hw2_yilmaz_emirkan_1901042659

April 30, 2024

1 Abstract

The study aims to predict the age of abalone based on physical measurements, approaching it as a regression problem. The dataset provided comprises one categorical and seven numerical features, making a total of eight. The decision tree structure was designed to accommodate both categorical and numerical features. The optimal split was determined by experimenting with various threshold values and selecting the one that minimizes the mean square error (MSE). During the selection of the best split point, a method similar to what was taught in lectures was employed. Numerical values were sorted, and their midpoint was assessed for the minimum MSE.

The decision tree structure was designed to be adaptable for use in a random forest by incorporating an additional parameter for random feature selection. This parameter ranges between 0 and 1, indicating the proportion of random features to be utilized in a node. Additionally, both pre-pruning and post-pruning techniques were implemented, though only post-pruning was utilized, as specified in the homework guidelines.

The performance of the trained models was evaluated using k-fold cross-validation, with the results clearly favoring the random forest. The decision tree model reported a mean squared error (MSE) of 8.87, indicating less precision compared to the random forest model, which achieved a substantially lower MSE of 4.74. This difference can be attributed to the inherent design of the random forest approach, which aggregates predictions from a multitude of random decision trees, thus providing a more robust and generalized outcome than a single decision tree.

2 Setup

[1]:
```
!pip install ucimlrepo
```

```
Requirement already satisfied: ucimlrepo in /usr/local/lib/python3.10/dist-
packages (0.0.6)
```

[2]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, KFold
```

3 Loading the dataset

[3]:
```
from ucimlrepo import fetch_ucirepo
from enum import Enum

class FeatureType(Enum):
    Numeric = 1
    Categorical = 2

def convert_feature_types(feature_types):
    feature_t = []

    for t in feature_types:
        if t == 'Categorical':
            feature_t.append(FeatureType.Categorical)
        else:
            feature_t.append(FeatureType.Numeric)
    return feature_t

# fetch dataset
abalone = fetch_ucirepo(id=1)

# data as pandas dataframes, convert to numpy
X = abalone.data.features.values
y = abalone.data.targets['Rings'].values

feature_names = abalone.data.features.columns
feature_types = convert_feature_types(abalone.variables['type'])

# variable information
print(abalone.variables)
```

```
             name     role         type demographic  \
0             Sex  Feature  Categorical        None
1          Length  Feature   Continuous        None
2        Diameter  Feature   Continuous        None
3          Height  Feature   Continuous        None
4    Whole_weight  Feature   Continuous        None
5  Shucked_weight  Feature   Continuous        None
6  Viscera_weight  Feature   Continuous        None
7    Shell_weight  Feature   Continuous        None
8           Rings   Target      Integer        None

                 description  units missing_values
0         M, F, and I (infant)   None             no
1    Longest shell measurement     mm             no
2      perpendicular to length     mm             no
```

```
3          with meat in shell     mm            no
4                whole abalone  grams           no
5               weight of meat  grams           no
6  gut weight (after bleeding)  grams           no
7            after being dried  grams           no
8  +1.5 gives the age in years   None           no
```

4 Implementation of Decision Tree

[4]:
```python
class DecisionNode:
    def __init__(self, feature_idx=None, threshold=None, left=None, right=None, value=None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

class DecisionTree:
    def __init__(self, max_depth=None, r_features=1.0, min_samples_split=1, n_max_thresholds=32):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.r_features = r_features # Rate of features to be seen for each node (less than 1 for a Random Decition Tree)
        self.root = None
        self.feature_types = None
        self.feature_names = None
        self.n_features = 0
        self.n_max_thresholds = n_max_thresholds # Maximum number of numerical thresholds to be tried for minimum MSE

    def fit(self, X, y, feature_types, feature_names):
        self.n_features = X.shape[1]
        self.feature_types = feature_types
        self.feature_names = feature_names
        self.root = self._grow_tree(X, y)

    def _calculate_mse(self, left_y, right_y):
        total_samples = len(left_y) + len(right_y)
        left_mse = np.mean((left_y - np.mean(left_y))**2)
        right_mse = np.mean((right_y - np.mean(right_y))**2)

        return (len(left_y) / total_samples) * left_mse + (len(right_y) / total_samples) * right_mse
```

