Tutorial_6_TrAdaBoost_R2 14 April 2025

Transfer Learning for Class AMAT 6000A: Advanced Materials Informatics Spring 2025, HKUST (GZ)

This tutorial extends the instance transfer learning algorithm, **TrAdaBoost** from applying on classification to regression task. We will go step by step an example of utilizing TrAdaBoost for property classification, using Python and Jpyter Noebooks.

Code Example, Data, and Illustrations

The codes and examples are provided by Bin CAO on https://github.com/Bin-Cao/TrAdaboost/tree/main/TrAdaBoost.

Preparing for the Class

To run the code examples in this tutorial, ensure you have Python and Jupyter Notebook installed. Below is a comprehensive guide to help you get set up

Requirements

- python >= 3.7
- sklearn

Introduction of TrAdaBoost_R2

TrAdaBoost_R2 (Transfer AdaBoost) is a transfer learning model adapted from the base TraAdaBoost for regression problem when the training data and test data come from different distributions. Different from TrAdaBoost on classification, the R2 version uses regression loss based on numerical differences between ground truth values and prediction values, with different forms (e.g., absolute, square, exponential). The calculation of error rate is the same but the updates of weights are not.

Initialization and Setup

In [50]:
```
import numpy as np
import pandas as pd
import warnings
import matplotlib.pyplot as plt
import copy
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import LeaveOneOut
from sklearn.metrics import mean_squared_error
```

Function Definition and Explanation

In [34]:
```
def TrAdaBoost_R2(trans_S, Multi_trans_A, response_S, Multi_response_A, test, N):
    """Boosting for regression transfer.
```

```
    Please feel free to open issues in the Github : https://github.com/Bin-Cao/TrAdaboost
    or
    contact Bin Cao (bcao@shu.edu.cn)
    in case of any problems/comments/suggestions in using the code.

    Parameters
    ----------
    trans_S : feature matrix of same-distribution training data

    Multi_trans_A : dict, feature matrix of diff-distribution training data
    e.g.,
    Multi_trans_A = {
    'trans_A_1' :  data_1 ,
    'trans_A_2' : data_2 ,
    ......
    }

    response_S : responses of same-distribution training data, real number

    Multi_response_A : dict, responses of diff-distribution training data, real number
    e.g.,
    Multi_response_A = {
    'response_A_1' :  response_1 ,
    'response_A_2' : response_2 ,
    ......
    }

    test : feature matrix of test data

    N: int, the number of estimators in TrAdaBoost_R2


    References
    ----------
    .. [1] section 4.1
    Pardoe, D., & Stone, P. (2010, June).
    Boosting for regression transfer.
    In Proceedings of the 27th International Conference
    on International Conference on Machine Learning (pp. 863-870).
    """

     # prepare trans_A
    trans_A = list(Multi_trans_A.values())[0]
    if len(Multi_trans_A) == 1:
        pass
    else:
        for i in range(len(Multi_trans_A)-1):
            p = i + 1
            trans_A = np.concatenate((trans_A, list(Multi_trans_A.values())[p]), axis=0)
    # prepare response_A
    response_A = list(Multi_response_A.values())[0]
    if len(Multi_response_A) == 1:
        pass
    else:
        for i in range(len(Multi_response_A)-1):
            p = i + 1
            response_A = np.concatenate((response_A, list(Multi_response_A.values())[p]), axi

    trans_data = np.concatenate((trans_A, trans_S), axis=0)
    trans_response = np.concatenate((response_A, response_S), axis=0)

    row_A = trans_A.shape[0]
    row_S = trans_S.shape[0]
    row_T = test.shape[0]

    if N > row_A:
```

```python
        print('The maximum of iterations should be smaller than ', row_A)

    test_data = np.concatenate((trans_data, test), axis=0)

    # Initialize the weights
    weights_A = np.ones([row_A, 1]) / row_A
    weights_S = np.ones([row_S, 1]) / row_S
    weights = np.concatenate((weights_A, weights_S), axis=0)

    bata = 1 / (1 + np.sqrt(2 * np.log(row_A / N)))

    # Save prediction response and bata_t
    bata_T = np.zeros([1, N])
    result_response = np.ones([row_A + row_S + row_T, N])

    # Save the prediction response of test data
    predict = np.zeros([row_T])
    print ('params initial finished.')
    print('='*60)

    trans_data = np.asarray(trans_data, order='C')
    trans_response = np.asarray(trans_response, order='C')
    test_data = np.asarray(test_data, order='C')

    for i in range(N):
        weights = calculate_P(weights)
        result_response[:, i] = base_regressor(trans_data, trans_response, test_data, weights
        error_rate = calculate_error_rate(response_S, result_response[row_A:row_A + row_S, i]
        # Avoiding overfitting
        if error_rate <= 1e-10 or error_rate > 0.5:
            N = i
            break
        bata_T[0, i] = error_rate / (1 - error_rate)
        print ('Iter {}-th result :'.format(i))
        print ('error rate :', error_rate, '|| bata_T :', error_rate / (1 - error_rate))
        print('-'*60)

        D_t = np.abs(np.array(result_response[:row_A + row_S, i]) - np.array(trans_response))
        # Changing the data weights of same-distribution training data
        for j in range(row_S):
            weights[row_A + j] = weights[row_A + j] * np.power(bata_T[0, i], -(np.abs(result_
        # Changing the data weights of diff-distribution training data
        for j in range(row_A):
            weights[j] = weights[j] * np.power(bata, np.abs(result_response[j, i] - response_
    for i in range(row_T):
        predict[i] = np.sum(
            result_response[row_A + row_S + i, int(np.floor(N / 2)):N]) / (N-int(np.floor(N /

    print("TrAdaBoost_R2 is done")
    print('='*60)
    print('The prediction responses of test data are :')
    print(predict)
    return predict

def calculate_P(weights):
    total = np.sum(weights)
    return np.asarray(weights / total, order='C')


def base_regressor(trans_data, trans_response, test_data, weights):
    """
    Base on sampling
    # weight resampling
    cdf = np.cumsum(weights)
    cdf_ = cdf / cdf[-1]
    uniform_samples = np.random.random_sample(len(trans_data))
```

```
    bootstrap_idx = cdf_.searchsorted(uniform_samples, side='right')
    # searchsorted returns a scalar
    bootstrap_idx = np.array(bootstrap_idx, copy=False)
    reg = DecisionTreeRegressor(max_depth=2,splitter='random',max_features="log2",random_state
    reg.fit(trans_data[bootstrap_idx], trans_response[bootstrap_idx])
    """
    reg = DecisionTreeRegressor(max_depth=1,splitter='random',max_features="log2",random_state
    reg.fit(trans_data, trans_response,sample_weight=weights[:,0])
    return reg.predict(test_data)


def calculate_error_rate(response_R, response_H, weight):
    total = np.abs(response_R - response_H).max()
    return np.sum(weight[:] * np.abs(response_R - response_H) / total)
```

