Intro to Machine Learning

CS 7646 "Part 2"
Instructor: David Byrd

June 12, 2016

1 Taxonomy

1.1 Learner vs Model

A *model* is a construct that turns *observations* into *predictions*. We input values that we can know or measure and receive an estimate of values we do not know or can not directly measure. Models do not exist solely within Machine Learning. The Black-Scholes model is a popular "formula" for estimating what the value of a stock's options *should* be based on other factors, and it is not related to ML at all.

A *learner* (or *learning algorithm*) is a computer program that improves its performance at some task by exposure to data. We train a learner with data to obtain a model. We then use the model to make predictions. The training data given to the learner is usually denoted as a tuple (X, y), where for a single data point, X is a 1-D vector of factors, and y is the single value the model should return if given that X vector.

A *factor* is simply some value derived from the raw observation. If the observations are stock prices, factors might be Bollinger Bands, simple moving averages, or daily returns. In this class, I use the terms *factor* and *feature* interchangeably when discussing values computed from the raw observation data.

1.2 Supervised vs Unsupervised

In *supervised* learning, the learner is given labelled training data points. A *label* in this case is simply a correct value of y for the X factors. If our input tuple for one data point is $(X, y) = ((0.05, 40.1, 51000000), 41.2)$, then our X factors might be: $dailyreturn = 0.05, sma = 40.1, volume = 51000000$. The value we would want to predict for these X factors is $y = cumret(10) = 41.2$, imagining that we wish to predict cumulative return ten days in the future. That value (41.2) is the *label*, the correct answer.

In *unsupervised* learning, the learner is given unlabelled data points. This amounts to giving it the X observations only, with no y values. This results in a learner that can *not* make predictions for a y value, but *can* still analyze the observations for similarity to one another, to “cluster” the observations into groups, or to detect outlying observations.

1.3 Classification vs Regression

The *classification* problem in Machine Learning attempts to correctly map a set of discrete or continuous observations to a *discrete* and *limited* set of output predictions. For example, trying to learn what type of animal is depicted in a computer image (dog, cat, horse, or cow) is a classification problem with four output classes.

The *regression* problem in Machine Learning attempts to correctly map a set of discrete or continuous observations to a *continuous* output prediction. For example, trying to estimate the price of a stock ten days from now is a regression problem (with arbitrary real-valued output).

1.4 Parametric vs Instance

A *parametric* learner is one that, during training, optimizes a set of parameters or coefficients. Predictions will be made using only the input test data point and these parameters. The original training data is “thrown away” after use. For example, a linear regression learner with two input factors will learn some values for m_1, m_2, b in the equation $y = m_1x_1 + m_2x_2 + b$. When queried, the learner plugs the test x_1 and x_2 into that formula with the learned coefficients and produces an answer directly.

An *instance* learner is one that directly uses the training data at query time to make a prediction. For example, K-Nearest Neighbors stores the training data in its entirety. When a query arrives, it finds the K nearest training data points (normally using Euclidean distance between X vectors) and outputs their mean y as its prediction.

Parametric learners typically take longer to train. It is also difficult to incorporate new training points without completely retraining. On the other hand, the model (disk/memory) is very compact and fast to query. Instance learners are generally quick to train and accept additional data easily, but are large (disk/memory) and can be slower to query.

A parametric model also usually requires us to have at least a general idea of the underlying principle and the ability to mathematically express it prior to training. That is, the learner begins with a parameterized mathematical expression (or function) and optimizes the parameters during training. If we select the wrong initial mathematical model, it will not perform well — for example training a linear regression learner on highly-non-linear data. This requirement for a correct general understanding of the data up-front results in a *biased learner*. The model is biased by our preconceived idea of what the relationship will look like.

An instance learner does not typically require an up-front model. It starts from a blank slate with no assumptions about the data at all. Thus we say it is an *unbiased learner*.

If you *have* a correct understanding of the general relationship in your data (e.g. firing a cannonball at some elevation vs the distance it will travel), you should use that knowledge to inform a biased learner and produce better results. If you do not have any idea what the relationship might be, you should choose an unbiased learner to avoid providing an incorrect set of assumptions.

Note that this usage of the word *bias* is not related to the *bias-variance trade-off* or the statistical idea of a *biased estimator*!

1.5 Discriminative vs Generative

A *discriminative* learner is one that models the conditional probability distribution $p(y|X)$. Because it only learns y conditioned on X, it is useful only for making predictions of y given some X. It does not model how the data was generated, it does not find a full underlying distribution of the data, and thus one can not sample from it to produce new data points. Discriminative learners are where most of the current ML "action" is, because these learners are producing better predictions for very complex problems like recognizing the activities being performed in a raw movie clip.

A *generative* learner is one that models the joint probability distribution $p(X, y)$. Thus, it does model a full underlying distribution of the data and one can sample from it to produce new data points. Generative learners are still useful when one *needs* the joint distribution and generally produce better results if the available training data set is quite small. (If I have only five total training points, I can still find a best fit line (generative model) and make sane predictions. I am not likely to get far with a KNN learner (discriminative model)).

