1 Basics

Question A: The hypothesis set is a set of hypotheses that attempt to approximate the target function.

Question B: A linear model has the following form for its hypothesis set: $w^T x + b$.

Question C: Overfitting is when test error is much greater than training error; this means that the model trained itself to specifically fit only to the training set (and the set's specific kind of noise) and, when deployed to work on other tests, becomes less accurate.

Question D: One, we can use some kind of k-fold validation to determine which models perform less accurately than others. Two, we can train on more data to reduce levels of variance.

Question E: Training data is what we use to learn and what we use to find parameters of our final model. Test data is what we will use to test our final model to see how accurately it can perform. We should never change our model based on information from test data because doing so might introduce bias into our model that's attuned to the test set.

Question F: We assume that our sampled data has a variety of outputs based on inputs; we also assume that the data is sampled without regards to the actual distribution.

Question G: If want to decide whether or not an email is spam with machine learning, the input space, X can be all of our emails in an organized vector form of some sort. Y could be whether or not the email is spam, 0 or 1 in a binary choice.

Question H: The k-fold cross-validation procedure enables users to reuse data as both training and testing data. K-fold cross-validation occurs when the data set is split into k number of partitions; and k-1 partitions are used as training, with the last partition used as testing. This procedure rotates so that every partition is at sometime used as testing.

2 Bias-Variance Tradeoff

Question A:

Given four definitions,

$$\begin{aligned} F(x) &= E_s[f_s(x)] \\ E_{out}(f_s) &= E_x[(f_s(x) - y(x))^2] \\ Bias(x) &= (F(x) - y(x))^2 \\ Var(x) &= E_s[f_s(x) - F(x))^2] \end{aligned}$$

we derive the bias-variance decomposition for the squared error loss function in the following way:

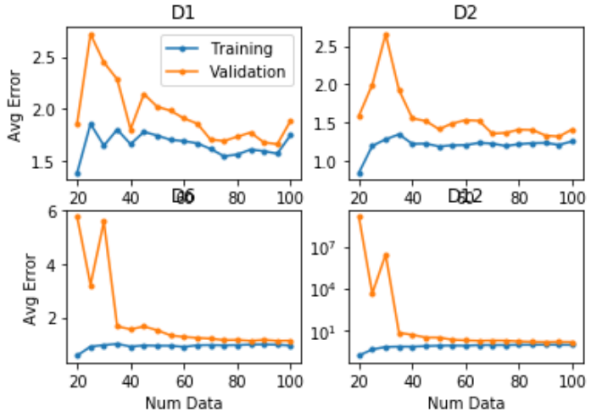
We start with

$$E_s[E_{out}(f_s)]$$

and substitute our second definition. Since we see that both the bias and variance function use $F(x)$, we continue with manipulation:

$$\begin{aligned} &= E_s[E_x[(f_s(x) - y(x))^2]] \\ &= E_s[E_x[(f_s(x) - y(x) + F(x) - F(x))^2]] \\ &= E_s[E_x[(f_s(x) - y(x) + F(x) - F(x))^2]] \\ &= E_s[E_x[(f_s(x) - F(x))^2 + (F(x) - y(x)^2 + 2(f_s(x) - F(x))(F(x) - y(x))] \\ &= E_x[E_s[(f_s(x) - F(x))^2 + (F(x) - y(x)^2 + 2(f_s(x) - F(x))(F(x) - y(x))] \\ &= E_x[E_s[(f_s(x) - F(x))^2)]] + E_x[(F(x) - y(x))^2)] \\ &= [Bias(x) + Var(x)] \end{aligned}$$

Question B:



Question C:
Since the graph for D1 (degree 1) has a higher average error compared to the other graphs, it must have the bias. Also, we know from lecture that bias decreases with model complexity so it makes sense that the low complexity of the model means it has high bias and underfits the model.

Question D:
The model with highest variance should be the D12 (degree 12) model because it has low training error but large validation error at earlier data points. This makes sense since this model has a high complexity, which would correlate with overfitting and high variance.

Question E:
The learning curve of the quadratic model tells us the model would predict more accurately with more data; we see that the graph stabilizes out after the number of data points increases.

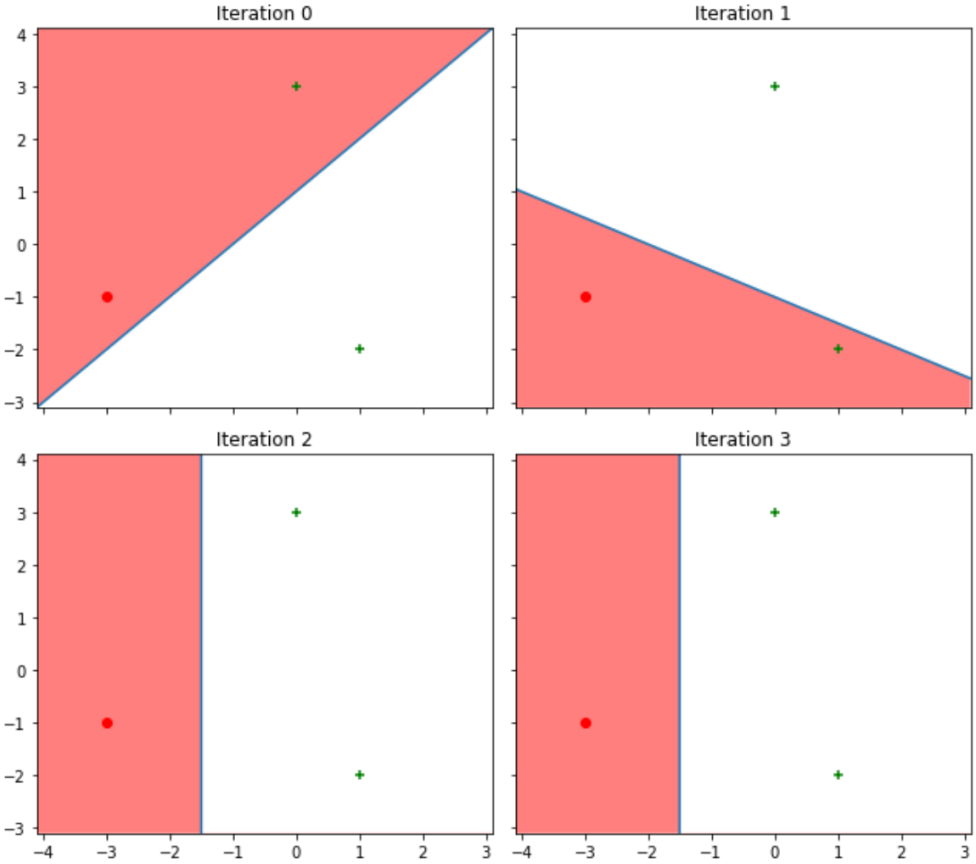
Question F:
Training error is generally lower than validation error because our model is trained on the training data and therefore is more attuned to the training data. We never train on our validation error so it makes sense so that there would be higher error on our validation data, making training error lower than validation error.

Question G:
Based on the learning curves, I would expect D6 (degree 6) model to perform best on unseen data since it seems to have lower validation errors in regards to larger amounts of data points.

3 The Perceptron

Question A:

t	b	w1	w2
0	0.0	0.0	1.0
1	1.0	1.0	-1.0
2	2.0	1.0	2.0
3	3.0	2.0	0.0
4	3.0	2.0	0.0



Question B: In a 2D dataset, the smallest dataset that is not linearly separable has 4 data points that resembles

\+

— —

\+

A 3D dataset would have a similar arrangement of points, except in 3D space and with 5 points. More generally, this means that for an N-dimensional case, the smallest data set that

is NOT linearly separable has N+2 points.

Question C: If a dataset is not linearly separable, then there exists no way for a linear model to correctly identify/classify all the data. Therefore, based on the PL algorithm, there will never be an iteration where all datapoints are correctly classified so the algorithm will continue running and never converge.

