RandomForest_sklearn

May 19, 2024

[44]:
```
from collections import Counter

import numpy as np
from numpy import genfromtxt
import scipy.io
from scipy.stats import mode
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,␣
↪VotingClassifier
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.model_selection import cross_val_score
from pydot import graph_from_dot_data
import io
import random
import pandas as pd

eps = 1e-5  # a small number

# Dataset
spam = scipy.io.loadmat('datasets/spam_data/spam_data.mat')
spam_training_data, spam_training_labels = spam['training_data'], np.
↪squeeze(spam['training_labels'])
spam_test = spam['test_data']


# Preprocess for titanic data
def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
    # fill_mode = False

    # Temporarily assign -1 to missing data
    data[data == b''] = '-1'

    # Hash the columns (used for handling strings)
    onehot_encoding = []
    onehot_features = []
    for col in onehot_cols:
        counter = Counter(data[:, col])
```

```
        for term in counter.most_common():
            if term[0] == b'-1':
                continue
            if term[-1] <= min_freq:
                break
            onehot_features.append(term[0])
            onehot_encoding.append((data[:, col] == term[0]).astype(float))
        data[:, col] = '0'
    onehot_encoding = np.array(onehot_encoding).T
    data = np.hstack(
        [np.array(data, dtype=float),
         np.array(onehot_encoding)])
    # Replace missing data with the mode value.
    if fill_mode:
        # TODO
        for col in range(data.shape[1]):
            missing_idx = (data[:, col] == -1)
            if missing_idx.any():
                col_mode = mode(data[~missing_idx, col])[0]
                data[missing_idx, col] = col_mode

    return data, onehot_features


# Load titanic data
data = genfromtxt('datasets/titanic/titanic_training.csv', delimiter=',',␣
 ↪dtype=None)
test_data = genfromtxt('datasets/titanic/titanic_testing_data.csv',␣
 ↪delimiter=',', dtype=None)
y = data[1:, 0]  # label = survived
class_names = ["Died", "Survived"]

labeled_idx = np.where(y != b'')[0]
y = np.array(y[labeled_idx], dtype=float).astype(int)
print("Preprocessing the titanic dataset")
X, onehot_features = preprocess(data[1:, 1:], onehot_cols=[1, 5, 7, 8])  #␣
 ↪onehot_cols: [pclass, parch, fare, cabin]
X = X[labeled_idx, :]
Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
assert X.shape[1] == Z.shape[1]
features = list(data[0, 1:]) + onehot_features

# Rename titanic data
titanic_training, titanic_training_labels = X, y
titanic_test = Z
```

```
Preprocessing the titanic dataset
```

```
/var/folders/3j/yt011p3d543_vhy83qz20ctc0000gr/T/ipykernel_56996/2462536374.py:6
1: VisibleDeprecationWarning: Reading unicode strings without specifying the
encoding argument is deprecated. Set the encoding, use None for the system
default.
  data = genfromtxt('datasets/titanic/titanic_training.csv', delimiter=',',
dtype=None)
/var/folders/3j/yt011p3d543_vhy83qz20ctc0000gr/T/ipykernel_56996/2462536374.py:6
2: VisibleDeprecationWarning: Reading unicode strings without specifying the
encoding argument is deprecated. Set the encoding, use None for the system
default.
