Training Logistic Regression via Stochastic Gradient Ascent

The goal of this assignment is to implement a logistic regression classifier using stochastic gradient ascent. You will:

- Extract features from Amazon product reviews.
- Convert an SFrame into a NumPy array.
- Write a function to compute the derivative of log likelihood function (with L2 penalty) with respect to a single coefficient.
- Implement stochastic gradient ascent with L2 penalty
- Compare convergence of stochastic gradient ascent with that of batch gradient ascent

If you are doing the assignment with IPython Notebook

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

Make sure that you are using GraphLab Create 1.8.3. See this post for installing the correct version of GraphLab Create.

What you need to download

If you are using GraphLab Create:

- Download the Amazon product review dataset (subset) in SFrame format. Notice the subset suffix:amazon_baby_subset.gl.zip
- Download the companion IPython notebook: module-10-online-learning-assignment-blank.ipynb
- Download the list of 193 significant words: important_words.json
- If you are using Amazon EC2, download the binary files for NumPy arrays: module-10-assignment-numpy-arrays.npz. See the companion notebook for the instructions.

- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

If you are not using GraphLab Create:

- If you are using SFrame, download the the Amazon product review dataset (subset) in SFrame format:amazon_baby_subset.gl.zip
- If you are using a different package, download the Amazon product review dataset (subset) in CSV format:amazon_baby_subset.csv.zip
- Download the list of 193 significant words: important_words.json

If you are using GraphLab Create and the companion IPython Notebook

Open the companion IPython notebook and follow the instructions in the notebook.

If you are using other tools

This section is designed for people using tools other than GraphLab Create. **You will not need any machine learning packages** since we will be implementing decision trees from scratch. **We highly suggest you useSFrame since it is open source.** In this part of the assignment, we describe general instructions, however we will tailor the instructions for SFrame.

- If you choose to use SFrame, you should be able to follow the instructions in the next section and complete the assessment. **All code samples given here will be applicable to SFrame**.
- You are free to experiment with any tool of your choice, but **some many not produce correct numbers for the quiz questions.**

If you are using SFrame

Make sure to download the companion IPython notebook: module-10-online-learning-assignment-blank.ipynb. You will be able to follow along exactly **if you replace the first two lines of code with these two lines:**

```
import sframe
```

```
products = sframe.SFrame('amazon_baby_subset.gl/')
```

After running this, **you can follow the rest of the IPython notebook and disregard the rest of this reading.**

Note: To install SFrame (without installing GraphLab Create), run

```
pip install sframe
```

If you are NOT using SFrame

Load and process review dataset

1. For this assignment, we will use the same subset of the Amazon product review dataset that we used in Module 3 assignment. The subset was chosen to contain similar numbers of positive and negative reviews, as the original dataset consisted of mostly positive reviews. Load the data file into a data frame **products**.

Apply text cleaning on the review data

2. In this section, we will perform some simple feature cleaning using data frames. The last assignment used all words in building bag-of-words features, but here we limit ourselves to 193 words (for simplicity). We compiled a list of 193 most frequent words into the JSON file named **important_words.json**. Load the words into a list**important_words**.

3. Let us perform 2 simple data transformations:

- Remove punctuation
- Compute word counts (only for **important_words**)

We start with the first item as follows:

- If your tool supports it, fill n/a values in the review column with empty strings. The n/a values indicate empty reviews. For instance, Pandas's the fillna() method lets you replace all N/A's in the review columns as follows:

```
products = products.fillna({'review':''})  # fill in N/A's in the review column
```

- Write a function **remove_punctuation** that takes a line of text and removes all punctuation from that text. The function should be analogous to the following Python code:

```
def remove_punctuation(text):
   import string
   return text.translate(None, string.punctuation)
```

- Apply the **remove_punctuation** function on every element of the **review** column and assign the result to the new column **review_clean**. **Note.** Many data frame packages support **apply** operation for this type of task. Consult appropriate manuals.