```python
    def _split_numeric(self, X, y, feature_index):
        best_mse = np.inf
        best_threshold = None

        # Sort the column values and take the medium of two consecutive values (np.unique returns the ordered unique elements)
        values = np.unique(X[:, feature_index])
        thresholds = (values[1:] + values[:-1]) / 2

        # Randomly sample potential thresholds, in case there are many possible splitting points
        if len(thresholds) > self.n_max_thresholds:
            thresholds = np.random.choice(thresholds, size=self.n_max_thresholds, replace=False)

        for thresh in thresholds:
            # Split the DataFrame according to the split value
            left_indices = X[:, feature_index] > thresh
            right_indices = ~left_indices

            # If there is no information gain
            if np.sum(left_indices) == 0 or np.sum(right_indices) == 0:
                continue

            # Calculate the errors E(X - X_expected) for left and right splits
            mse = self._calculate_mse(y[left_indices], y[right_indices])
            if mse < best_mse:
                best_mse = mse
                best_threshold = thresh

        return best_mse, best_threshold

    def _split_categorical(self, X, y, feature_index):
        best_mse = np.inf
        best_threshold = None

        categorical_values = np.unique(X[:, feature_index])
        for thres in categorical_values:
            left_indices = X[:, feature_index] == thres
            right_indices = ~left_indices

            # If there is no information gain
            if np.sum(left_indices) == 0 or np.sum(right_indices) == 0:
                continue

            # Calculate the errors E(X - X_expected) for left and right splits
            mse = self._calculate_mse(y[left_indices], y[right_indices])
```

```
            if mse < best_mse:
                best_mse = mse
                best_threshold = thres

        return best_mse, best_threshold

    def _get_features(self):
        # Get all features
        if self.r_features == 1.0:
            features = [i for i in range(self.n_features)]
        else: # self.r_features < 1.0
            # Randomly select a subset of the features
            n_random_features = max(1, round(self.n_features * self.r_features))
            features = np.random.choice(self.n_features, size=n_random_features, replace=False)

        return features

    # Generated a decision tree recursively, in a greedy approach
    def _grow_tree(self, X, y, depth=0):
        n_samples, n_features = X.shape

        # In case of leaf node
        if depth == self.max_depth or n_samples < self.min_samples_split:
            return DecisionNode(value=np.mean(y))

        features = self._get_features()
        best_mse = np.inf
        best_threshold = None
        best_feature_idx = None

        # Systematically iterate over the features
        for feature_idx in features:
            # Use '>' for numeric, '=' for categorical features
            if self.feature_types[feature_idx] == FeatureType.Numeric:
                mse, threshold = self._split_numeric(X, y, feature_idx)
            else:
                mse, threshold = self._split_categorical(X, y, feature_idx)

            # Update the node, when a better split is found
            if mse < best_mse:
                best_threshold = threshold
                best_mse = mse
                best_feature_idx = feature_idx

        if not best_threshold:
```

```
            return DecisionNode(value=np.mean(y))

        # Split the dataset according to threshold value
        if self.feature_types[best_feature_idx] == FeatureType.Numeric:
            left_indices = X[:, best_feature_idx] > best_threshold
            right_indices = ~left_indices # X[feature] <= best_threshold
        else:
            left_indices = X[:, best_feature_idx] == best_threshold
            right_indices = ~left_indices # X[feature] != best_threshold

        left_tree = self._grow_tree(X[left_indices], y[left_indices], depth + 1)
        right_tree = self._grow_tree(X[right_indices], y[right_indices], depth␣
↪+ 1)

        return DecisionNode(feature_idx=best_feature_idx,␣
↪threshold=best_threshold, left=left_tree, right=right_tree)

    def predict(self, X):
        return np.array([self._predict_tree(x, self.root) for x in X])

    def _predict_tree(self, x, node):
        # If the current node is a leaf, return its value as prediction
        if node.value:
            return node.value

