https://github.com/oetia/dsc148-finalproject

Models Implemented:
Linear Regression
Logistic Regression
Naive Bayes
KMeans + Gaussian Mixture Models
Decision Tree + Random Forests

## Linear Regression Writeup

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 while 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 that 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 unnormalized 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.

## Linear Regression Code

```
class Linear_Regression():
    def __init__(self, alpha = 1e-10 , num_iter = 10000, early_stop = 1e-50, intercept = True,
init_weight = None, verbose = False):


        """
            Some initializations, if necessary

            attributes:
                        alpha: Learning Rate, default 1e-10
                        num_iter: Number of Iterations to update coefficient with training
data
                        early_stop: Constant control early_stop.
                        intercept: Bool, If we are going to fit a intercept, default True.
                        init_weight: Matrix (n x 1), input init_weight for testing.


            TODO: 1. Initialize all variables needed.
        """

        self.model_name = 'Linear Regression'

        # store the variables in object scope
        self.alpha: float = alpha
        self.num_iter: int = num_iter
        self.early_stop: float = early_stop
        self.intercept: bool = intercept
        self.init_weight = init_weight  ### For testing correctness.

        self.verbose = verbose


    def fit(self, X_train, y_train):
        """
            Save the datasets in our model, and perform gradient descent.

            Parameter:
                X_train: Matrix or 2-D array. Input feature matrix.
                Y_train: Matrix or 2-D array. Input target value.


                TODO: 2. If we are going to fit the intercept, add a col with all 1's to the
first column. (hint: np.hstack, np.ones)
                      3. Initilaize our coef with uniform from [-1, 1] with the number of col
in training set.
                      4. Call the gradient_descent function to train.
        """

        self.X = np.array(X_train)
        self.y = np.array(y_train).reshape(-1, 1)

        if self.intercept:
            ones = np.ones((self.X.shape[0], 1))
            self.X = np.hstack([ones, self.X])
            self.log(self.X.shape)
            self.log(self.X)

        if self.init_weight is None:
            np.random.seed(16)
            self.coef = np.random.uniform(-1, 1, (self.X.shape[1], 1),)
            self.log(self.coef)
        else:
            self.coef = np.array(self.init_weight).reshape(self.X.shape[1], 1)

        self.gradient_descent()

        # self.coef = self.init_weight #### Please change this after you get the example
right.

    # unused function. functionality merged into gradient_descent
    # with a manual backward pass, intermediate forward pass calclations are out of scope
    def gradient(self):
        """
            Helper function to calculate the gradient respect to coefficient.

            TODO: 5. Think about the matrix format of the gradient of the loss function.
        """

        d_loss = 1
        d_diff = d_loss * 2 * (self.y - self.X @ self.coef)
        d_pred = d_diff * -1
        d_coef = (np.asarray(d_pred) * np.asarray(self.X)).sum(axis=0)

        # doesn't make sense to have to calculate these values twice
        # i can just keep the gradient inside the gradient_descent function
        grad_coef = (-2 * (self.y - self.X @ self.coef) * self.X).sum(axis=0)

        print(np.isclose(d_coef, grad_coef))
        self.grad_coef = grad_coef

    def gradient_descent(self):

        """
            Training function

            TODO: 6. Calculate the loss with current coefficients.
                  7. Update the temp_coef with learning rate and gradient.
                  8. Calculate the loss with temp_coef.
                  9. Implement the self adeptive learning rate.
                      a. If current error is less than previous error, increase learning rate
by a factor 1.3.
                         And update coef, with temp_coef.
                      b. If previous error is less than current error, decrease learning rate
by a factor of 0.9.
                         Don't update coef.
                  10. Add the loss to loss list we create.
        """

        self.loss = []

        for i in range(self.num_iter):

            # forward pass
            preds = self.X @ self.coef
            errors = self.y - preds
            squared_errors = (errors**2)
            loss = squared_errors.sum()

            self.loss.append(loss)

            # backward pass
            dloss = 1
            dsquared_errors = dloss * np.ones_like(squared_errors)
            derrors = dsquared_errors * 2 * errors
            dpreds = derrors * -1
            # dcoef = (dpreds * self.X).sum(axis=0) # for a given coefficients affects a
column of x's, which each affect a different prediction
            dcoef = (dpreds * self.X).sum(axis=0).reshape(-1, 1) # for a given coefficients
affects a column of x's, which each affect a different prediction

            temp_coef = self.coef - self.alpha * dcoef
            temp_loss = np.sum((self.y - self.X @ temp_coef)**2)
            pre_error = loss

            current_error = temp_loss

            ### This is the early stop, don't modify fllowing three lines.
            if (abs(pre_error - current_error) < self.early_stop) | (abs(abs(pre_error -
current_error) / pre_error) < self.early_stop):
                print(f"EARLY STOP @ {i}")
                self.coef = temp_coef
                return self

            if current_error <= pre_error:
                self.alpha *= 1.3
                self.coef = temp_coef
            else:
                self.alpha *= 0.9 # don't update coefs

            self.loss.append(loss)
            if i % 10000 == 0:
                print('Iteration: ' +  str(i))
                print('Coef: '+ str(self.coef))
                print('Loss: ' + str(current_error))

        return self

    def ind_predict(self, x: list):
        """
            Predict the value based on its feature vector x.

            Parameter:
            x: Matrix, array or list. Input feature point.

            Return:
                result: prediction of given data point
        """

        """
            TODO: 11. Implement the prediction function
        """
        # assumes x has intercept added if used here
        return np.array(x) @ self.coef

    def predict(self, X):
        """
            X is a matrix or 2-D numpy array, represnting testing instances.
            Each testing instance is a feature vector.

            Parameter:
            X: Matrix, array or list. Input feature point.

            Return:
                ret: prediction of given data matrix
        """

        """
        TODO: 12. Make sure add the 1's column like we did to add intercept.
                  13. Revise the following for-loop to call ind_predict to get predictions.

        """

        if self.intercept:
            ones = np.ones((X.shape[0], 1))
            X = np.hstack([ones, X])

        return X @ self.coef

    def log(self, str):
        if self.verbose == True:
            print(str)
```

## Logistic Regression Writeup

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.

