Week 1 Lab

The questions below are due on Sunday September 17, 2017; 11:00:00 PM.

A code file that will be useful for debugging on your computer can be found here (https://introml.mit.edu/__STATIC__/fall17/labs/lab01/code_for_lab1.py.zip).

Lab 1 is divided into four parts:

- Evaluating learning methods
- Implement evaluation strategies
- Implement perceptron and averaged perceptron
- Testing

Part 1 will require a checkoff with a staff member. The remaining parts are submitted and evaluated via this online tutor.

1) EVALUATING LEARNING METHODS

In Lab 0, we implemented a very simple learning algorithm that, when given a set of possible hyperplanes and a data set $\mathcal{D}_{train}$ (consisting of a data array and label vector), returned the hyperplane that minimized the number of errors on the training set. We have also read and thought about the perceptron algorithm (and will implement it later in this lab). And we will go on to study a number of other algorithms, all of which take in data as input and return a classification hypothesis as output.

In this part of the lab, we are going to explore ways of evaluating learning methods. You will have a conversation with a staff member (a "checkoff") and then proceed to implementing evaluation strategies.

1.1) Evaluating a classifier

A classifier is a function h that takes an example $x \in R^d$ as input and returns $+1$ or -1 as output. Consider the situation in which you have run some machine learning algorithm on some training data $\mathcal{D}_{train}$, and it has returned to you a specific h. Your job is to design (but not implement yet!) a procedure for evaluating h's effectiveness as a classifier.

Imagine that you have an infinite source of labeled data, generated by the some underlying process. So, you can call the generator, $\mathcal{G}$, with an argument n indicating the size of the data set, and it will return an X, y pair where X is a d by n array and y is a 1 by n array of labels $\{+1, -1\}$).

- Percy Eptron suggests reusing the training data to assess h:

```
def eval_classifier(h, D_train):
 X, y = D_train
 return score(h, X, y)
```

  Explain why Percy's strategy might not be so good.

- Write pseudocode for a procedure that takes h, $\mathcal{D}_{train}$, and $\mathcal{G}$ and returns a score (the syntax for pseudocode is not important, but do write something down).
- Say what your output score is going to be, what it means, what the best and worst values are.
- Explain why your method might be more desirable than Percy's.

1.2) Evaluating a learning algorithm

A learning algorithm is a function L that takes a data set $\mathcal{D}_{train}$ as input and returns a classifier h. Now, consider a situation in which someone is trying to sell you a new learning algorithm, and you want to know how good it is. There is an interesting result that says that, in the absence of some assumptions about your data, *there is no learning algorithm that is better than every other algorithm for all data sources* (https://en.wikipedia.org/wiki/No_free_lunch_theorem). So, you'll need to assess the learning algorithm's performance in the context of a particular data source.

Assume that you have a generator of labeled data, $\mathcal{G}$, and further assume that data generated by $\mathcal{G}$ is "similar enough" to data associated with your application. And by "similar enough", we mean that the learning algorithm's performance on $\mathcal{G}$-generated data will be a good predictor of the learning algorithm's performance on data from your application.

Furthermore, assume that when you run this learning algorithm in your application domain next week, you will only have 100 labeled training examples.

How could you use the generator to evaluate the quality of the learning algorithm, L, when used for your application.

- Linnea Separatorix suggests the following procedure:

```
def eval_learning_alg(L, G):
 # draw a set of n training examples (points and labels)
 X, y = G(n)
 # run L
 h = L(X, y)
 # evaluate using your classifier scoring procedure, on some new labeled data
 return eval_classifier(h, G)
```

  Explain why Linnea's strategy might not be so good.

- Write pseudocode for a procedure that takes L and $\mathcal{G}$ and returns the score.
- Say what your output score is, what it means, what the best and worst values are.

- Explain why your method might be more desirable than Linnea's.

