A data folder for doing this lab can be found **here**. Download this to your computer.

In this lab we will look at feature engineering for car data, written food reviews, images, and sound. Please make sure you read the lecture notes covering feature representations for this lab.

1) Feature engineering for car data

Last week, given a dataset of feature vectors and their corresponding labels, we saw how to implement two algorithms for building linear classifiers. This week we are looking at how these features are defined from raw collected data and some implications of those feature choices on the learning algorithm.

In this part of the lab, we are going to explore different ways of defining features for data.

Open `auto-mpg.tsv` in a text editor (or view on the web here). This file is in a common format, called "tab separated values". In the first line you will find the names of the columns. Each row is a data point; **the first column is the label for that point (1 or -1).**

"*The data concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 3 multivalued discrete and 5 continuous attributes.*" (Quinlan, 1993)

The original data came from the StatLib library from CMU. It was modified by Ross Quinlan to remove entries with unknown mpg (miles per gallon). We have modified it further by removing six entries with unknown horsepower. We have also binarized the mpg column to turn this into a classification problem (later in the term, we will look at predicting continuous values, i.e. regression problems). Here are the nine columns in the dataset:

```
1. mpg:           continuous (modified by us to be -1 for mpg < 23, 1 for others)
2. cylinders:     multi-valued discrete
3. displacement:  continuous
4. horsepower:    continuous
5. weight:        continuous
6. acceleration:  continuous
```

```
7. model year:    multi-valued discrete
8. origin:        multi-valued discrete
9. car name:      string (many values)
```

There are 392 entries in the dataset, evenly split between positive and negative labels. The field names should be self-explanatory except possibly `displacement` and `origin`. You can read about `displacement` here; in this data set `displacement` is in cubic inches. `origin` is 1=USA, 2=Europe, 3=Asia. We'll ignore `car name` in this assignment.

A new student, Hiper Playne, suggests that we should just use all the numeric features in their raw form to predict whether the car will get good or bad gas mileage. He will use the Perceptron algorithm to learn the classifier. Once he trains the model on this dataset, he wants to predict whether cars in 2019 will get good gas mileage.

1A) What problems do you think Hiper might have with this method?

- [x] Because `weight` values and cylinders values are on different scales, perceptron might take many iterations
- [ ] `cylinders` is a discrete valued feature
- [x] `origin` is a discrete valued feature
- [ ] There are too many features and the classifier will overfit
- [x] `model year` is in the 70s, so a classifier based on this data might not perform well in 2019

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

1B) For each feature from the following:
`[cylinders, displacement, horsepower, weight, acceleration, model_year, origin]`
indicate how you can represent it so as to make classification easier and get good generalization on unseen data, by choosing one of:

- `'drop'` - leave the feature out,
- `'raw'` - use values as they are,
- `'standard'` - standardize values by subtracting out average value and dividing by standard deviation,
- `'one-hot'` - use a one-hot encoding.

There could be multiple answers that make sense for each feature; please mention the tradeoffs between each answer. Write down your choices.

1C) How can `car name`, a textual feature, be transformed into a feature which can be used by the perceptron algorithm?

1D) Given a learning algorithm (example: perceptron), say you found 3 ideas for how to represent the textual feature from 1C. Talk with your partner to determine which of the following options would be appropriate, for determining the best feature representation out of the 3 ideas:

- For each feature representation, compute a classifier by running the algorithm on all the data and compare the number of errors of that classifier on the data.
- Split the data into two sets: training is 90%, test is 10%. For each feature representation, compute a classifier by running the algorithm on the training data, compare the sum of the number of errors of that classifier on the training and test data.
- Split the data into two sets: training is 90%, test is 10%. For each feature representation, compute a classifier by running the algorithm on the training data, compare the number of errors of that classifier on the test data.
- For each feature representation, perform 10-fold cross-validation and compare the resulting average error rate.

2) Feature engineering for food reviews

In this part of the assignment, we are going to explore ways of defining features for textual data.

Open `reviews.tsv` in a text editor (or view on the web here). This file is in "tab separated values" (tsv) format, and it consists of 10,000 fine food reviews from Amazon. You can find additional information here. We will be focusing on the `text` field and use it to predict the `sentiment` field (-1 or 1).

We will convert review texts into fixed-length feature vectors using a bag-of-words (BOW) approach. We start by compiling a list of all the words

that appear in a **training** set of reviews into a *dictionary*, thereby producing a list of d *unique* words.

2A) For instance, consider two simple documents "Mary is selling apples." and "Tom is buying apples to eat". If we had only these two sentences in our training set of reviews, which option corresponds to the dictionary that we build when using the bag-of-words approach discussed above?

- $(Mary,\ is,\ selling,\ apples,\ Tom,\ is,\ buying,\ apples,\ to,\ eat)$
- **(selected)** $(Mary,\ is,\ selling,\ apples,\ Tom,\ buying,\ to,\ eat)$
- $(Mary,\ is,\ selling,\ red,\ apples,\ Tom,\ buying,\ blue,\ to,\ eat)$

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

We can then transform each of the reviews into a feature vector of length d by setting the i^{th} coordinate of the feature vector to 1 if the i^{th} word in the dictionary appears in the review or 0 if it does not.

2B) Hence, how would we represent the sentence "Tom is buying apples to eat" as a feature vector using the bag-of-words approach and the dictionary we found above?

- $(1, 1, 0, 0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1, 0, 1, 1)$
- **(selected)** $(0, 1, 0, 1, 1, 1, 1, 1)$

Submit | View Answer | Ask for Help **100.00%**

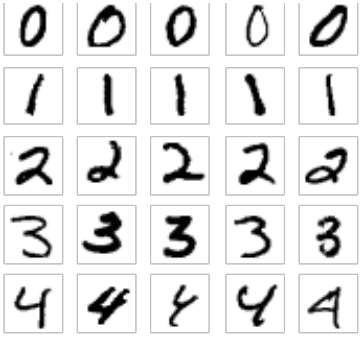
You have infinitely many submissions remaining.

2C) Talk with your partner about the weaknesses of the bag-of-words approach seen above. For instance, how would you interpret the feature vector $(1, 1, 1, 1, 1, 0, 1, 0)$? (Who is selling what to whom?)

2D) Words like "the", "to", "a", "and" and so on occur with very high frequency in English sentences. Do they help in detecting the sentiment in a review? Talk to your partner about how to deal with these words, when building your dictionary, using the bag-of-words approach.

3) Image features

We will be exploring in the homework how the perceptron algorithm might be applied to classify images of handwritten digits, from a well-known ("MNIST") dataset, with items like these:

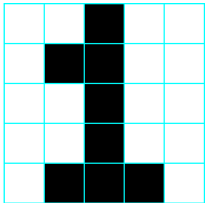


For today, assume we only have images of digits '1' and '3' and we would like to find a linear classifier which can classify these images correctly, giving label $+1$ for the digit '1' and label -1 for the digit '3'.

For simplicity, assume each image provided is a 5-pixel wide and 5-pixel tall (5x5) black and white image.

3A) Assume images from the MNIST dataset are provided to you as 5x5 matrices. An image is represented as a matrix, say A, with the entry of the i^{th} row and j^{th} column ($A_{i,j}$) representing the image pixel found at the i^{th} row and j^{th} column of pixels. Black pixels have value 0 while while pixels have value 1 in the matrix.

With the help of your partner, write down the matrix corresponding to the image shown here:



3B) Imagine we want to use the perceptron algorithm to perform the task of distinguishing between images of digits '1' and '3'. Each image is a data point, which we need in **vector** form, to feed to the perceptron algorithm. One approach is to take the two-dimensional matrix representation of an image and concatenate the rows one after the other, into a one-dimensional vector of the form
$[A_{1,1}, A_{1,2}, A_{1,3}, A_{1,4}, A_{1,5}, A_{2,1}, A_{2,2}, ..., A_{5,5}]$

For the image shown above, what will be the last 3 entries of the one-dimensional vector representing that image? Enter a python list of length 3 for your answer. [0,0,1]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

3C) How well would you expect the perceptron algorithm to work on classifying images if you simply represented them as vectors? Is there any information lost when representing images this way?

3D) What approaches might you suggest to extract more meaningful features from the images, such that the perceptron algorithm might do a good job of classification? (Hint: What makes the image of the digit '1' different compared to the image of the digit '3'?)

In the homework for this week we will investigate these datasets in more detail. And later in the semester, we'll explore other algorithms, including neural networks and convolutional neural networks, which can essentially do feature extraction on their own.

4) Acoustic features

Hiper Playne now has access to a large number of one-second sound clips, each one sounding like a dog bark or a cat meow. Each sound clip is

a vector of length 100, with an amplitude value in the range [0,1] assigned for discrete time steps $0.00, 0.01, 0.02, ..., 0.99$. He wants to use the perceptron algorithm to learn a classifier which can separate dog bark from cat meow sound clips.

4A) How well would you expect the perceptron algorithm to work, if it was given these raw sound clips as input?

4B) What makes a cat's meow different from a dog's bark? If you had a black box which can take in a sound clip and return the 5 most dominant frequencies heard in the sound clip, how would you use the black box and extract features from the sound clips? How would you represent those features as a vector which will then be fed to the perceptron algorithm?

> For each feature from the following: [cylinders, displacement, horsepower, weight, acceleration, model_year, origin] indicate how you can represent it so as to make classification easier and get good generalization on unseen data, by choosing one of: 'drop' - leave the feature out, 'raw' - use values as they are, 'standard' - standardize values by subtracting out average value and dividing by standard deviation, 'one-hot' - use a one-hot encoding. There could be multiple answers that make sense for each feature; please mention the tradeoffs between each answer. Write down your choices.

- cylinders

Raw as value is meaningful, standard

- displacement

Standard as larger variance.

- Horsepower Standard as larger variance.
- Weight Standard as larger variance.
- Acceleration Raw, or standard
- model_year Perhaps change to num of years till 2022. Use raw or standard.
- origin One-hot

> 1C) How can car name, a textual feature, be transformed into a feature which can be used by the perceptron algorithm?

1) Tokenize all words in names. And use 'one-hot' that allows several selected. 2) Factor names: extract producer name. That is take the first word. Use 'one-hot' 3) Use unsupervised learning to get clusters of names 4) Order, numerate, standardize.

> 1D)
>
> - For each feature representation, compute a classifier by running the algorithm on all the data and compare the number of errors of that classifier on the data.

Not good as classifier will fit to the training data.

> Split the data into two sets: training is 90%, test is 10%. For each feature representation, compute a classifier by running the algorithm on the training data, compare the sum of the number of errors of that classifier on the training and test data.

Not good as comparing the sum so it is similar to the above.

> Split the data into two sets: training is 90%, test is 10%. For each feature representation, compute a classifier by running the algorithm on the training data, compare the number of errors of that classifier on the test data.

Good.

> For each feature representation, perform 10-fold cross-validation and compare the resulting average error rate.

Good.

2)

> 2C) Talk with your partner about the weaknesses of the bag-of-words approach seen above. For instance, how would you interpret the feature vector (1,1,1,1,1,0,1,0)(1,1,1,1,1,0,1,0)(1,1,1,1,1,0,1,0)? (Who is selling what to whom?)

Weaknesses:

- Sentence units (subject, predicate, objective, etc.) lost, i.e. meaning is lost.
- Doesn't count frequency. So 'one plus one plus one plus one' is the same as 'one plus one' but 4 is not 2.
- Takes less important words like adjectives, conjuctions, etc.

> 2D) Words like "the", "to", "a", "and" and so on occur with very high frequency in English sentences. Do they help in detecting the sentiment in a review?

Remove them manually. Or perhaps use some threashold: remove all top ones till they some large difference.

3)

> With the help of your partner, write down the matrix corresponding to the image shown here:

```
11011
10011
11011
11011
10001

11011 10011 11011 11011 10001
```

> 3C) How well would you expect the perceptron algorithm to work on classifying images if you simply represented them as vectors? Is there any information lost when representing images this way?

Might not be linearly separable. Losing dimension info might result in bad predictions? Example is adding 11 at beginning and removing 01 at end:

```
11110
11100
11110
11110
11100
```

Seems ok? It should be classified '1'. Will perceptron classify such digits?

> 3D) What approaches might you suggest to extract more meaningful features from the images, such that the perceptron algorithm might do a good job of classification? (Hint: What makes the image of the digit '1' different compared to the image of the digit '3'?)

2 dimentions change, not 1? More dots? Larger, longer? Number of 'ends'? 3 for 3, 2 for 1.

> 4) 4A) How well would you expect the perceptron algorithm to work, if it was given these raw sound clips as input?

If it might be not lin. separable then add extra dim. Then how?

- Long to converge? What if we have very long and very large amplitude: 0.00000001 and 1? Then it will increment very slowly. So to standardize the data? So that those far will be closer?
- What if the sound is shifted towards beginning or end? But we can shift the sample to solve it in the case when we take an 1 sec interval from a larger sound clip.

> 4B) What makes a cat's meow different from a dog's bark? If you had a black box which can take in a sound clip and return the 555 most dominant frequencies heard in the sound clip, how would you use the black box and extract features from the sound clips? How would you

> dominant frequencies heard in the sound clip, how would you use the black box and extract features from the sound clips? How would you represent those features as a vector which will then be fed to the perceptron algorithm?

A meow is longer than a dog's bark. Also the bark with higher max amplitude. Use that black box to get those 5 frequencies. Then add 'height' feature (diff between min and max) in absolute value, median feature in absolute value, the top frequent one.

In the first part of this assignment, we will consider several general issues in representing features and their impact on classification. In the second part of the assignment, we will experiment with these strategies in the context of realistic data sets. Please make sure your read the lecture notes covering feature representations for this assignment.

Feature Transformations

A code file that is required for this assignment can be found here, and a colab notebook here.

1) Scaling

Consider a linearly separable dataset with two features:

```
data = ([[200, 800, 200, 800],
         [0.2,  0.2,  0.8,  0.8]])
labels = [[-1, -1, 1, 1]]
```

Consider the separator defined by $\theta = (0, 1), \theta_0 = -0.5$.

In order to apply the perceptron mistake bound (see notes), we transform our problem from $\theta^T x + \theta_0 = 0$ to some $\theta'^T x = 0$. We do this by appending θ_0 to θ, and appending 1 to x, as follows:

$$\theta'^T x = \begin{bmatrix} \theta_1 & \theta_2 & \dots & \theta_0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ 1 \end{bmatrix} = 0$$

In this phrasing, our new "θ" is (0, 1, -0.5). For a separator through the origin, recall that the margin of the data set is the minimum of $\gamma =$

$y^{(i)}(\theta^T x^{(i)})/||\theta||$ over all data points $(x^{(i)}, y^{(i)})$.

For the following questions, assume we are working in the transformed (3d) feature space, with perceptron through the origin, and where if the data has bounded magnitude R, then the theoretical upper bound on mistakes made by perceptron is $(\frac{R}{\gamma})^2$, for a separable data set.

You are free to use your perceptron algorithm implemented in the previous homework to answer the following questions. (Some parts require more runs of the perceptron algorithm than one could reasonably perform by hand.)

1A)

What is the margin γ of this data set with respect to that separator (up to 3 decimal places)?

0.26832816

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1B)

What is the theoretical bound on the number of mistakes perceptron will make on this problem?

8888912

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1C)

How many mistakes does perceptron through origin have to make in order to find a perfect separator on the data provided above, in the order given? (Try it on your computer, not by hand!) 666696

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1D)

If we were to multiply both original features of all of the points by $.001$, and considered the separator through origin $\theta = (0, 1, -0.0005)$, what would the margin of the new dataset be? 0.0003

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1E)

How would the performance of the perceptron (as predicted by the mistake bound) change? More mistakes

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1F)

If we multiplied just the first original feature (first row of the data) by .001, and used our original separator, what would the new margin be? 0.2683281572999748

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1G)

What would the mistake bound be in this case? 31.66666

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1H)

Run the perceptron algorithm on this data; how many mistakes does it make?

11

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

2) Encoding Discrete Values

Some data sets have features that take on discrete values drawn from a set. Examples might be:

- which section of a class a student is in (1, 2, 3, 4)
- manufacturer of a cell phone (Samsung, Xiaomi, Sony, Apple, LG, Nokia)
- which laboratory performed a particular medical test

Sometimes they already have an obvious encoding into integers; other times, they don't but it's easy to make one (e.g., Samsung = 1, Xiaomi = 2, Sony = 3, Apple = 4, LG = 5, Nokia = 6)

2A) Let's consider the case of the cell phones, using the encoding above, and imagine there is some prediction problem, such as predicting whether the phone will last three years, for which we have the data set:

```
data =   [[2, 3,  4,  5]]
labels = [[1, 1, -1, -1]]
```

What value of θ and θ_0 would we get when running perceptron on this data? You are free to use the perceptron implemented in homework 2.

Enter a Python list with two floats, one for θ and one for θ_0. [-2., 7.]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2B) What prediction would we make about other phone types based on this classifier?

Enter a Python list with two labels (1 or -1), the first one for a Samsung phone and the second for a Nokia phone.

[1, -1]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2C)

Are these predictions meaningful given the training data we used? No

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2D) It is common to encode a feature which takes on a value from a set of discrete values, not as a single multi-valued feature, but using a *one hot* encoding.

Here, assume you have a feature f which can take on any value from the set $\{1, 2, ..., k\}$. If f takes on value i, then we represent it as a vector of length k of all zeros, except for a +1 at the ith coordinate.

Write a function `one_hot` that takes as input x, a single feature value (between 1 and k), and k, the total possible number of values this feature can take on, and transform it to a numpy column vector of k binary features using a one-hot encoding (remember vectors have zero-based indexing).

For example, `one_hot(3,7)` should return a column vector of length 7 with the entry at index 2 taking value 1 (indices start at 0) and other entries taking value 0.

```python
def one_hot(x, k):
    a = np.zeros((k, 1))
    a[x-1,0] = 1.0
    return a
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

2E) What happens if we use one-hot encoding on the data set part 2A) above, and put it into the perceptron? Recall that for a classifier $h(x)$, the prediction is $+1$ if $h(x) > 0$ and -1 otherwise. Further note that the perceptron algorithm makes an update whenever $y^{(i)}(\theta^T x^{(i)} + \theta_0) \leq 0$.

2E i) What is the separator produced by the perceptron algorithm?

Enter a Python list with 7 floats, six for θ and one for θ_0. [0., 2., 1., -2., -1., 0., 0.]

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

2E ii) What are the predictions for Samsung and Nokia?

Enter a Python list with two labels (1 or -1), the first one for a Samsung phone and the second for a Nokia phone.

[-1, -1]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2E iii) What are the distances for the Samsung and Nokia data points from the separator?

Enter a Python list with two distances, the first one for a Samsung phone and the second for a Nokia phone.

[0., 0.]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2F) Now, what if we have this dataset:

```
data =   [[1, 2, 3, 4, 5, 6]]
labels = [[1, 1, -1, -1, 1, 1]]
```

Is it linearly separable in the original encoding? No

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2G) Is it linearly separable in the one-hot encoding? If so, provide the separator found by the perceptron.

Enter a Python list with 7 floats, six for θ and one for θ_0 or `'none'` [1., 1., -2., -2., 1., 1., 0.]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2H) Enter an assignment of data values to labels (with distinct data points) that is not linearly separable using the one-hot encoding, or enter None if no such assignment exists.

Enter a Python list with 6 tuples `(value, label)` or `'none'` 'none'

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3) Polynomial Features

One systematic way of generating non-linear transformations of your input features is to consider the polynomials of increasing order. Given a feature vector $x = [x_1, x_2, ..., x_d]^T$, we can map it into a new feature vector that contains all the factors in a polynomial of order d. For example, for $x = [x_1, x_2]^T$ and order 2, we get

$$\phi(x) = [1, x_1, x_2, x_1 x_2, x_1^2, x_2^2]^T$$

and for order 3, we get

$$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1^2x_2, x_1x_2^2, x_1^3, x_2^3]^T.$$

In the code file, we have defined `make_polynomial_feature_fun` that, given the order, returns a feature transformation function (analogous to ϕ in the description). You should use it in doing this problem.

3A)

Enter a list of 6 integers indicating the number of polynomial features of degrees [1, 10, 20, 30, 40, 50] for a 2-dimensional feature vector. [3, 66, 231, 496, 861, 1326]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3B) Consider this data-set of four points in two-dimensional space:

```
data = ([[1, 1, 2, 2],
         [1, 2, 1, 2]])
labels = [[-1, 1, 1, -1]]
```

It is standardly called the "exclusive-or" or "xor" problem. These points are not linearly separable, and you could interpret each point as being a pair of truth values, with their label being the XOR of the values.

In the code file, we have defined 4 sample data sets, (1) `super_simple_separable_through_origin`, (2) `super_simple_separable`, (3) `xor`, and (4) `xor_more`. On your own machine, you should run the code we have provided (`test_with_features`) for various orders of polynomial features and enter below the order of the smallest feature that separates the data. Make sure that you have included your implementation of `perceptron` in that file or you can use the implementation we have provided. You may need to adjust the number of iterations that the perceptron runs.

The separators are displayed when the code runs; it's instructive to watch them to see the range of separators that these non-linear

transformations produce. Note that the separators are drawn by evaluating the feature transformations on a grid of points in the feature space and using the separator to classify them. (Note: If you have issues with the graphic not moving forward, try pressing the keys within your terminal.)

Enter a Python list of integers indicating the smallest polynomial order for which a separator exists for each of the four datasets in the code file (in order). [1,1,2,3]

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

Experiments

A code and data folder that will be necessary for doing this homework can be found at the top of the page. In the file `code_for_hw3_part2.py`, include your learner code from HW 2. **You will want to modify the evaluation algorithms so that they take a `T` argument to pass to the learners.**

The rest of this assignment will require running the code on your computer; we will be asking only for the results of your runs.

4) Evaluating algorithmic and feature choices for AUTO data

We now want to build a classifier for the auto data, with a focus on the numeric data. In the `code_for_hw3_part2.py`, we have supplied you with the `load_auto_data` function, which can read the relevant .tsv file. It returns a list of dictionaries, one for each data item.

We then specify what feature function to use for each column in the data. The file `hw3_part2_main.py` has an example that constructs the data and label arrays using `raw` features for all the columns.

In the list `features` of `hw3_part2_main.py`, you will find a list of feature name, feature function tuples. There are three options for feature functions: `raw`, `standard` and `one_hot`. `raw` uses the original value; `standard` subtracts out the mean value and divides by the standard deviation; and `one_hot` will one-hot encode the input, as described in the notes.

The function `auto_data_and_labels` processes the dictionaries and return `data, labels`. `data` has dimension $(d, 392)$, where d is the total number of features specified, and `labels` has dimension $(1, 392)$. The data in the file is sorted by class, but it will be shuffled when loaded.

We have included staff implementations of `perceptron` and `average perceptron` in `code_for_hw3_part2.py`. Using the feature arrays and these implementations, you will be able to compute θ and θ_0.

We have also included staff implementations of `eval_classifier` and `xval_learning_alg` (in the same code file). You should use these functions to report accuracies.

4.1) Making choices

We know of two algorithm classes: perceptron and averaged perceptron (which we implemented in HW 1). We have a several parameters that specify the settings for these learning algorithms.

A) Which parameters should we use for the learning algorithm? In the perceptron and averaged perceptron, there is a single parameter, T, the number of iterations.

B) Which features should we use? We have lots of choices here: we can use any subset of the data columns and for each column we have choices of how to compute features.

C) We will use expected accuracy, estimated by 10-fold cross-validation (we have included the definition in the code file), to make these choices of parameters.

- We will try two types of algorithms: perceptron and averaged perceptron.
- We will try 3 values of T: $T = 1$, $T = 10$, $T = 50$.
- We will try 2 feature sets:

1. `[cylinders=raw, displacement=raw, horsepower=raw, weight=raw, acceleration=raw, origin=raw]`

2. `[cylinders=one_hot, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`

Perform 10-fold cross-validation for all combinations of the two algorithms, three T values, and the two choices of feature sets. It will be worthwhile investing in a piece of code to carry out all of the evaluations, in case you need to do this more than once.

In general, you should shuffle the dataset before evaluating, but for this exercise, please use `hw3.xval_learning_alg`, which shuffles the dataset for you, so that your results match ours.

4.1C i)

Enter accuracies (perceptron, averaged perceptron) for T=1, feature set 1:

[0.6526282051282052, 0.8441025641025641]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

4.1C ii)

Enter accuracies (perceptron, averaged perceptron) for T=1, feature set 2:

(0.7908333333333333, 0.9004487179487182)

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

4.1C iii)

Enter accuracies (perceptron, averaged perceptron) for T=10, feature set 1:

(0.7423076923076924, 0.8366025641025641)

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

4.1C iv)

Enter accuracies (perceptron, averaged perceptron) for T=10, feature set 2:

(0.8061538461538461, 0.8979487179487181)

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

4.1C v)

Enter accuracies (perceptron, averaged perceptron) for T=50, feature set 1:

(0.6909615384615384, 0.8366025641025641)

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

4.1C vi)

Enter accuracies (perceptron, averaged perceptron) for T=50, feature set 2:

(0.8060256410256409, 0.9005128205128207)

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Now we have the data we need to make rational choices.

4.1D) Which algorithm class is typically more effective?

Pick one: Averaged Perceptron

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.1E) For the better algorithm, which combination of T and feature would you use? Consider expected accuracy as of primary importance, take into account running time for near ties in accuracy.

Enter a tuple of two integers (T, feature_set): (1, 2)

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4.2) Analysis

4.2 A) For the best algorithm type, best T and best feature set, construct your best classifier (θ, θ_0) using all the data. Based on the values of the coefficients, which feature has the most impact on the output predictions?

Choose the name of one feature: weight

Submit | View Answer | Ask for Help | 100.00%

You have infinitely many submissions remaining.

4.2 B) (Optional) Is there any set of two features you can use to attain comparable results as your best accuracy? What are they?

5) Evaluating algorithmic and feature choices for review data

In the code file (and the colab notebook), we have supplied you with the `load_review_data` function, that can be used to read a .tsv file and return the labels and texts. We have also supplied you with the `bag_of_words` function, which takes the raw data and returns a dictionary of unigram words. The resulting dictionary is an input to `extract_bow_feature_vectors` which computes a feature matrix of ones and zeros that can be used as the input for the classification algorithms. The file `hw3_part2_main.py` has code for constructing the data and label arrays. Using these arrays and our implementation of the learning algorithms, you will be able to compute θ and θ_0. You will need to add your (or the one written by staff) implementation of perceptron and averaged perceptron.

5.1) Making choices

We have two algorithm classes: perceptron and averaged perceptron. We have a couple of choices of parameters to make to completely specify the learning algorithms.

5.1A) Which parameters should we use for the learning algorithm? In the perceptron and averaged perceptron, there is a single parameter, T, the number of iterations.

5.1B) Which features should we use? We could do variations of bag-of-words, for example, simply indicating whether a word is present or, alternatively, using a count of how many times it is present. We can also remove commonly used words with little information; the code distribution includes a file of those words: `stopwords.txt`. You're welcome to explore these on your own; we'll use only a binary indicator for all the words.

5.1C) Perform 10-fold cross-validation for all combinations of the two algorithms and three T values (1, 10, 50). Record the accuracies for each combination (there should be 6 values total).

Note: These tests may take a couple of minutes to run; don't expect instant response, but it shouldn't run for 10 minutes.

Now we have the data we need to make rational choices.

5.1D) Which algorithm class is typically more effective?

Pick one: Averaged Perceptron

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

5.1E) For the better algorithm, which value of T would you use? Consider expected accuracy as of primary importance, take into account running time for near ties in accuracy.

Enter a value of T: 10

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

5.1F) For the better algorithm and best value of T, what is your accuracy?

Enter a number between 0 and 1: 0.8237

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

5.2) Analysis

For the best algorithm and best T, construct your best classifier (θ, θ_0) using all the data.

Note: We have included a function called `reverse_dict` in `code_for_hw3_part2.py` that you may find convenient. You are not required to use this function.

5.2A) What are the 10 most positive words in the dictionary, that is, the words that contribute most to a positive prediction?

Enter a Python list of ten strings: ['great', 'individually', 'bright', 'yummy', 'skeptical', 'perfect', 'easily

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

5.2B) What are the 10 most negative words in the dictionary, that is, the words that contribute most to a negative prediction.

Enter a Python list of ten strings: ['worst', 'awful', 'unfortunately', 'horrible', 'stuck', 'changed', 'disap

Submit | View Answer | Ask for Help | 100.00%

You have infinitely many submissions remaining.

5.2C) (Optional) You might find it amusing to find the most positive and most negative reviews. That is, ones with the most positive and negative signed distance to the hyperplane.

6) Evaluating features for MNIST data

This problem explores how well the perceptron algorithm works to classify images of handwritten digits, from the well-known ("MNIST") dataset, buiding on your thoughts from lab about extracting features from images. This exercise will highlight how important feature extraction is, before linear classification is done, using algorithms such as the perceptron.

Dataset setup

Often, it may be easier to work with a vector whose spatial orientation is preserved. In previous parts, we have represented features as one long feature vector. For images, however, we often represent a m by n image as a `(m,n)` array, rather than a `(mn,1)` array (as the previous parts have done).

In the code file, we have supplied you with the `load_mnist_data` function, which will read from the provided image files and populate a dictionary, with image and label vectors for each numerical digit from 0 to 9. These images are already shaped as `(m,n)` arrays.

6.1) Feature extraction

In the real world, there may be complicated ways to extract meaningful features from images, but in this section we will explore several simple methods.

6.1A) You may notice that some numbers (like 8) take up more horizontal space than others (like 1). We can compute a feature based on the average value in each row. Write a Python function that takes in a `(m,n)` array and returns a `(m,1)` array, where element `i` is the average value in row `i`.

```
                lambda a: [np.average(a)],
                axis=1,
                arr=xi
            )
            for xi
            in x
        ])
    elif ndims == 2:
        return np.apply_along_axis(
            lambda a: [np.average(a)],
            axis=1,
            arr=x
        )

```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

6.1B) We can also compute a feature based on the average value in each column. Write a Python function that takes in a `(m,n)` array and returns a `(n,1)` array, where element `j` is the average value in column `j`.