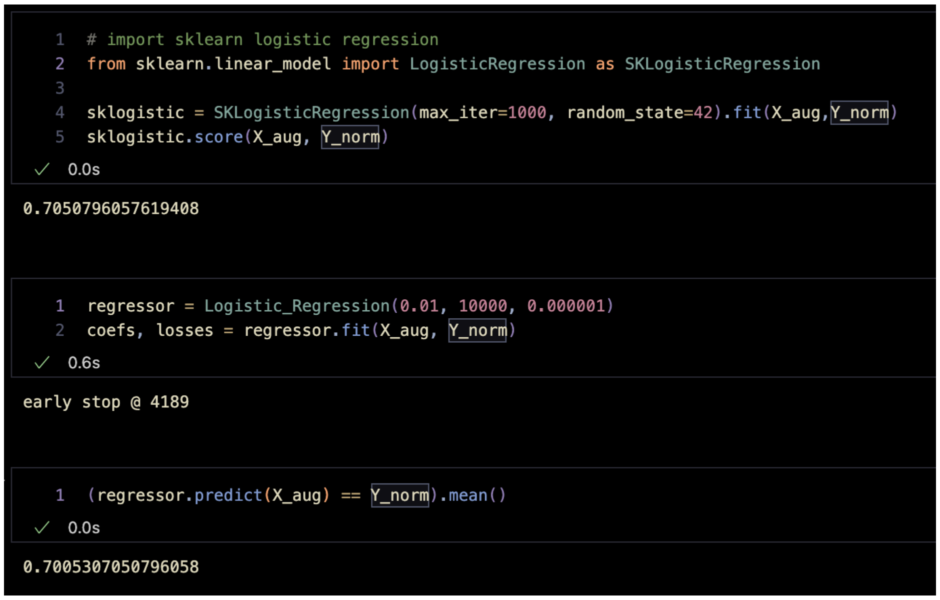
Furthermore, I used the pytorch autograd engine to sanity check my gradients, they all matched up. When I went to Shang's office hours, even he said that my manual backward pass & autograd sanity check seemed OK, and if he had to guess, it was an issue with my gradients diverging from the true gradients 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, and everything converged properly.

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

## Logistic Regression Code

```
def z_standardize(X: pd.DataFrame):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    return (X - means) / stds

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# It's really clunky to add in intercept weights + standardization IN the model
# It shouldn’t be the model’s job to do preprocessing
# Assumes that ones are hstacked and, labels are 0-1’d, and standardization is applied BEFORE
being put into the model
# If you need it for grading, here would be the code. Would be in fit()
# self.X = np.hstack([
#     np.ones((X_train.shape[0], 1)),
#     z_standardize(X_train)
# ])
# self.y = np.where( # creates boolean mask then maps
#     y_train == max(np.unique(y_train)),
#     0, 1
# ).reshape(-1, 1)

# Here’s the code I’m using outside to do this preprocessing
# X_std = z_standardize(pd.DataFrame(X))
# X_aug = np.hstack((np.ones((X.shape[0], 1)), X_std))
# Y_norm = np.where(Y == 5, 0, 1)


class Logistic_Regression():

    def __init__(self, alpha: float, iterations: int, early_stop: float = 0.0001):
        self.alpha = alpha
        self.iterations = iterations
        self.early_stop = early_stop

    def fit(self, X, y):
        self.X = X
        self.y = y.reshape(-1)
        self.theta = np.zeros(X.shape[1])

        return self.gradient_descent()

    def cost_function(self):
        z = self.X @ self.theta
        predict_1 = self.y * -np.log(sigmoid(z))
        predict_0 = (1 - self.y) * -np.log(1 - sigmoid(z))
        loss = predict_1 + predict_0
        return np.mean(loss)

    def gradient_descent(self):
        losses = []
        for i in range(self.iterations):
            z = self.X @ self.theta
            diff = sigmoid(z) - self.y
            gradient = self.X.T @ diff / self.X.shape[0]
            self.theta -= self.alpha * gradient
            loss = self.cost_function()
            losses.append(loss)

            temp_theta = self.theta - self.alpha * gradient
            pre_error = loss
            temp_sigmoid = sigmoid(self.X @ temp_theta)
            temp_error = (-self.y * np.log(temp_sigmoid) - (1 - self.y) * np.log(1 -
temp_sigmoid)).mean()

            if (abs(pre_error - temp_error) < self.early_stop) | (abs(abs(pre_error -
temp_error) / pre_error) < self.early_stop):
                # return temp_theta, temp_b
                print(f"early stop @ {i}")
                return temp_theta, losses
            # my attempted manual backward pass before i tilted and gave up
            # # FORWARD
            # dot_products = self.X @ self.theta
            # dot_products_neg = -dot_products
            # exp = np.exp(dot_products_neg)
            # exp_plus_1 = 1 + exp
            # y_hat = exp_plus_1 ** -1

            # om_yhat = 1 - y_hat
            # log_yhat = np.log(y_hat)
            # log_om_yhat = np.log(om_yhat)
            # losses = -self.y * log_yhat + -(1-self.y) * log_om_yhat
            # risk = losses.mean()

            # self.loss.append(risk)

            # # BACKWARD
            # g_risk = np.array(1.0)
            # g_losses = g_risk * (np.ones_like(losses) / losses.shape[0]) # forgot to divide
by # training examples
            # g_log_yhat = g_losses * -self.y # forgot to multiply by -1
            # g_log_om_yhat = g_losses * -(1-self.y)
            # g_om_yhat = g_log_om_yhat * (1 / om_yhat)
            # g_yhat = (g_log_yhat * (1 / y_hat) +
            #         g_om_yhat * -1)
            # g_exp_plus_1 = g_yhat * (-1 / exp_plus_1 ** 2)
            # g_exp = g_exp_plus_1 * 1
            # g_dot_products_neg = g_exp * exp
            # g_dot_products = g_dot_products_neg * -1
            # g_theta = (g_dot_products * self.X).sum(axis=0).reshape(-1,


        return self.theta, losses

    def predict(self, X):
        return np.where(sigmoid(X @ self.theta) >= 0.5, 1, 0)
```



```
# import sklearn logistic regression
from sklearn.linear_model import LogisticRegression as SKLogisticRegression

sklogistic = SKLogisticRegression(max_iter=1000, random_state=42).fit(X_aug,Y_norm)
sklogistic.score(X_aug, Y_norm)
```
✓ 0.0s

```
0.7050796057619408
```

```
regressor = Logistic_Regression(0.01, 10000, 0.000001)
coefs, losses = regressor.fit(X_aug, Y_norm)
```
✓ 0.6s

```
early stop @ 4189
```