  test_data = genfromtxt('datasets/titanic/titanic_testing_data.csv',
delimiter=',', dtype=None)
```

[2]:
```
# Helper func
def evaluate(clf):
    print("Cross validation", cross_val_score(clf, X, y))
    if hasattr(clf, "decision_trees"):
        counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
        first_splits = [
            (features[term[0]], term[1]) for term in counter.most_common()
        ]
        print("First splits", first_splits)

def evaluate_simple(pred, y):
    return np.mean(pred == y)

def train_valid_split(X, y, holdout):
    num = X.shape[0]
    split = int(num * holdout)
    X_train, X_valid = X[:split], X[split:]
    y_train, y_valid = y[:split], y[split:]
    return X_train, X_valid, y_train, y_valid

def results_to_csv(y_test, name, method):
    y_test = y_test.astype(int)
    df = pd.DataFrame({'Category': y_test})
    df.index += 1 # Ensures that the index starts at 1
    df.to_csv(f'{name}_{method}_pred.csv', index_label='Id')
```

[3]:
```
# Decision Trees
class DecisionTree:
    def __init__(self, max_depth=3, feature_labels=None):
        self.max_depth = max_depth
        self.features = feature_labels
        self.left, self.right = None, None  # for non-leaf nodes
        self.split_idx, self.thresh = None, None  # for non-leaf nodes
        self.data, self.pred = None, None  # for leaf nodes
```

```python
    @staticmethod
    def entropy(y):
        # TODO
        base_probabilities = []
        for class_label in np.unique(y):
            count = len(y[np.where(y==class_label)])
            base_probabilities.append(float(count / len(y)))

        H_S = -1* sum([p_c * np.log2(p_c) for p_c in base_probabilities])
        return H_S

    @staticmethod
    def information_gain(X, y, idx, thresh):
        # TODO
        H_S = DecisionTree.entropy(y)
        left_indices, right_indices = np.where(X[:, idx] < thresh)[0], np.
↪where(X[:, idx] >= thresh)[0]
        yl, yr = y[left_indices], y[right_indices]
        Sl, Sr = len(yl) / len(y), len(yr) / len(y)
        H_after = (Sl * DecisionTree.entropy(yl) + Sr * DecisionTree.
↪entropy(yr) ) / (Sl + Sr)
        return H_S - H_after

    @staticmethod
    def gini_impurity(X, y, thresh):
        # TODO
        _, counts = np.unique(y, return_counts=True)
        prob = counts / counts.sum()
        return 1 - np.sum(prob**2)

    @staticmethod
    def gini_purification(X, y, thresh):
        # TODO
        init_gini = self.gini_impurity(X, y, thresh)
        left_indices, right_indices = X[:, self.split_idx] < thresh, X[:, self.
↪split_idx] >= thresh
        yl, yr = y[left_indices], y[right_indices]
        Sl, Sr = len(yl) / len(y), len(yr) / len(y)
        after_gini = (Sl * yl + Sr * yr) / (Sl + Sr)
        return init_gini - after_gini

    def split(self, X, y, idx, thresh):
        X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
        y0, y1 = y[idx0], y[idx1]
        return X0, y0, X1, y1
```

```python
    def split_test(self, X, idx, thresh):
        idx0 = np.where(X[:, idx] < thresh)[0]
        idx1 = np.where(X[:, idx] >= thresh)[0]
        X0, X1 = X[idx0, :], X[idx1, :]
        return X0, idx0, X1, idx1

    def fit(self, X, y):
        # TODO
        y = y.astype(int)
        # Base case
        if self.max_depth <= 0 or len(np.unique(y)) == 1:
            self.pred = np.bincount(y).argmax()  # Find the most common value
            return

        num_samples, num_features = X.shape
        best_gain = 0
        # GrowTree
        for idx in range(num_features):
            thresholds = np.unique(X[:, idx])
            for thresh in thresholds:  # Repeatly call information_gain to validate the best split
                gain = DecisionTree.information_gain(X, y, idx, thresh)
                if gain > best_gain:
                    best_gain = gain
                    self.split_idx, self.thresh = idx, thresh

        # If no improvement, make this a leaf node
        if best_gain == 0:
            self.pred = np.bincount(y).argmax()
            return
        # If found a useful split, proceed recursively
        left_indices, right_indices = np.where(X[:, self.split_idx] < self.thresh)[0], np.where(X[:, self.split_idx] >= self.thresh)[0]
        self.left, self.right = DecisionTree(self.max_depth - 1, self.features), DecisionTree(self.max_depth - 1, self.features)
        # Recursively fit the left, right child
        self.left.fit(X[left_indices], y[left_indices])
        self.right.fit(X[right_indices], y[right_indices])

    def predict(self, X):
        # TODO
        # Traverse down to the leaf node
        def predict_once(sample, node):
            if node.pred is not None:
                return node.pred
            # Recursive down the left children
            if sample[node.split_idx] < node.thresh:
```

```python
                return predict_once(sample, node.left)
            else:
                return predict_once(sample, node.right)

        predictions = [predict_once(sample, self) for sample in X]
        return np.array(predictions)

    def __repr__(self):
        if self.max_depth == 0:
            return "%s (%s)" % (self.pred, self.labels.size)
        else:
            return "[%s < %s: %s | %s]" % (self.features[self.split_idx],
                                            self.thresh, self.left.__repr__(),
                                            self.right.__repr__())

# Output Model Accuracy
spam_decisionTree_clf = DecisionTree(max_depth=3)
spam_decisionTree_clf.fit(spam_training_data, spam_training_labels)
spam_decisionTree_pred = spam_decisionTree_clf.predict(spam_training_data)
print(f'Spam Decision Tree Accuracy: {evaluate_simple(spam_decisionTree_pred,
 ↪spam_training_labels)}')

titanic_decisionTree_clf = DecisionTree(max_depth=3)
titanic_decisionTree_clf.fit(titanic_training, titanic_training_labels)
titanic_decisionTree_pred = titanic_decisionTree_clf.predict(titanic_training)
print(f'Titanic Decision Tree Accuracy:
 ↪{evaluate_simple(titanic_decisionTree_pred, titanic_training_labels)}\n')


# Validation
random.seed(246810)
np.random.seed(246810)

spam_train, spam_valid, spam_train_labels, spam_valid_labels =
 ↪train_valid_split(spam_training_data, spam_training_labels, 0.2)
tit_train, tit_valid, tit_train_labels, tit_valid_labels =
 ↪train_valid_split(titanic_training, titanic_training_labels, 0.2)

def decisionTree_eval(X_train, X_valid, y_train, y_valid):
    clf = DecisionTree(max_depth=3)
    clf.fit(X_train, y_train)
    train_pred, valid_pred = clf.predict(X_train), clf.predict(X_valid)
    train_accuracy, valid_accuracy = evaluate_simple(train_pred, y_train),
 ↪evaluate_simple(valid_pred, y_valid)
    return train_accuracy, valid_accuracy

# Spam decisionTree
```

```
spam_train_accuracy, spam_valid_accuracy = decisionTree_eval(spam_train,␣
 ↪spam_valid, spam_train_labels, spam_valid_labels)
print(f"Spam training accuracy for decisionTree: {spam_train_accuracy},\nSpam␣
 ↪validation accuracy for decisionTree: {spam_valid_accuracy}\n")

# Titanic decisionTree
tit_train_accuracy, tit_valid_accuracy = decisionTree_eval(tit_train,␣
 ↪tit_valid, tit_train_labels, tit_valid_labels)
print(f"Titanic training accuracy for decisionTree:␣
 ↪{tit_train_accuracy},\nTitanic validation accuracy for decisionTree:␣
 ↪{tit_valid_accuracy}\n")
```

```
Spam Decision Tree Accuracy: 0.7937466690353526
Titanic Decision Tree Accuracy: 0.8126858275520317

Spam training accuracy for decisionTree: 0.8071111111111111,
Spam validation accuracy for decisionTree: 0.7708703374777975