A model that tries to classify the language of a text fragment by learning words or phrases in many languages, then comparing a new fragment to the languages it knows would be discriminative. (Note that it can't create a "new" language by sampling.) A model that tries to understand how languages are created (one factor might be word order: subject-verb-object vs subject-object-verb vs verb-subject-object) and then uses that underlying knowledge to classify a language would be generative. (Note that it *could* create a new language just by sampling from all those factors.)

1.6 Batch vs Online

A *batch* learner is one that requires all training data to be available to the learner simultaneously. In other words, it needs to consider *all* of the data to make proper, optimal (or at least greedy-optimal) training decisions. Batch learners usually produce the best results, but must be fully retrained from scratch if new data is added to the training set.

An *online* learner is one that can integrate a new piece of training data (i.e. (X, y) tuple) without significant retraining. That is, it can incorporate new observations into its existing model without starting over. Online learners are the obvious choice for a system that must accumulate data over a long period of time while refining their performance automatically.

Most learning algorithms *can* be structured as either a batch or online learner depending on your approach and the trade-offs you choose to make. For instance, a linear regression learner trained with classical gradient descent is effectively a batch learner, while one trained with stochastic gradient descent can be treated as online. KNN is online by its nature (it only stores data, querying the data for each new test point).

2 Training/Assessing a Time-Series Learner

2.1 Constructing Learning Tuples

Unstructured learning algorithms (the most common class) do not take explicitly ordered observations. Each learning tuple that arrives is treated separately from the others, with no memory of which came first. (*Note:* an online learner may find a different local optimum if the data are presented in a different order, but this does not mean it is *learning* anything from the order of the data.)

Thus for a time series learner that must predict the future, it is up to us to construct the tuples to incorporate the necessary timeline. Typically in this class, we are using an array of *(days, features)* to predict a vector of *(cumulative returns)*. In particular, we are using a historical set of features from days $T-N$ to T to predict a value for day $T+M$, where T is today, N is the look-back period length, and M is the look-ahead period length.

In our raw data, `X[20]` and `y[20]` refer to the same day. If we give the data to the learner like this, it will use its knowledge of day 20 to predict the price/return on day 20. This is not very helpful, so we assemble time-shifted tuples, for example `(X[10:20],y[25])` and `(X[11:21],y[26])`. Now the learner will use its knowledge from the last ten days to predict the return five days in the future.

Bear in mind that our number of tuples will be less than the total amount of data. We cannot predict y values for days 0 to $N-1$, because there will not be enough X values in the past (off the top end of the data). We cannot use the X values within M days of the end, because we will not have a y value to predict (off the bottom end of the data).

2.2 Backtesting Market Strategies

Assume we have trained a time-series learner and we want some way to evaluate how well it performs. What we *want* to is to predict the future, so let's test how well our model does that. Problem: we do not have data from the future (Star Trek tachyon beams aside).

We could open a brokerage account, fund it with a decent amount of money, and start placing the orders recommended by our learner. This is a pretty good way to lose that money. We could *paper trade* the strategy, carefully "doing the math" each day to see how much money we *would* have made if we had really invested it. This is tedious and slow.

Instead, we will substitute historical data by *pretending* some of the data is still in "the future". This is called *backtesting* a strategy. It is used for any sort of investment or trading strategy, not just those produced by a learning algorithm.

To perform backtesting, we take a set of historical stock data from consecutive trading days. Let's imagine we have 1,000 days of data. We start by pretending it is day $T = N$ (the first day on which we have enough "past" data to make predictions). We look at days 0 to T, calculate our prediction for day $T + M$, consult our strategy to buy, sell, or do nothing on day T, and record the orders we would have placed. We then advance $T \leftarrow T + 1$ and repeat, until we reach the end of the data.

Now we have 1,000 days worth of orders that we *would* have placed (to open and close market positions) *if* we had been using this strategy during that time period. We can then use a portfolio calculator to see our gain/loss, volatility, and so forth, and compare them to other strategies or learners in the same time period.

2.3 Assessing Regression Learners

2.3.1 Measures of accuracy

We will assess our learners using a *loss function* or an *error function.* This calculates how "wrong" the model prediction is. Assuming we construct a valid loss function, minimizing its result will produce the least bad (i.e. best) learner. A typical choice of loss function for regression is *root mean squared error* or RMSE. Also common is the *sum of squared errors* or SSE.

$$RMSE = \sqrt{\frac{\sum(y_{pred} - y_{actl})^2}{N}}$$

where N is the number of data points tested. The lower RMSE becomes, the better our model is performing.

Another typical evaluation of a regression learner is Pearson's r, informally called the correlation coefficient.

$$r = \frac{\text{cov}(y_{pred}, y_{actl})}{\text{std}(y_{pred})\text{std}(y_{actl})}$$

Pearson's correlation coefficient reveals how much of the total variation in y_{pred} and y_{actl} can be explained by their relationship to one another. The resulting ratio will vary from -1 indicating perfect anti-correlation, through 0 indicating no correlation, to $+1$ indicating perfect correlation. *Important note:* the slope of the "best fit" line through a scatterplot of y_{pred} vs y_{actl} is often mistaken for *r*. This is incorrect. *r* is a measure of the amount of "scatter" in the scatterplot.

Consider using `numpy.corrcoef()` when you need Pearson's *r*.