4 Stochastic Gradient Descent

Question A:
To define x and w such that the model includes the bias term, we can include a bias term as the zeroth values, as $w_0 = b$ and $x_0 = 1$. So we have $w = (w_0, w_1, ..., w_d)$ and $x = (1, x_1, x_2, ..., x_d)$

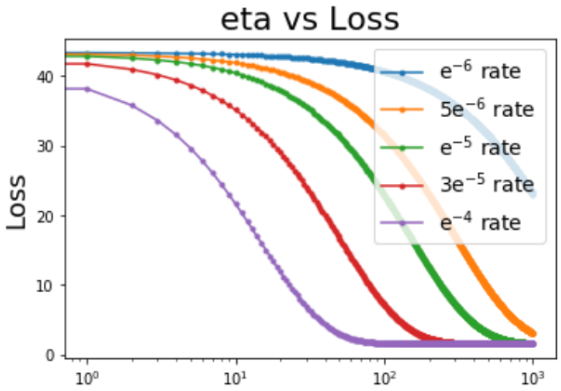
Question B:
$$\sum_{i=1}^{N} -2(y_i - w^T x_i)x_i$$

Question C:
A sample of weights.

```
0: 43.08446558593781, [0.01161911 0.00960535]
1: 42.80858797312999, [0.01323284 0.00921207]
2: 42.534545680530826, [0.01484121 0.00882016]
3: 42.26232649682018, [0.01644424 0.00842962]
4: 41.991918291935725, [0.01804194 0.00804044]
5: 41.72330901653238, [0.01963433 0.00765262]
6: 41.456486701444874, [0.02122143 0.00726615]
7: 41.191439457154495, [0.02280326 0.00688102]
8: 40.92815547325838, [0.02437983 0.00649724]
9: 40.666623017943884, [0.02595116 0.0061148 ]
10: 40.4068304374648, [0.02751727 0.00573369]
11: 40.14876615562224, [0.02907818 0.00535391]
12: 39.8924186732485, [0.0306339  0.00497546]
13: 39.63777656769436, [0.03218445 0.00459832]
14: 39.3848284923196O6, [0.03372985 0.0042225 ]
15: 39.133563175987646, [0.03527012 0.00384 8 ]
16: 38.88396942256283, [0.03680527 0.00347479]
17: 38.63603611041127, [0.03833531 0.00310289]
18: 38.38975219190503, [0.03986027 0.00273228]
19: 38.145106692929716, [0.04138017 0.00236297]
20: 37.90208871239513, [0.04289502 0.00199495]
21: 37.66068742174931, [0.04440483 0.00162821]
22: 37.42089206449599, [0.04590963 0.00126274]
23: 37.18269195571486, [0.04740942 0.00089856]
24: 36.94607648158497, [0.04890423 0.00053564]
25: 36.711035098912426, [0.05039408 0.00017399]
26: 36.477557733465941, [ 0.05187898 -0.0001864 ]
27: 36.24563278547776, [ 0.05335894 -0.00054554]
28: 36.015251117245 3, [ 0.05483398 -0.00090342]
29: 35.7864020646049, [ 0.05630413 -0.00126005]
30: 35.55907543050686, [ 0.05776939 -0.00161544]
31: 35.33326108575444, [ 0.05922978 -0.00196959]
32: 35.10894896855218, [ 0.06068532 -0.0023225 ]
33: 34.88612908405768, [ 0.06213602 -0.00267418]
34: 34.66479150393542, [ 0.0635819  -0.00302464]
35: 34.44492636591486, [ 0.06502298 -0.00337387]
36: 34.226523873350246, [ 0.06645927 -0.00372189]
37: 34.00957429478419, [ 0.06789079 -0.00406869]
38: 33.794067963513 48, [ 0.06931755 -0.00441428]
39: 33.579995277158865, [ 0.07073957 -0.00475867]
40: 33.36734669723621, [ 0.07215686 -0.00510185]
41: 33.15611274873163, [ 0.07356945 -0.00544384]
42: 32.94628401967916, [ 0.07497734 -0.00578463]
43: 32.73785116074099, [ 0.07638056 -0.00612424]
44: 32.53080488479094, [ 0.07777911 -0.00646266]
45: 32.325135966500 03, [ 0.07917302 -0.0067900 ]
46: 32.12083524192584 6, [ 0.0805623  -0.00713596]
47: 31.917893608103 44, [ 0.08194695 -0.00747085]
48: 31.716302022639727, [ 0.08332701 -0.00780458]
49: 31.516051503310642, [ 0.08470249 -0.00813714]
50: 31.317133127660234, [ 0.08607339 -0.00846853]
51: 31.119538032603334, [ 0.08743974 -0.00879877]
52: 30.92325741403041, [ 0.08880155 -0.00912786]
53: 30.728282526414773, [ 0.09015883 -0.0094558 ]
54: 30.534604682423076, [ 0.09151161 -0.0097826 ]
55: 30.342215252527957, [ 0.09285989 -0.01010826]
56: 30.15110566462312, [ 0.09420369 -0.01043278]
127: 19.380405750648016, [ 0.17903787 -0.03080057]
128: 19.262215807260556, [ 0.18009535 -0.03105284]
129: 19.144812054061887, [ 0.18114932 -0.03130424]
130: 19.028189260675 71, [ 0.18219979 -0.03155476]
131: 18.91234223152692, [ 0.18324676 -0.0318044 ]
132: 18.797265805609833, [ 0.18429026 -0.03205317]
133: 18.682954856258288, [ 0.18533029 -0.03230107]
134: 18.569404290917106, [ 0.18636687 -0.03254811]
135: 18.456609050915 12, [ 0.1874     -0.03279428]
136: 18.344564111239638, [ 0.1884297  -0.03303959]
137: 18.233264480312446, [ 0.18945598 -0.03328405]
138: 18.122705199767 52, [ 0.19047884 -0.03352765]
139: 18.012881344229672, [ 0.19149831 -0.0337704 ]
140: 17.903788021095327, [ 0.1925144 -0.0340123]
141: 17.795420370314414, [ 0.19352711 -0.03425336]
142: 17.687773564173 46, [ 0.19453645 -0.03449357]
143: 17.580842807080707, [ 0.19554244 -0.03473295]
144: 17.474623333532256, [ 0.19654509 -0.03497148]
145: 17.369110416999 85, [ 0.1975444  -0.03520919]
146: 17.264299351519803, [ 0.1985404  -0.0354606]
147: 17.160185469683455, [ 0.19953309 -0.03568211]
148: 17.056764133329 29, [ 0.20052248 -0.03591733]
149: 16.954030735155847, [ 0.20150858 -0.03615173]
150: 16.851980698516 85, [ 0.20249141 -0.03638531]
151: 16.750609477216 79, [ 0.20347097 -0.03661808]
152: 16.649912553085, [ 0.20444728 -0.03685003]
153: 16.549885446891768, [ 0.20542034 -0.03708117]
154: 16.450523695913603, [ 0.20639017 -0.0373115 ]
155: 16.351822875969 28, [ 0.20735678 -0.03754103]
156: 16.253778590105362, [ 0.20832018 -0.03776975]
157: 16.156386470623 61, [ 0.20928038 -0.03799768]
158: 16.059642178886 21, [ 0.21023739 -0.03822481]
159: 15.963541405122639, [ 0.21119121 -0.03845115]
160: 15.868079868237267, [ 0.21214187 -0.03867669]
161: 15.773253315618755, [ 0.21308937 -0.03890145]
162: 15.679057522950565, [ 0.21403373 -0.03912542]
163: 15.585488294022387, [ 0.21497494 -0.0393486 ]
164: 15.492541460543476, [ 0.21591303 -0.03957101]
165: 15.400212881956556, [ 0.216848   -0.03979264]
166: 15.308498445253445, [ 0.21777987 -0.0400135 ]
167: 15.217394064791753, [ 0.21870864 -0.04023358]
168: 15.126895682112659, [ 0.21963433 -0.0404529 ]
169: 15.036999265760134, [ 0.22055694 -0.04067144]
170: 14.947700811101 22, [ 0.22147648 -0.04088923]
171: 14.858996340147447, [ 0.22239297 -0.04110625]
172: 14.770881901377813, [ 0.22330641 -0.04132251]
173: 14.683353569562286, [ 0.22421682 -0.04153802]
174: 14.596407445587168, [ 0.22512421 -0.04175277]
175: 14.510039656281185, [ 0.22602858 -0.04196678]
176: 14.424246352 4428, [ 0.22692995 -0.04218003]
177: 14.339023717668827, [ 0.22782832 -0.04239254]
178: 14.254367950184138, [ 0.22872371 -0.04260431]
179: 14.170275280672287, [ 0.22961612 -0.04281534]
180: 14.086741963107663, [ 0.23050557 -0.04302562]
181: 14.003764276388255, [ 0.23139206 -0.04323518]
182: 13.921338524170022, [ 0.23227561 -0.043444  ]
183: 13.839461034702154, [ 0.23315623 -0.04365209]
254: 9.224829413808063, [ 0.28875019 -0.05670918]
255: 9.174190230592801, [ 0.2894432  -0.05687086]
256: 9.123887855864 97, [ 0.2901339  -0.05703198]
257: 9.073920049147256, [ 0.29082231 -0.05719253]
258: 9.024284584868086, [ 0.29150843 -0.05735252]
259: 8.974979252262479, [ 0.29219227 -0.05751195]
260: 8.926001855273643, [ 0.29287384 -0.05767082]
261: 8.877350212454987, [ 0.29355315 -0.05782914]
262: 8.829022156872 87, [ 0.2942302 -0.0579869]
263: 8.781015536010 38, [ 0.294905   -0.05814411]
264: 8.7333282116709, [ 0.29557755 -0.05830076]
265: 8.685958059883157, [ 0.29624788 -0.05845687]
266: 8.638902970806468, [ 0.29691598 -0.05861244]
267: 8.592160848636805, [ 0.29758185 -0.05876746]
268: 8.545729611513 3, [ 0.29824552 -0.05892193]
269: 8.499607191423 57, [ 0.29890698 -0.05907587]
270: 8.453791534120926, [ 0.29956625 -0.05922926]
271: 8.408280599014455, [ 0.30022332 -0.05938212]
272: 8.363072359096156, [ 0.30087821 -0.05953445]
273: 8.318164800841 89, [ 0.30153093 -0.05968624]
274: 8.273555924123055, [ 0.30218148 -0.05983749]
275: 8.229243742117673, [ 0.30282987 -0.05998822]
276: 8.185226281222197, [ 0.3034761  -0.06013842]
277: 8.141501580962005, [ 0.30412019 -0.0602881 ]
278: 8.098067693906794, [ 0.30476213 -0.06043725]
279: 8.054922685581312, [ 0.30540195 -0.06058588]
280: 8.012064634380147, [ 0.30603964 -0.06073399]
281: 7.969491631482051, [ 0.30667521 -0.06088158]
282: 7.927201780765014, [ 0.30730867 -0.06102865]
283: 7.885193198721391, [ 0.30794003 -0.06117521]
284: 7.843464014374158, [ 0.30856928 -0.06132125]
285: 7.802012369193719 5, [ 0.30919645 -0.06146678]
286: 7.760836417014807, [ 0.30982154 -0.06161181]
287: 7.719934323954 28, [ 0.31044455 -0.06175632]
288: 7.679304268329511, [ 0.31106549 -0.06190033]
289: 7.638944440577725, [ 0.31168436 -0.06204383]
290: 7.598853043172610 5, [ 0.31230118 -0.06218684]
291: 7.559028290549523, [ 0.31291595 -0.06232934]
292: 7.519468409020709, [ 0.31352868 -0.06247134]
293: 7.480171636698927, [ 0.31413938 -0.06261284]
294: 7.441136223418253, [ 0.31474804 -0.06275385]
295: 7.402360430656202, [ 0.31535469 -0.06289436]
296: 7.363842531456 35, [ 0.31595932 -0.06303439]
297: 7.325580810351 315, [ 0.31656194 -0.06317392]
298: 7.287573563285659, [ 0.31716255 -0.06331296]
299: 7.249819097541505, [ 0.31776118 -0.06345152]
300: 7.212315731661635, [ 0.31835781 -0.06358959]
301: 7.175061795375189, [ 0.31895247 -0.06372717]
302: 7.138055629523147, [ 0.31954514 -0.06386428]
303: 7.101295585984366, [ 0.32013585 -0.0640009 ]
304: 7.064780027602233, [ 0.3207246  -0.06413705]
305: 7.028507328111582, [ 0.32131139 -0.06427272]
306: 6.992475872066351, [ 0.32189623 -0.06440791]
307: 6.956684045476 75, [ 0.32247913 -0.06454263]
308: 6.921130282191494, [ 0.3230601  -0.06467688]
309: 6.885812970919 46, [ 0.32363913 -0.06481065]
310: 6.850730548066378 5, [ 0.32421624 -0.06494396]
```

