## Linear Regression

This ended up taking me 3x longer than I thought to implement.

First time seeing np.mat objects. I just assumed that they worked identically to np.array objects, but with the constraint of being 2 dimensional. Turned out to not be the case and the API's are different. When I got an issue with doing mat1 * mat2 trying to broadcast elementwise multiply (hadamard) two matrices, I was very confused when I got errors saying that the dimensions didn't line up for a dot product.
It's been a while since I've done stuff in numpy, so I wasn't sure whether I was incorrectly remembering what "*" did for arrays, and I spent a whlie pouring over np.array documentation wondering what the hell I was doing wrong. In the end, I forgot that the provided template code used np.mat for initialization and I was looking in the wrong place. Turns out for np.mat, "*" uses matrix multiplication and I was supposed to do np.multiply(mat1, mat2) to get what I wanted. I'm used to the */@ notation for np.array, so with instructor permission, I swapped np.mat's to np.array's to keep things consistent in my mind.

I also shot myself in the foot with broadcasting. For the example, I assumed that my gradients were a (2, 1) array, when in reality they were a (2,) array (one-dimensional). This ended up leading to gradient descent failing. After a lot of confusion, I printed out my coefficient array and found that instead of a (2, 1) array I had a (2, 2) array. Turns out, when I was doing the update step on the coefficients, which were a (2, 1) array, the broadcasting went something like this:

```
2 1 => 2 2 => 2 2
  2      2    2 2
```

After reshaping my gradients from (2,) to (2,1). Everything worked out.

For the normalization, I knew that it would help with the training speed, but I was genuinely surprised to see just how much it helped. For the unormalized version, after about a minute and a half of training (did not wait for it to finish), it only got to about 11.6/20 for the intercept coefficient. What's interesting to note is that the slope coefficient converged almost instantaneously in comparison. Only the intercept term had the slow convergence issue. After normalizing, the intercept term also converged near instantaneously. The numbers were of course different, due to X values now representing % distance between min & max, but undoing the transformation, the intercept and slope values were the same.

Coefficients before transformation:
[20, 30]

Coefficients after transformation:
[50, 29850]

Transformations:
1) Subtraction by a constant: -min. min=1. Entire line shifted to the left by one unit. Old intercept was @ 20, after the shift of "1", since the slope is 30, the new intercept should be 20+30=50.
2) Division by a constant: /(max-min). max=996, min=1, max-min=995. x values are all "squished" together increasing the slope by a factor of 995. old slope is 30. 30*995=29850.

Numbers check out.

When performing predictions with new data, you need to provide the same exact transformation, with the same min and max, and plug those values into the model.

## Logistic Regression

Similar to my linear regression implementation, I was doing a manual backward pass involving applying the chain rule and accumulating gradients. As logistic regression is essentially linear regression but with a softmax slapped at the end, I thought that this would be fairly straightforward. Boy was I wrong.

After I coded up my forward and backward passes and tried plotting the loss over time, it ended up being pretty much the exact opposite of what I wanted with it increasing over time. I realized that I was miscalculating the gradient at a step, and then fixed it.

Now, loss was decreasing, however, this time it never converged. Huh… strange. Then I found out that I mis-defined logistic loss and my function had no minima. It just infinitely stretched downwards. Turns out that I forgot a negative sign on one of the terms:

```
g_log_yhat = g_losses * -self.y # forgot to multiply by -1
```

It took me 40 minutes to find the negative bug above… Subtle mistakes like these really dragged out the time it took me to implement a manual backward pass, and when it finally converged properly, I then realized that the accuracy was terrible. I spent a long long time pouring over my code, and I couldn’t spot anything that seemed off. Furthermore, I used the pytorch autograd engine to sanity check my gradients, and they all matched up. When I went to Shang’s office hours, even he said that my manual backward pass seemed OK, and if he had to guess, it was an issue with my gradients diverging from the true gradients on subsequent gradient descent iterations.

After that OH session, I spent another hour or so pouring over my code to see where it all went wrong, but in the end, I got frustrated and decided to give up on my ideals of a manual chain rule’d backward pass. I just used the pre-computed update rule calculated with calculus (same as in linear regression), and everything converged properly.

Perhaps one day, I’ll have enough sanity to go back and figure out what went wrong.

## Naive Bayes

This was one of the models that actually ended up being pretty quick to implement. Since it's just naive bayes, and not gaussian naive bayes, it simplifies greatly and it's essentially just constructing a dictionary with pre-computed values. Iterate over each class label and then compute all the conditional probabilities for each feature. Using for-loops like this did not feel great, and I was wondering if there was some vectorized way to do this, similar to how I did it in linear and logistic regression. In the end, I couldn't think of anything and just decided to follow the sample template and do it with for loops.

Since probabilities are being multiplied, there can be an issue with extremely small numbers being multiplied by other extremely small numbers, leading to underflow. With this problem here, as suggested in the template, you can just take the log of these values, and then accumulate them with a sum instead of a multiplication. It's important to change the accumulation variable from a 1 to a 0 in this case.

Prediction is also fairly straightforward. Iterate over the class labels, and calculate the likelihoods for each class. In this case, there is no smoothing applied, so if log returns a NaN, the comparisons later on for max, just end up excluding it from comparison.

When evaluating against the sklearn Naive Bayes classifier, I saw that there was a discrepancy in accuracies and values. When investigating by looking at individual values, it was a silly bug of just forgetting to multiply by the priors. Adding in the corresponding dictionary lookup and addition of a logprob, solved the issue.

