What happens? (Hint: look at W)

[12]:
```
# Parameters to feed the SGD.
w_start = [0.01, 0.01]
eta = 10
N_epochs = 100

# Presto!
W, losses = SGD(X1, Y1, w_start, eta, N_epochs)
```

```
C:\Users\Changhao\Anaconda3\lib\site-packages\ipykernel_launcher.py:63:
RuntimeWarning: overflow encountered in multiply
```

1.4 Extra Visualization (not part of the homework problem)

One final visualization! What happens if the loss function has multiple optima?

[13]:
```
# Import different SGD & loss functions.
# In particular, the loss function has multiple optima.
from sgd_multiopt_helper import SGD, loss

# Parameters to feed the SGD.
params = (
    {'w_start': [0.9, 0.9], 'eta': 0.01},
    { 'w_start': [0.0, 0.0], 'eta': 0.01},
    {'w_start': [-0.8, 0.6], 'eta': 0.01},
    {'w_start': [-0.8, -0.6], 'eta': 0.01},
    {'w_start': [-0.4, -0.3], 'eta': 0.01},
)
```

```
N_epochs = 100
FR = 2

# One more time!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)
HTML(anim.to_html5_video())
```

```
␣
↪---------------------------------------------------------------------------

       ModuleNotFoundError                       Traceback (most recent call␣
↪last)

       <ipython-input-13-7d6c7ee7d701> in <module>
         1 # Import different SGD & loss functions.
         2 # In particular, the loss function has multiple optima.
   ----> 3 from sgd_multiopt_helper import SGD, loss
         4
         5 # Parameters to feed the SGD.


       ModuleNotFoundError: No module named 'sgd_multiopt_helper'
```

4_notebook_part2

January 15, 2020

1 Problem 4, Parts F-H: Stochastic Gradient Descent with a Larger Dataset

Use this notebook to write your code for problem 4 parts F-H by filling in the sections marked `# TODO` and running all cells.

[1]:
```
# Setup.

import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

1.1 Problem 4F: Perform SGD with the new dataset

For the functions below, you may re-use your code from parts 4C-E. Note that you can now modify your SGD function to return the final weight vector instead of the weights after every epoch.

[2]:
```
def loss(X, Y, w):
    '''
    Calculate the squared loss function.

    Inputs:
        X: A (N, D) shaped numpy array containing the data points.
        Y: A (N, ) shaped numpy array containing the (float) labels of the data␣
 ↪points.
        w: A (D, ) shaped numpy array containing the weight vector.

    Outputs:
        The loss evaluated with respect to X, Y, and w.
    '''

    sq_loss = np.sum((Y - np.dot(X, w)) ** 2)
    return sq_loss

    pass
```

```
def gradient(x, y, w):
    '''
    Calculate the gradient of the loss function with respect to
    a single point (x, y), and using weight vector w.

    Inputs:
        x: A (D, ) shaped numpy array containing a single data point.
        y: The float label for the data point.
        w: A (D, ) shaped numpy array containing the weight vector.

    Output:
        The gradient of the loss with respect to x, y, and w.
    '''

    sq_gradient = -2 * x * (y - np.dot(w, x))
    return sq_gradient

    pass

def SGD(X, Y, w_start, eta, N_epochs):
    '''
    Perform SGD using dataset (X, Y), initial weight vector w_start,
    learning rate eta, and N_epochs epochs.

    Inputs:
        X: A (N, D) shaped numpy array containing the data points.
        Y: A (N, ) shaped numpy array containing the (float) labels of the data␣
 ↪points.
        w_start:  A (D, ) shaped numpy array containing the weight vector␣
 ↪initialization.
        eta: The step size.
        N_epochs: The number of epochs (iterations) to run SGD.

    Outputs:
        w: A (D, ) shaped array containing the final weight vector.
        losses: A (N_epochs, ) shaped array containing the losses from all␣
 ↪iterations.
    '''
    # No longer W vector, final w only

    w = w_start
    losses = np.array([])

    for i in range(N_epochs):
        index = np.random.permutation(len(Y))
        losses = np.append(losses, loss(X, Y, w))
        for j in index:
```

```
            w = w - eta * gradient(X[j], Y[j], w)

    return w, losses

    pass
```

Next, we need to load the dataset. In doing so, the following function may be helpful:

[3]:
```
def load_data(filename):
    """
    Function loads data stored in the file filename and returns it as a numpy␣
 ↪ndarray.

    Inputs:
        filename: GeneratorExitiven as a string.