2.3.2 In-sample vs out-of-sample error

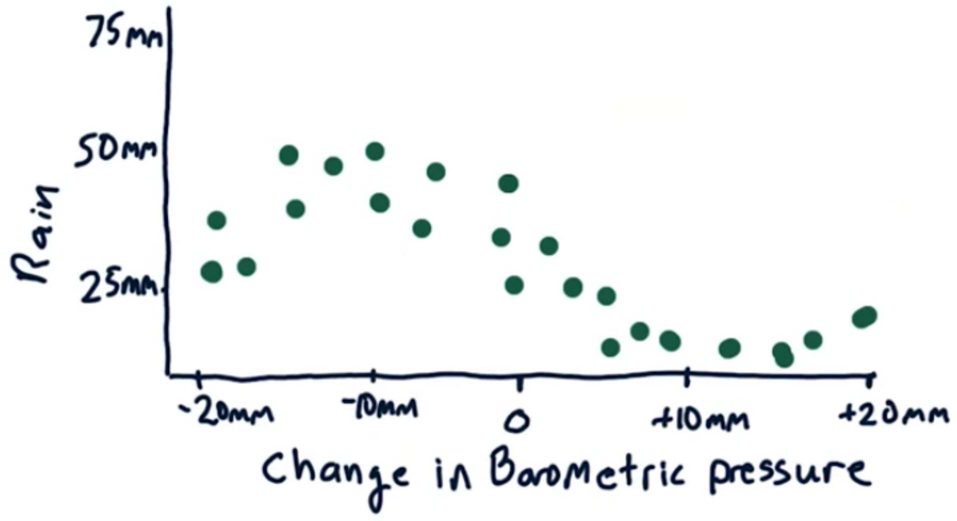
It is important to distinguish between *in-sample* error and *out-of-sample* error. In-sample error is measured by querying the trained learner on the *training* data. If in-sample error is high, our learning algorithm may not be powerful enough to learn the pattern (if any) of this data. If in-sample error is low, our algorithm is powerful (or "expressive") enough, but this alone is not sufficient!

Out-of-sample error is measured by querying the trained learner on data it has never seen (i.e. the *test* or *validation* set of data). This is the "true" performance of the model, because of course what we want is accurate predictions of data where we *don't* already know the answers.

2.3.3 Overfitting

One easy to explain, but frequently overlooked, problem in machine learning is *overfitting* the model to the training data. When this occurs, the model will not generalize well for predictions on unseen data. It has in some sense learned the *exact* pattern of the training data (including its "noise") rather than learning the general *underlying* relationship in the data.

In class, we discussed a plot of the change in barometric pressure to the next day's rainfall, similar to this one:



Imagine that we construct a model by simply connecting these dots in order from left to right. If the change in barometric pressure (CBP) is -17mm, our model will predict 27mm of rain. If the CBP rises to -15mm, it predicts 50mm of rain (twice as much!). If the CBP rises again to -14mm, it predicts only 40mm of rain. But then for -12 CBP, it predicts almost 50mm again. As CBP slowly increases, rainfall sharply increases, then decreases again, then increases again, and so on. Does this seem like it is capturing the "real" relationship between these two natural forces? Of course not. Our model has *overfit* this data and is predicting *relationship* + *noise* rather than just *relationship*. Predicting the noise *separately* might be very useful, but that is not what is happening.

Formal definition: hyperparameter choices that result in *decreasing training error* concurrent with *increasing test error* indicate overfitting. (Your model's performance on the training data is continuing to improve *at the expense* of its ability to generalize.) The optimal *fit* is the point at which *test error is lowest.* When neither of these are true, the model is *underfitting* the data (it lacks the capacity to fully capture the relationship among the variables).

2.3.4 Cross-Validation

The most basic method for separating a data set into training and test data is to randomly select some percentage of the data as the training set and leave the rest for testing. This is easy and valid, but we are not getting as much from our data as we could. We only get one training set and one testing set. Therefore, we get only a single aggregate assessment of our model's accuracy.

A common way to "get more" from the data is to use *cross-validation.* Instead of producing one model from our data, we will produce N models by dividing the data into N "chunks". This is called *N-fold* cross-validation. Model M will be tested using chunk M and trained using all the other chunks. Now having N models, we can average their predictions to produce a less noisy model, or select the model with the lowest test error, or some other idea that may occur to you. Also common for small data sets is *leave-one-out cross-validation*, in which each model is tested on a single data point and trained on all other points.

These methods pose a problem for our time-series data, however. We have established that backtesting is only valid if the model can not see the future, and all of these cross-validation methods create some models that are given data from "later" than the time they are predicting!

2.3.5 Roll-Forward Cross-Validation

Roll-forward cross-validation follows the exact same process as backtesting itself. We start at the beginning of the data time series and expose our learner to the data set one chronological day at a time. Each day, it must make its future predictions *before* being given the next data point. In this way, at some day T, the learner will be allowed to train on days 0 to T in any way desired, but can not access data from days $T + 1$ forward.

2.4 Shortcomings of Regression Learning

Regression learning tends to produce noisy, uncertain models. We get a numerical prediction, but it can be difficult to produce a reliable confidence interval around that prediction. For stock market prediction in particular, regression only solves part of our real problem. We want to make actual trades in the stock market. A regression learner can only predict the price (or price change). *We* still have to create a trading policy outside of the learner that decides whether to trade on a prediction, how many shares to buy or sell, and how long to hold that position.

