CS 6375 Assignment 2

Lakshmipriya Narayanan

Problem 1

(a): Nearest neighbor regression

Yes, we can extend KNN method to regression because the KNN classifier method is used for a categorical response whereas the KNN regression method can be used for a quantitative response (continuous numerical response).
Algorithm for KNN Regression will take the average value of the K-nearest neighbors' values to make a prediction whereas the algorithm for classification looks at the k nearest data points and assigns the class of the observation according to the maximum number of the nearest neighbors.

(b): Irrelevant attributes

When we have irrelevant features or inputs ($x^{(i)}$) in the example of diagnosing a medical condition, they can consist of relevant attributes like age, weight, symptoms, etc. and irrelevant attributes like favorite color, year of graduation, what movie they watched on the day of illness do not help KNN because these attributes lead to kNN performing badly due to the distance calculations in the algorithm and for attributes like those mentioned above, it would be hard for kNN to find a similar neighbor that contributes to the medical diagnosis.

(c): SEE CODE

Problem 2

(a): Part I:

This greedy strategy is still a good strategy because information gain in decision trees help identify the most significant features for our corresponding response. For example let's consider one similar to what was done in class. Suppose we use predictors like age, Blood pressure and blood sugar levels to predict whether a person has diabetes or not. Then, we estimate potential splits on above mentioned predictors and calculate information gains for each. For example, consider blood sugar levels, we split this category into patients with low, normal, and high. This will help in predicting diabetes because patients in the high category are more likely to have diabetes.
If we used $H(Y|X_i)$, we would want to minimize it because lower conditional entropy of a feature $X_i \implies X_i$ provides more information about Y. Consider the above mentioned diabetes example. Then the blood sugar level feature would have low conditional entropy. Given the level of blood sugar for a patient, the remaining entropy of the response variable (diabetes) is decreased.

(b): Part II:

The Gini split condition, $G(S) = 1 - \sum_{i=1}^{N} {p_i}^2$ is a good splitting condition because it helps create the most informative nodes in a decision tree. This is because it measures the probability of a feature being misclassified when chosen randomly.
The split condition $G(S) = 1 - \sum_{i=1}^{N} {p_i}^3$ might not be very effective because when the probability p_i is cubed, it gives more weight to the class with a higher probability, hence making the splits very sensitive to changes in the distribution of the class.

(c): Part III:

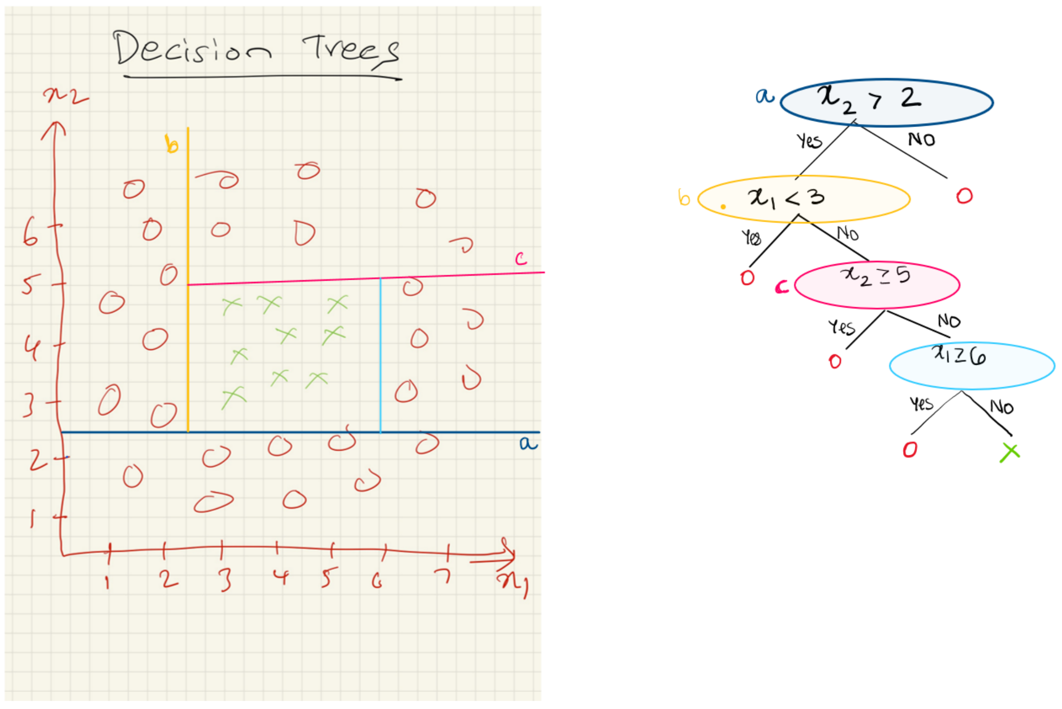

Figure 1: Approximate decision boundary and final solution obtained by a decision tree algorithm

When we assume that we have no limit on the depth of the tree, then the solution will overfit our data. The depth of a tree can be increased by maximizing the number of splits in our tree or by setting a minimum number of elements in our leaf node.

Problem 3

(a):

Parameters of the dice roll process: $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6 \; = \; 1 - \theta_1 - \theta_2 - \theta_3 - \theta_4 - \theta_5 \;\}$ where $\theta_i = P(X = i)$ for $i = 1, ..., 5$ is the probability of getting i on the dice. We have 5 parameters.

(b):

Observe that, $N = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 + \alpha_6$.
Then, likelihood function, $P(N|\theta) = \theta_1^{\alpha_1} \cdot \theta_2^{\alpha_2} \cdot \theta_3^{\alpha_3} \cdot \theta_4^{\alpha_4} \cdot \theta_5^{\alpha_5} \cdot (1 - \theta_1 - \theta_2 - \theta_3 - \theta_4 - \theta_5)^{\alpha_6}$

$$P(N|\theta) = \prod_{i=1}^{5} \theta_i{}^{\alpha_i} \cdot (1 - \textstyle\sum_{i=1}^{5} \theta_i)^{\alpha_6}$$

$$\ln\, P(N|\theta) = \textstyle\sum_{i=1}^{5} \alpha_i \cdot \theta_i + \alpha_6 \cdot (1 - \sum_{i=1}^{5} \theta_i)$$

$$\tfrac{d}{d\theta}\, \ln\, P(N|\theta) = \tfrac{d}{d\theta} \left[\textstyle\sum_{i=1}^{5} \alpha_i \cdot \theta_i + \alpha_6 \cdot (1 - \sum_{i=1}^{5} \theta_i) \right]$$

$$\tfrac{d}{d\theta}\, \ln\, P(N|\theta) = \textstyle\sum_{i=1}^{5} \frac{\alpha_i}{\theta_i} + \alpha_6 + (0 - 5 \cdot (1)) = 0.$$

$$\text{Solving for } \theta, \text{ we get } \hat{\theta}_{MLE} = \textstyle\sum_{i=1}^{5} \frac{\alpha_i}{\theta_i} - 5 \cdot \alpha_6$$

(c):

They will simply be the ratio of the number of occurrences and the size of the sample space: $\theta_i = \frac{\alpha_i}{N}$ for $i = 1, ..., 6$

(d):

Since this is a discrete case, we can use the Dirichlet prior because this distribution is over $\theta_i \in (0, 1)$ characterized by parameters $\alpha_1, ..., \alpha_6$. And as seen in b above, $f(\theta_i; \alpha_i) \propto \prod\limits_{i=1}^{6} {\theta_i}^{\alpha_i - 1}$

(e):

Posterior Distribution,

$$P(\theta|N) = P(N|\theta) \cdot P(\theta)$$

$$P(\theta|N) = \theta_1^{\alpha_1} \cdot \theta_2^{\alpha_2} \cdot \theta_3^{\alpha_3} \cdot \theta_4^{\alpha_4} \cdot \theta_5^{\alpha_5} \cdot (1 - \theta_1 - \theta_2 - \theta_3 - \theta_4 - \theta_5)^{\alpha_6} \cdot \frac{1}{6}$$

Therefore, $P(\theta|N) = \prod\limits_{i=1}^{5} {\theta_i}^{\alpha_i} \cdot (1 - \sum_{i=1}^{5} \theta_i)^{\alpha_6}$

MAP estimate, $\theta_{MAP} = \frac{d}{d\theta}\ \ln\ P(\theta|N) = \frac{d}{d\theta}\ \ln\ \prod\limits_{i=1}^{6} {\theta_i}^{\alpha_i} \cdot (1 - \sum_{i=1}^{5} \theta_i)^{\alpha_6}$

$$\theta_{MAP} = \frac{d}{d\theta}\Big[\textstyle\sum_{i=1}^{5} \alpha_i \cdot \theta_i + \alpha_6 \cdot (1 - \sum_{i=1}^{5} \theta_i)\Big]$$

$$= \textstyle\sum_{i=1}^{5} \alpha_i \cdot \frac{1}{\theta_i} + \alpha_6 \cdot (0 - \sum_{i=1}^{5} 1)$$

$$= \textstyle\sum_{i=1}^{5} \alpha_i \cdot \frac{1}{\theta_i} + \alpha_6 \cdot (-5)$$

Therefore, $\theta_{MAP} = \sum_{i=1}^{5} \frac{\alpha_i}{\theta_i} - 5 \cdot \alpha_6$

We also observe that $\theta_{MLE} = \theta_{MAP}$

Following chunk taken completely from the demo:
https://colab.research.google.com/drive/1coQx_fGMyiOhli6xjETUaXfy3AX1PqWX#scrollTo=0266859b

```
#Import scikit-learn dataset library
from sklearn import datasets
from sklearn.model_selection import train_test_split

#Load dataset
wine = datasets.load_wine()

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(wine.data, wine.target, test_size=0.3) # 70% training and 30% test
```

Problem 1(c): .

Implementation of the KNN algorithm form scratch

```
import numpy as np
from collections import Counter

# Function to calculate euclidean distance between two points because KNN
# requires calculation of distance to determine closest neighbors.
# Eucliean distance = sqrt(x1^2 - x2^2)
def calc_euclid_dist(x1, x2):
  return np.sqrt(np.sum((x1-x2)**2))

class KNN:
  #Initialization function for KNN.
  def __init__(self, k):
    self.k = k

  #Fit a model function to assign values for features X and labels Y
  def fit_trainData(self, X, y):
    self.X_train = X
    self.y_train = y

  #Perform calculations to calculate euclidean distances between points in
  # the test data to determine its corresponding nearest neighbors.
  def predict(self, X):
    pred_labels = []
    for x in X:
      #calculate the distances
      distances = [calc_euclid_dist(x, x_train) for x_train in self.X_train]

      #get the k nearest neighbor out of all the distnaces calculated above
      k_indices = np.argsort(distances)[:self.k]
      k_nearest_labels = [self.y_train[i] for i in k_indices]

      # Determine the label with majority vote(classification is qualitative so
      # take the most common label)
      most_common = Counter(k_nearest_labels).most_common(1)
      pred_labels.append(most_common[0][0])

    return np.array(pred_labels)
```

Calculating testing and trainind data error for various values of K

```
#create a classifier with k=1, Fit the model and make the prediction
knn1 = KNN(k=1)
knn1.fit_trainData(X_train, y_train)
y_pred1 = knn1.predict(X_test)
```

```
#training data accuracy
y_pred_train1 = knn1.predict(X_train)
accurtrain1 = np.sum(y_pred_train1 == y_train) / len(y_train)
print(f"Training Data accuracy for K = 1 is {accurtrain1}")

#test data accuracy
accur1 = np.sum(y_pred1 == y_test) / len(y_test)
print(f"Test Data accuracy for K = 1 is {accur1}")
```

```
Training Data accuracy for K = 1 is 1.0
Test Data accuracy for K = 1 is 0.7037037037037037
```

These error rates match the ones obtained in the demo done in class. Clearly, for k=1, error rate is 1 implying best accuracy as per definition of KNN algorithm with k = 1(No ties.)

```
#K= 7
knn7 = KNN(k=7)
knn7.fit_trainData(X_train, y_train)
y_pred7 = knn7.predict(X_test)

y_pred_train7 = knn7.predict(X_train)
accurtrain7 = np.sum(y_pred_train7 == y_train) / len(y_train)
print(f"Training Data accuracy for K = 7 is {accurtrain7}")
accur7 = np.sum(y_pred7 == y_test) / len(y_test)
print(f"Test Data accuracy for K = 7 is {accur7}")
```

```
Training Data accuracy for K = 7 is 0.8145161290322581
Test Data accuracy for K = 7 is 0.6851851851851852
```

In the demo, the optimal k value was k = 7. The error rates match the ones in the demo. And this is a good k value because test error rate is lower than training error rate which is what we need.

```
#K= 100
knn100 = KNN(k=100)
knn100.fit_trainData(X_train, y_train)
y_pred100 = knn100.predict(X_test)

y_pred_train100 = knn100.predict(X_train)
accurtrain100 = np.sum(y_pred_train100 == y_train) / len(y_train)
print(f"Training Data accuracy for K = 100 is {accurtrain100}")
accur100 = np.sum(y_pred100 == y_test) / len(y_test)
print(f"Test Data accuracy for K = 100 is {accur100}")
```

```
Training Data accuracy for K = 100 is 0.7016129032258065
Test Data accuracy for K = 100 is 0.5925925925925926
```

For high k value k = 100, even though the test error rate is lower than training error rate and both are smaller, we might have overfit our data hence affecting the performance of the model.

Problem 4:

Implementation of decision tree

```
# Node class to represent each node in the decision tree
class TreeNode:
    def __init__(self, feat_x=None, threshold=None, leftchild=None, rightchild=None, *, val=None):
        self.feat_x = feat_x  # Index of the feature that was didvided with/split on
        self.threshold = threshold  # Threshold value for the split decides whether
                      #we should split the tree with respect the value in the
                      #left child or the right child
```

```python
        self.leftchild = leftchild  # Left child of the node
        self.rightchild = rightchild  # Right child of the node
        self.val = val  # Value present in the node if it is a leaf node, ie all values are classified

    #Check if a node is a leaf node to ensure that we have reached the end of the tree
    def is_leaf_node(self):
        return self.val is not None

# Decision Tree class for building and using the decision tree
class Dec_Tree:
    def __init__(self, min_split=2, max_depth=100, n_feat=None, split_condition = 'cond_ent'):
        # Minimum samples required to split a node is always equal to 2 because
        # we can split a node only if the node has 2 or more child nodes
        self.min_split = min_split
        # Maximum depth = 100. This can be considered as a stopping criterion to
        # avoid building a tree that's very deep
        self.max_depth = max_depth
        self.n_feat = n_feat  # Number of features to consider for splits at each node
        self.root = None  # Root node
        self.split_condition = split_condition

    # Fit a decision tree to learn the data
    def fit(self, X, y):
        # Determine the number of features to consider for splits. If there are
        # no features specified then the tree uses all the features. If the
        # number of features are specified then the tree will use the minimum of
        # the features and go sequentially
        self.n_feat = X.shape[1] if not self.n_feat else min(X.shape[1], self.n_feat)
        #self.feature_names = X.columns if isinstance(X, pd.DataFrame) else None
        # Build the tree by "growing" or adding nodes and child nodes to the
        # tree. This requires creating a helper function that performs the
        # adding of nodes to create a tree
        self.root = self._grow_tree(X, y)

    # Initially tree is of depth zero because the tree is not created yet
    # (Tree is built recursively)
    def _grow_tree(self, X, y, depth=0):
        # Set features as size of X and labels as size of y
        n_samples, n_feats = X.shape
        n_labels = len(np.unique(y))

        # Check stopping criteria:
        # If the depth of the tree >= maximum depth (100) then, all samples have the same label.
        # OR If the number of samples < 2 (minimum split condition) then, the
            # tree stops growing because we have split them all and we return the
            # leaf node with the majority label
        if (depth >= self.max_depth or n_labels == 1 or n_samples < self.min_split):
            leaf_val = self._most_common_label(y)
            return TreeNode(val=leaf_val)

        # Randomly select features to consider for splitting
        feat_idxs = np.random.choice(n_feats, self.n_feat, replace=False)

        # Find the best split to determine the best feature and threshold to
        # split it based on the information gaian
        best_feat, best_thresh = self._best_split(X, y, feat_idxs)

        # Split the data into left sub-tree and right sub-tree based on the best
        # split above and since we create these sub-trees, we increment our tree
        # height by 1. Return the new Node with the best feature, threshold and
        # child nodes.
        left_st, right_st = self._split(X[:, best_feat], best_thresh)
        leftchild = self._grow_tree(X[left_st, :], y[left_st], depth + 1)
        rightchild = self._grow_tree(X[right_st, :], y[right_st], depth + 1)
        return TreeNode(best_feat, best_thresh, leftchild, rightchild)

    def _best_split(self, X, y, feat_idxs):
        best_gain = -1  # Initialize the best gain to -1 because we need high information gain
        split_idx, split_threshold = None, None
        # Iterate over each feature index of X
        for feat_idx in feat_idxs:
            X_column = X[:, feat_idx] # get the corresponding X
            # find all unique values in the column which gives the threshold for
            # splitting
            thresholds = np.unique(X_column)

            # Iterate over each unique threshold
            for thr in thresholds:
                # Calculate the information gain for each threshold
                gain = self._IG(y, X_column, thr, self.split_condition)

                # If information gain > the best gain then update best gain, feature index, and threshold
                if gain > best_gain:
                    best_gain = gain
                    split_idx = feat_idx
                    split_threshold = thr

        #Return the best split based on the highest information gain
        return split_idx, split_threshold

    def _IG(self, y, X_column, threshold, split_condition):
        # Calculate conditional entropy of the parent node
        parent_entropy = self._entropy(y) if split_condition == 'cond_ent' else self._gini(y)

        # Split the data into left and right children based on best threshold
        left_st, right_st = self._split(X_column, threshold)
        if len(left_st) == 0 or len(right_st) == 0:
            return 0 #If the child node is empty, Information gain = 0

        # Calculate the average entropy of child nodes. Here we take weighted
        # average because the size of the child nodes may differ
        n_l, n_r = len(left_st), len(right_st)

        # Calculate the entropy of the left and right child
        if split_condition == 'cond_ent':
            e_l, e_r = self._entropy(y[left_st]), self._entropy(y[right_st])
        else:
            e_l, e_r = self._gini(y[left_st]), self._gini(y[right_st])

        child_entropy = (n_l / len(y)) * e_l + (n_r / len(y)) * e_r

        # Compute information gain
        info_gain = parent_entropy - child_entropy
        return info_gain

    def _split(self, X_column, split_thresh):
        # Split the data based on the threshold specified
        left_st = np.argwhere(X_column <= split_thresh).flatten()
        right_st = np.argwhere(X_column > split_thresh).flatten()
        return left_st, right_st

    def _entropy(self, y):
        hist = np.bincount(y)  # Count the frequency of each label.
        ps = hist / len(y)     # probability of each label
        # Calculate entropy = sum of p_i * log p
        # (This function ignores p > 0 because log 0 Does not exist)
        return -np.sum([p * np.log(p) for p in ps if p > 0])

    def _gini(self, y):
        hist = np.bincount(y) # Count the number of times each unique value in y occurs at each index
        ps = hist / len(y) # probability of each unique label y
        # Calculate gini coefficient = 1 - sum of p_i^2 (sum from 1 to total number of labels)
        # (This function ignores p > 0 because log 0 Does not exist)
        return 1 - np.sum([p ** 2 for p in ps])

    # Find the most common label when creating a leaf node in order to figure
    # out the majority class to classify labels accurately.
    def _most_common_label(self, y):
        counter = Counter(y)
        val = counter.most_common(1)[0][0]
        return val

    # Make predictions for feature matrix X by tarversing through the tree created
    def predict(self, X):
        return np.array([self._traverse_tree(x, self.root) for x in X])

    # Helper function to traverse the decision tree
    def _traverse_tree(self, x, node):
        # If current node is a leaf node then return the classification label
        if node.is_leaf_node():
            return node.val
        # If not a leaf node, then traverse recursively to the left child else right child
        if x[node.feat_x] <= node.threshold:
            return self._traverse_tree(x, node.leftchild)
        return self._traverse_tree(x, node.rightchild)
```

Following code chunk completely taken from demo: https://colab.research.google.com/drive/1RKC12-hsxRx7GqogkQjN0ylqFy6ay2As

```
import pandas as pd
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_wine
from sklearn import tree

data = load_wine()
#convert to a dataframe
df = pd.DataFrame(data.data, columns = data.feature_names)
#create the species column
df['Class'] = data.target
#replace this with the actual names
target = np.unique(data.target)
target_names = np.unique(data.target_names)
targets = dict(zip(target, target_names))
df['Class'] = df['Class'].replace(targets)

#extract features and target variables
x = df.drop(columns="Class")
y = df["Class"]
#save the feature name and target variables
feature_names = x.columns
labels = y.unique()
#split the dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x,y,
                                                    test_size = 0.3,
                                                    random_state = 42)


from collections import Counter
from sklearn.preprocessing import LabelEncoder  # Helps encode target labels y into 0 and 1

def load_data(X_train, X_test, y_train, y_test):
    # Convert to numpy and encode labels by converting categorical labels to numeric
    le = LabelEncoder()
    # Convert training and test data to numpy arrays because they were initially pandas arrays.
    return (X_train.to_numpy(), X_test.to_numpy(),
            le.fit_transform(y_train), le.transform(y_test))

# Evaluate the decision tree by calculating accuarcy to compare with sklearn library in demo
def eval_tree(y_true, y_pred, model_name, depth):
    accur_dt = accuracy_score(y_true, y_pred)
    return accur_dt

# Evaluate the decision tree model with splitting criteria conditional entropy
# or gini split condition
def eval_models(X_train, X_test, y_train, y_test, max_depth):
    # Specify common parameters like minimum split condition and max depth of tree
    params = {'min_split': 2, 'max_depth': max_depth, 'random_state': 42}

    # Define models to test and specify the splitting conditions
    models = {
        'Gini(from scratch)': Dec_Tree(split_condition='gini', n_feat=None, min_split=params['min_samples_split'], max_depth=max
        'Conditional Entropy (from scratch)': Dec_Tree(split_condition='cond_ent', n_feat=None, min_split=params['min_samples_sp
    }

    # Train and evaluate each model for above specified splitting condition
    results = {}
    for name, model in models.items():
        model.fit(X_train, y_train)

        # Make predictions for training and test set
        train_pred = model.predict(X_train)
        test_pred = model.predict(X_test)

        # Evaluate the decision tree and store results for displaying prediction accuarcy
        train_acc = eval_tree(y_train, train_pred, f"{name} Train", max_depth)
        test_acc = eval_tree(y_test, test_pred, f"{name} Test", max_depth)
        results[name] = (train_acc, test_acc)

    # Print results
    for name, (train_acc, test_acc) in results.items():
        print(f"\n{name} Results:")
        print(f"Train Accuracy: {train_acc:.8f}")
        print(f"Test Accuracy: {test_acc:.8f}")

    return models

# Prepare the data by passing the parameters required
X_train_np, X_test_np, y_train_enc, y_test_enc = load_data(X_train, X_test,
                                                           y_train, y_test)
```

```
# Evaluate model with max_depth = 1
models_depth_1 = eval_models(X_train_np, X_test_np, y_train_enc, y_test_enc, 1)
```

```
Gini(from scratch) Results:
Train Accuracy: 0.66129032
Test Accuracy: 0.61111111

Conditional Entropy (from scratch) Results:
Train Accuracy: 0.62903226
Test Accuracy: 0.55555556
```

In the above results obtained for a tree with maximum depth = 1 (Underfit), while comparing with the results obtained in sklearn in the demo, the testing accuracy was 0.55 which matches the conditional entropy but not the gini split. Whereas, the training accuracy was 0.62 which also matched the conditional entropy and is very close to 0.66 obtained by the gini coefficient.

```
# Evaluate model with max_depth = 3
models_depth_3 = eval_models(X_train_np, X_test_np, y_train_enc, y_test_enc, 3)
```

```
Gini(from scratch) Results:
Train Accuracy: 0.99193548
Test Accuracy: 0.96296296

Conditional Entropy (from scratch) Results:
Train Accuracy: 0.98387097
Test Accuracy: 0.81481481
```

When we fit the decision tree model with max depth = 3, (The right fit), the sklearn library gave 0.88 and 0.91 for test and training accuracy respectively. Comparing this to the above obtained results, the gini coefficient results are very high for both test and training which is what is desired but they don't match the sklearn results. For conditional entropy, again, they differ from the sklearn results. Hence, we can use the gini split condition for this tree which gives high accuracies indicating that the tree has classified the wine labels with 96% accuarcy.

```
# Evaluate model with max_depth = 5
models_depth_5 = eval_models(X_train_np, X_test_np, y_train_enc, y_test_enc, 5)
```

```
Gini(from scratch) Results:
Train Accuracy: 1.00000000
Test Accuracy: 0.96296296

Conditional Entropy (from scratch) Results:
Train Accuracy: 1.00000000
Test Accuracy: 0.81481481
```

For max depth = 5 (Overfit) we see that the training accuracy for both methods is 1 which was also what was obtained by the sklearn library. This clearly indiactes that our data is overfit. The test accuarcy from sklearn was 0.85. The entropy test accuarcy is close to this. The gini accuracy is very high (0.96) compared to 0.85.

In conclusion, the gini splitting criteria have high test accuracy when the data was fit correctly and when the data was overfit.