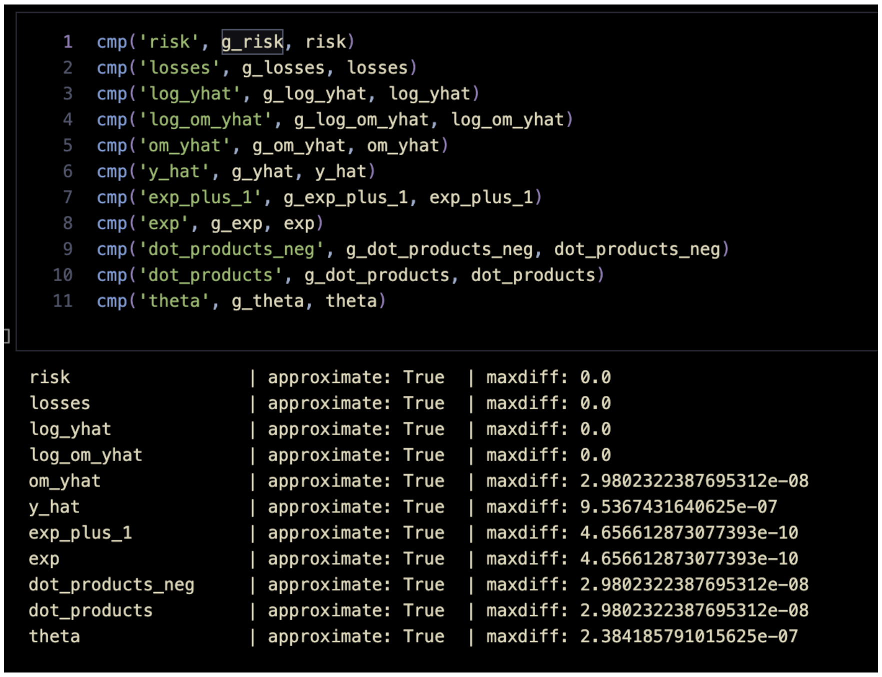
```
(regressor.predict(X_aug) == Y_norm).mean()
```
✓ 0.0s

```
0.7005307050796058
```

## Logistic Regression Pytorch Autograd Engine Sanity Check

Not necessary, but included since I thought it was interesting. You can just skip, but if you can find anything wrong with my gradient calculations, I'd appreciate it. I have no clue where this is going wrong.

```
def cmp(s, dt, t):
  ex = torch.all(dt == t.grad).item()
  app = torch.allclose(dt, t.grad)
  maxdiff = (dt - t.grad).abs().max().item()
  # print(f'{s:15s} | exact: {str(ex):5s} | approximate: {str(app):5s} | maxdiff: {maxdiff}')
  print(f'{s:20s} | approximate: {str(app):5s} | maxdiff: {maxdiff}')

X = torch.tensor([
    [1, 1, 2, 3],
    [1, 4, 5, 6],
    [1, 7, 8, 9]
], dtype=torch.float32)

y = torch.tensor([1, 0, 1]).reshape(-1, 1)
g = torch.Generator().manual_seed(2147483647)
theta = torch.randn((4, 1), generator=g, requires_grad=True)
theta.grad = None # i guess that i didn't zero the gradients when rerunning this
Theta

# forward pass
theta.grad = None # forgot to flush gradients. things ended up weird b.c. of this.
dot_products = X @ theta
dot_products_neg = -dot_products
exp = torch.exp(dot_products_neg)
exp_plus_1 = 1 + exp
y_hat = exp_plus_1 ** -1
om_yhat = 1 - y_hat
log_yhat = torch.log(y_hat)
log_om_yhat = torch.log(om_yhat)
losses = -y * log_yhat + -(1-y) * log_om_yhat
risk = losses.mean()

for t in [theta, dot_products, dot_products_neg, exp, exp_plus_1, y_hat, om_yhat, log_yhat,
log_om_yhat, losses, risk]:
    t.retain_grad()
risk.backward()

g_risk = torch.tensor(1.0)
g_losses = g_risk * (torch.ones_like(losses) / losses.shape[0]) # forgot to divide by #
training examples
g_log_yhat = g_losses * -y # forgot to multiply by -1
g_log_om_yhat = g_losses * -(1-y)
g_om_yhat = g_log_om_yhat * (1 / om_yhat)
g_yhat = (g_log_yhat * (1 / y_hat) +
          g_om_yhat * -1)
g_exp_plus_1 = g_yhat * (-1 / exp_plus_1 ** 2)
g_exp = g_exp_plus_1 * 1
g_dot_products_neg = g_exp * exp
g_dot_products = g_dot_products_neg * -1
g_theta = (g_dot_products * X).sum(dim=0).reshape(-1, 1)
```


```
cmp('risk', g_risk, risk)
cmp('losses', g_losses, losses)
cmp('log_yhat', g_log_yhat, log_yhat)
cmp('log_om_yhat', g_log_om_yhat, log_om_yhat)
cmp('om_yhat', g_om_yhat, om_yhat)
cmp('y_hat', g_yhat, y_hat)
cmp('exp_plus_1', g_exp_plus_1, exp_plus_1)
cmp('exp', g_exp, exp)
cmp('dot_products_neg', g_dot_products_neg, dot_products_neg)
cmp('dot_products', g_dot_products, dot_products)
cmp('theta', g_theta, theta)
```

```
risk                 | approximate: True  | maxdiff: 0.0
losses               | approximate: True  | maxdiff: 0.0
log_yhat             | approximate: True  | maxdiff: 0.0
log_om_yhat          | approximate: True  | maxdiff: 0.0
om_yhat              | approximate: True  | maxdiff: 2.9802322387695312e-08
y_hat                | approximate: True  | maxdiff: 9.5367431640625e-07
exp_plus_1           | approximate: True  | maxdiff: 4.656612873077393e-10
exp                  | approximate: True  | maxdiff: 4.656612873077393e-10
dot_products_neg     | approximate: True  | maxdiff: 2.9802322387695312e-08
dot_products         | approximate: True  | maxdiff: 2.9802322387695312e-08
theta                | approximate: True  | maxdiff: 2.384185791015625e-07
```

## Naive Bayes Writeup

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 -inf, there shouldn't be any issues with comparisons later on.

