SENG 474: Assignment 1 Report

Nathan Woloshyn

Part 1: Processing the data

The first step is to load the data into a pandas dataframe. The data is stored in a csv file, so we can use the read_csv function to load it into a dataframe. After this, we split the data into a training set and a test set. What fraction of the data is used for training and what fraction is used for testing is a hyperparameter that can be tuned. We choose a 75/25 split as our default, but will also analyze the results of other splits. We also specify a random seed for the sampling, so that we can reproduce the results of our experiments.

```
import pandas as pd

'''Reads the data from the csv file and returns a pandas dataframe.'''
def read_data():
    df = pd.read_csv('./cleaned_adult.csv')

    return df



'''Partitions the data into train and test sets, using a taking what
percentage of the data train / test on as input. Returns the train and test'''
def partition_data(df, train_size=0.75, random_state=99):
    # Split the data into train and test sets. (0.75, 0.25) split.
    train_df = df.sample(train_size, random_state)
    test_df = df.drop(train_df.index)

    return train_df, test_df
```

Part 2: Decision Trees

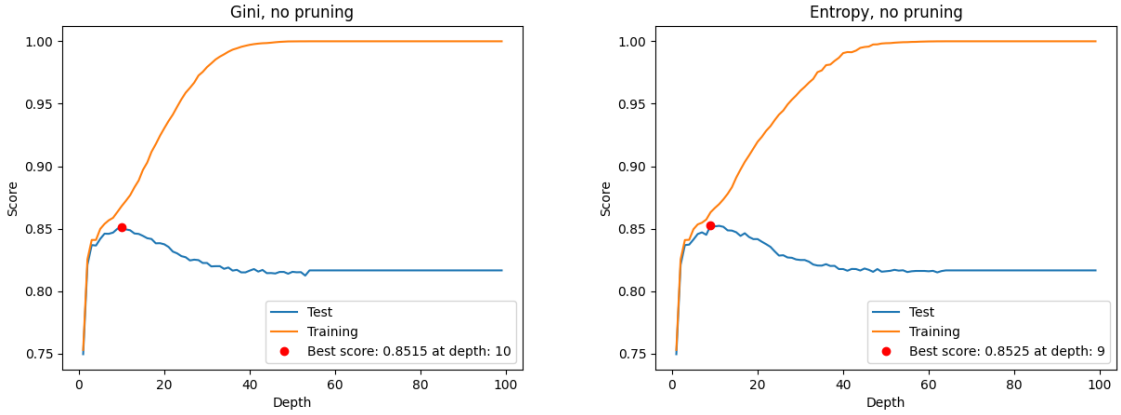
Part 2.1: No Pruning

In our first experiment we use the sklearn implementation of a decision tree classifier. We use the default parameters, which means that the tree is not pruned. We use test both entropy and Gini impurity as our criterion for splitting the tree. Using the code below we test every depth of tree from 1 to 100 using these two criteria. We use our default choice of 75/25 train/test split, giving all trees the same train/test split.

```
train, test = read_data.partition_data(read_data.read_data())


'''Test various depths, with entropy as the criterion,
store and plot the scores on both training and test sets using matplotlib'''
def test_entropy_depths():
    best_depth = 0
    best_score = 0
    test_scores = []
    training_scores = []
    for i in range(1, 100):
        eD = DecisionTreeClassifier(random_state=0, max_depth=i,
         criterion="entropy").fit(train.drop(columns=['income']),
         train['income'])


        test_scores.append(eD.score(test.drop(columns=['income']), test['income']))
        training_scores.append(eD.score(train.drop(columns=['income']), train['income']))
        if eD.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = eD.score(test.drop(columns=['income']), test['income'])
            best_depth = i
    plt.plot(range(1, 100), test_scores, label='Test')
    plt.plot(range(1, 100), training_scores, label='Training')
    plt.plot(best_depth, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at depth: ' + str(best_depth))
    plt.legend(loc="lower right")
    plt.xlabel('Depth')
    plt.ylabel('Score')
    plt.title('Entropy, no pruning')
    #plt.show()
    plt.savefig('entropy_no_pruning_scores_varying_depth.png')
    plt.clf()
```

```
'''Test various depths, with gini as the criterion,
store and plot the scores on both the training and test sets using matplotlib'''
def test_gini_depths():
    best_depth = 0
    best_score = 0
    test_scores = []
    training_scores = []
    for i in range(1, 100):
        gD = DecisionTreeClassifier(random_state=0, max_depth=i,
        criterion="gini").fit(train.drop(columns=['income']), train['income'])

        test_scores.append(gD.score(test.drop(columns=['income']), test['income']))
        training_scores.append(gD.score(train.drop(columns=['income']), train['income']))
        if gD.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = gD.score(test.drop(columns=['income']), test['income'])
            best_depth = i
    plt.plot(range(1, 100), test_scores, label='Test')
    plt.plot(range(1, 100), training_scores, label='Training')
    plt.plot(best_depth, best_score, 'ro', label='Best score: '
         + "{:.4f}".format(best_score) + ' at depth: ' + str(best_depth))
    plt.legend(loc="lower right")
    plt.xlabel('Depth')
    plt.ylabel('Score')
    plt.title('Gini, no pruning')
    #plt.show()
    plt.savefig('gini_no_pruning_scores_varying_depth.png')
    plt.clf()

test_entropy_depths()
test_gini_depths()
```

