decision_trees

March 3, 2024

0.1 Problem 4.1

[28]:
```
import numpy as np
import pandas as pd

col_names = ['survived', 'pclass', 'sex', 'age', 'siblings_spouse',␣
↪'parents_children', 'fare']
data = pd.read_csv("titanic_data.csv", skiprows=1, header=None, names=col_names)

data["first_class"] = (data["pclass"] == 1).astype(int)
data["isFemale"] = data["sex"]
data["isChild"] = (data['age'] < 18).astype(int)
data["sib_sp_present"] = (data["siblings_spouse"] > 0).astype(int)
data["par_chi_present"] = (data["parents_children"] > 0).astype(int)
data["highFare"] = (data["fare"] > data["fare"].median()).astype(int)

data.drop(['age','sex','pclass', 'siblings_spouse', 'parents_children','fare'],␣
↪axis=1, inplace=True)
new_col_names = ["first_class", "isFemale", "isChild", "sib_sp_present",␣
↪"par_chi_present", "highFare"]
data.head(5)
```

[28]:

	survived	first_class	isFemale	isChild	sib_sp_present	par_chi_present	highFare
0	0	0	0	0	1	0	0
1	1	1	1	0	1	0	1
2	1	0	1	0	0	0	0
3	1	1	1	0	1	0	1
4	0	0	0	0	0	0	0

0.2 Problem 4.2

[29]:
```
def entropy(y):
    unique, count = np.unique(y, return_counts=True, axis=0)
    prob = count/len(y)
    H = np.sum((-1) * prob * np.log2(prob))
    return H

def cond_entropy(y, X):
    return entropy(np.c_[y,X]) - entropy(X)

def mutual_information(y, X):
    return entropy(y) - cond_entropy(y,X)

data_np = data.to_numpy()
X_np = data_np[:, 1:]
y_np = data_np[:, 0]

for i in range(6):
    print(mutual_information(y_np, X_np[:, i]))
```

```
0.057274865894062166
0.21684950483126542
0.006697930333195434
0.009236225402886045
0.015040080377706544
0.05510153466815071
```

0.3 Problem 4.3

[30]:
```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None,␣
 ↪right=None, info_gain=None, value=None):
        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain

        # for leaf node
        self.value = value
```

[31]:
```
# Much of this code is courtesy of "Decision Tree Classification in Python␣
 ↪(from scratch!)"" by Normalized Nerd on YouTube.
# I watched through his YT series to learn how to code this, so it primariliy␣
 ↪comes from his repo
```

```
# I really struggled porting over the entropy and mutual information code from␣
 ↪4.2 into creating a decision tree, so
# I went with his versions, which were more robust to a decision tree

import graphviz

class DecisionTreeClassifier():
    def __init__(self, min_samples_split=2, max_depth=2):
        # initialize the root of the tree
        self.root = None

        # stopping conditions
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)

        # split until stopping conditions are met
        if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
            # find the best split
            best_split = self.get_best_split(dataset, num_samples, num_features)
            # check if information gain is positive
            if best_split["info_gain"]>0:
                # recur left
                left_subtree = self.build_tree(best_split["dataset_left"],␣
 ↪curr_depth+1)
                # recur right
                right_subtree = self.build_tree(best_split["dataset_right"],␣
 ↪curr_depth+1)
                # return decision node
                return Node(best_split["feature_index"],␣
 ↪best_split["threshold"],
                            left_subtree, right_subtree,␣
 ↪best_split["info_gain"])

        # compute leaf node
        leaf_value = self.calculate_leaf_value(Y)
        # return leaf node
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        # dictionary to store the best split
        best_split = {}
        max_info_gain = -float("inf")
```

```
        # loop over all the features
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            # loop over all the feature values present in the data
            for threshold in possible_thresholds:
                # get current split
                dataset_left, dataset_right = self.split(dataset,␣
↪feature_index, threshold)
                # check if childs are not null
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:, -1],␣
↪dataset_right[:, -1]
                    # compute information gain
                    curr_info_gain = self.information_gain(y, left_y, right_y,␣
↪"entropy")
                    # update the best split if needed
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain

        # return best split
        return best_split

    def split(self, dataset, feature_index, threshold):
        dataset_left = np.array([row for row in dataset if␣
↪row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if␣
↪row[feature_index]>threshold])
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        if mode=="gini":
            gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child)␣
↪+ weight_r*self.gini_index(r_child))
        else:
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child) +␣
↪weight_r*self.entropy(r_child))
        return gain
```

```
    def entropy(self, y):
        unique, count = np.unique(y, return_counts=True, axis=0)
        prob = count/len(y)
        H = np.sum((-1) * prob * np.log2(prob))
        return H

    def gini_index(self, y):
        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

    def calculate_leaf_value(self, Y):
        Y = list(Y)
        return max(Y, key=Y.count)

    def print_tree(self, tree=None, feature_names=None):
        dot = graphviz.Digraph()

        if tree is None:
            tree = self.root

        if feature_names is None:
            feature_names = ["Feature " + str(i) for i in␣
↪range(len(new_col_names))]

        def add_nodes_edges(tree, dot=None):
            # Create node
            if isinstance(tree, Node) and tree.value is not None:
                dot.node(str(id(tree)), str(tree.value), shape='oval')
            elif isinstance(tree, Node):
                dot.node(str(id(tree)), feature_names[tree.feature_index],␣
↪shape='box')

            # Add children
            if isinstance(tree, Node) and tree.left is not None:
                add_nodes_edges(tree.left, dot)
                dot.edge(str(id(tree)), str(id(tree.left)), label="<= " +␣
↪str(tree.threshold))
            if isinstance(tree, Node) and tree.right is not None:
                add_nodes_edges(tree.right, dot)
                dot.edge(str(id(tree)), str(id(tree.right)), label="> " +␣
↪str(tree.threshold))

        add_nodes_edges(tree, dot)
```

```
        return dot


    def fit(self, X, Y):
        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)

    def predict(self, X):
        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions

    def make_prediction(self, x, tree):
        if tree.value!=None: return tree.value
        feature_val = x[tree.feature_index]
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)
```