1. Parameters:

- `trans_S` : Feature matrix of the source domain training data.
- `Multi_trans_A` : Dictionary containing feature matrices of different-distribution training data.
- `response_S` : Responses (target values) of same-distribution training data.
- `Multi_response_A` : Dictionary containing responses of different-distribution training data.
- `test` : Feature matrix of the test data.
- `N` : Number of weak tree regressors in TrAdaBoost_R2.

2. Prepare Data from Multiple Sources

- **Purpose**: Combines the source and auxiliary domain data into a single dataset for training.

```
trans_A = list(Multi_trans_A.values())[0]
if len(Multi_trans_A) == 1:
    pass
else:
    for i in range(len(Multi_trans_A)-1):
        p = i + 1
        trans_A = np.concatenate((trans_A, list(Multi_trans_A.values())[p]), axis=0)

response_A = list(Multi_response_A.values())[0]
if len(Multi_response_A) == 1:
    pass
else:
    for i in range(len(Multi_response_A)-1):
        p = i + 1
        response_A = np.concatenate((response_A, list(Multi_response_A.values())
[p]), axis=0)
```

3. Initialization of Weights

- Initializes weights for each instance in the source and target datasets, using the instance numbers for each dataset as the denominator, respectively. The weights are normalized to ensure they sum up to 1.

```
trans_data = np.concatenate((trans_A, trans_S), axis=0)
trans_response = np.concatenate((response_A, response_S), axis=0)

row_A = trans_A.shape[0]
row_S = trans_S.shape[0]
row_T = test.shape[0]
```

```
weights_A = np.ones([row_A, 1]) / row_A
weights_S = np.ones([row_S, 1]) / row_S
weights = np.concatenate((weights_A, weights_S), axis=0)
```

4. Parameter Initialization

- **Beta Calculation**: `bata` is a hyperparameter that controls the weight update for different-distribution source data, which is the same as on classification.

```
bata = 1 / (1 + np.sqrt(2 * np.log(row_A / N)))
```

5. Initialize Variables for Storing Results

- **Beta_T Calculation**: Different from `bata`, which is a stable preset rate, `bata_T` stores the beta values calculated from the new error rate for each iteration.
- `result_response` stores the predictions for every samples in each iteration, with n+m (source+target) rows and N (iteration) columns.

```
bata_T = np.zeros([1, N])
result_response = np.ones([row_A + row_S + row_T, N])
predict = np.zeros([row_T])
```

6. Iterative Boosting Process

```
for i in range(N):
    weights = calculate_P(weights)
    result_response[:, i] = base_regressor(trans_data, trans_response, test_data, 
weights)
    error_rate = calculate_error_rate(response_S, result_response[row_A:row_A + 
row_S, i], weights[row_A:row_A + row_S, 0])
    if error_rate <= 1e-10 or error_rate > 0.5:
        N = i
        break
    bata_T[0, i] = error_rate / (1 - error_rate)
```

- **Iteration**: The algorithm iterates `N` times, training a weak regressor in each iteration.
- **Normalize Weights**: `calculate_P(weights)` normalizes the weights so that they sum to 1.
- **Train Base Regressor**: `base_regressor` trains a decision tree regressor using the weighted data and makes predictions on the combined dataset.
- **Error Calculation**: The error rate is calculated for the same-distribution training data.
- **Early Stopping**: If the error rate is very low or exceeds 0.5, the algorithm stops early. Recall that in classification we have to make sure the error is smaller then 0.5, otherwise we flip the predictions and convert the errors.
- **Update Beta Values**: `bata_T` is calculated from the new error rate for each iteration.

8. Weight Update

```
D_t = np.abs(np.array(result_response[:row_A + row_S, i]) - 
np.array(trans_response)).max()
for j in range(row_S):
    weights[row_A + j] = weights[row_A + j] * np.power(bata_T[0, i], -
(np.abs(result_response[row_A + j, i] - response_S[j])/D_t))
for j in range(row_A):
    weights[j] = weights[j] * np.power(bata, np.abs(result_response[j, i] - 
response_A[j])/D_t)
```

- Adjusts the weights of the source and target instances based on their prediction accuracy.

9. Final Predictions

```
for i in range(row_T):
    predict[i] = np.sum(result_response[row_A + row_S + i, int(np.floor(N / 2)):N])
/ (N-int(np.floor(N / 2)))
```

- The final prediction for each test data point is the average of the predictions from the last half of the iterations.