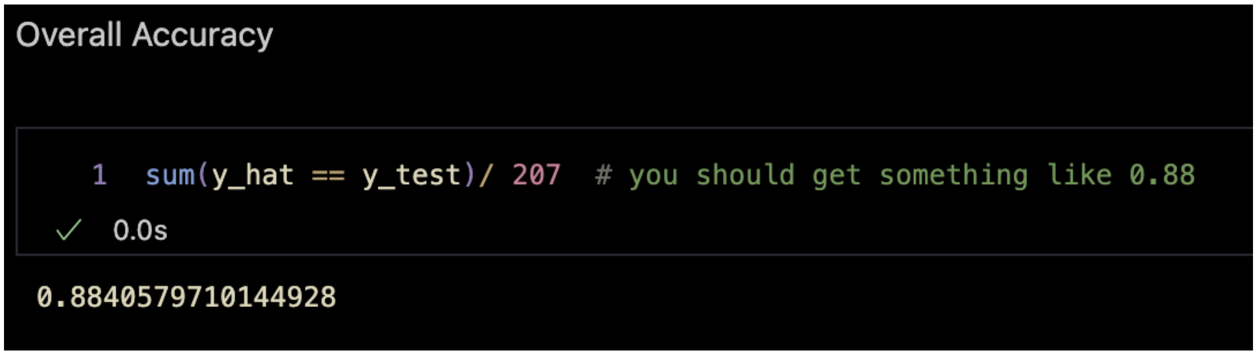
When evaluating against the sklearn Naive Bayes classifier, I saw that there was a discrepancy in accuracies and values. When investigating by looking at individual values, it was a silly bug of just forgetting to multiply by the priors. Adding in the corresponding dictionary lookup and addition of a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

## Naive Bayes Code

```
class Naive_Bayes():
    """

    Naive Bayes classifer

    Attributes:
        prior: P(Y)
        likelihood: P(X_j | Y)
    """

    def __init__(self):
        """
            Some initializations, if neccesary
        """

        self.model_name = 'Naive Bayes'


    def fit(self, X_train: np.array, y_train: np.array):

        """
```

```
            The fit function fits the Naive Bayes model based on the training data.
            Here, we assume that all the features are **discrete** features.

            X_train is a matrix or 2-D numpy array, represnting training instances.
            Each training instance is a feature vector.

            y_train contains the corresponding labels. There might be multiple (i.e., > 2)
classes.
        """

        """
            TODO: 1. Modify and add some codes to the following for-loop
                     to compute the correct prior distribution of all y labels.
                  2. Make sure they are normalized to a distribution.
        """

        # might as well store for making prediction easier. this tag index system is kind of
annoying and i would prefer to not have to parse that information out.
        self.y_labels = np.unique(y_train)
        # return self.y_labels

        self.y_counts = dict()
        for value in y_train:
            tag = f"Y = {value}"
            self.y_counts[tag] = self.y_counts.get(tag, 0) + 1
        self.prior = {k: v/len(y_train) for k, v in self.y_counts.items()}
        # return self.prior

        """
            TODO: 3. Modify and add some codes to the following for-loops
                     to compute the correct likelihood P(X_j | Y).
                  4. Make sure they are normalized to distributions.
        """

        self.likelihood = dict()
        for x, y in zip(X_train, y_train):
            x = np.array(x).reshape(-1) # taking the index of a matrix seems to force you to
get back a matrix no matter what.
            for j in range(len(x)):
                tag = f"X{j} = {x[j]} | Y = {y}"
                # self.likelihood[tag] = self.likelihood.get(tag, 0) + (1)
                self.likelihood[tag] = self.likelihood.get(tag, 0) + (1/self.y_counts[f"Y =
{y}"])

        return self.likelihood

        """
            TODO: 5. Think about whether we really need P(X_1 = x_1, X_2 = x_2, ..., X_d =
x_d)
                     in practice?
                  6. Does this really matter for the final classification results?
        """

        # no you don't. you can calculate that information on the fly when you need it. i
suppose that if you have a ton of queries, it might be slightly more efficient to calculate
those values beforehand.


    def ind_predict(self, x : list):

        """
```

```
            Predict the most likely class label of one test instance based on its feature
vector x.
        """

        """
            TODO: 7. Enumerate all possible class labels and compute the likelihood
                     based on the given feature vector x. Don't forget to incorporate
                     both the prior and likelihood.
                  8. Pick the label with the higest probability.
                  9. How to deal with very small probability values, especially
                     when the feature vector is of a high dimension. (Hint: log)
                  10. How to how to deal with unknown feature values?
        """

        # ok so we're going to be calculating log probs. log is a monotonically increasing
function so we can just compare the log probs.

        ret, max_logprob = None, -np.inf
        for y in self.y_labels:
            logprob = np.log(self.prior[f"Y = {y}"])
            for index, value in enumerate(x):
                tag = f"X{index} = {value} | Y = {y}"
                logprob += np.log(self.likelihood.get(tag, 0)) # no smoothing

            if logprob > max_logprob:
                max_logprob = logprob
                ret = y
        return ret


    def predict(self, X):

        print(X.shape)
        """
            X is a matrix or 2-D numpy array, represnting testing instances.
            Each testing instance is a feature vector.

            Return the predictions of all instances in a list.
        """

        """
            TODO: 11. Revise the following for-loop to call ind_predict to get predictions.
        """

        ret = []
        for x in X:
            ret.append(self.ind_predict(np.array(x).reshape(-1)))

        return ret
```


Overall Accuracy

```
sum(y_hat == y_test)/ 207  # you should get something like 0.88
✓ 0.0s

0.8840579710144928
```

## KMeans Writeup

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 each “K” dimension entry essentially is a collection of differences.