Question D:
The convergence behavior of the SGD changes as the starting point varies; the slope greatly differs between the two behaviors since the data we begin with changes as well, maybe because the variance in starting points encounters different steeper inclines (rather than optimum ones).

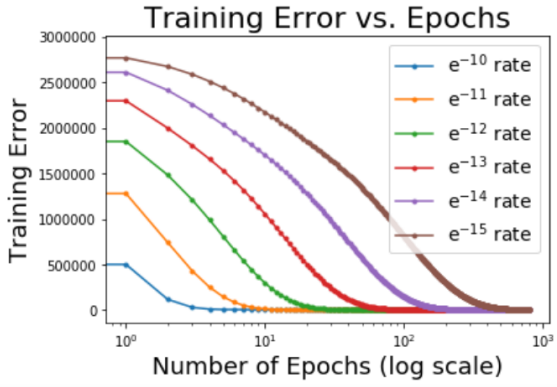
Question E:



Question F:
After running my code, my finals weights were:

[-0.22720591 -5.94229011 3.94369494 -11.72402388 8.78549375]

Question G:



Question H:
Computing this analytical solution gives me: [matrix([[-0.31644251]]), matrix([[-5.99157048]]), matrix([[4.01509955]]), matrix([[-11.93325972]]), matrix([[8.99061096]])].

Which seems close to our previously generated solution of [-0.22720591 -5.94229011 3.94369494 -11.72402388 8.78549375]

Question I:
We would use SGD when a closed form solution exists if we are given some very large input data and using closed form solution would take time and resources. SGD is faster to compute and would be the preferred method since its less computationally complex as data grows in

size.

Question J:
Based on the SGD convergence plots generated earlier, a stopping condition more sophisticated than a pre-defined number of epochs would be finding convergence points for certain etas and stopping around those pre-defined convergence points.

Question K:
The convergence behavior of the weight vector differs between the perceptron and SGD algorithms in that SGD convergence behavior varies significantly based on the size of steps used in the algorithm.

Problem 2

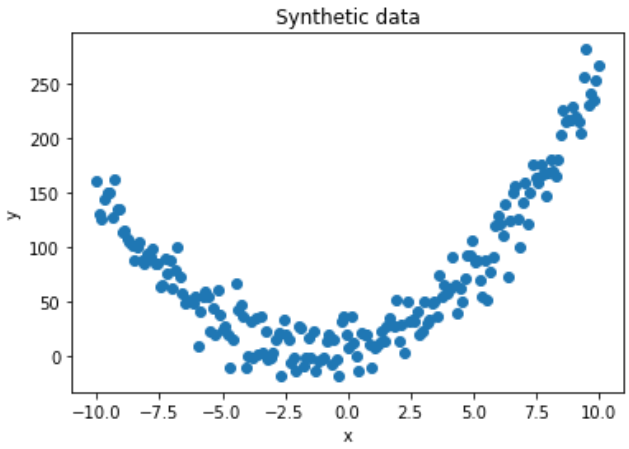
In [3]:
```
# Setup:

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")
```

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.

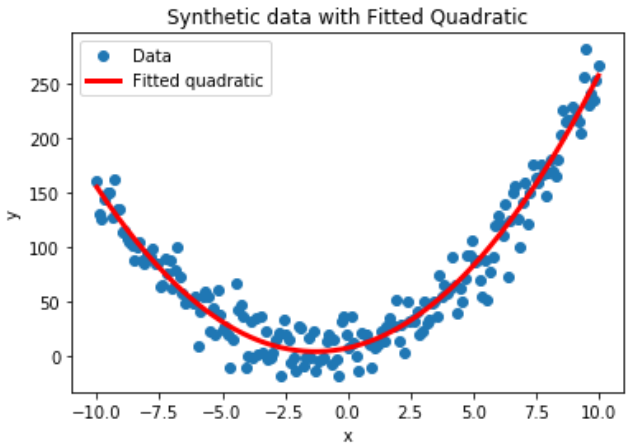
In [4]:
```
# Coefficients of the quadratic function, y(x) = ax^2 + bx + c:
a = 2
b = 5
c = 7

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.

In [5]:
```
# 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) = 1.985406*x^2 + 5.113636*x + 7.60
0382

f(4) = 59.821414
```



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.

In [6]:
```
# 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)

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

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.