```
        return np.apply_along_axis(
            lambda a: [np.average(a)],
            axis=0,
            arr=arr
        ).T
    if ndims == 3:
        return np.array([
            f(xi)
            for xi
            in x
        ])
    elif ndims == 2:
        return f(x)

```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

6.1C) Finally, you may notice that some features are more "top heavy" while others are more "bottom heavy." Write a function that takes in a `(m,n)` array and returns a `(2,1)` array, where the first element is the average value in the top half of the image, and the second element is the average value in the bottom half of the image.

You may use the `cv` function from homework 1.

```
    @return (2,n_samples) array where the first entry of each column is the average of the
    top half of the image = rows 0 to floor(m/2) [exclusive]
    and the second entry is the average of the bottom half of the image
    = rows floor(m/2) [inclusive] to m
    """
    ndims = len(x.shape)
    def f(arr):
        n, m = x.shape
        return cv([np.average(x[0:n//2,:]), np.average(x[n//2:,:])])
    if ndims == 3:
        return np.array([
            f(xi)
            for xi
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

Important: In `hw3_part2_main.py`, you may need to **modify** these functions to accept a data matrix of size `(n, 28, 28)`, or you may use these functions in other ways (loops are allowed).

6.2) Feature evaluation

We can use these features to distinguish between numbers.

Important: For this section, we will be using `T=50` with the base perceptron algorithm to train our classifier and 10-fold cross validation to evaluate our classifier. A function called `get_classification_accuracy` has already been implemented for you in `code_for_hw3_part2` to compute the accuracy, given your selected data and labels.

You *must* use our implementation of cross validation to report accuracies (you may call `hw3.xval_learning_alg`).

6.2A) First we will find baseline accuracies using the raw 0-1 features.

Convert each image into a `(28*28, 1)` vector for input into the perceptron algorithm. **Hint:** `np.reshape` may be helpful here.

Run the perceptron on four tasks: 0 vs. 1, 2 vs. 4, 6 vs. 8, and 9 vs. 0.

Enter a list of accuracies [0 vs. 1, 2 vs. 4, 6 vs. 8, 9 vs. 0]: (0.975, 0.8641666666666665, 0.9479166666666667, 0.647083333333

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

Now let's evaluate the features we extracted.

6.2B) Using the extracted features from above, run the perceptron algorithm on the set of 0 vs. 1 images.

Enter a list of accuracies [row, column, top/bottom]: [0.48125, 0.6375, 0.48125]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

6.2C) Using the extracted features from above, run the perceptron algorithm on the set of 2 vs. 4 images.

Enter a list of accuracies [row, column, top/bottom]: [0.7754166666666668, 0.49749999999999994, 0.497499999999999

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

6.2D) Using the extracted features from above, run the perceptron algorithm on the set of 6 vs. 8 images.

Enter a list of accuracies [row, column, top/bottom]: [0.92125, 0.52125, 0.5650000000000001]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

6.2E) Using the extracted features from above, run the perceptron algorithm on the set of 9 vs. 0 images.

Enter a list of accuracies [row, column, top/bottom]: [0.49749999999999994, 0.5041666666666667, 0.497499999999999

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

6.2F) (Optional) What does it mean if a binary classification accuracy is below 0.5, if your dataset is balanced (same number from each class)? Are these datasets balanced?

6.2G) (Optional) Feel free to classify other images from each other. Which combinations perform the best, and which perform the worst? Do these make sense? Other than row and column average, are there any other features you could think of that would preserve some spatial information?

MIT 6.036 Spring 2019: Homework 3

This colab notebook provides code and a framework for problems 1-7 of the homework. You can work out your solutions here, then submit your results back on the homework page when ready.

Setup

First, download the code distribution for this homework that contains test cases and helper functions.

Run the next code block to download and import the code for this lab.

In [1]:
```
!pwd
```

```
/home/art/mydir/ref/Учеба_конспекты_решения/mit_6.036_intro_to_ml
```

In [2]:
```
#!rm -rf code_and_data_for_hw3*
#!rm -rf mnist
#!wget --quiet https://introml_oll.odl.mit.edu/6.036/static/homework/hw03/code_and_data_for_hw3.zip
#!unzip code_and_data_for_hw3.zip
#!mv code_and_data_for_hw3/* .

from code_for_hw3_part1 import *
import code_for_hw3_part2 as hw3
```

```
Importing code_for_hw03
Imported tidy_plot, plot_separator, plot_data, plot_nonlin_sep, cv, rv, y, positive, score
Datasets: super_simple_separable_through_origin(), super_simple_separable(), xor(), xor_more()
Tests for part 2: test_linear_classifier_with_features, mul, make_polynomial_feature_fun,
                  test_with_features
Also loaded: perceptron, one_hot_internal, test_one_hot
Importing code_for_hw03 (part 2, imported as hw3)
Imported tidy_plot, plot_separator, plot_data, plot_nonlin_sep, cv, rv, y, positive, score
         xval_learning_alg, eval_classifier
Tests: test_linear_classifier
Dataset tools: load_auto_data, std_vals, standard, raw, one_hot, auto_data_and_labels
               load_review_data, clean, extract_words, bag_of_words, extract_bow_feature_vectors
               load_mnist_data, load_mnist_single
```

In [3]:
```
help(tidy_plot)
```

```
Help on function tidy_plot in module code_for_hw3_part1:

tidy_plot(xmin, xmax, ymin, ymax, center=False, title=None, xlabel=None, ylabel=None)
```

Feature Transformation

Running Perceptron

In problems 1,2 and 3, you will have to run the Perceptron algorithm several times to obtain linear classifiers. We provide you with an implementation of the algorithm which you can use to obtain your results.

The specifications for the `perceptron` method provided are:

- `data` is a numpy array of dimension d by n
- `labels` is numpy array of dimension 1 by n
- `params` is a dictionary specifying extra parameters to this algorithm; your algorithm runs a number of iterations equal to T
- `hook` is either None or a function that takes the tuple `(th, th0)` as an argument and displays the separator graphically.

It should return a tuple of θ (a d by 1 array) and θ_0 (a 1 by 1 array).

Note that you are free to modify the method. For example, a useful modification for this homework would be to make the method return the number of mistakes made on the input data, while it runs.

In [4]:

```
# Perceptron algorithm with offset.
# data is dimension d by n
# labels is dimension 1 by n
# T is a positive integer number of steps to run
def perceptron(data, labels, params = {}, hook = None):
    # if T not in params, default to 100
    T = params.get('T', 50)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    for t in range(T):
        for i in range(n):
            x = data[:,i:i+1]
            y = labels[:,i:i+1]
            if y * positive(x, theta, theta_0) <= 0.0:
                theta = theta + y * x
                theta_0 = theta_0 + y
                if hook: hook((theta, theta_0))
    return theta, theta_0

def averaged_perceptron(data, labels, params = {}, hook = None):
    T = params.get('T', 50)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    theta_sum = theta.copy()
    theta_0_sum = theta_0.copy()
    for t in range(T):
        for i in range(n):
            x = data[:,i:i+1]
            y = labels[:,i:i+1]
            if y * positive(x, theta, theta_0) <= 0.0:
                theta = theta + y * x
                theta_0 = theta_0 + y
                if hook: hook((theta, theta_0))
            theta_sum = theta_sum + theta
            theta_0_sum = theta_0_sum + theta_0
    theta_avg = theta_sum / (T*n)
    theta_0_avg = theta_0_sum / (T*n)
    if hook: hook((theta_avg, theta_0_avg))
    return theta_avg, theta_0_avg


def eval_classifier(learner, data_train, labels_train, data_test, labels_test):
    th, th0 = learner(data_train, labels_train)
    return score(data_test, labels_test, th, th0)/data_test.shape[1]

def positive(x, th, th0):
```

```
    return np.sign(th.T@x + th0)

def score(data, labels, th, th0):
    return np.sum(positive(data, th, th0) == labels)

def xval_learning_alg(learner, data, labels, k):
    _, n = data.shape
    idx = list(range(n))
    np.random.seed(0)
    np.random.shuffle(idx)
    data, labels = data[:,idx], labels[:,idx]

    score_sum = 0
    s_data = np.array_split(data, k, axis=1)
    s_labels = np.array_split(labels, k, axis=1)
    for i in range(k):
        data_train = np.concatenate(s_data[:i] + s_data[i+1:], axis=1)
        labels_train = np.concatenate(s_labels[:i] + s_labels[i+1:], axis=1)
        data_test = np.array(s_data[i])
        labels_test = np.array(s_labels[i])
        score_sum += eval_classifier(learner, data_train, labels_train,
                                             data_test, labels_test)
    return score_sum/k
```

In [ ]:
```
perceptron(data, labels, params = {'T':100}, hook = None)
```

In [41]:
```
def my_perceptron(data, labels, params={}, hook=None):
    # if T not in params, default to 100
    T = params.get('T', 100)
    d = data.shape[0]
    n = data.shape[1]
    theta = np.transpose([[0.0] * d])
    theta_0 = 0.0
    #ax = plot_data(data, labels)
    num = 0
    for test in range(T):
        founderror = False
        for i in range(n):
            xi = data[:,i:i+1]
            yi = labels[0, i]
            if yi * np.sign(theta.T @ xi + theta_0) <= 0:
                num += 1
                founderror = True
                theta += yi * xi
                theta_0 += yi
                if hook:
                    hook(theta, theta_0)
        if not founderror:
            break
    #plot_separator(ax=ax, th=theta, th_0=theta_0)
    return (theta, np.array([[theta_0]]), num)
```

In [62]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8],
    [1, 1, 1, 1]
])
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1, -0.5]])

gamma = np.abs(np.min(labels.T * (th @ data) / np.linalg.norm(th) ))
r = np.max(np.linalg.norm(data, axis=0))
r, gamma, (r / gamma)**2
```

Out[62]:
```
(800.0010249993434, 0.2683281572999748, 8888911.666666666)
```

In [43]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8]
])
labels = np.array([[-1, -1, 1, 1]])
```

In [44]:
```
perceptron(data, labels, params={'T':1000})
```

Out[44]:
```
(array([[  0.],
        [600.]]),
 array([[0.]]),
 2000)
```

In [45]:
```
perceptron(data, labels, params={'T':10000})
```

Out[45]:
```
(array([[   0.],
        [6000.]]),
 array([[0.]]),
 20000)
```

In [46]:
```
perceptron(data, labels, params={'T':100000})
```

Out[46]:
```
(array([[  600.        ],
        [69997.80000031]]),
 array([[0.]]),
 233326)
```

In [47]:
```
perceptron(data, labels, params={'T':1000000})
```

Out[47]:
```
(array([[-2.000000e+02],
        [ 2.000068e+05]]),
 array([[-4.]]),
 666696)
```

In [63]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8],
    [1, 1, 1, 1]
])
data[0:2] *= 0.001
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1, -0.0005]])

gamma = np.abs(np.min(labels.T * (th @ data) / np.linalg.norm(th) ))

data, gamma
r = np.max(np.linalg.norm(data, axis=0))
r, gamma, (r / gamma)**2
```

Out[63]:
```
(1.2806250973645645, 0.00029999996250000706, 18222233.88889067)
```

In [64]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8],
    [1, 1, 1, 1]
])
data[0:1] *= 0.001
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1, -0.5]])

gamma = np.abs(np.min(labels.T * (th @ data) / np.linalg.norm(th) ))

data, gamma
r = np.max(np.linalg.norm(data, axis=0))
r, gamma, (r / gamma)**2
```

Out[64]:
```
(1.50996688705415, 0.2683281572999748, 31.666666666666664)
```

In [65]:
```
perceptron(data, labels, params={'T':1000000})
```

Out[65]:
```
(array([[-0.2],
        [ 2.8],
        [-1. ]]),
 array([[-1.]]),
 11)
```

2D) Encoding Discrete Values

It is common to encode sets of discrete values, for machine learning, not as a single multi-valued feature, but using a one hot encoding. So, if there are k values in the discrete set, we would transform that single multi-valued feature into k binary-valued features, in which feature i has value $+1$ if the original feature value was i and has value 0 (or -1) otherwise.

Write a function `one_hot` that takes as input x, a single feature value (between 1 and k), and k, the total possible number of values this feature can take on, and transform it to a numpy column vector of k binary features using a one-hot encoding (remember vectors have zero-based indexing).

In [71]:
```
data = np.array([[2, 3,  4,  5]])
labels = np.array([[1, 1, -1, -1]])
th, th0 = perceptron(data, labels)

th, th0, labels * (th @ data + th0), th @ np.array([[1, 6]]) + th0
```

Out[71]:
```
(array([[-2.]]), array([[7.]]), array([[3., 1., 1., 3.]]), array([[ 5., -5.]]))
```

In [84]:
```
a = np.zeros((1, 10))
a[:, 1] = 1
a
```

Out[84]:
```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [8]:
```
def one_hot(x, k):
    a = np.zeros((k, 1))
    a[x-1,0] = 1.0
    return a
```

In [9]:
```
test_one_hot(one_hot)
```

```
Passed!
```

In [16]:
```
data =   [[2, 3,  4,  5]]
data[0]
```

Out[16]:
```
[2, 3, 4, 5]
```

In [47]:
```
data =   np.array([[2, 3,  4,  5]])
labels = np.array([[1, 1, -1, -1]])
data = np.concatenate(
    [one_hot(e, 6) for e in data[0]],
    axis=1
)
data, labels
```

Out[47]:
```
(array([[0., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.],
        [0., 0., 0., 0.]]),
 array([[ 1,  1, -1, -1]]))
```

In [49]:
```
th, th0 = perceptron(data, labels)
th.T, th0
```

Out[49]:
```
(array([[ 0.,  2.,  1., -2., -1.,  0.]]), array([[0.]]))
```

In [71]:
```
samsung = one_hot(1, 6)
nokia = one_hot(6, 6)
samsung, nokia, th.T
```

Out[71]:
```
(array([[1.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]]),
 array([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [1.]]),
 array([[ 0.,  2.,  1., -2., -1.,  0.]]))
```

In [70]:
```
[ th.T @ xi + th0 for xi in (samsung, nokia)]
```

Out[70]:
```
[array([[0.]]), array([[0.]])]
```

In [72]:
```
[ (th.T @ xi + th0) / np.linalg.norm(th) for xi in (samsung, nokia)]
```

Out[72]:
```
[array([[0.]]), array([[0.]])]
```

In [75]:
```
data =   [[1, 2, 3, 4, 5, 6]]
labels = np.array([[1, 1, -1, -1, 1, 1]])

data = np.concatenate(
    [one_hot(e, 6) for e in data[0]],
    axis=1
)
data, labels
```

Out[75]:
```
(array([[1., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 1.]]),
 array([[ 1,  1, -1, -1,  1,  1]]))
```

In [76]:
```
th, th0 = perceptron(data, labels)
th.T, th0
```

Out[76]:
```
(array([[ 1.,  1., -2., -2.,  1.,  1.]]), array([[0.]]))
```

3) Polynomial Features

One systematic way of generating non-linear transformations of your input features is to consider the polynomials of increasing order. Given a feature vector $x = [x_1, x_2, ..., x_d]^T$, we can map it into a new feature vector that contains all the factors in a polynomial of order d. For example, for $x = [x_1, x_2]^T$ and order 2, we get $$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]^T$$ and for order 3, we get $$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1^2x_2, x_1x_2^2, x_1^3, x_2^3]^T.$$

In the code that has been loaded, we have defined `make_polynomial_feature_fun` that, given the order, returns a feature transformation function (analogous to ϕ in the description). You should use it in doing this problem.

In [77]:
```
## For example, make_polynomial_feature_fun could be used as follows:
import numpy as np

# Data
data = np.zeros((5,1))

# Generate transformation of order 2
transformation = make_polynomial_feature_fun(2)

# Use transformation on data
print(transformation(data))
```

```
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

In [79]:
```
#  Enter a list of 6 integers indicating the number of polynomial features of degrees [1, 10, 20, 30, 40, 50] for a 2-dimension
# (x1, x2)
# 1: 2
# 10:
# 1, 1
# 2 11 2
# 3 21 12 3
# 4 31 22 13 4
# 5 41 32 23 14 5
# 6 51 42 33 24 15 6
# ..
# 10 91 82 73 64 55 46 37 28 19 10
#     1 + 2 + 3 + 4 + ..
# 20: 2 + .. + 21 = 23 * 20 / 2 = 230
# 30: 2 + .. + 31 = 33 * 30 / 2 = 330 + 165 = 495
# 40: 43 * 40 / 2 = 860
# 50: 53 * 50 / 2 = 5300 / 4 = 1325
# 2, 65, 230, 495, 860, 1325 ; +1
[1 + (n+3) * n // 2 for n in (10, 20, 30, 40, 50)]
```

Out[79]:
```
[66, 231, 496, 861, 1326]
```

Note that iterative animations, which update a plot within a loop, don't work the same way in colab, as with a local python console installation. One workaround for colab to be able to show such plot iterations is to show all the plots, and this can be done for the test code using this patched function:

In [162…

```
def test_linear_classifier_with_features(dataFun, learner, feature_fun,
                                draw = True, refresh = True, pause = True):
    raw_data, labels = dataFun()
    data = feature_fun(raw_data) if feature_fun else raw_data
    if draw:
        def hook(params):
            ax = plot_data(raw_data, labels)   # create plot axis on each iteration
            (th, th0) = params
            predictor = lambda x1,x2: int(positive(feature_fun(cv([x1, x2])), th, th0))
            plot_nonlin_sep(
                predictor,
                ax = ax)
            plot_data(raw_data, labels, ax)
            plt.show()                         # force plot to push to the colab notebook and be displayed
            print('th', th.T, 'th0', th0)
            if pause: input('press enter here to continue:')
    else:
        hook = None
    th, th0 = learner(data, labels, hook = hook)
    if hook: hook((th, th0))
    print("Final score", int(score(data, labels, th, th0)))
    print("Params", np.transpose(th), th0)

def test_with_features(dataFun, order = 2, draw=True, pause=True, learner=perceptron):
    test_linear_classifier_with_features(
        dataFun,                        # data
        learner,                      # learner
        make_polynomial_feature_fun(order), # feature maker
        draw=draw,
        pause=pause)
```

Here's a test you can run to see plots:

In [163…

```
def perceptron_with_params(T=100):
    myparams ={'T': T}
    def f(data, labels, params = {}, hook = None):
        return perceptron(data, labels, myparams, hook)
    return f
```

In [144…

```
print(super_simple_separable_through_origin()[0].shape)
test_with_features(super_simple_separable_through_origin, order=2, draw=False, pause=False)
```

```
(2, 4)
Final score 4
Params [[  2.   4.  17. -46.  59. 107.]] [[2.]]
```

In [143…
```
print(super_simple_separable()[0].shape)
test_with_features(super_simple_separable, order=2, draw=False, pause=False, learner=perceptron_with_params(T=1000))
```

```
(2, 4)
Final score 4
Params [[ -11.  -26.   11. -190.  140.  235.]] [[-11.]]
```

In [115…
```
print(xor()[0].shape)
test_with_features( xor , order=2, draw=False, pause=False, learner=perceptron_with_params(T=1000))
```

```
(2, 4)
Final score 4
Params [[ 1. -1. -1. -5. 11. -5.]] [[1.]]
```

In [165…
```
print(xor_more())
test_with_features( xor_more , order=3, draw=False, pause=False, learner=perceptron_with_params(T=10000))
```

```
(array([[1, 2, 1, 2, 2, 4, 1, 3],
       [1, 2, 2, 1, 3, 1, 3, 3]]), array([[ 1,  1, -1, -1,  1,  1, -1, -1]]))
2202
Final score 8
Params [[ -78.   28.  -39.   72.  248.  -19.   76. -522.  476. -153.]] [[-78.]]
```

We know that a better way to do this exists (eg using colab plot animations) - if you are willing to contribute some nice code which lets our plotting functions do this, please do share!

Experiments

4) Evaluating algorithmic and feature choices for AUTO data

We now want to build a classifier for the auto data, focusing on the numeric data. In the code file for this part of the assignment, we have supplied you with the `load_auto_data` function, that can be used to read the relevant .tsv file. It will return a list of dictionaries, one for each data item.

We then have to specify what feature function to use for each column in the data. The file `hw3_part2_main.py` has an example for constructing the data and label arrays using `raw` feature function for all the columns. Look at the definition of `features` in `hw3_part2_main.py`, this indicates a feature name to use and then a feature function, there are three defined in the `code_for_hw3_part2.py` file (`raw` , `standard` and `one_hot`). `raw` just uses the original value, `standard` subtracts out the mean value and divides by the standard deviation and `one_hot` does the encoding described in the notes.

The function `auto_data_and_labels` will process the dictionaries and return `data, labels` where `data` are arrays of dimension $(d, 392)$, with d the total number of features specified, and `labels` is of dimension $(1, 392)$. The data in the file is sorted by class, but it will be shuffled when you read it in.

In [166…

```
# Returns a list of dictionaries.  Keys are the column names, including mpg.
auto_data_all = hw3.load_auto_data('auto-mpg.tsv')

# The choice of feature processing for each feature, mpg is always raw and
# does not need to be specified.  Other choices are hw3.standard and hw3.one_hot.
# 'name' is not numeric and would need a different encoding.
features = [('cylinders', hw3.raw),
            ('displacement', hw3.raw),
            ('horsepower', hw3.raw),
            ('weight', hw3.raw),
            ('acceleration', hw3.raw),
            ## Drop model_year by default
            ## ('model_year', hw3.raw),
            ('origin', hw3.raw)]

# Construct the standard data and label arrays
auto_data, auto_labels = hw3.auto_data_and_labels(auto_data_all, features)
print('auto data and labels shape', auto_data.shape, auto_labels.shape)
```

```
avg and std {}
entries in one_hot field {}
auto data and labels shape (6, 392) (1, 392)
```

In [ ]:

In [172…

```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 1}),
    auto_data,
    auto_labels,
    10)
```

```
97
97
95
90
98
94
99
97
92
94
```

Out[172]: 0.6526282051282052

In [173…
```
hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 1}),
    auto_data,
    auto_labels,
    10)
```

Out[173]: 0.8441025641025641

In [174…
```
features2 = [('cylinders', hw3.one_hot),
             ('displacement', hw3.standard),
             ('horsepower', hw3.standard),
             ('weight', hw3.standard),
             ('acceleration', hw3.standard),
             ## Drop model_year by default
             ## ('model_year', hw3.raw),
             ('origin', hw3.one_hot)]

# Construct the standard data and label arrays
auto_data2, auto_labels2 = hw3.auto_data_and_labels(auto_data_all, features2)
print('auto data and labels shape', auto_data2.shape, auto_labels2.shape)
```

```
avg and std {'displacement': (388.3482142857143, 302.0458123396403), 'horsepower': (509.3545918367347, 333.6521151716361), 'wei
ght': (2977.5841836734694, 848.3184465698365), 'acceleration': (15.541326530612228, 2.7553429127509963)}
entries in one_hot field {'cylinders': [3.0, 4.0, 5.0, 6.0, 8.0], 'origin': [1.0, 2.0, 3.0]}
auto data and labels shape (12, 392) (1, 392)
```

In [188…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 1}),
    auto_data2,
    auto_labels2,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 1}),
    auto_data2,
    auto_labels2,
    10)
```

Out[188]: (0.7908333333333333, 0.9004487179487182)

In [182…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 10}),
    auto_data,
    auto_labels,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 10}),
    auto_data,
    auto_labels,
    10)
```

Out[182]: (0.7423076923076924, 0.8366025641025641)

In [177…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 10}),
    auto_data2,
    auto_labels2,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 10}),
    auto_data2,
    auto_labels2,
    10)
```

```
545
540
563
529
546
540
531
495
532
547
```

Out[177]: (0.8061538461538461, 0.8979487179487181)

In [183…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 50}),
    auto_data,
    auto_labels,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 50}),
    auto_data,
    auto_labels,
    10)
```

Out[183]: (0.6909615384615384, 0.8366025641025641)

In [187…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 50}),
    auto_data2,
    auto_labels2,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 50}),
    auto_data2,
    auto_labels2,
    10)
```

Out[187]:
```
(0.8060256410256409, 0.9005128205128207)
```

In [189…
```
th, th0 = averaged_perceptron(auto_data2, auto_labels2, params={'T':50})
th, th0
```

Out[189]:
```
(array([[-1.98173469],
        [ 0.34622449],
        [ 0.51530612],
        [-0.95596939],
        [ 2.80469388],
        [-1.46206452],
        [ 0.27203955],
        [-6.55860703],
        [ 0.83288456],
        [-0.10352041],
        [ 1.1647449 ],
        [-0.33270408]]),
 array([[0.72852041]]))
```

5) Evaluating algorithmic and feature choices for review data

We have supplied you with the `load_review_data` function, that can be used to read a .tsv file and return the labels and texts. We have also supplied you with the `bag_of_words` function, which takes the raw data and returns a dictionary of unigram words. The resulting dictionary is an input to `extract_bow_feature_vectors` which computes a feature matrix of ones and zeros that can be used as the input for the classification algorithms. The file `hw3_part2_main.py` has code for constructing the data and label arrays. Using these arrays and our implementation of the learning algorithms, you will be able to compute θ and θ_0. You will need to add your (or the one written by staff) implementation of perceptron and averaged perceptron.

In [5]:
```
# Returns lists of dictionaries.  Keys are the column names, 'sentiment' and 'text'.
# The train data has 10,000 examples
review_data = hw3.load_review_data('reviews.tsv')

# Lists texts of reviews and list of labels (1 or -1)
review_texts, review_label_list = zip(*((sample['text'], sample['sentiment']) for sample in review_data))

# The dictionary of all the words for "bag of words"
dictionary = hw3.bag_of_words(review_texts)

# The standard data arrays for the bag of words
review_bow_data = hw3.extract_bow_feature_vectors(review_texts, dictionary)
review_labels = hw3.rv(review_label_list)
print('review_bow_data and labels shape', review_bow_data.shape, review_labels.shape)
```

```
review_bow_data and labels shape (19945, 10000) (1, 10000)
```

In [6]:
```
import time
```

In [7]:
```
for T in (1, 10, 50):
    print('T', T)
    start = time.time()
    print('P', xval_learning_alg(
        lambda data, labels: perceptron(data, labels, {"T": T}),
        review_bow_data, review_labels, 10))
    print(time.time() - start)
    start = time.time()
    print('AP', xval_learning_alg(
        lambda data, labels: averaged_perceptron(data, labels, {"T": T}),
        review_bow_data, review_labels, 10))
    print(time.time() - start)
```

```
T 1
P 0.7672000000000001
7.181673765182495
AP 0.8120999999999998
7.737758159637451
T 10
P 0.7871
31.119680643081665
AP 0.8237
38.83753275871277
T 50
P 0.8036
130.36860251426697
AP 0.8157
179.88108468055725
```

In [14]:
```
th, th0 = averaged_perceptron( review_bow_data, review_labels, {"T": 10})
th, th0
```

Out[14]:
```
(array([[ 0.15984],
        [-2.74048],
        [-1.23668],
        ...,
        [ 0.     ],
        [-1.2001 ],
        [ 0.     ]]),
 array([[-1.72795]]))
```

In [16]:
```
mysorted = sorted((e, i) for i,e in enumerate(th))
rdict = hw3.reverse_dict(dictionary)
[rdict[i] for (e, i) in mysorted[:10]], [rdict[i] for (e, i) in mysorted[-10:]]
```

Out[16]:
```
(['worst',
  'awful',
  'unfortunately',
  'horrible',
  'stuck',
  'changed',
  'disappointment',
  'bland',
  'poor',
  'formula'],
 ['great',
  'individually',
  'bright',
  'yummy',
  'skeptical',
  'perfect',
  'easily',
  'satisfied',
  'delicious',
  'excellent'])
```

6) Evaluating features for MNIST data

This problem explores how well the perceptron algorithm works to classify images of handwritten digits, from the well-known ("MNIST") dataset, buiding on your thoughts from lab about extracting features from images. This exercise will highlight how important feature extraction is, before linear classification is done, using algorithms such as the perceptron.

Dataset setup

Often, it may be easier to work with a vector whose spatial orientation is preserved. In previous parts, we have represented features as one long feature vector. For images, however, we often represent a m by n image as a `(m,n)` array, rather than a `(mn,1)` array (as the previous parts have done).

In the code file, we have supplied you with the `load_mnist_data` function, which will read from the provided image files and populate a dictionary, with image and label vectors for each numerical digit from 0 to 9. These images are already shaped as `(m,n)` arrays.

In [55]:
```
mnist_data_all = hw3.load_mnist_data(range(10))

print('mnist_data_all loaded. shape of single images is', mnist_data_all[0]["images"][0].shape)

# HINT: change the [0] and [1] if you want to access different images
def get_data_lables(leftd, rightd):
    d0 = mnist_data_all[leftd]["images"]
    d1 = mnist_data_all[rightd]["images"]
    y0 = np.repeat(-1, len(d0)).reshape(1,-1)
    y1 = np.repeat(1, len(d1)).reshape(1,-1)
    # data goes into the feature computation functions
    data = np.vstack((d0, d1))
    # labels can directly go into the perceptron algorithm
    labels = np.vstack((y0.T, y1.T)).T
    return data, labels
```

```
mnist_data_all loaded. shape of single images is (28, 28)
```

In [15]:
```
np.array([np.average([[1, 2, 3], [4, 5, 6], [7, 8, 9]], axis=0)]).T
```

Out[15]:
```
array([[4.],
       [5.],
       [6.]])
```

In [104…
```
arr = np.arange(24).reshape(2,3,4)
print(arr.ndim)
print(arr)
out_ = np.array([
    np.apply_along_axis(
        lambda a: np.average(a),
        axis=0,
        arr=item
    )
    for item
    in arr
])
out_
```

```
3
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Out[104]:
```
array([[ 4.,  5.,  6.,  7.],
       [16., 17., 18., 19.]])
```