For each of these distance vectors, you want to calculate its length. This can be achieved pretty simply by 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. Afterward, 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]
```

## KMeans Code

```
class KMeans():
    def __init__(self, k = 3, num_iter = 1000):
        self.model_name = 'KMeans'
        self.k = k
        self.num_iter = num_iter
        self.centers = None
        self.RM = None

        self.centroids = None
        self.closest_centroid = None

    def train(self, X):
        np.random.seed(42)
        initial_centroid_indices = np.random.choice(X.shape[0], size=self.k, replace=False)
        self.centroids = X[initial_centroid_indices]

        for _ in range(self.num_iter):
            distance_vectors = (self.centroids.reshape(self.k, 1, -1) - X)
            distances_to_centroids = np.linalg.norm(distance_vectors, axis=2)
            self.closest_centroid = np.argmin(distances_to_centroids, axis=0)
            new_centroids = np.array([X[self.closest_centroid == centroid_index].mean(axis=0)
for centroid_index in range(self.k)])

            if np.all(new_centroids == self.centroids):
                break
            else:
                self.centroids = new_centroids

        # using an identity matrix to one hot encode
        self.RM = np.eye(self.k)[self.closest_centroid]

        return self # not sure why the the api wants it to be this way
```

## Gaussian Mixture Models Writeup

I made a couple of mistakes when writing out the multivariate gaussian. I think the issue was that I didn’t understand it very well, so it was difficult to contextualize around what they were doing, and so it turned into just copying the slides. However, after looking into it, it makes much more sense and it became pretty straightforward to write everything out.

My understanding of it:
e^x is exponential growth, add a negative sign e^-x to make it exponential decay. e^(-x^2) makes the exponential decay go in both directions. This gives the bell-curve shape. Since it has to be a probability distribution, you want the area under the curve to sum to 1. The integral of e^(-x^2) from -inf to inf ends up being sqrt(pi).
You can use e^(-(x-mu)^2) to control the location and sigma in e^(-(x/sigma)^2) to control width width. The larger the sigma, the wider the bell curve becomes, so you need to normalize by the factor area increases by, which is where the sqrt(sigma) comes from.
This gives you a univariate gaussian.
To go to a multivariate gaussian, you can first think about getting an “axis-aligned” (diagonal cov matrix) multivariate gaussian from a joint distribution of univariate gaussians. You can multiply them together, as you’re assuming they’re independent, and there’s no covariance between them. The determinant of the diagonal covariance matrix is a computational trick for essentially multiplying the variances of each univariate distribution together. The added to the power of n, is due to the multiplication of all the normalization factors from calculating the joint distribution. Since there are separate variances, for the e^(stuff) term, the “stuff” is going to be of the form ((x-mu)/sigma)^2+..., where each gaussian has a separate mu and sigma being added together. The (x-mu).T @ C.inv @ (x-mu) is pretty much also just another computational trick. Inverse of C, which is a diagonal matrix essentially makes the matrix multiplication a division by the variances, which corresponds to the ((x-mu)/**sigma)^2** portion. The (x-mu).T @ (x-mu) is just a linear algebra way of calculating a dot product, i.e. a sum of (x-mu)^2’s.
When you remove the independence condition, you technically can no longer just multiply the univariate gaussians together, but interestingly enough the formula is still the same. The covariance matrix is just no longer diagonal as there’s correlation between the univariate gaussians. This can be thought of as applying a rotation matrix to the old diagonal covariance matrix.

AFAIK, that’s how this expression is derived.

$$f_j(\boldsymbol{x}) = f\left(\boldsymbol{x} \middle| \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right) = \frac{1}{\sqrt{(2\pi)^d |\Sigma_j|}} e^{-\frac{(x-\boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j^{-1} (x-\boldsymbol{\mu}_j)}{2}}$$

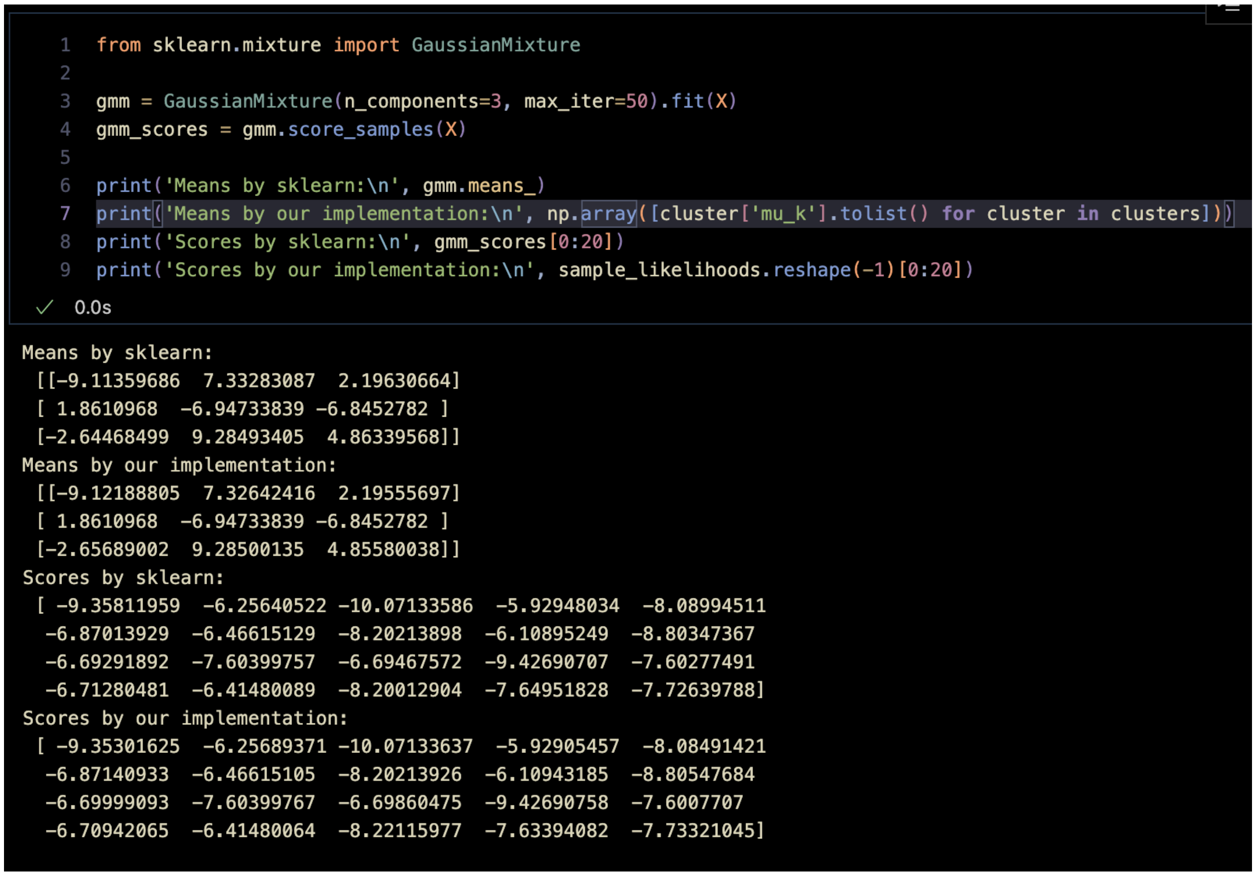
Contextualizing this around GMM, you're using parametric multivariate gaussians to assess the probability of a point belonging to a cluster. Unlike KMeans which is hard clustering, GMM is "soft" clustering, which means that points aren't really "assigned" to a cluster, but instead the more probable a point is, the greater impact it has during the "update"/maximization step for a given cluster. If it's very far away from a cluster, then it essentially has close to no effect on it.

These probabilities are used to essentially calculate a new weighted mean and covariance matrix for each gaussian. When calculating these probabilities for a specific cluster, we're conditioning on the given that points are actually from that specific cluster. With this, we need to multiply by the prior, which can be calculated by considering what proportion of the total "responsibility" probabilities a cluster has.

## Gaussian Mixture Models Code