In [7]:
```
def load_data(filename):
    """
    Function loads data stored in the file filename and returns it as a numpy ndar
ray.
    Input:
        filename: given as a string.
    Output:
        Data contained in the file, returned as a numpy ndarray
    """
    return np.loadtxt(filename, skiprows=1, delimiter=',')
```

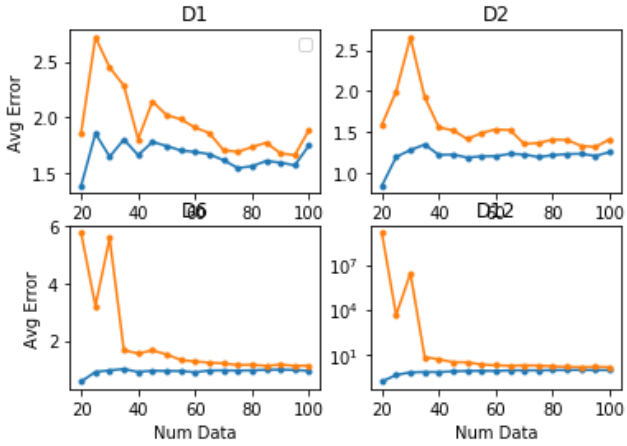
In [8]:
```
data = load_data('data/bv_data.csv')
x = data[:, 0]
y = data[:, 1]
```

Write your code below for solving problem 2 part B:

In [9]:
```
def sqErr(n1, n2):
    # return square error
    return (n1 - n2) ** 2


def getData(data):
    # return arrays representing x and y
    x = []
    y = []

    for i in data:
        x.append(i[0])
        y.append(i[1])

    return np.asarray(x), np.asarray(y)


def retKFoldData(n, numFolds, data):
    # return training data and validation data for one fold
    # fold number = section number that serves as validation number

    foldNumber = int(n)
    part = int(len(data) / numFolds)
    numFolds = int(numFolds)

    lower = (foldNumber - 1) * part
    upper = foldNumber * part

    training = np.concatenate((data[:lower], data[upper:]), axis = 0)
    validation = data[lower : upper]
    return training, validation


def avgErr(data, folds, degreePoly):
    # return avg training & validation errors

    trainingErrs = []
    validationErrs = []

    for k in range(1, folds + 1):
        trainingErr = 0
        validationErr = 0
        training, validation = retKFoldData(k, folds, data)  #Get training and val
idation data from the retKFoldData function

        trainX, trainY = getData(training)
        validationX, validationY = getData(validation)
        z = np.polyfit(trainX, trainY, degreePoly)    #np.polyfit will return coef
ficients

        # use training data to find training error
        for a in range(len(trainX)):
            p1 = np.polyval(z, trainX[a])
            trainingErr += sqErr(p1, trainY[a])

        # use validation data to find validation error
        for b in range(len(validationX)):
            p2 = np.polyval(z, validationX[b])
            validationErr += sqErr(p2, validationY[b])

        trainingErrs.append(trainingErr / len(trainX))
        validationErrs.append(validationErr / len(validationX))

    return sum(trainingErrs) / folds, sum(validationErrs) / folds
```

In [10]:
```
finalData = load_data("data/bv_data.csv")

trainingErr = []
validationErr = []

for d in [1, 2, 6, 12]:
    for size in range(20,101,5):
        t = finalData[:size]
        avgTrain, avgValidation = avgErr(t, 5, d)

        trainingErr.append(avgTrain)
        validationErr.append(avgValidation)

train1 = trainingErr[:17]
valid1 = validationErr[:17]

train2 = trainingErr[17:34]
valid2 = validationErr[17:34]

train6 = trainingErr[34: 51]
valid6 = validationErr[34:51]

train12 = trainingErr[51:]
valid12 = validationErr[51:]


# plotting figures
N = list(range(20,101,5))
x = N

plt.figure(1)
plt.subplot(221)

plt.title('D1')
plt.legend(('Training', 'Validation'))
plt.plot(x, train1, x, valid1, marker = '.')
plt.ylabel('Avg Error')
plt.margins(tight=True)

plt.subplot(222)
plt.title('D2')
plt.plot(x, train2, x, valid2, marker = '.')
plt.margins(tight=True)

plt.subplot(223)
plt.title('D6')
plt.plot(x, train6, x, valid6, marker = '.')
plt.xlabel('Num Data')
plt.ylabel('Avg Error')
plt.margins(tight=True)

plt.subplot(224)
plt.title('D12')
plt.plot(x, train12, x, valid12, marker = '.')
plt.xlabel('Num Data')
# use log scale for fit
plt.yscale('log')
plt.margins(tight=True)

# plt.subplots_adjust(left=.5, bottom=.5, right=.5, top=.5, wspace=.5, hspace=Non
e)
```

Out[10]: (0.05, 0.05)



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Problem 3

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

In [61]:
```
import numpy as np
import matplotlib.pyplot as plt
import itertools
from prettytable import PrettyTable

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

%matplotlib inline
```

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.

In [62]:
```
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 di
mensions.
        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)):
        # if misclassified
        if predict(X[i], w, b) != Y[i]:
            for j in range(len(X[i])):
                # adjust weights for every dimension
                next_w[j] = w[j] + Y[i]*X[i][j]
                next_b = b + Y[i]

    return next_w, next_b
```

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.

In [63]:
```
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 di
mensions.
        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.
    """

    #============================================
    # TODO: Implement perceptron update loop.
    #============================================

    for _ in range(max_iter):
        w, b = update_perceptron(X, Y, w, b)

    return w, b
```

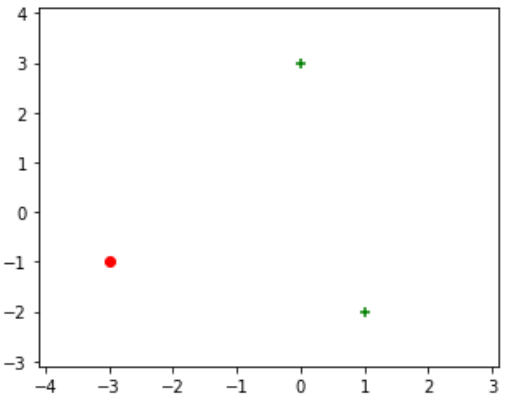
Problem 3A

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.

In [64]:
```
X = np.array([[ -3, -1], [0, 3], [1, -2]])
Y = np.array([ -1, 1, 1])
```

In [65]:
```
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)
```



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.

In [66]:
```
# 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))
```

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

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]
```

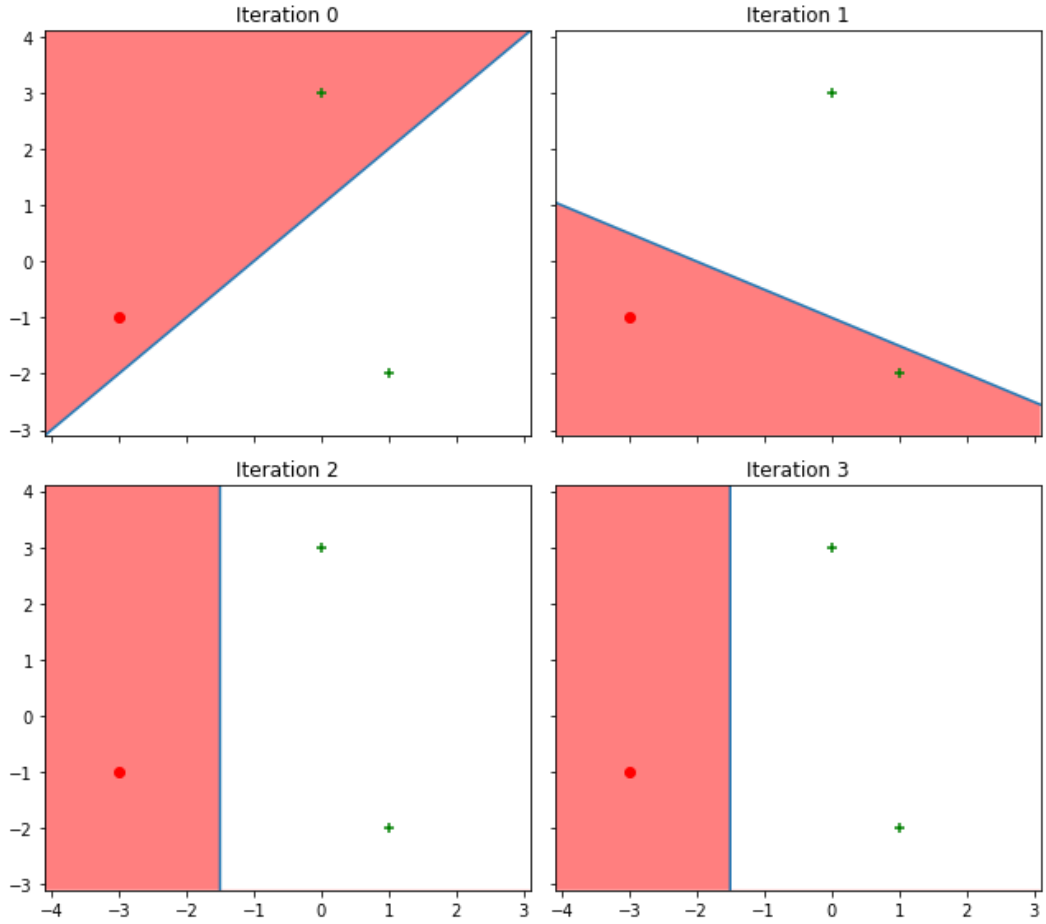
to get the plot correponding to $t = i$. You can then use ax.set_title() to title each subplot. You will want to use the `plot_data()` and `plot_perceptron()` helper methods.

In [78]:
```
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 di
mensions.
        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.
    """