[32]:
```
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values.reshape(-1,1)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2,␣
↪random_state=41)
```

[33]:
```
classifier = DecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier.fit(X_train, y_train)
```

0.4 Problem 4.4

[34]:
```
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz-10.0.
↪1-win64/bin/'
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz-10.0.
↪1-win64/bin/dot.exe'

classifier.print_tree(feature_names=new_col_names).render("decision_tree_graph")
```

[34]:
```
'decision_tree_graph.pdf'
```

0.5 Problem 4.5

[35]:
```
y_pred = classifier.predict(X_test)
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

[35]: 0.8932584269662921

0.6 Problem 4.6

[36]:
```
new_feature = np.array([[1,0,0,1,1,1], [0,1,0,1,1,0]])
classifier.predict(new_feature)
```

[36]: [1, 1]

0.7 Problem 4.7

[37]:
```
from sklearn.model_selection import KFold
import graphviz

class RandomForestClassifier:
    def __init__(self, n_estimators=5, min_samples_split=2, max_depth=2):
        self.n_estimators = n_estimators
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.estimators = []

    def fit(self, X, y):
        tree = DecisionTreeClassifier(min_samples_split=self.min_samples_split,␣
↪max_depth=self.max_depth)
        # total_samples = len(X)
        for _ in range(self.n_estimators):
            indices = np.random.choice(len(X), size=int(0.8 * len(X)),␣
↪replace=False)
            # shuffled_indices  = np.random.permutation(total_samples)
            # subset_size = int(0.8 * total_samples)
            # subset_indices = shuffled_indices[:subset_size]
            X_subset, y_subset = X[indices], y[indices]
            tree.fit(X_subset, y_subset)
            self.estimators.append(tree)

    def predict(self, X):
        predictions = np.zeros((len(X), self.n_estimators), dtype=int)
        for i, estimator in enumerate(self.estimators):
            predictions[:, i] = estimator.predict(X)
        return np.array([np.bincount(prediction).argmax() for prediction in␣
↪predictions])


def display_tree(tree, feature_names):
    dot = tree.print_tree(feature_names=feature_names)
    return dot
```

```
# Perform 10-fold cross-validation
def cross_validation(X, y, n_estimators=5, min_samples_split=2, max_depth=2,␣
↪n_splits=10):
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    accuracies = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        forest = RandomForestClassifier(n_estimators=n_estimators,␣
↪min_samples_split=min_samples_split, max_depth=max_depth)
        forest.fit(X_train, y_train)
        y_pred = forest.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        accuracies.append(accuracy)

    mean_accuracy = np.mean(accuracies)
    return mean_accuracy

# RandomForest with 5 trees
forest = RandomForestClassifier(n_estimators=5, min_samples_split=2,␣
↪max_depth=3)
forest.fit(X, y)

# Displaying the trees
for i, tree in enumerate(forest.estimators):
    display_tree(tree, new_col_names).render(f"tree_{i}", format="png")

# Perform 10-fold cross-validation
accuracy = cross_validation(X, y, n_estimators=5, min_samples_split=2,␣
↪max_depth=2)
print("Mean accuracy:", accuracy)
```

```
Mean accuracy: 0.8962972420837589
```

[38]:
```
new_feature = np.array([[1,0,0,1,1,1], [0,1,0,1,1,0]])
forest.predict(new_feature)
```

[38]:
```
array([1, 1], dtype=int64)
```

0.8 Problem 4.8

[39]:
```
# from sklearn.tree import DecisionTreeClassifier

def new_fit(self, X, y):
    num_features = X.shape[1]
```

```
    for feature_index in range(num_features):
        X_subset = np.delete(X, feature_index, axis=1)
        tree = DecisionTreeClassifier(min_samples_split=self.min_samples_split,␣
 ↪max_depth=self.max_depth)
        tree.fit(X_subset, y)
        self.estimators.append(tree)

def new_predict(self, X):
    predictions = np.zeros((len(X), self.n_estimators), dtype=int)
    for i, estimator in enumerate(self.estimators):
        X_subset = np.delete(X, i, axis=1)
        predictions[:, i] = estimator.predict(X_subset)
    return np.array([np.bincount(prediction).argmax() for prediction in␣
 ↪predictions])

RandomForestClassifier.fit = new_fit
RandomForestClassifier.predict = new_predict

# RandomForest with 6 trees
forest = RandomForestClassifier(n_estimators=6, min_samples_split=2,␣
 ↪max_depth=3)
forest.fit(X, y)

# Displaying the trees
for i, tree in enumerate(forest.estimators):
    display_tree(tree, new_col_names).render(f"tree_{i}", format="png")

# Perform 10-fold cross-validation
accuracy = cross_validation(X, y, n_estimators=6, min_samples_split=2,␣
 ↪max_depth=2)
print("Mean accuracy:", accuracy)
```

```
Mean accuracy: 0.8962972420837589
```

[40]:
```
# Create a feature array so I can test multiple at time
new_feature = np.array([[1,0,0,1,1,1], [0,1,0,1,1,0]])
new_pred = forest.predict(new_feature)

for i in new_pred:
    if new_pred[i] == 1:
        print("Survived")
    else:
        print("X(")
```

```
Survived
Survived
```