4. Now we proceed with the second item. For each word in **important_words**, we compute a count for the number of times the word occurs in the review. We will store this count in a separate column (one for each word). The result of this feature processing is a single column for each word in **important_words** which keeps a count of the number of times the respective word occurs in the review text.

Note: There are several ways of doing this. One way is to create an anonymous function that counts the occurrence of a particular word and apply it to every element in the **review_clean** column. Repeat this step for every word in **important_words**. Your code should be analogous to the following:

```
for word in important_words:
   products[word] = products['review_clean'].apply(lambda s : s.split().count(word))
```

5. After #4 and #5, the data frame **products** should contain one column for each of the 193 **important_words**. As an example, the column **perfect** contains a count of the number of times the word **perfect** occurs in each of the reviews.

Split data into training and validation sets

6. We will now split the data into a 90-10 split where 90% is in the training set and 10% is in the validation set. We use seed=1 so that everyone gets the same result.

```
train_data, validation_data = products.random_split(.9, seed=1)
```

If you are not using SFrame, download the list of indices for the training and validation sets: module-10-assignment-train-idx.json, module-10-assignment-validation-idx.json. IMPORTANT: If you are using a programming language with 1-based indexing (e.g. R, Matlab), make sure to increment all indices by 1.

Call the training and validation sets **train_data** and **validation_data**, respectively.

7. Convert **train_data** and **validation_data** into multi-dimensional arrays.

Using the function given in #8 of Module 3 assignment, extract two arrays **feature_matrix_train** and**sentiment_train** from **train_data**. The 2D array **feature_matrix_train** would contain the content of the columns given by the list **important_words**. The 1D array **sentiment_train** would contain the content of the column**sentiment**. Do the same for **validation_data**, producing the arrays **feature_matrix_valid** and **sentiment_valid**. The code should be analogous to this cell:

```
feature_matrix_train, sentiment_train = get_numpy_data(train_data, important_words, 'sentiment')
feature_matrix_valid, sentiment_valid = get_numpy_data(validation_data, important_words, 'sentiment')
```

Quiz question: In Module 3 assignment, there were 194 features (an intercept + one feature for each of the 193 important words). In this assignment, we will use stochastic gradient ascent to train the classifier using logistic regression. How does the changing the solver to stochastic gradient ascent affect the number of features?

Building on logistic regression

8. Let us now build on Module 3 assignment. Recall from lecture that the link function for logistic regression can be defined as:

$$P(y_i = +1 | \mathbf{x}_i, \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{w}^T h(\mathbf{x}_i))}$$

where the feature vector h(x_i) is given by the word counts of **important_words** in the review x_i.

We will use the **same code** as in Module 3 assignment to make probability predictions, since this part is not affected by using stochastic gradient ascent as a solver. Only the way in which the coefficients are learned is affected by using stochastic gradient ascent as a solver. Refer to #10 of Module 3 assignment in order to obtain the function **predict_probability**.

Derivative of log likelihood with respect to a single coefficient

9. Let us now work on making minor changes to how the derivative computation is performed for logistic regression.

Recall from the lectures and Module 3 assignment that for logistic regression, **the derivative of log likelihood with respect to a single coefficient** is as follows:

$$\frac{\partial \ell}{\partial w_j} = \sum_{i=1}^{N} h_j(\mathbf{x}_i) \left(\mathbf{1}[y_i = +1] - P(y_i = +1 | \mathbf{x}_i, \mathbf{w}) \right)$$

Recall that, in the Module 3 assignment, we wrote the function **feature_derivative** to compute the derivative of log likelihood with respect to a single coefficient w_j. Refer to #11 of Module 3 assignment to obtain the function**feature_derivative**.

Note. We are not using regularization in this assignment, but, as discussed in the optional video, stochastic gradient can also be used for regularized logistic regression.

10. To verify the correctness of the gradient computation, we provide a function for computing average log likelihood (which we recall from the last assignment was a topic detailed in an advanced optional video, and used here for its numerical stability).