```
def gaussian(X, mu, cov):
    """
        Fucntion to create mixtures using the Given matrix X, given covariance and given mu

        Return:
        transformed x.
    """
    # X should be matirx-like
    n = X.shape[1]
    diff = (X - mu).T
    return np.diagonal(1 / ((2 * np.pi) ** (n / 2) * np.linalg.det(cov) ** 0.5) * np.exp(-0.5
* np.dot(np.dot(diff.T, np.linalg.inv(cov)), diff))).reshape(-1, 1)


def expectation_step(X, clusters):
    """
        "E-Step" for the GM algorithm

        Parameter:
            X: Input feature matrix
            clusters: List of clusters
    """

    # totals = np.zeros((X.shape[0], 1), dtype=np.float64)
    numerators = []
    for cluster in clusters:
        w_k = cluster['w_k']
        mu_k = cluster['mu_k']
        cov_k = cluster['cov_k']
        numerator = gaussian(X, mu_k, cov_k) * w_k # likelihood * prior
        # totals = totals + numerator
        numerators.append(numerator)

        # cluster["likelihood * prior"] = numerator
        # cluster["total"] = totals.sum()

    numerators = np.hstack(numerators)
    denominators = numerators.sum(axis=1, keepdims=True)
    posteriors = numerators / denominators

    return posteriors
```

```
def maximization_step(X, clusters, posteriors):
    """
        "M-Step" for the GM algorithm

        Parameter:
            X: Input feature matrix
            clusters: List of clusters
    """

    for j, cluster in enumerate(clusters):
        cluster_posterior = posteriors[:, j]
        total_posterior = cluster_posterior.sum()
        w_k = total_posterior / X.shape[0]
        mu_k = np.dot(cluster_posterior, X) / total_posterior
        diff = X - mu_k
        cov_k = np.dot(cluster_posterior * diff.T, diff) / total_posterior
        cluster["w_k"] = w_k
        cluster["mu_k"] = mu_k
        cluster["cov_k"] = cov_k

def get_likelihood(X, clusters):
    sample_likelihoods = np.zeros((X.shape[0], 1), dtype=np.float64)
    for cluster in clusters:
        sample_likelihoods = sample_likelihoods + gaussian(X, cluster["mu_k"],
cluster["cov_k"]) * cluster["w_k"]
    return np.log(sample_likelihoods).sum(), np.log(sample_likelihoods)

def train_gmm(X, n_clusters, n_epochs):
    clusters = initialize_clusters(X, n_clusters)
    print(clusters)
    likelihoods = np.zeros((n_epochs, ))
    scores = np.zeros((X.shape[0], n_clusters))
    sample_likelihoods = None

    for i in range(n_epochs):

        posteriors = expectation_step(X, clusters)
        maximization_step(X, clusters, posteriors)

        likelihood, sample_likelihoods = get_likelihood(X, clusters)
        likelihoods[i] = likelihood

    for i, cluster in enumerate(clusters):
        scores[:, i] = np.log(cluster['w_k']).reshape(-1)

    return clusters, likelihoods, scores, sample_likelihoods
```



```python
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=3, max_iter=50).fit(X)
gmm_scores = gmm.score_samples(X)

