30955101 郭育麟

Patter Recognition hw3

Part 1 Coding

1. (5%) Please compute the Entropy and Gini Index of the given array by the formula on the slides.

$$\text{Gini} = 1 - \sum_j p_j^2$$

$$\text{Entropy} = -\sum_j p_j \log_2 p_j$$

```
In [80]: print("Gini of data is ", gini(data))
         Gini of data is  0.4628099173553719

In [81]: print("Entropy of data is ", entropy(data))
         Entropy of data is  0.9456603046006402
```

2. (20%) Implement the Decision Tree algorithm (CART, Classification and Regression Trees) and train the model by the given arguments, and print the accuracy score on the test data. You should implement **two arguments** for the Decision Tree algorithm,

1) **Criterion**: The function to measure the quality of a split. Your model should support "gini" for the Gini impurity and "entropy" for the information gain.

2) **Max_depth**: The maximum depth of the tree. If Max_depth=None, then nodes are expanded until all leaves are pure. Max_depth=1 equals to split data once

Note: You should get the same results when re-building the model with the same rguments, ***no need to prune the trees***

Note: You can find the best split threshold by two methods. First one: 1) Try N-1 threshold values, where the i-th threshold is the average of the i-th and (i+1)-th sorted values. Second one: Use the unique sorted value of the feature as the threshold to split the data.

Hint: You can use the recursive method to build the nodes

Leaf()

```python
class Leaf():
    def __init__(self, data):
        counts = {}
        for row in data :
            label = row[-1]
            if label not in counts:
                counts[label] = 0
            counts[label] = counts[label] + 1
        self.predictions = counts
```

In leaf node , we saved the prediction result .

DecisionNode()

```
class Decision_Node():
    def __init__(self, question, left_branch, right_branch, col, feature_importance):
        self.question = question
        self.left_branch = left_branch
        self.right_branch = right_branch
        self.col = col
        self.feature_importance = feature_importance
```

In Decision Node , we saved the which question should we asked and what the left and right branch is , and the feature importance .