To track the performance of stochastic gradient ascent, write yourself a function to compute **average log likelihood**. The average log likelihood is given by the formula

$$\ell\ell_A(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \Big((\mathbf{1}[y_i = +1] - 1)\mathbf{w}^T h(\mathbf{x}_i) - \ln\left(1 + \exp(-\mathbf{w}^T h(\mathbf{x}_i))\right) \Big)$$

Call this function **compute_avg_log_likelihood**. It should be analogous to the following Python function:

```
def compute_avg_log_likelihood(feature_matrix, sentiment, coefficients):


    indicator = (sentiment==+1)
    scores = np.dot(feature_matrix, coefficients)
    logexp = np.log(1. + np.exp(-scores))


    # Simple check to prevent overflow
    mask = np.isinf(logexp)
```

```
logexp[mask] = -scores[mask]

lp = np.sum((indicator-1)*scores - logexp)/len(feature_matrix)

return lp
```

Note. We made one tiny modification to the log likelihood function (called **compute_log_likelihood**) in our earlier assignments. We added a 1/N term which averages the log likelihood accross all data points. The 1/N term makes it easier for us to compare stochastic gradient ascent with batch gradient ascent. We will use this function to generate plots that are similar to those you saw in the lecture.

Quiz question: Recall from the lecture and the earlier assignment, the log likelihood (without the averaging term) is given by

$$\ell\ell(\mathbf{w}) = \sum_{i=1}^{N} \Big((\mathbf{1}[y_i = +1] - 1)\mathbf{w}^T h(\mathbf{x}_i) - \ln\left(1 + \exp(-\mathbf{w}^T h(\mathbf{x}_i))\right) \Big)$$

How are the functions ll(w) and ll_A(w) related?

Modifying the derivative for stochastic gradient ascent

11. Recall from the lecture that the gradient for a single data point x_i can be computed using the following formula:

$$\frac{\partial \ell_i(\mathbf{w})}{\partial w_j} = h_j(\mathbf{x}_i)\left(\mathbf{1}[y_i = +1] - P(y_i = +1 | \mathbf{x}_i, \mathbf{w})\right)$$

Computing the gradient for a single data point

Do we really need to re-write all our code to modify

$$\frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

to

$$\frac{\partial \ell_i(\mathbf{w})}{\partial w_j}$$

? Thankfully **No!** We access **x_i** in the training data using **feature_matrix_train[i:i+1,:]**. Similarly, we access **y_i** in the training data using **sentiment_train**[i:i+1]. We can compute

$$\frac{\partial \ell_i(\mathbf{w})}{\partial w_j}$$

by re-using all the code written in **feature_derivative** and **predict_probability**.

Carry out the following steps:

- First, compute P(y_i = +1 | x_i, w) using the **predict_probability** function with **feature_matrix_train**[i:i+1,:] as the first parameter.
- Next, compute the indicator value 1[y_i = +1] using **sentiment_train**[i:i+1].
- Finally, call the **feature_derivative** function with **feature_matrix_train**[i:i+1,j] as one of the parameters.

The following Python cell carries out the steps for j=1 and i=10. (Try this in your own tool.)

```
j = 1                    # Feature number
i = 10                   # Data point number
coefficients = np.zeros(194) # A point w at which we are computing the gradient.

predictions = predict_probability(feature_matrix_train[i:i+1,:], coefficients)
indicator = (sentiment_train[i:i+1]==+1)

errors = indicator - predictions
gradient_single_data_point = feature_derivative(errors, feature_matrix_train[i:i+1,j])
print "Gradient single data point: %s" % gradient_single_data_point
print "           --> Should print 0.0"
```

Quiz Question: The code block above computed the derivative

$$\frac{\partial \ell_i(\mathbf{w})}{\partial w_j}$$

for j = 1 and i = 10. Is this a scalar or a 194-dimensional vector?

