stable_tree_bertsimas

April 17, 2025

0.1 Notebook implementing the algorithm in Bertsimas et al. (ttps://arxiv.org/abs/2305.17299)

[1]:
```python
import numpy as np
import itertools
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score
```

0.1.1 Import dataset and split

[2]:
```python
data_breast_cancer = load_breast_cancer(as_frame=True)
X_full = data_breast_cancer["data"]
y_full = data_breast_cancer["target"]
```

[3]:
```python
print("X_full shape: ", X_full.shape)
X_full.head()
```

```
X_full shape:  (569, 30)
```

[3]:
```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  …  worst radius  worst texture  worst perimeter  \
0                 0.07871  …         25.38          17.33           184.60
1                 0.05667  …         24.99          23.41           158.80
```

```
2               0.05999  ...        23.57        25.53          152.50
3               0.09744  ...        14.91        26.50           98.87
4               0.05883  ...        22.54        16.67          152.20

   worst area  worst smoothness  worst compactness  worst concavity  \
0      2019.0            0.1622             0.6656           0.7119
1      1956.0            0.1238             0.1866           0.2416
2      1709.0            0.1444             0.4245           0.4504
3       567.7            0.2098             0.8663           0.6869
4      1575.0            0.1374             0.2050           0.4000

   worst concave points  worst symmetry  worst fractal dimension
0                0.2654          0.4601                  0.11890
1                0.1860          0.2750                  0.08902
2                0.2430          0.3613                  0.08758
3                0.2575          0.6638                  0.17300
4                0.1625          0.2364                  0.07678

[5 rows x 30 columns]
```

[4]:
```python
print("y_full shape: ", y_full.shape)
y_full.head()
```

```
y_full shape:  (569,)
```

[4]:
```
0    0
1    0
2    0
3    0
4    0
Name: target, dtype: int64
```

[5]:
```python
X_train, X_test, y_train, y_test = train_test_split(X_full, y_full, test_size=0.
 ↪2, random_state=42)

print("X_train shape: {}, X_test shape: {}".format(X_train.shape, X_test.shape))
print("y_train shape: {}, y_test shape: {}".format(y_train.shape, y_test.shape))
```

```
X_train shape: (455, 30), X_test shape: (114, 30)
y_train shape: (455,), y_test shape: (114,)
```

0.1.2 Generate first collection (T0) of trees (trained on X0)

[6]:
```python
def train_trees(X, y, depths=[3,5,7], min_samples=[5,10]):
    """Train multiple trees for different hyperparams & possibly bootstrap."""
    trees = []
    for depth, min_leaf in itertools.product(depths, min_samples):
        # need to boostrap later
```

```
        clf = DecisionTreeClassifier(
            max_depth=depth,
            min_samples_leaf=min_leaf,
            random_state=42
        )
        clf.fit(X, y)
        trees.append(clf)
    return trees

T0 = train_trees(X_train, y_train)
print("Generated {} trees for T0".format(len(T0)))
```

```
Generated 6 trees for T0
```

0.1.3 Generate second collection of trees (T) (trained on full data)

[7]:
```
T = train_trees(X_full, y_full)
```

0.1.4 Get global ranges of numerical features and their names

(todo: categorical features)

[8]:
```
feature_names = X_full.columns
global_lower = X_full.min().values
global_upper = X_full.max().values
```

0.1.5 Compute average distance of each tree in T to the T0 collection

[9]:
```
from bertsimas_stable.Paths import tree_distance
```

Distances to T0

[10]:
```
distances = []
max_depth = 7  # largest in our hyperparameter search
for i, tree_b in enumerate(T):
    # average distance to all trees in T0
    d_b = 0.0
    for tree_beta in T0:
        d_b += tree_distance(
            tree_beta, tree_b,
            global_lower=global_lower,
            global_upper=global_upper,
            lambda_val=2*max_depth
        )
    d_b /= len(T0)
    distances.append(d_b)
distances
```

[10]:
```
[np.float64(9.941638853041566),
 np.float64(9.726568660248738),
 np.float64(9.34090448178685),
 np.float64(9.872533253986653),
 np.float64(9.716180344437472),
 np.float64(9.824395104992929)]
```