Later in the course, we will examine *reinforcement learning* as an improvement to those shortcomings. A *reinforcement learner* learns a mapping of world states to optimal actions called a *policy* and can thus tell us what we should *do* at some moment in time, rather than simply make a numerical prediction.

3 Supervised Regression Learners

When implementing supervised regression learners in Python, we recommend holding them all to the same Application Programming Interface (API). That is, create each Learner as a reusable class that contains the *same* set of methods for initializing, training, and testing. If you do this, you can "swap out" one learner for another in some program you are writing, without having to alter anything except the name of the learner you want to use.

For example, if your linear regression learner and KNN learner both use the suggested API, there is little difference in using them.

```
learner = LinRegLearner()
learner.addEvidence(Xtrain,Ytrain)
y = learner.query(Xtest)
```

To switch to KNN, change the first line to: `learner = KNNLearner(k=3)`. The rest stays the same.

3.1 K-Nearest Neighbors

3.1.1 Hyperparameters

KNN requires a single hyperparameter K, the number of neighbors to consult when making a prediction.

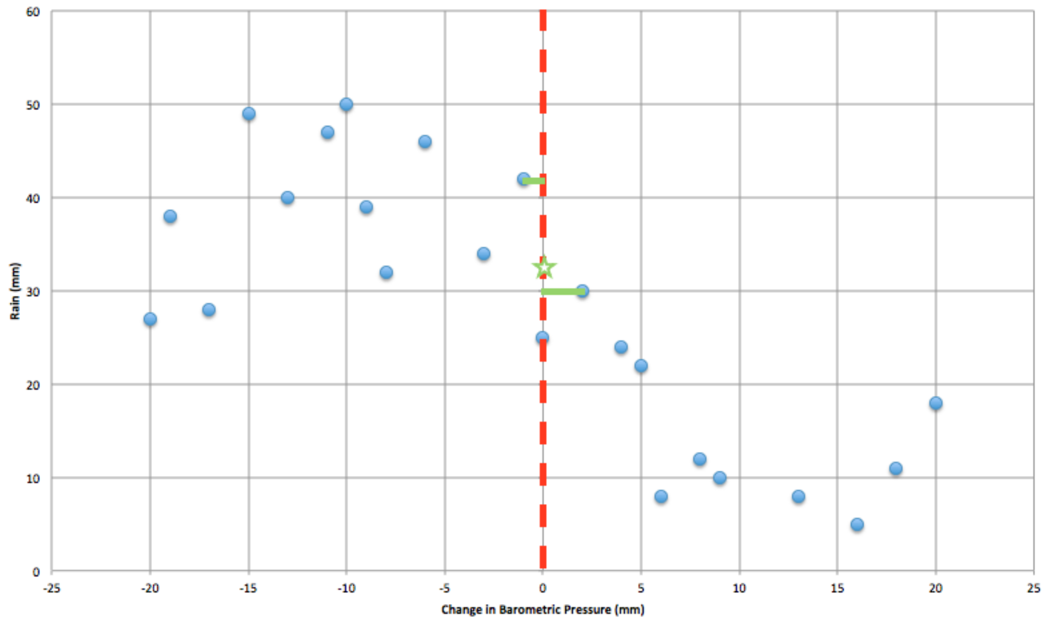
3.1.2 Training

As an instance learner, KNN simply stores the complete set of (X, y) tuples provided to it as training data.

3.1.3 Testing

When queried with a new data point, KNN will calculate (usually by Euclidean distance between X vectors) the K nearest neighbors from the training set. The y prediction for the new data point will be the mean y value of those K nearest neighbors.

For example, if a 3NN learner is asked to predict a y value for $x = 0$ in the following data:



it would calculate that the training data points at $x = 0, x = -1, x = 2$ are the three nearest neighbors to the test point $x = 0$. Averaging the y value of those training points gives an estimate of $y_{pred} = 32.33$, denoted by the green star. Note that this is a better estimate (more central to the data) than the actual $x = 0$ training point, which lies well below the general "curve" of the data.

When KNN is used for classification, rather than regression, the K nearest neighbors take a simple majority "vote" on the y prediction. For example, if the y value for two of the neighbors is *cat* and the third neighbor is *dog*, the prediction is *cat*. If there is a tie for the winning vote, selection is arbitrary. If there is a tie for the next nearest neighbor, selection is arbitrary.

Important notes: due to the nature of KNN, it can never extrapolate from the data, regardless of K. It can only interpolate. That is, the predictions for any x value to the left of the given data will be an unchanging horizontal line. The same goes for the right side. $K = 1$ produces a step function that exactly fits (and overfits!) the training data. $K = N$, where N is the number of data points, produces a single horizontal line for y_{pred} (that is, the model always predicts the same value, the mean of all y).

3.1.4 Python implementation tips

- Consider calculating a `distance` ndarray that holds the Euclidean distance from the test point to each training point, with the same indices as the training data.

- Consider using `numpy.argsort()` to quickly obtain the indices of the K nearest neighbors.

