MASTER wk6 Demo - Supervised Learning & Gradient Descent

`MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | Fall 2022`

In Supervised Machine Learning we use labeled training data to learn a decision function (a.k.a 'model') and make evaluations about how well that decision function might perform when applied to new data. Of course the biggest factor that will determine the performance of your model is the quality of the data you train on. However another key challenge is the question of what models to consider & how to compare their performance so that you can choose the best one. Gradient Descent solves this challenge for a certain class of functions. By the end of this live session you should be able to:

- **... define** the loss function for OLS Regression and its gradient.
- **... explain** the relationship between model space and parameter space.
- **... recognize** convex optimization problems and why they are desirable.
- **... describe** the process of Gradient Descent & how it can be parallelized.

Introduction

In today's demo, we'll use Linear Regression on a simple example in order to explore key topics related to distributed learning of parametric models. Broadly speaking, the supervised learning of a parametric model can be split into to two components:

1. **Optimization Task (a.k.a. Learning)**: Given a strategy for making a prediction, return the specific parameters which guarantee the optimal prediction.
2. **Prediction Task**: Given an input vector, return an output value.

> **DISCUSSION QUESTION:** *In the case of Linear Regression, which of the two tasks above are we most likely to want to parallelize? Why?*

OK, Let's start with a quick review of some notation you will have seen in w207.

Notation Review

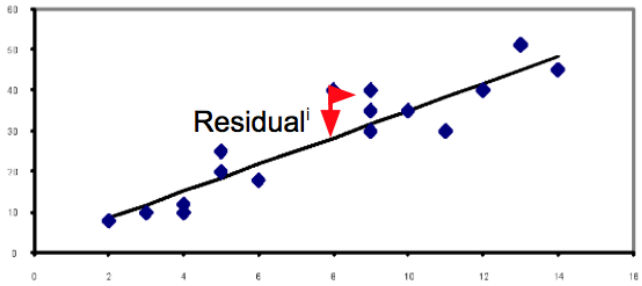
Linear Regression tackles the **prediction task** by assuming that we can compute our output variable, y, using a linear combination of our input variables. That is we assume there exist a set of **weights**, $\mathbf{w}$, and a **bias** term, $\mathbf{b}$, such that for any input $\mathbf{x}_j \in \mathbb{R}^m$:

$$y_j = \sum_{i=1}^{m} w_i \cdot x_{ji} + b \tag{1.1}$$

In vector notation, this can be written:

$$y_j = \mathbf{w}^T \mathbf{x}_j + b \tag{1}$$

Of course, this perfect linear relationship never holds over a whole dataset X, so Linear Regression attempts to fit (i.e. **learn**) the best line (in 1 dimension) or hyperplane (in 2 or more dimensions) to the data. In the case of **ordinary least squares (OLS)** linear regression, best fit is defined as minimizing the Euclidean distances of each point in the dataset to the line or hyperplane. These distances are often referred to as **residuals**.



The calculation of the average residual (a.k.a.**mean squared error, MSE**) over our test or training set allows us to measure how good a fit we've achieved. We call this function the **loss** or **objective** or **cost** function because our goal in the **optimization task** is to find the parameters which minimize it. (Ok, yes, *technically* MSE is *not actually equal* to the average residual but it is conceptually equivalent & guaranteed to have the same minimum.)

$$f(\mathbf{w}, b) = \frac{1}{n} \sum_{j=1}^{n} \left[(\mathbf{w}^T \mathbf{x}_j + b) - y_i \right]^2, \tag{1.2}$$
$$n = |X_{\text{train}}|$$

For convenience, we sometimes choose to think of the bias b as weight w_{m+1}. To operationalize this, we'll *augment* our input vectors by setting $x_{m+1} = 1$. This gives us a simpler way to write the loss function:

$$\mathbf{x}' := \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \quad \boldsymbol{\theta} := \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$$

$$f(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left[\boldsymbol{\theta}^T \cdot \mathbf{x}'_i - y_i \right]^2 \tag{1.3}$$

Machine Learning packages like `sklearn` and `tensorflow` take this one step further by representing the entire training set in a single matrix were each row is an input vector and each column represents a feature:

$$\mathrm{X}' = \begin{bmatrix} \mathbf{x}'^{\mathrm{T}}_1 \\ \vdots \\ \mathbf{x}'^{\mathrm{T}}_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$f(\boldsymbol{\theta}) = \frac{1}{n} \| \mathrm{X}' \cdot \boldsymbol{\theta} - \mathbf{y} \|_2^2 \tag{1.4}$$

As you see here, it is customary to write loss as a function of the parameters θ (or equivalently $\mathbf{w}$ and b). However it is important to note that the MSE loss depends on both the parameters/weights *and* the data X, we'll talk more about that later.

> **DISCUSSION QUESTIONS:**
>
> - *In equation 1.1 what do $x{ji},wi, and|mathbf{w}$ each represent?*
> - *In the asynch's version of the loss function α and β appear as parameters... what do they represent? How are they captured in equations 1.2 and 1.3 respectively?*
> - *If we were computing loss over a really large data set what might be the arguments in favor / against using the augmented version of the loss function calculation?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *In equation 1.1 what do $x{ji},wi, and|mathbf{w}$ each represent?*

  > x_{ij} is the i^{th} variable in the j^{th} data example. w_i is the i^{th} weight (parameter), and $\mathbf{w}$ is the entire weight (parameter) vector.

- *In the asynch's version of the loss function α and β appear as parameters... what do they represent? How are they captured in equations 1.2 and 1.3 respectively?*

  > α represents the weights and β the bias, in equation 1.2 these are w and b, in equation 1.3 we append b to the weights vecotor to get an augmented weight vector θ which is just another way of representing α and β. NOTE -- the greek letter α is also often (and confusingly) used to represent the learning rate in gradient descent - we'll try to use η "eta" instead to avoid confusion, but its something to keep an eye out for.

- *If we were computing loss over a really large data set what might be the arguments in favor / against using the augmented version of the loss function calculation?*

  > Having to augment the entire data set prior to learning adds an additional pass over the data. In addition, it doubles the storage required. Instead, we can "augment" each example as we encounter it.

A warning about OLS before we start:

Supervised learning models, especially interprable ones, and especially linear/logistic regression, tend to get used for two different kinds of tasks: prediction and inference -- it is important to remember the difference between these two use cases. While it is practically possible to fit a linear model to any dataset and then use that model to make predictions... it is *not* always fair to use the coefficients of your model to infer relationships (causal or otherwise) between your features and outcome variable. As you will rememeber from w203 and w207 if you are going to perform inference using OLS, your data should satisfy the following conditions:

1. Residuals are homoscedastic - they have constant variance
2. Residuals are normaly distributed
3. No multicolinearity - features are not correlated

For more info see the reading ISL 3.1.3 ISL Slides

Notebook Set Up

In [3]:
```
# general imports
import sys
import csv
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import re
import ast
import time
import itertools
from IPython.display import Image
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

# magic commands
%matplotlib inline
%reload_ext autoreload
%autoreload 2
```

In [6]:
```
import os
from google.cloud import storage
DATA_BUCKET = os.getenv('DATA_BUCKET','')[:-1] # our private storage bucket location
DEMO06_FOLDER = f"{DATA_BUCKET}/notebooks/jupyter/LiveSessionMaterials/wk06Demo_Optimiza
```

In [4]:
```
# import helper modules
# import helperFunc
# import linRegFunc

# OPTIONAL - uncomment to print helper file docstrings
# print(helperFunc.__doc__)
#print(linRegFunc.__doc__)
```