Modifying the derivative for using a batch of data points

12. Stochastic gradient estimates the ascent direction using 1 data point, while gradient uses N data points to decide how to update the the parameters. In an optional video, we discussed the details of a simple change that allows us to use a **mini-batch** of B<=N data points to estimate the ascent direction. This simple approach is faster than regular gradient but less noisy than stochastic gradient that uses only 1 data point. Although we encorage you to watch the optional video on the topic to better understand why mini-batches help stochastic gradient, in this assignment, we will simply use this technique, since the approach is very simple and will improve your results.

Given a mini-batch (or a set of data points) x_i,x_(i+1), …, x_(i+B), the gradient function for this mini-batch of data points is given by:

$$\sum_{s=i}^{i+B} \frac{\partial \ell_s}{\partial w_j} = \sum_{s=i}^{i+B} h_j(\mathbf{x}_s)\left(\mathbf{1}[y_s = +1] - P(y_s = +1 | \mathbf{x}_s, \mathbf{w})\right)$$

Computing the gradient for a "mini-batch" of data points

We access the points x_i,x_(i+1), …, x_(i+B) in the training data using **feature_matrix_train[i:i+B,:]** and y_i in the training data using **sentiment_train[i:i+B]**.

Write come code to compute

$$\sum_{s=i}^{i+B} \frac{\partial \ell_s}{\partial w_j}$$

for i = 10, j = 1, and B = 10.

Your code should be equivalent to the following Python cell:

```
j = 1                        # Feature number
i = 10                       # Data point start
B = 10                       # Mini-batch size
coefficients = np.zeros(194) # A point w at which we are computing the gradient.

predictions = predict_probability(feature_matrix_train[i:i+B,:], coefficients)
indicator = (sentiment_train[i:i+B]==+1)

errors = indicator - predictions
gradient_mini_batch = feature_derivative(errors, feature_matrix_train[i:i+B,j])
print "Gradient mini-batch data points: %s" % gradient_mini_batch
print "                --> Should print 1.0"
```

Quiz Question: The code block above computed

$$\sum_{s=i}^{i+B} \frac{\partial \ell_s}{\partial w_j}$$

for j = 10, i = 10, and B = 10. Is this a scalar or a 194-dimensional vector?

Quiz Question: For what value of B is the term

$$\sum_{s=1}^{B} \frac{\partial \ell_s(\mathbf{w})}{\partial w_j}$$

the same as the full gradient

$$\frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

?

Averaging the gradient across a batch

13. It is a common practice to normalize the gradient update rule by the batch size B:

$$\frac{\partial \ell_A(\mathbf{w})}{\partial w_j} \approx \frac{1}{B} \sum_{s=i}^{i+B} h_j(\mathbf{x}_s) \left(\mathbf{1}[y_s = +1] - P(y_s = +1 | \mathbf{x}_s, \mathbf{w})\right)$$

In other words, we update the coefficients using the **average gradient over data points** (instead of using a summation). By using the average gradient, we ensure that the magnitude of the gradient is approximately the same for all batch sizes. This way, we can more easily compare various batch sizes of stochastic gradient ascent (including a batch size of **all the data points**), and study the effect of batch size on the algorithm as well as the choice of step size.

Implementing stochastic gradient ascent

14. Now we are ready to implement our own logistic regression with stochastic gradient ascent. The function**logistic_regression_SG** should accept the following parameters:

- **feature_matrix**: 2D array of features
- **sentiment**: 1D array of class labels
- **initial_coefficients**: 1D array containing initial values of coefficients
- **step_size**: a parameter controlling the size of the gradient steps
- **batch_size**: size of mini-batch
- **max_iter**: number of iterations to run stochastic gradient ascent

The function should return the final set of coefficients, along with the list of log likelihood values over time. (In practice, the final set of coefficients is rarely used; it is better to use the average of the last K sets of coefficients instead, where K should be adjusted depending on how fast the log likelihood oscillates around the optimum.)