Experiment

1. Load Data

In [35]:
```
# same-distribution training data
train_data = pd.read_csv('M_Sdata.csv')
# two diff-distribution training data
A1_train_data = pd.read_csv('M_Adata1.csv')
# test data
test_data = pd.read_csv('M_Tdata.csv')

Multi_trans_A = {
'trans_A_1' : A1_train_data.iloc[:,:-1],
}
Multi_response_A = {
'response_A_1' :  A1_train_data.iloc[:,-1] ,
}

trans_S = train_data.iloc[:,:-1]
response_S = train_data.iloc[:, -1]
test = test_data.iloc[:,:-1]
```

2. Data Inspection

In [36]:
```
print("Same-distribution training data: ")
print(train_data, "\n")
print("Diff-distribution training data: ")
print(A1_train_data, "\n")
print("Test data: ")
print(test_data, "\n")
```

```
Same-distribution training data:
     deta     G
0  0.1179  35.5
1  0.1306  39.7
2  0.1269  43.7

Diff-distribution training data:
      deta     G
0   0.1356  36.8
1   0.1325  36.7
2   0.1446  41.7
3   0.1338  34.5
4   0.1329  31.8
5   0.1326  34.0
6   0.1362  34.7
7   0.1361  38.7
8   0.1355  36.5
9   0.1360  35.0
10  0.1358  36.3
11  0.1356  36.1
12  0.1427  39.5
13  0.1401  35.3
14  0.1401  35.7

Test data:
     deta     G
0  0.1313  35.7
1  0.1250  39.2
```

3. Transfer Learning and Prediction

In [37]:
```python
N = 10

TrAdaBoost_R2(trans_S, Multi_trans_A, response_S, Multi_response_A, test,N)
```

```
params initial finished.
============================================================
Iter 0-th result :
error rate : 0.20726172465960674 || bata_T : 0.26145038167938944
------------------------------------------------------------
Iter 1-th result :
error rate : 0.4211373662764193 || bata_T : 0.7275255677973531
------------------------------------------------------------
TrAdaBoost_R2 is done
============================================================
The prediction responses of test data are :
[40.33843838 40.33843838]
```

Out[37]:
```
array([40.33843838, 40.33843838])
```

Introduction of Transfer Stacking

Transfer Stacking is a machine learning technique that combines the strengths of transfer learning and ensemble learning. It is particularly useful when dealing with datasets that have different distributions but share some underlying patterns or relationships. The goal is to leverage the knowledge from multiple source datasets (with different distributions) to improve the performance on a target dataset.

Key concepts:

- **Ensemble Learning**: It combines results of multiple models (weak learners) to improve the overall performance of the system.
- **Stacking**: Multiple models are combined with one after-trained meta-model to make predictions.

Function Definition and Explanation

In [39]:
```
def Transfer_Stacking(trans_S, Multi_trans_A, response_S, Multi_response_A, test,):
    """Boosting for Regression Transfer

    Please feel free to open issues in the Github : https://github.com/Bin-Cao/TrAdaboost
    or
    contact Bin Cao (bcao@shu.edu.cn)
    in case of any problems/comments/suggestions in using the code.

    Parameters
    ----------
    trans_S : feature matrix of same-distribution training data

    Multi_trans_A : dict, feature matrix of diff-distribution training data
    e.g.,
    Multi_trans_A = {
    'trans_A_1' :  data_1 ,
    'trans_A_2' : data_2 ,
    ......
    }

    response_S : responses of same-distribution training data, real number

    Multi_response_A : dict, responses of diff-distribution training data, real number
    e.g.,
    Multi_response_A = {
    'response_A_1' :  response_1 ,
    'response_A_2' : response_2 ,
    ......
    }

    test : feature matrix of test data


    References
    ----------
    .. [1] Pardoe, D., & Stone, P. (2010, June).
    Boosting for regression transfer.
    In Proceedings of the 27th International Conference
    on International Conference on Machine Learning (pp. 863-870).

    """
    # generate a pool of experts according the diff-dis datasets
    weak_classifiers_set = []
    reg = DecisionTreeRegressor(max_depth=2,splitter='random',max_features="log2",random_stat
    for source in range(len(Multi_trans_A)):
        trans_A = list(Multi_trans_A.values())[source]
        response_A = list(Multi_response_A.values())[source]

        trans_A = np.asarray(trans_A, order='C')
        response_A = np.asarray(response_A, order='C')

        weak_classifier = reg.fit(trans_A, response_A, )
        weak_classifiers_set.append(weak_classifier)
    print('A set of experts is initilized and contains {} classifier'.format(len(weak_classif
    print('='*60)
```

```python
    row_S = trans_S.shape[0]
    row_T = test.shape[0]
    print ('params initial finished.')

    X = np.array(trans_S)
    Y = np.array(response_S)
    LOOCV_LS_matrix = np.ones([row_S, len(weak_classifiers_set)+1])
    LOOCV_LS_matrix[:,-1] = LOOCV_output(X,Y)
    for j in range(len(weak_classifiers_set)):
        LOOCV_LS_matrix[:,j] = weak_classifiers_set[j].predict(X)

    # find the linear combination of hypotheses that minimizes squared error.
    reg = LinearRegression().fit(LOOCV_LS_matrix, Y)
    print('The linear combination of hypotheses is founded:')
    print('coef:', reg.coef_ ,'|| intercept :', reg.intercept_)
    coef = reg.coef_
    intercept = reg.intercept_
    # add the newly clf into the set
    weak_classifiers_set.append(reg.fit(X, Y))

    # save the prediction results of weak classifiers
    result_response = np.ones([row_T, len(weak_classifiers_set)])
    for item in range(len(weak_classifiers_set)):
        result_response[:,item] = weak_classifiers_set[item].predict(np.array(test))
    predict = np.ones(row_T) * intercept
    for j in range(len(coef)):
        predict += coef[j] * result_response[:,j]
    print('Transfer_Stacking is done')
    print('='*60)
    print('The prediction responses of test data are :')
    print(predict)
    return predict


def LOOCV_output(X,Y):
    loo = LeaveOneOut()
    reg = DecisionTreeRegressor(max_depth=2,splitter='random',max_features="log2",random_stat
    y_pre_loocv = []
    for train_index, test_index in loo.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, _ = Y[train_index], Y[test_index]
        weak_classifier_new = reg.fit(X_train, y_train)
        y_pre = weak_classifier_new.predict(X_test)
        y_pre_loocv.append(y_pre[0])
    return y_pre_loocv
```

1. Parameters:

- `trans_S` : Feature matrix of the source domain training data.
- `Multi_trans_A` : Dictionary containing feature matrices of different-distribution training data.
- `response_S` : Responses (target values) of same-distribution training data.
- `Multi_response_A` : Dictionary containing responses of different-distribution training data.
- `test` : Feature matrix of the test data.