1.3) Evaluating a learning algorithm with a finite amount of data

In reality, it's almost never possible to have a generator of all the data you want; in fact, in some domains data is very expensive to collect, and so you are given a fixed set of samples.

- How might you modify your solution to the previous part to handle a fixed total number of examples?
- You don't need to write out new pseudocode, but do think about what trade-offs might be involved in formulating an exact version of your procedure.

Checkoff 1:
Have a check-off conversation with a staff member, to explain your answers.

Ask for Help Ask for Checkoff

After you receive the checkoff, refresh the page to access the remaining parts.

There are some longer coding problems in this next part. We strongly suggest that you debug your code in a Python environment on your computer, and not inside the tutor. If you do use the tutor, save frequently. You are at serious risk of losing your work if it crashes.

2) IMPLEMENT EVALUATION STRATEGIES

Note: for all of the problems in today's lab, you are allowed to use `for` loops.

2.1) Evaluating a classifier

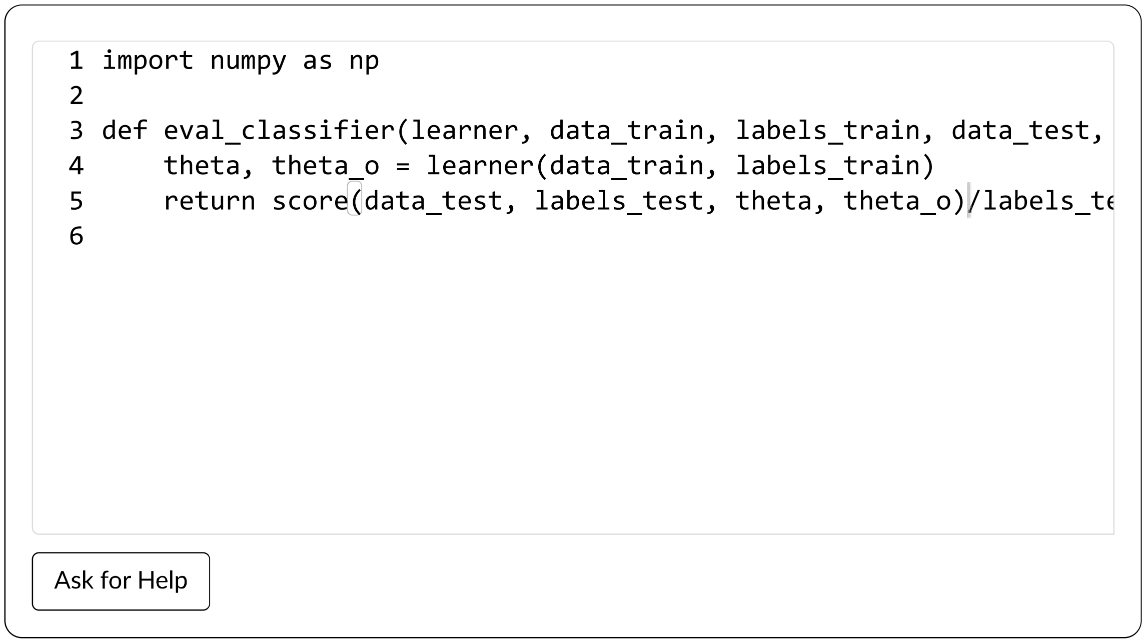
To evaluate a classifier, we are interested in how well it performs on data that it wasn't trained on. Construct a testing procedure that uses a training data set, calls a learning algorithm to get a linear separator (a tuple of θ, θ_0), and then reports the percentage correct on a new testing set as a float between 0. and 1..

The learning algorithm is passed as a function that takes a data array and a labels vector. It should be able to interchangeably call `perceptron` or `averaged_perceptron` or future algorithms we come up with that have the same spec.

