**Sample bagging results**

**ripple.csv**

**#learner=KNNLearner, bags = 20, k=3, boost=False**

In sample results  
RMSE: 0.128785487103  
corr: 0.98478181469

Out of sample results  
RMSE: 0.195888429176  
corr: 0.962814074991

(0.8 seconds to execute on weak ass virtual box VM)

**#learner = KNNLearner, bags = 200, k=3, boost=False**

In sample results  
RMSE: 0.123079825223  
corr: 0.986340038967

Out of sample results  
RMSE: 0.192913144806  
corr: 0.964096549553

20 bags

In sample results  
RMSE:  0.126117439866  
corr:  0.985267741668  
  
Out of sample results  
RMSE:  0.18866278464  
corr:  0.965391918262

Avoid looping with Bag Learner

See: <https://piazza.com/class/idadrtx18nie1?cid=1148>

https://d1b10bmlvqabco.cloudfront.net/photos/hq9zldrlaq4vt/1406417492_35.png

[**Brandon Guttersohn**](https://piazza.com/class/idadrtx18nie1?cid=1148) [7 days ago](https://piazza.com/class/idadrtx18nie1?cid=1148)

Haven't tested any of this, but...

1.

predictions = np.array # array class

...will assign the numpy array class to the variable predictions -- you want aninstance of that class;

predictions = np.array([]) # array with no items

Since you know what the size of predictions will need to be before iterating, you should just create an array of that size right off the bat;

predictions = np.zeros( ( self.bags, len( x ) ) ) # array with 'self.bags' rows and enough columns for each point (row) in x

2. To avoid appending to 'predictions' while iterating (since you already have a large enough array allocated), assign in-place;

predictions[i,:] = self.learners[i].query(x)

The above means "assign all columns of the ith row in 'predictions' to whatever query() returns". Query should be returning a 1D array of length equal to the rows of x....

3. At this point, predictions is an MxN array, where each row 1..M represents the output of a single learner/bag, and each column 1..N represents the results on any given test point. For example:

predictions[2,:] # 1D array, the results of learner 2 for all given test points in X

predictions[:,2] # 1D array, the results of all learners for the index-2 test point in X

predictions[2,2] # single value, the result of learner 2 for the index-2 test point in X

What you ultimately want is the average result, over all learners, for each test point. So the value you return will be an array of length N. Another way to say this is that you want numpy to produce a single row whose values are the means of each column in predictions. You're right to use...

predictions.mean()

...but you'll need to take a look at the "axis" parameter in the numpy documentation.

disclaimer; all of this could be wrong, but mine worked and I did something similar

**Bagging**

What i ended up doing was getting my random values as a big index and using that index to select both the x's and y's so that they matched up when i sent them in addEvidence.

**numpy.random.choice(**a**,**size=None**,**replace=True**,**p=None**)**

<https://piazza.com/class/idadrtx18nie1?cid=1061>

The term "bag" is a bit odd. All it means is instead of training one model, let's train X of them and somehow combine (e.g. average) their results. Because if 1 is good more must be better, right?! So if someone passes in bags=20 to your BagLearner, you should instantiate 20 instances of the "learner" argument (e.g. KNNLearner) and store them in a list. In this way it's a "bag" of 20 KNN learners. Having many instances of the same learning algorithm only makes sense if they differ in some way. The "Bagging" approach is to have all of them randomly sample from the available training data so that they all have a somewhat different view of the world and can strengthen/reinforce each other's predictions. If there are Y training instances available and each underlying learner samples Y values with replacement from the train instances, it's expected they'll each see about 66% of the training instances. Of course they will most likely all see a different 66%.

After the base learners have been trained, and a new point is queried on, you simply invoke all of the base learners and average the result of their predictions. In this way you're kind of crowd-sourcing the prediction.

As for the term, 'bag' comes from 'bootstrap aggregate,' which is to say, take a bunch of bootstrap samples, train separate models on them, then aggregate the models.