Running the code above gives us the following results:



We can see that at low depth accuracy quickly grows as we allow a tree to make more splits, but around depth = 10 the models performance quickly declines. This is likely due to overfitting, as we can see that as depth increases the training accuracy continues to increase, but the test accuracy starts to decrease. This is a sign that the model is overfitting to the training data, and is not generalizing well to the test data. This is a common problem with decision trees, and why we will be using pruning in our next experiment. Overall, the two choices of criterion, entropy and Gini impurity, seem to perform similarly, with the best score being around 0.85 for both, and both having a best depth of around 10.

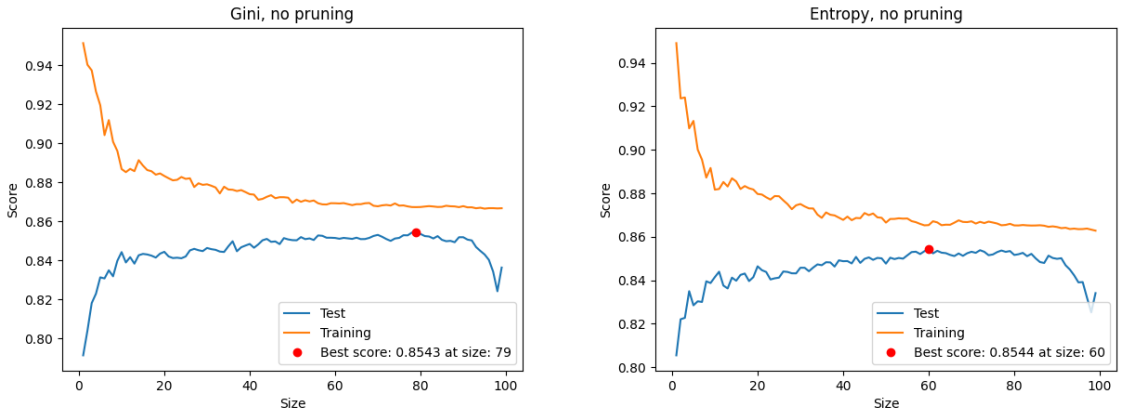
Next we test the effect of varying what percentage of the data we allocate to training and test, using the same depth of 10 for the tree. We use the code below to test the effect of varying the train/test split from 0% to 100% in 1% increments.

```
'''Test various training set sizes, with entropy as the criterion, using depth = 10'''

def test_entropy_sizes():
    best_size = 0
    best_score = 0
    test_scores = []
    training_scores = []
    for i in range(1, 100):
        train, test = read_data.partition_data(read_data.read_data(), train_size=i/100)
        eS = DecisionTreeClassifier(random_state=0, max_depth=10,
            criterion="entropy").fit(train.drop(columns=['income']), train['income'])
        test_scores.append(eS.score(test.drop(columns=['income']), test['income']))
        training_scores.append(eS.score(train.drop(columns=['income']), train['income']))
        if eS.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = eS.score(test.drop(columns=['income']), test['income'])
            best_size = i
```

```
    plt.plot(range(1, 100), test_scores, label='Test')
    plt.plot(range(1, 100), training_scores, label='Training')
    plt.plot(best_size, best_score, 'ro', label='Best score: '
         + "{:.4f}".format(best_score) + ' at size: ' + str(best_size))
    plt.legend(loc="lower right")
    plt.xlabel('Size')
    plt.ylabel('Score')
    plt.title('Entropy, no pruning')
    #plt.show()
    plt.savefig('entropy_no_pruning_scores_varying_size.png')
    plt.clf()

'''Test various training set sizes, with gini as the criterion, using depth = 10'''

def test_gini_sizes():
    best_size = 0
    best_score = 0
    test_scores = []
    training_scores = []
    for i in range(1, 100):
        train, test = read_data.partition_data(read_data.read_data(), train_size=i/100)
        gS = DecisionTreeClassifier(random_state=0, max_depth=10,
            criterion="gini").fit(train.drop(columns=['income']), train['income'])
        test_scores.append(gS.score(test.drop(columns=['income']), test['income']))
        training_scores.append(gS.score(train.drop(columns=['income']), train['income']))
        if gS.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = gS.score(test.drop(columns=['income']), test['income'])
            best_size = i
    plt.plot(range(1, 100), test_scores, label='Test')
    plt.plot(range(1, 100), training_scores, label='Training')
    plt.plot(best_size, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at size: ' + str(best_size))
    plt.legend(loc="lower right")
    plt.xlabel('Size')
    plt.ylabel('Score')
    plt.title('Gini, no pruning')
    #plt.show()
    plt.savefig('gini_no_pruning_scores_varying_size.png')
    plt.clf()

test_entropy_sizes()
```

```
test_gini_sizes()
```