Assume that you have available the function `score` from Lab 0, which takes inputs:

- `data`: a `d` by `n` array of floats (representing `n` data points in `d` dimensions)
- `labels`: a `1` by `n` array of elements in `(+1, -1)`, representing target labels
- `th`: a `d` by `1` array of floats that together with
- `th0`: a single scalar or 1 by 1 array, represents a hyperplane

and returns the number of points for which the label is equal to the output of the `positive` function on the point.



```
import numpy as np

def eval_classifier(learner, data_train, labels_train, data_test,
    theta, theta_o = learner(data_train, labels_train)
    return score(data_test, labels_test, theta, theta_o)/labels_te

```

2.2) Evaluating a learning algorithm using a data source

Construct a testing procedure that takes a learning algorithm and a data source as input and runs the learning algorithm multiple times, each time evaluating the resulting classifier as above. It should report the overall average classification accuracy.

You can use our implementation of `eval_classifier` as above.

Write the function `eval_learning_alg` that takes:

- `learner` - a function, such as perceptron or averaged_perceptron
- `data_gen` - a data generator, call it with a desired data set size.
- `n_train` - the size of the learning sets
- `n_test` - the size of the test sets
- `it` - the number of iterations to average over

and returns the average classification accuracy as a float between 0. and 1..

```
import numpy as np

def eval_learning_alg(learner, data_gen, n_train, n_test, it):
    performance = 0
    for i in range(it):
        data_train, labels_train = data_gen(n_train)
        data_test, labels_test = data_gen(n_test)
        performance += eval_classifier\
                       (learner, data_train, labels_train, data_t
    return performance/it

```

Ask for Help

2.3) Evaluating a learning algorithm with a fixed dataset

Cross-validation is a strategy for evaluating a learning algorithm, using a single training set of size n. Cross-validation takes in a learning algorithm L, a fixed data set $\mathcal{D}$, and a parameter k. It will run the learning algorithm k different times, then evaluate the accuracy of the resulting classifier, and ultimately return the average of the accuracies over each of the k "runs" of L. It is structured like this:

```
divide D into k parts, as equally as possible;  call them D_i for i == 0 .. k-1
# be sure the data is shuffled in case someone put all the positive examples first in the data!
for j from 0 to k-1:
    D_minus_j = union of all the datasets D_i, except for D_j
    h_j = L(D_minus_j)
    score_j = accuracy of h_j measured on D_j
return average(score0, ..., score(k-1))
```

So, each time, it trains on $k-1$ of the pieces of the data set and tests the resulting hypothesis on the piece that was not used for training.

When $k = n$, it is called *leave-one-out cross validation*.

Implement cross validation assuming that the input data is shuffled already so that the positives and negatives are distributed randomly. If the size of the data does not evenly divide by k, truncate the size of the data sets and place any extra data points in the last (highest numbered) data set.

```
    d, n = data.shape
    r = n%k
    k_data = np.split(data[:,0: (n-r)], k, axis = 1)
    k_data[-1] = np.concatenate((k_data[-1], data[:,n-r:n]), a
    k_labels = np.split(labels[0:(n-r),:], k, axis = 0)
    k_labels[-1] = np.concatenate(k_labels[-1], labels[n-r:n,:

    performance = 0
    for i in range(k):
        data_train = np.concatenate(tuple(k_data[:i]+k_data[i+1
        labels_train = np.concatenate(tuple(k_labels[:i]+k_labe
        performance += eval_classifier\
                       (learner, data_train, labels_train, k_da
    return performance/k
```

Ask for Help

3) IMPLEMENT PERCEPTRON

Implement the perceptron algorithm, where

- `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 should run 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. We won't be testing this in the Tutor, but it will help you in debugging on your own machine.

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

We have given you some data sets in the code file for you to test your implementation.

Your function should initialize all parameters to 0, then run through the data, in the order it is given, performing an update to the parameters whenever the current parameters would make a mistake on that data point. Perform T iterations through the data. After every parameter update, if `hook` is defined, call it on the current `(th, th0)` (as a single parameter in a python tuple).