In [5]:
```
### helperFunc

#!/opt/anaconda/bin/python
"""
This file contains helper functions for generating, transforming
and plotting 2 dimensional data to use in testing & for ML demos.

Avaliable functions include:
    augment(X)
    plot2DModels(data, models=[], names = [], title=None)
    plotErrorSurface(data, weight_grid, loss, title=None)

"""
import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
```

```
from mpl_toolkits.mplot3d import Axes3D

def augment(X):
    """
    Takes an np.array whose rows are data points and augments each
    each row with a 1 in the last position to represent the bias.
    """
    return np.insert(X, -1, 1.0, axis=1)


def plot2DModels(data, models=[], names = [], title=None):
    """
    Plot a set of 2d models for comparison.
    INPUT:  data       - numpy array of points x, y
            model_list - [(label,[w_0, w_1]), ...]
            title      - (optional) figure title
    """
    # create plot
    fig, ax = plt.subplots()
    # plot data
    ax.plot(data[:,0], data[:,1],'o')
    domain = [min(data[:,0]), max(data[:,0])]
    # plot models
    for W,label in zip(models, names):
        m , b = W[0], W[1]
        yvals = [m*x + b for x in domain]
        ax.plot(domain, yvals, linewidth=1, label=label)
    if models:
        plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    # title & formatting
    if title:
        plt.title(title)

def plotErrorSurface(data, weight_grid, loss, title=None):
    """
    Plot a set of 2d models for comparison.
    INPUT:  data    - numpy array of points x, y
            weight_grid  - numpy array of weight vectors [w_0, w_1]
            loss    - list/array of loss corresponding to ^
            title   - (optional) figure title
    """
    # create figure
    fig = plt.figure(figsize=plt.figaspect(0.25))

    # plot data
    ax1 = fig.add_subplot(1, 3, 1)
    ax1.plot(data[:,0], data[:,1],'o')
    ax1.set_title("Linear Models in 2D", fontsize=14)
    plt.xlabel('Input Data')
    plt.ylabel('Output Value')
    domain = [min(data[:,0]), max(data[:,0])]

    # plot models
    for idx, W in enumerate(weight_grid):
        m , b = W[0], W[1]
        yvals = [m*x + b for x in domain]
        ax1.plot(domain, yvals, linewidth=1)

    # plot loss in 3D
    ax2 = fig.add_subplot(1, 3, 2, projection='3d')
    ax2.set_title("Loss as a function of weights.", fontsize=15)
    plt.xlabel('W_0')
```

```
    plt.ylabel('W_1')
    X,Y = weight_grid.T
    ax2.scatter(X,Y,loss, c=loss, cmap=cm.rainbow)


    # plot error surface in 3D
    ax3 = fig.add_subplot(1, 3, 3, projection='3d')
    ax3.set_title("Inferred Error Surface", fontsize=15)
    plt.xlabel('W_0')
    plt.ylabel('W_1')
    X,Y = weight_grid.T
    surf = ax3.plot_trisurf(X,Y,loss, cmap=cm.rainbow,
                            linewidths = 2.0, alpha=0.65)
    fig.colorbar(surf, alpha=0.65, shrink = 0.5)

    # title & formatting
    if title:
        plt.title(title)
```

In [7]:

```
def OLSLoss(X, y, model):
    """
    Computes mean squared error for a linear model.
    INPUT:  X - numpy array (each row = augmented input point)
            y - numpy array of true outputs
            model  - [w_0, w_1] (coefficient & bias)
    """
    N = len(X)
    W = np.array(model)
    return 1/float(N) * sum((W.dot(X.T) - y)**2)

def OLSGradient(X, y, model):
    """
    Computes the gradient of the OLS loss function for
    the provided data & linear model.
    INPUT:  X - numpy array (each row = augmented input point)
            y - numpy array of true outputs
            model  - [w_0, w_1] (coefficient & bias)
    """
    N = len(X)
    W = np.array(model)
    return 2.0/N *(W.dot(X.T) - y).dot(X)

def GDUpdate(X, y, nIter, init_model, learning_rate, verbose = False):
    """
    Performs Gradient Descent Updates for linear models using OLS Loss.
    INPUT:  X - numpy array (each row = augmented input point)
            y - numpy array of true outputs
            nIter - number of updates to perform
            init_model  - [w_0, w_1] starting coefficient & bias
            learning_rate - step size for the update
            verbose - (optional) printout each update
    OUTPUT: models , loss  - two lists
    """
    # keep track of our progress
    models = [init_model]
    loss = [OLSLoss(X,y,init_model)]

    # perform updates
    for idx in range(nIter):
        gradient = OLSGradient(X, y, models[-1])

        update = np.multiply(gradient,learning_rate)
```

```
        new_model = models[-1] - update

        if verbose:
            print(f'Model {idx}: [{models[-1][0]:.2f}, {models[-1][1]:.2f}]')
            print(f'Loss: {loss[-1]}')
            print(f'     >>> gradient: {gradient}')
            print(f'     >>> update: {update}')
        models.append(new_model)
        loss.append(OLSLoss(X,y,new_model))
    if verbose:
            print(f'Model {nIter}: [{models[-1][0]:.2f}, {models[-1][1]:.2f}]')
            print(f'Loss: {loss[-1]}')
    return np.array(models), loss


def SGDUpdate(X, y, nIter, B, init_model, learning_rate, verbose = False):
    """
    WARNING: SGD should be choosing points at random.
             mini-batch should be shuffling the data at each iteration!
             This is an oversimlified implementation without any randomness/shuffling


    Performs Stoichastic Gradient Descent Updates for linear models using OLS Loss.
    INPUT:  X - numpy array (each row = augmented input point)
            y - numpy array of true outputs
            nIter - number of updates to perform
            B - batchsize (integer)
            init_model  - [w_0, w_1] starting coefficient & bias
            learning_rate - step size for the update
            verbose - (optional) printout each update
    OUTPUT: models , loss  - two lists
    """
    # keep track of our progress
    models = [init_model]
    loss = [OLSLoss(X,y,init_model)]

    # perform updates
    n = len(X)
    for idx in range(nIter):
        j = (idx*B)%n # index to start batch
        batch_X, batch_y = X[j:j+B], y[j:j+B]

        gradient = OLSGradient(batch_X, batch_y, models[-1])
        update = np.multiply(gradient,learning_rate)
        new_model = models[-1] - update

        if verbose:
            print(f'Model {idx}: [{models[-1][0]:.2f}, {models[-1][1]:.2f}]')
            print(f'Loss: {loss[-1]}')
            print(f'     >>> gradient: {gradient}')
            print(f'     >>> update: {update}')
        models.append(new_model)
        loss.append(OLSLoss(X,y,new_model))

    if verbose:
            print(f'Model {nIter}: [{models[-1][0]:.2f}, {models[-1][1]:.2f}]')
            print(f'Loss: {loss[-1]}')
    return np.array(models), loss


def plotGDProgress(data, models, loss, loss_fxn = OLSLoss, show_contours = True):
```

```
    """
    Plot a set of 2d models for comparison.
    INPUT:  data    - numpy array of points x, y
            models  - numpy array of weight vectors [w_0, w_1]
            loss    - list/array of loss corresponding to ^
            title   - (optional) figure title
    """
    # Create figure w/ two subplots
    fig = plt.figure(figsize=plt.figaspect(0.35))
    ax1 = plt.subplot(1, 2, 1)
    ax1.grid(color='grey', linestyle='-', linewidth=0.5, alpha=0.5)
    ax2 = plt.subplot(1, 2, 2)
    ax2.grid(color='grey', linestyle='-', linewidth=0.5, alpha=0.5)
    fig.subplots_adjust(wspace=.6)

    ##### Problem Domain Space #####

    # plot data
    ax1.plot(data[:,0], data[:,1],'o')
    ax1.set_title("Problem Domain Space", fontsize=18)
    ax1.set_xlabel('Input Data')
    ax1.set_ylabel('Output Value')
    domain = [min(data[:,0]), max(data[:,0])]

    # plot models
    for idx, W in enumerate(models):
        m , b = W[0], W[1]
        yvals = [m*x + b for x in domain]
        name = 'm%s:%.2f' %(idx, loss[idx])
        ax1.plot(domain, yvals, label=name, linewidth=1)
    ax1.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

    ##### Model Parameter Space #####

    # plot loss for our models
    w0, w1 = models.T
    ax2.plot(w0,w1)
    ax2.plot(w0,w1, 's')
    ax2.set_title("Model Parameter Space", fontsize=18)
    ax2.set_xlabel('W_0 (slope)')
    ax2.set_ylabel('W_1 (intercept)')

    # plot contour lines
    if show_contours:
        # grid parameters -- just a bit larger than models
        w0_min, w0_max= min(w0)*0.9, max(w0)*1.1
        w1_min, w1_max = min(w1)*0.9, max(w1)*1.2
        w0_step = (w0_max - w0_min)/20
        w1_step = (w1_max - w1_min)/20
        # create loss grid for contour plot
        grid_w0, grid_w1 = np.mgrid[w0_min:w0_max:w0_step,
                                    w1_min:w1_max:w1_step]
        grid_loss = [loss_fxn(augment(data)[:,:2], data[:,1], model)
                     for model in zip(grid_w0.flatten(), grid_w1.flatten())]
        grid_loss = np.array(grid_loss).reshape(20,20)
        # plot loss contours
        topo_levels = np.logspace(min(np.log(min(loss)),0.1),
                                  min(np.log(max(loss))/10,20))
        CS = ax2.contour(grid_w0, grid_w1, grid_loss,
                         levels = topo_levels, cmap = 'rainbow',
                         linewidths = 2.0, alpha=0.35)
```

```
def mean_absolute_percentage_error(y_true, y_pred):
    """
    Use of this metric is not recommended because can cause division by zero
    See other regression metrics on sklearn docs:
      http://scikit-learn.org/stable/modules/classes.html#regression-metrics
    Use like any other metric
    >>> y_true = [3, -0.5, 2, 7]; y_pred = [2.5, -0.3, 2, 8]
    >>> mean_absolute_percentage_error(y_true, y_pred)
    Out[]: 24.791666666666668
    """

    return np.mean(np.abs((y_true.ravel() - y_pred.ravel()) / y_true.ravel())) * 100
```