    #============================================
    # TODO: Implement perceptron update loop.
    #============================================
    time = 0
    t = PrettyTable()

    t.field_names = ["t", "b", "w1", "w2"]
    t.add_row([time, b, w[0], w[1]])

    for i in range(max_iter):
        time += 1
        w, b = update_perceptron(X, Y, w, b)
        ax = axs[i]
        title = "Iteration " + str(i)
        ax.set_title(title)
        plot_perceptron(w, b, ax)
        plot_data(X, Y, ax)
        t.add_row([time, b, w[0], w[1]])

    print(t)
    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.

In [79]:
```
# 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 updat
e.
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	b	w1	w2
0	0.0	0.0	1.0
1	1.0	1.0	-1.0
2	2.0	1.0	2.0
3	3.0	2.0	0.0
4	3.0	2.0	0.0



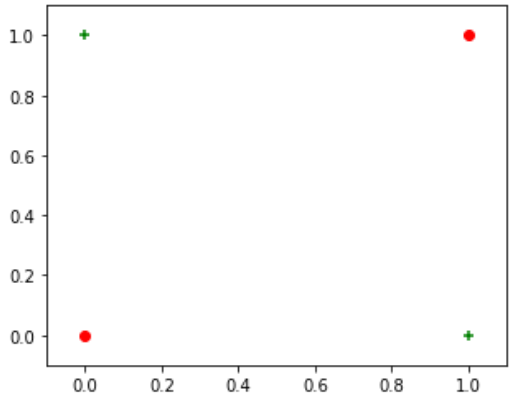
Problem 3C

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.

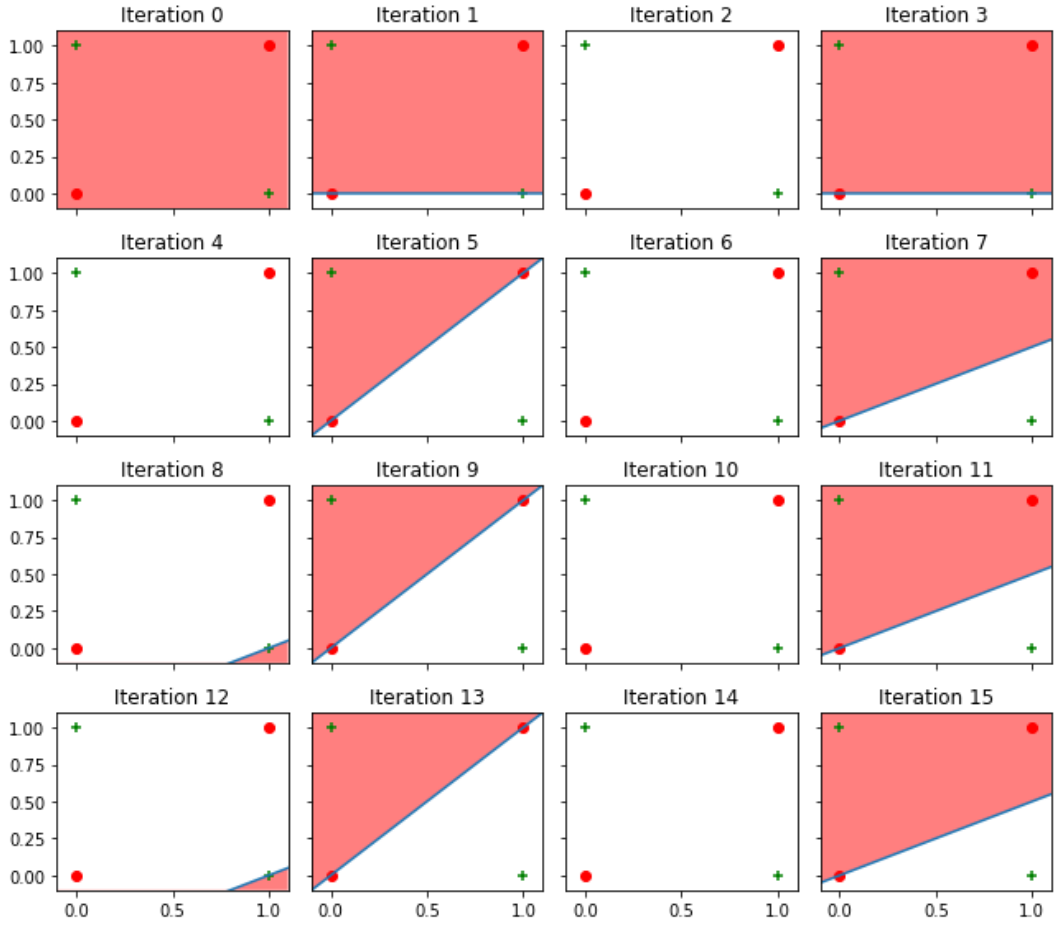
In [33]:
```
X = np.array([[0, 1], [1, 0], [0, 0], [1, 1]])
Y = np.array([1, 1, -1, -1])
```

In [34]:
```
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.

In [35]:
```
# 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()
```



In [ ]:
```
```

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.

In [81]:
```
# Setup.

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

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

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.

In [82]:
```
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 po
ints.
        w: A (D, ) shaped numpy array containing the weight vector.

    Outputs:
        The loss evaluated with respect to X, Y, and w.
    '''
    predict = []
    for x in X:
        predict.append(np.inner(w, x))
    predict = np.asarray(predict)

    loss = 0
    for i in range(len(predict)):
        loss += (predict[i] - Y[i]) ** 2

    return loss

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.
    '''

    #=============================================
    # TODO: Implement the gradient of the loss function.
    #=============================================
    grad = -2 * (y - np.inner(w, x)) * x
    return grad
```

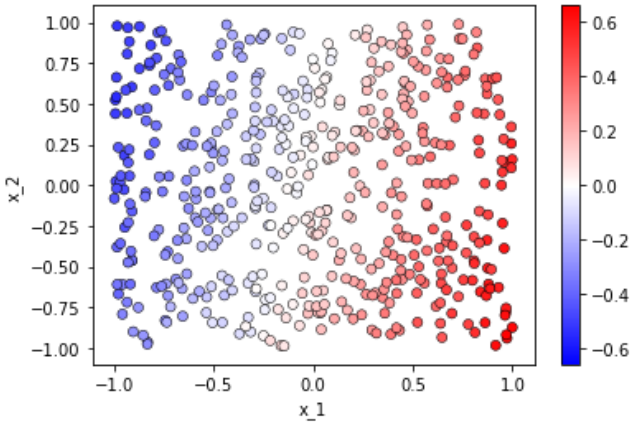
In [ ]:
```
def SGD(X, Y, w_start, eta, N_epochs):
    '''
    Perform SGD using dataset (X, Y), initial weight vector w_start,
    learning rate eta, and N_epochs epochs.