Running the code above gives us the following results:



We can see that the accuracy of of the model is sensitive to how we partition our data. As we increase the size of the training set, the accuracy of the model increases, but at some point the accuracy the returns become diminishing, and eventually the test set is so small that the model is not able to generalize well. However, unlike the last experiment, there seems to be meaningfully different behavior from our two criteria. We were surprised by this, given the first experiment. So we ran several trials of this experiment, varying the random seed used to partition the data, and the results were consistent. The Gini criterion sees it's best performance when around 80% of the data is allocated to training, while the entropy criterion consistently reaches peak performance when around 60% of the data is allocated to training.

Part 2.2: Pruning

Now we will introduce pruning to our decision trees. We will be performing similar experiments with varying depth and varying training set size, but this time we will be using the pruning parameter to control the depth of the tree. We will be using the same methodology as before, but with the addition of the pruning parameter. Similar to our previous experiment, we will test the effect varying the pruning parameter

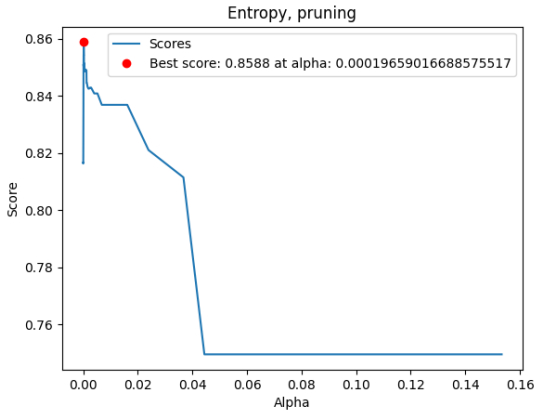
$$\alpha$$

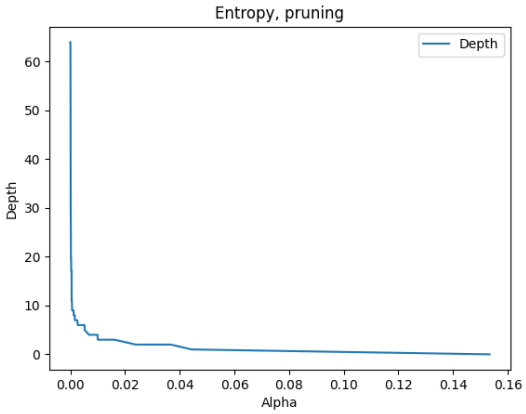
through all possible values and observing the effect on the accuracy of the model. We will be using the entropy criterion for this experiment, and the same training set size of 75% for the data.

```
train, test = read_data.partition_data(read_data.read_data())


#create dt
dt = DecisionTreeClassifier(random_state=0, criterion="entropy").fit(train.drop(columns=['

#prune dt
path = dt.cost_complexity_pruning_path(train.drop(columns=['income']), train['income'])
ccp_alphas, impurities = path.ccp_alphas, path.impurities

depths = []
scores = []
i = 0
best_alpha = ccp_alphas[0]
best_score = 0
for ccp_alpha in ccp_alphas:
    if i % 100 == 0:
        print(i)
    dt_pruned = DecisionTreeClassifier(random_state=0, criterion="entropy", ccp_alpha=ccp_
