Creating a random forest classifier from our prexiesting code is pretty simple, we just need to select a random subset of the features at every split.

We can do that with this code snippet before we split our trees

```
numFeatures = int(np.round_(np.sqrt(X.shape[1])))
X = X.sample(n = numFeatures, axis="columns")
```

beyond that, we just need to use the same general bagging strategy we did on lab 5. That is, create many different trees with resampling and then taking the mode of their predictions.

But first we need to set up the project and import the data

```
from pathlib import Path
home = str(Path.home()) # all other paths are relative to this path. change to something el

%load_ext autoreload
%autoreload 2

# make sure your run the cell above before running this

import Lab4_helper
import Lab5_helper
import helper
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

We're going to use the diabetes dataset from lab 4

loading the data in and grabbing the features we want

```
import pandas as pd
import numpy as np

diabetes_df = pd.read_csv(
    f"./diabetes_indicators.csv"
)
features = ['Sex','Age','Education','Income','Fruits','Veggies','Smoker', "HighChol", "BMI"
dia_X = diabetes_df.loc[:,features][:1000]
dia_X = dia_X.dropna()
dia_t = diabetes_df.loc[dia_X.index,'Diabetes_012']
```

Displaying our input data

```
dia_X
```

	Sex	Age	Education	Income	Fruits	Veggies	Smoker	HighChol	BMI
0	0.0	9.0	4.0	3.0	0.0	1.0	1.0	1.0	40.0
1	0.0	7.0	6.0	1.0	0.0	0.0	1.0	0.0	25.0
2	0.0	9.0	4.0	8.0	1.0	0.0	0.0	1.0	28.0
3	0.0	11.0	3.0	6.0	1.0	1.0	0.0	0.0	27.0
4	0.0	11.0	5.0	4.0	1.0	1.0	0.0	1.0	24.0
...	...	...	...	...	...	...	...	...	...
995	0.0	2.0	6.0	8.0	1.0	0.0	0.0	0.0	31.0
996	0.0	10.0	5.0	8.0	0.0	1.0	0.0	0.0	21.0
997	1.0	7.0	4.0	1.0	0.0	0.0	0.0	1.0	31.0
998	0.0	5.0	4.0	8.0	1.0	1.0	0.0	0.0	37.0
999	1.0	11.0	4.0	7.0	0.0	1.0	1.0	0.0	28.0

1000 rows × 9 columns

Displaying our target

```
dia_t
```

```
0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
      ...
995    0.0
996    0.0
997    0.0
998    0.0
999    0.0
Name: Diabetes_012, Length: 1000, dtype: float64
```

Creating Random Forest Algorithm

Below are the f1 scores for various algorithms. I have this helper function, run_[algorithm name] that splits the given dataset into a train test split, runs the algorithm on the data, and averages a

preformance metric - f1 in classification and RMSE in regresison - overal several trials.

```
classification_results = {}
helper.run_myrf(X=dia_X, t= dia_t, results=classification_results, type="classifier",ntria
helper.run_bagging(X=dia_X, t= dia_t, results=classification_results, type="classifier",ntr
helper.run_skrf(X=dia_X, t= dia_t, results=classification_results, type="classifier",ntria
helper.run_boost(X=dia_X, t= dia_t, results=classification_results, type="classifier",ntri
classification_results
```

```
{'myrf_classifier': 0.8569341877380964,
 'bagging_classifier': 0.7700010085851262,
 'skrf_classifier': 0.7977920649619576,
 'boost_classifier': 0.8203250785647629}
```

This is really cool! Our random forest preformed better than sklearn's random forest, as well as the bagging and boosting algorithms

Regression

The only major changes we need to make a decision regression are in how we split our trees, our base case in regression, and how we make our predictions.

Splitting Trees

In the classification case, our goal was to minimize entropy. In the regression case, our goal is to minimize variance. In the code, this was simply changing their criterion in the regressive case to the variance.

```
def gain(y, x, type="classifier"):
    g = 0
    possibleValues = x.unique()
    weightedCriterions = []

    for value in possibleValues:
        #splitting the data by values
        xAtVal = x.loc[x == value]
        yAtVal = y.loc[x == value]
        #calculating our gain
        if type=="classifier":
            unweightedCriterion = entropy(yAtVal)
        elif type=="regressor":
```

```
            unweightedCriterion = yAtVal.var()
        #weighting it
        weight = xAtVal.size / x.size
        weightedCriterions.append(weight * unweightedCriterion)