3.2 Kernel Regression

Kernel regression functions very similarly to K-Nearest Neighbors. It requires a kernel function as a hyperparameter, which usually serves to weight each neighbor's influence over y_{pred} by the neighbor's distance from the test point. In the KNN example above, the neighbors are weighted equally at 0.333. A Kernel Regression learner might weight the training point $x = 0$ at 0.5, the training point at $x = -1$ at 0.35, and the training point at $x = 2$ at 0.15 when considering what the final y result should be.

3.3 Linear Regression

3.3.1 Hyperparameters

Linear regression requires no hyperparameters.

3.3.2 Training

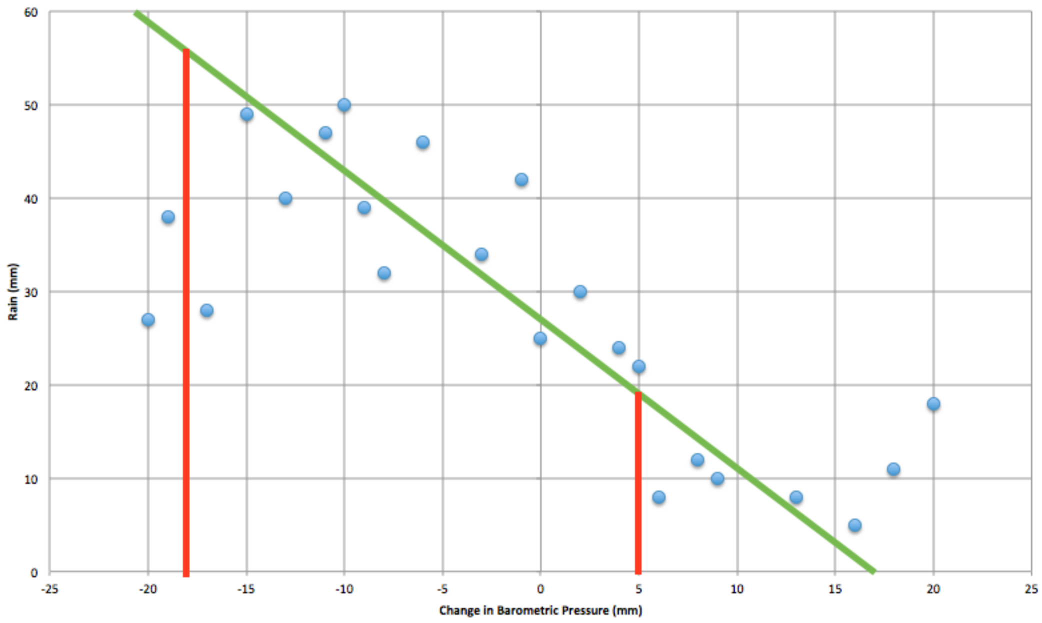
A linear regression learner is usually trained by an optimization algorithm such as *gradient descent*. A gradient descent optimizer will calculate each partial derivative of the loss function to be minimized. These partial derivatives are then captured as *update steps* performed at each iteration of the algorithm to take a "step" in the direction that will reduce the value of the cost function. To avoid passing over the optimal set of parameters (and increasing the cost function), a fractional *learning rate* is multiplied into the update step, so the optimizer only takes very small steps against the gradient. The optimizer should stop and return its final parameters (or coefficients) when the change in the cost function becomes very small.

Example: for a set of observations on a single feature, a linear regression learner will take the form of $y = mx + b$. A reasonable cost function to use is *sum of squared errors* (SSE). Thus our cost function, traditionally labelled $J(\theta)$ will be $J(m, b) = \frac{1}{2}\big(\sum_i (y_p[i] - y[i])^2\big)$, where $y_p[i] = mx[i] + b$ is the prediction for point $x[i]$, and $y[i]$ is the actual value for point $x[i]$. The partial derivatives will be $\frac{dJ}{dm} = (y_{pred} - y_{actl})x$ and $\frac{dJ}{db} = y_{pred} - y_{actl}$. The update steps will be $m_{new} = m - (y_{pred} - y_{actl})x \cdot lr$ and $b_{new} = (y_{pred} - y_{actl}) \cdot lr$, where lr is the fractional learning rate. We iteratively apply those step functions until ΔJ is small enough (subjective), then stop and use the final values for m and b as our locally-optimal coefficients.

For each additional X feature in the observations, one new coefficient will be needed, which means one more partial derivative and one additional update step. The process is otherwise identical for any dimensionality of X.

3.3.3 Testing

To query the linear regression learner: $y = mx + b$. That's it.



The non-linear nature of the underlying relationship in the data is obviously confounding our *biased* linear regression learner (because the bias we have given it via our linear assumption is *incorrect*). For $x = 5$, the model predicts $y = 19.5$, which is reasonable given our training data. However, for $x = -18$, the model predicts $y = 56$, which does not seem likely from our training data. (If not specifically constrained, the model would also begin predicting negative rainfall at $x > 17$, which is *definitely* not reasonable!)

3.3.4 Python implementation tips

- Use `scipy.stats.linregress` or `numpy.linalg.lstsq` or `numpy.polyfit` or `sklearn.linear_model.LinearRegression` or ...

3.4 Polynomial Regression

A polynomial regression learner works exactly as a linear regression learner except it takes a hyperparameter d, the degree of the polynomial. Calculating the partial derivatives for the optimization algorithm is a little more complex, but follows the same method.