Titanic training accuracy for decisionTree: 0.8208955223880597,
Titanic validation accuracy for decisionTree: 0.7178217821782178
```

[13]:
```
# sklearn decision Tree accuracy
random.seed(246810)
np.random.seed(246810)

clf = DecisionTreeClassifier(max_depth=3)
clf.fit(spam_training_data, spam_training_labels)
new_pred = clf.predict(spam_training_data)
print(f'Sklearn:\nSpam Decision Tree Accuracy: {evaluate_simple(new_pred,␣
 ↪spam_training_labels)}')

tit_clf = DecisionTreeClassifier(max_depth=3)
tit_clf.fit(titanic_training, titanic_training_labels)
new_tit_pred = tit_clf.predict(titanic_training)
print(f'Titanic Decision Tree Accuracy: {evaluate_simple(new_tit_pred,␣
 ↪titanic_training_labels)}\n')

# sklearn decisionTree validation accuracy
spam_train, spam_valid, spam_train_labels, spam_valid_labels =␣
 ↪train_valid_split(spam_training_data, spam_training_labels, 0.2)
tit_train, tit_valid, tit_train_labels, tit_valid_labels =␣
 ↪train_valid_split(titanic_training, titanic_training_labels, 0.2)

def decisionTree_eval(X_train, X_valid, y_train, y_valid):
    clf = DecisionTreeClassifier(max_depth=3)
    clf.fit(X_train, y_train)
```

```
    train_pred, valid_pred = clf.predict(X_train), clf.predict(X_valid)
    train_accuracy, valid_accuracy = evaluate_simple(train_pred, y_train),␣
 ↪evaluate_simple(valid_pred, y_valid)
    return train_accuracy, valid_accuracy

# Spam decisionTree
spam_train_accuracy, spam_valid_accuracy = decisionTree_eval(spam_train,␣
 ↪spam_valid, spam_train_labels, spam_valid_labels)
print(f"Spam training accuracy for decisionTree: {spam_train_accuracy},\nSpam␣
 ↪validation accuracy for decisionTree: {spam_valid_accuracy}\n")

# Titanic decisionTree
tit_train_accuracy, tit_valid_accuracy = decisionTree_eval(tit_train,␣
 ↪tit_valid, tit_train_labels, tit_valid_labels)
print(f"Titanic training accuracy for decisionTree:␣
 ↪{tit_train_accuracy},\nTitanic validation accuracy for decisionTree:␣
 ↪{tit_valid_accuracy}\n")
```

```
Sklearn:
Spam Decision Tree Accuracy: 0.8008527269497246
Titanic Decision Tree Accuracy: 0.8107036669970268

Spam training accuracy for decisionTree: 0.8071111111111111,
Spam validation accuracy for decisionTree: 0.7795293072824157

Titanic training accuracy for decisionTree: 0.8308457711442786,
Titanic validation accuracy for decisionTree: 0.7834158415841584
```

[81]:

```
# sklearn BaggedTrees
# Validation on n2
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

random.seed(246810)
np.random.seed(246810)

baggedTree_clf = BaggingClassifier(base_estimator=clf, n_estimators=200)
baggedTree_clf.fit(spam_training_data, spam_training_labels)
baggedTree_pred = baggedTree_clf.predict(spam_training_data)
print(f"Sklearn bagging:\nSpam bagging Training Accuracy:␣
 ↪{round(evaluate_simple(baggedTree_pred, spam_training_labels), 5)}")

tit_bagging_clf = BaggingClassifier(base_estimator=clf, n_estimators=200)
tit_bagging_clf.fit(titanic_training, titanic_training_labels)
tit_bagging_predictions = tit_bagging_clf.predict(titanic_training)
```

```python
print(f"Titanic baggedTree Training Accuracy:␣
↪{round(evaluate_simple(tit_bagging_predictions, titanic_training_labels),␣
↪5)}\n")


# sklearn baggedTrees validation accuracy
spam_train, spam_valid, spam_train_labels, spam_valid_labels =␣
↪train_valid_split(spam_training_data, spam_training_labels, 0.2)
tit_train, tit_valid, tit_train_labels, tit_valid_labels =␣
↪train_valid_split(titanic_training, titanic_training_labels, 0.2)

def bagging_eval(X_train, X_valid, y_train, y_valid):
    clf = BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=3),␣
↪n_estimators=200)
    clf.fit(X_train, y_train)
    train_pred, valid_pred = clf.predict(X_train), clf.predict(X_valid)
    train_accuracy, valid_accuracy = evaluate_simple(train_pred, y_train),␣