A Small Example

We'll start with a small example of 5 2-D points:

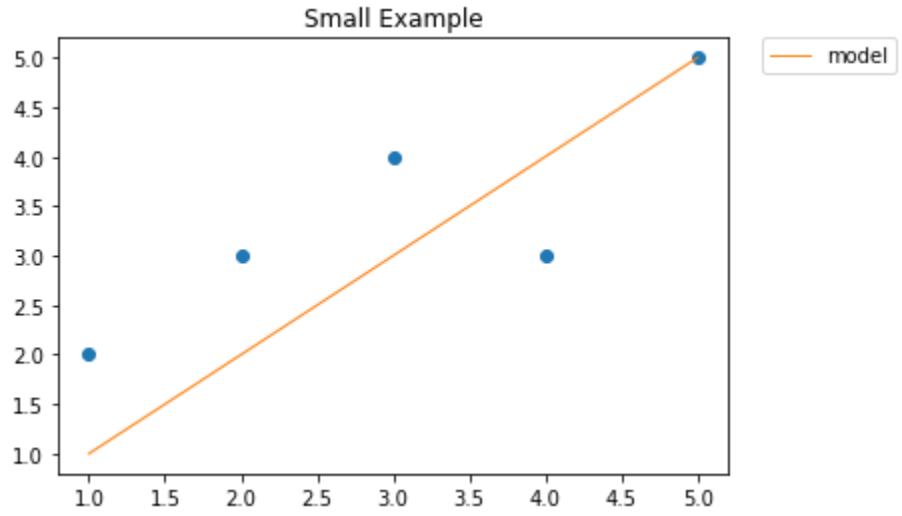
In [8]:
```
# Creates fivePoints.csv and fivePoints
fivePoints="""1,2
3,4
5,5
4,3
2,3"""

fivePoints_loc = f'{DEMO06_FOLDER}/data/fivePoints.csv'

!echo "{fivePoints}" | gsutil cp - {fivePoints_loc}
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

In [9]:
```
# load data from file
from io import StringIO
points = np.genfromtxt(StringIO(fivePoints), delimiter=',')
```

Here's what they look like next to a the simplest possible linear model: $y = x$

In [10]:
```
# easy plotting with a helper function
plot2DModels(points, [[1,0]],['model'], title = 'Small Example')
```



Looks reasonable, but its hard to gauge exactly how good a fit we have just by looking.

> **A TASK FOR YOU:** Fill in the calculations below to compute the "Training Loss" for our data. These are easy and intuitive calculations that you will know from long-ago math classes... but instead of relying on your visual intuition, challenge yourself to think through these numbers in the context of our matrix equation for loss. Here it is again for your reference:
>
> $$f(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left[\boldsymbol{\theta}^T \cdot \mathbf{x}_i' - y_i\right]^2 \tag{1.3}$$

In [21]:
```
!gsutil cat {DEMO06_FOLDER}/data/fivePoints.csv
```

```
1,2
3,4
5,5
4,3
2,3
```

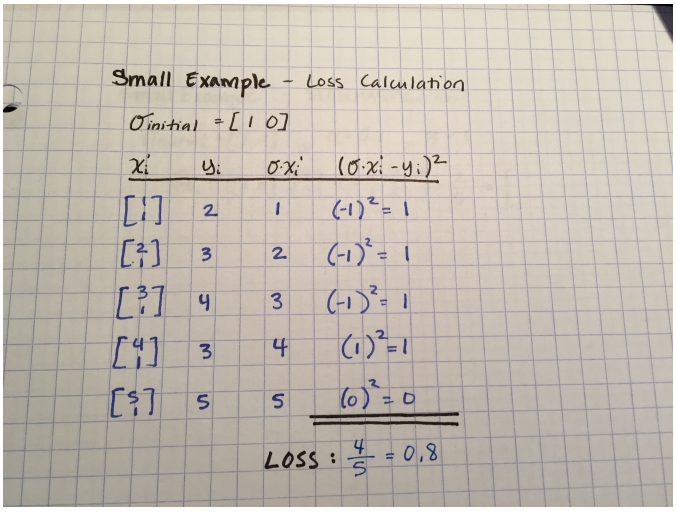
The parameter vector θ for our initial line $y = x$ is: $\begin{bmatrix} ? & ? \end{bmatrix}$

The (augmented) data points x_j are: $\begin{bmatrix} ? \\ ? \end{bmatrix}$ $\begin{bmatrix} ? \\ ? \end{bmatrix}$ $\begin{bmatrix} ? \\ ? \end{bmatrix}$ $\begin{bmatrix} ? \\ ? \end{bmatrix}$ $\begin{bmatrix} ? \\ ? \end{bmatrix}$

Our loss calculations will be:

i	y_i	$\boldsymbol{\theta}^T \cdot \mathbf{x}_i'$	$\left[\boldsymbol{\theta}^T \cdot \mathbf{x}_i' - y_i\right]^2$
-	true y	predicted y	squared residual
1	-	-	-
2	-	-	-
3	-	-	-
4	-	-	-
5	-	-	-

The training loss $f(\boldsymbol{\theta})$ for this data and these weights is: ___



In [11]:
```
# Run this cell to confirm your Hand Calculations
X = augment(points)[:,:-1]
y = points[:,-1]
print("Loss:", OLSLoss(X, y,[1,0]))
```

```
Loss: 0.8
```

> **DISCUSSION QUESTIONS:**
>
> - *What parts of this computation could be parallelized? What, if any, aggregation has to happen at the end?*
> - *What key-value format, partitioning, sorting would help? Could you use a combiner?*
> - *In addition to the data stream, what other information would your map or reduce tasks need access to?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *What parts of this computation could be parallelized? What, if any, aggregation has to happen at the end?*

  > Everything to the right of the summation can be calculated on a per row basis. The aggregation (summation) as well as the final division has to be dne at the end.

- *What key-value format, partitioning, sorting would help? Could you use a combiner?*

  > No key or special partitioning is needed as there is no grouping required. A combiner can be used to sum local results. Remember to pass the number of rows in the payload to be able to take the mean at the end.

- *In addition to the data stream, what other information would your map or reduce tasks need access to?*

  > number of rows. see above.

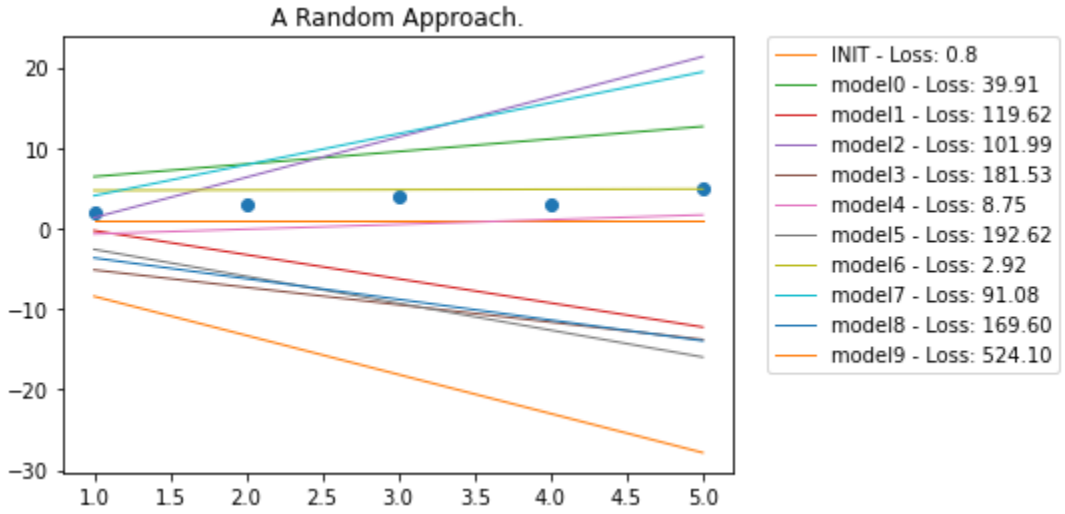
Demo: Random Parameter Search.

Ok, so we know the model looks ok and we know its loss is 0.8 but is that any good? A naive approach to "learning" a Linear Model might be to randomly generate a few more models and then pick the model with the lowest loss. Let's try it.

In [12]:
```
#################### Demo Parameters ####################
# TRY CHANGING THESE & SEE HOW IT AFFECTS OUR SEARCH
NUM_MODELS = 10
PARAM_RANGE = [-5,5]

#################### Random Search Demo ####################
# Load & pre-process data
X = augment(points)[:,:2]
y = points[:,1]
```

