Project 3 Report

1. Report Spec (75%)

A. Explain the concept and code detail of your Decision Tree model. (40%)

- `__init__`

  ```
  def __init__(self, max_depth=1, min_samples_split=2, min_samples_leaf=1,
  min_gain_split=0.01):
      self.max_depth = max_depth
      self.max_depth = max_depth
      self.min_samples_split = min_samples_split
      self.min_samples_leaf = min_samples_leaf
      self.min_gain_split = min_gain_split
  ```

  This is the constructor of the Decision Tree model. The parameters:

  - `max_depth`: The maximum depth of the tree. Limits the number of levels in the tree to prevent overfitting.
  - `min_samples_split`: The minimum number of samples required to split an internal node. Helps control the growth of the tree.
  - `min_samples_leaf`: The minimum number of samples required to be at a leaf node. Ensures that leaf nodes have a minimum number of samples.
  - `min_gain_split`: The minimum gain required to make a split. This could be used to control the quality of the splits.

- `fit`

  ```
  def fit(self, X, y):
      if X.shape[0] == 0 or len(y) == 0:
          raise ValueError("Training data cannot be empty.")
      self.tree = self._grow_tree(X, y)
  ```

  This method is used to train the decision the tree model. It checks for empty training data and then calls the `_growtree` method to build the tree using the provided feature matrix and target vector.

- `_grow_tree`

  This method builds the model's decision tree recursively. Note that `X` is the feature matrix, `y` is the target vector and `depth` is the current depth of the tree. The following is the detail explanation of the method.

1. The first line gets the number of samples and the number of features from the shape of `X`.

    ```
    num_samples, num_features = X.shape
    ```

1. The block below checks several shopping conditions, and if any ofthese conditions are met, the method creates a lead node with the most common label in `y` and returns it.
    a. If the current depth is greater than or equal to the maximum depth.
    b. If the number of samples is less than the minimum number of samples required to split.
    c. If the number of samples is less than or equal to the minimum number of samples required to be at a leaf node
    d. If all samples belong to the same class.

        ```
        if (depth >= self.max_depth or num_samples < self.min_samples_split or
            num_samples <= self.min_samples_leaf or len(np.unique(y)) == 1):
            leaf_value = self._most_common_label(y)
            return {"leaf": leaf_value, "idxs": np.arange(num_samples)}
        ```

1. Then, this block calls the `find_best_split` to dint the best feature and threshold to split the data.
    a. If no valid split is found or the information gain from the split is less than the minimum gain required to split, the method creates a leaf node with the most common label in `y` and returns it.

        ```
        best_split = self.find_best_split(X, y)
        if best_split is None or best_split["gain"] < self.min_gain_split:
            leaf_value = self._most_common_label(y)
            return {"leaf": leaf_value, "idxs": np.arange(num_samples)}
        ```

1. This line splits the dat into left and right subsets based on the best feature and threshold found.

    ```
    left_idxs, right_idxs = self.split_dataset(X, y, best_split["feature_index"],
    best_split["threshold"])
    ```

1. Then we recursively grow the tree by calling `_grow_tree` to build the left and right subtrees.

    ```
    left_tree = self._grow_tree(X[left_idxs], y[left_idxs], depth + 1)
    right_tree = self._grow_tree(X[right_idxs], y[right_idxs], depth + 1)
    ```

1. Finally, this block returns a dictionary representing the current node, which includes:
    - `feature_index`: The index of the feature used for the split.
    - `threshold`: The threshold value used for the split.
    - `left`: The left subtree.
    - `right`: The right subtree.
    - `left_idxs`: The indices of the samples in the left subset.
    - `right_idxs`: The indices of the samples in the right subset.

- **`find_best_split`**

  This method is responsible for finding the best feature and threshold to split the data at a given node in the decision tree.

  ```
  def find_best_split(self, X, y):
      best_gain = 0
      best_split = None
      num_samples, num_features = X.shape
      for feature_index in range(num_features):
          thresholds = np.unique(X[:, feature_index])
          for threshold in thresholds:
              gain = self._information_gain(X, y, feature_index, threshold)
              if gain > best_gain:
                  best_gain = gain
                  best_split = {"feature_index": feature_index, "threshold": threshold,
  "gain": gain}
      return best_split
  ```

  We iterate overt the features in the given feature matrix. In each iteration:

  1. Get the unique values (thresholds) for the current feature.
  2. Then we iterate through the thresholds:
      a. `gain = self._information_gain(X, y, feature_index, threshold)` calls the `_information_gain` method to get the information gain for the current feature and threshold.

      b. If the calculated information gain is greater than the current `best_gain`, update `best_gain` and `best_split` with the current feature, threshold and gain.

  1. Return the `best_split` found.

- **`_most_common_label`**

This method is used to determine the most frequent label in an array

```
def _most_common_label(self, y):
    return np.bincount(y).argmax()
```

- `_split_dataset`

  This method is responsible for splitting the dataset into two subsets based on the given feature and threshold.

  ```
  def split_dataset(self, X, y, feature_index, threshold):
      left_idxs = np.where(X[:, feature_index] <= threshold)[0]
      right_idxs = np.where(X[:, feature_index] > threshold)[0]
      return left_idxs, right_idxs
  ```

- `_gini_index`

  ```
  def _gini_index(self, y):
      y = np.ravel(y)
      hist = np.bincount(y)
      ps = hist / len(y)
      return 1.0 - np.sum([p ** 2 for p in ps if p > 0])
  ```

  - This function calculates the Gini index for an array of target labels, `y`
  - `np.bincount(y)` counts occurrences of each label
  - `ps` is the probability of each class in `y`
  - The Gini index formula:

$$\text{Gini Index} = 1 - \sum_{i=1}^{n} (P_i)^2$$

- `_information_gain`

  ```
  def _information_gain(self, X, y, feature_index, threshold):
      parent_gini = self._gini_index(y)
      left_idxs, right_idxs = self.split_dataset(X, y, feature_index, threshold)
      if len(left_idxs) == 0 or len(right_idxs) == 0:
          return 0
      num_samples = len(y)
      num_left, num_right = len(left_idxs), len(right_idxs)
      left_gini = self._gini_index(y[left_idxs])
      right_gini = self._gini_index(y[right_idxs])
      child_gini = (num_left / num_samples) * left_gini + (num_right / num_samples) *
  right_gini
      return parent_gini - child_gini
  ```

  - This function computes the information gain of a split using the Gini index above.

    a. Calculate `parent_gini`, which is the Gini index before splitting.

   b. Split the samples infto left and right groups based on the given `feature_index` and `threshold`.
      i. If either group is empty, it returns 0 as information gain, as no valid split occurs.
      ii. Otherwise, we continue the following steps.
   c. The two Gini indices `left_gini` and `right_gini` are calculated.
   d. The child Gini index, `child_gini`, is computed by weighting the Gini indices of `left_gini` and `right_gini` by their proportions of the total samples.
   e. The information gain is then the difference between `parent_gini` and `child_gini`.

- **`print_tree`**
  - `count_labels`: This function counts the number of labels in the left and right subtrees.
  - For the remaining part, The method recursively traverses the tree and prints the feature index, threshold, and count labels at each node. If the node is a leaf or the maximum print depth is reached, it prints the lead node with the count of labels.
- **`predict`**
  - This method takes a dataset X and returns predictions for each sample by calling `_predict_tree`.
- **`_predict_tree`**
  - This method recursively traverses the decision tree to make a predication for a single sample `x`.
    a. Base Case: If the current `tree_node` is a leaf node, it returns the value of the leaf.
    b. Recursive Case:
       - It retrieves the `feature_index` and `threshold` from the current `tree_node`
       - It checks if the value of the feature at `feature_index` in sample `x` is less than or equal to the `threshold`
       - Based on the comparison, it recursively calls itself with either the left or right child node of the current `tree_node`

B. What is the problem if you use Information Gain to split data in your decision tree model? What is the solution to solve this issue? Please be

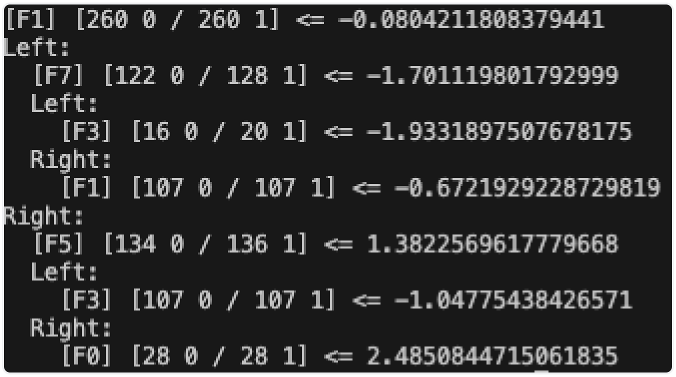
precise and concise. (15%)

The main drawback of using Information Gain to split data in a decision tree model is its bias towards features with high cardinality, as it tends to prioritize splits on features with many unique values regardless of their predictive power. This can lead to overfitting and less generalizable models, as it may choose splits that do not meaningfully improve classification.

To address this, I implemented the Gini Index instead of Information Gain. Unlike Information Gain, the Gini Index reduces bias by focusing on class distribution without favoring features with more values, resulting in more balanced and meaningful splits that improve model accuracy and generalizability.

C. Screenshot and paste your tree structure. (10%)

(Please briefly explain what information you print in each row)



```
[F1] [260 0 / 260 1] <= -0.0804211808379441
Left:
  [F7] [122 0 / 128 1] <= -1.701119801792999
  Left:
    [F3] [16 0 / 20 1] <= -1.9331897507678175
  Right:
    [F1] [107 0 / 107 1] <= -0.6721929228729819
Right:
  [F5] [134 0 / 136 1] <= 1.3822569617779668
  Left:
    [F3] [107 0 / 107 1] <= -1.04775438426571
  Right:
    [F0] [28 0 / 28 1] <= 2.4850844715061835
```

The output represents the structure of the decision tree. In each row, The selected feature, number of 0 and number of 1, and the threshold. Here is an example of the first row, which is the root node.

- root:
  - `[F1]`: The root node splits on feature 1.
  - `[260 0 / 260 1]`: There are 260 samples with label 0 and 260 samples with label 1 at this node.
  - `<= -0.0804211808379441`: The threshold for the split is -0.0804211808379441. Samples with feature 1 values less than or equal to this threshold go to the left subtree, and samples with feature 1 values greater than this threshold go to the right subtree.

D. Discussion (10%)

i. Discuss your strategy to find the best model for the testing data in the project.

   To find the best model for the testing data, I implemented `k_fold_cross_validation` to evaluate and select the model that performs best across multiple training-validation splits as recommended.

```
def k_fold_cross_validation(X, y, k=5):
  fold_size = len(X) // k
  indices = np.arange(len(X))
  np.random.shuffle(indices)
  best_model = None
  best_score = 0

  for i in range(k):
      val_indices = indices[i * fold_size:(i + 1) * fold_size]
      train_indices = np.concatenate([indices[:i * fold_size], indices[(i + 1) * fold_size:]])

      X_train, y_train = X[train_indices], y[train_indices]
      X_val, y_val = X[val_indices], y[val_indices]

      model = DecisionTreeClassifier(max_depth=10)
      model.fit(X_train, y_train)
      model.post_prune(X_val, y_val)
      val_predictions = model.predict(X_val)
      score = calculate_metrics(val_predictions, y_val)

      if score > best_score:
          best_score = score
          best_model = model

    return best_model, best_score
```

1. First, I shuffle the data indices to ensure that each fold represents a different subset of the dataset. This enhances the robustness of the model evaluation by reducing any potential bias caused by data ordering.
2. I split the shuffled data into `k` folds (in this case, `k=5`), where each fold will be used as a validation set once, while the remaining `k-1` folds serve as the training data. This rotation through the data provides a more comprehensive view of model performance on different data subsets.
3. For each fold, I train a `DecisionTreeClassifier` with a maximum depth of 10. After training on the `k-1` folds, I use the remaining fold for validation.
4. During each validation, I apply post-pruning on the decision tree. This step further prevents overfitting by trimming branches that don't improve the validation accuracy.
5. I predict the labels for the validation set and calculate performance metrics (like accuracy or other measures) to evaluate the model's effectiveness.
6. The model with the highest validation score across all folds is saved as the `best_model`. This model is the most likely to generalize well to the testing data, as it has consistently performed well across various training-validation splits.

Reference: https://ithelp.ithome.com.tw/articles/10279240

E. Bonus (10%)

i. Pre-pruning: In the constructor, the parameters restrict the tree growth.

```
def __init__(self, max_depth=1, min_samples_split=2, min_samples_leaf=1,
min_gain_split=0.01):
    self.max_depth = max_depth
    self.max_depth = max_depth
    self.min_samples_split = min_samples_split
    self.min_samples_leaf = min_samples_leaf
    self.min_gain_split = min_gain_split

model = DecisionTreeClassifier(max_depth=10)
```

This would:

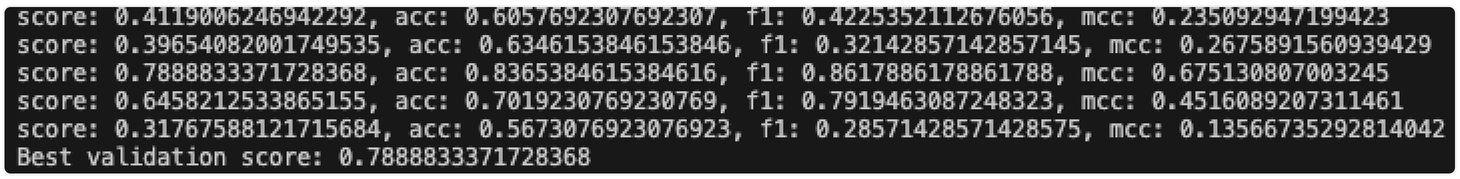
- Limit the tree depth to 10 levels (`max_depth=10`).
- Require at least 2 samples to split an internal node (`min_samples_split=2`).
- Require at least 1 samples in each leaf node (`min_samples_leaf=1`).
- Only split if it reduces impurity by at least 0.01 (`min_impurity_decrease=0.01`).

i. Post-pruning

```
def post_prune(self, X_val, y_val):
    def prune_tree(tree, X_val, y_val):
        if "leaf" in tree:
            return tree

        left_tree = prune_tree(tree["left"], X_val, y_val)
        right_tree = prune_tree(tree["right"], X_val, y_val)

        if "leaf" in left_tree and "leaf" in right_tree:
            left_idxs, right_idxs = self.split_dataset(X_val, y_val,
tree["feature_index"], tree["threshold"])
            y_val_left = y_val[left_idxs]
            y_val_right = y_val[right_idxs]

            error_before_pruning = np.sum(y_val_left != left_tree["leaf"]) +
np.sum(y_val_right != right_tree["leaf"])
            leaf_value = self._most_common_label(np.concatenate([y_val_left,
y_val_right]))
            error_after_pruning = np.sum(y_val != leaf_value)

            if error_after_pruning <= error_before_pruning:
                return {"leaf": leaf_value}
        return {"feature_index": tree["feature_index"], "threshold": tree["threshold"],
"left": left_tree, "right": right_tree}
    self.tree = prune_tree(self.tree, X_val, y_val)
```