## KMeans

I'm honestly really happy with my KMeans implementation. I leveraged reshaping and broadcasting to merge together a bunch of for loops in the template.

First you can use np.random.choice to get indices for datapoints to use as initial cluster centroids, and then pass in an array index to get the datapoint values. For calculating the distance of each point to the centroids, you can make this extremely efficient with broadcasting. The data array X is a MxN matrix. (M=#datapoints, N=#features), and the centroid array is KxN (K=#centroids).

Essentially, for every centroid which can be thought of as an (N,) array, you want to calculate the difference between every datapoint and it. X - centroids => (M,N). Broadcasting in of itself, accomplishes this pretty straightforward for a single centroid.

```
M N => M N
  N    M N
```

However, there isn’t a single centroid, but instead K centroids. If you were to do X - centroids as is, it would be:

```
M N => INVALID
K N
```

This ends up being an invalid operation as M & K conflict in dimensions. To fix this, if you reshape K, M to K, 1, N, the broadcasting becomes valid. The broadcasting is now:

```
  M N =>   M N => K M N
K 1 N    K M N    K M N
```

The way I think of this is as duplicating a single centroid vector for every datapoint so that there’s an easy elementwise subtraction, and then duplicating the dataset for each centroid comparison. Once again, it’s essentially calculating for each centroid, calculate differences for the dataset. Where the “K” dimension essentially is a collection of differences.

For each of these distance vectors, you want to calculate its length. This can be achieved pretty simply with inducing a norm. Since each distance vector has N values, it makes sense to induce the norm along the third axis, or in 0-index speak, axis 2.

```
np.linalg.norm(distance_vectors, axis=2)
```

This gets you a K M matrix where each row are the distance vectors for each vector to the centroid. Afterwards, since you want the “closest” centroid for each datapoint, you can do this by calculating an argmin across the K axis. This now gets you a M dimensional vector with the index of the closest centroid. This completes the assignment step.

For the update step, you update centroids to the mean of the assigned datapoints. I couldn’t think of a good way to broadcast this and do it in a single line, so I just resorted to iterating over each centroid, and calculating means across the datapoint axis (M). This results in a N dimensional vector that represents the mean.

Since the API specified in the template wants us to return predictions one-hot encoded, I found that using the identity matrix is a very clean way to get the encodings. With this, I can generate an array of indices, and use a single line to index out the one hot encodings for every datapoint.

```
self.RM = np.eye(self.k)[self.closest_centroid]
```

## Decision Tree + Random Forests

I'm quite happy with my decision tree implementation. I loosely followed the code template provided, mainly removing a lot for loops inplace for broadcasting. Reduced the cell from ~300 liens to ~120 lines.

In order to avoid bugs, I created a 2d dataset, where the optimal splits were easily verifiable, and then used that every step of the way to make sure that my code was working as intended. To test max_depth, I added an opposite class in the middle of a data cluster, which effectively made the decision tree add for more splits to achieve purity. I guess this wasn't especially needed to test max_depth, but I thought it was a nice test case for my code.

Calculating entropy, information gain, and generating the splitting rules was pretty straightforward. For the splitting rules, since all features provided are numeric, for a feature you can have two copies, remove the first element from one, and last element from the other, stack them, and then calculate elementwise averages. This very efficiently gets the midpoints.

For finding the best split, it's pretty much just a case enumerating all possible splits and calculating information gain, and from that point, choosing the best split. The code calculates the best split for each feature, and takes a max over those values, calculating the best split across features.

An interesting bug that I didn't notice until much later was that if there's a set of datapoints which are all the same, but have different labels, it passes through the purity check, and when the code tries to split that dataset, since there's only one unqiue value for each feature, the threshold ends up being undefined. This caused me a bit of confusion when I was implementing random forests, but it was a pretty quick fix of just checking if the threshold is none, and then returning the node early, like the other base cases.

On the topic of base cases: recursion. The way I implemented the tree generation was starting from the bottom-most left-most node, and then going up, and then down the right side. This meant that the root node was the last node to be created. The traversal method shouldn't affect the decision tree, but I thought that this was a bit interesting.

A notable way which my implementation slightly differed from the template was the Node nested class. Nodes only track feature_col, threshold, children, and majority_class. Nothing else. Every single node stores a majority_class value. If there are no thresholds defined, or no children, then it knows that it's a leaf node, and can just return that value during prediction. Furthermore, when predicting, I put the predict() function inside the node class. If it does have a threshold and children, then it calls one of its child's predict functions. Because of this, I didn't really need a separate ind_predict class and could just iterate through datapoints and call the root's predict function.

When transitioning to the random forest ensemble, it was pretty much sampling a subset of the data, and then training a new decision tree on that data. The only part that was tricky, was the predict function. Since the decision trees operate on separate subsets, it means that the data passed into it also has to be of the same feature subset. On top of tracking the decision tree references, I also had to track which columns were being used. So self.trees was an array of tuples with the tuple's first element being the tree reference, and the second being a list of features used.

Since the wine dataset labels are ordinal, it might make sense to use the median to average predictions, but since our loss is purely accuracy and considers (pred: 6 actual: 7) to be equally wrong as (pred: 1 actual: 7), I took the mode of the predictions.

Something else that's interesting is just how much faster it is to train the weak learners. When taking a subset of features, that already cut down on time drastically, and when taking a subset of the data, the entire training process for the 100 trees took 3 seconds.