In [174…

```python
# change these implementations to support whole datasets

def raw_mnist_features(x):
    """
    @param x (n_samples,m,n) array with values in (0,1)
    @return (m*n,n_samples) reshaped array where each entry is preserved
    """
    n_samples, m, n = x.shape
    return x.reshape((n_samples, n*m, 1))


def row_average_features(x):
    """
    This should either use or modify your code from the tutor questions.

    @param x (n_samples,m,n) array with values in (0,1)
    @return (m,n_samples) array where each entry is the average of a row
    """
    return np.apply_along_axis(
        lambda a: [np.average(a)],
        axis=1,
        arr=x
    )

def col_average_features(x):
    """
    This should either use or modify your code from the tutor questions.

    @param x (n_samples,m,n) array with values in (0,1)
    @return (n,n_samples) array where each entry is the average of a column
    """
    return np.apply_along_axis(
        lambda a: [np.average(a)],
        axis=0,
        arr=x
    ).T

def top_bottom_features(x):
    """
    This should either use or modify your code from the tutor questions.

    @param x (n_samples,m,n) array with values in (0,1)
    @return (2,n_samples) array where the first entry of each column is the average of the
    top half of the image = rows 0 to floor(m/2) [exclusive]
    and the second entry is the average of the bottom half of the image
    = rows floor(m/2) [inclusive] to m
    """
    n, m = x.shape
```

```
    return cv([np.average(x[0:n//2,:]), np.average(a=x[n//2:,:])])
```

In [175…
```
#Your Code Here
ans=row_average_features(np.array([[1,2,3],[3,9,2]])).tolist()
ans
```

Out[175]:
```
[[2.0], [4.666666666666667]]
```

In [176…
```
ans=col_average_features(np.array([[1,2,3],[3,9,2],[2,1,9]])).tolist()
ans
```

Out[176]:
```
[[2.0], [4.0], [4.666666666666667]]
```

In [177…
```
top_bottom_features(np.arange(12).reshape((3,4))), np.arange(12).reshape((3,4))
```

Out[177]:
```
(array([[1.5],
        [7.5]]),
 array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]))
```

In [57]:
```
import time
# use this function to evaluate accuracy
#print( data.shape, raw_mnist_features(data).shape, raw_mnist_features(data).T[0].shape )

for l, r in ((0, 1), (2, 4), (6, 8), (9, 0)):
    start = time.time()

    data, labels = get_data_lables(l, r)
    raw_data = raw_mnist_features(data).T[0]
    acc = hw3.get_classification_accuracy(raw_data, labels)
    print('Done', time.time() - start)
    print('left', l)
    print('right', r)
    print(acc)
```

```
Done 0.46145176887512207
left 0
right 1
0.975
Done 0.47538232803344727
left 2
right 4
0.8641666666666665
Done 0.4565155506134033
left 6
right 8
0.9479166666666667
Done 0.500361442565918
left 9
right 0
0.6470833333333333
```

In [ ]:
```
(0.975, 0.8641666666666665, 0.9479166666666667, 0.6470833333333333)
```

In [178…

```
data, labels = get_data_lables(0, 1)
print(data.shape)
rfd = np.concatenate(
    list(
        row_average_features(xi).T
        for xi
        in data
    )
).T
cfd = np.concatenate(
    list(
        col_average_features(xi).T
        for xi
        in data
    )
).T
tbfd = np.concatenate(
    list(
        top_bottom_features(xi).T
        for xi
        in data
    )
).T

print(rfd.shape)
print(cfd.shape)
print(tbfd.shape)

np.concatenate(
    (rfd, cfd, tbfd),
    axis=0
).shape
```

```
(160, 28, 28)
(28, 160)
(28, 160)
(2, 160)
```

Out[178]: (58, 160)

In [183…

```
for l, r in ((0, 1), (2, 4), (6, 8), (9, 0)):
    data, labels = get_data_lables(l, r)
    rfd = np.concatenate(
        list(
            row_average_features(xi).T
            for xi
            in data
        )
    ).T
    cfd = np.concatenate(
        list(
            col_average_features(xi).T
            for xi
            in data
        )
    ).T
    tbfd = np.concatenate(
        list(
            top_bottom_features(xi).T
            for xi
            in data
        )
    ).T
    res = []
    for fdata in (rfd, cfd, tbfd):
        acc = hw3.get_classification_accuracy(fdata, labels)
        res += [acc]
    print(repr(res))
```

```
[0.48125, 0.6375, 0.48125]
[0.7754166666666668, 0.49749999999999994, 0.49749999999999994]
[0.92125, 0.52125, 0.5650000000000001]
[0.49749999999999994, 0.5041666666666667, 0.49749999999999994]
```

In [ ]:

```
0.48125,
0.6375,
0.48125
```

> 6.2F) (Optional) What does it mean if a binary classification accuracy is below 0.5, if your dataset is balanced (same number from each class)? Are these datasets balanced?

Means it is worse than randomly picking up labels.

In [188…

```
# 6.2G) (Optional) Feel free to classify other images from each other. Which combinations perform the best, and which perform t

res = []
for l in range(10):
    for r in range(10):
        if l == r:
            continue
        start = time.time()
        data, labels = get_data_lables(l, r)
        raw_data = raw_mnist_features(data).T[0]
        acc = hw3.get_classification_accuracy(raw_data, labels)
        res += [(acc, l, r)]
sres = sorted(res)
print(sres[:10])
print(sres[-10:])
```

```
[(0.48250000000000004, 5, 8), (0.4841666666666667, 4, 9), (0.5079166666666667, 4, 6), (0.525, 1, 8), (0.5429166666666666, 9,
8), (0.5487500000000001, 8, 5), (0.575, 5, 3), (0.5758333333333334, 3, 5), (0.5912499999999999, 7, 3), (0.5954166666666667, 0,
5)]
[(0.9737500000000001, 9, 2), (0.975, 0, 1), (0.975, 1, 0), (0.975, 9, 1), (0.9808333333333333, 7, 5), (0.98125, 1, 4), (0.9812
5, 5, 7), (0.9875, 4, 1), (0.9875, 4, 3), (0.99375, 8, 0)]
```

In [194…

```
print('Best 10:')
print('\n'.join('{} vs {}: {}'.format(l, r, a) for a,l,r in reversed(sres[-10:])))
print('Worst 10:')
print('\n'.join('{} vs {}: {}'.format(l, r, a) for a,l,r in sres[:10]))
```

```
Best 10:
8 vs 0: 0.99375
4 vs 3: 0.9875
4 vs 1: 0.9875
5 vs 7: 0.98125
1 vs 4: 0.98125
7 vs 5: 0.9808333333333333
9 vs 1: 0.975
1 vs 0: 0.975
0 vs 1: 0.975
9 vs 2: 0.9737500000000001
Worst 10:
5 vs 8: 0.48250000000000004
4 vs 9: 0.4841666666666667
4 vs 6: 0.5079166666666667
1 vs 8: 0.525
9 vs 8: 0.5429166666666666
8 vs 5: 0.5487500000000001
5 vs 3: 0.575
3 vs 5: 0.5758333333333334
7 vs 3: 0.5912499999999999
0 vs 5: 0.5954166666666667
```

In [ ]:

These exercises will prepare you for understanding how to maximize margins, as discussed in the lecture notes. You may want to review the definition of the margin γ.

1) Margin definition

Recall that the signed distance to a point x from a hyperplane θ, θ_0 is $sd(x, \theta, \theta_0) = \frac{\theta^T x + \theta_0}{\|\theta\|}$.

Ex1a:

You start with a hyperplane θ, θ_0 and a point x. Suppose a new separator is given, where $\hat{\theta} = -\theta$ and $\hat{\theta}_0 = -\theta_0$.

Which of the following is true? the signed distance changes sign but not magnitude

Submit View Answer 100.00%

You have 2 submissions remaining.

Ex1b:

You start with a hyperplane θ, θ_0 and a point x. Suppose a new separator is given, where $\hat{\theta} = \theta$ and $\hat{\theta}_0 = -\theta_0$.

Which of the following is true: both the sign and the magnitude may change

Submit View Answer 100.00%

You have 2 submissions remaining.

Ex1c:

The margin of labeled point x, y with respect to separator θ, θ_0 is:

$$\gamma(x, y, \theta, \theta_0) = \frac{y(\theta^T x + \theta_0)}{\|\theta\|}$$

Let `sd` stand for $sd(x, \theta, \theta_0)$, the signed distance from the separator to x. Define the margin in terms of `sd` and `y`, the label of x. Note that both of these are scalars. Provide an expression in Python syntax.

$\gamma(x, y, \theta, \theta_0) =$ y * sd

Check Syntax | Submit | View Answer | 100.00%

You have infinitely many submissions remaining.

Ex1d:

What is the sign of the signed distance when the prediction is incorrect?

Which of the following is true: could be either

Submit | View Answer | 100.00%

You have 1 submission remaining.

Ex1e:

What is the sign of the margin when the prediction is incorrect?

Which of the following is true: could be either

0.00%

You have 0 submissions remaining.

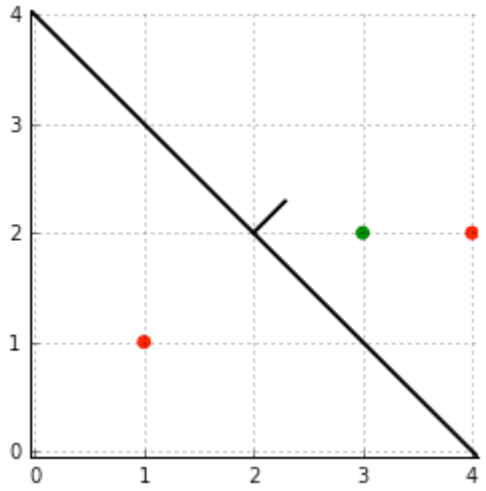
Solution: negative

Explanation:

The prediction is on the wrong side of the classifier, so the margin is negative.

2) Margin practice

What are the margins of the labeled points (x,y) = ((3, 2), +1), ((1, 1), -1), and ((4, 2), -1) with respect to the separator defined by $\theta = (1, 1)$, $\theta_0 = -4$? The situation is illustrated in the figure below.



Enter the three margins in order as a Python list of three numbers. Note that you can enter sqrt(x) as x**0.5 in Python.

[0.7071067811865475, 1.414213562373095, -1.414213562373095]

Submit View Answer **100.00%**

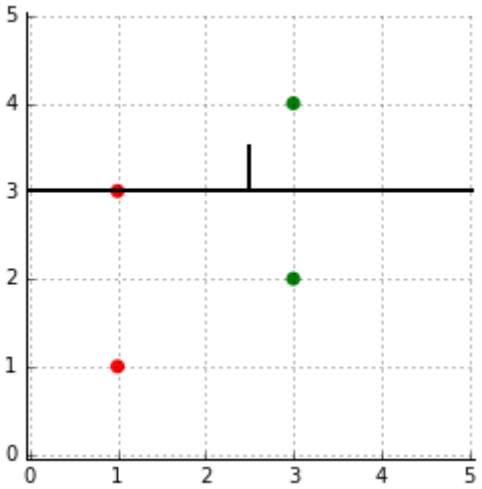
You have infinitely many submissions remaining.

3) Max Margin Separator

Consider the four points and separator:

```
data = np.array([[1, 1, 3, 3],[3, 1, 4, 2]])
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1]]).T
th0 = -3
```

The situation is shown below:



Ex3a:

Enter the four margins in order as a Python list of four numbers. [0., 2., 1., -1.]

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

Ex3b:

A maximum margin separator is a separator that maximizes the minimum margin between that separator and all points in the dataset.

Enter θ and θ_0 for a maximum margin separator as a Python list of three numbers.

[1, 0, -2]

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

Ex3c:

If you scaled this separator by a positive constant k (i.e., replace θ by $k\theta$, and θ_0 by $k\theta_0$), would it still be a maximum margin separator? Yes

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

Gradient descent

This week's lab and homework explore the concept of machine learning as optimization, building on the lecture and lecture notes on margin maximization and gradient descent.

Group information

1) Explore gradient descent

We've established that machine learning problems can be posed as optimization problems. We begin by studying general strategies for finding the minimum of a function. In general, unless the function is convex, it may be computationally difficult to find its global minimum.

Note: A function is convex if the line segment between any two points on the graph of the function lies above or on the graph.

We will sometimes study convex objectives, but in other cases we will content ourselves with finding a local minimum (where the gradient is zero) which may not be a global minimum. One method to find a local minimum of a function is called gradient descent (there are better ways, but this one is simple and computationally efficient in high dimensions and with lots of data). The idea is that we start with an initial guess, x_0, and move "downhill" in the direction of the gradient, leading to an update step

$$x^{(1)} = x^{(0)} - \eta \nabla_x f(x^{(0)})$$

where η is a "step size" parameter with the constraint that $\eta > 0$. We continue updating until $x^{(i+1)}$ does not differ too much from $x^{(i)}$. This approach is guaranteed to find the minimum if the function is convex and η is sufficiently small.

The questions below are concerned with running gradient descent on the parabola

$$f(x) = (2x + 3)^2 \ .$$

1A) Formulate the update rule that will be executed on every step when performing gradient descent on $f(x)$. You may use `eta` and `x` in your Python expression, where `eta` represents η.

x ← x - eta * 4 *(2*x + 3)

Check Syntax | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1B)

What is the optimal value of x that minimizes f? -1.5

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1C) This question asks you to explore convergence of gradient descent for our $f(x)$, to see how the step size affects the rate of convergence and whether there are oscillations or lack of convergence.

We implement minimization of $f(x)$ using a function `t1`, which runs gradient descent for the minimization of $f(x)$. We halt the algorithm when the value of x changes by less than 10^{-5}. In general, we may use some small tolerance such as this to say whether gradient descent has *converged*. Conversely, *divergence* is defined as being when x will not converge to a single finite value, even with an infinite number of updates.

When you click Submit, the tutor question generates a plot of $f(x)$ in blue and the history of x values you have tried in red with a blue 'x' at the initial x value. You can change the values for `step_size` or `init_val` and click Submit again (all submits get 100%).

Experiment with the following step sizes: [0.01, 0.1, 0.2, 0.3]. For which one(s) does $x^{(k)}$ converge without oscillation? For which one(s) does $x^{(k)}$ diverge?

```
def run():
    return t1(step_size= 0.1, init_val = 0)

```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

Why do oscillations happen for some values of `step_size` and not others? To gain insight into this behavior (and to understand how gradient descent proceeds), we note that we can rewrite our update rule:

$$x^{(k+1)} = x^{(k)} - \eta \nabla_x f(x^{(k)})$$

in the form:

$$z^{(k+1)} = \alpha z^{(k)}$$

for some $z = x - C$, and then consider how or if z approaches 0 for large k (or equivalently, x approaches C for large k). In particular, this formulation will soon be useful for us to analyze how the step-size η affects convergence.

1D) Show that the gradient descent update rule for our function $f(x) = (2x+3)^2$ can be written in the form:

$$x^{(k+1)} + 3/2 = (1 - 8\eta)(x^{(k)} + 3/2)$$

and equivalently in the form

$$z^{(k+1)} = \alpha z^{(k)}$$

by defining $z^{(k)}$ and α appropriately. What is $z^{(k)}$ in terms of $x^{(k)}$? What is α in terms of η?

Written in this form, we make the following key observation: **we can think of our gradient descent step as simply a multiplication of the previous value by α on each step.**

Inductively, we can see that the value of z (and x) at step k is related to the initial value as

$$z^{(k)} = \alpha^k z^{(0)}$$

and equivalently

$$x^{(k)} + 3/2 = (1 - 8\eta)^k (x^{(0)} + 3/2) \ .$$

(Note: if this is not clear, please feel free to join the help queue.)

The above equations are useful because they explain the relationship between the initial value of z (or x) and the output of gradient descent as the repeated multiplication by the same factor α on each step. From this, we can identify cases of

convergence, oscillation, and divergence by the following values on α:

$\alpha > 1$	**Gradient descent diverges without oscillation;** $z \to \infty$
$\alpha = 1$	$z^{(k)} = z^{(0)}$**, so no gradient descent steps occur**
$1 > \alpha \geq 0$	α^∞ **approaches 0, so gradient descent converges;** $z \to 0$
$0 > \alpha > -1$	α^∞ **approaches 0 while changing signs every step, so converges with some oscillation**
$\alpha = -1$	**At every step, the sign of** z **flips. Gradient descent oscillates between** $z^{(0)}$ **and** $-z^{(0)}$ **endlessly**
$-1 > \alpha$	**Gradient descent diverges with oscillation, since** z **grows but the sign of** z **flips at every step**

Since our ultimate goal is convergence, we are interested in the cases where $|\alpha| < 1$.

We can use the rules above about α to reason about how η affects convergence, given that $\alpha = 1 - 8\eta$.

Answer the following questions algebraically (using the expressions above).

1E)

What is the range of step size, as an interval, that causes x to converge to the global minimum starting from an arbitrary initial value? Use ( ) for open intervals and [ ] for closed intervals

(0, 0.25)

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

Does your algebraic answer agree with your numerical experiments above?

1F)

What is the largest step size that causes x to converge without oscillating?

1/8

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

How many iterations are needed for convergence in this case? Run `t1` with this value of step size.

1G)

For a step size of 0.1, is there an initial value of x_0 for which x does not converge to the global minimum? No

Submit View Answer Ask for Help **100.00%**

You have infinitely many submissions remaining.

1H) What value(s) of step size in the set $\{0.1, 0.11, 0.12, 0.13, 0.14, 0.15\}$ makes gradient descent take the most steps before convergence? Find the answer algebraically and enter your value(s) in a Python list. (Hint: think about the magnitude of $1 - 8\eta$ and how this might affect the rate of convergence).

Enter your answer as a python list: [0.1, 0.15]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Now try running `t1` with the above step sizes. Which ones are slowest? Which ones oscillate? Do these behaviors match with your algebraic results?

2) Where to meet?

A group of friends is planning to host a baby shower over the weekend. They want to find a location for the party that minimizes the sum of *squared* distances from their houses to the location of the party. Assume for now that they can host the party at any location in the town.

Assuming that the friends live in a 1-dimensional town, solve the following problems:

2A) Pose this problem as an (unconstrained) optimization problem. Assume there are n friends and the i-th friend is located at l_i. Denote the location of the party by p. What is the objective as a function of p? Write it down.

2B) Compute the gradient (write down/show your computation). Where is it zero?

2C) Which of the following is true?

- [x] There is necessarily a unique location that minimizes the objective function
- [ ] The optimization problem may have local minima that are not global minima
- [ ] The party can be always be hosted in one of the houses without loss of optimality
- [x] There is necessarily a choice of step size that makes gradient descent converge with oscillations for this problem

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Exercises

In [1]:
```
import numpy as np
# Exercies

xy = [
     ((3, 2), +1), ((1, 1), -1), ((4, 2), -1)
]
th = (1,1)
th0 =-4

for xi, yi in xy:
    print(xi,yi, yi* ( xi[0] * th[0] + xi[1] * th[1] + th0) / (th[0]**2+t
```

```
(3, 2) 1 0.7071067811865475
(1, 1) -1 1.414213562373095
(4, 2) -1 -1.414213562373095
```

In [ ]:
```
# 0.7071067811865475, 1.414213562373095, -1.414213562373095
```

In [7]:
```
data = np.array([[1, 1, 3, 3],[3, 1, 4, 2]])
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1]]).T
th0 = -3

labels * (th.T @ data + th0) / np.linalg.norm(th)
```

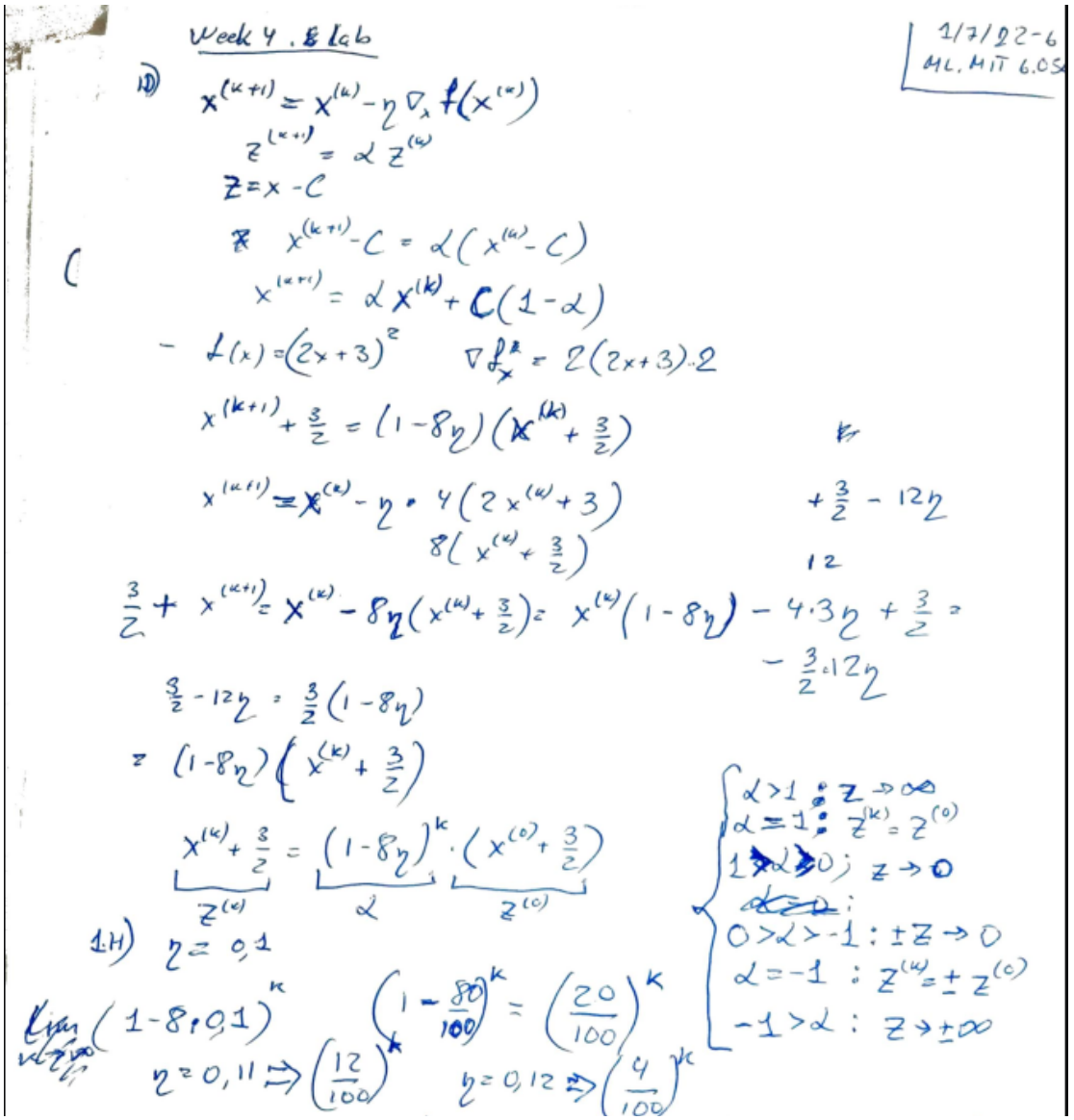
Out[7]:
```
array([[ 0.,  2.,  1., -1.]])
```

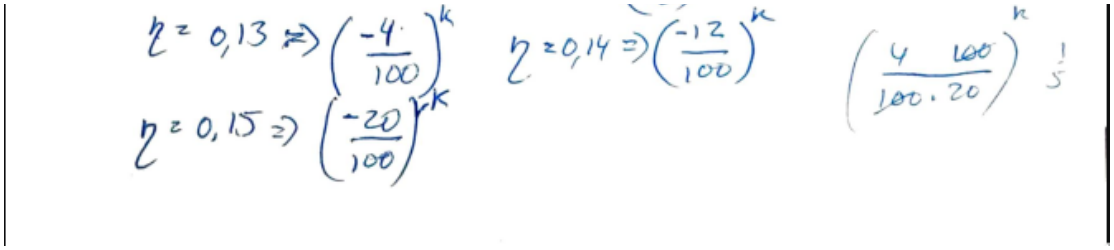
Lab

1)

In [ ]:
```
```

```
(2 x + 3) ** 2
2 * ( 2x + 3) * 2 = 0
x = -3/2

|1 - 8*e|  < 1
-1 < 1 - 8*e < 1
-2 < -8*e < 0
1/4 > e > 0

1 - 8*e = 0
1 - 8/8 = 0

x(k)+3/2=(1-8η)**k(x(0)+3/2).

1 - 8 * 10 / 100
(100 - 8 * 10 ) ^ k / 100 ^ k

1 - 8 * 11 / 100
1 - 8 * 12 / 100

x = 10..15
xk + 3/2 = (x0 + 3/2) * (1 - 8*x/100)^k
```

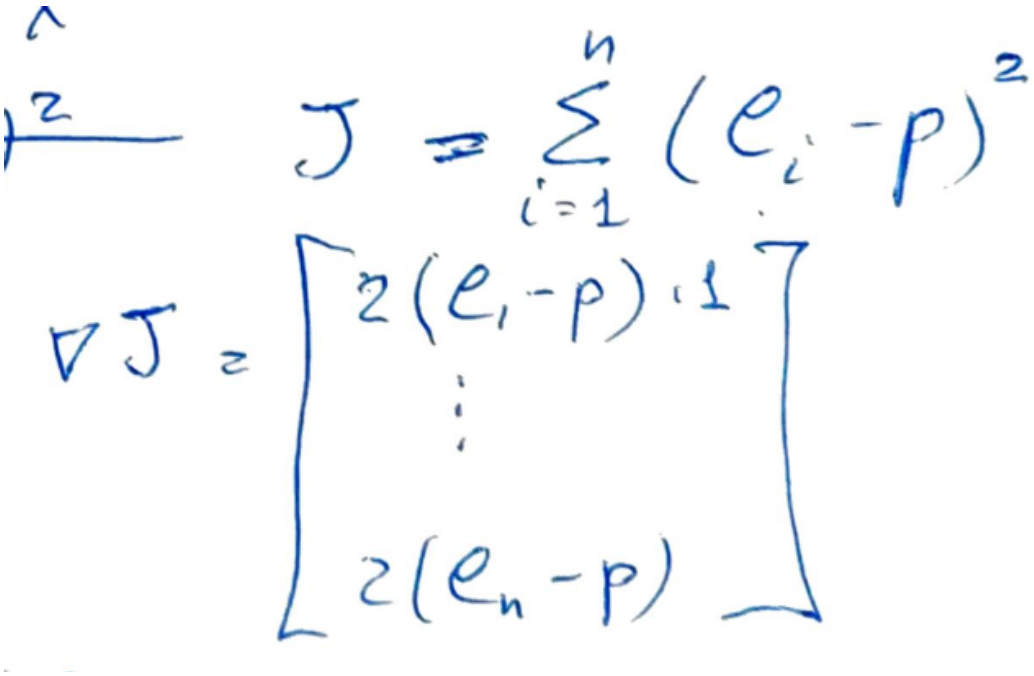




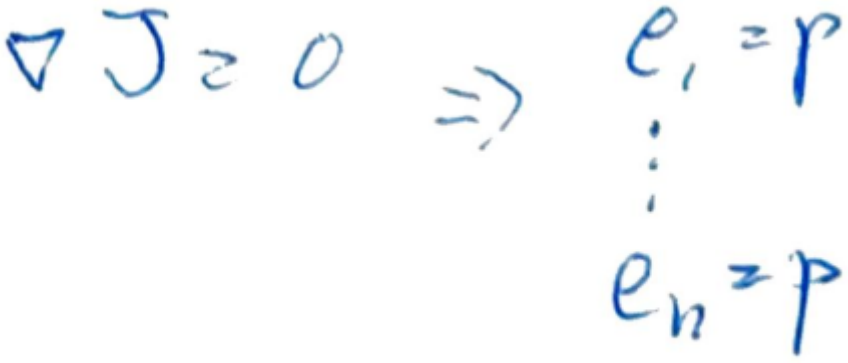
2) Where to meet?

> 2A) Pose this problem as an (unconstrained) optimization problem. Assume there are nnn friends and the iii-th friend is located at lil_ili. Denote the location of the party by ppp. What is the objective as a function of ppp? Write it down.

```
l_i - location of i-th friend
p - party location
```



> 2B) Compute the gradient (write down/show your computation). Where is it zero?



This homework does not provide Python code. Instead, we encourage you to write your own code to help you answer some of these problems, and/or to test and debug the code components we do ask for. Some of the problems below are simple enough that hand calculation should be possible; your hand solutions can serve as test cases for your code. You may also find that including utilities written in previous labs (like an `sd` or signed distance function) will be helpful, as you build up additional functions and utilities for calculation of margins, different loss functions, gradients, and other functions needed for margin maximization and gradient descent here.

For your convenience, we have copied the hands-on section into a colab notebook, which may be found here.

1) Margin

When we train a classifier, it is desirable for the classifier to have a large margin with regard to the points in our data set, in the hope that this will make the classifier more robust to any new points we might see.

We have previously defined the margin of a single example (a single data point) with respect to a separator, but that does not directly indicate whether a separator will perform well on a large data set. Thus, we would like to find a score function S for a separator (θ, θ_0), such that maximizing S leads to a better separator.