2. Initialization of Weak Classifiers

- Multiple weak classifiers (experts) are trained on different source datasets (`Multi_trans_A`).
- Each source dataset has its own feature matrix and response vector.
- A decision tree regressor with a maximum depth of 2 is used as the base weak classifier.

```
weak_classifiers_set = []
reg = DecisionTreeRegressor(max_depth=2, splitter='random', max_features="log2",
random_state=0)
for source in range(len(Multi_trans_A)):
    trans_A = list(Multi_trans_A.values())[source]
    response_A = list(Multi_response_A.values())[source]
    weak_classifier = reg.fit(trans_A, response_A)
    weak_classifiers_set.append(weak_classifier)
```

3. Leave-One-Out Cross-Validation (LOOCV) Definition

- Each prediction on the target dataset is trained with LOOCV manner to avoid overfitting.

```
def LOOCV_output(X,Y):
    loo = LeaveOneOut()
    reg =
DecisionTreeRegressor(max_depth=2,splitter='random',max_features="log2",random_state=0)
    y_pre_loocv = []
    for train_index, test_index in loo.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, _ = Y[train_index], Y[test_index]
        weak_classifier_new = reg.fit(X_train, y_train)
        y_pre = weak_classifier_new.predict(X_test)
        y_pre_loocv.append(y_pre[0])
    return y_pre_loocv
```

4. Generating Result Table for Meta-Learning

- LOOCV is performed on the target dataset (`trans_S`) to generate a matrix of predictions from the weak classifiers.
- This matrix is used to train a meta-model (linear regression) to find the optimal combination of weak classifiers.

```
LOOCV_LS_matrix = np.ones([row_S, len(weak_classifiers_set) + 1])
LOOCV_LS_matrix[:, -1] = LOOCV_output(X, Y)
for j in range(len(weak_classifiers_set)):
    LOOCV_LS_matrix[:, j] = weak_classifiers_set[j].predict(X)
```

5. Training the Meta-Model

- A linear regression model is trained on the LOOCV matrix to find the optimal weights for combining the weak classifiers.
- The coefficients and intercept of the linear regression model are used to combine the predictions.

```
reg = LinearRegression().fit(LOOCV_LS_matrix, Y)
coef = reg.coef_
intercept = reg.intercept_
```

6. Prediction on Test Data

- All weak classifiers are used to make predictions on the test data.
- The final predictions are generated using the coefficients and intercept obtained from the meta-model.

```
result_response = np.ones([row_T, len(weak_classifiers_set)])
for item in range(len(weak_classifiers_set)):
    result_response[:, item] = weak_classifiers_set[item].predict(np.array(test))
```

```
predict = np.ones(row_T) * intercept
for j in range(len(coef)):
    predict += coef[j] * result_response[:, j]
```

Experiment

1. Load Data

In [40]:
```
# same-distribution training data
train_data = pd.read_csv('M_Sdata.csv')
# two diff-distribution training data
A1_train_data = pd.read_csv('M_Adata1.csv')
# test data
test_data = pd.read_csv('M_Tdata.csv')

Multi_trans_A = {
'trans_A_1' : A1_train_data.iloc[:,:-1],
}
Multi_response_A = {
'response_A_1' :  A1_train_data.iloc[:,-1] ,
}

trans_S = train_data.iloc[:,:-1]
response_S = train_data.iloc[:, -1]
test = test_data.iloc[:,:-1]
```

2. Prediction

In [41]:
```
Transfer_Stacking(trans_S, Multi_trans_A, response_S, Multi_response_A, test,)
```

```
A set of experts is initilized and contains 1 classifier
============================================================
params initial finished.
The linear combination of hypotheses is founded:
coef: [ 0.    -1.525] || intercept : 104.24250000000006
Transfer_Stacking is done
============================================================
The prediction responses of test data are :
[39.67453141 43.89090202]
```

Out[41]:
```
array([39.67453141, 43.89090202])
```

Key Differences from Boosting Strategy:

1. Source Data:

- **Transfer Stacking**: Uses all available data from the source datasets to train weak classifiers.
- **TrAdaBoost**: Also uses all available source data to train weak classifiers. However, it adjusts the weights of the source data samples based on their relevance to the target domain.

2. Target Data:

- **Transfer Stacking**: Uses LOOCV to optimize the combination of weak classifiers on the target dataset.
- **TrAdaBoost**: Focuses on reweighting the target data samples iteratively to improve the performance on the target domain. It does not use LOOCV but rather adjusts the weights of the samples based on their classification accuracy.

3. Objective:

- **Transfer Stacking**: The goal is to combine multiple weak classifiers from different source datasets using a meta-model to improve performance on the target dataset.
- **TrAdaBoost**: The goal is to adapt the source data to the target domain by iteratively reweighting the samples. This approach is more focused on sample-level adaptation rather than model-level.

4. Mechanism:

- **Transfer Stacking**: The current version does not involve any iteration process. All weak learnings are trained parallelly and then a final meta-model, facilitating a fast computation.
- **TrAdaBoost**: Involves several iteration to update sample weights to boost prediction results.

Introduction of Two-Stage TrAdaBoost_R2

The two-stage TrAdaBoost_R2 algorithm is proposed mitigate the decreased performance after a certain iteration in the original TrAdaBoost_R2 process.

Main Idea of Two-Stage TrAdaBoost.R2

Two-Stage TrAdaBoost.R2 is an advanced transfer learning algorithm designed to address the limitations of the original TrAdaBoost.R2, particularly the issue of performance degradation when the number of boosting iterations increases beyond a certain point. This degradation is often due to inappropriate updates of the point-wise weights of the source data, leading to overfitting or negative transfer.