```
# "Training"
models = [[0,1]]
names = ["INIT - Loss: 0.8"]
best = {'loss':0.8, 'W': [1,0]}
for idx in range(NUM_MODELS):
    # initialize a random weight vector w/ values in specified range
    W = np.random.uniform(PARAM_RANGE[0],PARAM_RANGE[1], size=(2))
    # compute loss & store for plotting
    loss = OLSLoss(X, y, W)
    models.append(W)
    names.append("model%s - Loss: %.2f" % (idx, loss))
    # track best model
    if loss < best['loss']:
        best['loss'] = loss
        best['W'] = W

# Display Results
print(f"Best Random Model: {best['W']}, Loss: {best['loss']}")
plot2DModels(points, models, names, "A Random Approach.")
```

```
Best Random Model: [1, 0], Loss: 0.8
```



So, that was pretty poor. One idea would be to run a lot more iterations.

> **DISCUSSION QUESTION:**
>
> - *To what extent could parallelization help us redeem this approach? What exactly would you parallelize?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *To what extent could parallelization help us redeem this approach? What exactly would you parallelize?*

  > While parallelization could help us train a lot more models in the same amount of time, we'd have no real guarantee that we'd get better results in exchange for our efforts because of the 'randomness' of what models we try.

Demo: Systematic Brute Force.

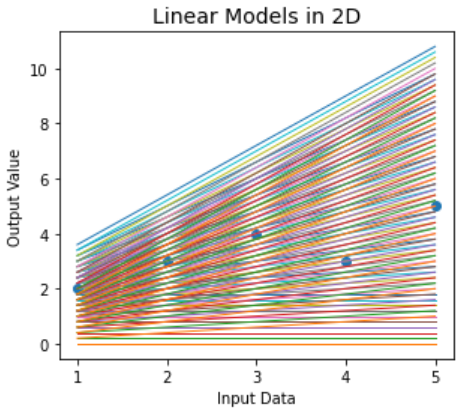
For obvious reasons a more systematic approach is desirable. Instead of randomly guessing, let's use what we know to search an appropriate section of the the model space.

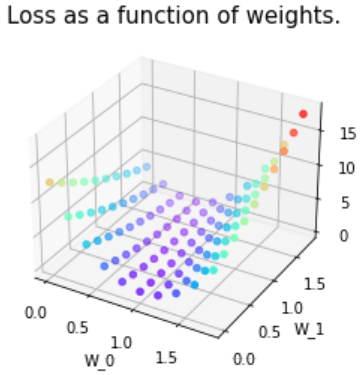
We can tell from the data that the linear model should probably have a fairly shallow positive slope and a positive intercept between 0 and 2. So lets initialize every possible combination of weights in that range up to a granularity of, say 0.2, and compute the loss for each one.

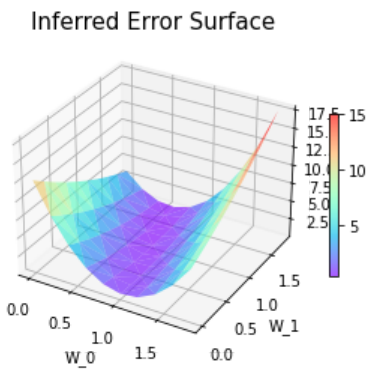
In [13]:
```
#################### Demo Parameters ####################
# TRY CHANGING THESE & SEE HOW IT AFFECTS OUR SEARCH
W0_MIN = 0
W0_MAX = 2
W0_STEP = 0.2

W1_MIN = 0
W1_MAX = 2
W1_STEP = 0.2

#################### Grid Search Demo ####################
### Load & Pre-process Data
X = augment(points)[:,:2]
y = points[:,1]

### "Training"
# create a model for each point in our grid
grid = np.mgrid[W0_MIN:W0_MAX:W0_STEP,W1_MIN:W1_MAX:W1_STEP]
size = int(np.product(grid.shape)/2)
models = grid.reshape(2,size).T
# compute loss for each model
loss = []
for W in models:
    loss.append(OLSLoss(X,y,W))

### Display Results
print(f"Searched {size} models...")
print(f"Best model: {models[np.argmin(loss)]}, Loss: {min(loss)}")
plotErrorSurface(points,models,loss)
```