Marge Inovera suggests that because big margins are good, we should maximize the sum of the margins. So, she defines:

$$S_{sum}(\theta, \theta_0) = \sum_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

Minnie Malle suggests that it would be better to just worry about the points closest to the margin, and defines:

$$S_{min}(\theta, \theta_0) = \min_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

Maxim Argent suggests:

$$S_{max}(\theta, \theta_0) = \max_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

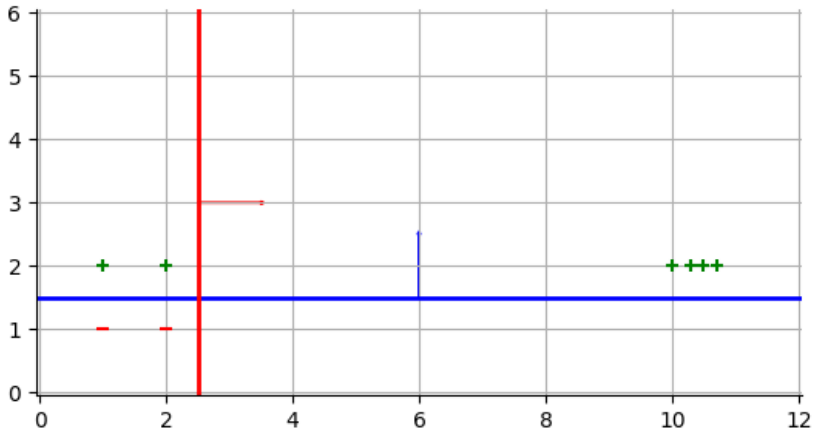
Recall that the margin of a given point is defined as

$$\gamma(x, y, \theta, \theta_0) = \frac{y(\theta \cdot x + \theta_0)}{||\theta||}.$$

Consider the following data, and two potential separators (red and blue).

```
data = np.array([[1, 2, 1, 2, 10, 10.3, 10.5, 10.7],
                 [1, 1, 2, 2,  2,  2,  2, 2]])
labels = np.array([[-1, -1, 1, 1, 1, 1, 1, 1]])
blue_th = np.array([[0, 1]]).T
blue_th0 = -1.5
red_th = np.array([[1, 0]]).T
red_th0 = -2.5
```

The situation is illustrated in the figure below.



1A) What are the values of each score $(S_{sum}, S_{min}, S_{max})$ on the red separator?

Enter a Python list of three numbers. [31.5, -1.5, 8.2]

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1B) What are the values of each score $(S_{sum}, S_{min}, S_{max})$ on the blue separator?

Enter a Python list of three numbers. [4.0, 0.5, 0.5]

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1C) Which of these separators maximizes S_{sum}?

Choose one: red

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1D) Which separator maximizes S_{min}?

Choose one: blue

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1E) Which separator maximizes S_{max}?

Choose one: red

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1F) Which score function should we prefer if our goal is to find a separator that generalizes better to new data?

Choose one: S_min

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

2) What a loss

Based on the previous part, we've decided to try to find a linear separator θ, θ_0 that **maximizes** the **minimum margin** (the distance between the separator and the points that come closest to it.) We define the margin of a data set (X, Y), with respect to a separator as

$$\gamma(X, Y, \theta, \theta_0) = \min_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

As discussed in the notes, an approach to this problem is to specify a value γ_{ref} for the margin of the data set, and then seek to find a linear separator that maximizes γ_{ref}.

2A) We can think about a (not necessarily maximal) margin γ_{ref} for the data set as a value such that:

- for at least one point $x^{(i)}, y^{(i)}$, we have $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \geq \gamma_{ref}$
- for every point $x^{(i)}, y^{(i)}$, we have $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \geq \gamma_{ref}$
- for at least one point $x^{(i)}, y^{(i)}$, we have $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \leq \gamma_{ref}$
- for every point $x^{(i)}, y^{(i)}$, we have $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \leq \gamma_{ref}$

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

2B) Suppose for our data set we find that the maximum γ_{ref} across all linear separators is 0.

Is our data linearly separable? No

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

2C) For this subproblem, assume that $\gamma_{ref} > 0$ (i.e., the data is linearly separable). Note that in this case, the Perceptron algorithm is guaranteed to find a separator that correctly classifies all of the data points.

What is the **largest minimum margin** guaranteed by running the Perceptron algorithm on a data set that has a maximum margin equal to $\gamma_{ref} > 0$?

- 0
- ∞
- γ_{ref}
- some ϵ where $\epsilon > 0$

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

Now we want to improve on the (infinitesimally small) guaranteed margin of the Perceptron algorithm. We saw in the lecture that a powerful way of designing learning algorithms is to describe them as optimization problems, then use relatively general-purpose optimization strategies to solve them.

A typical form of the optimization problem is to minimize an objective that has the form

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0)$$

where L is a per-point *loss function* that characterizes how much error was made by the hypothesis (θ, θ_0) on the point, and R is a *regularizer* that describes some prior knowledge or general preference over hypotheses.

We first consider the objective of finding a maximum-margin separator using the format above, using the so-called "zero-infinity" loss, $L_{0,\infty}$:

$$L_{0,\infty}(\gamma(x, y, \theta, \theta_0), \gamma_{ref}) = \begin{cases} \infty & \text{if } \gamma(x, y, \theta, \theta_0) < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

and

$$J_{0,\infty}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_{0,\infty}(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0).$$

2D) For a linearly separable data set, positive λ, and positive R given nonzero θ, what is true about the **minimal** value of $J_{0,\infty}$?

Which of the following is true: It is always finite and positive

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

2E) For a **non** linearly separable data set, and positive λ and γ_{ref}, what is true about the **minimal** value of $J_{0,\infty}$:

Which of the following is true: It is infinite

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

3) Simply inseparable

We would prefer a loss function that helps steer optimization toward a solution, in the case when the data is linearly separable. Furthermore, in real data sets it is relatively rare that the data is linearly separable, so our algorithm should be able to handle this case also and still work toward an optimal, though imperfect, linear separator. Instead of using $(0, \infty)$ loss, we should design a loss function that will let us "relax" the constraint that all of the points have margin bigger than γ_{ref}, while still encouraging large margins.

The *hinge loss* is one such more relaxed loss function; we will define it in a way that makes a connection to the problem we are facing:

$$L_h\left(\frac{\gamma(x,y,\theta,\theta_0)}{\gamma_{ref}}\right) = \begin{cases} 1 - \frac{\gamma(x,y,\theta,\theta_0)}{\gamma_{ref}} & \text{if } \gamma(x,y,\theta,\theta_0) < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

When the margin of the point is greater than or equal to γ_{ref}, we are happy and the loss is 0; while when the margin is less than γ_{ref}, we have a positive loss that increases the further away the margin is from γ_{ref}.

3A) Given this definition, if γ_{ref} is positive what can we say about $L_h(\gamma(x,y,\theta,\theta_0)/\gamma_{ref})$, no matter what finite values θ and θ_0 take on?

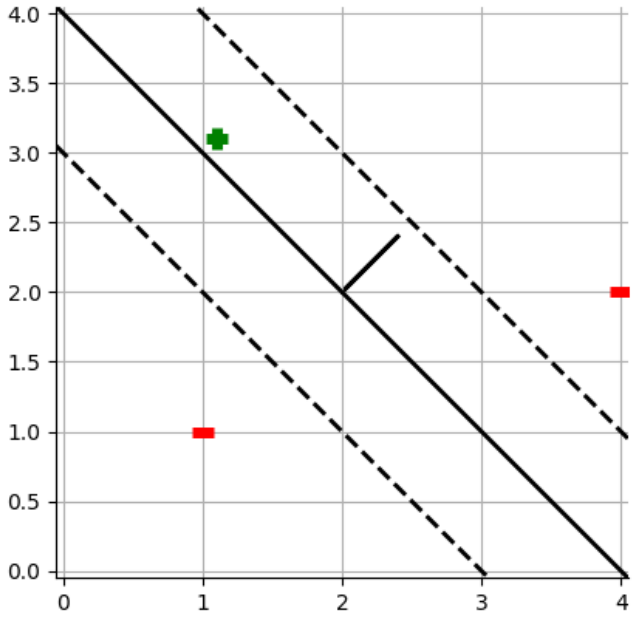
Which of the following is true about L_h: It is always >= 0

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Here is a separator and three points. The dotted lines represent the margins determined by γ_{ref}.

```
data = np.array([[1.1, 1, 4],[3.1, 1, 2]])
labels = np.array([[1, -1, -1]])
th = np.array([[1, 1]]).T
th0 = -4
```



3B) What is $L_h(\gamma(x,y,\theta,\theta_0)/\gamma_{ref})$ for each point, where $\gamma_{ref} = \sqrt{2}/2$? Enter the values in the same order as the respective points are listed in `data`. You can do this computationally or by hand.

Enter the three hinge loss values in order as a Python list of three numbers:

[0.8, 0, 3]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4) It hinges on the loss

Putting hinge loss and regularization together, we can look at regularized average hinge loss:

$$\frac{1}{n}\sum_{i=1}^{n} L_h\left(\frac{\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0)}{\gamma_{ref}}\right) + \lambda\frac{1}{\gamma_{ref}^2} \quad .$$

We only need to minimize this over two parameters θ, θ_0, since the third parameter γ_{ref} can be expressed as $\frac{1}{\|\theta\|}$, as they both represent the distance from the decision boundary to the margin boundary. Plugging in $\gamma_{ref} = \frac{1}{\|\theta\|}$ and also expanding γ, we arrive at the SVM (support vector machine) objective:

$$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_h(y^{(i)}(\theta^T x^{(i)} + \theta_0)) + \lambda\|\theta\|^2 \quad .$$

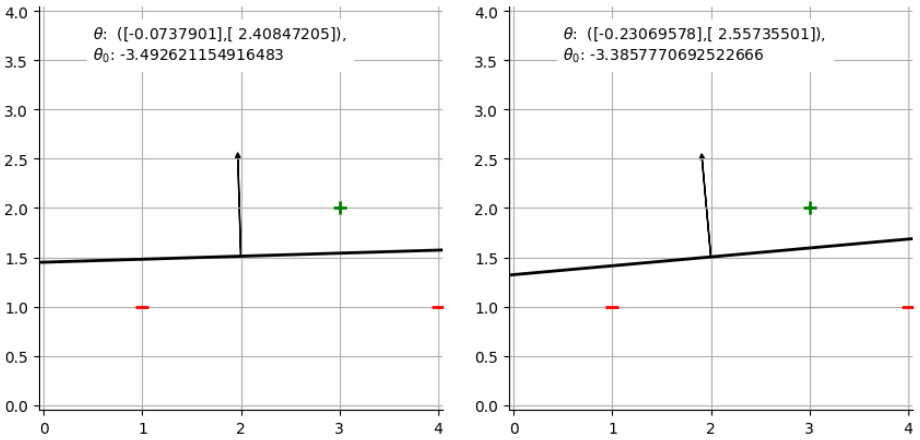
4A) If the data is linearly separable and we use the SVM objective, if we now let $\lambda = 0$ and find the minimizing values of θ, θ_0, what will happen?

Which of the following is true: The minimal objective value will be 0

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4B) Consider the following plots of separators. They are for λ values of 0 and 0.001. Match each λ to a plot.



A B

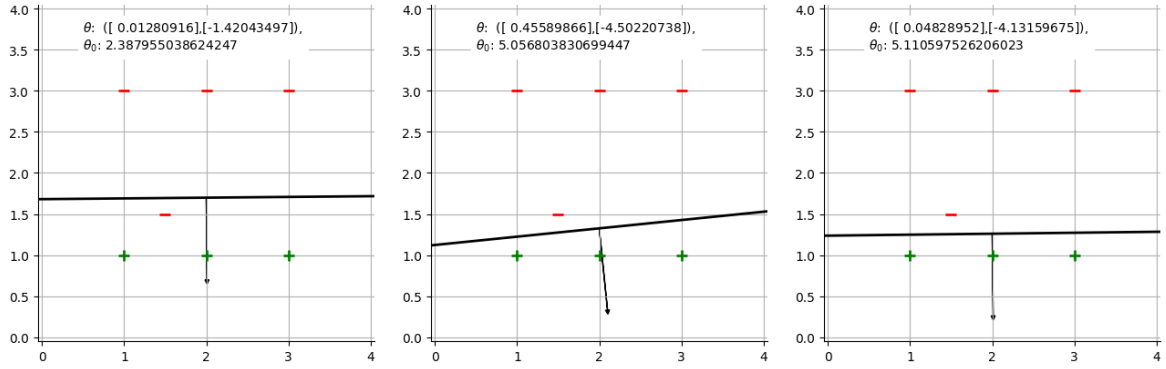
Enter a Python list with the values of λ for the two graphs above (i.e., `[lambdaA, lambdaB]`).

[0.001, 0]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

4C) Consider the following three plots of separators. They are for λ values of 0, 0.001, and 0.03. Match to the plot.



A B C

Enter a Python list with the values of λ for the three graphs above (i.e., `[lambdaA, lambdaB, lambdaC]`).

[0.03, 0, 0.001]

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

5) Linear Support Vector Machines

The training objective for the Support Vector Machine (with slack) can be seen as optimizing a balance between the average hinge loss over the examples and a regularization term that tries to keep θ small (or equivalently, increase the margin). This balance is set by the regularization parameter λ. Here we only consider the case without the offset parameter θ_0 (setting it to zero) and rewrite the training objective as an average so that it is given by

$$\left[\frac{1}{n}\sum_{i=1}^{n} L_h(y^{(i)}\,\theta\cdot x^{(i)})\right] + \frac{\lambda}{2}||\theta||^2 = \frac{1}{n}\sum_{i=1}^{n}\left[L_h(y^{(i)}\,\theta\cdot x^{(i)}) + \frac{\lambda}{2}||\theta||^2\right]$$

where $L_h(y(\theta\cdot x)) = \max\{0, 1-y(\theta\cdot x)\}$ is the hinge loss. (Note that we will also sometimes write the hinge loss as $L_h(v) = \max(0, 1-v)$.) Now we can minimize the above overall objective function with the Pegasos algorithm that iteratively selects a training point at random and applies a gradient descent update rule based on the corresponding term inside the brackets on the right hand side.

In this problem we will optimize the training objective using a single training example, so that we can gain a better understanding of how the regularization parameter, λ, affects the result. To this end, we refer to the single training example as the feature vector and label pair, (x, y). We will then try to find a θ that minimizes

$$J_\lambda^1(\theta) \equiv L_h(y(\theta\cdot x)) + \frac{\lambda}{2}||\theta||^2\ .$$

In the next subparts, we will try to show that the θ minimizing J_λ^1, denoted $\hat{\theta}$, is necessarily of the form

$$\hat{\theta} = \eta\, yx$$

for some real $\eta > 0$.

In the expressions below, you can use `lambda` to stand for λ, `x` to stand for x, `transpose(x)` for transpose of an array, `norm(x)` for the length (norm) of a vector, `x@y` to indicate a matrix product of two arrays, and `x*y` for elementwise (or scalar) multiply.

5A) Consider first the case where the loss is positive: $L_h(y(\theta\cdot x)) > 0$. We can minimize J_λ^1 with respect to θ by computing a

formula for its gradient with respect to θ, and then solving for the θ for which the gradient is equal to 0. Let us denote that value as $\hat{\theta}$.

Enter an expression for $\hat{\theta}$: x*y/lambda

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: `y * x / lambda`

$$\frac{y \times \mathbf{x}}{\lambda}$$

Explanation:

We would like to set

$$\frac{dJ_\lambda^1}{d\theta} = 0$$

and solve for θ. Expanding, we have

$$\frac{d}{d\theta}\left(L_h(y(\theta \cdot x)) + \frac{\lambda}{2}\|\theta\|^2\right) = 0$$

which is equivalent to

$$\frac{d}{d\theta}\left(\max(0, 1 - y(\theta \cdot x)) + \frac{\lambda}{2}(\theta^T\theta)\right) = 0\ .$$

When L_h is zero, this gives a minimum at $\theta = 0$, but as θ goes to 0 the L_h term is no longer zero. Thus the operative zero point in the derivative above is when $-yx + \lambda\theta = 0$, giving us the minimum at $\theta = yx/\lambda$.

This $\hat{\theta} = yx/\lambda$ thus gives us the optimal setting of θ that can minimize our loss.

Note that along the way, the derivative may not evaluate to be a scalar: for example, we see that $-yx + \lambda\theta$ is a vector. When taking derivatives with matrices this is fine; these derivatives can be used to help us find a scalar minimum of our objective function.

5B) Now find the smallest (in the norm sense) $\hat{\theta}$ for which $L_h(y(\theta \cdot x)) = 0$.

Note: Be careful -- you cannot simply divide by a vector!

Enter your answer as a Python expression: $\hat{\theta} =$ x/norm(x)**2*1/y

Check Syntax Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

Note that if the hinge loss is zero, the point is correctly classified.

5C) Let $\hat{\theta} = \hat{\theta}(\lambda)$ be the minimizer of $J_\lambda^1(\theta)$. Is it possible to pick a value for λ so that the training example x, y wil be misclassified by $\hat{\theta}(\lambda)$?

To answer this question, recall that a point is misclassified when $y(\theta \cdot x) \leq 0$. Use your result from part 5A where you found the $\hat{\theta}$ that minimizes $J_\lambda^1(\theta)$ to write an expression for $y(\hat{\theta} \cdot x)$ in terms of x, y and λ.

For θ that minimizes $J_\lambda^1(\theta)$, enter an expression for $y(\hat{\theta} \cdot x)$:

y**2*norm(x)**2/lambda

Ask for Help 100.00%

You have infinitely many submissions remaining.

Multiple Possible Solutions:

Solution 1: `(y**2 * norm(x)**2)/lambda`

$$\frac{y^2 \times \|\mathbf{x}\|^2}{\lambda}$$

Solution 2: `norm(x)**2/lambda`

$$\frac{\|\mathbf{x}\|^2}{\lambda}$$

Explanation:

We plug $\hat{\theta} = yx/\lambda$ from part A into $y(\hat{\theta} \cdot x)$. Since $x \cdot x = ||x||^2$ and $y \times y = 1$, we obtain the above result.

5D)

Under what conditions is $y(\hat{\theta} \cdot x) \leq 0$? Select all that are true.

- [x] $x = 0$
- [ ] $y < 0$
- [ ] $y > 0$
- [ ] $\lambda = 0$
- [x] $\lambda = \infty$

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution:

✓ $x = 0$

✗ $y < 0$

✗ $y > 0$

✗ $\lambda = 0$

✓ $\lambda = \infty$

Explanation:

The expression is never less than zero, so the only setting under which the expression equals zero when $x = 0$ or $\lambda = \infty$.

That is, the only time our optimal $\hat{\theta}$ misclassifies a point is if that point is $x = 0$, or $\lambda = \infty$. It is worth noting that if $\lambda = \infty$, then we do not care about finding a separator that classifies points correctly; we only care about having a small θ.

5E) You will notice that if $y(\hat{\theta} \cdot x) \leq 0$, then $\hat{\theta}$ will misclassify x. The above result shows that our optimal classifier $\hat{\theta}$ won't misclassify (except in edge cases); however, we might still be concerned about correctly classified points that are "too close" to the separator, and thereby increase our regularized loss function.

Suppose we have a linear classifier described by θ. We say a correctly classified datapoint $\hat{x}, \hat{y}$ is on the margin boundary of the classifier if

$$\hat{y}(\theta \cdot \hat{x}) = 1.$$

When a classifier is determined by minimizing a regularized loss function with a single training example, like J_λ^1 above, too much regularization can result in a classifier that puts a correctly classified training point *inside* the margin, and thus incur hinge loss. That is, if we have a single training example (x, y) and regularize with a λ that is too large, we may discover that $y(\hat{\theta} \cdot x) < 1$. Fortunately, for this single training example case, we can ensure that λ is not too large.

Write an expression for the maximum value of λ, in terms of x and y, that ensures that the (x, y) example is NOT inside the margin: y*transpose(x)@theta**-1

Check Syntax | Submit | View Answer | Ask for Help 0.00%

You have infinitely many submissions remaining.

So, where are we? We now have a good objective function, the SVM objective, that will strive to correctly classify data points, but also seek to maximize the margin (minimize the norm of θ), given a judicious choice of $lambda$. This objective function can be used in either batch optimization (calculating average losses across the whole data set), or on a data point by data point basis. This gives us powerful flexibility in optimizing (minimizing) this objective function using gradient descent, which we will consider next.

6) Implementing gradient descent

In this section we will implement generic versions of gradient descent and apply these to the SVM objective.

Reminder: For your convenience, we have copied the hands-on section into a colab notebook, which may be found here.

6.1) Gradient descent

Note: If you need a refresher on gradient descent, you may want to reference this week's notes.

We want to find the x that minimizes the value of the *objective function* $f(x)$, for an arbitrary scalar function f. The function f will be implemented as a Python function of one argument, that will be a numpy column vector. For efficiency, we will work with Python functions that return not just the value of f at $f(x)$ but also return the gradient vector at x, that is, $\nabla_x f(x)$.

We will now implement a generic gradient descent function, `gd`, that has the following input arguments:

- `f`: a function whose input is an `x`, a column vector, and returns a scalar.
- `df`: a function whose input is an `x`, a column vector, and returns a column vector representing the gradient of `f` at `x`.
- `x0`: an initial value of x, `x0`, which is a column vector.
- `step_size_fn`: a function that is given the iteration index (an integer) and returns a step size.
- `max_iter`: the number of iterations to perform

Our function `gd` returns a tuple:

- `x`: the value at the final step
- `fs`: the list of values of `f` found during all the iterations (including `f(x0)`)
- `xs`: the list of values of `x` found during all the iterations (including `x0`)

Hint: This is a short function!

Hint 2: If you do `temp_x = x` where `x` is a vector (numpy array), then `temp_x` is just another name for the same vector as `x` and changing an entry in one will change an entry in the other. You should either use `x.copy()` or remember to change entries back after modification.

Some test or example functions that you may find useful are included below. You may also find `rv` and `cv` (from previous weeks) useful, though not necessary.

```
def f1(x):
    return float((2 * x + 3)**2)

def df1(x):
    return 2 * 2 * (2 * x + 3)
```

```
def f2(v):
    x = float(v[0]); y = float(v[1])
    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2

def df2(v):
    x = float(v[0]); y = float(v[1])
    return cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
               2 * (-1. + x + y),
               2 * (-1. + x + y)])
```

To evaluate results, we also use a simple `package_ans` function, which checks the final `x`, as well as the first and last values in `fs`, `xs`.

```
def package_ans(gd_vals):
    x, fs, xs = gd_vals
    return [x.tolist(), [fs[0], fs[-1]], [xs[0].tolist(), xs[-1].tolist()]]
```

The test cases are provided below, but you should feel free (and are encouraged!) to write more of your own.

```
# Test case 1
ans=package_ans(gd(f1, df1, cv([0.]), lambda i: 0.1, 1000))

# Test case 2
ans=package_ans(gd(f2, df2, cv([0., 0.]), lambda i: 0.01, 1000))
```

```
1 def gd(f, df, x0, step_size_fn, max_iter):
2     fs = [f(x0)]
3     xs = [x0]
4     for t in range(max_iter):
5         xs += [xs[-1] - step_size_fn(t) * df(xs[-1])]
6         fs += [f(xs[-1])]
7     return xs[-1], fs, xs
```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

6.2) Numerical Gradient

Getting the analytic gradient correct for complicated functions is tricky. A very handy method of verifying the analytic gradient or even substituting for it is to estimate the gradient at a point by means of *finite differences*.

Assume that we are given a function $f(x)$ that takes a column vector as its argument and returns a scalar value. In gradient descent, we will want to estimate the gradient of f at a particular x_0.

The i^{th} component of $\nabla_x f(x_0)$ can be estimated as

$$\frac{f(x_0 + \delta^i) - f(x_0 - \delta^i)}{2\delta}$$

where δ^i is a column vector whose i^{th} coordinate is δ, a small constant such as 0.001, and whose other components are zero. Note that adding or subtracting δ^i is the same as incrementing or decrementing the i^{th} component of x_0 by δ, leaving the other components of x_0 unchanged. Using these results, we can estimate the i^{th} component of the gradient.

For example, if $x_0 = (1, 1, \ldots, 1)^T$ and $\delta = 0.01$, we may approximate the first component of $\nabla_x f(x_0)$ as

$$\frac{f((1,1,1,\ldots)^T + (0.01, 0, 0, \ldots)^T) - f((1,1,1,\ldots)^T - (0.01, 0, 0, \ldots)^T)}{2 \cdot 0.01}.$$

(We add the transpose so that these are column vectors.) **This process should be done for each dimension independently, and together the results of each computation are compiled to give the estimated gradient, which is d dimensional.**

Implement this as a function `num_grad` that takes as arguments the objective function `f` and a value of `delta`, and returns a new **function** that takes an `x` (a column vector of parameters) and returns a gradient column vector.

Note: As in the previous part, make sure you do not modify your input vector.

The test cases are shown below; these use the functions defined in the previous exercise.

```
x = cv([0.])
ans=(num_grad(f1)(x).tolist(), x.tolist())

x = cv([0.1])
ans=(num_grad(f1)(x).tolist(), x.tolist())

x = cv([0., 0.])
ans=(num_grad(f2)(x).tolist(), x.tolist())

x = cv([0.1, -0.1])
ans=(num_grad(f2)(x).tolist(), x.tolist())
```

```
def num_grad(f, delta=0.001):
    def df(x):
        d, n = x.shape
        ds = np.identity(d) * delta
        gr = np.zeros((d, n))
        for i in range(d):
            gr[i] = ( f(x + ds[:,i:i+1]) - f(x - ds[:,i:i+1]) ) / (2 * delta)
        return gr
    return df
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

A faster (one function evaluation per entry), though sometimes less accurate, estimate is to use:

$$\frac{f(x_0 + \delta^i) - f(x_0)}{\delta}$$

for the i^{th} component of $\nabla_x f(x_0)$.

6.3) Using the Numerical Gradient

Recall that our generic gradient descent function takes both a function `f` that returns the value of our function at a given point, and `df`, a function that returns a gradient at a given point. Write a function `minimize` that takes only a function `f` and uses this function and numerical gradient descent to return the local minimum. We have provided you with our implementations of `num_grad` and `gd`, so you should not redefine them in the code box below. You may use the default of `delta=0.001` for `num_grad`.

Hint: Your definition of `minimize` should call `num_grad` exactly once, to return a function that is called many times. You should return the same outputs as `gd`.

The test cases are:

```
ans = package_ans(minimize(f1, cv([0.]), lambda i: 0.1, 1000))

ans = package_ans(minimize(f2, cv([0., 0.]), lambda i: 0.01, 1000))
```

```
def minimize(f, x0, step_size_fn, max_iter):
    fs = [f(x0)]
    xs = [x0]
    df = num_grad(f, delta=0.001)
    for t in range(max_iter):
        xs += [xs[-1] - step_size_fn(t) * df(xs[-1])]
        fs += [f(xs[-1])]
    return xs[-1], fs, xs
```

Run Code | Submit | View Answer | Ask for Help | 100.00%

You have infinitely many submissions remaining.

7) Applying gradient descent to SVM objective

Now that we've implemented gradient descent in the general case, let's go back to hinge loss and the SVM objective. Our goal in this section will be to derive and implement appropriate gradient calculations that we can use with `gd` for optimization of the SVM objective. In the derivations below, we'll consider linear classifiers *with* offset, i.e., θ, θ_0.

Recall that hinge loss is defined as:

$$L_h(v) = \begin{cases} 1 - v & \text{if } v < 1 \\ 0 & \text{otherwise} \end{cases} .$$

This is usually implemented as:

$$hinge(v) = \max(0, 1 - v) \ .$$

The hinge loss function, in the context of our problem, takes in the distance from a point to the separator as x. This loss function helps us penalize a model for leaving points within a distance to our separator:

$$L_h(y\,(\theta \cdot x + \theta_0)) = \begin{cases} 1 - y\,(\theta \cdot x + \theta_0) & \text{if } y\,(\theta \cdot x + \theta_0) < 1 \\ 0 & \text{otherwise} \end{cases}$$

The SVM objective function incorporates the mean of the hinge loss over all points and introduces a regularization term to this equation to make sure that the magnitude of θ stays small (and keeps the margin large). Note that we have used λ instead of $\frac{\lambda}{2}$ for simplicity (and without loss of generality).

$$J(\theta, \theta_0) = \left[\frac{1}{n}\sum_{i=1}^{n} L_h(y^{(i)}\,(\theta \cdot x^{(i)} + \theta_0)\,)\right] + \lambda||\theta||^2$$

We're interested in applying our gradient descent procedure to this function in order to find the 'best' separator for our data, where 'best' is measured by the lowest possible SVM objective.

7.1) Calculating the SVM objective

Implement the single-argument function `hinge`, which computes L_h, and use that to implement hinge loss for a data point and separator. Using the latter function, implement the SVM objective. Note that these functions should work for matrix/vector arguments, so that we can compute the objective for a whole dataset with one call.

Note that `x` is $d \times n$, `y` is $1 \times n$, `th` is $d \times 1$, `th0` is 1×1, `lam` is a scalar.

Hint: Look at `np.where` for implementing `hinge`.

In the test cases for this problem, we'll use the following `super_simple_separable` test dataset and test separator for some of the tests. A couple of the test cases are also shown below.

```
def super_simple_separable():
    X = np.array([[2, 3, 9, 12],
                  [5, 2, 6, 5]])
    y = np.array([[1, -1, 1, -1]])
    return X, y

sep_e_separator = np.array([[-0.40338351], [1.1849563]]), np.array([[-2.26910091]])

# Test case 1
x_1, y_1 = super_simple_separable()
th1, th1_0 = sep_e_separator
ans = svm_obj(x_1, y_1, th1, th1_0, .1)

# Test case 2
ans = svm_obj(x_1, y_1, th1, th1_0, 0.0)
```

Note: In this section, you will code many individual functions, each of which depends on previous ones. We **strongly recommend** that you test each of the components on your own to debug.

```python
def hinge(v):
    return max(0, 1 - v)

# x is dxn, y is 1xn, th is dx1, th0 is 1x1
def hinge_loss(x, y, th, th0):
    return np.where(
        y * (th.T @ x + th0) < 1,
        1 - y * (th.T @ x + th0),
        0)

# x is dxn, y is 1xn, th is dx1, th0 is 1x1, lam is a scalar
def svm_obj(x, y, th, th0, lam):
    n = y.shape[1]
    return np.sum(hinge_loss(x, y, th, th0) + lam * np.linalg.norm(th)**2) / n
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

7.2) Calculating the SVM gradient

Define a function `svm_obj_grad` that returns the gradient of the SVM objective function with respect to θ and θ_0 in a single column vector. The last component of the gradient vector should be the partial derivative with respect to θ_0. Look at `np.vstack` as a simple way of stacking two matrices/vectors vertically. We have broken it down into pieces that mimic steps in the chain rule; this leads to code that is a bit inefficient but easier to write and debug. We can worry about efficiency later.