The function **logistic_regression_SG** carries out the steps shown in the following pseudocode:

```
* Create an empty list called log_likelihood_all
* Initialize coefficients to initial_coefficients
* Set random seed = 1
* Shuffle the data before starting the loop below
* Set i = 0, the index of current batch

* Run the following steps max_iter times, performing linear scans over the data:
  * Predict P(y_i = +1|x_i,w) using your predict_probability() function
    Make sure to slice the i-th row of feature_matrix with [i:i+batch_size,:]
  * Compute indicator value for (y_i = +1)
    Make sure to slice the i-th entry with [i:i+batch_size]
  * Compute the errors as (indicator - predictions)
  * For each coefficients[j]:
    - Compute the derivative for coefficients[j] and save it to derivative.
      Make sure to slice the i-th row of feature_matrix with [i:i+batch_size,j]
    - Compute the product of the step size, the derivative, and (1./batch_size).
    - Increment coefficients[j] by the product just computed.
  * Compute the average log likelihood over the current batch.
    Add this value to the list log_likelihood_all.
  * Increment i by batch_size, indicating the progress made so far on the data.
  * Check whether we made a complete pass over data by checking
    whether (i+batch_size) exceeds the data size. If so, shuffle the data. If not, do nothing.

* Return the final set of coefficients, along with the list log_likelihood_all.
```

At the end of the day, your **logistic_regression_SG** function should be analogous to this Python function:

```
def logistic_regression_SG(feature_matrix, sentiment, initial_coefficients, step_size, batch_size, max_iter)
:
    log_likelihood_all = []

    # make sure it's a numpy array
    coefficients = np.array(initial_coefficients)
    # set seed=1 to produce consistent results
    np.random.seed(seed=1)
    # Shuffle the data before starting
    permutation = np.random.permutation(len(feature_matrix))
    feature_matrix = feature_matrix[permutation,:]
```

```
sentiment = sentiment[permutation]

i = 0 # index of current batch
# Do a linear scan over data
for itr in xrange(max_iter):
    # Predict P(y_i = +1|x_i,w) using your predict_probability() function
    # Make sure to slice the i-th row of feature_matrix with [i:i+batch_size,:]
    ### YOUR CODE HERE
    predictions = ...

    # Compute indicator value for (y_i = +1)
    # Make sure to slice the i-th entry with [i:i+batch_size]
    ### YOUR CODE HERE
    indicator = ...

    # Compute the errors as indicator - predictions
    errors = indicator - predictions
    for j in xrange(len(coefficients)): # loop over each coefficient
        # Recall that feature_matrix[:,j] is the feature column associated with coefficients[j]
        # Compute the derivative for coefficients[j] and save it to derivative.
        # Make sure to slice the i-th row of feature_matrix with [i:i+batch_size,j]
        ### YOUR CODE HERE
        derivative = ...
              # Compute the product of the step size, the derivative, and
        # the **normalization constant** (1./batch_size)
        ### YOUR CODE HERE
        coefficients[j] += ...

    # Checking whether log likelihood is increasing
    # Print the log likelihood over the *current batch*
    lp = compute_avg_log_likelihood(feature_matrix[i:i+batch_size,:], sentiment[i:i+batch_size],
                                    coefficients)
    log_likelihood_all.append(lp)
    if itr <= 15 or (itr <= 1000 and itr % 100 == 0) or (itr <= 10000 and itr % 1000 == 0) \
        or itr % 10000 == 0 or itr == max_iter-1:
            data_size = len(feature_matrix)
            print 'Iteration %*d: Average log likelihood (of data points  [%0*d:%0*d]) = %.8f' % \
                (int(np.ceil(np.log10(max_iter))), itr, \
                 int(np.ceil(np.log10(data_size))), i, \
                 int(np.ceil(np.log10(data_size))), i+batch_size, lp)

        # if we made a complete pass over data, shuffle and restart
        i += batch_size
        if i+batch_size > len(feature_matrix):
            permutation = np.random.permutation(len(feature_matrix))
            feature_matrix = feature_matrix[permutation,:]
            sentiment = sentiment[permutation]
            i = 0

    # We return the list of log likelihoods for plotting purposes.
    return coefficients, log_likelihood_all
```

Compare convergence behavior of stochastic gradient ascent

15. For the remainder of the assignment, we will compare stochastic gradient ascent against batch gradient ascent. For this, we need a reference implementation of batch gradient ascent. But do we need to implement this from scratch?