```
Searched 100 models...
Best model: [0.6 1.6], Loss: 0.31999999999999984
```

```
/tmp/ipykernel_15314/1485538972.py:92: MatplotlibDeprecationWarning: The 'alpha' paramet
er to Colorbar has no effect because it is overridden by the mappable; it is deprecated
since 3.3 and will be removed two minor releases later.
  fig.colorbar(surf, alpha=0.65, shrink = 0.5)
```







> **DISCUSSION QUESTIONS:**
>
> - *When we think about scaling up, is this still a better approach than guessing? How could it be parallelized?*
> - *What would change about this approach if we had higher dimension data?*
> - *In practice, when we're training Linear Models why don't we just look at the error surface and identify the lowest point?*
> - *What about if we're training other kinds of models?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *When we think about scaling up, is this still a better approach than guessing? How could it be parallelized?*

  > Yes, we can at least methodically and incrementally improve the solution. The same parralellization methods still apply.

- *What would change about this approach if we had higher dimension data?*

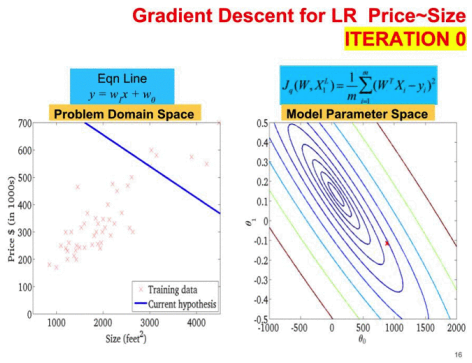
  > With more and more dimensions there would be an exponential number of models to search even a small grid.

- *In practice, when we're training Linear Models why don't we just look at the error surface and identify the lowest point?*

  > We do not have access to the error surface until we've computed the loss for every possible comnination of parameters. Doing so is computationally challenging. There are better methods at arriving at the optimal solution.

Parameter Space, Gradients, and Convexity

As suggested by the systematic search demo, when we train parametric models we tend to switch back and forth between two different ways of visualizing our goal.



- When we look at a model next to our data represented in the Problem Domain Space, it is natural to think about loss as a measure of ***how far off the data are from our model***. In other words, this visual suggests loss is a function of the training data X.
- By contrast, looking at an error surface plotted in Model Parameter Space, we intuitively see loss as an indicator of ***how far off the current model is from the optimal model***. In other words, this view helps us think of loss as a function of the parameters θ.

Of course in one sense, this distinction is just a matter of sematics. As we saw in equations 1.2, 1.3 and 1.4, MSE loss depends on *both* the data and the parameters. However, in the context of 'inventing' ways to train a model, this distinction is a useful one. If we think of the data as fixed and focus on how loss varies *with respect to the parameters*, then we can take advantage of a little theory to speed up our search for the optimal parameters.

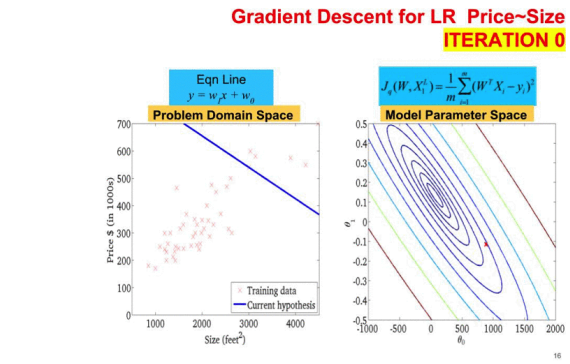
Optimization Theory ... a short digression

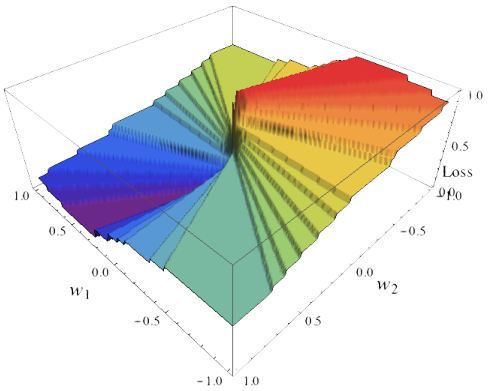
Calculus gives us the simple solution to optimizing a real function. The **First Order Conditions** (a.k.a. 'first derivative rule') says that the maximum or minimum of an unconstrained function must occur at a point where the first derivative = 0. In higher dimensions we extend this rule to talk about a **gradient** vector of partial derivatives which all must equal 0.

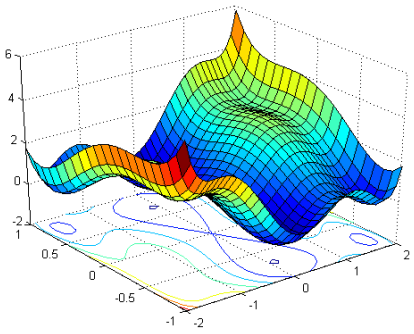
When the first order partial derivatives are equal to zero, then we know we are at a local maximum or minimum of the real function. But which one is it? In order to tell, we must take the second derivatives of the real function. If the second derivatives are positive at that point, then we know we are at a minimum. If the second derivatives are negative, then we know we are at a maximum. These are the **second order conditions.**

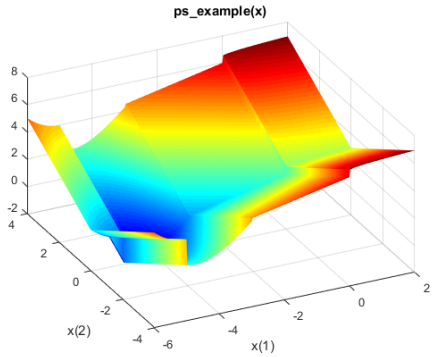
Convex Optimization is the lucky case where we know that the second derivatives never change sign. There are lots of complicated loss functions for which we can't easily visualize the error surface but for which we *can* prove mathematically that this 2nd order condition is met. If this is the case, then we can think of the suface as *always curving up* or *always curving down* which guarantees that any minimum we reach will be an absolute minimum. More powerfully still, this result can be shown to *also* apply to a class of "pseudo-convex" functions - functions whose second derivative might not be well defined, but satisfy certain conditions that allow us to guarantee convergence.

> **DSICUSSION QUESTIONS:**
>
> - *In the case of Linear Regression performed on data $X \in \mathbb{R}^m$, how many dimensions does the gradient vector have? What do each of the values in this vector represent visually?*
> - *If we are systematically searching the parameter space for a lowest point, why might it be useful to know that our loss function is convex?*
> - *In general (i.e. beyond Linear Regression) if finding the ideal parameters θ, is as simple as solving the equation $f'(\theta) = 0$, why don't we always train our models by solving that equation?*
> - *Condider the loss curves illustrated below -- do these illustrations represent problem space or parameter space? which ones are convex?*









Sources: first image | second image | third image

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *In the case of Linear Regression performed on data $X \in \mathbb{R}^m$, how many dimensions does the gradient vector have? What do each of the values in this vector represent visually?*

  > There are m dimentions each of which can be thought of as an axis - it is difficult to visualize when $m > 3$

- *If we are systematically searching the parameter space for a lowest point, why might it be useful to know that our loss function is convex?*

  > If our loss function was not convex, we might get stuck in a local minimum before finding the optimal solution.

- *In general (i.e. beyond Linear Regression) if finding the ideal parameters θ, is as simple as solving the equation $f'(\theta) = 0$, why don't we always train our models by solving that equation?*

> Not all functioins are diferentiable. In addition, it becomes computationaly difficult in high dimensional spaces - specifically, it is difficult to invert large matrices.

Demo: Gradient Descent

To take advantage of these lessons from Optimization Theory, we'll start by taking the derivative of the loss function with respect to the parameters $\boldsymbol{\theta}$. Recall the matrix formulation of our loss function:

$$f(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left[\boldsymbol{\theta}^T \cdot \mathbf{x}_i' - y_i \right]^2 \tag{1.3}$$

We can apply the sum and chain derivation rules to compute the gradient:

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \frac{2}{n} \sum_{i=1}^{n} \left[\boldsymbol{\theta}^T \cdot \mathbf{x}_i' - y_i \right] \cdot \mathbf{x}_i' \tag{3.1}$$

We *could* now set this equation equal to 0 and then solve for $\boldsymbol{\theta}$... but it turns out that this **closed form solution** can be computationally challenging in higher dimensions. It also turns out that a simple approximation technique will work almost as well.

The strategy of **Gradient Descent** is to start somewhere random in the Model Parameter Space and then move down the error surface to find a minimum point with the optimal parameters for our training data. Its ingeniousness is that we can do this without actually knowing the full shape of the error surface. Think of it like walking down a hill while blindfolded. You test each direction to see which way is down, then take a little step in that direction and repeat the process until you can't feel any more 'down' to go. The 'size' of our steps is controled by a hyperparameter, α, the **learning rate**. The whole process can be summarized in 3 steps:

1. Initialize the parameters θ.
2. Compute the gradient $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$.
3. Update the parameters: $\theta_{\text{new}} = \theta_{\text{old}} - \alpha \cdot \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$

We repeat these steps until we reach a stopping criteria.

> **A TASK FOR YOU:** Compute one Gradent Descent update step for the small example from Part 2. Recall that our initial parameters were:
>
> $$\boldsymbol{\theta} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$
>
> For your convenience the augmented input data vectors are already entered in the table below:

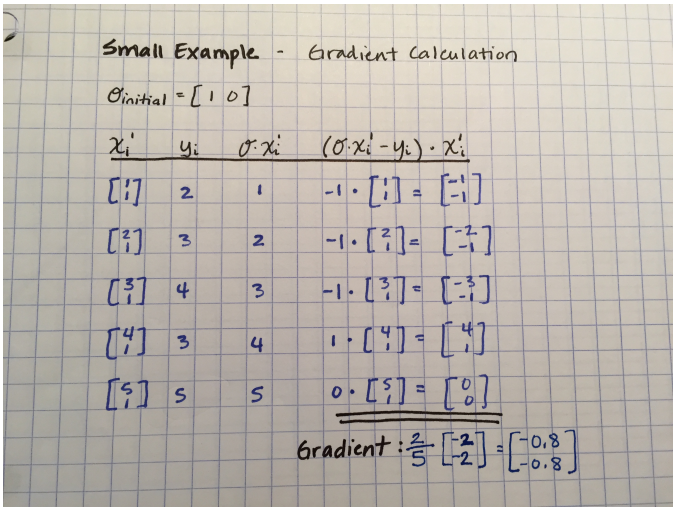
Hand Calculations:

x'_j	y_j	$\boldsymbol{\theta}^T \cdot \mathbf{x}'_j$	$\left[\boldsymbol{\theta}^T \cdot \mathbf{x}'_j - y_j\right] \cdot \mathbf{x}'_j$
input	true y	predicted y	gradient component for x_j
$\begin{bmatrix}1\\1\end{bmatrix}$	2		
$\begin{bmatrix}2\\1\end{bmatrix}$	3		
$\begin{bmatrix}3\\1\end{bmatrix}$	4		
$\begin{bmatrix}4\\1\end{bmatrix}$	3		
$\begin{bmatrix}5\\1\end{bmatrix}$	5		

The gradient $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ for this data and these weights is: [-0.8, -0.8]

If $\alpha = 0.1$ the update for this step will be: [-0.08 -0.08]

The new parameters will be $\theta_{\text{new}} =$ [1.08, 0.08]



> **DISCUSSION QUESTIONS:**
>
> - *How would you go about parallelizing this calculation? What would the mapper do, what would the reducers do? What key-value structure, sorting, partitioning, etc would you use?*
> - *How do the computational demands of performing GD compare to the task of computing the loss?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *How would you go about parallelizing this calculation? What would the mapper do, what would the reducers do? What key-value structure, sorting, partitioning, etc would you use?*

  > For each row of data the mappers would output the result of the equation to the right of the summand. Since we are not grouping anything, no key is needed, and no particular partitioning scheme is necessary. If using combiners, one would need to ensure that the number of rows is passed along in the payload to the reducers. The reducers would sum the incomning values, and a final reducer would compute the mean.

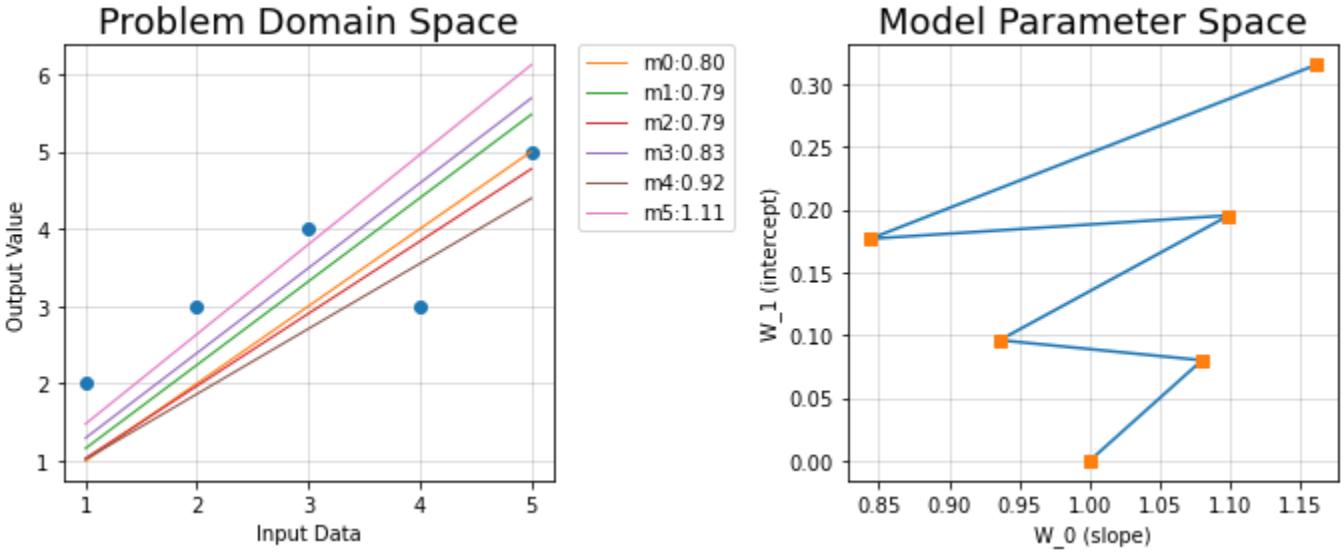
- *How do the computational demands of performing GD compare to the task of computing the loss?*

  > Computing the loss is a component of computing the gradient. In term of algorithm complexity they are almost the same.

Run this demo to confirm your hand calculations & examine a few more GD steps.

In [14]:
```
#################### Demo Parameters ####################
# TRY CHANGING THESE & SEE HOW IT AFFECTS OUR SEARCH
N_STEPS = 5
LEARNING_RATE = 0.1
ORIGINAL_MODEL = [1,0]
SHOW_CONTOURS = False