Some test cases that may be of use are shown below:

```
X1 = np.array([[1, 2, 3, 9, 10]])
y1 = np.array([[1, 1, 1, -1, -1]])
th1, th10 = np.array([[-0.31202807]]), np.array([[1.834     ]])
X2 = np.array([[2, 3, 9, 12],
               [5, 2, 6, 5]])
y2 = np.array([[1, -1, 1, -1]])
th2, th20=np.array([[ -3.,  15.]]).T, np.array([[ 2.]])

d_hinge(np.array([[ 71.]])).tolist()
d_hinge(np.array([[ -23.]])).tolist()
d_hinge(np.array([[ 71, -23.]])).tolist()

d_hinge_loss_th(X2[:,0:1], y2[:,0:1], th2, th20).tolist()
d_hinge_loss_th(X2, y2, th2, th20).tolist()
d_hinge_loss_th0(X2[:,0:1], y2[:,0:1], th2, th20).tolist()
d_hinge_loss_th0(X2, y2, th2, th20).tolist()

d_svm_obj_th(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
d_svm_obj_th(X2, y2, th2, th20, 0.01).tolist()
d_svm_obj_th0(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
d_svm_obj_th0(X2, y2, th2, th20, 0.01).tolist()

svm_obj_grad(X2, y2, th2, th20, 0.01).tolist()
svm_obj_grad(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
```

Note: In this section, you will code many individual functions, each of which depends on previous ones. We **strongly**

recommend that you test each of the components on your own to debug.

```python
#eta = 1
# Returns the gradient of hinge(v) with respect to v.
def d_hinge(v):
    return np.where(v >= 1, 0, -1)

# Returns the gradient of hinge_loss(x, y, th, th0) with respect to th
def d_hinge_loss_th(x, y, th, th0):
    d, n = x.shape
    g = np.where(y * (th.T @ x + th0) < 1, -1, 0)
    return x * y * g

# Returns the gradient of hinge_loss(x, y, th, th0) with respect to th0
def d_hinge_loss_th0(x, y, th, th0):
    return y * np.where(y * (th.T @ x + th0) < 1, -1, 0)

# Returns the gradient of svm_obj(x, y, th, th0) with respect to th
def d_svm_obj_th(x, y, th, th0, lam):
    d, n = x.shape
    # x - d x n
    # y - 1 x n
    dhlth = d_hinge_loss_th(x, y, th, th0)
    assert dhlth.shape == (d, n)
    s = np.sum(dhlth, axis=1, keepdims=True)
    return s / n  + 2 * lam * th

# Returns the gradient of svm_obj(x, y, th, th0) with respect to th0
def d_svm_obj_th0(x, y, th, th0, lam):
    d, n = x.shape
    dth0s = d_hinge_loss_th0(x, y, th, th0)
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

7.3) Batch SVM minimize

Putting it all together, use the functions you built earlier to write a gradient descent minimizer for the SVM objective. You do not need to paste in your previous definitions; you can just call the ones defined by the staff. You will need to call `gd`, which is already defined for you as well; your function `batch_svm_min` should return the values that `gd` does.

- Initialize all the separator parameters to zero,
- use the step size function provided below, and
- specify 10 iterations.

Test cases are shown below, where an additional separable test data set has been specified.

```
def separable_medium():
    X = np.array([[2, -1, 1, 1],
                  [-2, 2, 2, -1]])
    y = np.array([[1, -1, 1, -1]])
    return X, y
sep_m_separator = np.array([[ 2.69231855], [ 0.67624906]]), np.array([[-3.02402521]])

x_1, y_1 = super_simple_separable()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))

x_1, y_1 = separable_medium()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))
```

```python
def batch_svm_min(data, labels, lam):
    def svm_min_step_size_fn(i):
       return 2/(i+1)**0.5
    d, n = data.shape
    x0 = np.vstack((
        np.repeat(0, d).reshape((d,1)),
        np.repeat(0, 1)
    ))
    print(x0[0:d])
    return gd(
        lambda x: svm_obj(data, labels, x[0:d], x[d:], lam),
        lambda x: svm_obj_grad(data, labels, x[0:d], x[d:], lam),
        x0,
        svm_min_step_size_fn,
        10
    )

```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

7.4) Numerical SVM objective (Optional)

Recall from the previous question that we were able to closely approximate gradients with numerical estimates. We may apply the same technique to optimize the SVM objective.

Using your definition of `minimize` and `num_grad` from the previous problem, implement a function that optimizes the SVM objective through numeric approximations.

How well does this function perform, compared to the analytical result? Consider both accuracy and runtime.

MIT 6.036 Spring 2019: Homework 4

This homework does not include provided Python code. Instead, we encourage you to write your own code to help you answer some of these problems, and/or test and debug the code components we do ask for. Some of the problems below are simple enough that hand calculation should be possible; your hand solutions can serve as test cases for your code. You may also find that including utilities written in previous labs (like a `sd` or signed distance function) will be helpful, as you build up additional functions and utilities for calculation of margins, different loss functions, gradients, and other functions needed for margin maximization and gradient descent.

In [1]:
```
import numpy as np
```

1)

In [2]:
```
data = np.array([[1, 2, 1, 2, 10, 10.3, 10.5, 10.7],
                 [1, 1, 2, 2,  2,  2,  2, 2]])
labels = np.array([[-1, -1, 1, 1, 1, 1, 1, 1]])
blue_th = np.array([[0, 1]]).T
blue_th0 = -1.5
red_th = np.array([[1, 0]]).T
red_th0 = -2.5
```

In [3]:
```
def gamma(x, y, th, th0):
    return y * (th.T @ x + th0) / np.linalg.norm(th)

def s_sum(x, y, th, th0):
    return sum(gamma(x[:,i:i+1], y[:,i], th, th0) for i in range(x.shape[1]))[0][0]

def s_min(x, y, th, th0):
    return min(gamma(x[:,i:i+1], y[:,i], th, th0) for i in range(x.shape[1]))[0][0]

def s_max(x, y, th, th0):
    return max(gamma(x[:,i:i+1], y[:,i], th, th0) for i in range(x.shape[1]))[0][0]
```

In [34]:
```
[
    s_sum(data, labels, red_th, red_th0),
    s_min(data, labels, red_th, red_th0),
    s_max(data, labels, red_th, red_th0)
]
```

Out[34]:
```
[31.5, -1.5, 8.2]
```

In [35]:
```
[
    s_sum(data, labels, blue_th, blue_th0),
    s_min(data, labels, blue_th, blue_th0),
    s_max(data, labels, blue_th, blue_th0)
]
```

Out[35]:
```
[4.0, 0.5, 0.5]
```

3)

In [2]:
```
np.linalg.norm([[-0.0737901],[2.40847205]]), np.linalg.norm([[-0.23069578],[2.5573550
```

Out[2]:
```
(2.409602165190182, 2.567739315055543)
```

In [3]:
```
np.linalg.norm([[-0.01280916],[-1.42043497]]), np.linalg.norm([[0.45589866],[-4.50220
```

Out[3]:
```
(1.4204927238739404, 4.525230920153827, 4.131878940912039)
```

4)

```
          [1, 2],
          [2, 3],
          [3, 5],
          [1, 4]
  1 2 3 1
  2 3 5 4
```

1+4+9+1 2+6+15+4 2+6+15+4 4+9+25+16

15 27 27 44

15 *44* - *27* 27

-69

6) Implementing gradient descent

In this section we will implement generic versions of gradient descent and apply these to the SVM objective.

Note: If you need a refresher on gradient descent, you may want to reference this week's notes.

6.1) Implementing Gradient Descent

We want to find the x that minimizes the value of the *objective function* $f(x)$, for an arbitrary scalar function f. The function f will be implemented as a Python function of one argument, that will be a numpy column vector. For efficiency, we will work with Python functions that return not just the value of f at $f(x)$ but also return the gradient vector at x, that is, $\nabla_x f(x)$.

We will now implement a generic gradient descent function, `gd`, that has the following input arguments:

- `f`: a function whose input is an `x`, a column vector, and returns a scalar.
- `df`: a function whose input is an `x`, a column vector, and returns a column vector representing the gradient of `f` at `x`.
- `x0`: an initial value of x, `x0`, which is a column vector.
- `step_size_fn`: a function that is given the iteration index (an integer) and returns a step size.
- `max_iter`: the number of iterations to perform

Our function `gd` returns a tuple:

- `x`: the value at the final step
- `fs`: the list of values of `f` found during all the iterations (including `f(x0)`)
- `xs`: the list of values of `x` found during all the iterations (including `x0`)

Hint: This is a short function!

Hint 2: If you do `temp_x = x` where `x` is a vector (numpy array), then `temp_x` is just another name for the same vector as `x` and changing an entry in one will change an entry in the other. You should either use `x.copy()` or remember to change entries back after modification.

Some utilities you may find useful are included below.

In [4]:
```
def rv(value_list):
    return np.array([value_list])

def cv(value_list):
    return np.transpose(rv(value_list))

def f1(x):
    return float((2 * x + 3)**2)

def df1(x):
    return 2 * 2 * (2 * x + 3)

def f2(v):
    x = float(v[0]); y = float(v[1])
    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2

def df2(v):
    x = float(v[0]); y = float(v[1])
    return cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
               2 * (-1. + x + y),
               2 * (-1. + x + y)])
```

The main function to implement is `gd`, defined below.

In [8]:
```
def gd(f, df, x0, step_size_fn, max_iter):
    fs = [f(x0)]
    xs = [x0]
    for t in range(max_iter):
        xs += [xs[-1] - step_size_fn(t) * df(xs[-1])]
        fs += [f(xs[-1])]
    return xs[-1], fs, xs
```

To evaluate results, we also use a simple `package_ans` function, which checks the final `x`, as well as the first and last values in `fs`, `xs`.

In [9]:
```
def package_ans(gd_vals):
    x, fs, xs = gd_vals
    return [x.tolist(), [fs[0], fs[-1]], [xs[0].tolist(), xs[-1].tolist()]]
```

The test cases are provided below, but you should feel free (and are encouraged!) to write more of your own.

In [10]:
```
# Test case 1
ans=package_ans(gd(f1, df1, cv([0.]), lambda i: 0.1, 1000))
print(ans)

# Test case 2
ans=package_ans(gd(f2, df2, cv([0., 0.]), lambda i: 0.01, 1000))
print(ans)
```

```
[[[-1.5]], [9.0, 0.0], [[[0.0]], [[-1.5]]]]
[[[-2.2058239041648853], [3.205823890926977]], [19.0, -20.967239611348752], [[[0.0],
[0.0]], [[-2.2058239041648853], [3.205823890926977]]]]
```

6.2) Numerical Gradient

Getting the analytic gradient correct for complicated functions is tricky. A very handy method of verifying the analytic gradient or even substituting for it is to estimate the gradient at a point by means of *finite differences*.

Assume that we are given a function $f(x)$ that takes a column vector as its argument and returns a scalar value. In gradient descent, we will want to estimate the gradient of f at a particular $x_0.$

The i^{th} component of $\nabla_x f(x_0)$ can be estimated as $$\frac{f(x_0+\delta^{i}) - f(x_0-\delta^{i})}{2\delta}$$ where δ^{i} is a column vector whose i^{th} coordinate is δ, a small constant such as 0.001, and whose other components are zero. Note that adding or subtracting δ^{i} is the same as incrementing or decrementing the i^{th} component of x_0 by δ, leaving the other components of x_0 unchanged. Using these results, we can estimate the i^{th} component of the gradient.

For example, if $x_0 = (1,1,\dots,1)^T$ and $\delta = 0.01$, we may approximate the first component of $\nabla_x f(x_0)$ as $$\frac{f((1,1,1,\dots)^T+(0.01,0,0,\dots)^T) - f((1,1,1,\dots)^T-(0.01,0,0,\dots)^T)}{2\cdot 0.01}.$$ (We add the transpose so that these are column vectors.) **This process should be done for each dimension independently, and together the results of each computation are compiled to give the estimated gradient, which is d dimensional.**

Implement this as a function `num_grad` that takes as arguments the objective function `f` and a value of `delta`, and returns a new **function** that takes an `x` (a column vector of parameters) and returns a gradient column vector.

Note: As in the previous part, make sure you do not modify your input vector.

In [11]:
```
def num_grad(f, delta=0.001):
    def df(x):
        d, n = x.shape
        ds = np.identity(d) * delta
        gr = np.zeros((d, n))
        for i in range(d):
            gr[i] = ( f(x + ds[:,i:i+1]) - f(x - ds[:,i:i+1]) ) / (2 * delta)
        return gr
    return df
```

The test cases are shown below; these use the functions defined in the previous exercise.

In [83]:
```
x = cv([0.])
#ans=(num_grad(f1)(x).tolist(), x.tolist())
#print(ans)

x = cv([0.1])
#ans=(num_grad(f1)(x).tolist(), x.tolist())
#print(ans)

x = cv([0., 0.])
ans=(num_grad(f2)(x).tolist(), x.tolist())
expected = [[6.99999899999959], [-2.000000000000668]],
print(ans ,expected)
print(ans[0] == expected)


x = cv([0.1, -0.1])
ans=(num_grad(f2)(x).tolist(), x.tolist())
print(ans)
```

```
ds [[0.001 0.   ]
 [0.    0.001]]
ds[:,i:i+1] [[0.001]
 [0.   ]]
i gr 0 [[6.999999]
 [0.      ]]
ds[:,i:i+1] [[0.   ]
 [0.001]]
i gr 1 [[ 6.999999]
 [-2.      ]]
([[6.99999899999959], [-2.000000000000668]], [[0.0], [0.0]]) ([[6.99999899999959], [-
2.000000000000668]],)
False
ds [[0.001 0.   ]
 [0.    0.001]]
ds[:,i:i+1] [[0.001]
 [0.   ]]
i gr 0 [[4.7739994]
 [0.       ]]
ds[:,i:i+1] [[0.   ]
 [0.001]]
i gr 1 [[ 4.7739994]
 [-2.       ]]
([[4.77399940000011166], [-2.000000000000668]], [[0.1], [-0.1]])
```

In [ ]:

A faster (one function evaluation per entry), though sometimes less accurate, estimate is to use: $$\frac{f(x_0+\delta^{i}) - f(x_0)}{\delta}$$ for the i^{th} component of $\nabla_x f(x_0).$

6.3) Using the Numerical Gradient

Recall that our generic gradient descent function takes both a function `f` that returns the value of our function at a given point, and `df`, a function that returns a gradient at a given point. Write a function `minimize` that takes only a function `f` and uses this function and numerical gradient descent to return the local minimum. We have provided you with our implementations of `num_grad` and `gd`, so you should not redefine them in the code box below. You may use the default of `delta=0.001` for `num_grad`.

Hint: Your definition of `minimize` should call `num_grad` exactly once, to return a function that is called many times. You should return the same outputs as `gd`.

In [12]:
```
def minimize(f, x0, step_size_fn, max_iter):
    fs = [f(x0)]
    xs = [x0]
    df = num_grad(f, delta=0.001)
    for t in range(max_iter):
        xs += [xs[-1] - step_size_fn(t) * df(xs[-1])]
        fs += [f(xs[-1])]
    return xs[-1], fs, xs
```

The test cases are below.

In [13]:
```
ans = package_ans(minimize(f1, cv([0.]), lambda i: 0.1, 1000))
print(ans)

ans = package_ans(minimize(f2, cv([0., 0.]), lambda i: 0.01, 1000))
print(ans)
```

```
[[[-1.5]], [9.0, 0.0], [[[0.0]], [[-1.5]]]]
[[[-2.2058237062057517], [3.205823692967833]], [19.0, -20.967239611347775], [[[0.0],
[0.0]], [[-2.2058237062057517], [3.205823692967833]]]]
```

7) Applying gradient descent to SVM objective

Note: In this section, you will code many individual functions, each of which depends on previous ones. We **strongly recommend** that you test each of the components on your own to debug.

7.1) Calculating the SVM objective

Implement the single-argument hinge function, which computes L_h, and use that to implement hinge loss for a data point and separator. Using the latter function, implement the SVM objective. Note that these functions should work for matrix/vector arguments, so that we can compute the objective for a whole dataset with one call.

```
x is d x n, y is 1 x n, th is d x 1, th0 is 1 x 1, lam is a scalar
```

Hint: Look at `np.where` for implementing `hinge`.

In [6]:
```
a = np.arange(10)
np.where(a < 5, a, 0)
```

Out[6]:
```
array([0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

In [14]:
```
def hinge(v):
    return max(0, 1 - v)

# x is dxn, y is 1xn, th is dx1, th0 is 1x1
def hinge_loss(x, y, th, th0):
    return np.where(
        y * (th.T @ x + th0) < 1,
        1 - y * (th.T @ x + th0),
        0)

# x is dxn, y is 1xn, th is dx1, th0 is 1x1, lam is a scalar
def svm_obj(x, y, th, th0, lam):
    n = y.shape[1]
    return np.sum(hinge_loss(x, y, th, th0) + lam * np.linalg.norm(th)**2) / n
```

In [15]:
```
# add your tests here

assert hinge(1) == 0, 'Actual: "{}"'.format(hinge(1))
assert hinge(-1) == 2
assert hinge(1.5) == 0
assert hinge(0) == 1
assert hinge(0.5) == 0.5
```

In the test cases for this problem, we'll use the following `super_simple_separable` test dataset and test separator for some of the tests. A couple of the test cases are also shown below.

In [16]:
```
def super_simple_separable():
    X = np.array([[2, 3, 9, 12],
                  [5, 2, 6, 5]])
    y = np.array([[1, -1, 1, -1]])
    return X, y

sep_e_separator = np.array([[-0.40338351], [1.1849563]]), np.array([[-2.26910091]])

# Test case 1
x_1, y_1 = super_simple_separable()
th1, th1_0 = sep_e_separator
ans = svm_obj(x_1, y_1, th1, th1_0, .1)
print(ans)

# Test case 2
ans = svm_obj(x_1, y_1, th1, th1_0, 0.0)
print(ans)
```

```
0.15668396890496103
0.0
```

7.2) Calculating the SVM gradient

Define a function `svm_obj_grad` that returns the gradient of the SVM objective function with respect to θ and θ_0 in a single column vector. The last component of the gradient vector should be the partial derivative with respect to θ_0. Look at `np.vstack` as a simple way of stacking two matrices/vectors vertically. We have broken it down into pieces that mimic steps in the chain rule; this leads to code that is a bit inefficient but easier to write and debug. We can worry about efficiency later.

In [ ]:

In [17]:
```
#eta = 1
# Returns the gradient of hinge(v) with respect to v.
def d_hinge(v):
    return np.where(v >= 1, 0, -1)

# Returns the gradient of hinge_loss(x, y, th, th0) with respect to th
def d_hinge_loss_th(x, y, th, th0):
    d, n = x.shape
    g = np.where(y * (th.T @ x + th0) < 1, -1, 0)
    return x * y * g

# Returns the gradient of hinge_loss(x, y, th, th0) with respect to th0
def d_hinge_loss_th0(x, y, th, th0):
    return y * np.where(y * (th.T @ x + th0) < 1, -1, 0)

# Returns the gradient of svm_obj(x, y, th, th0) with respect to th
def d_svm_obj_th(x, y, th, th0, lam):
    d, n = x.shape
    # x - d x n
    # y - 1 x n
    dhlth = d_hinge_loss_th(x, y, th, th0)
    assert dhlth.shape == (d, n)
    s = np.sum(dhlth, axis=1, keepdims=True)
    return s / n  + 2 * lam * th

# Returns the gradient of svm_obj(x, y, th, th0) with respect to th0
def d_svm_obj_th0(x, y, th, th0, lam):
    d, n = x.shape
    dth0s = d_hinge_loss_th0(x, y, th, th0)
    assert dth0s.shape == (1, n)
    return np.sum(dth0s, axis=1, keepdims=True) / n

# Returns the full gradient as a single vector
def svm_obj_grad(X, y, th, th0, lam):
    return np.vstack((
        d_svm_obj_th(X, y, th, th0, lam),
        d_svm_obj_th0(X, y, th, th0, lam)
    ))
```

Some test cases that may be of use are shown below.

In [18]:
```
X1 = np.array([[1, 2, 3, 9, 10]])
y1 = np.array([[1, 1, 1, -1, -1]])
th1, th10 = np.array([[-0.31202807]]), np.array([[1.834     ]])
X2 = np.array([[2, 3, 9, 12],
               [5, 2, 6, 5]])
y2 = np.array([[1, -1, 1, -1]])
th2, th20=np.array([[ -3.,  15.]]).T, np.array([[ 2.]])
```

In [19]:
```
X2 * X2
```

Out[19]:
```
array([[  4,   9,  81, 144],
       [ 25,   4,  36,  25]])
```

In [20]:
```
(
d_hinge(np.array([[ 71.]])).tolist(),
d_hinge(np.array([[ -23.]])).tolist(),
d_hinge(np.array([[ 71, -23.]])).tolist(),
)
```

Out[20]:
```
([[0]], [[-1]], [[0, -1]])
```

In [21]:
```
def mytest(actual, expected):
    assert actual == expected, f"Actual: '{actual}' <> Expected: '{expected}'"
```

In [22]:
```
mytest(d_hinge_loss_th(X2[:,0:1], y2[:,0:1], th2, th20).tolist(), [[0], [0]])

#d_hinge_loss_th(X2, y2, th2, th20).tolist()
#d_hinge_loss_th0(X2[:,0:1], y2[:,0:1], th2, th20).tolist()
#d_hinge_loss_th0(X2, y2, th2, th20).tolist()

mytest(
    d_hinge_loss_th(X2, y2, th2, th20).tolist(),
    [[0, 3, 0, 12], [0, 2, 0, 5]]
)

mytest(
    d_hinge_loss_th0(X2, y2, th2, th20).tolist() ,
[[0.0, 1.0, 0.0, 1.0]]
)
```

In [23]:
```
mytest(
    d_svm_obj_th(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist(),
    [[-0.06], [0.3]]
)

#d_svm_obj_th(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
#d_svm_obj_th(X2, y2, th2, th20, 0.01).tolist()

#d_svm_obj_th0(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
# d_svm_obj_th0(X2, y2, th2, th20, 0.01).tolist()
```

In [24]:
```
mytest(
    d_svm_obj_th(X2, y2, th2, th20, 0.01).tolist(),
    [[3.69], [2.05]]
)
```

In [25]:
```
mytest(
    d_svm_obj_th0(X2, y2, th2, th20, 0.01).tolist(),
    [[0.5]]
)
```

In [26]:
```
svm_obj_grad(X2, y2, th2, th20, 0.01).tolist()
svm_obj_grad(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
```

Out[26]:
```
[[-0.06], [0.3], [0.0]]
```

7.3) Batch SVM minimize

Putting it all together, use the functions you built earlier to write a gradient descent minimizer for the SVM objective. You do not need to paste in your previous definitions; you can just call the ones defined by the staff. You will need to call `gd`, which is already defined for you as well; your function `batch_svm_min` should return the values that `gd` does.

- Initialize all the separator parameters to zero,
- use the step size function provided below, and
- specify 10 iterations.

In [59]:
```
def batch_svm_min(data, labels, lam):
    def svm_min_step_size_fn(i):
       return 2/(i+1)**0.5
    d, n = data.shape
    x0 = np.vstack((
        np.repeat(0, d).reshape((d,1)),
        np.repeat(0, 1)
    ))
    print(x0[0:d])
    return gd(
        lambda x: svm_obj(data, labels, x[0:d], x[d:], lam),
        lambda x: svm_obj_grad(data, labels, x[0:d], x[d:], lam),
        x0,
        svm_min_step_size_fn,
        10
    )
```

Test cases are shown below, where an additional separable test data set has been specified.

In [58]:
```
def separable_medium():
    X = np.array([[2, -1, 1, 1],
                  [-2, 2, 2, -1]])
    y = np.array([[1, -1, 1, -1]])
    return X, y
sep_m_separator = np.array([[ 2.69231855], [ 0.67624906]]), np.array([[-3.02402521]])

x_1, y_1 = super_simple_separable()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))
x_1,y_1=super_simple_separable()
ans=package_ans(batch_svm_min(x_1, y_1, 0.0001))

mytest(ans,
[
    [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]],
    [1.0, 0.37283613860066195],
    [
        [[0.0], [0.0], [0.0]],
        [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]]
    ]

])
x_1, y_1 = separable_medium()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))
```

```
[[0]
 [0]]
[[0]
 [0]]
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [58], in <cell line: 13>()
     10 x_1,y_1=super_simple_separable()
     11 ans=package_ans(batch_svm_min(x_1, y_1, 0.0001))
---> 13 mytest(ans,
     14 [
     15     [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]],
     16     [1.0, 0.37283613860066195],
     17     [
     18         [[0.0], [0.0], [0.0]],
     19         [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]]
     20     ]
     21
     22 ])
     23 x_1, y_1 = separable_medium()
     24 ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))

Input In [21], in mytest(actual, expected)
      1 def mytest(actual, expected):
----> 2     assert actual == expected, f"Actual: '{actual}' <> Expected: '{expecte
d}'"

AssertionError: Actual: '[[[-1.4810507930100065], [4.406219189763341], [-0.4037730527
9098563]], [1.0, 2.457217409802802], [[[0], [0], [0]], [[-1.4810507930100065], [4.406
219189763341], [-0.40377305279098563]]]]' <> Expected: '[[[-2.430041469694636], [3.77
42410656579057], [-0.40377305279098563]], [1.0, 0.37283613860066195], [[[0.0], [0.0],
[0.0]], [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]]]]'
```

7.4) Numerical SVM objective (Optional)

Recall from the previous question that we were able to closely approximate gradients with numerical estimates. We may apply the same technique to optimize the SVM objective.

Using your definition of `minimize` and `num_grad` from the previous problem, implement a function that optimizes the SVM objective through numeric approximations.

How well does this function perform, compared to the analytical result? Consider both accuracy and runtime.

In [ ]:
```
# your code here
```

For these exercises, it will be helpful to review the notes on Regression.

1) Intro to linear regression

So far, we have been looking at classification, where predictors are of the form

$$y = \text{sign}(\theta^T x + \theta_0)$$

making a binary classification as to whether example x belongs to the positive or negative class of examples.

In many problems, we want to predict a real value, such as the actual gas mileage of a car, or the concentration of some chemical. Luckily, we can use most of a mechanism we have already spent building up, and make predictors of the form:

$$y = \theta^T x + \theta_0.$$

This is called a *linear regression* model.

We would like to learn a linear regression model from examples. Assume X is a d by n array (as before) but that Y is a 1 by n array of floating-point numbers (rather than +1 or -1). Given data (X, Y) we need to find θ, θ_0 that does a good job of making predictions on new data drawn from the same source.

We will approach this problem by formulating an objective function. There are many possible reasonable objective functions that implicitly make slightly different assumptions about the data, but they all typically have the form:

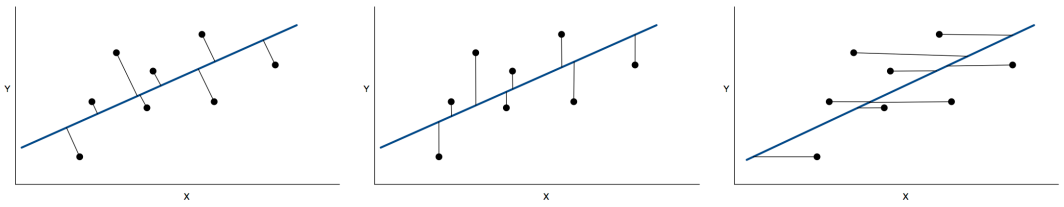
$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0).$$

For regression, we most frequently use *squared loss*, in which

$$L_s(x, y, \theta, \theta_0) = (y - \theta^T x - \theta_0)^2.$$

The term with $R(\theta, \theta_0)$ is termed the *regularizer*, and penalizes more complex predictors. We will explore different choices of regularizer later in this set of exercises.

Ex1.1: Which of the following pictures illustrates the squared loss metric? Assume that the dark line is described by θ, θ_0, the black dots are the (x, y) data, and the light lines indicate the errors.



A B C

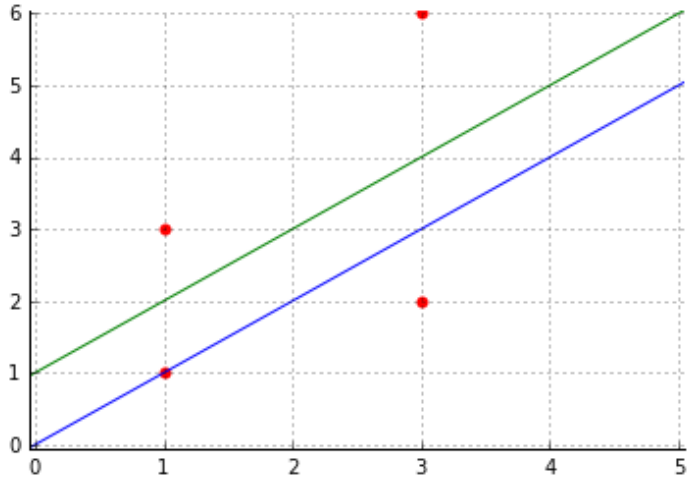
Select the picture which best illustrates the squared loss metric: B

View Answer 100.00%

You have 0 submissions remaining.

2) Linear Regression

Consider the data set and regression lines in the plot below.



- The equation of the blue (lower) line is: $y = x$
- The equation of the green (upper) line is: $y = x + 1$
- The data points (in x, y pairs) are: $((1, 3), (1, 1), (3, 2), (3, 6))$

Ex2.1: What is the squared error of each of the points with respect to the **blue** line?

Provide a Python list of four numbers (in the order of the points given above).

[4, 0, 1, 9]

Submit View Answer **100.00%**

You have 19 submissions remaining.

The gradient of the mean squared error regression criterion has the form of a sum over contributions from individual points. The formula for the gradient of the squared error with respect to parameters of a line, θ, θ_0 for a single point (x, y) (without regularizer), is:

$$\left(-2(y - \theta^T x - \theta_0)x, \quad -2(y - \theta^T x - \theta_0)\right).$$

Ex2.2: What is the gradient contribution from each point to the parameters of the blue (lower) line?

Provide a list of four pairs of numbers (as tuples, in the order of the points given above).