    #seeing how much we improved
    g = sum(weightedCriterions)
    if type == "classifier":
        origCriterion = entropy(y)
    if type == "regressor":
        origCriterion = y.var()

    return origCriterion - g
```

Regression Base Case and Predictions

When we're constructing a tree and we reach a point to make a node, such as when we run out of feature or when we go under our minimum samples need to split, we needed to change the code to be the average of the remaining targets instead of the mode. Similarly when we're predicting with all of our classifiers, we need to take the prediction on average instead of the mode of the predictions for each sample

Testing

I'm going to use a dataset that contains the chemical properties of various red wines and their numerical rating of "quality" by experts. I'm going to try to do a regression between those chemical properties and the quality. I obtained the dataset from https://archive.ics.uci.edu/ml/datasets/wine+quality

Loading the dataset in and preprocessing displaying the raw dataset:

```
df = pd.read_csv("./winequality-red.csv", delimiter=";")
df.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	su
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	

1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16

displaying the preprocessed X:

```
wine_X = df.drop(columns=["quality"])
wine_t = df["quality"]
wine_X = wine_X.apply(lambda col: (col - col.mean()) / col.std())
wine_X.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	
0	-0.528194	0.961576	-1.391037	-0.453077	-0.243630	-0.466047	-0.379014	0.558100	
1	-0.298454	1.966827	-1.391037	0.043403	0.223805	0.872365	0.624168	0.028252	-(
2	-0.298454	1.296660	-1.185699	-0.169374	0.096323	-0.083643	0.228975	0.134222	-(
3	1.654339	-1.384011	1.483689	-0.453077	-0.264878	0.107558	0.411372	0.664069	-(

Applying random forest and calculating the RMSE for different algorithms

```
regression_results = {}
ntrials = 5
ntrees = 25
helper.run_myrf(X=wine_X, t=wine_t, results=regression_results, type="regressor", ntrials=r
helper.run_boost(X=wine_X, t=wine_t, results=regression_results, type="regressor", ntrials=
helper.run_skrf(X=wine_X, t=wine_t, results=regression_results, type="regressor", ntrials=r
helper.run_bagging(X=wine_X, t=wine_t, results=regression_results, type="regressor", ntria
regression_results
```

```
{'myrf_regressor': 0.7060103298790785,
 'boost_regressor': 0.7291675413269797,
 'skrf_regressor': 0.5934734308985723,
 'bagging_regressor': 0.5986632100773788}
```

It is disapointing that our implementation of random forest performed worse than sklearn's, but since it preformed better than boosting, I don't think there's any errors with it -- it's clearly doing something right. If I had to guess it's because we're selecting the split in continuous data differently than sklearn does.

Feature Importance

We're going to measure the feature importance by seeing how the feature, on average across all the trees, increases the purity in the resulting dataset. This increase in purity is mathematically defined by a decrease in the gini inpurity

Gini Gain at a Node

Gini Impurity

The Gini Impurity score at node n is defined as the probability of picking two different classes if you picked randomly from all the samples at node n. It ranges from 0 to 1 and the higher it is, the more impure the data and the lower it is, the more pure the data. It's mathematically described as

$$g(n) = 1 - \sum_{i=1}^{j} (P_i)^2$$

where:

n is the current node (split in the dataset based on the value of some feature)

j is the number of distinct classes,

P_i is what portion the i-th class is of all the classes (the probaiblity of selecting the i-th class at random)

We can implement it with this function

```
def gini(x):
    counts = x.value_counts()
    fracs = counts / len(x)
    ans = 1 - (fracs ** 2).sum()
    return ans
```

Testing with a more impure dataset A and a more pure dataset B

```
A = pd.Series([1,2, 2, 3, 4, 4, 1, 5, 5])
B = pd.Series([0, 1, 1, 1, 1, 0, 1, 1, 0])

print("Impurity  of A is", gini(A))
print("Impurity of B is", gini(B))
```

```
Impurity  of A is 0.7901234567901234
Impurity of B is 0.4444444444444444
```

Looks good!

Mean Gini Impurity Gain

We're interested in average decrease in gini impurity at the current node -- aka average increase in purity. That is, the difference between the gini impurity at n and the weighted sum of the gini impurity of its two children. The weight of each child is what proportion of the samples at n are included in the child. Mathematically this is

$$gg(n) = g(n) - \sum_{i=1}^{c} \frac{s_i}{s_n} g(c_i)$$

Where:

n is the current node

$gg(n)$ is the mean gini impurity decrease at n

c is the number of children

s_i is the number of samples included in the i-th child

s_n is the number of samples at n

$g(c_i)$ is the gini impurity of the i-th child node

we can implement it with this function

```
def split_data(x, t, tree):
    #helper function to split the data by the feature given in the tree
    feature_name, threshold = list(tree.keys())[0].split("<")
    threshold = float(threshold)

    #Split the data
    x_l = x[x[feature_name] < threshold]
    x_r = x[x[feature_name] >= threshold]
    t_l = t[x[feature_name] < threshold]
    t_r = t[x[feature_name] >= threshold]

    return x_l, x_r, t_l, t_r

def gid(x, t, tree):
    #split the data by the metric in the tree. The node n is the head node of the tree.
    #Grab the metric in question
    x_l, x_r, t_l, t_r  = split_data(x, t, tree)
```