################### Gradient Update Demo ####################
### Load & Pre-process Data
X = augment(points)[:,:2]
y = points[:,1]

### Perform GD Update & save intermediate model performance
models, loss = GDUpdate(X, y, N_STEPS, ORIGINAL_MODEL, LEARNING_RATE, verbose = True)

### Display Results
print(f"\nSearched {len(models)} models...")
print(f"Best model: {models[np.argmin(loss)]}, Loss: {loss[np.argmin(loss)]}")
plotGDProgress(points, models, loss, show_contours = SHOW_CONTOURS)
```

```
Model 0: [1.00, 0.00]
Loss: 0.8
     >>> gradient: [-0.8 -0.8]
     >>> update: [-0.08 -0.08]
Model 1: [1.08, 0.08]
Loss: 0.7872
     >>> gradient: [ 1.44 -0.16]
     >>> update: [ 0.144 -0.016]
Model 2: [0.94, 0.10]
Loss: 0.7918080000000005
     >>> gradient: [-1.632 -0.992]
     >>> update: [-0.1632 -0.0992]
Model 3: [1.10, 0.20]
Loss: 0.82701312
     >>> gradient: [2.5536 0.1856]
     >>> update: [0.25536 0.01856]
Model 4: [0.84, 0.18]
Loss: 0.9175584768000005
     >>> gradient: [-3.17568 -1.38368]
     >>> update: [-0.317568 -0.138368]
Model 5: [1.16, 0.32]
Loss: 1.1097440747520002

Searched 6 models...
Best model: [1.08 0.08], Loss: 0.7872
```



> **DISCUSSION QUESTIONS:**
>
> - *Look closely at the loss for each model, what problem do you notice?*
> - *Use the Model Parameter Space view to explain why this problem might be occurring.* **HINT:** Try `SHOW_CONTOURS = True`. *Based upon your insights, propose a solution to this problem.*
> - *When performing GD 'in the wild' will we be able to visualize the error surface (eg. using contour lines, heatmaps or 3D plots)?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *Look closely at the loss for each model, what problem do you notice?*

  > The loss is growing

- *Use the Model Parameter Space view to explain why this problem might be occurring.* **HINT:** Try `SHOW_CONTOURS = True`. *Based upon your insights, propose a solution to this problem.*

  > We could reduce the learning rate to prevent the parameter updates from being too large. We may also consider increasing the number of iterations.

- *When performing GD 'in the wild' will we be able to visualize the error surface (eg. using contour lines, heatmaps or 3D plots)?*

  > We would not be able to visualize anything beyond 3 dimensions.

Demo : Stochastic Gradient Descent

In full Gradient Descent (what we did above) we do a descent step only after the calculation of the gradient over the whole set of data. That means we only update the weight vector once each **epoch** (pass over the data) thus making one small but "good" step towards the minimum. However since gradient descent is an interative algorithm that requires many updates to find the minimum, with large datasets, waiting to process every record before performing an update can result in a slow and computationaly costly training process.

The alternatives are:

1. **Stochastic GD** -- compute the gradient *with respect to a single point at a time* and update the entire weight vector after each record. By the time we have seen the whole data set, we will have made N (num of observations), perhaps "not so good", steps with a general trend towards the minimum. SGD will "zig-zag" towards the minimum and eventually oscillate around the minimum but never converge. The advantage of SGD is that we can make progress at every example - if the data is very large, we may only need 1 pass over the whole dataset.
2. **Mini-batch GD** -- compute the gradient *with respect to a small **batch** (size of B) of points at a time* and update the entire weight vector after each batch. If we are smart about shuffling the data, this can reduce the "zig-zaging" because the points in a batch will temper each other's influence. This is especially advantageous for noisy data where a single point might result in a gradient update that is dramatically in the wrong direction for the rest of the data. For this reason, MBGD can potentially finish even faster than SGD.

Other than the denominator in front, the loss function for SGD/MBGD should look very familiar (note that SGD is basically just the special case where $B = 1$):

$$\nabla f(\boldsymbol{\theta}) \approx \nabla_{\text{batch}}\ f(\boldsymbol{\theta}) = \frac{2}{B} \sum_{i=1}^{B} \left(\boldsymbol{\theta}^T \cdot \mathbf{x}'_{a_i} - y_{a_i}\right) \cdot \mathbf{x}'_{a_i} \tag{3.2}$$

where a_i is an array of indices of objects which are in this batch. After obtaining this gradient we do a descent step in this approximate direction and proceed to the next stage of batch descent.

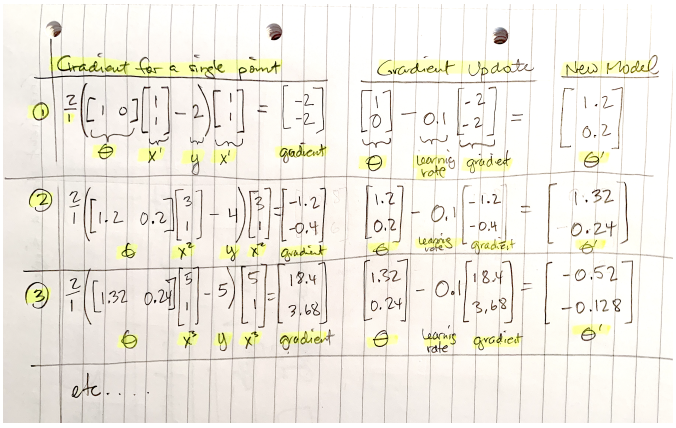
> **A TASK FOR YOU:** Perform 5 update steps of Stochastic Gradient Descent with batchsize = 1 on our small data set. Recall that our initial parameters were:
>
> $$\boldsymbol{\theta} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$
>
> ... and we used a learning rate of $\boldsymbol{\eta} = 0.1$

(η is pronounced 'eh-ta', sometimes we also use α, "apha" to denote learning rate, the two are equivalent)

Hand Calculations:

x'_j	y_j	$\boldsymbol{\theta} \cdot \mathbf{x}'_j$	$\frac{2}{B}\left[\boldsymbol{\theta}^T \cdot \mathbf{x}'_j - y_j\right] \cdot \mathbf{x}'_j$	$\eta \nabla_{\boldsymbol{\theta}} f$	$(\theta) - \eta \nabla_{\boldsymbol{\theta}} f$
input	true y	predicted y	gradient for this 'batch'	update	new parameters
$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$	2				
$\begin{bmatrix} 3 \\ 1 \end{bmatrix}$	4				
$\begin{bmatrix} 5 \\ 1 \end{bmatrix}$	5				
$\begin{bmatrix} 4 \\ 1 \end{bmatrix}$	3				
$\begin{bmatrix} 2 \\ 1 \end{bmatrix}$	3				

> **DISCUSSION QUESTIONS:**
>
> - *How does this result compare to our result from the hand calculations in the last section? What implications does this have for our quest to find the optimal paramters?*
> - *How will parallelizing Stoichastic Gradient Descent be similar/different to parallelizing regular GD?*



<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *How does this result compare to our result from the hand calculations in the last section? What implications does this have for our quest to find the optimal paramters?*

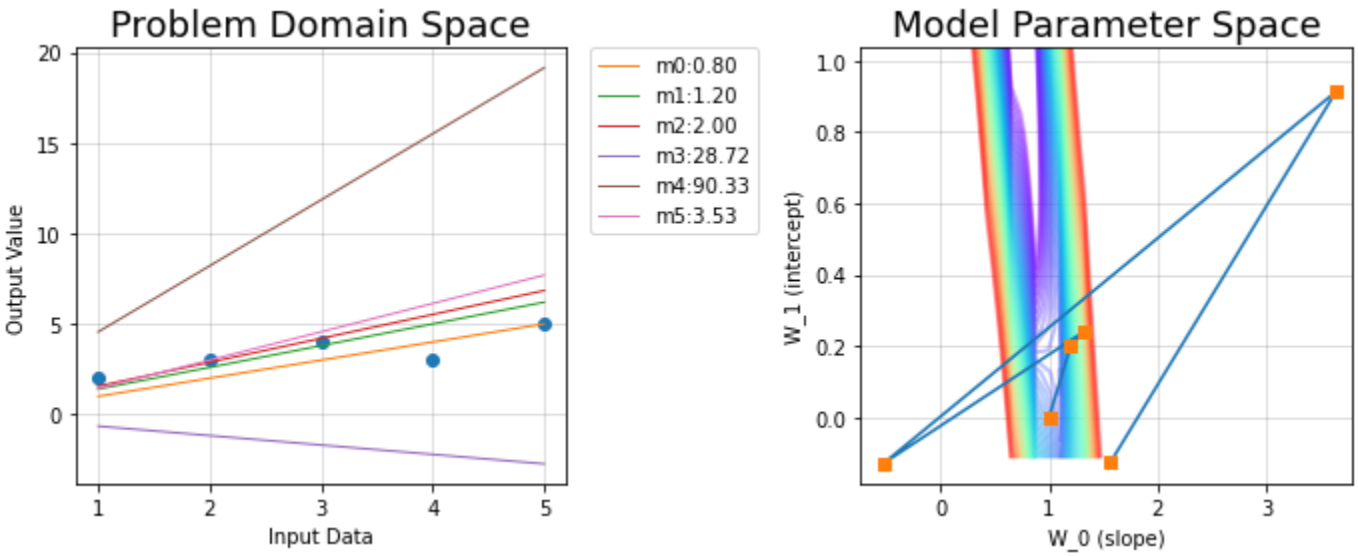
> Although the first few individual updates seem to go in odd directions by then end of one pass through the data we've got a much better model than the equivalent model after 1 GD step over the full dataset. This suggests that trainging via SGD will require fewer passes over the data.

- *How will parallelizing Stochastic Gradient Descent be similar/different to parallelizing regular GD?*

> With SGD, each data point produces a new parameter update for use in the subsequent computation of the next data point. We can see that SGD is by its nature a sequential algorithm, and paralellizing it is a challenge in a map reduce framework.

In [15]:
```
#################### Demo Parameters ####################
# TRY CHANGING THESE & SEE HOW IT AFFECTS OUR SEARCH
N_STEPS = 5
BATCHSIZE = 1
LEARNING_RATE = 0.1
ORIGINAL_MODEL = [1,0]
SHOW_CONTOURS = True