3.5 Decision Tree

In this class, we discuss and use the CART algorithm for decision trees. It stands for Classification and Regression Trees, because the algorithm works for both. There are many other decision tree algorithms, such as ID3 and C4.5.

3.5.1 Hyperparameters

CART frequently uses hyperparameters `max_depth` (the greatest number of layers the tree is permitted to have) and `max_leaf_size` (the greatest number of data points a leaf node may contain).

3.5.2 Training

Training a decision tree with CART requires starting with a single node of a tree data structure and all of the training data. The training data is sorted and split in two along whatever X feature will give the most even split (greatest information gain for this "decision"). This tree node must remember both which X feature was split on and what the split value was. (*e.g.* `X[1] < 14.3`)

Once the split decision is made, the root tree node will add a new tree node as its left "child" in the tree, passing it the appropriate training data points (*e.g.* those where `X[1] < 14.3`). It will also add a new tree node as its "right" child in the tree, passing it the other data points (*e.g.* those where `X[1] >= 14.3`).

Each created tree node will perform the same steps to further subdivide its data and grow the tree downwards.

A node will designate itself a leaf node (and stop growing the tree in this direction) when its depth in the tree is too great (per the hyperparameter) or when it contains few enough data points (per the other hyperparameter), whichever happens first. The node will remember the mean y value of its data points. (For classification, the node will remember the *most common* y value among its data points.)

3.5.3 Testing

Testing a decision tree works exactly like searching a Binary Search Tree. Begin at the root node with your new data point. Test the X feature of your new data point according to this node's "decision". If the data point's matching feature is less than the split value, move to the left child node. Otherwise, move to the right child node. Continue this until you reach a leaf node. The y value in the leaf node is your prediction.

Important notes: Decision Trees are popular in ML because:

- they are easy to understand and visualize
- little-to-no filtering, preprocessing, or normalization of the data is required
- few-to-no assumptions are made about the data
- they support both classification and regression with few (or no) changes
- very large datasets (with max depth, etc) are no problem

- they are a “white box” that can be inspected to analyze precisely what the learner has captured during training

Decision Trees also have notable weaknesses:

- learning an “optimal” tree is an NP-Complete problem
- thus trees are built using greedy algorithms that only find a local optimum at each node/split
- it is easy for an unconstrained tree to overfit a data set (one data point per leaf node == 1NN)
- some learning tasks (like XOR, Parity) cause a degenerate tree that must enumerate every single feature combination (hence 2^N nodes required)

3.5.4 Python implementation

There was significant interest in the exact Python details of the Decision Tree algorithm, so here is a simple version of CART that I just wrote. It is similar to what we discussed in class, but written in exact Python syntax (and in fact tested). Additional error checking could be added, and the speed and memory compactness could be improved by vectorizing everything. This object-oriented, recursive code should very well illustrate the tree structure, however. As a bonus, I wrote a small tree visualization routine. Tree sibling nodes (same parent) are shown at the same indent level, left child first.

First, the `Node` class, which should be in a file named `Node.py`.

```
class Node():

  def __init__(self):
    self.split_factor = -1
    self.split_val = None
    self.left = None
    self.right = None
    self.y = None
```

Next, the `CART` class itself. Place it in a file named `CART.py`.

```
from Node import Node

class CART():

  def __init__(self, max_depth, max_leaf_size):
    self.max_depth = max_depth
    self.max_leaf_size = max_leaf_size

  def addEvidence(self, Xtrain, Ytrain):
    self.tree = self.addNode(Xtrain,Ytrain,0)
    self.draw_tree()


  def addNode(self, X, y, depth):
    node = Node()

    if (depth >= self.max_depth) or (X.shape[0] <= self.max_leaf_size):
      node.y = y.mean()
      return node

    best_unevenness = float('Inf')

    for factor in range(0,X.shape[1]):
      sorted = X[:,factor].argsort()
      split_idx = int(X.shape[0] / 2)

      while split_idx > 0 and
```

```
          (X[sorted][split_idx,factor] == X[sorted][split_idx-1,factor]):
        split_idx -= 1

      unevenness = abs(split_idx - (X.shape[0] - split_idx))

      if unevenness < best_unevenness:
        best_unevenness = unevenness
        best_factor = factor
        best_sorted = sorted
        best_split_idx = split_idx
        best_split_val = X[sorted][split_idx,factor]

  node.split_factor = best_factor
  node.split_val = best_split_val

  node.left = self.addNode(X[best_sorted][:best_split_idx],
                           y[best_sorted][:best_split_idx],
                           depth+1)
  node.right = self.addNode(X[best_sorted][best_split_idx:],
                            y[best_sorted][best_split_idx:],
                            depth+1)

  return node


def query(self, Xtest):
  node = self.tree

  while node.y == None:
    if Xtest[node.split_factor] < node.split_val:
      node = node.left
    else:
      node = node.right

  return node.y


def draw_tree(self):
  node = self.tree
  self.viz_tree_node(node,0)


def viz_tree_node(self, node, depth):
  print ' ' * (depth * 4),
  if node.y != None:
    print '( y =', node.y, ')'
```

```
      return

    print '(', node.split_factor, ',', node.split_val, ')'
    if node.left:
      self.viz_tree_node(node.left,depth+1)
    if node.right:
      self.viz_tree_node(node.right,depth+1)
```