        # Otherwise, determine the prediction based on the node's split␣
↪condition
        test_value = x[node.feature_idx]

        if self.feature_types[node.feature_idx] == FeatureType.Numeric:
            child_node = node.left if (test_value > node.threshold) else node.
↪right
        else: # Feature is categorical
            child_node = node.left if (test_value == node.threshold) else node.
↪right

        return self._predict_tree(x, child_node)

    def prune(self, X_val, y_val):
        self._prune_tree(self.root, X_val, y_val)

    def _prune_tree(self, node, X_val, y_val):
        # No pruning for a leaf node
        if node.value:
            return

        # Prune children first
```

```
        self._prune_tree(node.left, X_val, y_val)
        self._prune_tree(node.right, X_val, y_val)

        # Prune if both children are leaves
        if node.left.value and node.right.value:
            # Evaluate performance with and without the node
            y_pred_before_prune = self.predict(X_val)

            # Prune the subtree by making this node a leaf
            node.value = np.mean([node.left.value, node.right.value])
            y_pred_after_prune = self.predict(X_val)

            # Compare performance before and after pruning
            mse_before_prune = np.mean((y_val - y_pred_before_prune) ** 2)
            mse_after_prune = np.mean((y_val - y_pred_after_prune) ** 2)

            # If pruning improves performance, keep the pruned subtree
            if mse_after_prune < mse_before_prune:
                node.left = None
                node.right = None
            else:
                node.value = None

    def display(self):
        self._display(self.root, depth=0)

    def _display(self, node, depth):
        margin = " " * (depth * 5)
        if node.value:
            print(f"{margin}return [{node.value:.3f}]")
        else:
            feature = self.feature_names[node.feature_idx] if self.
↪feature_names.any() else node.feature_idx

            if self.feature_types[node.feature_idx] == FeatureType.Numeric:
                if_stmt = f"if x['{feature}'] > {node.threshold:.3f}:"
                else_stmt = f"else: # x['{feature}'] <= {node.threshold:.3f}"
            else:
                if_stmt = f"if x['{feature}'] == {node.threshold}:"
                else_stmt = f"else: # x['{feature}'] != {node.threshold}"

            print(f"{margin}{if_stmt}")
            self._display(node.left, depth + 1)

            print(f"{margin}{else_stmt}")
            self._display(node.right, depth + 1)
```

[5]:
```python
def build_dt(X, y, feature_types, feature_names=None):
    dt = DecisionTree()
    dt.fit(X, y, feature_types, feature_names)
    return dt

def predict_dt(dt, X):
    return dt.predict(X)
```

5 Decision Tree Testing

[6]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

dt = build_dt(X_train, y_train, feature_types, feature_names)

y_pred = predict_dt(dt, X_train)
mse = mean_squared_error(y_train, y_pred)
print(f"MSE on training set: {mse:.2f}")

y_pred = predict_dt(dt, X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"MSE on test set: {mse:.2f}")
```

```
MSE on training set: 0.00
MSE on test set: 8.91
```

The MSE for the training set on a non-pruned decision tree is 0.00, indicating that the tree is overfitted, as expected.

[ ]:
```python
dt.display()
```

5.1 Results of k-fold cross validation

[8]:
```python
# Perform k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

mse_scores = []

for train_index, val_index in kf.split(X):

    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]

    # Train the model
    dt = build_dt(X_train, y_train, feature_types, feature_names)

    # Make predictions
```

```
    y_pred = predict_dt(dt, X_val)

    # Compute MSE
    mse_fold = mean_squared_error(y_val, y_pred)
    mse_scores.append(mse_fold)

    print(f"MSE for fold: {mse_fold:.2f}")

# Average MSE across all folds on training set
avg_mse = sum(mse_scores) / len(mse_scores)
print(f"Average MSE from k-fold Cross-Validation: {avg_mse:.2f}")
```

```
MSE for fold: 8.48
MSE for fold: 8.52
MSE for fold: 9.68
MSE for fold: 9.49
MSE for fold: 8.17
Average MSE from k-fold Cross-Validation: 8.87
```