################### Stoichastic GD Demo ####################
### Load & Pre-process Data
X = augment(points)[:,:2]
y = points[:,1]

### Perform SGD Updates & save intermediate model performance
models, loss = SGDUpdate(X, y, N_STEPS,
                                    BATCHSIZE,
                                    ORIGINAL_MODEL,
                                    LEARNING_RATE,
                                    verbose = False)

### Display Results
print(f"\nSearched {len(models)} models..." %())
print(f"Best model: {models[np.argmin(loss)]}, Loss: {loss[np.argmin(loss)]}")
plotGDProgress(points, models, loss,
                           show_contours = SHOW_CONTOURS)
```

```
Searched 6 models...
Best model: [1. 0.], Loss: 0.8
```



> **DISCUSSION QUESTIONS:**
>
> - *At first glance does this seem to work as well as regular gradient descent? Why might our initial impression be deceiving?*
> - *Does adjusting the batchsize and/or learning rate fix the problem that we're seeing?*
> - *What do you notice about the direction of the first 3 updates? From the perspective of the first three points, what should our line look like?*
> - *How does the scale of our data impact the direction of our updates & time to convergence?*

<--- SOLUTION --->

INSTRUCTOR TALKING POINTS

- *At first glance does this seem to work as well as regular gradient descent? Why might our initial impression be deceiving?*

  > SGD will oscillate a lot so at first glance it looks very unstable - as if it is not traveling in the direction of the minimum.

- *Does adjusting the batchsize and/or learning rate fix the problem that we're seeing?*

  > For such a small dataset adjusting the batchsize and/or learning rate won't be very helpful.

- *What do you notice about the direction of the first 3 updates? From the perspective of the first three points, what should our line look like?*

  > The first thre points form a line in an upward right direction. Without seeing the other points, one might think this line would continue up towards the solution.

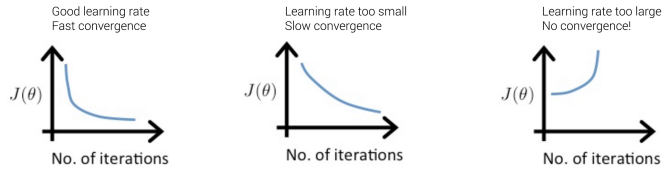
- *How does the scale of our data impact the direction of our updates & time to convergence?*

> In Stochastic GD, the scale of the data does not impact direction of the updates as the updates are made at each data point. Compared to Batch GD a large data set will need fewer passes (epochs) over the entire dataset to converge.

Selecting the learning rate

As you saw in the earlier eaxmples, increasing the learning rate reduces the number of iterations with which GD converges. However, care must be taken not to select a learning rate so high that the algorithm ends up overshooting the minimum and never converging at all!

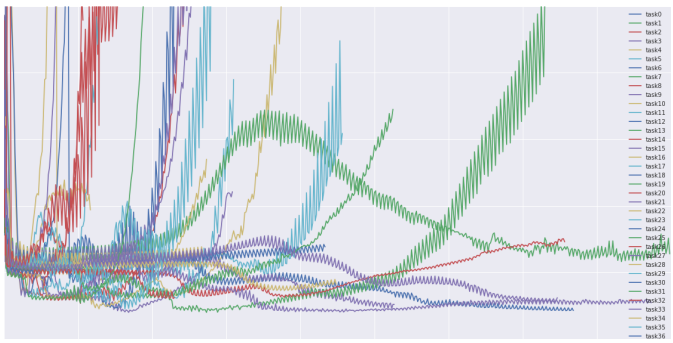
A standard approach to selecting the "best" learning rate is to do a grid search using a range of rates, say, `[0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3]`, then plotting the losses at each iteration to see which converges fastest.



https://www.coursera.org/learn/machine-learning/home/week/2
Gradient Descent in Practice II - Learning Rate

Taking this a step further (or even a few steps), a lot of work has been done on ways to "select" the learning rate to get GD to both converge faster, and perform better on unseen data. These variants of the GD algorithm include ideas like ADAM, AMSGrad, etc.. They work by dynamically adjusting the learning rate based on both recent gradients as well as time steps. A detailed discussion of these efforts are beyond the scope of this course. Below is an excellent overview of the most popular GD optimizers.

An overview of gradient descent optimization algorithms by Sebastian Ruder
http://ruder.io/optimizing-gradient-descent/



See also: https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9

For more info, here are a few of rabbit holes:

> https://arxiv.org/pdf/1707.00424.pdf
> https://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf http://papers.nips.cc/paper/4006-parallelized-stochastic-gradient-descent.pdf
> https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/distr_mini_batch.pdf

That's it for today!

Next week we will discuss...

- **L1 and L2 Regularization**
- **Common GD variants**
- **What to do if you can't compute a gradient for your loss function.**
- **Logistic Regression & classification**

Start Spark on the DataProc cluster

Note: If you get an error with launching Spark, please restart your cluster.

In [16]:
```
# start Spark Session (RUN THIS CELL AS IS)
from pyspark.sql import SparkSession
app_name = "Lab6_notebook"
master = "local[*]"
spark = SparkSession\
        .builder\
        .appName(app_name)\
        .master(master)\
        .getOrCreate()
sc = spark.sparkContext
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLev
el).
22/09/25 04:48:57 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
22/09/25 04:48:57 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
22/09/25 04:48:57 INFO org.apache.spark.SparkEnv: Registering BlockManagerMasterHeartbea
t
22/09/25 04:48:57 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
22/09/25 04:48:57 WARN org.apache.spark.util.Utils: Service 'SparkUI' could not bind on
port 4040. Attempting port 4041.
22/09/25 04:48:57 WARN org.apache.spark.util.Utils: Service 'SparkUI' could not bind on
port 4041. Attempting port 4042.
```

In [33]:
```
# get Spark Session info (RUN THIS CELL AS IS)
spark #NOTE the Spark UI link provided below as output from this command does not work.
```

Out[33]: **SparkSession - hive**

SparkContext

Spark UI

Version	`v3.1.3`
Master	`local[*]`
AppName	`Lab6_notebook`

Start Spark on the DataProc cluster

Note: If you get an error with launching Spark, please restart your cluster.

In [38]:
```
# get Spark Session info (RUN THIS CELL AS IS)
spark #NOTE the Spark UI link provided below as output from this command does not work.
```

Out[38]: **SparkSession - hive**

SparkContext

Spark UI

Version	`v3.1.3`
Master	`local[*]`
AppName	`Lab6_notebook`

Derive a gradient descent-based linear regression learning algorithm

$$\hat{y} = mx + b$$

Note we $\hat{y}$ as to represent a prediction or estimate of $f(x)$

$$error_i = \hat{y}_i - y_i$$

Linear Regression model prediction (in vectorized form)

$$\hat{y} = h_\theta(\mathbf{X}) = \mathbf{X} \cdot \theta$$

MSE cost function for a Linear Regression model

$$\mathrm{MSE}(\theta; \mathbf{X}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)} \cdot \theta - y^{(i)})^2$$

Partial derivatives notation :

$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\theta)$

Partial derivatives of the cost function with respect a single input θ_j

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)} \cdot \theta - y^{(i)})\, x_j^{(i)}$$

Gradient vector of the cost function vectorized

$$\nabla_{\mathrm{MSE}}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

Gradient Descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\mathrm{MSE}}(\theta)$$

In [21]:
```
# Simple linear regression
#           X              y
X_y =  [[  0.39050803,  -1.20623543],
        [  1.72151493,  13.57377242],
        [  0.82210701,   5.50818095],
        [  0.35906546,  -2.19996366],
        [ -0.61076161,  -3.90958845],
        [  1.1671529 ,  11.12900159],
        [ -0.49930231,  -3.63685934],
        [  3.13418401,  22.71362238],
        [  3.70930208,  25.53291143]]