    if dt_pruned.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = dt_pruned.score(test.drop(columns=['income']), test['income'])
            best_alpha = ccp_alpha
    depths.append(dt_pruned.get_depth())
    scores.append(dt_pruned.score(test.drop(columns=['income']), test['income']))
    i+=1

'''Plot the scores on both the training and test sets using matplotlib from varying alpha

def plot_alpha_scores():
    plt.plot(ccp_alphas, scores, label='Scores')
    plt.plot(best_alpha, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at alpha: ' + str(best_alpha))
    plt.legend(loc="upper right")
    plt.xlabel('Alpha')
    plt.ylabel('Score')
    plt.title('Entropy, pruning')
    plt.savefig('entropy_pruning_scores_varying_alpha.png')
    plt.clf()

'''plot the depths as alpha varies'''
```

```
def plot_alpha_depths():
    plt.plot(ccp_alphas, depths, label='Depth')
    plt.legend(loc="upper right")
    plt.xlabel('Alpha')
    plt.ylabel('Depth')
    plt.title('Entropy, pruning')
    plt.savefig('entropy_pruning_depths_varying_alpha.png')
    plt.clf()

plot_alpha_scores()
plot_alpha_depths()
```

This code lets a decision tree grow to full depth, and then we collect all the possible cost complexity pruning paths and their alpha values. Then we create a tree with each value and measure its score on the test set, we also record the max depth of each such tree. We can see the results below:





We can see that with a well chosen α we get a small but noticeable improvement over our best score in the non pruning experiments (0.86 vs 0.85). A one percent improvement might seem very small, but considering that the classification rate is already somewhat high, the absolute number of errors lower by a factor of $\frac{1}{15}$.

Part 3: Random Forests

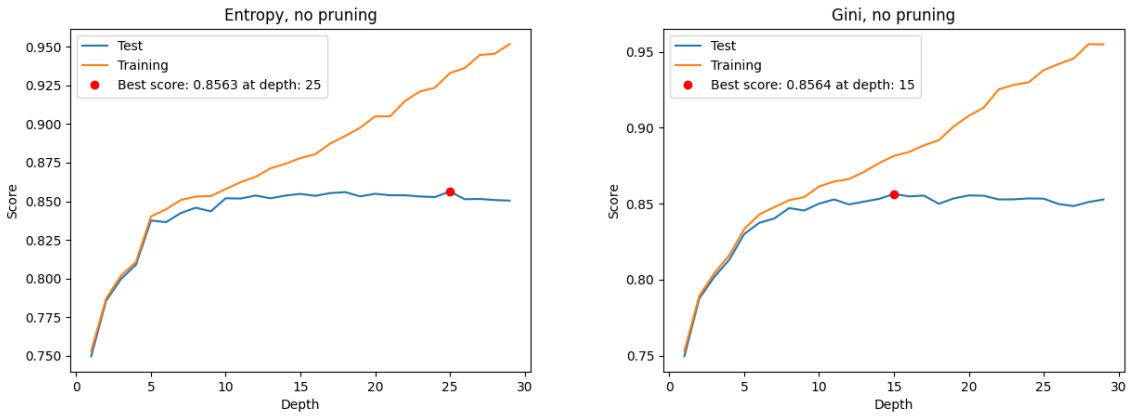
One important parameter for a random forest is the maximum depth. Our first experiment will be to find an optimal value for this parameter. We will be using the recommended number of estimators, $\sqrt{d}$, where d is the number of features. We will be testing both the entropy and gini criteria, and we will be using the same training set size of 75% for the data. By default SKL enables bootstrap sampling, so we will not be explicitly enabling it, and the default value