The main idea of Two-Stage TrAdaBoost.R2 is to introduce a more controlled and systematic approach to updating the weights of the source data. This is achieved by dividing the training process into two distinct stages:

1. **First Stage (Fixed Source Weights)**:

   - In this stage, the point-wise weights of the source data are fixed, while the weights of the target training data are updated iteratively using the AdaBoost.R2 mechanism. This ensures that the initial focus is on learning from the target data, leveraging the source data as a supplementary resource.
2. **Second Stage (Adjusting Source Weights)**:

   - After the first stage, the algorithm uses k-fold cross-validation (e.g., LOOCV) to gradually adjust the weights of the source data downwards. This adjustment is done iteratively until the weights reach a certain threshold, ensuring that the source data contributes effectively without overwhelming the target data.

Problem Solved

The primary problem addressed by Two-Stage TrAdaBoost.R2 is the **inappropriate update of point-wise weights** in the original TrAdaBoost.R2 algorithm. When the number of boosting iterations increases, the weights of the source data may become too dominant, leading to overfitting to the source domain and poor generalization to the target domain. By fixing the source data weights in the first stage and carefully adjusting them in the second stage, Two-Stage TrAdaBoost.R2 mitigates this issue and improves the overall performance of the model.

Process of Two-Stage TrAdaBoost.R2

1. **Initialization**:

   - Assign initial weights to the source data and target data. Typically, the weights are set such that the sum of weights for the source data is 0.5 and the sum of weights for the target data is also 0.5.

2. **First Stage (Fixed Source Weights)**:

   - **Training**: Train the model using the combined dataset (source + target) with the initial weights. The source data weights remain fixed throughout this stage.
   - **Weight Update**: Update the weights of the target data iteratively using the AdaBoost.R2 mechanism. This involves calculating the error rate and adjusting the weights based on the performance of the model on the target data.

3. **Second Stage (Adjusting Source Weights)**:

   - **Cross-Validation**: Use k-fold cross-validation on the target data to evaluate the performance of the model.
   - **Weight Adjustment**: Gradually adjust the weights of the source data downwards based on the cross-validation results. This is done iteratively until the weights reach a certain threshold or until the performance on the target data no longer improves.

Function Definition

First-stage implementation

This is a revised version of the original TrAdaBoost_R2 algorithm, fixing weights of a proportion of samples, outputing errors based on mean squared error (MAE).

In [42]:
```
def AdaBoost_R2_T_rv(trans_S, response_S, test, weight,frozen_N, N = 20):
    """Boosting for Regression Transfer.

    Please feel free to open issues in the Github : https://github.com/Bin-Cao/TrAdaboost
    or
    contact Bin Cao (bcao@shu.edu.cn)
    in case of any problems/comments/suggestions in using the code.

    Parameters
    ----------
    trans_S : feature matrix

    response_S : response of training data, real values

    test : feature matrix of test data

    weights : initial data weights

    frozen_N : int, the weights of first [frozen_N] instances in trans_S  are never modified

    N : int, default=20, the number of weak estimators


    References
    ----------
    .. [1] Algorithm 3
    Pardoe, D., & Stone, P. (2010, June).
    Boosting for regression transfer.
    In Proceedings of the 27th International Conference
```

```
    on International Conference on Machine Learning (pp. 863-870).
    """

    trans_data =  copy.deepcopy(trans_S)
    trans_response =  copy.deepcopy(response_S)

    row_S = trans_S.shape[0]
    row_T = test.shape[0]

    test_data = np.concatenate((trans_data, test), axis=0)
    weights = copy.deepcopy(weight)
    # initilize data weights
    _weights = weights / sum(weights)

    # Save prediction responses and bata_t
    bata_T = np.zeros(N)
    result_response = np.ones([row_S + row_T, N])

    # Save the prediction responses of test data
    predict = np.zeros(row_T)

    trans_data = np.asarray(trans_data, order='C')
    trans_response = np.asarray(trans_response, order='C')
    test_data = np.asarray(test_data, order='C')

    for i in range(N):
        _weights = calculate_P(_weights,)
        result_response[:, i]  = train_reg(trans_data, trans_response, test_data, _weights)
        error_rate = calculate_error_rate(response_S, result_response[0: row_S, i],_weights)
        if error_rate > 0.5 or error_rate <= 1e-10: break

        # we try to define a beta_t > 1 for given worst case a higher weight than Source doma
        bata_T[i] =  (1 - error_rate) / error_rate

        # Changing the data weights of unfrozen training data
        D_t = np.abs(result_response[frozen_N:row_S, i] - response_S[frozen_N:row_S]).max()
        for j in range(row_S - frozen_N):
            _weights[frozen_N + j] = _weights[frozen_N + j] * np.power(bata_T[i], (1-np.abs(r


    Cal_res = result_response[row_S:,:]
    # Sort the predictions
    sorted_idx = np.argsort(Cal_res, axis=1)

    # Find index of median prediction for each sample
    weight_cdf = np.cumsum(bata_T[sorted_idx], axis=1)
    # return True - False
    median_or_above = weight_cdf >= 0.5 * weight_cdf[:, -1][:, np.newaxis]
    median_idx = median_or_above.argmax(axis=1)

    median_estimators = sorted_idx[np.arange(row_T), median_idx]
    for j in range(row_T):
        predict[j] = Cal_res[j,median_estimators[j]]

    train_predictions = median_prediction(result_response[:row_S,:],bata_T,row_S)
    return predict,_weights,train_predictions

def calculate_P(weights,):
    total = np.sum(weights)
    weights / total
    return np.asarray(weights, order='C')

def train_reg(trans_data, trans_response, test_data, weights):
    reg = DecisionTreeRegressor(max_depth=2,splitter='random',max_features="log2",random_stat
    reg.fit(trans_data, trans_response,sample_weight = weights)
    return reg.predict(test_data)
```

```python
def calculate_error_rate(response_R, response_H, weight):
    total = np.abs(response_R - response_H).max()
    return np.sum(weight[:] * np.abs(response_R - response_H) / total)

def median_prediction(_Cal_res,_bata_T,row_S):
    _predict = np.zeros(row_S)
    # Sort the predictions
    _sorted_idx = np.argsort(_Cal_res, axis=1)

    # Find index of median prediction for each sample
    _weight_cdf = np.cumsum(_bata_T[_sorted_idx], axis=1)
    # return True - False
    _median_or_above = _weight_cdf >= 0.5 * _weight_cdf[:, -1][:, np.newaxis]
    _median_idx = _median_or_above.argmax(axis=1)

    _median_estimators = _sorted_idx[np.arange(row_S), _median_idx]
    for j in range(row_S):
        _predict[j] = _Cal_res[j,_median_estimators[j]]
    return _predict
```