↪evaluate_simple(valid_pred, y_valid)
    return train_accuracy, valid_accuracy

# Spam baggedTrees
spam_train_accuracy, spam_valid_accuracy = bagging_eval(spam_train, spam_valid,␣
↪spam_train_labels, spam_valid_labels)
print(f"Spam training accuracy for bagging: {round(spam_train_accuracy,␣
↪5)},\nSpam validation accuracy for bagging: {round(spam_valid_accuracy,␣
↪5)}\n")

# Titanic baggedTrees
tit_train_accuracy, tit_valid_accuracy = bagging_eval(tit_train, tit_valid,␣
↪tit_train_labels, tit_valid_labels)
print(f"Titanic training accuracy for bagging: {round(tit_train_accuracy,␣
↪5)},\nTitanic validation accuracy for bagging: {round(tit_valid_accuracy,␣
↪5)}\n")
```

```
Sklearn bagging:
Spam bagging Training Accuracy: 0.8037
Titanic baggedTree Training Accuracy: 0.81863

Spam training accuracy for bagging: 0.83822,
Spam validation accuracy for bagging: 0.80617

Titanic training accuracy for bagging: 0.85075,
Titanic validation accuracy for bagging: 0.78465
```

[ ]:
```python
## Sklearn RandomForest
# rf spam needs higher depth, titanic needs less depth
```

```
random.seed(246810)
np.random.seed(246810)

spam_rf_clf = RandomForestClassifier(n_estimators=200, max_depth=8)
spam_rf_clf.fit(spam_training_data, spam_training_labels)
spam_rf_pred = spam_rf_clf.predict(spam_training_data)
print(f"Sklearn randomForest:\nSpam randomForest Training Accuracy: ␣
 ↪{evaluate_simple(spam_rf_pred, spam_training_labels)}")

tit_rf_clf = RandomForestClassifier(n_estimators=200, max_depth=8)
tit_rf_clf.fit(titanic_training, titanic_training_labels)
tit_rf_predictions = tit_rf_clf.predict(titanic_training)
print(f"Titanic randomForest Training Accuracy: ␣
 ↪{evaluate_simple(tit_rf_predictions, titanic_training_labels)}\n")


# sklearn random Forest validation accuracy
spam_train, spam_valid, spam_train_labels, spam_valid_labels = ␣
 ↪train_valid_split(spam_training_data, spam_training_labels, 0.2)
tit_train, tit_valid, tit_train_labels, tit_valid_labels = ␣
 ↪train_valid_split(titanic_training, titanic_training_labels, 0.2)

def rf_eval(X_train, X_valid, y_train, y_valid):
    clf = RandomForestClassifier(n_estimators=200, max_depth=8)
    clf.fit(X_train, y_train)
    train_pred, valid_pred = clf.predict(X_train), clf.predict(X_valid)
    train_accuracy, valid_accuracy = evaluate_simple(train_pred, y_train),␣
 ↪evaluate_simple(valid_pred, y_valid)
    return train_accuracy, valid_accuracy

# Spam RandomForest
spam_train_accuracy, spam_valid_accuracy = rf_eval(spam_train, spam_valid,␣
 ↪spam_train_labels, spam_valid_labels)
print(f"Spam training accuracy for randomForest: {spam_train_accuracy},\nSpam␣
 ↪validation accuracy for randomForest: {spam_valid_accuracy}\n")

# Titanic RandomForest
tit_train_accuracy, tit_valid_accuracy = rf_eval(tit_train, tit_valid,␣
 ↪tit_train_labels, tit_valid_labels)
print(f"Titanic training accuracy for randomForest: ␣
 ↪{tit_train_accuracy},\nTitanic validation accuracy for randomForest: ␣
 ↪{tit_valid_accuracy}")
```

[57]:
```
# Sklearn ensemble learning

#create a dictionary of our models
```

```python
estimators=[('dt', clf), ('bag', baggedTree_clf), ('rf', spam_rf_clf)]

ensemble = VotingClassifier(estimators, voting='hard')
ensemble.fit(spam_train, spam_train_labels)
print(f"Spam ensemble learning Accuracy: {ensemble.score(spam_valid,␣
↪spam_valid_labels)}")



# Titanic
tit_estimators=[('tdt', tit_clf), ('tbag', tit_bagging_clf), ('trf',␣
↪tit_rf_clf)]
tit_ensemble = VotingClassifier(tit_estimators, voting='hard')
tit_ensemble.fit(tit_train, tit_train_labels)
print(f"Titanic ensemble learning Accuracy: {tit_ensemble.score(tit_valid,␣
↪tit_valid_labels)}")
```

```
Spam ensemble learning Accuracy: 0.8068383658969804
Titanic ensemble learning Accuracy: 0.7933168316831684
```