Quiz Question: For what value of batch size B above is the stochastic gradient ascent function**logistic_regression_SG** act as a standard gradient ascent algorithm?

Running gradient ascent using the stochastic gradient ascent implementation

16. Instead of implementing batch gradient ascent separately, we save time by re-using the stochastic gradient ascent function we just wrote — **to perform gradient ascent**, it suffices to set **batch_size** to the number of data points in the training data. Yes, we did answer above the quiz question for you, but that is an important point to remember in the future :)

Small Caveat. The batch gradient ascent implementation here is slightly different than the one in the earlier assignments, as we now normalize the gradient update rule.

We now **run stochastic gradient ascent** over the **feature_matrix_train** for 10 iterations using:

- **initial_coefficients** = np.zeros(194)
- **step_size** = 5e-1
- **batch_size** = 1
- **max_iter** = 10

Quiz Question. When you set batch_size = 1, as each iteration passes, how does the average log likelihood in the batch change?

17. Now run **batch gradient ascent** over the **feature_matrix_train** for 200 iterations using:

- **initial_coefficients** = np.zeros(194)
- **step_size** = 5e-1
- **batch_size** = number of rows in **feature_matrix_train**
- **max_iter** = 200

Quiz Question. When you set batch_size = len(train_data), as each iteration passes, how does the average log likelihood in the batch change?

Make "passes" over the dataset

18. To make a fair comparison between stochastic gradient ascent and batch gradient ascent, we measure the average log likelihood as a function of the number of passes (defined as follows):

$$[\# \text{ of passes}] = \frac{[\# \text{ of data points touched so far}]}{[\text{size of dataset}]}$$

Quiz Question Suppose that we run stochastic gradient ascent with a batch size of 100. How many gradient updates are performed at the end of two passes over a dataset consisting of 50000 data points?

Log likelihood plots for stochastic gradient ascent

19. With the terminology in mind, let us run stochastic gradient ascent for 10 passes. We will use

- **step_size**=1e-1
- **batch_size**=100
- **initial_coefficients** set to all zeros.

20. Write yourself a function to generate a plot of the average log likelihood as a function of the number of passes. The function should accept the following parameters:

- **log_likelihood_all**, the list of average log likelihood over time
- **len_data**, number of data points in the training set
- **batch_size**, size of each mini-batch
- **smoothing_window**, a parameter for computing moving averages

The function should first compute moving averages of **log_likelihood_all**. To do this efficiently, convolve**log_likelihood_all** with the vector of length **smoothing_window** that is filled with the value**1/smoothing_window**.

The function then plot the moving averages over the number of passes over the data.
Use **len_data** and**batch_size** to convert iteration number of (fractional) number of passes.