5.2 Decision Tree Testing with Pruning

[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

dt = build_dt(X_train, y_train, feature_types, feature_names)

y_pred = predict_dt(dt, X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"MSE: {mse:.2f}")

dt.prune(X_test, y_test)
y_pred = predict_dt(dt, X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"MSE (post-pruning): {mse:.2f}")
```

```
MSE: 8.95
MSE (post-pruning): 8.74
```

After applying post-pruning, there is a slight decrease in MSE.

6 Implementation of RDF

[10]:
```
class RandomForest:
    def __init__(self, n_estimators=10, max_depth=None, r_features=0.3):
        self.n_estimators = n_estimators
        self.trees = []     # List of decision trees
        self.max_depth = max_depth
```

```python
        self.r_features = r_features  # Rate of features to be seen on each node

    def fit(self, X, y, feature_types, feature_names):
        n_samples = X.shape[0]
        self.trees = []

        for _ in range(self.n_estimators):
            X_boot, y_boot = self._bootstrap(X, y)

            dt = DecisionTree(max_depth=self.max_depth, r_features=self.
 ↪r_features)
            dt.fit(X_boot, y_boot, feature_types, feature_names)
            self.trees.append(dt)

    def _bootstrap(self, X, y):
        num_samples = len(X)
        indices = np.random.choice(len(X), size=num_samples, replace=True)
        return X[indices], y[indices]

    def predict(self, X):
        predictions = np.zeros(len(X))

        for tree in self.trees:
            tree_predictions = tree.predict(X)
            predictions += tree_predictions

        return predictions / self.n_estimators
```

[11]:
```python
def build_rdf(X, y, feature_types, feature_names, n_estimators=10,␣
 ↪max_depth=None, r_features=0.3):
    rdf = RandomForest(n_estimators=n_estimators, max_depth=max_depth,␣
 ↪r_features=r_features)
    rdf.fit(X, y, feature_types, feature_names)
    return rdf

def predict_rdf(rdf, X):
    return rdf.predict(X)
```

7 Results of k-fold cross validation

[12]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Train the model
rdf = build_rdf(X_train, y_train, feature_types, feature_names,␣
 ↪n_estimators=50, r_features=0.5)
```

```python
# Make predictions
y_pred = predict_rdf(rdf, X_test)

# Compute MSE
mse = mean_squared_error(y_test, y_pred)

print(f"MSE: {mse:.2f}")
```

```
MSE: 4.96
```

[13]:
```python
# Perform k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

n_trees = 50
mse_scores = []
r_features = 0.5

for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]

    # Train the model
    rdf = build_rdf(X_train, y_train, feature_types, feature_names,␣
 ↪n_estimators=n_trees, r_features=r_features)

    # Make predictions
    y_pred_fold = predict_rdf(rdf, X_val)

    # Compute MSE
    mse = mean_squared_error(y_val, y_pred_fold)
    mse_scores.append(mse)

    print(f"MSE for fold: {mse:.2f}")

# Average MSE across all folds on training set
avg_mse = sum(mse_scores) / len(mse_scores)
print(f"Average MSE from k-fold Cross-Validation: {avg_mse:.2f}")
```

```
MSE for fold: 5.04
MSE for fold: 4.41
MSE for fold: 5.14
MSE for fold: 5.31
MSE for fold: 3.80
Average MSE from k-fold Cross-Validation: 4.74
```

After assessing the models using k-fold cross-validation, the random forest outperformed the decision tree. The decision tree had an MSE of 8.87, while the random forest achieved a lower MSE of

4.74. This confirms the random forest's advantage in providing more generalized predictions due to its ensemble nature, as evidenced by our experimental findings.