DecisionTree()

```
class DecisionTree():
    def __init__(self, criterion='gini', max_depth=None, Plot_tree = False, show_procedure = False):
        self.criterion = criterion
        self.max_depth = max_depth + 1
        self.Plot = Plot_tree
        self.show_procedure = show_procedure
        if criterion == 'gini':
            self.measure_func = gini
        if criterion == 'entropy':
            self.measure_func = entropy
        return None

    def partition(self, data, col, question):
        left_data, right_data = [], []
        for index in range(data.shape[0]):
            if data[index, col] <= question :
                left_data.append(data[index])
            else :
                right_data.append(data[index])
        left_data = np.array(left_data)
        right_data = np.array(right_data)
        return left_data, right_data

    def info_gain(self, left_data, right_data, current_uncertainty):
        p = float(len(left_data) / (len(left_data) + len(right_data)))
        return current_uncertainty - p * self.measure_func(left_data[:, -1]) - (1-p) * self.measure_func(right_data[:, -1])
```

```
    def find_best_split(self, data, depth):
        feature = data[:, 0:-1]
        label = data[:, -1]
        best_gain = 0
        best_question = None
        best_col = 0
        best_feature_importance = 0
        current_uncertainty = self.measure_func(label)
        for col in range(feature.shape[1]):
            # Get one columns feature
            values = set([row[col] for row in feature])
            for val in values :
                left_data, right_data = self.partition(data, col, val)
                if (len(left_data) == 0 or len(right_data) == 0):
                    continue
                gain = self.info_gain(left_data, right_data, current_uncertainty)
                feature_importance = gain * len(data)
                if gain > best_gain :
                    best_gain, best_question, best_col = gain, val, col
                    best_feature_importance = feature_importance

        return best_gain, best_question, best_col, best_feature_importance

    def build_tree(self, data, depth):
        if depth+1 < self.max_depth:
            best_gain, best_question, best_col, best_feature_importance = self.find_best_split(data, depth)
            if self.show_procedure:
                print("======================================================================")
                print(f"Depth is {depth+1}\nThe question is {feature_names[best_col]} <= {best_question}\t Gain is {best_gai
            if best_gain == 0 :
                if self.show_procedure :
                    print(f"Found leaves")
                return Leaf(data)
            else:
                left_data, right_data = self.partition(data, best_col, best_question)
                left_branch = self.build_tree(left_data, depth+1)
                right_branch = self.build_tree(right_data, depth+1)
            return Decision_Node(best_question, left_branch, right_branch, best_col, best_feature_importance)
        else :
            if self.show_procedure:
                print(f"Found leaves")
            return Leaf(data)
```

```python
    def fit(self, X, y):
        print(f"============================Start training============================")
        print(f"Training example shape {X.shape}")
        print(f"Training lable shape {y.shape}")
        data = np.concatenate((X, y), axis = 1)
        depth = 0
        tree = self.build_tree(data, depth)
        if self.Plot:
            print("============================================")
            self.print_tree(tree)
        return tree

    def print_tree(self, node, spacing =""):
        # Base case: we've reached a leaf
        if isinstance(node, Leaf):
            print (spacing + "Predict", node.predictions)
            return
        # Print the question at this node
        print (spacing + feature_names[node.col] +"<=" +str(node.question))

        # Call this function recursively on the true branch
        print (spacing + '--> Left:')
        self.print_tree(node.left_branch, spacing + "  ")

        # Call this function recursively on the false branch
        print (spacing + '--> Right:')
        self.print_tree(node.right_branch, spacing + "  ")

    def classify(self, data, node):
        if isinstance(node, Leaf):
            return node.predictions
        else :
            col = node.col
            question = node.question
            if (data[col] <= question):
                return self.classify(data, node.left_branch)
            else :
                return self.classify(data, node.right_branch)
```

```python
    def print_leaf(self, counts):
        total = float(sum(counts.values()))
        probs = {}
        for class_ in counts.keys():
            probs[class_] = round(float(counts[class_] / total), 3)
        return probs

    def count_acc(self, testing_feature, testing_label, node, show_every = False):
        count = 0
        for i in range(len(testing_feature)):
            predict = self.print_leaf(self.classify(testing_feature[i], node))
            predict_answer = int(max(predict, key=predict.get))
            if(predict_answer == testing_label[i]):
                count = count + 1
            if show_every :
                print(f"Ground truth is {y_test[i,0]}\t Predict is {predict}")
        return round(count/testing_feature.shape[0],3)

    def Get_feature_importance(self, node):
        dictonary = {}
        self.Calculate_feature_importance(node, dictonary)
        sum_ = sum(dictonary.values())
        dictonary = {k: v / sum_ for k, v in dictonary.items()}
        return dictonary

    def Calculate_feature_importance(self, node, dictonary) :
        if isinstance(node, Leaf):
            return
        if feature_names[node.col] not in dictonary:
            dictonary[feature_names[node.col]] = 0
        dictonary[feature_names[node.col]] += node.feature_importance
        self.Calculate_feature_importance(node.left_branch, dictonary)
        self.Calculate_feature_importance(node.right_branch, dictonary)
```

In this step , we use recursive function to ask the question and find the best question which can get highest gain , after asking the question , we separate data into two parts first and generate decision node , and recursive it until when it could not be separated or reach the max level , and use these data to generate leaf node .
When testing data , we asking the question from decision tree , when it achieve leaf node , we could get the final prediction .
We use

$$\text{gain} = 1 - \frac{|\text{left branch}|}{|left+right\ branch|} * gini(left) - \frac{|right\ branch|}{|left+right\ branch|} * gini(right)$$

to measure the gain .

2.1 Using Criterion='gini' to train the model and show the accuracy score of test data by Max_depth=3 and Max_depth=10, respectively.

Question 2.1

Using Criterion='gini', showing the accuracy score of test data by Max_depth=3 and Max_depth=10, respectively.

In [10]:
```
clf_depth3 = DecisionTree(criterion='gini', max_depth=3, Plot_tree = False, show_procedure = False)
tree = clf_depth3.fit(x_train,y_train)
acc_clf_depth3 = clf_depth3.count_acc(x_test, y_test, tree, show_every = False)
clf_depth3_FI = clf_depth3.Get_feature_importance(tree)


clf_depth10 = DecisionTree(criterion='gini', max_depth=10, Plot_tree = False, show_procedure = False)
tree = clf_depth10.fit(x_train,y_train)
acc_clf_depth10 = clf_depth10.count_acc(x_test, y_test, tree, show_every = False)
clf_depth10_FI = clf_depth10.Get_feature_importance(tree)

print(f"clf_depth3 accuracy is {acc_clf_depth3}")
print(f"clf_depth10 accuracy is {acc_clf_depth10}")
```

```
=============================Start training=============================
Training example shape (426, 30)
Training lable shape (426, 1)
=============================Start training=============================
Training example shape (426, 30)
Training lable shape (426, 1)
clf_depth3 accuracy is 0.937
clf_depth10 accuracy is 0.93
```

2.2 Using Max_depth=3 to train the model and show the accuracy score of test data by Criterion=‘gini’ and Criterion=’entropy’, respectively.

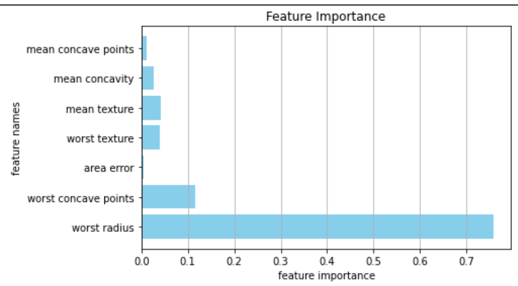
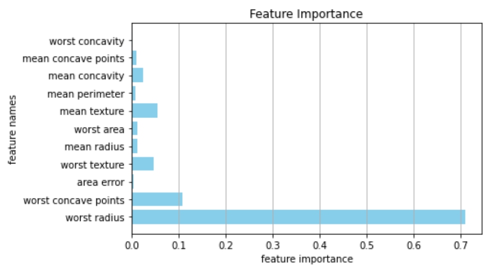
Question 2.2

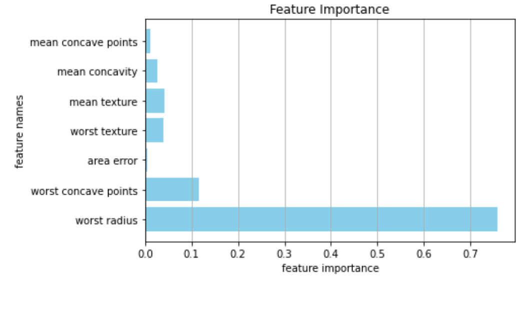
Using Max_depth=3, showing the accuracy score of test data by Criterion='gini' and Criterion='entropy', respectively.

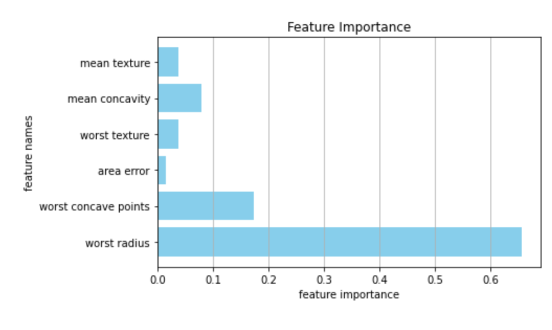
In [11]:
```
clf_gini = DecisionTree(criterion='gini', max_depth=3, Plot_tree = False, show_procedure = False)
tree = clf_gini.fit(x_train,y_train)
acc_gini = clf_gini.count_acc(x_test, y_test, tree, show_every = False)
clf_gini_FI = clf_gini.Get_feature_importance(tree)

clf_entropy = DecisionTree(criterion='entropy', max_depth=3, Plot_tree = False, show_procedure = False)
tree = clf_entropy.fit(x_train,y_train)
acc_entropy = clf_entropy.count_acc(x_test, y_test, tree, show_every = False)
clf_entropy_FI = clf_entropy.Get_feature_importance(tree)
print(f"clf_gini accuracy is {acc_gini}")
print(f"clf_entropy accuracy is {acc_entropy}")
```

```
=============================Start training=============================
Training example shape (426, 30)
Training lable shape (426, 1)
=============================Start training=============================
Training example shape (426, 30)
Training lable shape (426, 1)
clf_gini accuracy is 0.937
clf_entropy accuracy is 0.951
```

3. (15%) Plot the feature importance of your Decision Tree model. You can use the model for Question 2.1, max_depth=10. (You can simply count the number of a feature used in the tree, instead of the formula in the reference. Find more details on the sample code. (matplotlib is allowed to use)

<table>
<tr><td>criterion='gini', max_depth=3</td><td>criterion='gini', max_depth=10</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>criterion='gini', max_depth=3,</td><td>criterion='entropy', max_depth=3,</td></tr>
</table>





4.(20%) Implement the [random forest]() algorithm by using the CART you just implemented for Question 2. You should implement **three arguments** for the random forest model.

1) **N_estimators**: The number of trees in the forest.
2) **Max_features**: The number of features to consider when looking for the best split
3) **Bootstrap**: Whether bootstrap samples are used when building trees

```
class RandomForest():
    def __init__(self, n_estimators, max_features, boostrap=True, criterion='gini', max_depth=None, Plot_tree = False, show_
        self.n_estimators = n_estimators
        self.max_features = int(max_features)
        self.boostrap = boostrap
        self.max_depth = max_depth
        if max_depth == None:
            self.max_depth = 1000
        self.Plot_tree = Plot_tree
        self.show_procedure = show_procedure
        self.criterion = criterion
        if criterion == 'gini':
            self.measure_func = gini
        if criterion == 'entropy':
            self.measure_func = entropy

        self.top_decision = []
        self.forest = []
        for i in range(self.n_estimators):
            self.forest.append(DecisionTree(criterion=self.criterion, max_depth=self.max_depth, Plot_tree=self.Plot_tree, sh

        self.random_choose = []
```

```
    def fit(self, x, y):
        number , feature_num = x.shape
        feature_all = np.arange(feature_num)
        number_all = np.arange(number)
        for i in range(self.n_estimators):
            print(f"------------------------------------Processing forest {i}------------------------------------")
            random_feature = random.sample(list(feature_all), self.max_features)
            self.random_choose.append(random_feature)
            if self.boostrap:
                sample_number = random.sample(list(number_all), int(number * 2/3))
                self.top_decision.append(self.forest[i].fit(x[sample_number][:,random_feature], y[sample_number]))
            else:
                self.top_decision.append(self.forest[i].fit(x[:,random_feature], y))
        return self.top_decision

    def count_acc(self, x, y):
        pred = np.zeros((len(y),1))
        count = 0
        for i in range(len(x)):
            vote= []
            for j in range(self.n_estimators):
                Now_tree = self.forest[j]
                Now_top_decision = self.top_decision[j]
                Now_choose = self.random_choose[j]
                predict = Now_tree.print_leaf(Now_tree.classify(x[i, Now_choose], Now_top_decision))
                predict_answer = int(max(predict, key = predict.get))
                vote.append(predict_answer)
            predict , count_ = np.unique(np.array(vote), return_counts=True)
            pred[i] = predict[np.argmax(count_)]
            if pred[i] == y[i] :
                count += 1
        return round(count/len(y),3)
```

We implement a random forest by decision tree , and using the Bootstrap algorithm to randomly select 2/3 of data to generate one tree , so when we generate n tree to become a forest , all the trees are not the same , it has some different . And we use the max_features to condition the number of features that tree can use . Finally , we use voting mechanism to generate predict result from all trees .

4.1 Using Criterion='gini', Max_depth=None, Max_features=sqrt(n_features), Bootstrap=True to train the model and show the accuracy score of test data by n_estimators=10 and n_estimators=100, respectively.

```
clf_10tree = RandomForest(n_estimators=10, max_features=np.sqrt(x_train.shape[1]), boostrap=True, criterion='gini', max_dept
clf_10tree_forest = clf_10tree.fit(x_train, y_train)
clf_10tree_acc = clf_10tree.count_acc(x_test, y_test)

clf_100tree = RandomForest(n_estimators=100, max_features=np.sqrt(x_train.shape[1]), boostrap=True, criterion='gini', max_de
clf_100tree_forest = clf_100tree.fit(x_train, y_train)
clf_100tree_acc = clf_100tree.count_acc(x_test, y_test)
```

```
print(f"The accuracy of n_estimators=10 is {clf_10tree_acc}")
print(f"The accuracy of n_estimators=100 is {clf_100tree_acc}")
```

```
The accuracy of n_estimators=10 is 0.944
The accuracy of n_estimators=100 is 0.93
```

4.2 Using Criterion='gini', Max_depth=None, N_estimators=10, Bootstrap=True, to train the model and show the accuracy score of test data by Max_features=sqrt(n_features) and Max_features=n_features, respectively.

```
clf_random_features = RandomForest(n_estimators=10, max_features=np.sqrt(x_train.shape[1]), boostrap=True, criterion='gini',
clf_random_features_forest = clf_random_features.fit(x_train, y_train)
clf_random_features_acc = clf_random_features.count_acc(x_test, y_test)

clf_all_features = RandomForest(n_estimators=10, max_features=x_train.shape[1], boostrap=True, criterion='gini', max_depth=N
clf_all_features_forest = clf_all_features.fit(x_train, y_train)
clf_all_features_acc = clf_all_features.count_acc(x_test, y_test)
```

```
print(f"The accuracy of random_feature is {clf_random_features_acc}")
print(f"The accuracy of all_feature is {clf_all_features_acc}")
```

```
The accuracy of random_feature is 0.916
The accuracy of all_feature is 0.93
```

Part 2 Question

1 . By differentiating the error function below with respect to α_m

$$\mathrm{E} = \mathrm{e}^{-\frac{\alpha_m}{2}} \sum_{n \in T_m} w_n^{(m)} + e^{\frac{\alpha_m}{2}} \sum_{n \in M_m} w_n^{(m)}$$

$$= \left(\mathrm{e}^{-\frac{\alpha_m}{2}} - e^{\frac{\alpha_m}{2}} \right) \sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n) + e^{-\frac{\alpha_m}{2}} \sum\nolimits_{n=1}^{N} w_n^{(m)}$$

Show that the parameters α_m in AdaBoost algorithm are updated using $\alpha_\mathrm{m} = \ln(\frac{1-\varepsilon}{\varepsilon_\mathrm{m}})$ in which ε_m is defined by $\varepsilon_\mathrm{m} = \frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$

Proof.

By Derive the error function , we can get

$$\frac{dE}{d\alpha_m} = \frac{d(\left(\mathrm{e}^{-\frac{\alpha_m}{2}} - e^{\frac{\alpha_m}{2}} \right) \sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n) + e^{-\frac{\alpha_m}{2}} \sum_{n=1}^{N} w_n^{(m)})}{d\alpha_m} = 0$$

Then

$$\left(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n) \right) \left(-\frac{1}{2} e^{-\frac{\alpha_m}{2}} - \frac{1}{2} e^{\frac{\alpha_m}{2}} \right) + (\sum\nolimits_{n=1}^{N} w_n^{(m)})(-\frac{1}{2} e^{-\frac{\alpha_m}{2}}) = 0$$

$$\rightarrow (\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n))(e^{-\frac{\alpha_m}{2}} + e^{\frac{\alpha_m}{2}}) = (\sum\nolimits_{n=1}^{N} w_n^{(m)})(e^{-\frac{\alpha_m}{2}})$$

$$\xrightarrow{minus(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)) e^{-\frac{\alpha_m}{2}})}$$

$$\rightarrow \left(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n) \right) e^{\frac{\alpha_m}{2}} = (\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) = t_n))\, e^{-\frac{\alpha_m}{2}}$$

$$\xrightarrow{Take\ natural\ log}$$

$$\rightarrow \ln\left(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)\right) + \frac{\alpha_m}{2} = \ln(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) = t_n)) - \frac{\alpha_m}{2}$$

$$\mathrm{a_m} = \ln(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) = t_n)) - \ln(\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n))$$

$$= \ln\left(\frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) = t_n)}{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)} \right) = \ln \left(\frac{\frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) = t_n)}{\sum_{n=1}^{N} w_n^{(m)}}}{\frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}} \right) =$$

$$\ln\left(\frac{1-\frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n)\neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}}{\frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n)\neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}}\right)$$

Let $\varepsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(x_n)\neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$, we could have $\alpha_m = \ln(\frac{1-\varepsilon_m}{\varepsilon_m})$

2.