Finally a simple test class that throws random integers from [0,9] into ten different `X` factors, and random real numbers from `[0,1)` into `y`. It trains the tree on 1,000 data points, prints the constructed decision tree, then queries a single new (also random) data point. Call it `test.py`.

```
from CART import CART
import numpy as np

np.random.seed(1)

learner = CART(max_depth=5, max_leaf_size=10)
learner.addEvidence(np.random.randint(0,10,(1000,10)),
                    np.random.random((1000)))

test_x = np.random.randint(0,10,(10))
print (test_x)

y = learner.query(test_x)

print y
```

Here is the output when `test.py` is run:

```
 ( 4 , 5 )
     ( 2 , 5 )
         ( 1 , 5 )
             ( 8 , 5 )
                 ( 6 , 4 )
                     ( y = 0.506843749407 )
                     ( y = 0.437679850308 )
```

```
            ( 3 , 4 )
                ( y = 0.416112682817 )
                ( y = 0.433983579597 )
        ( 0 , 5 )
            ( 2 , 2 )
                ( y = 0.505386599754 )
                ( y = 0.476225164209 )
            ( 6 , 5 )
                ( y = 0.352004941032 )
                ( y = 0.522232421971 )
    ( 6 , 5 )
        ( 1 , 5 )
            ( 3 , 5 )
                ( y = 0.48086569684 )
                ( y = 0.528874301914 )
            ( 1 , 8 )
                ( y = 0.514486008055 )
                ( y = 0.542548647806 )
        ( 5 , 5 )
            ( 1 , 4 )
                ( y = 0.493067047936 )
                ( y = 0.596097973342 )
            ( 0 , 5 )
                ( y = 0.559703192934 )
                ( y = 0.477686997068 )
( 3 , 5 )
    ( 5 , 5 )
        ( 6 , 5 )
            ( 0 , 4 )
                ( y = 0.485421747801 )
                ( y = 0.490697427904 )
            ( 8 , 5 )
                ( y = 0.569044836279 )
                ( y = 0.456174252322 )
        ( 6 , 5 )
            ( 8 , 6 )
                ( y = 0.605733003662 )
                ( y = 0.447743020573 )
            ( 2 , 5 )
                ( y = 0.394878571101 )
                ( y = 0.426039174536 )
    ( 8 , 5 )
        ( 5 , 5 )
            ( 1 , 6 )
                ( y = 0.425083177118 )
                ( y = 0.453633440153 )
```

```
                    ( 1 , 4 )
                        ( y = 0.37580739646 )
                        ( y = 0.582438814931 )
                ( 2 , 5 )
                    ( 1 , 5 )
                        ( y = 0.518392482903 )
                        ( y = 0.401703225427 )
                    ( 1 , 5 )
                        ( y = 0.461105348843 )
                        ( y = 0.482284851569 )
[6 5 6 1 9 9 1 3 2 7]
0.605733003662
```

You can trace the query for the new data point (features on next-to-last line) and validate the result yourself (y prediction on last line).

Node split	Test feature	Result
`X[4] < 5`	`X[4] == 9`	`right`
`X[3] < 5`	`X[3] == 1`	`left`
`X[5] < 5`	`X[5] == 9`	`right`
`X[6] < 5`	`X[6] == 1`	`left`
`X[8] < 6`	`X[8] == 2`	`left`

We have reached a leaf node with stored mean value `y == 0.605733003662`, and so that is our estimated `y` for this `test_x` point. Note that after training, the decision tree can only predict the exact mean `y` value stored in one of its leaf nodes. No other values will ever be predicted.

In class on Tuesday, we will discuss an alternate method of data storage: keeping the entire tree in a single cleverly-arranged `ndarray`!

4 Ensemble Learners

An *ensemble learner* (or ensemble *of* learners) is any learning algorithm that operates by combining multiple other learning algorithms. Each learner in the ensemble will be individually trained in some manner. When a query arrives at the ensemble, each contained learner will be independently queried and the resulting answers combined in some way to produce the ensemble's "final" answer.

For example, we could do the following:

$$\begin{tikzcd}
trainX \rightarrow & KNN \arrow[d] & LinReg \arrow[d] & DTree \arrow[d] & SVM \arrow[d] \\
testX \rightarrow & model \arrow[d] & model \arrow[d] & model \arrow[d] & model \arrow[d] \\
 & y_a \arrow[d] & y_b \arrow[d] & y_c \arrow[d] & y_d \arrow[d] \\
 & {} & {} & {} & {}
\end{tikzcd}$$

The four different answers produced for y are then combined. For a regression problem, it is common to take the mean value. For a classification (discrete output) problem, we usually take the most common answer.

For a "pure ensemble" as shown here, it is beneficial to select learners that are *as different as possible.* This will allow the ensemble to generalize better than its component learners, because the various biases built into each model can offset one another. (Example: combine several PolynomialLearners with different d and several KNNLearners with different K.)

Indeed, this is the hope of ensemble learning, proven out in the real world: to produce models with smaller error and less overfitting than the component learners. Virtually all Machine Learning competitions (Amazon, Netflix, Kaggle, etc) are won by ensemble learners.