    Outputs:
        w: A (D, ) shaped array containing the final weight vector.
        losses: A (N_epochs, ) shaped array containing the losses from all iterati
ons.
    '''

    #============================================
    # TODO: Implement the SGD algorithm.
    #============================================

    totalLoss = []
    weights = w_start

    #start loss func (for some reason it won't work when i call the function?)
    #Python TypeError: 'list' object is not callable.
    #and
    #TypeError: 'numpy.float64' object is not callable
    predict = []

    for x in X:
        predict.append(np.inner(weights, x))
    predict = np.asarray(predict)

    loss = 0
    for i in range(len(predict)):
        loss += (predict[i] - Y[i]) ** 2
    #end loss func

    #totalLoss.append(loss)

    for n in range(N_epochs):
        for i in range(len(X)):
            g = gradient(X[i],Y[i],weights)
            weights -= eta * g

        #start loss func
        predict = []
        for x in X:
            predict.append(np.inner(weights, x))
        predict = np.asarray(predict)

        loss = 0
        assert(len(predict) == len(Y))
        for i in range(len(predict)):
            loss += (predict[i] - Y[i]) ** 2

        #currLoss = loss(X, Y, weights)
        totalLoss.append(loss)
        #print(str(n) + ": " + str(loss) + ", " + str(weights))

    return np.asarray(weights), np.asarray(totalLoss)
```

Problem 4D: Visualization

Dataset

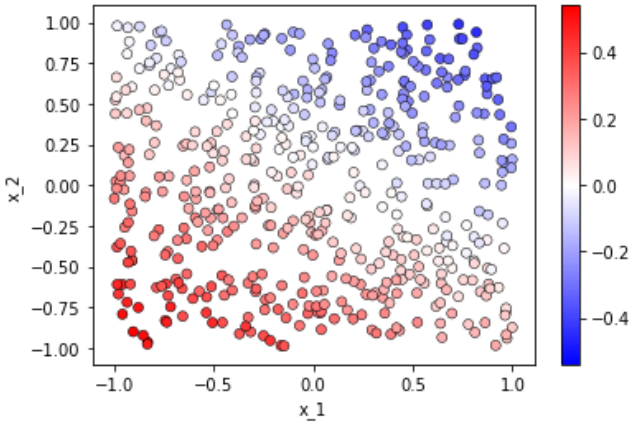
We'll start off by generating two simple 2-dimensional datasets. For simplicity we do not consider separate training and test sets.

In [83]:
```
X1, Y1 = generate_dataset1()
plot_dataset(X1, Y1)
```



Out[83]:
```
(<Figure size 432x288 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x11708fa90>)
```

In [84]:
```
X2, Y2 = generate_dataset2()
plot_dataset(X2, Y2)
```



Out[84]:
```
(<Figure size 432x288 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x1170e0750>)
```

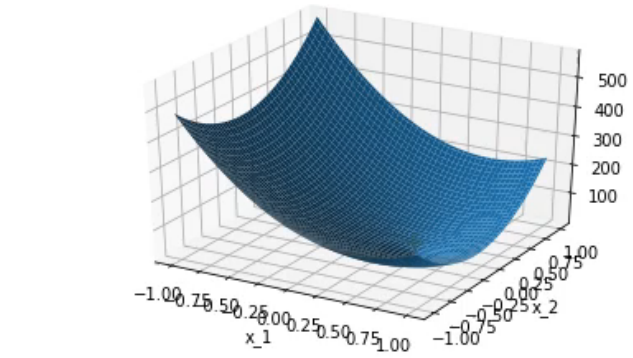
SGD from a single point

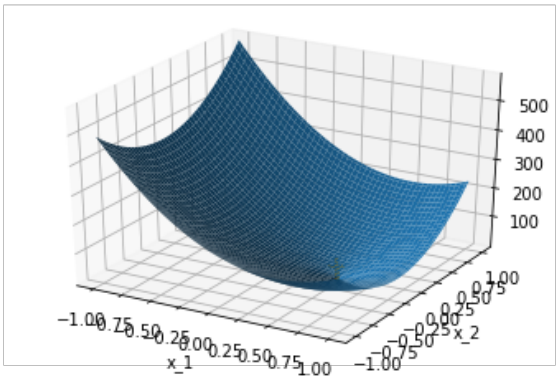
First, let's visualize SGD from a single starting point:

In [85]:
```
# 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...
```

Out[85]:



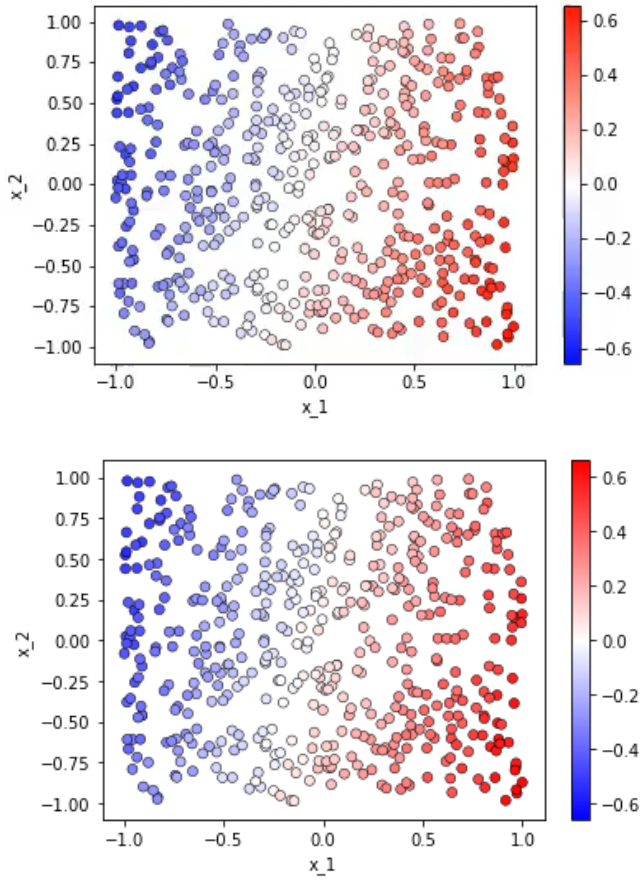


Let's view how the weights change as the algorithm converges:

In [86]:
```
# 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...
```

Out[86]:



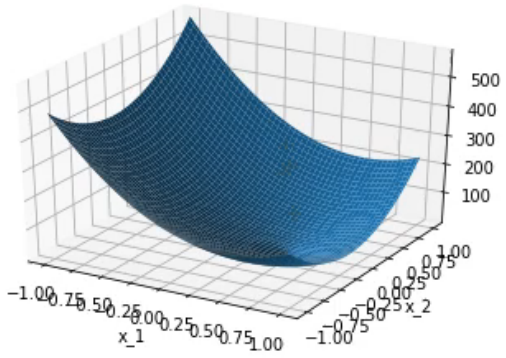
SGD from multiple points

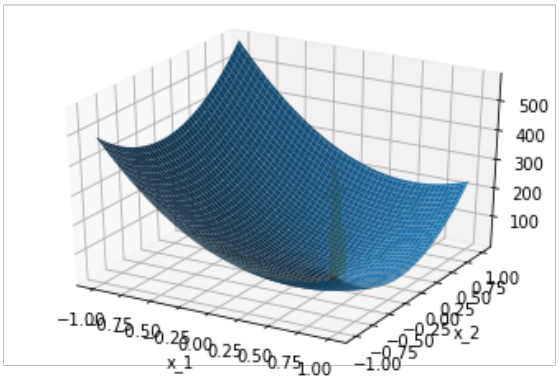
Now, let's visualize SGD from multiple arbitrary starting points:

In [87]:
```
# 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...
```

Out[87]:



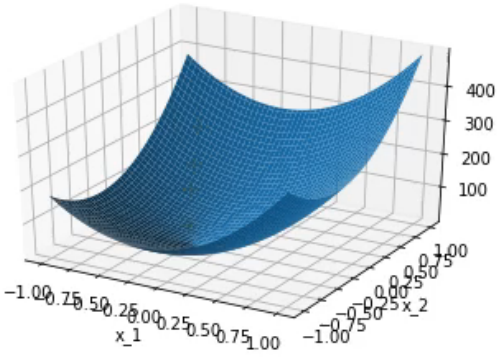


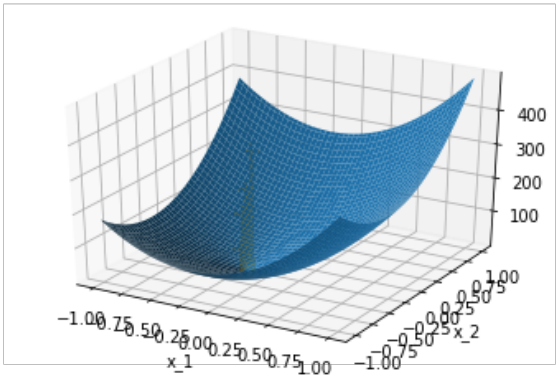
Let's do the same thing but with a different dataset:

In [88]:
```
# 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...
```

Out[88]:





Problem 4E: SGD with different step sizes

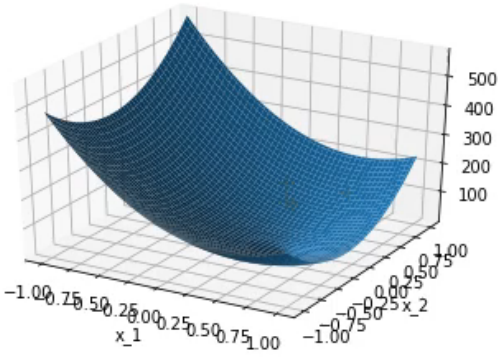
Now, let's visualize SGD with different step sizes (eta):

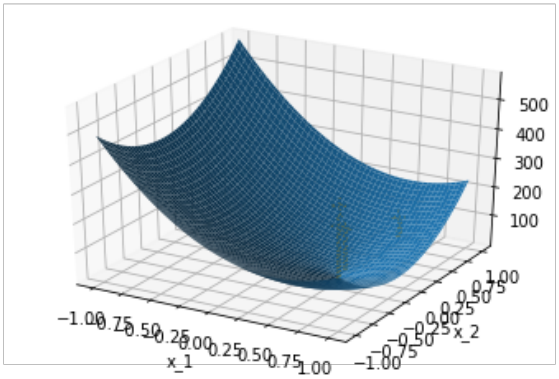
(For ease of visualization: the trajectories are ordered from left to right by increasing eta value. Also, note that we use smaller values of N_epochs and FR here for easier visualization.)

In [89]:
```
# 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...
```

Out[89]:



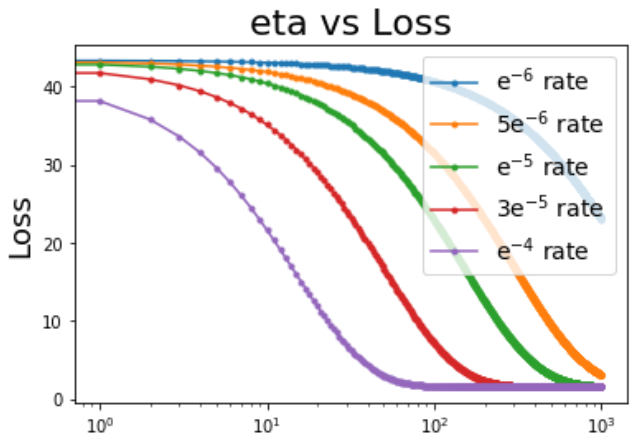


Plotting SGD Convergence

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.

In [90]:
```
'''Plotting SGD convergence'''