[(-4,-4),(0,0),(6,2),(-18,-6)]

Submit View Answer **100.00%**

You have 18 submissions remaining.

Ex2.3: What is the squared error of each of the points with respect to the green line?

Provide a list of four numbers (in the order of the points given above).

[1,1,4,4]

Submit View Answer 100.00%

You have 19 submissions remaining.

Ex2.4: What is the gradient contribution from each point to the parameters of the green line?

Provide a list of four pairs of numbers (as tuples, in the order of the points given above).

[(-2, -2), (2, 2), (12, 4), (-12, -4)]

Submit View Answer 100.00%

You have 19 submissions remaining.

Ex2.5: Mark all of the following that are true:

- [ ] The blue line minimizes mean squared error
- [x] The green line minimizes mean squared error
- [ ] The mean squared error from all the points to the blue line is 0
- [ ] The mean squared error from all the points to the green line is 0
- [ ] The sum of the gradient contributions from all the points for the blue line is 0
- [x] The sum of the gradient contributions from all the points for the green line is 0
- [ ] Neither line minimizes mean squared error
- [ ] It is impossible to minimize mean squared error
- [ ] Both lines minimize mean squared error

Submit View Answer 100.00%

You have 5 submissions remaining.

3) Ridge regression

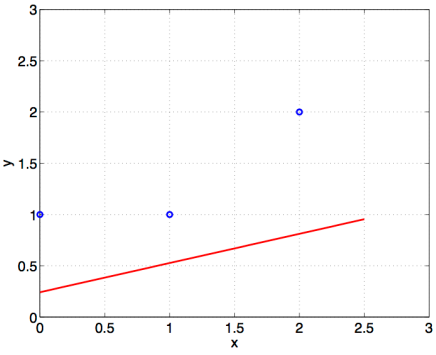
It may be help to review the notes on regularization.

If we add a squared-norm regularizer to the empirical risk, we get the so-called *ridge regression* objective:

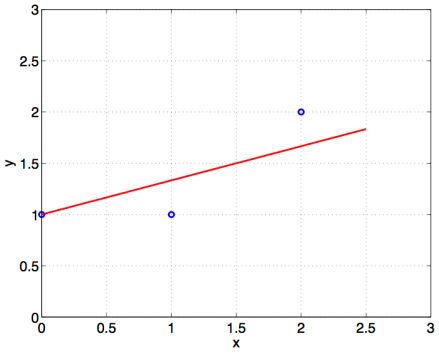
$$J_{ridge}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda ||\theta||^2.$$

It's a bit tricky to solve this analytically, because you can see that the penalty is on θ but not on θ_0.

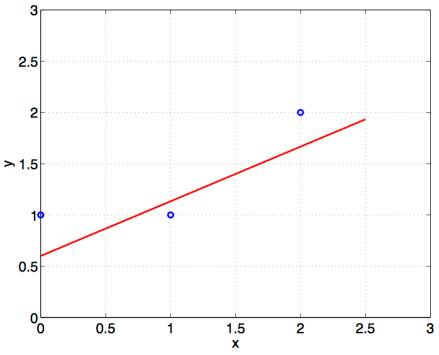
The figures below plot linear regression results on the basis of only three data points $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)})$. We used various types of regularization to obtain the plots (see below) but got confused about which plot corresponds to which regularization method. Please assign each plot to one (and only one) of the following regularization methods.



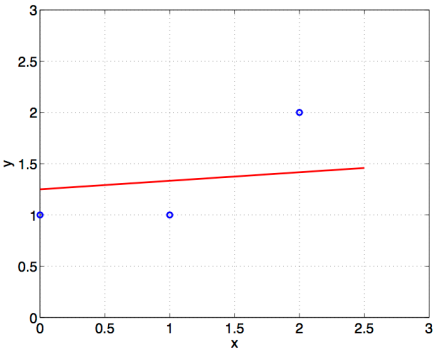
A



B



C



D

Ex3.1:

$\frac{1}{3}\sum_{i=1}^{3}(y^i - wx^i - w_0)^2 + \lambda w^2$ where $\lambda = 1$ B

Submit View Answer 100.00%

You have 2 submissions remaining.

Ex3.2:

$\frac{1}{3}\sum_{i=1}^{3}(y^i - wx^i - w_0)^2 + \lambda w^2$ where $\lambda = 10$ D

Submit View Answer 100.00%

You have 2 submissions remaining.

Ex3.3:

$\frac{1}{3}\sum_{i=1}^{3}(y^i - wx^i - w_0)^2 + \lambda(w^2 + w_0^2)$ where $\lambda = 1$ C

Submit View Answer 100.00%

You have 1 submission remaining.

Ex3.4:

$\frac{1}{3}\sum_{i=1}^{3}(y^i - wx^i - w_0)^2 + \lambda(w^2 + w_0^2)$ where $\lambda = 10$ A

Submit View Answer 100.00%

You have 1 submission remaining.

Regression

During this week, we are exploring regression. Along the way, we will also deepen our understanding and experience with gradient descent, particularly as applied to regression. The previous notes on gradient descent will be useful, as well as the notes on regression.

In this lab, we will start by running code for solving regression problems and doing gradient descent on the mean square loss to develop an understanding for how it behaves. In the homework, you will implement all of the important functions necessary to create these demos.

Exploring gradient descent for regression and classification

In many problems, we want to predict a real value, such as the actual gas mileage of a car, or the concentration of some chemical. Luckily, we can use most of a mechanism we have already spent building up, and make predictors of the form:

$$y = \theta^T x + \theta_0$$

This is called a *linear regression* model.

We would like to learn a linear regression model from examples. Assume X is a d by n array (as before) but that Y is a 1 by n array of *floating-point* numbers (rather than +1 or -1). Given data (X, Y) we need to find θ, θ_0 that does a good job of making predictions on new data drawn from the same source.

We will approach this problem by formulating an objective function. There are many possible reasonable objective functions that implicitly make slightly different assumptions about the data, but they all typically have the form:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) \text{ (without regularization)}$$

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta) \text{ (with regularization)}$$

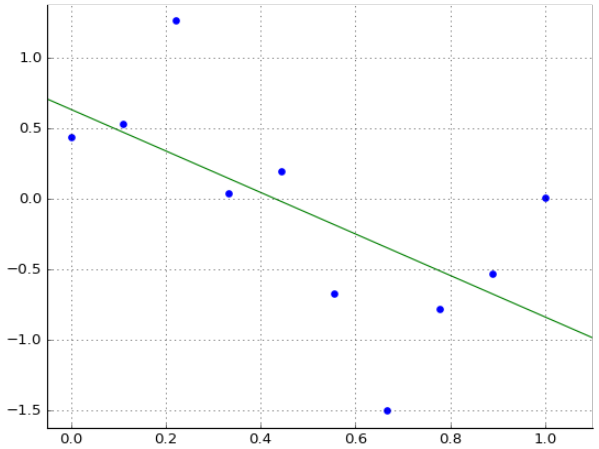
where L is our loss function and R is our regularization in terms of θ. For regression, we most frequently use *squared loss*, in which

$$L_s(x, y, \theta, \theta_0) = (\hat{y} - y)^2 = (\theta^T x + \theta_0 - y)^2$$

where our prediction is $\hat{y} = \theta^T x + \theta_0$ and y is our actual data.

We will come back to our regularization later...

In this lab, we will experiment with linear regression, gradient descent, and polynomial features. We will start with a simple data-set with one input feature and 10 training points, which is easy to visualize. In the plot below, the horizontal axis is our x value, and the vertical axis is the corresponding y value.



1) Least squares regression

In least squares regression problems, we assume that our objective function $J(\theta, \theta_0)$ comes without a regularization term; in other words,

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

Recall from the lecture and notes that it is possible to solve a least-squares regression problem directly via the matrix algebra expression for the parameter vector θ in terms of the input data X and desired output vector Y. In this section, we will explore this *analytic* solution strategy.

For the rest of this lab, we will be computing solutions by transforming our one-dimensional input features with a polynomial basis. Specifically, we will be transforming our single input feature, which we will call x, into the vector $\phi(x) = (x^1, ..., x^k)$ if we are using a k-th order basis. This means that solutions to our regression problem will take the form

$$\theta^T \phi(x) + \theta_0 = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_k x^k$$

To answer the next set of problems, you are given the function `t1` to experiment with, which:

- Takes as input `order`, which is the order of the polynomial basis.
- Outputs θ and θ_0 that minimizes the regression objective $J(\theta, \theta_0)$.
- Finds θ and θ_0 **analytically** (we will explore the analytical solution in the homework).

In the codebox below, experiment with `t1` and try different values of order k to examine the effects of the order of the polynomial basis on the resulting solution. Based on your observations, answer the questions below the codebox.

To answer 1A and 1B below, you should try to answer the questions first without running `t1`, and then run `t1` to match with your expectations.

```
def run():
    return t1(order=12)

```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

1A) Consider the solution with the 0th-order basis (the regression polynomial described by only θ_0). What is the shape of the solution that you expect, and why? How does this solution depend on training data? Does this match what you observe in running `t1` with order 0?

1B) Polynomial order 9 is particularly interesting. What do you expect to be true about the solution, and why? (Hint: you are given 10 points in the training data). Does this match what you observe in running `t1` with order 9?

To answer the questions below, you should run the codebox above with various orders from 1 to 9 and observe plots and values of θ, θ_0. Make sure to click "submit".

1C) From your observation in the previous question, what is problematic if the polynomial order gets too big?

1D) Look at the values of θ and θ_0 you obtain, shown in the results. How does the magnitude of θ change with order? (You don't have to answer this quantitatively.)

1E) What polynomial order do you feel represents the hypothesis that will be the most predictive for new data?

1F) When we run 10-fold cross validation on this data set, varying the order from 0 to 8 (because we train on 9 training data points for each instance of 10-fold cross validation), we get the following mean squared error results:

```
[0.69206670716618535, 0.53820006043438084, 0.73424762793041687, 0.2835495578961193, 0.74338564580774558,
0.61422551802155112, 6.156711267187811, 356.8873742619765, 408.34678081302491]
```

What is the best order, based on this data? Does it agree with your previous answer?

1G) Abstractly, if we were to use a polynomial model with orders higher than 9 (e.g., 12), should it be possible for that polynomial to go through all the points?

1H) Now, actually run `t1` with orders higher than 9 (e.g., 12). Does the learned polynomial go through all the points? Remember that `t1` is fitting the polynomial analytically (directly using matrix inversion) as discussed in the notes on regression. Would we expect matrix inversion to work well for models with orders higher than 9 for this data?

2) Regularizing the parameter vector

Recall that we can add a regularization term $R(\theta)$ to the empirical risk. If we use a squared-norm regularizer, we get the so-called *ridge regression* objective:

$$J_{ridge}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda\|\theta\|^2$$

The procedure `t2(order, lam)` performs ridge regression on polynomial features of order `order` with regularization coefficient `lam`. In the resulting plots, `t2` draws the original least-squares solution in orange, and raws the regularized version in green.

Using a 9-th order polynomial feature function, experiment with λ. Look at the detailed results after submitting your code, paying attition to the visualization of the polynomial that was fit using your chosen λ.

```
def run():
    return t2(order=9, lam = 0.001)

```

Run Code | Submit | View Answer | Ask for Help 100.00%

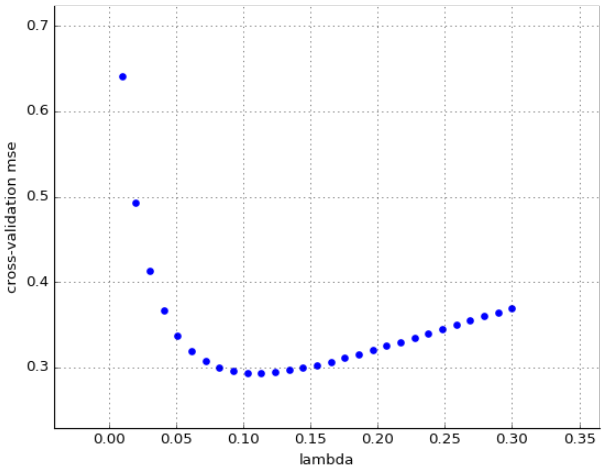
You have infinitely many submissions remaining.

2A) What happens to both $J_{ridge}(\theta, \theta_0)$ and the learned regression line, with very large (e.g., infinite) and very small (0) values of λ?

2B) If our goal is to solely minimize $\frac{1}{n}\sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0)$, what would be the best value of λ?

2C) What value of λ do you feel will give good performance on new data from the same source?

2D) When we run leave-one-out cross validation on this data set, using 9th order features, varying λ from 0.01 to 0.3, we get the following plot:



What is the best value of λ, in terms of generalization performance, based on this data? Does it agree with your previous answer? Is it the same as the best value for performance on the training set?

3) Gradient descent

Computing the analytic solution requires inverting a d by d matrix; as the size of the feature space increases, this becomes difficult. In addition, there are lots of other useful machine-learning models for which there is no closed-form analytic solution. So, we'll play with gradient descent here and see how it works. The procedure

```
t3(order, lam, step_size, max_iter)
```

performs gradient descent on the ridge regression objective using polynomial features of order `order`. You can specify the maximum number of iterations (we capped it at 10,000 to keep from killing the server) and the step size. It will print a convergence plot (objective value versus iteration number) and then show the solution it found in green and the analytic solution in orange for comparison.

Keeping $\lambda = 0$, experiment with this method for solving regression problems. (Note the difference in the plots when the solution diverges and when the solution converges very quickly.) Click 'Show/Hide Detailed Results' after submitting your code to view the convergence plot generated.

```
1 def run():
2     return t3(order=9, lam=0, step_size= 0.325, max_iter = 10000)
3
```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3A) With `max_iter = 1000` in the code snippet above, what step sizes were needed to match the analytic solution almost

exactly for 1st and 2nd order bases? (Use the convergence plot to help decide on whether the two curves "match almost exactly.")

3B) What step size allowed you to get similar curves to the analytical solution for 3rd order? (Use the convergence plot to help decide whether the two curves are "similar.")

3C) For 9th order, play around for a while (no more than 10 minutes) to see how close you can get to the analytic solution. How does it compare to the analytical solution?

In [2]:
```
def f1(x, y, th=1, th0=1):
    return (-2*(y - x * th - th0) * x, -2 * (y - th* x - th0))
[
    f1(1,3),
    f1(1,1),
    f1(3,2),
    f1(3,6)
]
```

Out[2]:
```
[(-2, -2), (2, 2), (12, 4), (-12, -4)]
```

In [5]:
```
print('a', 1/9+1/36)
print('c', 51/60)
print('b', 1/9+1)
```

```
a 0.1388888888888889
c 0.85
b 1.1111111111111112
```

LAB

> 1A) Consider the solution with the 0th-order basis (the regression polynomial described by only θ0\theta_0θ0). What is the shape of the solution that you expect, and why? How does this solution depend on training data? Does this match what you observe in running t1 with order 0?

This should be horizontal line that is shifted up or down towards better SE. Match.

> 1B) Polynomial order 9 is particularly interesting. What do you expect to be true about the solution, and why? (Hint: you are given 10 points in the training data). Does this match what you observe in running t1 with order 9?

Should be overfit. Exactly at those points. Match.

> To answer the questions below, you should run the codebox above with various orders from 1 to 9 and observe plots and values of θ\thetaθ, θ0\theta_0θ0. Make sure to click "submit".
>
> 1C) From your observation in the previous question, what is problematic if the polynomial order gets too big?

From about 7-8 order it goes almost directly by points. Overfitting.

> 1D) Look at the values of θ\thetaθ and θ0\theta_0θ0 you obtain, shown in the results. How does the magnitude of θ\thetaθ change with order? (You don't have to answer this quantitatively.)

Increases from about 1 digit to 3-4 digits before dot.

> 1E) What polynomial order do you feel represents the hypothesis that will be the most predictive for new data?

3-4 is good because it seems no such overfit.

> 1F) When we run 10-fold cross validation on this data set, varying the order from 0 to 8 (because we train on 9 training data points for each instance of 10-fold cross validation), we get the following mean squared error results: [0.69206670716618535, 0.53820006043438084, 0.73424762793041687, 0.2835495578961193, 0.74338564580774558, 0.61422551802155112, 6.156711267187811, 356.8873742619765, 408.34678081302491]
>
> What is the best order, based on this data? Does it agree with your previous answer?

Best is 3 order because error result is the least, 0.28 whereas the worst is 8 order (overfitting). Agree.

> 1G) Abstractly, if we were to use a polynomial model with orders higher than 9 (e.g., 12), should it be possible for that polynomial to go through all the points?

Yes (but no), if we use order that is close to the number of points. Example is points on a sin function.

> 1H) Now, actually run t1 with orders higher than 9 (e.g., 12). Does the learned polynomial go through all the points? Remember that t1 is fitting

> the polynomial analytically (directly using matrix inversion) as discussed in the notes on regression. Would we expect matrix inversion to work well for models with orders higher than 9 for this data?

It did't go through all. Matrix inversion don't work well for models higher than 9 as the experiment shows.

2)

> 2A) What happens to both Jridge(θ,θ0)J_{\it ridge}(\theta, \theta_0)Jridge(θ,θ0) and the learned regression line, with very large (e.g., infinite) and very small (0) values of λ\lambdaλ?

With lambda close to 0 (tried 1e-7) the overfit increases for the regulirized (green) model. With large lambda (1e10) it is 'underfit', just a lin because regularizer penalizes large norm of theta, i.e. making theta almost 0 and only theta_0 contributes to moving the line so that it fits closer to the training examples.

> 2B) If our goal is to solely minimize 1n ∑ i=1nL(x(i),y(i),θ,θ0)\frac{1}{n} \sum_{i = 1}^n L(x^{(i)}, y^{(i)}, \theta, \theta_0)n1 ∑ i=1nL(x(i),y(i),θ,θ0), what would be the best value of λ\lambdaλ?

Then lambda should be 0 as then we would minize only those L (but overfit).

> 2C) What value of λ\lambdaλ do you feel will give good performance on new data from the same source?

Don't know. Tried 10, 1, 0.1, 0.01, 0.001. It looks like overfit with 0.001 so probably 0.01 or 0.1 is better.

> 2D) When we run leave-one-out cross validation on this data set, using 9th order features, varying λ\lambdaλ from 0.01 to 0.3, we get the following plot:
>
> What is the best value of λ\lambdaλ, in terms of generalization performance, based on this data? Does it agree with your previous answer? Is it the same as the best value for performance on the training set?

The best is near 0.1 because mse on the new data is lowest for this. Yes, it agrees!

3)

> 3A) With max_iter = 1000 in the code snippet above, what step sizes were needed to match the analytic solution almost exactly for 1st and 2nd order bases? (Use the convergence plot to help decide on whether the two curves "match almost exactly.")

It diverges near 0.44 (figure 1). And before that it almost match exactly the analytical solution (fig. 2)

Figure 1. Diverge. With `t3(order=2, lam=0, step_size= 0.439, max_iter = 1000)`

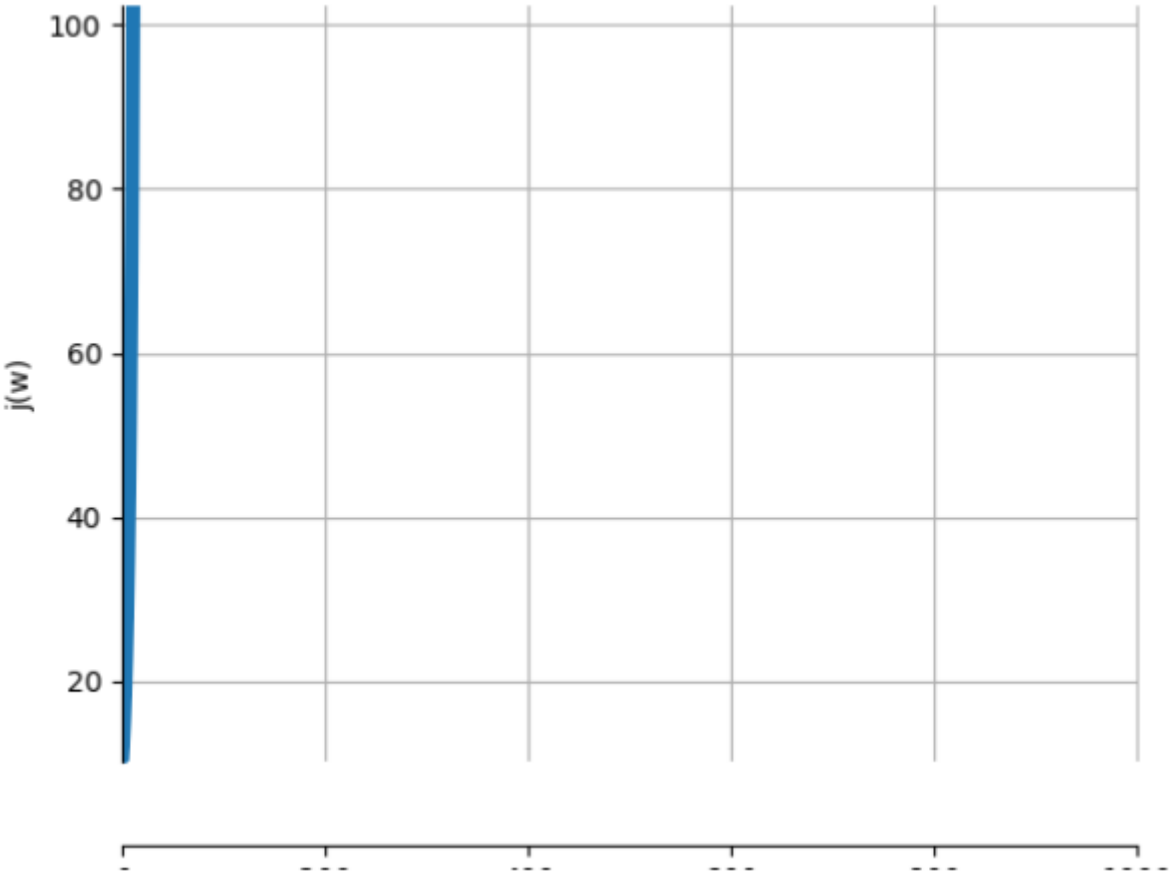
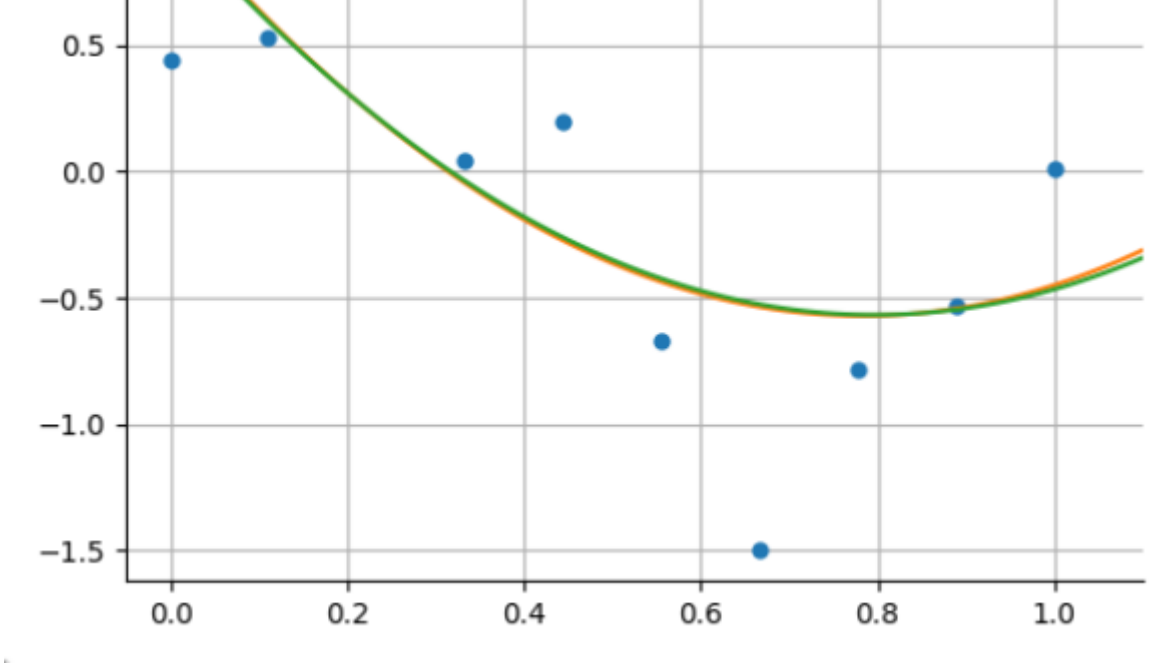
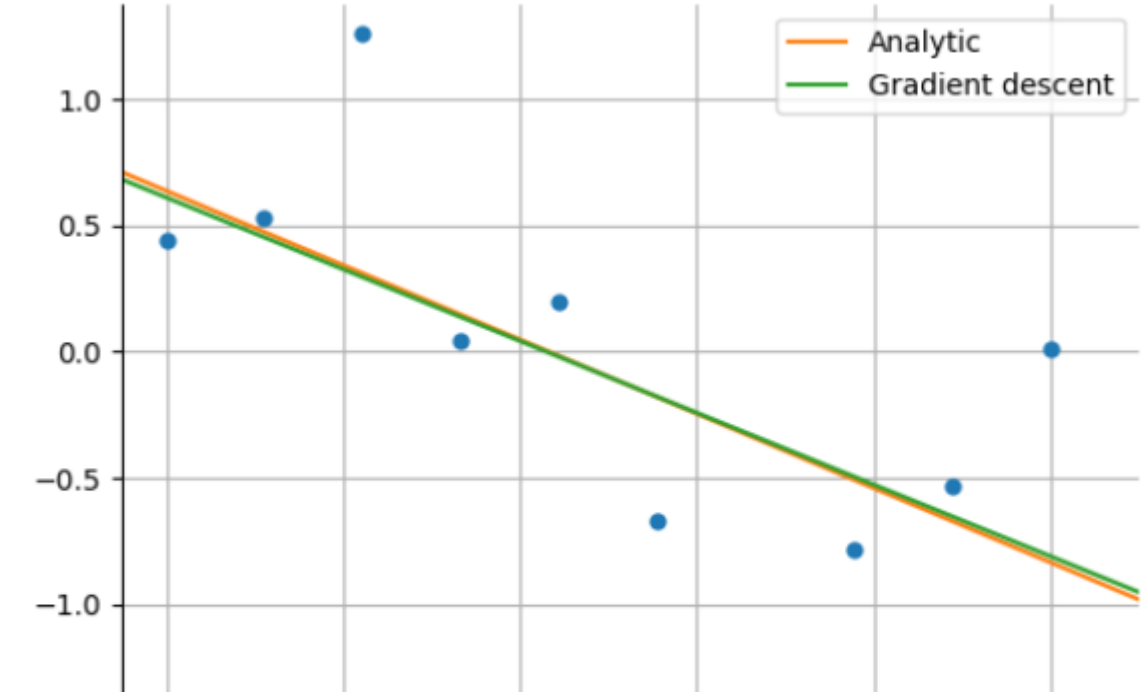


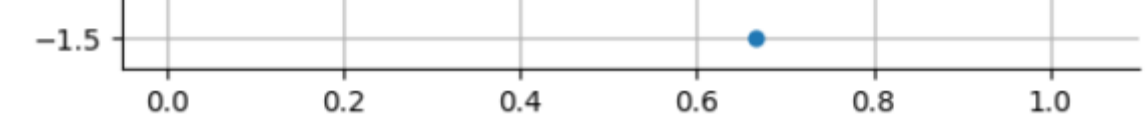
Fig. 2. Almost the same as analytical. `t3(order=2, lam=0, step_size= 0.41, max_iter = 1000)`



For order 1. It converges and matches the analytical solution around 0.02. See fig. 3

Fig 3. t3(order=1, lam=0, step_size= 0.02, max_iter = 1000)

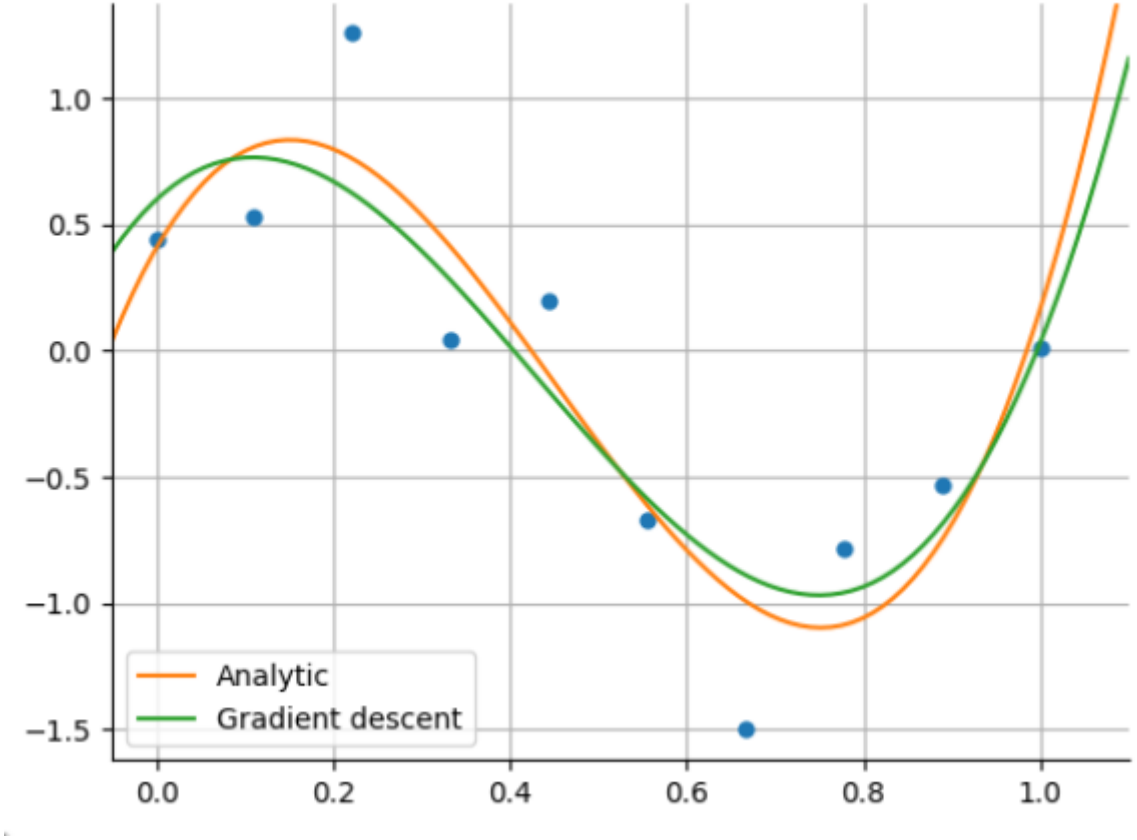




> 3B) What step size allowed you to get similar curves to the analytical solution for 3rd order? (Use the convergence plot to help decide whether the two curves are "similar.")