of $n' = n$ will be used, where n' is the number of samples in the bootstrap sample (with replacement) and n is the size of the original training set.

```
train, test = read_data.partition_data(read_data.read_data())
num_features = len(train.columns) - 1
num_estimators = int(np.floor(np.sqrt(num_features)))


'''Test various depths, with entropy as the criterion,
store and plot the scores on both training and test sets using matplotlib'''
def test_entropy_depths():
    best_depth = 0
    best_score = 0
    test_scores = []
    training_scores = []
    for i in range(1, 30):
        model = RandomForestClassifier(n_estimators=num_estimators,max_depth=i, random_sta
        model.fit(train.drop(columns=['income']), train['income'])
        test_scores.append(model.score(test.drop(columns=['income']), test['income']))
        training_scores.append(model.score(train.drop(columns=['income']), train['income']
        if model.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = model.score(test.drop(columns=['income']), test['income'])
            best_depth = i
    plt.plot(range(1, 30), test_scores, label='Test')
    plt.plot(range(1, 30), training_scores, label='Training')
    plt.plot(best_depth, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at depth: ' + str(best_depth))
    plt.legend(loc="best")
    plt.xlabel('Depth')
    plt.ylabel('Score')
    plt.title('Entropy, no pruning')
    #plt.show()
    plt.savefig('rf_entropy_scores_varying_depth.png')
    plt.clf()


'''Test various depths, with gini as the criterion'''

def test_gini_depths():
    best_depth = 0
    best_score = 0
```

```
    test_scores = []
    training_scores = []
    for i in range(1, 30):
        model = RandomForestClassifier(n_estimators=num_estimators,max_depth=i, random_sta
        model.fit(train.drop(columns=['income']), train['income'])
        test_scores.append(model.score(test.drop(columns=['income']), test['income']))
        training_scores.append(model.score(train.drop(columns=['income']), train['income']
        if model.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = model.score(test.drop(columns=['income']), test['income'])
            best_depth = i
    plt.plot(range(1, 30), test_scores, label='Test')
    plt.plot(range(1, 30), training_scores, label='Training')
    plt.plot(best_depth, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at depth: ' + str(best_depth))
    plt.legend(loc="best")
    plt.xlabel('Depth')
    plt.ylabel('Score')
    plt.title('Gini, no pruning')
    #plt.show()
    plt.savefig('rf_gini_scores_varying_depth.png')
    plt.clf()

test_entropy_depths()
test_gini_depths()
```

Running this code gives us the following results:



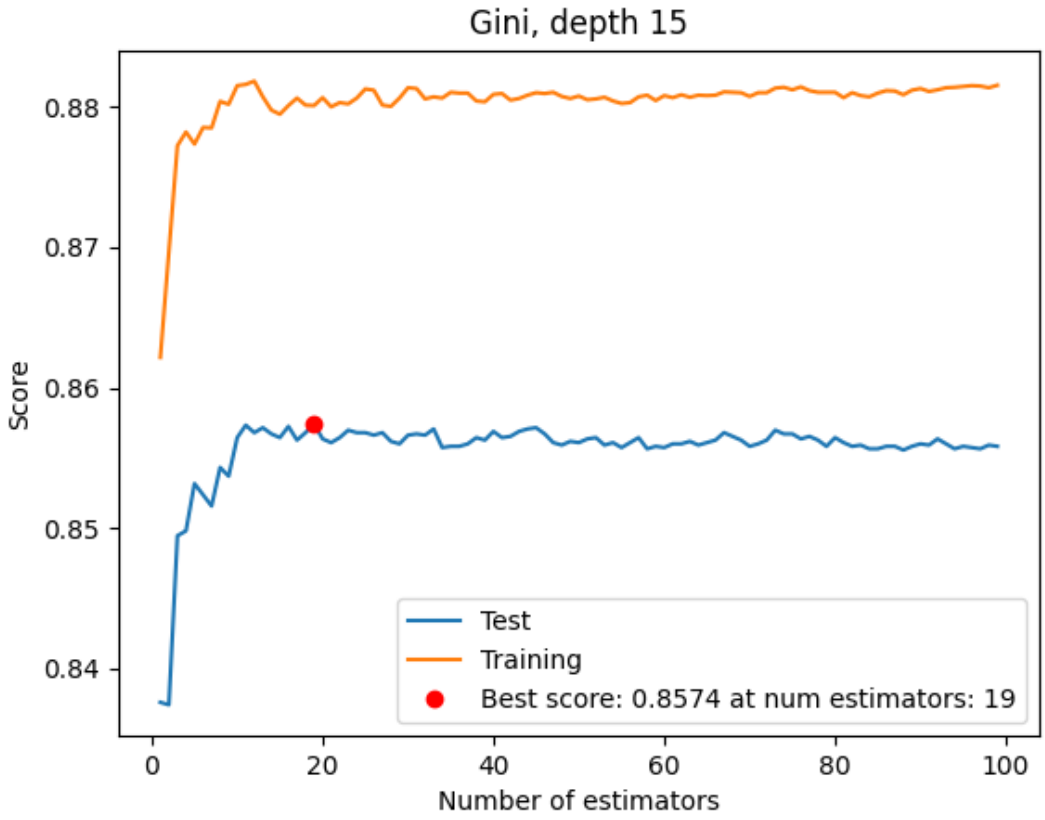
We can see that increasing depth is quite beneficial for both criteria, until about depth 10, where the test score stagnates. Again, there doesn't seem to be significant difference between

the two criteria, other than entropy hitting it's best test score at a somewhat higher depth, but the absolute difference is not very large. We will be using a depth of 15, with a gini criterion, for the rest of the experiments.

In our next experiment we vary the number of estimators, using the gini criterion and a fixed max depth of 15, as well as the same training set size of 75%. We will be using the default value of $n' = n$ for the bootstrap sample size.

```
'''test various number of estimators, with gini as the criterion, depth 15'''

def test_gini_estimators():
    best_num_estimators = 0
    best_score = 0
    test_scores = []
    training_scores = []
    for i in range(1, 100):
        model = RandomForestClassifier(n_estimators=i,max_depth=15, random_state=0, criter
        model.fit(train.drop(columns=['income']), train['income'])
        test_scores.append(model.score(test.drop(columns=['income']), test['income']))
        training_scores.append(model.score(train.drop(columns=['income']), train['income']
        if model.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = model.score(test.drop(columns=['income']), test['income'])
            best_num_estimators = i
    plt.plot(range(1, 100), test_scores, label='Test')
    plt.plot(range(1, 100), training_scores, label='Training')
    plt.plot(best_num_estimators, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at num estimators: ' + str(best_num_estimators))
    plt.legend(loc="best")
    plt.xlabel('Number of estimators')
    plt.ylabel('Score')
    plt.title('Gini, depth 15')
    #plt.show()
    plt.savefig('rf_gini_scores_varying_estimators.png')
    plt.clf()

test_gini_estimators()
```

Running this code gives us the following results:



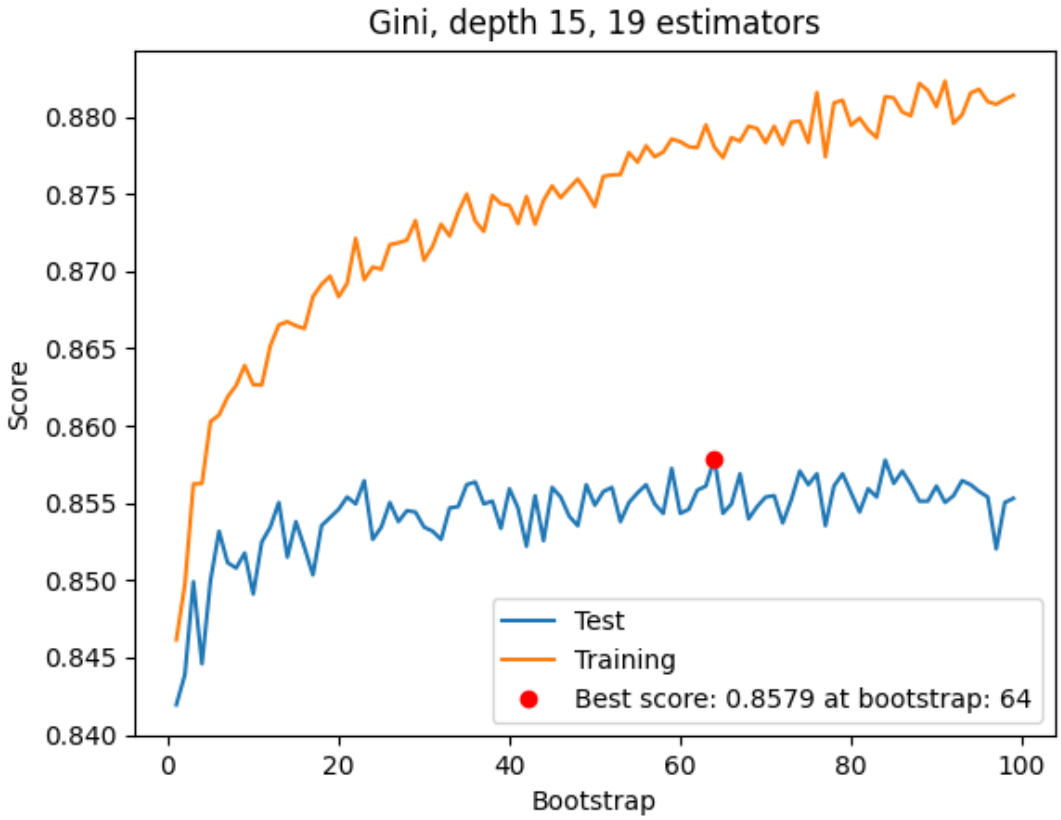
Once again we can see that the model struggles to make predictions when deprived of a resource, and performance quickly climbs as estimators are added. However, we also see performance plateau quite quickly at about 15 estimators. This somewhat aligns with the recommended value of $\sqrt{d}$ estimators, considering that we have $d \approx 1000$ and $\sqrt{d} \approx 32$. Our next step is to test the effect of varying the bootstrap sample size, n', while keeping the number of estimators fixed at 19, and the depth fixed at 15.

We use the following code to test the effects of different bootstrap sample sizes:

```
'''test various bootstrap values, with gini as the criterion, depth 15, 19 estimators'''

def test_gini_bootstrap():
    best_bootstrap = False
    best_score = 0
    test_scores = []
```

```
    training_scores = []
    for i in range(1,100):
        model = RandomForestClassifier(n_estimators=19,max_depth=15, random_state=0, crite
        model.fit(train.drop(columns=['income']), train['income'])
        test_scores.append(model.score(test.drop(columns=['income']), test['income']))
        training_scores.append(model.score(train.drop(columns=['income']), train['income']
        if model.score(test.drop(columns=['income']), test['income']) > best_score:
            best_score = model.score(test.drop(columns=['income']), test['income'])
            best_bootstrap = i
    plt.plot(range(1,100), test_scores, label='Test')
    plt.plot(range(1,100), training_scores, label='Training')
    plt.plot(best_bootstrap, best_score, 'ro', label='Best score: '
        + "{:.4f}".format(best_score) + ' at bootstrap: ' + str(best_bootstrap))
    plt.legend(loc="best")
    plt.xlabel('Bootstrap')
    plt.ylabel('Score')
    plt.title('Gini, depth 15, 19 estimators')
    #plt.show()
    plt.savefig('rf_gini_scores_varying_bootstrap.png')
    plt.clf()

test_gini_bootstrap()
```

Running this code gives us the following results:



We can see that the performance on the test set quickly increases as we increase bootstrap size from very small values, but then the increase slows down and becomes quite noisy after about 25%. This is likely because we are using an ensemble, and as long as the bootstrap size is not unreasonably small we get diverse enough samples to broadly cover the space and ensure our ensemble has good coverage.