0.1.6 Distances to T0 (using given method)

[11]:
```python
from dt_distance_repo.dt_distance.distance_calculator import DistanceCalculator
X_train = X_train.values
X_test = X_test.values
distances_method2 = []
for i, tree_b in enumerate(T):
    # average distance to all trees in T0
    d_b = 0.0
    for tree_beta in T0:
        distance_calculator = DistanceCalculator(tree_beta, tree_b, X=X_train,␣
 ↪y=y_train)
        d_b += distance_calculator.compute_tree_distance()
    d_b /= len(T0)
    distances_method2.append(d_b)
distances_method2
```

[11]:
```
[np.float64(0.05715),
 np.float64(0.05145),
 np.float64(0.038016666666666664),
 np.float64(0.026416666666666668),
 np.float64(0.020283333333333334),
 np.float64(0.014333333333333335)]
```

0.1.7 Compute predictive performance for each tree

[12]:
```python
performances = []
for i, tree_b in enumerate(T):
    y_pred = tree_b.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    performances.append(acc)

print("Accuracies on test set: {%s}" % ", ".join("%.3f" % a for a in␣
 ↪performances))
```

```
Accuracies on test set: {0.974, 0.965, 0.991, 0.965, 0.991, 0.956}
```

```
/Users/adb/stuff/gitclones/Suicide_Project/.venv/lib/python3.11/site-
packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid
feature names, but DecisionTreeClassifier was fitted with feature names
  warnings.warn(
```

```
/Users/adb/stuff/gitclones/Suicide_Project/.venv/lib/python3.11/site-
packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid
feature names, but DecisionTreeClassifier was fitted with feature names
  warnings.warn(
/Users/adb/stuff/gitclones/Suicide_Project/.venv/lib/python3.11/site-
packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid
feature names, but DecisionTreeClassifier was fitted with feature names
  warnings.warn(
/Users/adb/stuff/gitclones/Suicide_Project/.venv/lib/python3.11/site-
packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid
feature names, but DecisionTreeClassifier was fitted with feature names
  warnings.warn(
/Users/adb/stuff/gitclones/Suicide_Project/.venv/lib/python3.11/site-
packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid
feature names, but DecisionTreeClassifier was fitted with feature names
  warnings.warn(
/Users/adb/stuff/gitclones/Suicide_Project/.venv/lib/python3.11/site-
packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid
feature names, but DecisionTreeClassifier was fitted with feature names
  warnings.warn(
```

0.1.8 Identify Pareto frontier

[13]:
```python
pairs = list(zip(distances, performances))  # (distance, performance)

def is_dominated(i, pairs):
    di, pi = pairs[i]
    for j, (dj, pj) in enumerate(pairs):
        if j != i:
            # Condition for i is dominated by j: dj <= di and pj >= pi
            # with at least one strict inequality
            if (dj <= di and pj >= pi) and (dj < di or pj > pi):
                return True
    return False

pareto_indices = [i for i in range(len(pairs)) if not is_dominated(i, pairs)]
pareto_trees = [T[i] for i in pareto_indices]

print("Pareto indices:", pareto_indices)
print("Number of Pareto-optimal trees:", len(pareto_trees))
```

```
Pareto indices: [2]
Number of Pareto-optimal trees: 1
```

0.2 Choose the "best" stable tree from the Pareto set

[14]:
```
distance_threshold = min(distances) + 0.2 * (max(distances) - min(distances))
candidate_indices = [i for i in pareto_indices if distances[i] <=␣
 ↪distance_threshold]
if candidate_indices:
    best_idx = max(candidate_indices, key=lambda i: performances[i])
    stable_tree = T[best_idx]
    print(f"Chosen stable tree index = {best_idx}, dist={distances[best_idx]},␣
 ↪perf={performances[best_idx]}")
else:
    print("No tree satisfies the distance threshold; picking best accuracy from␣
 ↪all T.")
    best_idx = np.argmax(performances)
    stable_tree = T[best_idx]
    print(f"Chosen best accuracy tree index = {best_idx},␣
 ↪dist={distances[best_idx]}, perf={performances[best_idx]}")
```

```
Chosen stable tree index = 2, dist=9.34090448178685, perf=0.9912280701754386
```