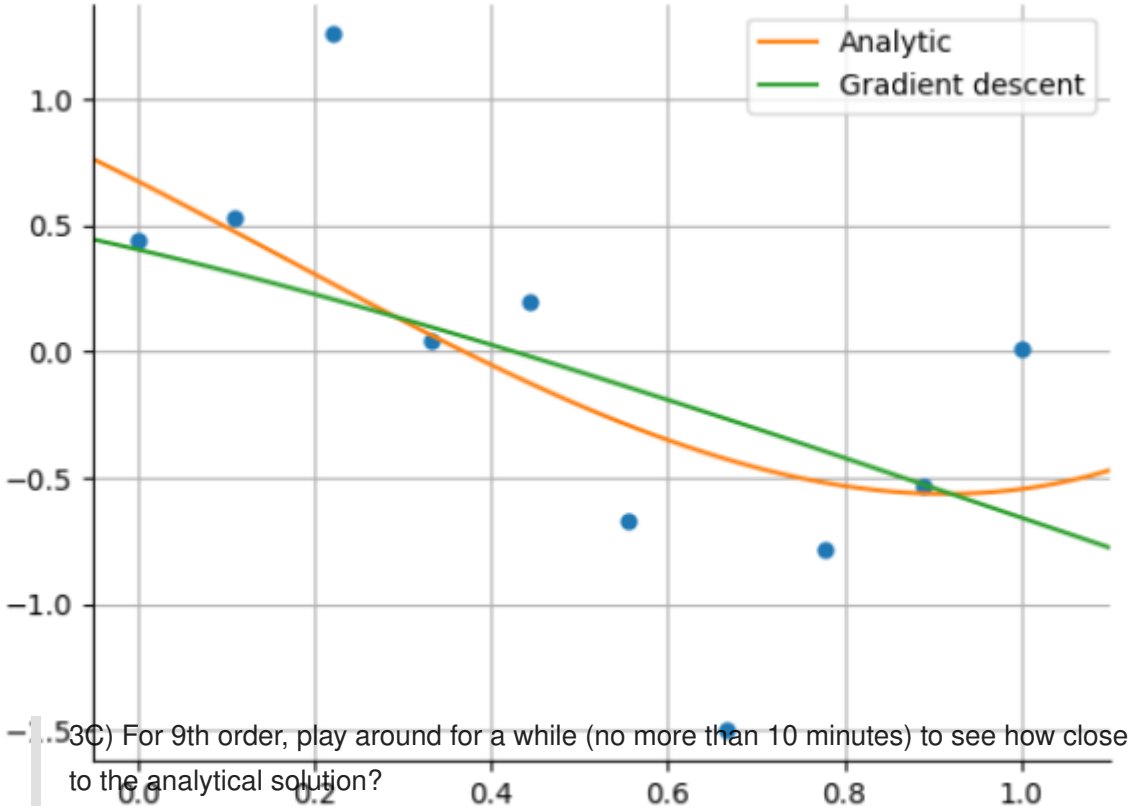
Best I could get to match GD curve to analytical one is with 0.39 step size and 10K steps. Fig. 4.

FIg. 5. Matching GD curve to analytical one. `t3(order=3, lam=0.00001, step_size= 0.39, max_iter = 10000)`



Increasing lambda resulted in increase of structural error (flat line). Meaning our line could fit the data well. See fig. 6.

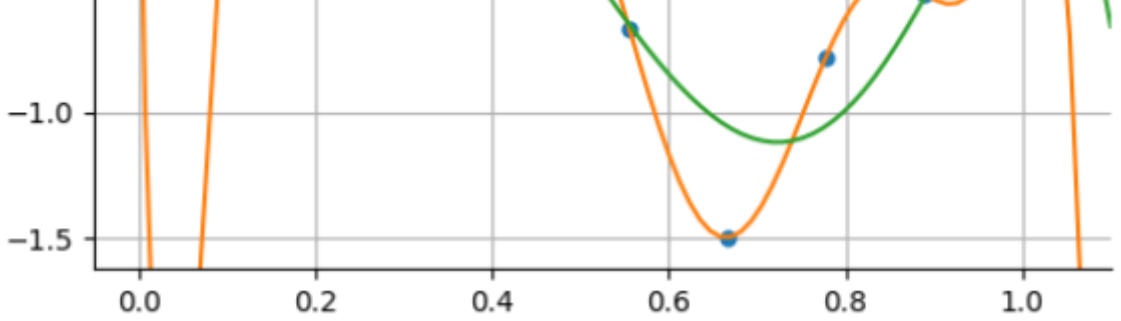
Fig. 6. Increasing lambda. `t3(order=3, lam=0.1, step_size= 0.39, max_iter = 10000)`



3C) For 9th order, play around for a while (no more than 10 minutes) to see how close you can get to the analytic solution. How does it compare to the analytical solution?

The closest one I could get had step size less than the order of 3: 0.325. With 0.39 as previously it diverged.

Fig. 7. Closest one `t3(order=9, lam=0, step_size= 0.325, max_iter = 10000)`


−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0

MIT 6.036 Spring 2019: Homework 5

In [4]:
```
import numpy as np
```

Setup

First, download the code distribution for this homework that contains test cases and helper functions.

Run the next code block to download and import the code for this lab.

In [5]:
```
#!rm -rf code_and_data_for_hw05*
#!wget --no-check-certificate --quiet https://introml_oll.odl.mit.edu/6.036/static/homework/hw05/code_and_data_for_hw05.zip
#!unzip code_and_data_for_hw05.zip
#!mv code_and_data_for_hw05/* .

import code_for_hw5 as hw5
```

6) Linear Regression - going downhill

We will now write some general Python code to compute the gradient of the squared-loss objective, following the structure of the expression, and the rules of calculus. Note that this style of writing the gradient functions maps directly into the chain-rule steps required to compute the gradient, but produces code that is inefficient, because of duplicated computations. It is straightforward to implement more efficient versions if you want to use them for larger problems.

6.1) Some basic functions

We start by defining some basic functions for computing the mean squared loss. Note that we want these to work for any value of n. That is, `x` could be a single feature vector or a full data matrix, and similarly for `y`.

In [6]:
```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def lin_reg(x, th, th0):
    """ Returns the predicted y

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 0.]])
    >>> lin_reg(X, th, th0).tolist()
    [[1.05, 2.05, 3.05, 4.05]]
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> lin_reg(X, th, th0).tolist()
    [[3.05, 4.05, 5.05, 6.05]]
    """
    return np.dot(th.T, x) + th0
def square_loss(x, y, th, th0):
    """ Returns the squared loss between y_pred and y

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> square_loss(X, Y, th, th0).tolist()
    [[4.2025, 3.4224999999999985, 5.0625, 3.8025000000000007]]
    """
    return (y - lin_reg(x, th, th0))**2
def mean_square_loss(x, y, th, th0):
    """ Return the mean squared loss between y_pred and y

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> mean_square_loss(X, Y, th, th0).tolist()
    [[4.1225]]
    """
    # the axis=1 and keepdims=True are important when x is a full matrix
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True)
```

6.2) Gradients with respect to θ

Now, let's compute the gradients with respect to θ. Make sure that they work both for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.

In [7]:

```
# Write a function that returns the gradient of lin_reg(x, th, th0)
# with respect to th
def d_lin_reg_th(x, th, th0):
    """ Returns the gradient of lin_reg(x, th, th0) with respect to th

    Note that for array (rather than vector) x, we get a d x n
    result. That is to say, this function produces the gradient for
    each data point i ... n, with respect to each theta, j ... d.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]); th0 = np.array([[ 2.]])
    >>> d_lin_reg_th(X[:,0:1], th, th0).tolist()
    [[1.0], [1.0]]

    >>> d_lin_reg_th(X, th, th0).tolist()
    [[1.0, 2.0, 3.0, 4.0], [1.0, 1.0, 1.0, 1.0]]
    """
    return x

# Write a function that returns the gradient of square_loss(x, y, th, th0) with
# respect to th.  It should be a one-line expression that uses lin_reg and
# d_lin_reg_th.
def d_square_loss_th(x, y, th, th0):
    """Returns the gradient of square_loss(x, y, th, th0) with respect to
       th.

       Note: should be a one-line expression that uses lin_reg and
       d_lin_reg_th (i.e., uses the chain rule).

       Should work with X, Y as vectors, or as arrays. As in the
       discussion of d_lin_reg_th, this should give us back an n x d
       array -- so we know the sensitivity of square loss for each
       data point i ... n, with respect to each element of theta.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist()
    [[4.1], [4.1]]

    >>> d_square_loss_th(X, Y, th, th0).tolist()
    [[4.1, 7.399999999999999, 13.5, 15.600000000000001], [4.1, 3.6999999999999993, 4.5, 3.9000000000000004]]

    """
    dths = d_lin_reg_th(x, th, th0) # d x n
    hs = lin_reg(x, th, th0) - y # 1 x n
    return dths * hs * 2
```

```
# Write a function that returns the gradient of mean_square_loss(x, y, th, th0) with
# respect to th.  It should be a one-line expression that uses d_square_loss_th.
def d_mean_square_loss_th(x, y, th, th0):
    """ Returns the gradient of mean_square_loss(x, y, th, th0) with
        respect to th.

        Note: It should be a one-line expression that uses d_square_loss_th.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_mean_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist()
    [[4.1], [4.1]]

    >>> d_mean_square_loss_th(X, Y, th, th0).tolist()
    [[10.15], [4.05]]
    """
    # print("X =", repr(X))
    # print("Y =", repr(Y))
    # print("th =", repr(th), "th0 =", repr(th0))
    d, n = x.shape
    return np.sum(d_square_loss_th(x, y, th, th0), axis=1, keepdims=True) / n
```

In [8]:
```
def mytest(a, e):
    assert a == e, f"Actual: '{a}' <> Expected: '{e}'"
```

In [9]:
```
X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
th = np.array([[ 1.  ], [ 0.05]]); th0 = np.array([[ 2.]])
print(X[:, 0:1])
mytest(d_lin_reg_th(X[:,0:1], th, th0).tolist(),
[[1.0], [1.0]])

mytest(d_lin_reg_th(X, th, th0).tolist(),
[[1.0, 2.0, 3.0, 4.0], [1.0, 1.0, 1.0, 1.0]])
```

```
[[1.]
 [1.]]
```

In [10]:
```
X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
mytest(
    d_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist(),
    [[4.1], [4.1]]
)
mytest(
    d_square_loss_th(X, Y, th, th0).tolist(),
    [[4.1, 7.399999999999999, 13.5, 15.600000000000001], [4.1, 3.6999999999999993, 4.5, 3.9000000000000004]]
)
```

In [11]:
```
X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
mytest(
    d_mean_square_loss_th(X[:,0:1], Y[:,0:1], th, th0).tolist(),
    [[4.1], [4.1]])

mytest(
    d_mean_square_loss_th(X, Y, th, th0).tolist(),
    [[10.15], [4.05]]
)
```

6.3) Gradients with respect to θ_0

Now, let's compute the gradients with respect to θ_0. Make sure that they work both for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.

In [12]:

```
# Write a function that returns the gradient of lin_reg(x, th, th0)
# with respect to th0. Hint: Think carefully about what the dimensions of the returned value should be!
def d_lin_reg_th0(x, th, th0):
    """ Returns the gradient of lin_reg(x, th, th0) with respect to th0.

    >>> x = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_lin_reg_th0(x, th, th0).tolist()
    [[1.0, 1.0, 1.0, 1.0]]
    """
    d, n = x.shape
    return np.repeat(1.0, n).reshape((1,n))

# Write a function that returns the gradient of square_loss(x, y, th, th0) with
# respect to th0.  It should be a one-line expression that uses lin_reg and
# d_lin_reg_th0.
def d_square_loss_th0(x, y, th, th0):
    """ Returns the gradient of square_loss(x, y, th, th0) with
        respect to th0.

    # Note: uses broadcasting!

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_square_loss_th0(X, Y, th, th0).tolist()
    [[4.1, 3.6999999999999993, 4.5, 3.9000000000000004]]
    """
    return (lin_reg(x, th, th0) - y) * d_lin_reg_th0(x, th, th0) * 2

# Write a function that returns the gradient of mean_square_loss(x, y, th, th0) with
# respect to th0.  It should be a one-line expression that uses d_square_loss_th0.
def d_mean_square_loss_th0(x, y, th, th0):
    """ Returns the gradient of mean_square_loss(x, y, th, th0) with
    respect to th0.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_mean_square_loss_th0(X, Y, th, th0).tolist()
    [[4.05]]
    """
    d, n = x.shape
    return np.sum(d_square_loss_th0(x, y, th, th0), axis=1, keepdims=True) / n
```

7) Going down the ridge

Now, let's add a regularizer. The ridge objective can be implemented as follows:

In [13]:
```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def ridge_obj(x, y, th, th0, lam):
    """ Return the ridge objective value

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> ridge_obj(X, Y, th, th0, 0.0).tolist()
    [[4.1225]]
    >>> ridge_obj(X, Y, th, th0, 0.5).tolist()
    [[4.623749999999999]]
    >>> ridge_obj(X, Y, th, th0, 100.).tolist()
    [[104.37250000000002]]
    """
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True) + lam * np.linalg.norm(th)**2
```

Let's extend our previous code for the gradient of the mean square loss to compute the gradient of the ridge objective w ith respect to θ. Our previous solutions for the non-ridge case: `d_mean_square_loss_th` and `d_mean_square_loss_th0` will be defined for you in the grader, so feel free to call them!

In [14]:

```
def d_ridge_obj_th(x, y, th, th0, lam):
    """Return the derivative of tghe ridge objective value with respect
    to theta.

    Note: uses broadcasting to add d x n to d x 1 array below

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_ridge_obj_th(X, Y, th, th0, 0.0).tolist()
    [[10.15], [4.05]]
    >>> d_ridge_obj_th(X, Y, th, th0, 0.5).tolist()
    [[11.15], [4.1]]
    >>> d_ridge_obj_th(X, Y, th, th0, 100.).tolist()
    [[210.15], [14.05]]
    """
    return d_mean_square_loss_th(x, y, th, th0) + th * 2 * lam

def d_ridge_obj_th0(x, y, th, th0, lam):
    """Return the derivative of tghe ridge objective value with respect
    to theta.

    Note: uses broadcasting to add d x n to d x 1 array below

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_ridge_obj_th0(X, Y, th, th0, 0.0).tolist()
    [[4.05]]
    >>> d_ridge_obj_th0(X, Y, th, th0, 0.5).tolist()
    [[4.05]]
    >>> d_ridge_obj_th0(X, Y, th, th0, 100.).tolist()
    [[4.05]]
    """
    return d_mean_square_loss_th0(x, y, th, th0)
```

8) Stochastic gradient

We will now implement stochastic gradient descent in a general way, similar to what we did with gradient descent (`gd`).

The calling conventions for `sgd` are similar to those of `gd` except that we need to pass in the data and labels for the problem.

(Recall that the *stochastic* part refers to using a randomly selected point and corresponding label from the given dataset to perform an update. Therefore, your objective function for a given step wil l need to take this into account.)

- `X` : a standard data array (d by n)
- `y` : a standard labels row vector (1 by n)
- `J` : a cost function whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector) (in that order), and which returns a scalar.
- `dJ` : a cost function gradient (corresponding to `J`) whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector) (also in that order), and which returns a column vector.
- `w0` : an initial value of weight vector w, which is a column vector.
- `step_size_fn` : a function that is given the (zero-indexed) iteration index (an integer) and returns a step size.
- `max_iter` : the number of iterations to perform

It returns a tuple (like `gd`):

- `w` : the value of the weight vector at the final step
- `fs` : the list of values of J found during all the iterations
- `ws` : the list of values of w found during all the iterations

Note: `w` should be the value one gets after applying stochastic gradient descent to `w0` for `max_iter-1` iterations (we call this the final step). The first element of `fs` should be the value of `J` calculated with `w0` , and `fs` should have length `max_iter` ; similarly, the first element of `ws` should be `w0` , and `ws` should have length `max_iter` .

You might find the function `np.random.randint(n)` useful in your implementation.

Hint: This is a short function; our implementation is around 10 lines.

The main function to implement is below.

In [130…

```
def sgd(X, y, J, dJ, w0, step_size_fn, max_iter):
    """Implements stochastic gradient descent

    Inputs:
    X: a standard data array (d by n)
    y: a standard labels row vector (1 by n)

    J: a cost function whose input is a data point (a column vector),
    a label (1 by 1) and a weight vector w (a column vector) (in that
    order), and which returns a scalar.

    dJ: a cost function gradient (corresponding to J) whose input is a
    data point (a column vector), a label (1 by 1) and a weight vector
    w (a column vector) (also in that order), and which returns a
    column vector.

    w0: an initial value of weight vector www, which is a column
    vector.

    step_size_fn: a function that is given the (zero-indexed)
    iteration index (an integer) and returns a step size.

    max_iter: the number of iterations to perform

    Returns: a tuple (like gd):
    w: the value of the weight vector at the final step
    fs: the list of values of JJJ found during all the iterations
    ws: the list of values of www found during all the iterations

    """
    d, n = X.shape
    ws = [w0]
    fs = [J(X[:,0:1], y[:,0:1], w0)]
    np.random.seed(0)
    for t in range(max_iter-1):
        i = np.random.randint(n)
        step_size = step_size_fn(t)
        xi = X[:,i:i+1]
        yi = y[:,i:i+1]
        ws += [ws[-1] - step_size * dJ(xi, yi, ws[-1])]
        fs += [J(xi, yi, ws[-1])]
    return ws[-1], fs, ws
```

In [128…

```
a = np.array([0.36, 0.028653685126009333])
b = np.array([0.16000000000000003, 0.0008607446103289605])
a / b
```

Out[128]:
```
array([ 2.25      , 33.28941568])
```

The test cases for this problem are provided below (but, as always, you are encouraged to write more if you want to better test your code!). They rely on the function num_grad (taken from the previous week's homework), also provided.

In [124…
```
def rv(value_list):
    return np.array([value_list])

def cv(value_list):
    return np.transpose(rv(value_list))
"""
def f1(x):
    return float((2 * x + 3)**2)

def df1(x):
    return 2 * 2 * (2 * x + 3)

def f2(v):
    x = float(v[0]); y = float(v[1])
    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2

def df2(v):
    x = float(v[0]); y = float(v[1])
    return cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
              2 * (-1. + x + y),
              2 * (-1. + x + y)])
"""
```

Out[124]:
```
'\ndef f1(x):\n    return float((2 * x + 3)**2)\n\ndef df1(x):\n    return 2 * 2 * (2 * x + 3)\n\ndef f2(v):\n    x = float(v
[0]); y = float(v[1])\n    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2\n\ndef df2(v):\n    x = float(v
[0]); y = float(v[1])\n    return cv([(-3. + x) * (-2. + x) * (1. + x) +               (-3. + x) * (-2. + x) * (3. + x) +
(-3. + x) * (1. + x) * (3. + x) +               (-2. + x) * (1. + x) * (3. + x) +               2 * (-1. + x + y),\n
2 * (-1. + x + y)])\n'
```

In [125…

```
def num_grad(f):
    def df(x):
        g = np.zeros(x.shape)
        delta = 0.001
        for i in range(x.shape[0]):
            xi = x[i,0]
            x[i,0] = xi - delta
            xm = f(x)
            x[i,0] = xi + delta
            xp = f(x)
            x[i,0] = xi
            g[i,0] = (xp - xm)/(2*delta)
        return g
    return df

def downwards_line():
    X = np.array([[0.0, 0.1, 0.2, 0.3, 0.42, 0.52, 0.72, 0.78, 0.84, 1.0],
                  [1.0, 1.0, 1.0, 1.0, 1.0,  1.0,  1.0,  1.0,  1.0,  1.0]])
    y = np.array([[0.4, 0.6, 1.2, 0.1, 0.22, -0.6, -1.5, -0.5, -0.5, 0.0]])
    return X, y

X, y = downwards_line()

def J(Xi, yi, w):
    # translate from (1-augmented X, y, theta) to (separated X, y, th, th0) format
    return float(ridge_obj(Xi[:-1,:], yi, w[:-1,:], w[-1:,:], 0))

def dJ(Xi, yi, w):
    def f(w): return J(Xi, yi, w)
    return num_grad(f)(w)
```

In [126…

```
ans=sgd(X, y, J, dJ, cv([0., 0.]), lambda i: 0.1, 1000)
# print(ans[0])
# print(ans[1][0])
# print(ans[1][-1])
# print(ans[2][0])
# print(ans[2][-1])
```

```
[-1.20232666]
```

9) Predicting mpg values

We will now try to synthesize the functions we have written in order to perform ridge regression on the auto-mpg dataset from lab03. Unlike in lab03, we will now try to predict the actual mpg values of the cars, instead of whether they are above or below the median mpg!

As a reminder, the dataset is as follows:

```
1. mpg:           continuous
2. cylinders:     multi-valued discrete
3. displacement:  continuous
4. horsepower:    continuous
5. weight:        continuous
6. acceleration:  continuous
7. model year:    multi-valued discrete
8. origin:        multi-valued discrete
9. car name:      string (many values)
```

For convenience, we will choose to not include `model year` and `car name` as features. For the remaining features, we again have the option to keep the raw values, standardize them, or use a one-hot encoding.

With this considered, we decide to standardize or one-hot encode all features in this section (we encourage you, though, to try raw features on your own time to see how their performance matches your expectations!).

One additional step we perform is to standardize the output values. Note that we did not have to worry about this in a classification context, as all outputs were ± 1. In a regression context, standardizing the output values can have practical performance gains, again due to better numerical performance of learning algorithms on data which is smaller in magnitude.

The metric we will use to measure the quality of our learned predictors is **Root Mean Square Error (RMSE).** RMSE is defined as follows:

$$ \text{RMSE} = \sqrt{ \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - f(x^{(i)}) \right)^2 } $$
where f is our learned predictor: in this case, $f(x) = \theta \cdot x + \theta_0$. This gives a measure of how far away the true values are from the predicted values, measured in units of mpg.

Note: One very important thing to keep in mind when employing standardization is that we need to reverse the standardization when we want to report results. If we standardize output values in the training set by subtracting μ and dividing by σ, we need to take care to:

1. Perform standardization with the same values of μ and σ on the test set (Why?) before predicting outputs using our learned predictor.
2. Multiply the RMSE calculated on the test set by a factor of σ to report test error (Why?)

Given all of this, we now will try using:

- Two choices of feature set:
    1. `[cylinders=standard, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`
    2. `[cylinders=one_hot, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`
- Polynomial features (we will construct the polynomial features after having standardized the input data) of orders 1-3
- Different choices of the regularization parameter, λ. Although, ideally, you would run a grid search over a large range of λ, we will ask you to look at the choices $\lambda = \{0.01, 0.02, \cdots, 0.1\}$ for polynomial features of orders 1 and 2, and the choices $\lambda = \{20, 40, \cdots, 200\}$ for polynomial features of order 3 (as this is approximately where we found the optimal λ to lie).

We will use 10-fold cross-validation to try all possible combinations of these feature choices and test which is best.

Your functions written above will be called by `ridge_min`, (defined for you below), which takes a dataset (X, y) and a hyperparameter, λ as input and returns θ and θ_0 minimizing the ridge regression objective using SGD (this is the analogue of the `svm_min` function that you wrote for homework last week). The learning rate and number of iterations are fixed in this function, and should not be modified for the purpose of answering the below questions (although you should feel free to experiment with these if you are interested!) This function will then further be called by `xval_learning_alg` (also defined below), which returns the average RMSE across all (here, 10) splits of your data when performing cross-validation.

Note: Even though these functions are also contained in the code file being imported (`code_for_hw5.py`), you should run the below code block so that they will use the version of the functions you have written above, and not the blank versions in the code file.

In [64]:
```
#Concatenates the gradients with respect to theta and theta_0
def ridge_obj_grad(x, y, th, th0, lam):
    grad_th = d_ridge_obj_th(x, y, th, th0, lam)
    grad_th0 = d_ridge_obj_th0(x, y, th, th0, lam)
    return np.vstack([grad_th, grad_th0])

def ridge_min(X, y, lam):
    """ Returns th, th0 that minimize the ridge regression objective

    Assumes that X is NOT 1-extended. Interfaces to our sgd by 1-extending
    and building corresponding initial weights.
    """
    def svm_min_step_size_fn(i):
        return 0.01/(i+1)**0.5

    d, n = X.shape
    X_extend = np.vstack([X, np.ones((1, n))])
```

```
    w_init = np.zeros((d+1, 1))

    def J(Xj, yj, th):
        return float(ridge_obj(Xj[:-1,:], yj, th[:-1,:], th[-1:,:], lam))

    def dJ(Xj, yj, th):
        return ridge_obj_grad(Xj[:-1,:], yj, th[:-1,:], th[-1:,:], lam)

    np.random.seed(0)
    w, fs, ws = sgd(X_extend, y, J, dJ, w_init, svm_min_step_size_fn, 1000)
    return w[:-1,:], w[-1:,:]

#First finds a predictor on X_train and X_test using the specified value of lam
#Then runs on X_test, Y_test to find the RMSE
def eval_predictor(X_train, Y_train, X_test, Y_test, lam):
    th, th0 = ridge_min(X_train, Y_train, lam)
    return np.sqrt(mean_square_loss(X_test, Y_test, th, th0))

#Returns the mean RMSE from cross validation given a dataset (X, y), a value of lam,
#and number of folds, k
def xval_learning_alg(X, y, lam, k):
    _, n = X.shape
    idx = list(range(n))
    np.random.seed(0)
    np.random.shuffle(idx)
    X, y = X[:,idx], y[:,idx]

    split_X = np.array_split(X, k, axis=1)
    split_y = np.array_split(y, k, axis=1)

    score_sum = 0
    for i in range(k):
        X_train = np.concatenate(split_X[:i] + split_X[i+1:], axis=1)
        y_train = np.concatenate(split_y[:i] + split_y[i+1:], axis=1)
        X_test = np.array(split_X[i])
        y_test = np.array(split_y[i])
        score_sum += eval_predictor(X_train, y_train, X_test, y_test, lam)
    return score_sum/k
```

In [ ]:

```
# Returns a list of dictionaries.  Keys are the column names, including mpg.
auto_data_all = hw5.load_auto_data('auto-mpg-regression.tsv')

# The choice of feature processing for each feature, mpg is always raw and
# does not need to be specified.  Other choices are hw5.standard and hw5.one_hot.
# 'name' is not numeric and would need a different encoding.
features1 = [('cylinders', hw5.standard),
             ('displacement', hw5.standard),
             ('horsepower', hw5.standard),
             ('weight', hw5.standard),
             ('acceleration', hw5.standard),
             ('origin', hw5.one_hot)]

features2 = [('cylinders', hw5.one_hot),
             ('displacement', hw5.standard),
             ('horsepower', hw5.standard),
             ('weight', hw5.standard),
             ('acceleration', hw5.standard),
             ('origin', hw5.one_hot)]

# Construct the standard data and label arrays
#auto_data[0] has the features for choice features1
#auto_data[1] has the features for choice features2
#The labels for both are the same, and are in auto_values
auto_data = [0, 0]
auto_values = 0
auto_data[0], auto_values = hw5.auto_data_and_values(auto_data_all, features1)
auto_data[1], _ = hw5.auto_data_and_values(auto_data_all, features2)

#standardize the y-values
auto_values, mu, sigma = hw5.std_y(auto_values)

#-------------------------------------------------------------------------------
# Analyze auto data
#-------------------------------------------------------------------------------

#Your code for cross-validation goes here
#Make sure to scale the RMSE values returned by xval_learning_alg by sigma,
#as mentioned in the lab, in order to get accurate RMSE values on the dataset


res = []
for feature_set_i in (0, 1):
    for order, lam_set in (
        (1, (np.arange(11))*0.01),
        (2, (np.arange(11))*0.01),
        (3, (np.arange(0,220,20)))
    ):
```

```
        print('feature_i:', feature_set_i)
        print('order:', order)
        print('lam_set:', lam_set)

        data = hw5.make_polynomial_feature_fun(order)(auto_data[feature_set_i])
        for lam in lam_set:
            print('lam', lam)
            mean_rmse = xval_learning_alg(data, auto_values, lam, 10) * sigma
            print(mean_rmse)
            res += [(mean_rmse, lam, order, feature_set_i)]
print(sorted(res)[:5])
```

```
feature_i: 0
order: 1
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[4.27349492]]
lam 0.01
[[4.27439892]]
lam 0.02
[[4.27535367]]
lam 0.03
[[4.27635832]]
lam 0.04
[[4.27741202]]
lam 0.05
[[4.27851395]]
lam 0.06
[[4.27966327]]
lam 0.07
[[4.28085918]]
lam 0.08
[[4.28210088]]
lam 0.09
[[4.28338758]]
lam 0.1
[[4.28471851]]
feature_i: 0
order: 2
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[4.02634119]]
lam 0.01
[[4.02714651]]
lam 0.02
[[4.02801017]]
lam 0.03
[[4.0289311]]
lam 0.04
```

```
[[4.02990823]]
lam 0.05
[[4.03094054]]
lam 0.06
[[4.03202697]]
lam 0.07
[[4.03316652]]
lam 0.08
[[4.03435819]]
lam 0.09
[[4.035601]]
lam 0.1
[[4.03689397]]
feature_i: 0
order: 3
lam_set: [  0  20  40  60  80 100 120 140 160 180 200]
lam 0
[[91158721.07809679]]
lam 20
[[6.47860272]]
lam 40
[[6.02418287]]
lam 60
[[6.03936845]]
lam 80
[[6.03256196]]
lam 100
[[6.03126881]]
lam 120
[[6.04675354]]
lam 140
[[6.07874785]]
lam 160
[[6.12733435]]
lam 180
[[6.19787407]]
lam 200
[[6.27648811]]
feature_i: 1
order: 1
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[4.14164798]]
lam 0.01
[[4.1431706]]
lam 0.02
[[4.1447585]]
lam 0.03
```

```
[[4.14641043]]
lam 0.04
[[4.14812515]]
lam 0.05
[[4.14990145]]
lam 0.06
[[4.15173813]]
lam 0.07
[[4.153634]]
lam 0.08
[[4.15558791]]
lam 0.09
[[4.15759869]]
lam 0.1
[[4.15966523]]
feature_i: 1
order: 2
lam_set: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 ]
lam 0.0
[[3.88436985]]
lam 0.01
[[3.88509423]]
lam 0.02
[[3.88586872]]
lam 0.03
[[3.88669241]]
lam 0.04
[[3.88756441]]
lam 0.05
[[3.88848386]]
lam 0.06
[[3.88944989]]
lam 0.07
[[3.89046164]]
lam 0.08
[[3.89151829]]
lam 0.09
[[3.89261901]]
lam 0.1
[[3.89376299]]
feature_i: 1
order: 3
lam_set: [  0  20  40  60  80 100 120 140 160 180 200]
lam 0
[[3741589.4937768]]
lam 20
[[5.73658168]]
lam 40
```

```
[[5.91137479]]
lam 60
[[5.99942908]]
lam 80
[[6.04674106]]
lam 100
[[6.08945111]]
lam 120
[[6.1372991]]
lam 140
[[6.19289135]]
lam 160
[[6.25670678]]
lam 180
[[6.32933722]]
lam 200
[[6.41186557]]
[(array([[3.88436985]]), 0.0, 2, 1), (array([[3.88509423]]), 0.01, 2, 1), (array([[3.88586872]]), 0.02, 2, 1), (array([[3.88669
241]]), 0.03, 2, 1), (array([[3.88756441]]), 0.04, 2, 1)]
```

You may find the lecture notes on regression helpful as you do this homework.