#=============================================
# TODO: For the given learning rates, plot the
# loss for each epoch.
#=============================================

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

weights = []
loss = []

for eta in eta_vals:
    initial = [0.001, 0.001, 0.001, 0.001, 0.001]
    finalWeights, allLoss = SGD(X1, Y1, w_start, eta, N_epochs)
    weights.append(finalWeights)
    loss.append(allLoss)

fig = plt.figure()
x = epochs
plt.title('eta vs Loss', fontsize = 22)
plt.plot(loss[0], marker = '.')
plt.plot(loss[1], marker = '.')
plt.plot(loss[2], marker = '.')
plt.plot(loss[3], marker = '.')
plt.plot(loss[4], marker = '.')
plt.legend(('e$^{-6}$ rate', '5e$^{-6}$ rate', 'e$^{-5}$ rate', '3e$^{-5}$ rate',
'e$^{-4}$ rate'), loc = 'upper right', fontsize = 14)
plt.xscale('log')
plt.ylabel('Loss', fontsize = 18)
```

Out[90]:
```
Text(0, 0.5, 'Loss')
```



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.)

In [91]:
```
# 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} ...

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-91-6f8ee3a8be68> in <module>
      5
      6 # Voila!
----> 7 anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)
      8 HTML(anim.to_html5_video())

~/Downloads/CS155_SET1/sgd_helper.py in animate_sgd_suite(SGD, loss, X, Y, param
s, N_epochs, FR, ms)
    121
    122     # Get the loss grid and plot it.
--> 123     w_grid, loss_grid = get_loss_grid((-1, 1, 100), (-1, 1, 100), X, Y,
loss)
    124     fig, ax = plot_loss_function(w_grid[0], w_grid[1], loss_grid)
    125

~/Downloads/CS155_SET1/sgd_helper.py in get_loss_grid(x_params, y_params, X, Y,
loss)
     77         for j in range(len(loss_grid[0])):
     78             w = np.array([w_grid[0][i, j], w_grid[1][i, j]])
---> 79             loss_grid[i, j] = loss(X, Y, w)
     80
     81     return w_grid, loss_grid

TypeError: 'list' object is not callable
```

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

In [ ]:
```
# 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)
```

Extra Visualization (not part of the homework problem)

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

In [ ]:
```
# 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())
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

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.

In [3]:
```
# Setup.
import csv
import numpy as np
import matplotlib.pyplot as plt
import math
import random

%matplotlib inline
```

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.

In [4]:
```
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 po
ints.
        w: A (D, ) shaped numpy array containing the weight vector.

    Outputs:
        The loss evaluated with respect to X, Y, and w.
    '''
    predict = []
    for x in X:
        predict.append(np.inner(w, x))
    predict = np.asarray(predict)

    loss = 0
    for i in range(len(predict)):
        loss += (predict[i] - Y[i]) ** 2

    return loss

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.
    '''

    #=============================================
    # TODO: Implement the gradient of the loss function.
    #=============================================
    grad = -2 * (y - np.inner(w, x)) * x
    return grad
```

In [ ]:
```
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.


    Outputs:
        w: A (D, ) shaped array containing the final weight vector.
        losses: A (N_epochs, ) shaped array containing the losses from all iterati
ons.
    '''

    #=============================================
    # TODO: Implement the SGD algorithm.
    #=============================================

    totalLoss = []
    weights = w_start

    #start loss func (for some reason it won't work when i call the function?)
    #Python TypeError: 'list' object is not callable.
    #and
    #TypeError: 'numpy.float64' object is not callable
    predict = []
    for x in X:
        predict.append(np.inner(weights, x))
    predict = np.asarray(predict)

    loss = 0
    assert(len(predict) == len(Y))
    for i in range(len(predict)):
        loss += (predict[i] - Y[i]) ** 2
    #end loss func

    totalLoss.append(loss)

    for n in range(N_epochs):
        assert(len(X) == 1000)

        for i in range(len(X)):
            g = gradient(X[i],Y[i],weights)
            assert(len(g) == len(weights))
            weights -= eta * g

        #start loss func
        predict = []
        for x in X:
            predict.append(np.inner(weights, x))
        predict = np.asarray(predict)

        loss = 0
        assert(len(predict) == len(Y))
        for i in range(len(predict)):
            loss += (predict[i] - Y[i]) ** 2

        #currLoss = loss(X, Y, weights)
        totalLoss.append(loss)

    return np.asarray(weights), np.asarray(totalLoss)
```

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

In [5]:
```
def load_data(filename):
    """
    Function loads data stored in the file filename and returns it as a numpy ndar
ray.

    Inputs:
        filename: GeneratorExitiven as a string.

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

def getData(data):
    '''
    This function takes the raw data and returns two arrays for inputs(x)
    and outputs(y)
    '''
    x = []
    y = []
    arr = np.asarray([1.0])
    for i in data:
        x.append(np.concatenate((arr, i[:-1]), axis = 0))
        y.append(i[(len(i) - 1)])
    return np.asarray(x), np.asarray(y)
```