Implementation integraing Stage 1 and Stage 2

In [51]:

```python
def Two_stage_TrAdaboost_R2_rv(trans_S, Multi_trans_A, response_S, Multi_response_A, test, st
    """Boosting for Regression Transfer

    Please feel free to open issues in the Github : https://github.com/Bin-Cao/TrAdaboost
    or
    contact Bin Cao (bcao@shu.edu.cn)
    in case of any problems/comments/suggestions in using the code.

    Parameters
    ----------
    trans_S : feature matrix of same-distribution training data

    Multi_trans_A : dict, feature matrix of diff-distribution training data
    e.g.,
    Multi_trans_A = {
    'trans_A_1' :  data_1 ,
    'trans_A_2' : data_2 ,
    ......
    }

    response_S : responses of same-distribution training data, real number

    Multi_response_A : dict, responses of diff-distribution training data, real number
    e.g.,
    Multi_response_A = {
    'response_A_1' :  response_1 ,
    'response_A_2' : response_2 ,
    ......
    }

    test : feature matrix of test data

    steps_S: int, the number of steps (see Algorithm 3)

    N: int, the number of estimators in AdaBoost_R2_T


    References
    ----------
    .. [1] Algorithm 3
    Pardoe, D., & Stone, P. (2010, June).
```

```
    Boosting for regression transfer.
    In Proceedings of the 27th International Conference
    on International Conference on Machine Learning (pp. 863-870).

    """
    # prepare trans_A
    trans_A = list(Multi_trans_A.values())[0]
    if len(Multi_trans_A) == 1:
        pass
    else:
        for i in range(len(Multi_trans_A)-1):
            p = i + 1
            trans_A = np.concatenate((trans_A, list(Multi_trans_A.values())[p]), axis=0)
    # prepare response_A
    response_A = list(Multi_response_A.values())[0]
    if len(Multi_response_A) == 1:
        pass
    else:
        for i in range(len(Multi_response_A)-1):
            p = i + 1
            response_A = np.concatenate((response_A, list(Multi_response_A.values())[p]), axi

    trans_data = np.concatenate((trans_A, trans_S), axis=0)
    trans_response = np.concatenate((response_A, response_S), axis=0)

    row_A = trans_A.shape[0]
    row_S = trans_S.shape[0]

    # Initialize the weights
    weight  = np.ones(row_A+row_S)/(row_A+row_S)
    bata_T = np.zeros(steps_S)

    print ('params initial finished.')
    print('='*60)

    # generate a pool of AdaBoost_R2_T_rv
    AdaBoost_pre = []
    model_error = []
    warnings.filterwarnings('ignore')
    for i in range(steps_S):
        res_ , new_weight , train_predictions= AdaBoost_R2_T_rv(trans_data, trans_response, t
        AdaBoost_pre.append(res_)
        LOOCV_MSE = LOOCV_test(trans_data, trans_response, weight,row_A, N)
        model_error.append(LOOCV_MSE)
        # the finial weights are assigned to the training weights
        weight = new_weight
        """
        The paper says that:
        In addition, it is not necessary to progress through all S steps once it has been dete
        """

        if len(model_error) > 2 and model_error[-1] > model_error[-2] and model_error[-1] > m
            steps_S = i
            break


        # at the outlier loop, the weights are updated with the prediction of the best_base_e
        pre_res = copy.deepcopy(train_predictions)

        E_t = calculate_error_rate(trans_response, pre_res, weight)

        bata_T[i] =  E_t / (1 - E_t)

        # Changing the data weights of same-distribution training data
        total_w_S =  0.5 + 0.5 * i/(steps_S-1)
        weight[row_A : row_A+row_S] =  (weight[row_A : row_A+row_S] / weight[row_A : row_A+ro
```

```python
        # Changing the data weights of diff-distribution training data
        """
        # for saving computation power, we apply the strategy in MultiSourceTrAdaBoost to upda
        # see: 10.1109/CVPR.2010.5539857
        for j in range(row_A):
            weight[j] = weight[j] * np.exp(-bata_T[i] * np.abs(trans_response[j] - pre_res[j]
        weight[0:row_A] =  weight[0:row_A] * (1-total_w_S) / weight[0:row_A].sum()
        """
        beta_t = binary_search(total_w_S,weight,trans_response,pre_res,row_A,beta_t_range = (
        if beta_t == None:
            for j in range(row_A):
                weight[j] = weight[j] * np.exp(-bata_T[i] * np.abs(trans_response[j] - pre_re
            weight[0:row_A] =  weight[0:row_A] * (1-total_w_S) / weight[0:row_A].sum()
        else:
            D_t = np.abs(trans_response[0:row_A] - pre_res[0:row_A]).max()
            for j in range(row_A):
                weight[j] = weight[j] * np.power(beta_t, np.abs(trans_response[j] - pre_res[j
            weight[0:row_A] =  weight[0:row_A] * (1-total_w_S) / weight[0:row_A].sum()

        print('Iter {}-th result :'.format(i))
        print('{} AdaBoost_R2_T model has been instantiated :'.format(len(model_error)), '||
        print('The LOOCV MSE on TARGET DOMAIN DATA : ',LOOCV_MSE)
        print('The beta_t calculated by binary search is : ',beta_t)
        print('-'*60)

    model_error = np.array(model_error)
    min_index = np.random.choice(np.flatnonzero(model_error == model_error.min()))
    print('Two_stage_TrAdaboost_R2 is done')
    print('='*60)
    print('The minimum mean square error :',model_error[min_index])
    print('The prediction responses of test data are :')
    print(AdaBoost_pre[min_index])
    return AdaBoost_pre[min_index]


def LOOCV_test(trans_data, trans_response, weight,row_A, N):
    loo = LeaveOneOut()
    X = np.array(trans_data)
    Y = np.array(trans_response)
    y_pre_loocv = []
    cal = 0
    for train_index, test_index in loo.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, _ = Y[train_index], Y[test_index]
        w_train, _ = weight[train_index], weight[test_index]
        if cal <= row_A-1:
            y_pre,_ ,_= AdaBoost_R2_T_rv(X_train, y_train, X_test, w_train,row_A-1, N )
        else:
            y_pre,_ ,_= AdaBoost_R2_T_rv(X_train, y_train, X_test, w_train,row_A, N )
        y_pre_loocv.append(y_pre[0])
    return mean_squared_error(trans_response[row_A:],y_pre_loocv[row_A:])


def calculate_error_rate(response_R, response_H, weight):
    total = np.abs(response_R - response_H).max()
    return np.sum(weight[:] * np.abs(response_R - response_H) / total)

# binary_search strategy
def binary_search(total_w_S,__weight,trans_response,pre_res,row_A,beta_t_range = (0.01,1,0.01
    # beta_t_range is the search range of beta_t, default = (0.01,1,0.01)
    # viz., beta_t is searched in the interval of 0 to 1, with the step of 0.01 by binary_sear

    D_t = np.abs(trans_response[0:row_A] - pre_res[0:row_A]).max()
    _list = np.arange(beta_t_range[0],beta_t_range[1],beta_t_range[2])
    low = 0
    high = len(_list)-1
```

```
    while low <= high:
        weight = copy.deepcopy(__weight)
        mid = int(np.floor((low+high)/2))
        guess = _list[mid]
        # test beta_t
        for j in range(row_A):
            weight[j] = weight[j] * np.power(guess, np.abs(trans_response[j] - pre_res[j])/D_
        diff = (1-total_w_S) -  weight[0:row_A].sum()
        if abs(diff) <= tal:
            return guess
        # exceed the convergence crtiterion
        elif diff > 0:
            low = mid + 1
        else:
            high = mid -1

    print("UNABLE TO COVERGEE IN BINARY SEARCHING")
    return None
```