1) Intro to linear regression

So far, we have been looking at classification, where predictors are of the form

$$\hat{y}^{(i)} = sign(\theta^T x^{(i)} + \theta_0)$$

where $\hat{y}^{(i)}$ is our prediction of the corresponding label $y^{(i)}$ making a binary classification as to whether example $x^{(i)}$ belongs to the positive or negative class of examples.

In many problems, we want to predict a real value, such as the actual gas mileage of a car, or the concentration of some chemical. Luckily, we can use much of the mechanism we have already developed, and make predictors of the form:

$$\hat{y}^{(i)} = \theta^T x^{(i)} + \theta_0 \quad .$$

This is called a *linear regression* model.

We would like to learn a linear regression model from examples. Assume X is a d by n array (as before) but that Y is a 1 by n array of floating-point numbers (rather than +1 or -1). Given data (X, Y) we need to find θ, θ_0 that does a good job of making predictions on new data drawn from the same source.

We will approach this problem by formulating an objective function. There are many possible reasonable objective functions that implicitly make slightly different assumptions about the data, but they all typically have the form:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0)$$

For regression, we most frequently use *squared loss*, in which

$$L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) = (\hat{y}^{(i)} - y^{(i)})^2 = (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2 \ .$$

We might start by simply trying to minimize the average squared loss on the training data; this is called the *empirical risk* or *mean square error*:

$$J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) \ .$$

Later, we will add in a regularization term.

We will see later in this assignment that we can find a closed form matrix formula (requiring a matrix inverse) for the optimal θ in a linear regression formula. Being able to solve a machine-learning problem in closed form is very awesome! But inverting a matrix is computationally expensive (a bit less than $O(m^3)$ where m is the dimension of our matrix), and so, as our data sets get larger, we will need to find some more efficient or approximate ways to approach the problem.

For your convenience, we have copied the hands-on section into a colab notebook, which may be found here. You can alternatively fill in these functions in `code_for_hw5.py`, which is part of this set of files (the other files will be useful for the last part of the homework).

Let's start by thinking about gradient descent to attack this problem:

1A) What is the gradient of the empirical risk with respect to θ? We can see that it is of the form:

$$\nabla_\theta J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} g(x^{(i)}, y^{(i)})$$

where $x^{(i)}, y^{(i)}$ are the i^{th} data point and its label.

Write an expression for $g(x^{(i)}, y^{(i)})$ using the symbols: `x_i`, `y_i`, `theta` and `theta_0`, where $g(x^{(i)}, y^{(i)})$ is the derivative of the L_s function described above with respect to θ. Remember that you can use `@` for matrix product, and you can use `transpose(v)` to transpose a vector. Note that this $g(\cdot)$ function is *just* the derivative with respect to a single data point $x^{(i)}, y^{(i)}$. We'll build up to the gradient of J_{emp} in a moment.

$g(x^{(i)}, y^{(i)}) =$ 2*(transpose(theta)@x_i + theta_0 - y_i) * x_i

Check Syntax | Submit | View Answer | Ask for Help | 100.00%

You have infinitely many submissions remaining.

1B) What is the gradient of the empirical risk **now with respect to θ_0**? We can see that it is of a similar form:

$$\nabla_{\theta_0} J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} g_0(x^{(i)}, y^{(i)}) \ .$$

Write an expression for $g_0(x^{(i)}, y^{(i)})$ using the symbols: `x_i`, `y_i`, `theta` and `theta_0`.

$g_0(x^{(i)}, y^{(i)}) =$ 2*(transpose(theta)@x_i + theta_0 - y_i)

Check Syntax | Submit | View Answer | Ask for Help | 100.00%

You have infinitely many submissions remaining.

1C) Next we're interested in the gradient of the empirical loss with respect to θ, $\nabla_\theta J_{emp}$, but now for a whole data set X (of dimensions d by n).

Write an expression for $\nabla_\theta J_{emp}$ using the symbols: `X` and `Y` for the full set of data, `theta`, `theta_0`, and `n`. Here `X` has dimensions d by n, and `Y` has dimensions 1 by n. Remember that you can use `@` for matrix product, and you can use `transpose(a)` to transpose a vector or array.

$\nabla_\theta J_{emp} =$ 2/ n * X @ (transpose(X) @ theta + theta_0 - transpose(Y))

Check Syntax | Submit | View Answer | Ask for Help | 100.00%

You have infinitely many submissions remaining.

2) Sources of Error

Recall that *structural* error arises when the hypothesis class cannot represent a hypothesis that performs well on the test data and *estimation* error arises when the parameters of a hypothesis cannot be estimated well based on the training data. (You can also refer to here in the notes.)

Following is a collection of potential cures for a situation in which your learning algorithm generates a hypothesis with a high test error.

For each one, indicate whether it can **can reduce** structural error, estimation error, both, or neither.

2A) Penalize $||\theta||^2$ during training.

Can reduce:

- ○ structural error
- ◉ estimation error
- ○ both
- ○ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2B) Penalize $||\theta||^2$ during testing.

Can reduce:

- ○ structural error
- ○ estimation error
- ○ both
- ◉ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2C) Increase the amount of training data.

Can reduce:

- ○ structural error
- ◉ estimation error
- ○ both
- ○ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2D) Increase the order of a fixed polynomial basis.

Can reduce:

- ◉ structural error
- ○ estimation error
- ○ both
- ○ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

2E) Decrease the order of a fixed polynomial basis.

Can reduce:

- ○ structural error
- ◉ estimation error
- ○ both
- ○ neither

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3) Minimizing empirical risk

We can also solve regression problems analytically.

Remember the definition of *squared loss*,

$$L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) = (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2$$

and *empirical risk*:

$$J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_s(x^{(i)}, y^{(i)}, \theta, \theta_0)$$

Later, we will add in a regularization term.

For simplicity in this section, assume that we are handling the constant term, θ_0, by adding a dimension to the input feature vector that always has the value 1. To review why this works, take a look at the introduction to problem 1 in HW2.

Let data matrix $Z = X^T$ be n by d, let target output vector $T = Y^T$ be n by 1, and recall that θ is d by 1. Then we can write the whole linear regression prediction as $Z\theta$.

3A) T is the n by 1 vector of target output values. Write an equation expressing the mean squared loss of θ in terms of Z, T, n, and θ. Hint: note that this loss $J(\theta)$ is a scalar, a sum of squared terms divided by n; we can write it as $(W^T W)/n$ for a column vector W.

Enter your answer as a Python expression. You can use symbols `Z`, `T`, `n` and `theta`. Recall that our expression syntax includes `transpose(x)` for transpose of an array, `inverse(x)` for the inverse of an array, and `x@y` to indicate a matrix product of two arrays.

$J(\theta) =$ transpose(Z@theta - T)@(Z@theta - T)/n

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

Now, how can we find the minimizing θ, given Z and T? Take the gradient (yes, even with a matrix expression), set it to zero(s) and solve for θ.

3B) What is $\nabla_\theta J(\theta)$ in terms of Z, T, θ, and n? You can use matrix derivatives or, compute the answer for some individual elements and deduce the matrix form.

$\nabla_\theta J(\theta) =$ 2*transpose(Z)@(Z@theta - T)/n

Check Syntax | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3C) What if you set this equation to 0 and solve for θ^*, the optimal θ? Hint: It's ok to ignore the constant scaling factor.

$\theta^* =$

- ○ $(Z^T T)^{-1}(Z^T Z)$
- ◉ $(Z^T Z)^{-1} Z^T T$
- ○ $(Z Z^T)^{-1} Z T^T$

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3D) Just converting back to the data matrix format we have been using (not transposed), we have

$\theta^* =$

- ○ $(X Y^T)^{-1}(X X^T)$
- ○ $(X^T X)^{-1} X^T Y$
- ◉ $(X X^T)^{-1} X Y^T$

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

3E) Now implement θ^* as found in **3D)**, using symbols `X` and `Y` for the data matrix and outputs, and `np.dot`, `np.transpose`, `np.linalg.inv`.

```
# Enter an expression to compute and set th to the optimal theta
th = np.linalg.inv(X@np.transpose(X)) @ (X@np.transpose(Y))
```

Run Code | Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

4) Adding regularization

Although we don't have the same notion of margin maximization as with the SVM formulation for classification, there is still a good reason to *regularize* or put pressure on the coefficient vector θ to prevent the model from fitting the training data too closely, especially in cases where we have few data points and many features.

And, as it happens, this same regularization will help address a problem that you might have anticipated when finding the analytical solution for θ, which is that XX^T might not be invertible (where we are using the definition of X as in problem 3, where each column of X is a d-length vector representing a d-feature sample point and there are n columns in X (or equivalently, there are n training examples)).

Consider this matrix:

```
X = np.array([[1, 2], [2, 3], [3, 5], [1, 4]])
```

Is XX^T invertible? If not, what's the problem? Mark all that are true.

- [ ] It is invertible
- [ ] It is not invertible because X is not square
- [ ] It is not invertible because two columns of X are linearly dependent
- [x] It is not invertible because the rows of XX^T are linearly dependent
- [x] It is not invertible because n is smaller than d
- [ ] We cannot compute the transpose of X

Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

5) Evaluation

We have been hired by Buy 'n' Large to deliver a predictor of change in sales volume from last year, for each of their stores. We have a machine-learning algorithm that can be used with regularization parameter λ. Our overall objective is to deliver a predictor that minimizes squared loss on predictions when actually used by the company. We have three data sets: D_{train}, D_{test} and D_{real}, each of size n. The D_{real} is owned by the company.

We will focus on a linear predictor with parameters θ without the offset parameter θ_0 for simplicity and use regularizer $\lambda||\theta||^2$, where λ is the regularization parameter. There are several phases of the training process (as represented in problems 5A through 5D below), and we need to select the appropriate objective for each of those tasks. In the problems below, we will have different expressions with different "slots" (A, B, C, D) to fill from, picking among the following options available to us related to

- what the minimization is over,
- the dataset used,
- the predictor used, and
- whether regularization is added.

Fill in the slots (A,B,C,D) for each phase by choosing the expressions for the indicated slots. The available expressions are shown below; please enter the

index for the expression for each of the slots.

1. 0
2. θ
3. $\theta_{best}(\lambda)$,
4. θ^*,
5. λ,
6. λ^*,
7. $\lambda||\theta||^2$,
8. $\lambda^*||\theta||^2$,
9. D_{train},
10. D_{test},
11. $D_{train} \cup D_{test}$,
12. D_{real}

Note that $\theta_{best}(\lambda)$ is a value of θ that is a function of λ; θ^*, λ^* are specific values found as described below; θ and λ are variables that range over d -dimensional column vectors and positive reals, respectively.

5A) Selecting the best hypothesis (parameters θ) for some fixed value of the regularization parameter λ. Call this $\theta_{best}(\lambda)$.

$$\theta_{best}(\lambda) = \arg\min_{A} \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 4 indices for [A, B, C, D]: [2, 9, 2, 7]

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `[2, 9, 2, 7]`

Explanation:

To obtain our value of $\theta_{best}(\lambda)$ for a given λ, we need to find the value of θ that minimizes the loss function while λ is fixed. This means that the parameter that we will be using for argmin is θ.

We are trying to find the best θ for a given λ so that we can later compare optimal models with each other using other data, so to obtain this θ and in order to have unseen data to use later, we must use D_{train} to minimize loss.

We are optimizing over values of θ in order to minimize the loss function, so we must use θ inside of our loss function.

Lastly, we need to regularize with λ, so our regularization term is $\lambda||\theta||^2$.

5B) Selecting the best value of the regularization parameter λ. We will call this best value λ^*.

$$\lambda^* = \arg\min_A \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 4 indices for [A, B, C, D]: [5, 10, 3, 1]

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `[5, 10, 3, 1]`

Explanation:

In the previous part of this problem, we were working on finding an optimal model θ for a given hyperparameter λ. In this part, we are trying to optimize our hyperparameter given multiple models, each one corresponding to an optimal model found with a value of λ we are considering. Therefore, to obtain our value of λ^*, we are optimizing the loss over values of λ. This means that the parameter that we will be using for argmin is λ.

To determine the best value of λ given an optimal model, we will need to observe the performance of the model with unseen data. Thus, it is necessary to use D_{test} as the data with which to minimize loss.

As noted before, we are trying to optimize the λ hyperparameter based on optimal models found with each one, so the model used while minimizing loss must be $\theta_{best}(\lambda)$.

Lastly, there is no need to regularize with anything as no training is being done (we are only comparing performance on the test data).

5C) Selecting the hypothesis (parameters θ) to deliver to the company. Call this θ^*.

$$\theta^* = \arg\min_A \frac{1}{|B|} \sum_{(x,y)\in B} (C^T x - y)^2/2 + D$$

Enter a list of 4 indices for [A, B, C, D]: [2, 11, 2, 8]

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `[2, [9, 10, 11], 2, 8]`

Explanation:

To find the value θ^* that minimizes our objective function, we want to search over all θ values. In the regularization term, we should use our best regularization parameter λ^*, which we found in the previous step.

All three datasets are accepted for B. You may choose $D_{train} \cup D_{test}$ because this allows us to use all the data we have. However, you may argue that D_{train} is the actual training data that was used to find the $\theta_{best}(\lambda)$ values in the first place. On the other hand, D_{test} is "unseen" and might generalize better.

5D) Evaluating the actual on-the-job performance ϵ^* of the selected hypothesis θ^*.

$$\epsilon^* = \frac{1}{|B|} \sum_{(x,y) \in B} (C^T x - y)^2 / 2 + D$$

Enter a list of 3 indices for [B, C, D]: [12, 4, 1]

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `[12, 4, 1]`

Explanation:

To evaluate actual performance, we must run our model on real data, or D_{real}. We want to use our selected hypothesis, θ^*, for our predictor. Because we are performing evaluation rather than training, we have no need for regularization and want only loss, so D is 0.

6) Linear regression - going downhill

We will now write some general Python code to compute the gradient of the squared-loss objective, following the structure of the expression and the rules of calculus. Note that this style of writing the gradient functions maps directly into the chain-rule steps required to compute the gradient, but produces code that is inefficient, because of duplicated computations. It is straightforward to implement more efficient versions if you want to use them for larger problems.

Reminder: For your convenience, we have copied the hands-on section into a colab notebook, which may be found here.. Alternatively, you can work with these functions on your own computer in `code_for_hw5.py`, contained in this zip file. That file has somewhat longer docstrings and doctests for many of these functions and other basic utiliities, that may be useful to you in debugging your implementations. (The other files therein will be useful in the last part of this homework).

We start by defining some basic functions for computing the mean squared loss. Note that we want these to work for any value of n, that is, `x` could be a single feature vector (of dimension d by 1) or a full data matrix (of dimension d by n), and similarly for `y`.

```
# In all the following definitions:
# x is d by n : input data
```

```
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar

def lin_reg(x, th, th0):
    return np.dot(th.T, x) + th0

def square_loss(x, y, th, th0):
    return (y - lin_reg(x, th, th0))**2

def mean_square_loss(x, y, th, th0):
    # the axis=1 and keepdims=True are important when x is a full matrix
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True)
```

These functions will already be defined when you are answering the questions below.

Warm up:

6A)

If X is d by n and Y is 1 by n, what is the dimension of θ? d by 1

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

6B)

If X is d by n and Y is 1 by n, what is the dimension of $\nabla_\theta J_{emp}(\theta, \theta_0)$? d by 1

Submit View Answer Ask for Help 100.00%

You have infinitely many submissions remaining.

6C) Now let's compute the gradients with respect to θ, make sure that they work for data matrices and label vectors. You can write one function at a time,

some of the checks will apply to each function independently.

In the code below, the following values are used in the test cases:

```
X = np.array([[1., 2., 3., 4.], [1., 1., 1., 1.]])
Y = np.array([[1., 2.2, 2.8, 4.1]])
th = np.array([[1.0],[0.05]])
th0 = np.array([[0.]])
```

```
# Write a function that returns the gradient of lin_reg(x, th, th0)
# with respect to th
def d_lin_reg_th(x, th, th0):
    """ Returns the gradient of lin_reg(x, th, th0) with respect to th

    Note that for array (rather than vector) x, we get a d x n
    result. That is to say, this function produces the gradient for
    each data point i ... n, with respect to each theta, j ... d.

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]); th0 = np.array([[ 2.]])
    >>> d_lin_reg_th(X[:,0:1], th, th0).tolist()
    [[1.0], [1.0]]

    >>> d_lin_reg_th(X, th, th0).tolist()
    [[1.0, 2.0, 3.0, 4.0], [1.0, 1.0, 1.0, 1.0]]
    """
    return x

```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

6D) Now let's compute the gradients with respect to θ_0, make sure that they work for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently. The test cases will include example variables for `X`, `Y`, `th`, and `th0` from 6C above.

```python
# Write a function that returns the gradient of lin_reg(x, th, th0)
# with respect to th0. Hint: Think carefully about what the dimensions of the returned value shoul
def d_lin_reg_th0(x, th, th0):
    """ Returns the gradient of lin_reg(x, th, th0) with respect to th0.

    >>> x = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> th = np.array([[ 1.  ], [ 0.05]]) ; th0 = np.array([[ 2.]])
    >>> d_lin_reg_th0(x, th, th0).tolist()
    [[1.0, 1.0, 1.0, 1.0]]
    """
    d, n = x.shape
    return np.repeat(1.0, n).reshape((1,n))

# Write a function that returns the gradient of square_loss(x, y, th, th0) with
# respect to th0.  It should be a one-line expression that uses lin_reg and
# d_lin_reg_th0.
def d_square_loss_th0(x, y, th, th0):
    """ Returns the gradient of square_loss(x, y, th, th0) with
        respect to th0.
```

Run Code | Submit | View Answer | Ask for Help 100.00%

You have infinitely many submissions remaining.

7) Going down the ridge

Now, let's add a regularizer. The ridge objective can be implemented as follows:

```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def ridge_obj(x, y, th, th0, lam):
```

```
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True) + lam * np.linalg.norm(th)**2
```

Let's extend our previous code for the gradient of the mean square loss to compute the gradient of the ridge objective with respect to θ. Our previous solutions for the non-ridge case: `d_mean_square_loss_th` and `d_mean_square_loss_th0` are defined for you and you can call them. The test cases will include example variables for `X`, `Y`, `th`, and `th0` from 6C above.

```
def d_ridge_obj_th(x, y, th, th0, lam):
    return None

def d_ridge_obj_th0(x, y, th, th0, lam):
    return None
def d_ridge_obj_th(x, y, th, th0, lam):
    """Return the derivative of tghe ridge objective value with respect
    to theta.

    Note: uses broadcasting to add d x n to d x 1 array below

    >>> X = np.array([[ 1.,  2.,  3.,  4.], [ 1.,  1.,  1.,  1.]])
    >>> Y = np.array([[ 1. ,  2.2,  2.8,  4.1]])
```

Run Code | Submit | View Answer | Ask for Help — 100.00%

You have infinitely many submissions remaining.

8) Stochastic gradient

We will now implement stochastic gradient descent in a general way, similar to what we did with gradient descent (`gd`).

`sgd` takes the following as input: (Recall that the *stochastic* part refers to using a randomly selected point and corresponding label from the given dataset to perform an update. Therefore, your objective function for a given step will need to take this into account.)

- `X`: a standard data array (d by n)
- `y`: a standard labels row vector (1 by n)

- `J`: a cost function whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector) (in that order), and which returns a scalar.
- `dJ`: a cost function gradient (corresponding to `J`) whose input is a data point (a column vector), a label (1 by 1) and a weight vector `w` (a column vector) (also in that order), and which returns a column vector.
- `w0`: an initial value of weight vector w, which is a column vector.
- `step_size_fn`: a function that is given the (zero-indexed) iteration index (an integer) and returns a step size.
- `max_iter`: the number of iterations to perform

It returns a tuple (like `gd`):

- `w`: the value of the weight vector at the final step
- `fs`: the list of values of J found during all the iterations
- `ws`: the list of values of w found during all the iterations

Note: `w` should be the value one gets after applying stochastic gradient descent to `w0` for `max_iter-1` iterations (we call this the final step). The first element of `fs` should be the value of `J` calculated with `w0`, and `fs` should have length `max_iter`; similarly, the first element of `ws` should be `w0`, and `ws` should have length `max_iter`.

You might find the function `np.random.randint(n)` useful in your implementation.

Hint: This is a short function; our implementation is around 10 lines.

The test cases are:

```
def downwards_line():
    X = np.array([[0.0, 0.1, 0.2, 0.3, 0.42, 0.52, 0.72, 0.78, 0.84, 1.0],
                  [1.0, 1.0, 1.0, 1.0, 1.0,  1.0,  1.0,  1.0,  1.0,  1.0]])
    y = np.array([[0.4, 0.6, 1.2, 0.1, 0.22, -0.6, -1.5, -0.5, -0.5, 0.0]])
    return X, y

X, y = downwards_line()

def J(Xi, yi, w):
    # translate from (1-augmented X, y, theta) to (separated X, y, th, th0) format
    return float(ridge_obj(Xi[:-1,:], yi, w[:-1,:], w[-1:,:], 0))
```

```
def dJ(Xi, yi, w):
    def f(w): return J(Xi, yi, w)
    return num_grad(f)(w)
```

where `num_grad` is taken from homework from the previous week:

```
def num_grad(f):
    def df(x):
        g = np.zeros(x.shape)
        delta = 0.001
        for i in range(x.shape[0]):
            xi = x[i,0]
            x[i,0] = xi - delta
            xm = f(x)
            x[i,0] = xi + delta
            xp = f(x)
            x[i,0] = xi
            g[i,0] = (xp - xm)/(2*delta)
        return g
    return df
```

```
    vector.

    step_size_fn: a function that is given the (zero-indexed)
    iteration index (an integer) and returns a step size.

    max_iter: the number of iterations to perform

    Returns: a tuple (like gd):
    w: the value of the weight vector at the final step
    fs: the list of values of JJJ found during all the iterations
    ws: the list of values of www found during all the iterations

    """
    d, n = X.shape
```

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Here is the solution we wrote:

```
def sgd(X, y, J, dJ, w0, step_size_fn, max_iter):
    n = y.shape[1]
    prev_w = w0
    fs = []; ws = []
    np.random.seed(0)
    for i in range(max_iter):
        j = np.random.randint(n)
        Xj = X[:,j:j+1]; yj = y[:,j:j+1]
        prev_f, prev_grad = J(Xj, yj, prev_w), dJ(Xj, yj, prev_w)
        fs.append(prev_f); ws.append(prev_w)
        if i == max_iter - 1:
            return prev_w, fs, ws
        step = step_size_fn(i)
        prev_w = prev_w - step * prev_grad
```

9) Predicting mpg values

We will now try to synthesize the functions we have written in order to perform ridge regression on the auto-mpg dataset from lab03. Unlike in lab03, we will now try to predict the actual mpg values of the cars, instead of whether they are above or below the median mpg!

As a reminder, the dataset is as follows:

```
1. mpg:           continuous
2. cylinders:     multi-valued discrete
3. displacement:  continuous
4. horsepower:    continuous
5. weight:        continuous
6. acceleration:  continuous
7. model year:    multi-valued discrete
8. origin:        multi-valued discrete
9. car name:      string (many values)
```

For convenience, we will choose to not include `model year` and `car name` as features. For the remaining features, we again have the option to keep the raw values, standardize them, or use a one-hot encoding.

9A) What is true about leaving features as raw versus deciding to standardize them, in the context of linear regression without regularization?

- [ ] The set of all possible linear regression models learnable on standardized features is smaller than the set of linear regression models learnable on raw features
- [x] The theoretical minimum value of the loss function is the same both with raw and standardized features
- [x] SGD will typically perform better on standardized features than on raw features.

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

9B) What is true about leaving features as raw versus deciding to standardize them, in the context of ridge regression (i.e., we have a nonzero regularizer)?

- [ ] The set of all possible models learnable on standardized features is smaller than the set of models learnable on raw features
- [ ] The theoretical minimum value of the loss function is the same both with raw and standardized features
- [x] SGD will typically perform better on standardized features than on raw features.

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

With this considered, we decide to standardize or one-hot encode all features in this section (we encourage you, though, to try raw features on your own to see how their performance matches your expectations!).

One additional step we perform is to standardize the output values. Note that we did not have to worry about this in a classification context, as all outputs were ± 1. In a regression context, standardizing the output values can have practical performance gains, again due to better numerical performance of learning algorithms on data that is in a good magnitude range.

The metric we will use to measure the quality of our learned predictors is **Root Mean Square Error (RMSE).** This is useful metric because it gives a sense of the deviation in the nature units of the predictor. RMSE is defined as follows:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f(x^{(i)})\right)^2}$$

where f is our learned predictor: in this case, $f(x) = \theta \cdot x + \theta_0$. This gives a measure of how far away the true values are from the predicted values; we are interested in this value, measured in units of mpg.

Note: One very important thing to keep in mind when employing standardization is that we need to reverse the standardization when we want to report results. If we standardize output values in the training set by subtracting μ and dividing by σ, we need to take care to:

1. Perform standardization with the same values of μ and σ on the test set (Why?) before predicting outputs using our learned predictor.
2. Multiply the RMSE calculated on the test set by a factor of σ to report test error. (Why?)

Given all of this, we now will try using:

- Two choices of feature set:

1. `[cylinders=standard, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`
2. `[cylinders=one_hot, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`

- Polynomial features (we will construct the polynomial features after having standardized the input data) of orders 1-3

- Different choices of the regularization parameter, λ. Although, ideally, you would run a grid search over a large range of λ, we will ask you to look at the choices $\lambda = \{0.0, 0.01, 0.02, \cdots, 0.1\}$ for polynomial features of orders 1 and 2, and the choices $\lambda = \{0, 20, 40, \cdots, 200\}$ for polynomial features of order 3 (as this is approximately where we found the optimal λ to lie).

We will use 10-fold cross-validation to try all possible combinations of these feature choices and test which is best. We have attached a code file with some predefined methods that will be useful to you here. Alternatively, a google colab link may be found here. If you choose to use the code file, a more detailed description of the roles of the files is below:

The file `code_for_hw5.py` contains functions, some of which will need to be filled in with your definitions from this homework. Your functions are then called by `ridge_min`, defined for you, which takes a dataset (X, y) and a hyperparameter, λ as input and returns θ and θ_0 minimizing the ridge regression objective using SGD (this is the analogue of the `svm_min` function that you wrote for homework last week). The learning rate and number of iterations are fixed in this function, and should not be modified for the purpose of answering the below questions (although you should feel free to experiment with these if you are interested!). This function will then further be called by `xval_learning_alg` (also defined for you in the same file), which returns the average RMSE across all (here, 10) splits of your data when performing cross-validation. (Note that this RMSE is reported in standardized `y` units; to convert this to RMSE in mpg (miles per gallon), you should multiply this by the `sigma` returned by the `hw5.std_y` function call.)

The file `auto.py` will be used to implement the auto data regression. The file contains code for creating the two feature sets that you are asked to work with here. Transforming those features further with `make_polynomial_feature_fun`, and running the cross-validation function, which uses your implementations in `code_for_hw5.py` (both from `code_for_hw5.py`), you should be able to answer the following questions:

9C) What combination minimizes the average cross-validation RMSE?

Enter a tuple of three numbers (feature_set, polynomial_order, lambda): (2, 2, 0)

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

9D) What is the cross-validation RMSE value (in mpg) that you obtain using the best combination?

Enter an accuracy value: 3.88436985

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.

9E) Say that we really wanted to fit an order 3 polynomial model using the first feature set. What value of lambda minimizes the average cross-validation RMSE, and what is the RMSE value at that value of lambda?

Enter a python list of two numbers [lambda, RMSE_value]: [40, 6.02418287]

Submit | View Answer | Ask for Help **100.00%**

You have infinitely many submissions remaining.