    Outputs:
        Data contained in the file, returned as a numpy ndarray
    """
    return np.loadtxt(filename, skiprows=1, delimiter=',')
```

Now, load the dataset in `sgd_data.csv` and run SGD using the given parameters; print out the final weights.

[5]:
```
data = load_data("sgd_data.csv")
X_0 = data[:,:-1]
Y = data[:,-1]
X = np.c_[np.ones(X_0.shape[0]), X_0] # add one more column for threshold

eta = np.exp(-15)
b = 0.001
w_start = [b, 0.001, 0.001, 0.001, 0.001]
N_epochs = 800

w, losses = SGD(X, Y, w_start, eta, N_epochs)

print(w)
```

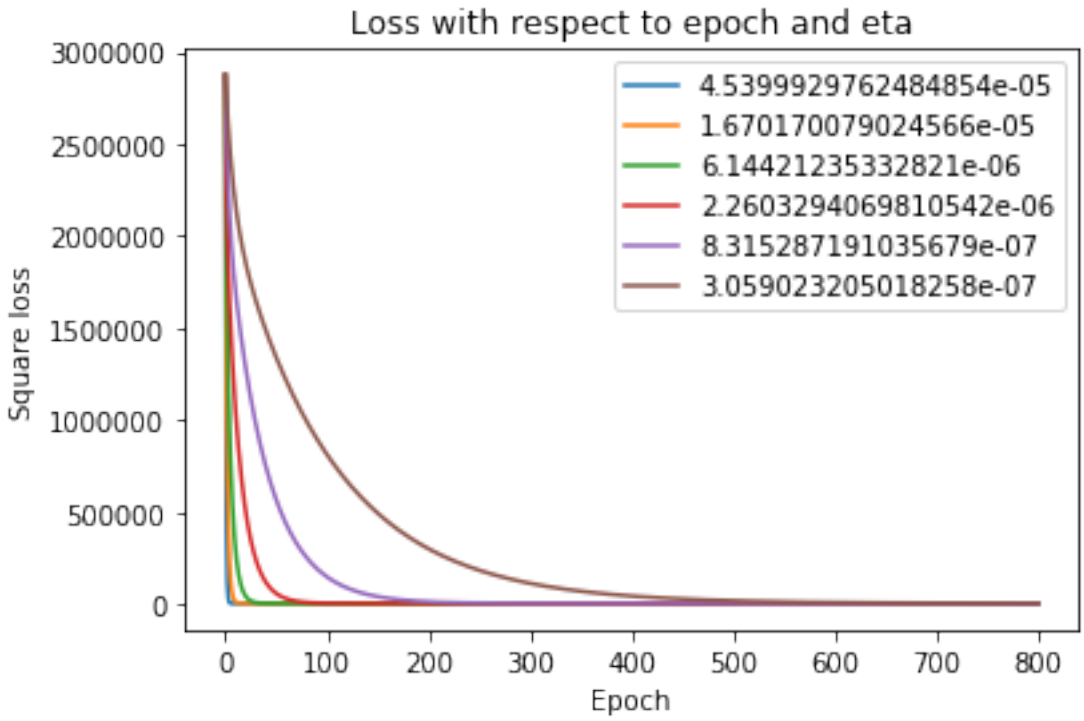
```
[ -0.22717707  -5.94209998   3.94391318 -11.72382875   8.78568403]
```

1.2 Problem 4G: Convergence of SGD

This problem examines the convergence of SGD for different learning rates. Please implement your code in the cell below:

[7]:
```
'''Plotting SGD convergence'''

eta_vals = [np.exp(-10), np.exp(-11), np.exp(-12), np.exp(-13), np.exp(-14), np.
 ↪exp(-15)]
w_start = [0.001, 0.001, 0.001, 0.001, 0.001]
```

```
N_epochs = 800

for eta in eta_vals:
    W, losses = SGD(X, Y, w_start, eta, N_epochs)
    plt.plot(losses, label=str(eta))

plt.title('Loss with respect to epoch and eta')
plt.xlabel('Epoch')
plt.ylabel('Square loss')
plt.legend(loc = "best")
plt.show()
```



1.3 Problem 4H

Provide your code for computing the least-squares analytical solution below.

[8]:
```
pinv_X = np.linalg.pinv(X) # compute pseudo-inverse of X
w = np.dot(pinv_X,Y) # w = pseudo-inverse * Y
print(w)
```

```
[ -0.31644251  -5.99157048   4.01509955 -11.93325972   8.99061096]
```