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

In [6]:
```
#============================================
# TODO:
# (1) load the dataset
# (2) run SGD using the given parameters
# (3) print out the final weights.
#============================================
weights = []
loss = []
numEpochs = 800
epochs = [800, 900, 1000, 1100, 1200, 1300]
steps = [math.exp(-10), math.exp(-11), math.exp(-12), math.exp(-13), math.exp(-1
4), math.exp(-15)]
stp = math.exp(-15)
allData = load_data("data/sgd_data.csv")
x, y = getData(allData)

initial = [0.001, 0.001, 0.001, 0.001, 0.001]
# X, Y, w_start, eta, N_epochs
fw, _ = SGD(x, y, initial, stp, numEpochs)
print(fw)

for e in epochs:
    for step in steps:
        initial = [0.001, 0.001, 0.001, 0.001, 0.001]
        fw, totalLoss = SGD(x, y, initial, step, numEpochs)
        weights.append(fw)
        loss.append(totalLoss)
```

```
[ -0.22720591  -5.94229011   3.94369494 -11.72402388   8.78549375]
```

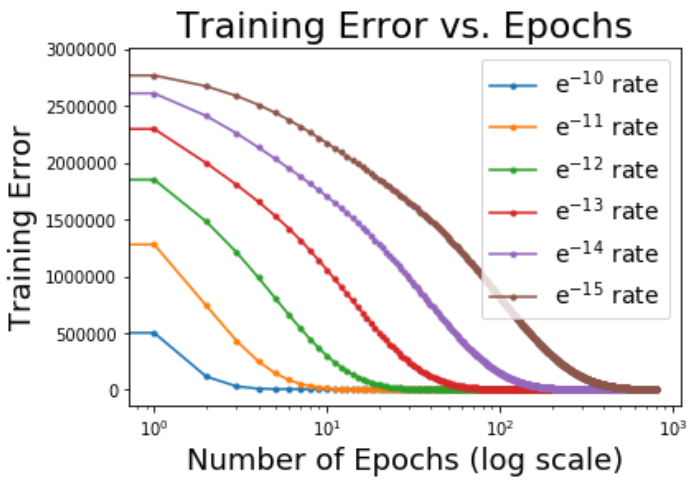
Problem 4G: Convergence of SGD

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

In [10]:
```
#=============================================
# TODO: create a plot showing the convergence
# of SGD for the different learning rates.
#=============================================
print(loss[0])


fig = plt.figure()
x = epochs
plt.title('Training Error vs. Epochs', fontsize = 22)
plt.plot(loss[0], marker = '.')
plt.plot(loss[1], marker = '.')
plt.plot(loss[2], marker = '.')
plt.plot(loss[3], marker = '.')
plt.plot(loss[4], marker = '.')
plt.plot(loss[5], marker = '.')
plt.legend(('e$^{-10}$ rate', 'e$^{-11}$ rate', 'e$^{-12}$ rate', 'e$^{-13}$ rate
', 'e$^{-14}$ rate', 'e$^{-15}$ rate'), loc = 'upper right', fontsize = 14)
plt.xlabel('Number of Epochs (log scale)', fontsize = 18)
plt.xscale('log')
plt.ylabel('Training Error', fontsize = 18)
```

```
[2874307.79361655  500820.74432231  116012.3436524    29495.32628487
    9933.17467149    5483.21974217    4463.39592301    4227.13315194
    4171.40613985    4157.83834506    4154.34642156    4153.36282484
    4153.04781185    4152.93004034    4152.87821425    4152.85132614
    4152.83483885    4152.82299375    4152.81332182    4152.80471177
    4152.79664943    4152.78889313    4152.78132811    4152.77389936
    4152.76658012    4152.75935713    4152.75222348    4152.74517535
    4152.73821038    4152.73132692    4152.72452366    4152.71779947
    4152.71115329    4152.70458412    4152.698091      4152.69167298
    4152.68532912    4152.6790585     4152.67286023    4152.6667334
    4152.66067713    4152.65469055    4152.64877281    4152.64292305
    4152.63714043    4152.63142413    4152.62577334    4152.62018724
    4152.61466504    4152.60920596    4152.60380922    4152.59847405
    4152.59319971    4152.58798544    4152.58283051    4152.57773419
    4152.57269577    4152.56771453    4152.56278978    4152.55792082
    4152.55310699    4152.54834759    4152.54364198    4152.53898949
    4152.53438948    4152.5298413     4152.52534434    4152.52089795
    4152.51650154    4152.51215449    4152.50785621    4152.5036061
    4152.49940357    4152.49524806    4152.491139      4152.48707581
    4152.48305795    4152.47908487    4152.47515603    4152.47127089
    4152.46742893    4152.46362962    4152.45987246    4152.45615693
    4152.45248254    4152.44884879    4152.4452552     4152.44170127
    4152.43818653    4152.43471053    4152.43127278    4152.42787283
    4152.42451024    4152.42118454    4152.41789531    4152.41464211
    4152.41142451    4152.40824207    4152.4050944     4152.40198106
    4152.39890165    4152.39585578    4152.39284303    4152.38986302
    4152.38691535    4152.38399965    4152.38111554    4152.37826264
    4152.37544058    4152.372649      4152.36988754    4152.36715584
    4152.36445355    4152.36178033    4152.35913583    4152.35651971
    4152.35393165    4152.3513713     4152.34883835    4152.34633248
    4152.34385336    4152.34140069    4152.33897415    4152.33657343
    4152.33419825    4152.33184829    4152.32952326    4152.32722288
    4152.32494686    4152.3226949     4152.32046674    4152.31826209
    4152.31608069    4152.31392226    4152.31178653    4152.30967325
    4152.30758215    4152.30551298    4152.30346548    4152.30143941
    4152.29943451    4152.29745054    4152.29548727    4152.29354444
    4152.29162184    4152.28971922    4152.28783635    4152.28597301
    4152.28412898    4152.28230403    4152.28049795    4152.27871052
    4152.27694152    4152.27519075    4152.273458      4152.27174307
    4152.27004574    4152.26836582    4152.26670312    4152.26505743
    4152.26342857    4152.26181634    4152.26022055    4152.25864103
    4152.25707758    4152.25553002    4152.25399818    4152.25248187
    4152.25098093    4152.24949517    4152.24802444    4152.24656855
    4152.24512735    4152.24370067    4152.24228834    4152.24089021
    4152.23950611    4152.2381359     4152.2367794     4152.23543649
    4152.23410699    4152.23279076    4152.23148766    4152.23019754
    4152.22892025    4152.22765565    4152.2264036     4152.22516397
    4152.22393661    4152.22272139    4152.22151818    4152.22032684
    4152.21914724    4152.21797926    4152.21682277    4152.21567763
    4152.21454373    4152.21342094    4152.21230914    4152.21120821
    4152.21011804    4152.20903849    4152.20796947    4152.20691085
    4152.20586252    4152.20482437    4152.20379629    4152.20277817
    4152.2017699     4152.20077137    4152.19978248    4152.19880313
    4152.19783321    4152.19687262    4152.19592126    4152.19497903
    4152.19404583    4152.19312157    4152.19220615    4152.19129947
    4152.19040144    4152.18951197    4152.18863097    4152.18775835
    4152.18689402    4152.18603788    4152.18518986    4152.18434986
    4152.1835178     4152.18269361    4152.18187718    4152.18106845
    4152.18026732    4152.17947373    4152.17868758    4152.17790881
    4152.17713733    4152.17637307    4152.17561595    4152.1748659
    4152.17412284    4152.1733867     4152.1726574     4152.17193488
    4152.17121907    4152.17050989    4152.16980728    4152.16911117
    4152.16842149    4152.16773817    4152.16706115    4152.16639036
    4152.16572574    4152.16506722    4152.16441474    4152.16376825
    4152.16312767    4152.16249294    4152.16186402    4152.16124083
```

Out[10]: Text(0, 0.5, 'Training Error')



Problem 4H

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

In [9]:
```
#=============================================
# TODO: implement the least-squares
# analytical solution.
#=============================================

x, y = getData(allData)
x = np.matrix(x)
y = np.matrix(y)
ret = np.linalg.inv(x.transpose() * x) * x.transpose() * y.transpose()

print(list(ret))
```

```
[matrix([[-0.31644251]]), matrix([[-5.99157048]]), matrix([[4.01509955]]), matri
x([[-11.93325972]]), matrix([[8.99061096]])]
```

In [ ]:
```

```