1. **Preparation of Data**

- The function `Two_stage_TrAdaboost_R2_rv` takes several inputs:
  - `trans_S` : Feature matrix of the source domain training data.
  - `Multi_trans_A` : A dictionary containing feature matrices of multiple auxiliary domain training data.
  - `response_S` : Responses (target values) of the source domain training data.
  - `Multi_response_A` : A dictionary containing responses of the auxiliary domain training data.
  - `test` : Feature matrix of the test data.
  - `steps_S` : The number of boosting steps, controlled by early stopping.
  - `N` : The number of estimators in the AdaBoost.R2 algorithm.

2. **Combining Data from Different Distributions**

- The code first combines the feature matrices and responses from all auxiliary domains into a single matrix `trans_A` and `response_A` .

```
trans_A = list(Multi_trans_A.values())[0]
if len(Multi_trans_A) == 1:
    pass
else:
    for i in range(len(Multi_trans_A)-1):
        p = i + 1
        trans_A = np.concatenate((trans_A, list(Multi_trans_A.values())[p]),
axis=0)
response_A = list(Multi_response_A.values())[0]
if len(Multi_response_A) == 1:
    pass
else:
    for i in range(len(Multi_response_A)-1):
        p = i + 1
        response_A = np.concatenate((response_A, list(Multi_response_A.values())
[p]), axis=0)
```

- The combined data from the source and the target domain are then concatenated into `trans_data` and `trans_response` .

```
trans_data = np.concatenate((trans_A, trans_S), axis=0)
trans_response = np.concatenate((response_A, response_S), axis=0)
```

3. **Initialization of Weights**

- The weights for the combined data are initialized to be uniform.
  `weight = np.ones(row_A + row_S) / (row_A + row_S)`
- An array `bata_T` is initialized to store the beta values for each boosting step.

4. **Boosting Process**

- The algorithm iterates for a specified number of steps (`steps_S`). In each iteration:
  - The `AdaBoost_R2_T_rv` function is called to train an AdaBoost.R2 model on the combined data, weighted by `weight` .
    ```
    res_, new_weight, train_predictions = AdaBoost_R2_T_rv(trans_data,
    trans_response, test, weight, row_A, N)
    ```
  - The model's performance is evaluated using Leave-One-Out Cross-Validation (LOOCV) Mean Squared Error (MSE).
    ```
    LOOCV_MSE = LOOCV_test(trans_data, trans_response, weight, row_A, N)
    ```
    Then weights are updated after the Stage 1 TrAdaBoost_R2 training:
    ```
    weight = new_weight
    ```
  - The model error is stored in `model_error` .
    ```
    model_error.append(LOOCV_MSE)
    ```

5. **Early Stopping**

- The algorithm checks if the model error has increased for the last **four** iterations. If so, it stops early to prevent overfitting.
  ```
  if len(model_error) > 2 and model_error[-1] > model_error[-2] and
  model_error[-1] > model_error[-3] and model_error[-1] > model_error[-4]:
      steps_S = i
      break
  ```