2. (15%) Consider a data set comprising 400 data points from class C_1 and 400 data points from class C_2. Suppose that a tree model A splits these into (300, 100) assigned to the first leaf node (predicting C_1) and (100, 300) assigned to the second leaf node (predicting C_2), where (n, m) denotes that n points come from class C_1 and m points come from class C_2. Similarly, suppose that a second tree model B splits them into (200, 0) and (200, 400), respectively. Evaluate the misclassification rates for the two trees and hence show that they are equal. Similarly, evaluate the pruning criterion $C(T) = \sum_{\tau=1}^{|T|} Q_\tau(T) + \lambda |T|$ for the cross-entropy case $Q_\tau(T) = -\sum_{k=1}^{K} p_{\tau k} ln(p_{\tau k})$ for the two trees and show that tree B is lower than tree A. Leaf nodes are indexed by $\tau = 1, ..., |T|$, with leaf node τ represents a region R_τ, and $p_{\tau k}$ is the proportion of data points in region R_τ assigned to class k, where $k = 1, ..., K$.

(a) Misclassification Rates

Tree A:

Predict C_1 for left leaf node and predict C_2 for right leaf node . Hence the misclassification for Tree A is $\frac{100}{800} + \frac{100}{800} = 0.25$

Tree B:

Predict C_1 for left leaf node and predict C_2 for right leaf node . Hence the misclassification for Tree B is $\frac{0}{800} + \frac{200}{800} = 0.25$

So the misclassification rates for the two trees are equal .

(b) Pruning Criterion : Cross-Entropy

Tree A and Tree B has two leaf nodes , so |T| = 2

Cross-Entropy(Tree A) = $\left(-\frac{3}{4}\ln\left(\frac{3}{4}\right) - \frac{1}{4}\ln\left(\frac{1}{4}\right) + 2\lambda\right) + \left(-\frac{1}{4}\ln\left(\frac{1}{4}\right) - \frac{3}{4}\ln\left(\frac{3}{4}\right) + 2\lambda\right)$

$$= \ln\left(\frac{16}{3\sqrt{3}}\right) + 4\lambda$$

Cross-Entropy(Tree B) = $(-1\ln(1) + 2\lambda + (-\frac{1}{3}\ln\left(\frac{1}{3}\right) - \frac{2}{3}\ln\left(\frac{2}{3}\right) + 2\lambda)$

$$= \ln\left(\frac{3}{\sqrt[3]{4}}\right) < \ln\left(\frac{16}{3\sqrt{3}}\right)$$

$1.890 \approx \frac{3}{\sqrt[3]{4}} < \frac{16}{3\sqrt{3}} \approx 3.079$. Hence $\ln\left(\frac{3}{\sqrt[3]{4}}\right) < \ln\left(\frac{16}{3\sqrt{3}}\right)$

So Cross-Entropy(Tree B)< Cross-Entropy(Tree A)

3.

Verify that if we minimize the sum-of-squares error between a set of training values $\{\mathrm{t_n}\}_{n=1\sim N}$ (N is number of training data) and a single predictive value t, then the optimal solution for t is given by the mean of the $\{\mathrm{t_n}\}_{n=1\sim N}$

Proof:

The sum-of-squares error is $\mathrm{f(t)} = \sum_{n=1}^{N}(t - t_n)^2$

The minimum values occurs when $\frac{\mathrm{df}}{dt} = 0$

$$\to \frac{\mathrm{df}}{dt} = \frac{d(\sum_{n=1}^{N}(t - t_n)^2)}{dt} = \sum_{n=1}^{N} 2(t - t_n) = 2\sum_{n=1}^{N}(t - t_n) = 0$$

$$\to \sum_{n=1}^{N}(t - t_n) = 0$$

$$\to \sum_{n=1}^{N} t - \sum_{n=1}^{N} t_n = Nt - \sum_{n=1}^{N} t_n = 0$$

Hence t = $\frac{1}{\mathrm{N}}\sum_{n=1}^{N} t_n$, So the optimal solution for t is given by the mean of

$\{\mathrm{t_n}\}_{n=1\sim N}$