print('Means by sklearn:\n', gmm.means_)
print('Means by our implementation:\n', np.array([cluster['mu_k'].tolist() for cluster in clusters]))
print('Scores by sklearn:\n', gmm_scores[0:20])
print('Scores by our implementation:\n', sample_likelihoods.reshape(-1)[0:20])
```

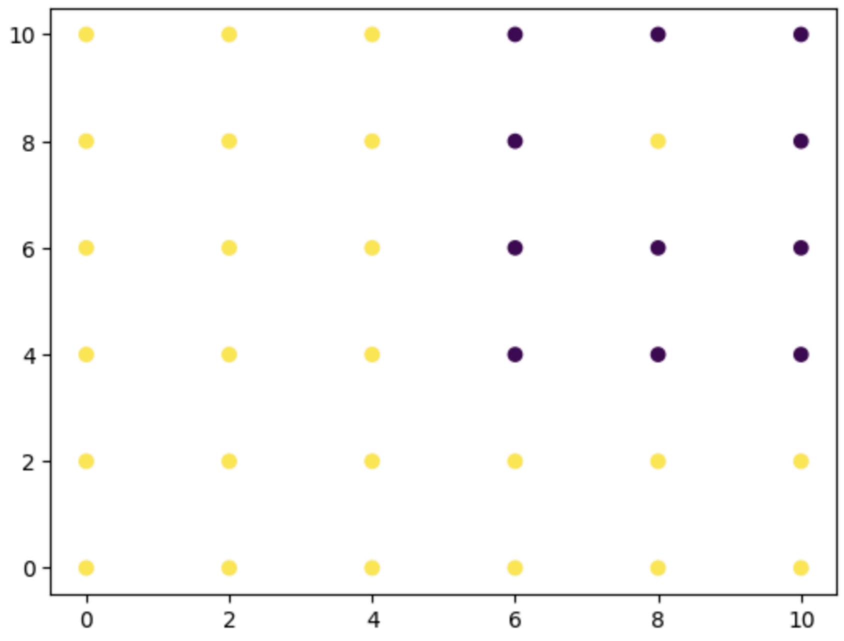
✓ 0.0s

```
Means by sklearn:
 [[-9.11359686  7.33283087  2.19630664]
 [ 1.8610968  -6.94733839 -6.8452782 ]
 [-2.64468499  9.28493405  4.86339568]]
Means by our implementation:
 [[-9.12188805  7.32642416  2.19555697]
 [ 1.8610968  -6.94733839 -6.8452782 ]
 [-2.65689002  9.28500135  4.85580038]]
Scores by sklearn:
 [ -9.35811959  -6.25640522 -10.07133586  -5.92948034  -8.08994511
  -6.87013929  -6.46615129  -8.20213898  -6.10895249  -8.80347367
  -6.69291892  -7.60399757  -6.69467572  -9.42690707  -7.60277491
  -6.71280481  -6.41480089  -8.20012904  -7.64951828  -7.72639788]
Scores by our implementation:
 [ -9.35301625  -6.25689371 -10.07133637  -5.92905457  -8.08491421
  -6.87140933  -6.46615105  -8.20213926  -6.10943185  -8.80547684
  -6.69999093  -7.60399767  -6.69860475  -9.42690758  -7.6007707
  -6.70942065  -6.41480064  -8.22115977  -7.63394082  -7.73321045]
```

## Decision Tree + Random Forests Writeup

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 unique value for each feature, the threshold ends up being undefined. This problem really only manifested when implementing random forests as subsets of features are far more likely to have label conflicts, 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 in 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.

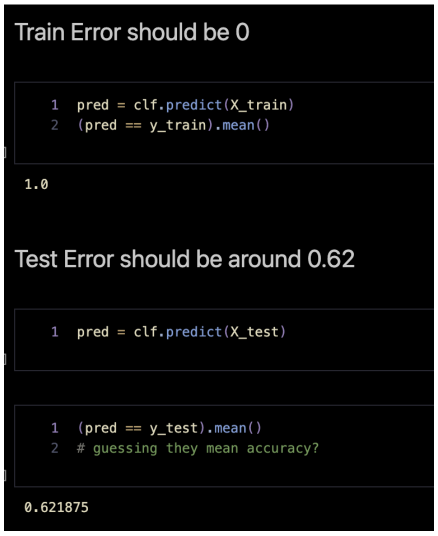
Something else that's interesting is just how much faster it is to train weak learners. When taking a subset of features, that already cut down on time drastically, and when taking a subset of the data, the entire training process for the 100 trees took 3 seconds.

## Decision Tree + Random Forests Code

```
class DecisionTree():

    def __init__(self, max_depth = 1000, min_samples_split = 2, n_features = None, n_split = 
None):
        self.root = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        # for random forest
        # self.n_features = n_features
        # self.n_split = n_split


    class Node():
        def __init__(self, feature_index_col: int, threshold: float, left:
"DecisionTree.Node", right: "DecisionTree.Node", majority_class):
            self.feature_index_col = feature_index_col
            self.threshold = threshold
            self.left = left
            self.right = right
            self.majority_class = majority_class # if a node is cut off due to max depth or
size constraint, this becomes useful. also works for purity.

        def predict(self, x: np.ndarray):
            if self.feature_index_col is None or self.threshold is None: # leaf node
                return self.majority_class
            else:
                # print(self.feature_index_col, self.threshold, x[self.feature_index_col])
                if x[self.feature_index_col] <= self.threshold:
                    return self.left.predict(x)
                else:
                    return self.right.predict(x)


    def entropy(self, y: np.ndarray):
        values, counts = np.unique(y, return_counts=True)
        probs = counts / counts.sum()
        entropy = - (probs * np.log2(probs)).sum()
        return entropy


    def information_gain(self, X_col: np.ndarray, y: np.ndarray, threshold: float) -> float:
        left_mask = X_col <= threshold # by convention left node meets condition
        right_mask = ~left_mask
        left_y = y[left_mask]
        right_y = y[right_mask]
        left_entropy = self.entropy(left_y)
        right_entropy = self.entropy(right_y)
        left_prob = left_y.shape[0] / y.shape[0]
        right_prob = right_y.shape[0] / y.shape[0]
        weighted_average_entropy = left_prob * left_entropy + right_prob * right_entropy

        total_entropy = self.entropy(y)
        return total_entropy - weighted_average_entropy


    def find_all_feature_split_rules(self, X: np.ndarray) -> List[np.ndarray]:
        all_feature_split_rules = []
        for feature_col in X.T:
            unique_values = np.unique(feature_col) # also sorts values
            unique_values_midpoints = (unique_values[:-1] + unique_values[1:]) / 2
            all_feature_split_rules.append(unique_values_midpoints)
        return all_feature_split_rules


    def find_best_split(self, X: np.ndarray, y: np.ndarray, all_feature_split_rules:
List[np.ndarray]) -> Tuple[int, float]:
        features_best_splits = [] # for each feature track what was the best split
        for index, feature_split_rules in enumerate(all_feature_split_rules):
            X_col = X[:, index]
            feature_best_split = (index, None, -1e9) # index, threshold, infogain
            for threshold in feature_split_rules:
                infogain = self.information_gain(X_col, y, threshold)
                if infogain > feature_best_split[2]:
                    feature_best_split = (index, threshold, infogain)
            features_best_splits.append(feature_best_split)