6. **Updating Weights**

- The weights are updated based on the model's predictions and errors.
  - The error rate `E_t` is calculated using the predictions and true responses.
    ```
    E_t = calculate_error_rate(trans_response, pre_res, weight)
    ```
  - The beta value for the current iteration is calculated as:
    ```
    bata_T[i] = E_t / (1 - E_t)
    ```
  - The weights for the target data are adjusted to gradually increase their importance over iterations.
    ```
    total_w_S = 0.5 + 0.5 * i / (steps_S - 1)
    weight[row_A:row_A + row_S] = (weight[row_A:row_A + row_S] /
    weight[row_A:row_A + row_S].sum()) * total_w_S
    ```
  - The weights for the source data are adjusted based on the prediction errors, using either exponential decay or a binary search strategy to find an optimal beta value.
    ```
    beta_t = binary_search(total_w_S, weight, trans_response, pre_res, row_A,
    beta_t_range=(0.01, 1, 0.01), tal=0.03)
    if beta_t is None:
        for j in range(row_A):
            weight[j] = weight[j] * np.exp(-bata_T[i] * np.abs(trans_response[j]
    - pre_res[j]))
        weight[0:row_A] = weight[0:row_A] * (1 - total_w_S) /
    weight[0:row_A].sum()
    else:
        D_t = np.abs(trans_response[0:row_A] - pre_res[0:row_A]).max()
        for j in range(row_A):
            weight[j] = weight[j] * np.power(beta_t, np.abs(trans_response[j] -
    pre_res[j]) / D_t)
    ```

```
        weight[0:row_A] = weight[0:row_A] * (1 - total_w_S) /
    weight[0:row_A].sum()
```

7. **Output and Monitoring**

- After each iteration, the algorithm prints the iteration number, the number of instantiated AdaBoost.R2 models, the error rate `E_t`, and the LOOCV MSE on the target domain data.
  ```
  print('Iter {}-th result :'.format(i))
  print('{} AdaBoost_R2_T model has been instantiated :'.format(len(model_error)),
  '|| E_t :', E_t)
  print('The LOOCV MSE on TARGET DOMAIN DATA : ', LOOCV_MSE)
  ```

Experiment

1. Load Data

In [44]:
```
# same-distribution training data
train_data = pd.read_csv('M_Sdata.csv')
# two diff-distribution training data
A1_train_data = pd.read_csv('M_Adata1.csv')
# test data
test_data = pd.read_csv('M_Tdata.csv')

Multi_trans_A = {
'trans_A_1' : A1_train_data.iloc[:,:-1],
}
Multi_response_A = {
'response_A_1' :  A1_train_data.iloc[:,-1] ,
}

trans_S = train_data.iloc[:,:-1]
response_S = train_data.iloc[:, -1]
test = test_data.iloc[:,:-1]
```

2. Prediction

In [52]:
```
steps_S = 30
N = 10

Two_stage_TrAdaboost_R2_rv(trans_S, Multi_trans_A, response_S, Multi_response_A, test, steps_
```

```
params initial finished.
============================================================
Iter 0-th result :
1 AdaBoost_R2_T model has been instantiated : || E_t : 0.2972424979724242
The LOOCV MSE on TARGET DOMAIN DATA :  51.04320000000001
The beta_t calculated by binary search is :  0.12
------------------------------------------------------------
Iter 1-th result :
2 AdaBoost_R2_T model has been instantiated : || E_t : 0.0894549369985037
The LOOCV MSE on TARGET DOMAIN DATA :  52.50215389042386
The beta_t calculated by binary search is :  0.75
------------------------------------------------------------
Iter 2-th result :
3 AdaBoost_R2_T model has been instantiated : || E_t : 0.08174647150155341
The LOOCV MSE on TARGET DOMAIN DATA :  52.091226764554456
The beta_t calculated by binary search is :  0.75
------------------------------------------------------------
Iter 3-th result :
4 AdaBoost_R2_T model has been instantiated : || E_t : 0.0749525946599858
The LOOCV MSE on TARGET DOMAIN DATA :  51.692426360108776
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 4-th result :
5 AdaBoost_R2_T model has been instantiated : || E_t : 0.047997119231908116
The LOOCV MSE on TARGET DOMAIN DATA :  50.56281043081474
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 5-th result :
6 AdaBoost_R2_T model has been instantiated : || E_t : 0.05691141272687233
The LOOCV MSE on TARGET DOMAIN DATA :  50.13845858845619
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 6-th result :
7 AdaBoost_R2_T model has been instantiated : || E_t : 0.03742428912893063
The LOOCV MSE on TARGET DOMAIN DATA :  48.794972405232706
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 7-th result :
8 AdaBoost_R2_T model has been instantiated : || E_t : 0.04520321224262851
The LOOCV MSE on TARGET DOMAIN DATA :  49.006931138866754
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 8-th result :
9 AdaBoost_R2_T model has been instantiated : || E_t : 0.04063799830368488
The LOOCV MSE on TARGET DOMAIN DATA :  48.53875327513325
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 9-th result :
10 AdaBoost_R2_T model has been instantiated : || E_t : 0.027105980639629778
The LOOCV MSE on TARGET DOMAIN DATA :  47.325052095761144
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 10-th result :
11 AdaBoost_R2_T model has been instantiated : || E_t : 0.024444707327448672
The LOOCV MSE on TARGET DOMAIN DATA :  47.57724949433313
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 11-th result :
12 AdaBoost_R2_T model has been instantiated : || E_t : 0.02980620244041362
The LOOCV MSE on TARGET DOMAIN DATA :  46.53369162242506
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 12-th result :
13 AdaBoost_R2_T model has been instantiated : || E_t : 0.01982968172516262
The LOOCV MSE on TARGET DOMAIN DATA :  47.05043432186536
The beta_t calculated by binary search is :  0.5
```

```
------------------------------------------------------------
Iter 13-th result :
14 AdaBoost_R2_T model has been instantiated : || E_t : 0.01780078177378222
The LOOCV MSE on TARGET DOMAIN DATA :  45.850692829297884
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 14-th result :
15 AdaBoost_R2_T model has been instantiated : || E_t : 0.015925951890199852
The LOOCV MSE on TARGET DOMAIN DATA :  45.54552018677635
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Iter 15-th result :
16 AdaBoost_R2_T model has been instantiated : || E_t : 0.014189816156888442
The LOOCV MSE on TARGET DOMAIN DATA :  45.26342099203487
The beta_t calculated by binary search is :  0.5
------------------------------------------------------------
Two_stage_TrAdaboost_R2 is done
============================================================
The minimum mean square error : 45.26342099203487
The prediction responses of test data are :
[36.07335695 36.07335695]
```

```
Out[52]:  array([36.07335695, 36.07335695])
```