data_rdd = sc.parallelize(X_y).cache()
# The true y = 8x - 2.
#.  [b, m]
W = [-2, 8]  # model
wBroadcast = sc.broadcast(W)  # make available in memory as read-only to
#                               the executors (for mappers and reducers)
#                            (               Xw             -     y)**2
#                              gradient  (Xw - y) X
MSE  = data_rdd.map(lambda d: (np.dot(np.append(1, d[:-1]),
                                      wBroadcast.value) - d[-1])**2).mean()
print(f"MSE:{MSE}")
```

```
[Stage 0:>                                                          (0 + 8) / 8]
MSE:5.832730881179018
```

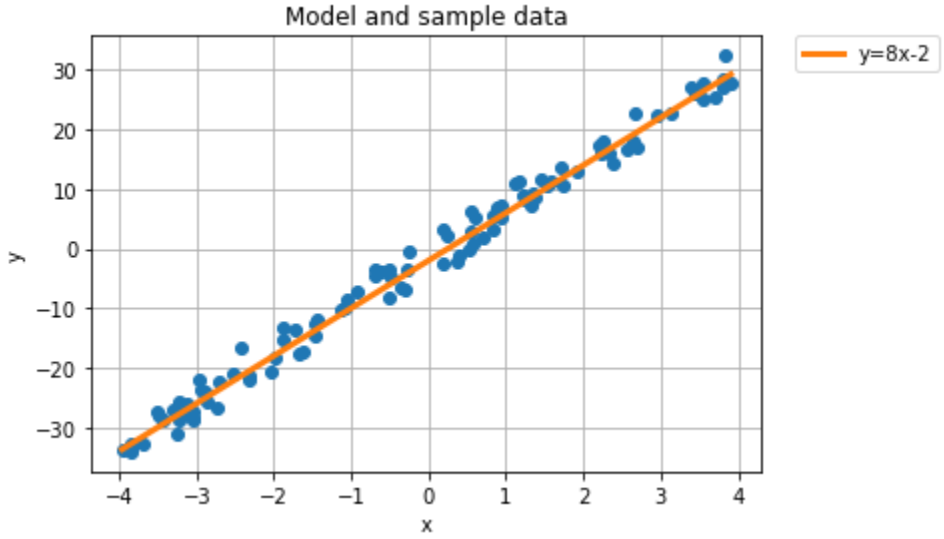
In [22]:
```
# gradient = (Xw - y) X
# gradient  =    X         *        (               Xw             -     y)**2
gradient  = data_rdd.map(lambda d: np.append(1,d[:-1]) *
                         (np.dot(np.append(1, d[:-1]),
                                 wBroadcast.value) - d[-1])).mean()
print(f"gradient:{gradient}")
#np.c_[1, d[:-1]]
#np.dot    OR  np.append(1,d[:-1]) @ W
```

```
gradient:[-0.43940865  0.64297919]
```

In [23]:
```
import numpy as np

def generate_data(w=[0,0], size=100):
    np.random.seed(0)
    x = np.random.uniform(-4, 4, size)
    noise = np.random.normal(0, 2, size)
    y = (x * w[1] + w[0] + noise)
    X_y = np.c_[x, y]
    return X_y

# The true y = 8x - 2.  giving a weight vector of [-2, 8]
X_y = generate_data([-2, 8], 100)
X_y[:10]
```

Out[23]:
```
array([[ 0.39050803, -1.20623543],
       [ 1.72151493, 13.57377242],
       [ 0.82210701,  5.50818095],
       [ 0.35906546, -2.19996366],
       [-0.61076161, -3.90958845],
       [ 1.1671529 , 11.12900159],
       [-0.49930231, -3.63685934],
       [ 3.13418401, 22.71362238],
       [ 3.70930208, 25.53291143],
       [-0.93246785, -7.35083934]])
```

In [24]:
```
def plot_model(samples, model, name, title=None):
    """
    Plot a model the and samples observed from the population.
    INPUT:  samples    - numpy array of points (x, y)
            model      - [W, b] where Y = Wx + b
            title      - (optional) figure title
    """

    # create plot
    fig, ax = plt.subplots()

    # plot samples
    ax.plot(X_y[:,0], X_y[:,1],'o')
    domain = [min(X_y[:,0]), max(X_y[:,0])]

    # plot model line
    w0 , w1 = model[0], model[1]
    yvals = [w1*x + w0 for x in domain]
    ax.plot(domain, yvals, linewidth=3, label=name)
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()
    # title
    if title:
        plt.title(title)
# The true y = 8x - 2.
model = [-2, 8]

# generate samples
X_y = generate_data(model, 100)

plot_model(X_y, model, 'y=8x-2', 'Model and sample data')
```



In [25]:
```
X_y_rdd = sc.parallelize(X_y).cache()
X_y_rdd.take(5)
```

Out[25]:
```
[array([ 0.39050803, -1.20623543]),
 array([ 1.72151493, 13.57377242]),
 array([0.82210701, 5.50818095]),
 array([ 0.35906546, -2.19996366]),
 array([-0.61076161, -3.90958845])]
```

compute LR MSE

In [26]:
```
def compute_LR_MSE(data_rdd, W=[0,0]):
    """ compute the MSE  for the provided linear regression model W over
        the data RDD where each record is of the form (X, y)
    INPUT:  data_rdd -an RDD where each record of the form  (X,y) = data[:-1], data[-1]
            W -a vector of weights, where W[0] corresponds to the y-intercept

    """

    wBroadcast = sc.broadcast(W)  # make available in memory as read-only to
    #                               the executors (for mappers and reducers)
    #                                 ( Xw                    -     y)**2
    return data_rdd.map(lambda d: (np.dot(np.append(1, d[:-1]), wBroadcast.value)
                                   - d[-1])**2).mean()

model = [0,0]
mean_squared_error = compute_LR_MSE(X_y_rdd, model)
print(f"the MSE of the  model is {model} is {mean_squared_error:.5}")

model = [-2,8]
mean_squared_error = compute_LR_MSE(X_y_rdd, model)
print(f"the MSE of the  model {model} is {mean_squared_error:.5}")

#the MSE of the  model [0, 0] is 354.35
#the MSE of the  model [-2, 8] is 4.119
```

```
the MSE of the  model is [0, 0] is 354.35
the MSE of the  model [-2, 8] is 4.119
```

In [27]:
```
# gradient = (Xw - y) X
# gradient  =    X         *        (                Xw             -     y)**2
gradient  = X_y_rdd.map(lambda d: np.append(1,d[:-1])  *
```

```
                        (np.dot(np.append(1, d[:-1]), wBroadcast.value) - d[-1])).mean()
print(f"gradient:{gradient}")
```

```
gradient:[-0.38466869  0.16759204]
```

In [ ]: