Assignment 1

Name: Pola Gnana Shekar

Roll No: 21CS10052

In [1]:
```
import pandas as pd
```

In [2]:
```
df=pd.read_csv('../../dataset/decision-tree.csv')
print(df.shape)
```

```
(768, 9)
```

In [2]:
```
import pandas as pd
import numpy as np
from collections import Counter

# Load the dataset
data_path = '../../dataset/decision-tree.csv'
df = pd.read_csv(data_path)

# Convert the DataFrame to a numpy array
data = df.values

# Split the data into features (X) and target (y)
X = data[:, :-1]  # All columns except the last one
y = data[:, -1]   # Last column

# Set a random seed for reproducibility
np.random.seed(42)

# Shuffle the indices
shuffled_indices = np.random.permutation(len(data))

# Define the ratio for train-test split
train_ratio = 0.8
train_size = int(len(data) * train_ratio)

# Split the shuffled indices into training and testing indices
train_indices = shuffled_indices[:train_size]
test_indices = shuffled_indices[train_size:]

# Split the data into training and testing sets
X_train, y_train = X[train_indices], y[train_indices]
X_test, y_test = X[test_indices], y[test_indices]

# Print the sizes of the training and testing samples
print(f"Training sample size: {len(X_train)}")
print(f"Testing sample size: {len(X_test)}")
```

```
Training sample size: 614
Testing sample size: 154
```

In [1]:
```
import pandas as pd
import numpy as np
from collections import Counter

class DecisionTree:
    def __init__(self, min_samples_split=10):
        self.min_samples_split = min_samples_split
        self.tree = None

    def fit(self, X, y, feature_names):
        self.feature_names = feature_names
        self.tree = self._build_tree(X, y)

    def _calculate_entropy(self, y):
        counts = np.bincount(y)
        probs = counts / len(y)
        return -np.sum(probs * np.log2(probs + 1e-6))

    def _calculate_information_gain(self, X, y, feature_index):
        parent_entropy = self._calculate_entropy(y)

        unique_values = np.unique(X[:, feature_index])
        weighted_child_entropies = 0

        for value in unique_values:
            subset_indices = X[:, feature_index] == value
            child_entropy = self._calculate_entropy(y[subset_indices])
            weight = len(y[subset_indices]) / len(y)
            weighted_child_entropies += weight * child_entropy

        information_gain = parent_entropy - weighted_child_entropies
        return information_gain

    def _build_tree(self, X, y):
        if len(y) <= self.min_samples_split or len(np.unique(y)) == 1:
            return Counter(y).most_common(1)[0][0]

        num_features = X.shape[1]
        information_gains = [self._calculate_information_gain(X, y, feature_ir
        best_feature = np.argmax(information_gains)

        if information_gains[best_feature] == 0:
            return Counter(y).most_common(1)[0][0]

        subtree = {self.feature_names[best_feature]: {}}
        unique_values = np.unique(X[:, best_feature])

        for value in unique_values:
            subset_indices = X[:, best_feature] == value
            subtree[self.feature_names[best_feature]][value] = self._build_tre

        return subtree

    def predict(self, X):
        return np.array([self._predict_sample(sample, self.tree) for sample ir

    def _predict_sample(self, sample, node):
```

```
        if isinstance(node, dict):
            feature_name = list(node.keys())[0]
            feature_index = self.feature_names.index(feature_name)
            value = sample[feature_index]
            if value in node[feature_name]:
                return self._predict_sample(sample, node[feature_name][value])
            else:
                return Counter(sample).most_common(1)[0][0]
        else:
            return node

# Load the dataset
data_path = '../../dataset/decision-tree.csv'
df = pd.read_csv(data_path)

# Convert the DataFrame to a numpy array
data = df.values

# Split the data into features (X) and target (y)
X = data[:, :-1]  # All columns except the last one
y = data[:, -1]   # Last column

# Set a random seed for reproducibility
np.random.seed(28)

# Shuffle the indices
shuffled_indices = np.random.permutation(len(data))

# Define the ratio for train-test split
train_ratio = 0.8
train_size = int(len(data) * train_ratio)

# Split the shuffled indices into training and testing indices
train_indices = shuffled_indices[:train_size]
test_indices = shuffled_indices[train_size:]

# Split the data into training and testing sets
X_train, y_train = X[train_indices], y[train_indices]
X_test, y_test = X[test_indices], y[test_indices]

# Convert y_train to integers
y_train = y_train.astype(int)

# Use the DecisionTree class for building and evaluating the decision tree
# feature_names = df.columns[:-1]  # Excluding the target column
# Convert feature_names to a list
# feature_names_list = list(feature_names)
feature_names = df.columns[:-1].tolist()
dt = DecisionTree(min_samples_split=10)
dt.fit(X_train, y_train, feature_names)

# Evaluate the decision tree on the test set
y_pred = dt.predict(X_test)

# Calculate accuracy or other metrics to evaluate the model's performance
accuracy = np.mean(y_pred == y_test)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.42207792207792205
```

In [29]:
```
import pandas as pd
import numpy as np
from collections import Counter
import matplotlib.pyplot as plt
from sklearn.metrics import precision_score, recall_score, accuracy_score

class DecisionTree:
    def __init__(self, min_samples_split=10):
        self.min_samples_split = min_samples_split
        self.tree = None

    def fit(self, X, y, feature_names):
        self.feature_names = feature_names
        self.tree = self._build_tree(X, y)

    def _calculate_entropy(self, y):
        counts = np.bincount(y)
        probs = counts / len(y)
        return -np.sum(probs * np.log2(probs + 1e-6))

    def _calculate_information_gain(self, X, y, feature_index):
        parent_entropy = self._calculate_entropy(y)

        unique_values = np.unique(X[:, feature_index])
        weighted_child_entropies = 0

        for value in unique_values:
            subset_indices = X[:, feature_index] == value
            child_entropy = self._calculate_entropy(y[subset_indices])
            weight = len(y[subset_indices]) / len(y)
            weighted_child_entropies += weight * child_entropy

        information_gain = parent_entropy - weighted_child_entropies
        return information_gain

    def _build_tree(self, X, y):
        if len(y) <= self.min_samples_split or len(np.unique(y)) == 1:
            return Counter(y).most_common(1)[0][0]

        num_features = X.shape[1]
        information_gains = [self._calculate_information_gain(X, y, feature_ir
        best_feature = np.argmax(information_gains)

        if information_gains[best_feature] == 0:
            return Counter(y).most_common(1)[0][0]

        subtree = {self.feature_names[best_feature]: {}}
        unique_values = np.unique(X[:, best_feature])

        for value in unique_values:
            subset_indices = X[:, best_feature] == value
            subtree[self.feature_names[best_feature]][value] = self._build_tre

        return subtree

    def predict(self, X):
        return np.array([self._predict_sample(sample, self.tree) for sample ir
```

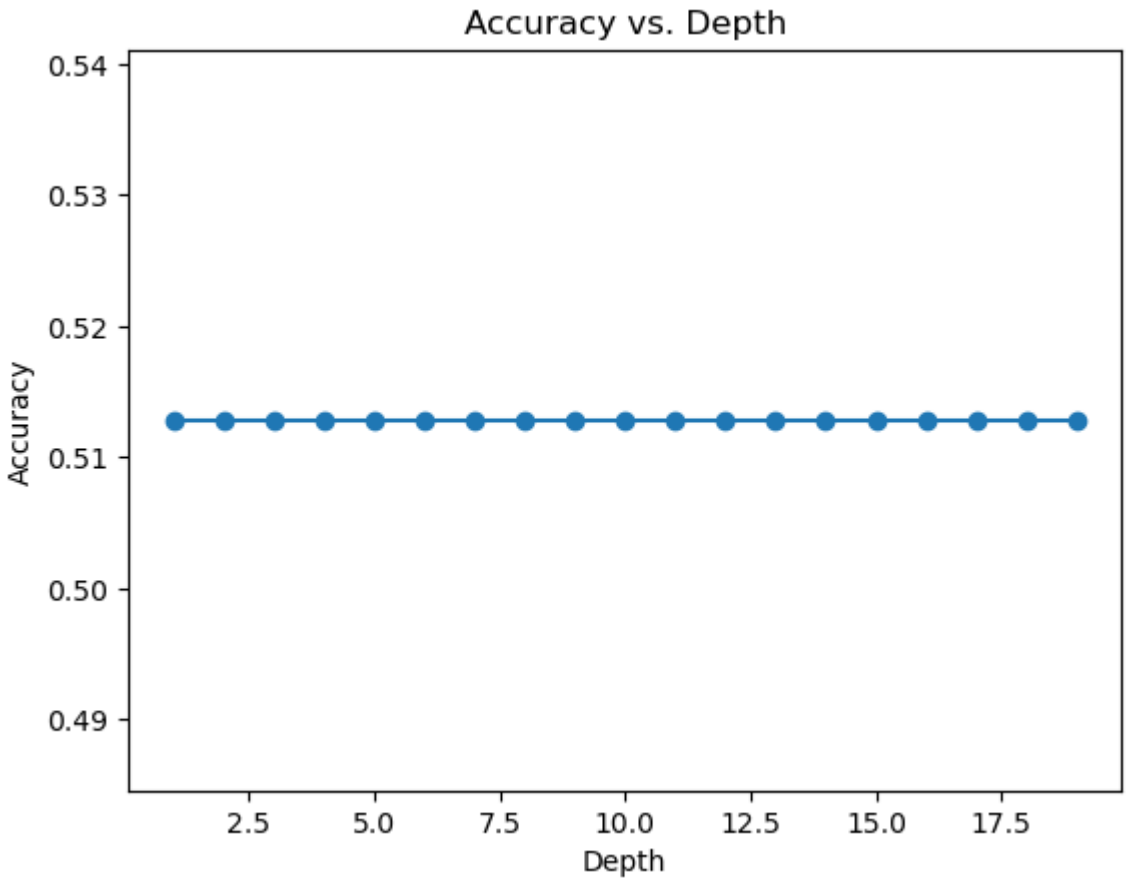
```
    def _predict_sample(self, sample, node):
        if isinstance(node, dict):
            feature_name = list(node.keys())[0]
            feature_index = self.feature_names.index(feature_name)
            value = sample[feature_index]
            if value in node[feature_name]:
                return self._predict_sample(sample, node[feature_name][value])
            else:
                return Counter(sample).most_common(1)[0][0]
        else:
            return node

    def _prune_tree(self, node, X_val, y_val):
        if isinstance(node, dict):
            for value, subtree in node[list(node.keys())[0]].items():
                node[list(node.keys())[0]][value] = self._prune_tree(subtree,

            if self._is_leaf(node):
                original_accuracy = self._calculate_accuracy(X_val, y_val)
                node_accuracy = self._calculate_accuracy(X_val, y_val, node)

                if node_accuracy >= original_accuracy:
                    return node[list(node.keys())[0]]

        return node

    def prune(self, X_val, y_val):
        self.tree = self._prune_tree(self.tree, X_val, y_val)

    def _is_leaf(self, node):
        return not isinstance(node[list(node.keys())[0]], dict)

    def _calculate_accuracy(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.mean(y_pred == y)
        return accuracy

    def _reduced_error_prune(self, node, X_val, y_val):
        if isinstance(node, dict):
            for value, subtree in node[list(node.keys())[0]].items():
                node[list(node.keys())[0]][value] = self._reduced_error_prune(

            if not self._is_leaf(node):
                original_accuracy = self._calculate_accuracy(X_val, y_val)
                pruned_node = self._prune_node(node, X_val, y_val)
                pruned_accuracy = self._calculate_accuracy(X_val, y_val)

                if pruned_accuracy >= original_accuracy:
                    return pruned_node

        return node

    def _prune_node(self, node, X_val, y_val):
        pruned_nodes = []

        for value, subtree in node[list(node.keys())[0]].items():
```

```
            pruned_subtree = self._prune_tree(subtree, X_val, y_val)
            pruned_nodes.append((value, pruned_subtree))

        majority_value = Counter(y_val).most_common(1)[0][0]
        pruned_node = {list(node.keys())[0]: {value: subtree for value, subtre

        return pruned_node

    def evaluate_metrics(self, y_true, y_pred):
        accuracy = accuracy_score(y_true, y_pred)
        precision = precision_score(y_true, y_pred, average='macro')
        recall = recall_score(y_true, y_pred, average='macro', zero_division=1
        return accuracy, precision, recall

# Load the dataset
data_path = '../../dataset/decision-tree.csv'
df = pd.read_csv(data_path)

# Convert the DataFrame to a numpy array
data = df.values

# Split the data into features (X) and target (y)
X = data[:, :-1]  # All columns except the last one
y = data[:, -1]   # Last column

# Set a random seed for reproducibility
np.random.seed(28)

# Shuffle the indices
shuffled_indices = np.random.permutation(len(data))


# Define the ratios for train-validation-test split
train_ratio = 0.8
val_ratio = 0.1
test_ratio = 0.1
train_size = int(len(data) * train_ratio)
val_size = int(len(data) * val_ratio)
test_size = len(data) - train_size - val_size

# Split the shuffled indices into training, validation, and testing indices
train_indices = shuffled_indices[:train_size]
val_indices = shuffled_indices[train_size : train_size + val_size]
test_indices = shuffled_indices[train_size + val_size :]

# Split the data into training, validation, and testing sets
X_train, y_train = X[train_indices], y[train_indices]
X_val, y_val = X[val_indices], y[val_indices]
X_test, y_test = X[test_indices], y[test_indices]

# Convert y_train to integers
y_train = y_train.astype(int)

# Use the DecisionTree class for building and evaluating the decision tree
# Convert feature_names to a list
# feature_names_list = list(feature_names)
feature_names = df.columns[:-1].tolist()
```

```
dt = DecisionTree(min_samples_split=10)
dt.fit(X_train, y_train, feature_names)

# # Evaluate the decision tree on the validation set
# y_val_pred = dt.predict(X_val)

# Evaluate the decision tree on the test set
y_pred = dt.predict(X_test)

# Calculate accuracy or other metrics to evaluate the model's performance
accuracy = np.mean(y_pred == y_test)
print("Accuracy:", accuracy)

# Perform reduced error pruning
dt_pruned = DecisionTree(min_samples_split=10)
dt_pruned.fit(X_train, y_train, feature_names)

val_accuracies = []

significant_improvement_threshold = 0.001  # You can adjust this threshold as

while True:
    original_accuracy = dt_pruned._calculate_accuracy(X_val, y_val)
    pruned_tree = dt_pruned._reduced_error_prune(dt_pruned.tree, X_val, y_val)
    pruned_accuracy = dt_pruned._calculate_accuracy(X_val, y_val)

    if pruned_accuracy - original_accuracy >= significant_improvement_thresho
        dt_pruned.tree = pruned_tree
        val_accuracies.append(pruned_accuracy)
    else:
        break

# Evaluate the pruned decision tree on the test set
y_test_pred = dt_pruned.predict(X_test)

test_accuracy = np.mean(y_test_pred == y_test)
print("Test Accuracy after Pruning:", test_accuracy)


depths = list(range(1, 20))
accuracies = []

for depth in depths:
    dt_pruned = DecisionTree(min_samples_split=10)
    dt_pruned.fit(X_train, y_train, feature_names)
    dt_pruned.prune(X_val, y_val)
    accuracy = dt_pruned._calculate_accuracy(X_test, y_test)
    accuracies.append(accuracy)
    print(f"Accuracy after pruning (depth {depth}): {accuracy:.8f}")

plt.plot(depths, accuracies, marker='o')
plt.xlabel('Depth')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Depth')
plt.show()

# Calculate and report the mean macro accuracy, macro precision, and macro rec
```

```
test_accuracy, test_precision, test_recall = dt_pruned.evaluate_metrics(y_test

print("Test Accuracy:", test_accuracy)
print("Test Macro Precision:", test_precision)
print("Test Macro Recall:", test_recall)

# Print the pruned tree
print("Pruned Decision Tree:")
print(dt_pruned.tree)
```

```
Accuracy: 0.5128205128205128
Test Accuracy after Pruning: 0.5128205128205128
Accuracy after pruning (depth 1): 0.51282051
Accuracy after pruning (depth 2): 0.51282051
Accuracy after pruning (depth 3): 0.51282051
Accuracy after pruning (depth 4): 0.51282051
Accuracy after pruning (depth 5): 0.51282051
Accuracy after pruning (depth 6): 0.51282051
Accuracy after pruning (depth 7): 0.51282051
Accuracy after pruning (depth 8): 0.51282051
Accuracy after pruning (depth 9): 0.51282051
Accuracy after pruning (depth 10): 0.51282051
Accuracy after pruning (depth 11): 0.51282051
Accuracy after pruning (depth 12): 0.51282051
Accuracy after pruning (depth 13): 0.51282051
Accuracy after pruning (depth 14): 0.51282051
Accuracy after pruning (depth 15): 0.51282051
Accuracy after pruning (depth 16): 0.51282051
Accuracy after pruning (depth 17): 0.51282051
Accuracy after pruning (depth 18): 0.51282051
Accuracy after pruning (depth 19): 0.51282051
```


Accuracy vs. Depth
0.54
0.53
0.52
0.51
0.50
0.49
Accuracy
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Depth

```
Test Accuracy: 0.5128205128205128
Test Macro Precision: 0.0961073500967118
Test Macro Recall: 0.896043771043771
Pruned Decision Tree:
{'DiabetesPedigreeFunction': {0.078: 0, 0.084: 0, 0.085: 0, 0.088: 1, 0.089:
0, 0.096: 0, 0.1: 0, 0.101: 0, 0.102: 0, 0.107: 0, 0.108: 0, 0.115: 0, 0.11
8: 0, 0.122: 0, 0.123: 0, 0.126: 0, 0.127: 1, 0.128: 1, 0.129: 1, 0.13: 0,
0.133: 0, 0.134: 0, 0.135: 1, 0.136: 0, 0.137: 1, 0.14: 0, 0.141: 1, 0.142:
0, 0.143: 0, 0.147: 0, 0.148: 0, 0.149: 0, 0.15: 0, 0.151: 1, 0.153: 0, 0.15
4: 0, 0.155: 0, 0.158: 0, 0.159: 0, 0.161: 1, 0.164: 0, 0.165: 1, 0.166: 0,
0.167: 0, 0.17: 0, 0.171: 0, 0.173: 0, 0.174: 0, 0.175: 0, 0.176: 0, 0.177:
0, 0.178: 1, 0.179: 0, 0.18: 0, 0.181: 0, 0.183: 1, 0.187: 0, 0.188: 0, 0.18
9: 0, 0.19: 0, 0.191: 0, 0.192: 0, 0.194: 0, 0.196: 1, 0.197: 0, 0.198: 0,
0.199: 1, 0.2: 0, 0.203: 0, 0.204: 0, 0.205: 1, 0.206: 0, 0.207: 0, 0.209:
1, 0.21: 0, 0.212: 1, 0.215: 0, 0.217: 0, 0.218: 0, 0.219: 1, 0.221: 0, 0.22
3: 0, 0.225: 0, 0.227: 1, 0.229: 0, 0.23: 0, 0.231: 0, 0.232: 1, 0.233: 0,
0.234: 0, 0.235: 0, 0.236: 0, 0.237: 0, 0.238: 0, 0.239: 1, 0.24: 1, 0.241:
1, 0.243: 0, 0.244: 0, 0.245: 0, 0.246: 0, 0.247: 0, 0.248: 1, 0.249: 0, 0.2
51: 0, 0.252: 0, 0.253: 0, 0.254: 1, 0.256: 0, 0.257: 1, 0.258: 0, 0.259: 0,
0.26: 0, 0.261: 0, 0.262: 0, 0.263: 0, 0.264: 1, 0.265: 0, 0.267: 0, 0.268:
0, 0.269: 0, 0.27: 1, 0.271: 0, 0.272: 1, 0.277: 1, 0.278: 1, 0.28: 0, 0.28
2: 0, 0.283: 0, 0.284: 0, 0.285: 0, 0.286: 0, 0.289: 0, 0.29: 0, 0.292: 0,
0.293: 0, 0.294: 0, 0.295: 0, 0.296: 1, 0.297: 1, 0.299: 0, 0.3: 0, 0.302:
1, 0.304: 0, 0.305: 0, 0.306: 0, 0.313: 0, 0.314: 0, 0.315: 0, 0.318: 0, 0.3
19: 1, 0.323: 0, 0.324: 0, 0.325: 1, 0.326: 0, 0.328: 1, 0.329: 0, 0.33: 0,
0.331: 1, 0.332: 0, 0.334: 0, 0.335: 1, 0.336: 0, 0.337: 0, 0.338: 0, 0.34:
0, 0.341: 0, 0.342: 0, 0.343: 0, 0.344: 1, 0.345: 1, 0.346: 1, 0.347: 0, 0.3
49: 0, 0.351: 0, 0.356: 1, 0.358: 1, 0.361: 1, 0.362: 0, 0.364: 0, 0.365: 1,
0.368: 0, 0.37: 0, 0.371: 1, 0.374: 0, 0.376: 1, 0.378: 1, 0.38: 1, 0.381:
0, 0.382: 0, 0.385: 0, 0.388: 0, 0.389: 0, 0.391: 0, 0.394: 0, 0.395: 1, 0.3
96: 0, 0.398: 1, 0.399: 0, 0.4: 0, 0.401: 0, 0.402: 1, 0.403: 1, 0.404: 0,
0.407: 0, 0.408: 1, 0.409: 0, 0.411: 0, 0.412: 1, 0.415: 0, 0.416: 0, 0.417:
0, 0.419: 0, 0.42: 0, 0.421: 0, 0.422: 0, 0.423: 1, 0.426: 0, 0.427: 0, 0.4
3: 0, 0.431: 1, 0.433: 1, 0.434: 0, 0.435: 1, 0.439: 0, 0.443: 0, 0.444: 0,
0.446: 0, 0.451: 1, 0.452: 0, 0.454: 0, 0.455: 0, 0.457: 0, 0.46: 0, 0.463:
0, 0.464: 0, 0.465: 1, 0.466: 0, 0.467: 1, 0.471: 0, 0.472: 0, 0.479: 1, 0.4
82: 0, 0.483: 0, 0.484: 1, 0.485: 0, 0.488: 0, 0.491: 0, 0.493: 0, 0.495: 0,
0.496: 0, 0.497: 0, 0.499: 0, 0.501: 0, 0.502: 1, 0.503: 1, 0.507: 0, 0.509:
0, 0.51: 1, 0.512: 0, 0.514: 1, 0.516: 1, 0.52: 1, 0.525: 0, 0.526: 0, 0.52
7: 0, 0.528: 1, 0.529: 1, 0.532: 0, 0.534: 1, 0.536: 0, 0.537: 1, 0.539: 1,
0.542: 1, 0.543: 1, 0.545: 0, 0.546: 0, 0.547: 0, 0.549: 1, 0.551: 1, 0.557:
0, 0.559: 0, 0.56: 0, 0.561: 0, 0.564: 0, 0.565: 1, 0.571: 0, 0.572: 0, 0.57
5: 1, 0.58: 0, 0.582: 0, 0.583: 1, 0.586: 0, 0.587: 0, 0.588: 1, 0.591: 0,
0.593: 1, 0.597: 0, 0.598: 0, 0.6: 0, 0.601: 0, 0.605: 0, 0.607: 0, 0.612:
0, 0.613: 1, 0.615: 1, 0.624: 0, 0.626: 0, 0.627: 1, 0.629: 0, 0.63: 1, 0.63
1: 0, 0.637: 1, 0.64: 1, 0.645: 1, 0.646: 1, 0.647: 0, 0.652: 1, 0.654: 0,
0.658: 0, 0.66: 1, 0.661: 1, 0.665: 1, 0.673: 0, 0.674: 0, 0.677: 0, 0.678:
0, 0.682: 1, 0.686: 0, 0.687: 0, 0.692: 1, 0.693: 1, 0.695: 0, 0.696: 0, 0.6
99: 0, 0.702: 1, 0.703: 0, 0.704: 0, 0.705: 0, 0.717: 0, 0.718: 0, 0.722: 1,
0.725: 0, 0.727: 0, 0.731: 1, 0.733: 0, 0.734: 1, 0.738: 0, 0.741: 1, 0.742:
1, 0.745: 1, 0.748: 0, 0.757: 1, 0.761: 1, 0.766: 0, 0.767: 0, 0.771: 1, 0.7
85: 1, 0.787: 1, 0.801: 0, 0.803: 1, 0.804: 0, 0.805: 1, 0.808: 1, 0.813: 0,
0.816: 0, 0.817: 1, 0.821: 0, 0.825: 1, 0.828: 0, 0.831: 1, 0.833: 0, 0.839:
1, 0.84: 0, 0.845: 0, 0.851: 1, 0.855: 1, 0.856: 0, 0.871: 1, 0.874: 0, 0.87
5: 1, 0.878: 0, 0.88: 0, 0.881: 0, 0.886: 0, 0.892: 0, 0.904: 0, 0.905: 1,
0.917: 0, 0.925: 1, 0.926: 1, 0.93: 0, 0.932: 0, 0.933: 1, 0.944: 0, 0.947:
0, 0.949: 0, 0.955: 1, 0.956: 1, 0.962: 1, 0.966: 0, 0.968: 1, 0.997: 0, 1.0
```

```
21: 0, 1.057: 1, 1.072: 1, 1.076: 0, 1.095: 0, 1.096: 0, 1.101: 0, 1.114: 1,
1.127: 1, 1.136: 1, 1.138: 0, 1.144: 1, 1.154: 1, 1.159: 0, 1.174: 0, 1.182:
1, 1.191: 1, 1.222: 1, 1.224: 1, 1.268: 0, 1.282: 1, 1.318: 1, 1.39: 1, 1.39
1: 1, 1.394: 1, 1.4: 0, 1.441: 0, 1.461: 0, 1.476: 0, 1.6: 0, 1.698: 0, 1.69
9: 0, 1.731: 0, 1.781: 0, 2.137: 1, 2.288: 1, 2.329: 0, 2.42: 1}}
```