```
    #calculate gid
    p_l = len(x_l) / len(x)
    p_r = len(x_r) / len(x)

    #calculating ginis
    gini_n = gini(t)
    gini_l = gini(t_l)
    gini_r = gini(t_r)

    ans = gini_n - (p_l * gini_l + p_r * gini_r)
    return ans
```

Let's test the function on a table that can be neatly divided in two

```
data = {"isHot": [0, 0, 0, 1, 1, 1], "shouldTouch": [1, 1, 1, 0, 0, 0,]}
test_df = pd.DataFrame(data)
tree = {
    "isHot<0.5": {"False": 1,
                  "True": 0}
}
test_x = test_df.drop(columns=["shouldTouch"])
test_t = test_df.drop(columns=["isHot"])
gid(test_x, test_t, tree)
```

```
0.5
```

This makes sense! As the impurity at the start is 0.5, and the purity of each of the split tables is 0, so the weighted decrease in impurity is 0.5

Testing on Lab 4 Code

```
gid(dia_X_test, dia_t_test, c45_tree)
```

```
0.015036706311076897
```

Does that seem reasonable? I have no idea!

Feature Importance from Gini Impurity Decrease

For every tree, for every node, we find the Gini impurity decrease and then weight it by the proportion of the number of samples at the node to the number of samples in total. We take that value and average it over each feature. This is our feature importance metric.

We can implement this like so:

```
#function for recursing through tree and calculating gid at every node:
def gini_importance_from_tree(x, t, tree, n_samples, feature_results):
    #defining base cases
    if len(x) == 0:
        return
    if not isinstance(tree, dict):
        return
    feature_name, threshold = list(tree.keys())[0].split("<")
    #getting the purity increase and weighting it
    gid_i  = gid(x, t, tree)
    importance = gid_i * (len(x) / n_samples)
    #adding it to the  results
    if feature_name in feature_results:
        feature_results[feature_name].append(importance)
    else:
        feature_results[feature_name] = list([importance])

    #recursing
    subtree = list(tree.values())[0]
    for expected_value, next_tree in subtree.items():
        sub_x = x[(x[feature_name] < float(threshold)) == (expected_value == "True")]
        sub_t = t[(x[feature_name] < float(threshold)) == (expected_value == "True")]
        gini_importance_from_tree(sub_x, sub_t, next_tree, n_samples, feature_results)
```

Running the feature importance metric on the diabetes dataset and displaying the results

```
#importing
from sklearn.model_selection import train_test_split
#setting up
ntrials = 5
ntrees = 25
default = 0
feature_results = {}
#running tests
for trial in range(ntrials):
    X_train, X_test, t_train, t_test = train_test_split(dia_X, dia_t, test_size=0.3,random_
    trees = helper.make_rf_trees(X_train,t_train,ntrees=ntrees)
    for tree in trees:
        gini_importance_from_tree(X_train, t_train, tree, len(X_train), feature_results)

data = {key: sum(value) / len(value) for key, value in feature_results.items()}
pd.Series(data).sort_values(ascending=False)
```

```
Age          0.010144
BMI          0.007309
Income       0.005576
HighChol     0.004016
Education    0.001489
Veggies      0.001478
Fruits       0.000559
Sex          0.000383
Smoker       0.000283
dtype: float64
```

This makes sense! This metrics intuitively seem important for estimating BMI

Calculating the feature importances for sklearn's implementation of random forest usign their built-in feature importances

```
from sklearn.ensemble import RandomForestClassifier
ntrials = 5
ntrees = 25
default = 0
sk_feature_results = {}
for trial in range(ntrials):
    X_train, X_test, t_train, t_test = train_test_split(dia_X, dia_t, test_size=0.3,random_
    classifier = RandomForestClassifier(n_estimators=ntrees, random_state=trial, min_sample
    for i in range(len(classifier.feature_names_in_)):
        feature = classifier.feature_names_in_[i]
        importance = classifier.feature_importances_[i]
        if feature not in sk_feature_results:
            sk_feature_results[feature] = [importance]
        else:
            sk_feature_results[feature].append(importance)

pd.DataFrame(sk_feature_results).T.mean(axis=1).sort_values(ascending=False)
```

```
BMI          0.274652
Age          0.207724
Income       0.169856
Education    0.105071
Fruits       0.054698
HighChol     0.049250
Sex          0.046786
Veggies      0.046446
Smoker       0.045517
dtype: float64
```

What's more is that it has very similar results to the built-in implementation! The top 3 are all the same and education is in the top 5 of both. I'm willing to chalk the remaining differences up to how we split continuous data again. My model and sklearn's model did have different F1 score,

so the models are different and it shouldn't be too surprising that the importances then differ also.