4.1 Bootstrap Aggregation

Bootstrapping is a statistical term for any metric that relies on random sampling with replacement and is thus related to *data augmentation* (methods to obtain more training/testing sets from the same initial data set). The bootstrap aggregating algorithm (abbreviated as *bagging*) is a type of ensemble usually applied to a single component learner type to improve its performance.

The BagLearner requires another type of learner as a hyperparameter (the learner it should make copies of as its "bags"), the number of copies (or bags) it should create, as well as whatever hyperparameters are needed by that "inner" learner. For example, you might construct a new BagLearner with hyperparameters *KNNLearner*, $bags = 10$, and $K = 5$. When coding this in Python, it is critical to examine and understand Python's `kwargs` (keyword arguments) ability. This is the capability that will allow your BagLearner to accept (from its perspective) the arbitrary hyperparameters needed for the "inner" learner and pass them along to that learner without caring what they are.

The BagLearner looks exactly like the general ensemble shown above, except:

1. Each contained learner is of the same type, with the same hyperparameters.

2. Each contained learner ("bag") has the same *size* as the full training set, but is *sampled with replacement*.

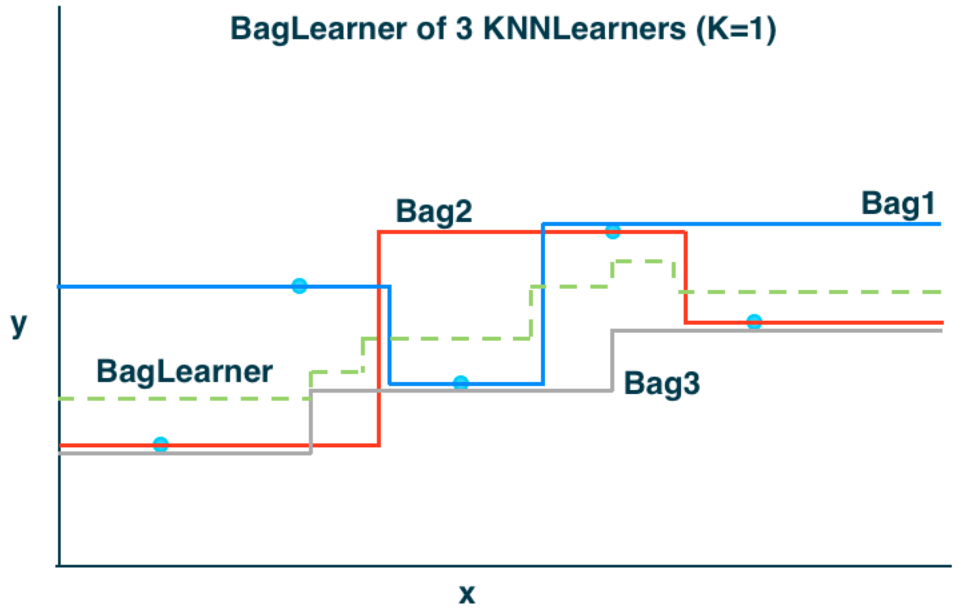
For example, if a `BagLearner` is created with hyperparameters `KNNLearner, bags=10, k=5`, then in its `__init__` method, it must create 10 separate copies of `KNNLearner`, passing each one the necessary `k=5` hyperparameter. Those ten learners must be retained in the `BagLearner`, perhaps in a list.

When `BagLearner.addEvidence` is called, it must in turn call `addEvidence` on each contained learner.

When `BagLearner.query` is called, it must call `query` on each contained learner, then combine the results to produce its answer.

In general, bagging is a good way to smooth out the results of *unstable* learners like CART, where a small change in the observations can create a large change in the prediction.

Here is a small visual example of bagging producing a more "central tendency" prediction in some data, with less overfitting. Overlapping lines are slightly separated for readability.



4.2 Boosting

Another common ensemble method is *boosting*, the process of training inner learners *one at a time*, evaluating the overall learner after each inner learner is added, and using this to better inform the next learner that is trained.

We use the popular AdaBoost method (named for *adaptive boosting*, not for early programmer Ada Lovelace) and apply it in a similar manner to our BagLearner.

Differences from bagging:

1. With AdaBoost, the training set data points are *weighted* for each sampling pass. (Bagging is uniform sampling.)

2. The weighting for a data point is proportional to the training error of the model-so-far on this specific data point.

The intent of AdaBoost is therefore obvious. We iteratively learn a series of weak classifiers. Data points for which the current model (all learners created so far, taken as an ensemble) performs *poorly* (high error) become more likely to be chosen in future samples. We could say the model is "concentrating" on improving data points it does not predict well.

When querying the boosted learner, each contained *model* is proportionally weighted (the strength of its contribution to the final answer) based on that model's total training error.

By virtue of this iterative training, AdaBoost is more powerful when learning "difficult" data, but could also be more capable of overfitting the data.

AdaBoost is far from the only popular boosting method. BrownBoost improves generalization by doing almost exactly the opposite. It slowly *underweights* points with high training error, effectively discarding them as probable outliers.

If you wish to attempt the Boosting extra credit, a good general source is the Wikipedia entry (https://en.wikipedia.org/wiki/AdaBoost) and a good academic source is Jan Šochman's presentation, which you can find at: http://cmp.felk.cvut.cz/~sochmj1/adaboost_talk.pdf . The latter has a particularly nice set of visualizations at each step of the boosting algorithm.