- Purpose: Remove certain branches in the decision tree if doing so does not significantly increase the classification error, which means we can simplify the model.
- Details:
  - `prune_tree` traversed through the decision tree recursively and determine whether to prune the specific branches based on the validation dataset.
  - If the current node is a lead, it returns the node as-is.
  - Otherwise, `prune_tree` is called recursively on both the left and right children.
  - After traversing to the left and right children, `prune_tree` checks if both children are leaves. If so, it evaluates whether merging the two leaves would reduce or maintain accuracy on the validation set.
  - The function uses `split_dataset` to split the validation data and labels into `y_val_left` and `y_val_right`
  - `error_before_pruning`: The classification error if the branches are left as they are, computed as the sum of misclassifications in the left and right validation sets.
  - `error_after_pruning`: The classification error if the branches are pruned. The leaf value that would replace the two nodes is set to the most common label of `y_val_left` and `y_val_right` combined, calculated using `_most_common_label`.
  - If `error_after_pruning` is less than or equal to `error_before_pruning`, the function returns a simplified tree node, now converted to a leaf with the most common label.
  - If pruning is not beneficial, the function retains the original branches and returns the unmodified node.
  - Finally, `self.tree = prune_tree(self.tree, X_val, y_val)` applies pruning to the entire tree, updating `self.tree` with the pruned version.

2. Performance (25%)



```
score: 0.4119006246942292, acc: 0.6057692307692307, f1: 0.4225352112676056, mcc: 0.235092947199423
score: 0.39654082001749535, acc: 0.6346153846153846, f1: 0.32142857142857145, mcc: 0.2675891560939429
score: 0.7888833371728368, acc: 0.8365384615384616, f1: 0.8617886178861788, mcc: 0.675130807003245
score: 0.6458212533865155, acc: 0.7019230769230769, f1: 0.7919463087248323, mcc: 0.4516089207311461
score: 0.31767588121715684, acc: 0.5673076923076923, f1: 0.28571428571428575, mcc: 0.13566735292814042
Best validation score: 0.7888833371728368
```

Since the score of each training model affects my choice of model, I evaluated performance by implementing the score calculation myself:

```
def calculate_metrics(pred, test_y):
    correct_predictions = np.sum(test_y == pred)
    total_predictions = len(test_y)
    acc = correct_predictions / total_predictions

    tp = np.sum((test_y == 1) & (pred == 1))
    tn = np.sum((test_y == 0) & (pred == 0))
    fp = np.sum((test_y == 0) & (pred == 1))
    fn = np.sum((test_y == 1) & (pred == 0))

    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    mcc_numerator = (tp * tn) - (fp * fn)
    mcc_denominator = np.sqrt((tp + fp) * (tp + fn) * (tn + fp) * (tn + fn))
    mcc = mcc_numerator / mcc_denominator if mcc_denominator > 0 else 0

    score = 0.3 * acc + 0.35 * f1 + 0.35 * mcc
    print(f'score: {score}, acc: {acc}, f1: {f1}, mcc: {mcc}')
    return score
```

- reference:
  - https://www.v7labs.com/blog/f1-score-guide
  - https://medium.com/@TheDataScience-ProF/understanding-the-matthews-correlation-coefficient-mcc-in-machine-learning-26e8049f8572

3. Other Notes

Project Structure

```
| HW3
|--| proj3_data
|----| train_X.csv
|----| train_y.csv
|----| test_X.csv
|----| predict_y.csv
|--| pycode
|----| main.py
|----| model.py
|----| preproceessor.py
```

Note that if you change the structure, the root path in `dataPreprocessing` would need to change in order to read the given data. Also, change `pd.DataFrame(predict_y, columns=['label']).to_csv('../proj3_data/predict_y.csv', index=True)` if you want to save the predicted csv file.

Reference

a. https://medium.com/@enozeren/building-a-decision-tree-from-scratch-324b9a5ed836