When debugging on your own, you can use the procedure `test_linear_classifier` for testing. By default, it pauses after every parameter update to show the separator. For data sets not in 2D, or just to get the answer, set `draw = False`. See top of this page for the code distribution.

```
    T = params.get('T', 100)
    d, n = data.shape
    labels = labels.T
    theta = (np.array([[0.0]*d])).T
    theta_o = 0
    for i in range(T):
        for i in range(n):
            if (np.dot(theta.T, data[:,[i]]) + theta_o)*labels[
                theta += (data[:,[i]]*labels[i])
                theta_o += labels[i][0]
                if hook:
                    hook((theta, theta_o))
    return theta, theta_o

```

Ask for Help

4) IMPLEMENT AVERAGED PERCEPTRON

Implement the averaged perceptron corresponding to the same spec above and test it on the same data sets.

You can use the more straightforward, albeit less efficient, form of the averaged perceptron:

```
procedure averaged_perceptron({(x^(i), y^(i)), i=1,...n}, T)
    th = 0 (d by 1); th0 = 0 (1 by 1)
    ths = 0 (d by 1); th0s = 0 (1 by 1)
    for t = 1,...,T do:
        for i = 1,...,n do:
            if y^(i)(th . x^(i) + th0) <= 0 then
                th = th + y^(i)x^(i)
                th0 = th0 + y^(i)
            ths = ths + th
            th0s = th0s + th0
    return ths/(nT), th0s/(nT)
```

- You should try to understand why the version in the notes is equivalent to this,

```
    T = params.get('T', 100)
    d, n = data.shape
    labels = labels.T
    theta = (np.array([[0.0]*d])).T ; thetas = (np.array([[0.0
    theta_o = 0; theta_os = 0
    for i in range(T):
        for i in range(n):
            if (np.dot(theta.T, data[:,[i]]) + theta_o)*labels[
                theta += (data[:,[i]]*labels[i])
                theta_o += labels[i][0]
                if hook:
                    hook((theta, theta_o))
            thetas += theta
```

Ask for Help

5) TESTING

In this section, we compare the effectiveness of perceptron and averaged perceptron on some data that are not necessarily linearly separable.

Use your `eval_learning_alg` and the `gen_flipped_lin_separable` generator in the code file to evaluate the accuracy of `perceptron` vs. `averaged_perceptron`. `gen_flipped_lin_separable` can be called with an integer to return a data set and labels. Note that this generates linearly separable data and then "flips" the labels with some specified probability (the argument `pflip`); so most of the results will not be linearly separable. You can also specifiy `pflip` in the call to the generator. You should use the default values of `th` and `th_0` to retain consistency with the Tutor.

Run enough trials so that you can confidently predict the accuracy of these algorithms on new data from that same generator; assume training/test sets on the order of 20 points. The Tutor will check that your answer is within 0.025 of the answer we got using the same generator.

Accuracy for perceptron (with flip probability 0.1):

0.75125

Ask for Help

Accuracy for averaged perceptron (with flip probability 0.1):

0.80275

Ask for Help

Accuracy for perceptron (with flip probability 0.25):

```
0.5885
```

Ask for Help

Accuracy for averaged perceptron (with flip probability 0.25):

```
0.6525
```

Ask for Help

Modify your `eval_learning_alg` so that it tests hypothesis on the training data instead of generating a new test data set. Run enough trials that you can confidently predict this "training accuracy" for the two learning algorithms. Note the differences from your results above.

Accuracy for perceptron (with flip probability 0.1) on training data:

```
0.82025
```

Ask for Help

Accuracy for averaged perceptron (with flip probability 0.1) on training data:

```
0.861
```

Ask for Help

Accuracy for perceptron (with flip probability 0.25) on training data:

```
0.67
```

Ask for Help

Accuracy for averaged perceptron (with flip probability 0.25) on training data:

```
0.72075
```

Ask for Help