**~ An instructor (Tucker Balch) endorsed this answer  ~**

 To make it simple, hardcode bags = 10.  
Create 10 instances of the knn learner, Use the sample code they provided:

learners = []

kwargs = {"k":10}

for i in range(0,bags):

learners.append(learner(\*\*kwargs))

- Initialize each one with a random subset of data.   
- Then query all 10 bags, and average their results.  
- Finally parametrize everything else.  
- It'll be about 15 lines of code (excluding boilerplate stuff)

we give each learner the whole entire test set, and then average the results

You randomly sample training data, train each model on the samples, then use all of the models to generate a whole bunch of predictions for the entire test set, then average the predictions.

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1061) [9 days ago](https://piazza.com/class/idadrtx18nie1?cid=1061)

>> randomly select the (size of the training set) number of samples, and do that bag number of times (one for each learner).

Should all be the same size. Say you have a 2d array of data. You can get the bags like so:

nbags = 20

M = all\_my\_data.shape[0]

bags = [all\_my\_data[random\_integers(0, M - 1, size=M), :] for \_ in range(nbags)]

Gives you a list of 20 arrays of the same size with the rows sampled with replacement from the original array.

or

You can just draw the indices separately:

idxs = [random\_integers(0, M - 1, size=M) for \_ in range(nbags)]

Then use them to index whatever you want. x[idxs[0]], y[idxs[0]], etc.

There is no need for you to have such a conditional in your BagLearner class.  You just pass the necessary arguments for the particular learner when you construct the BagLearner.

To instantiate a LinRegLearner:

learner = BagLearner(learner = LinRegLearner, kwargs = {})

Generating Data - Breaking KNN

How about when you add in noise...?

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1134) [6 days ago](https://piazza.com/class/idadrtx18nie1?cid=1134)

There are a number of valid ways to throw KNN off the scent.  Some involve adding noise, others involve clever ways of distributing the data.  Another involves carefully selecting parameters of a linear model that will confound KNN.  Probably even others I haven't thought of yet.  All of those are OK.

**Eucladian distnace**

**When in doubt, print it out!**  add a line of code in the right place:  (2nd line gives you the type of element within the array)...

print "data type", type(data)  
print "element type", type(data[0,0])

**Plotting**

<http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html>

**Boosting**

Use of pandas would not be penalized so long as you didn't use any learning libraries (which I don't know if exist).

[**Darren Bedwell**](https://piazza.com/class/idadrtx18nie1?cid=1143) [4 days ago](https://piazza.com/class/idadrtx18nie1?cid=1143)

This was the only Pandas routine that I used, and it was only for boosting:

<http://pandas.pydata.org/pandas-docs/version/0.17.0/generated/pandas.DataFrame.sample.html>

**Implementing KNN**

[**Geoffrey Shmigelsky**](https://piazza.com/class/idadrtx18nie1?cid=953) [11 days ago](https://piazza.com/class/idadrtx18nie1?cid=953)

How can kNN be only O(n)? Don't we have to :

O(n): (agreed)

1. Iterate through the entire kNN set X1, X2 in learner.query

2. Solve for Euclidian distance for each.

NLog(n): (request clarification)

Then to get the top k scoring NN's, do we not have to sort based on Euclidian distance ? :)

As I think about it - you could just iterate K times over the distance result - that technically would be O(n).

-G

Finding the k smallest values does not require sorting.  You can use the selection algorithm which is O(n)

https://en.wikipedia.org/wiki/Selection\_algorithm

https://d1b10bmlvqabco.cloudfront.net/photos/i4hdvab632i522/1420465436_35.png

[**MarkBenjamin1@gatech.edu**](https://piazza.com/class/idadrtx18nie1?cid=953) [8 days ago](https://piazza.com/class/idadrtx18nie1?cid=953) I think that's implemented in python as<https://docs.python.org/2.7/library/heapq.html#heapq.nsmallest>



[**Malcolm Haynes**](https://piazza.com/class/idadrtx18nie1?cid=953) [8 days ago](https://piazza.com/class/idadrtx18nie1?cid=953)

You only need to go through the data once.

Start with an empty array of 'k' values and store the first 'k' distances in it.

Now, go through your remaining distances.

If you find a distance smaller than stored in your array, replace the biggest value in the array with the smaller distance.