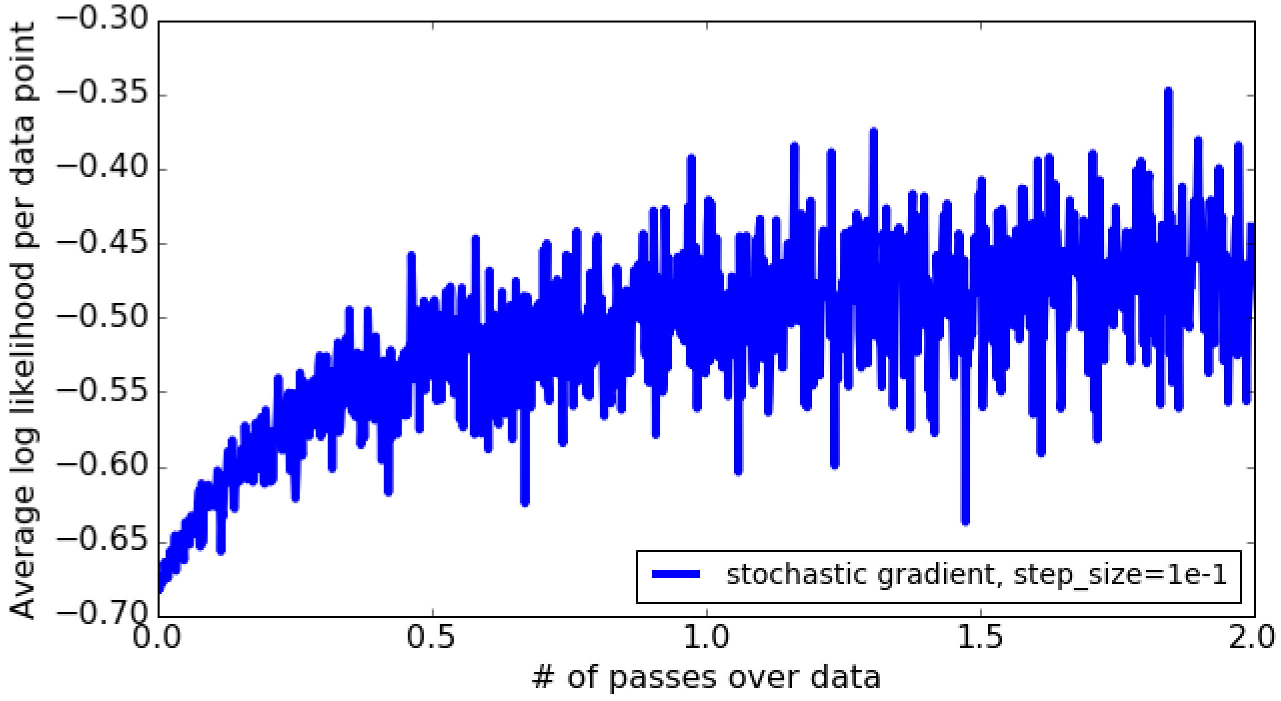
Using matplotlib, the plot-making function would be as follows:

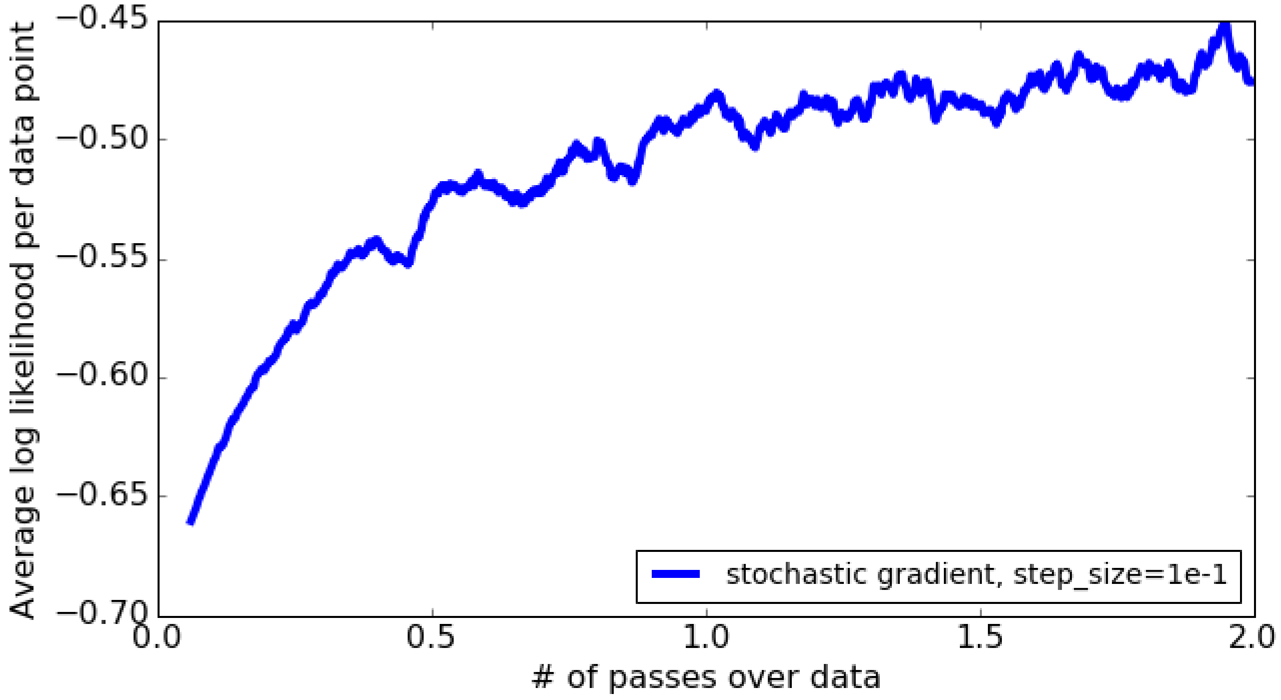
```
import matplotlib.pyplot as plt
%matplotlib inline

def make_plot(log_likelihood_all, len_data, batch_size, smoothing_window=1, label=''):
    plt.rcParams.update({'figure.figsize': (9,5)})
    log_likelihood_all_ma = np.convolve(np.array(log_likelihood_all), \
                                        np.ones((smoothing_window,))/smoothing_window, mode='valid')

    plt.plot(np.array(range(smoothing_window-1, len(log_likelihood_all)))*float(batch_size)/len_data,
             log_likelihood_all_ma, linewidth=4.0, label=label)
    plt.rcParams.update({'font.size': 16})
    plt.tight_layout()
    plt.xlabel('# of passes over data')
    plt.ylabel('Average log likelihood per data point')
plt.legend(loc='lower right', prop={'size':14})
```

The resulting plot should look like one of the two figures below:





Stochastic gradient ascent vs batch gradient ascent

21. To compare convergence rates for stochastic gradient ascent with batch gradient ascent, we call **make_plot**() multiple times.

We are comparing:

- **stochastic gradient ascent**: step_size = 0.1, batch_size=100
- **batch gradient ascent**: step_size = 0.5, batch_size=[# rows in feature_matrix_train]

Write code to run stochastic gradient ascent for 200 passes using:

- **step_size**=1e-1
- **batch_size**=100
- **initial_coefficients** set to all zeros.

For batch gradient ascent, use the results obtained from #17.

22. We compare the convergence of stochastic gradient ascent and batch gradient ascent by calling the **make_plot**function. Apply smoothing with **smoothing_window**=30.

Quiz Question: In the figure above, how many passes does batch gradient ascent need to achieve a similar log likelihood as stochastic gradient ascent?

Explore the effects of step sizes on stochastic gradient ascent

23. In previous sections, we chose step sizes for you. In practice, it helps to know how to choose good step sizes yourself.

To start, we explore a wide range of step sizes that are equally spaced in the log space. Run stochastic gradient ascent with **step_size** set to 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, and 1e2. Use

- **initial_coefficients**=np.zeros(194)
- **batch_size**=100
- **max_iter** initialized so as to run 10 passes over the data.

Plotting the log likelihood as a function of passes for each step size

24. Now, we will plot the change in log likelihood using the make_plot for each of the following values of step_size:

- **step_size** = 1e-4
- **step_size** = 1e-3
- **step_size** = 1e-2
- **step_size** = 1e-1
- **step_size** = 1e0
- **step_size** = 1e1
- **step_size** = 1e2

For consistency, use **smoothing_window**=30.

Quiz Question: Which of the following is the worst step size? Pick the step size that results in the lowest log likelihood in the end.

Quiz Question: Which of the following is the best step size? Pick the step size that results in the highest log likelihood in the end.