        # find the best split across features
        col_index, threshold, infogain = max(features_best_splits, key=lambda x: x[2])

        # if threshold is None: # happens when dataset has same X, but different Y.
        #     print('threshold is None')
        #     print(all_feature_split_rules)
        #     print(X, y)

        return col_index, threshold


    def build_tree(self, X: np.ndarray, y: np.ndarray, current_depth: int = 1):

        values, counts = np.unique(y, return_counts=True)
        majority_class = values[np.argmax(counts)]
        # base cases: max depth exceeded, node too small, or pure node
        if current_depth > self.max_depth or X.shape[0] < self.min_samples_split or
len(np.unique(y)) == 1:
            return DecisionTree.Node(None, None, None, None, majority_class)

        index_col, threshold = self.find_best_split(X, y,
self.find_all_feature_split_rules(X))

        if threshold is None: # happens when X is all same value with different y
            return DecisionTree.Node(None, None, None, None, majority_class)

        # print(index_col, threshold)
        X_left = X[X[:, index_col] <= threshold]
        y_left = y[X[:, index_col] <= threshold]
        X_right = X[X[:, index_col] > threshold]
        y_right = y[X[:, index_col] > threshold]

        # last node created is the root
        left_node = self.build_tree(X_left, y_left, current_depth + 1)
        right_node = self.build_tree(X_right, y_right, current_depth + 1)

        return DecisionTree.Node(index_col, threshold, left_node, right_node, majority_class)


    def fit(self, X: np.ndarray, y: np.ndarray):
        self.root = self.build_tree(X, y)
        return self


    def predict(self, dataset: np.ndarray):
        result = []
        for datapoint in dataset:
            result.append(self.root.predict(datapoint))
        return np.array(result)



from scipy.stats import mode
```

```
class RandomForest():
    def __init__(self, n_trees = 10, n_features = 'sqrt', n_split = 'sqrt', max_depth = 1000,
size_allowed = 1):
        self.n_trees = n_trees
        self.trees = []
        self.n_features = n_features
        self.n_split = n_split
        self.max_depth = max_depth
        self.size_allowed = size_allowed


    def fit(self, X: np.ndarray, y: np.ndarray):
        np.random.seed(42)
        for idx in range(self.n_trees):
            X_subset = X
            y_subset = y

            if self.n_features == 'sqrt':
                feature_sample_indices = np.random.choice(X.shape[1],
int(np.sqrt(X.shape[1])), replace = False)
                X_subset = X[:, feature_sample_indices]
                y_subset = y

            if self.n_split == 'sqrt':
                data_sample_indices = np.random.choice(X.shape[0], int(np.sqrt(X.shape[0])),
replace = False)
                X_subset = X_subset[data_sample_indices]
                y_subset = y_subset[data_sample_indices]

            print(idx, X_subset.shape, y_subset.shape)

            temp_clf = DecisionTree(max_depth=self.max_depth,
min_samples_split=self.size_allowed)
            temp_clf.fit(X_subset, y_subset)
            self.trees.append((temp_clf, feature_sample_indices))

        return self

    def predict(self, dataset: np.ndarray):
        result = []
        for datapoint in dataset:
            tree_predictions = []
            for tree, feature_sample_indices in self.trees:
                datapoint_subset = datapoint[feature_sample_indices]
                tree_predictions.append(tree.predict([datapoint_subset])[0])
            mode_prediction, _ = mode(tree_predictions)
            result.append(mode_prediction)
        return result
```

DECISION TREE



Train Error should be 0

```
pred = clf.predict(X_train)
(pred == y_train).mean()
```

```
1.0
```

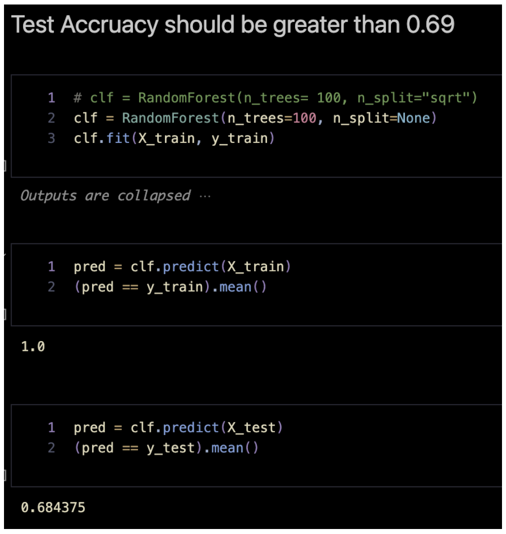
Test Error should be around 0.62

```
pred = clf.predict(X_test)
```

```
(pred == y_test).mean()
# guessing they mean accuracy?
```

```
0.621875
```

RANDOM FORESTS



Test Accruacy should be greater than 0.69

```
# clf = RandomForest(n_trees= 100, n_split="sqrt")
clf = RandomForest(n_trees=100, n_split=None)
clf.fit(X_train, y_train)
```

Outputs are collapsed ...

```
pred = clf.predict(X_train)
(pred == y_train).mean()
```

```
1.0
```

```
pred = clf.predict(X_test)
(pred == y_test).mean()
```

```
0.684375
```