STA 561 Homework 3

February 16, 2023

Authors
- Alonso Guerrero Castaneda (UID: 1194613)
- Eli Gnesin (UID: 1172961)
- Tommy Misikoff (UID: 1166813)
- Sanskriti Purohit (UID: 1179957)
- Will Tirone (UID: 1130904)

TA: Rick Presman

The homework is divided into two parts within the *.ipynb* file.

1. `Function Definition.`
2. `Function Implementation.`

Function Definition

The function *tune_bb* is defined to take inputs to automate the tuning of blackbox regression methods, more information on the inputs is included within the function definiton. To ensure stability of the function, error cases are defined. Further, the training data is standardized to make it internally consistent. Following which regularization-specific tuning is mapped out where-in the k-fold (with a user-defined value for k) cross validation is performed to achieve optimum values for the parameters which are used to generate more data. Once, the optimum parameters are achieved and more data is generated, the existing training data is transformed and the tuned model is fitted and returned.

Function Implementation

The function is implemented on the *iris* dataset to demonstrate the function's use on two regression methods, Ridge Regression and Linear Regression with MSE and MAD/MAE as their criterions respectively. It can be observed that the coeffecients and the criterions obtained by all 3 methods when applied the two models achieve approximately the same result, with minor deviations.

Similarly, the function can be implemented on different datasets with other blackbox models to optimize a specified criterion.

[1]:
```
import numpy as np
import math
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import LinearSVR
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.preprocessing import StandardScaler
```

```python
from sklearn import datasets
from sklearn.metrics import mean_squared_error, mean_absolute_error
import scipy.stats as st
```

[2]:
```python
def tune_bb(algo, X, y, regularization="Dropout", M=10, c=None, K=5,␣
 ↪criterion="MSE"):

    """function to automatically tune blackbox regression model

    Parameters:
    -----------

    algo : callable
        A learning algorithm that takes as input a matrix X in R nxp
        and a vector of responses Y in Rn and returns a function that
        maps inputs to outputs. Must have methods like .fit() and .predict()
    X : array-like of shape (n,p)
        training data X in R nxp
    y : array-like of shape (n,)
        training labels, Y, in Rn
    regularization : str, default="Dropout"
        regularization method, can be any of "Dropout",
        "NoiseAddition", or "Robust"
    M : int, default=
        A positive integer indicating the number of Monte Carlo
        replicates to be used if the method specified is Dropout or
        NoiseAddition
    c : default=None
        A vector of column bounds to be used if method specified is "Robust"
    K : int, default=5
        A positive integer indicating the number of CV-folds to be used to
        tune the amount of regularization, e.g., K = 5 indicates five-fold CV
    criterion : str, default="MSE"
        A criterion to be used to evaluate the method that belongs to the set
        {MSE, MAD} where MSE encodes mean square error and MAD encodes mean
        absolute deviation.

    Returns:
    -----------
    tuned_model : callable
        A tuned predictive model that optimizes the specific criterion using
        the specified method
    """

    # statements here to ensure model has the methods we need to tune it
    assert hasattr(algo, "fit"), "model object must have .fit() method"
    assert hasattr(algo, "predict"), "model object must have .predict() method"
```

```
    if criterion == "MSE":
        criterion = "neg_mean_squared_error"
    elif criterion == "MAE":
        criterion = "neg_mean_absolute_error"
    else:
        raise ValueError("Please input either MAE or MSE for criterion.")

    # Standardize X (useful for all methods with regularization)
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    if regularization == "Dropout":

        # parameter to tune
        phi_range = np.linspace(0,1,101)
        min_metric = None
        best_phi = None
        for phi in phi_range:
            metric = []
            # CV over K Folds
            for m in range(M):
                dropout_matrix = np.random.binomial(1,phi,size=X.shape) * X
                kf = KFold(n_splits=K)
                for train_index, test_index in kf.split(dropout_matrix):
                        X_train, y_train = dropout_matrix[train_index], y[train_index]
                        X_test, y_test = dropout_matrix[test_index], y[test_index]
                        model = algo
                        model.fit(X_train, y_train)
                        if criterion == "neg_mean_squared_error":
                            metric.append(mean_squared_error(y_test, model.predict(X_test)))
                        else:
                            metric.append(mean_absolute_error(y_test, model.predict(X_test)))

            new_metric = np.mean(metric)
            if (min_metric == None or new_metric < min_metric):
                best_phi = phi
                min_metric = new_metric

        # make a dropout matrix with the best choice of phi
        dropout_matrix = np.random.binomial(1,best_phi,size=X.shape) * X
        model = algo
        tuned_model = model.fit(dropout_matrix, y)
```

```
    elif regularization == "NoiseAddition":
        # possible levels of noise
        lambda_levels = np.linspace(0, 5, 101)
        min_metric = None
        best_lambda = None
        for lam in lambda_levels:
            #CV
            metric = []
            for m in range(M):
                # generate noise matrix with lambda (variance, not std)
                Z = np.random.normal(0, lam**2, size=X.shape)
                kf = KFold(n_splits=K)
                for train_index, test_index in kf.split(X):
                    X_train, y_train = X[train_index], y[train_index]
                    X_test, y_test = X[test_index], y[test_index]
                    X_train_disturbed = X_train + Z[train_index]
                    model = algo
                    model.fit(X_train_disturbed, y_train)
                    if criterion == "neg_mean_squared_error":
                        metric.append(mean_squared_error(y_test, model.
↪predict(X_test)))
                    else:
                        metric.append(mean_absolute_error(y_test, model.
↪predict(X_test)))
            new_metric = np.mean(metric)
            if (min_metric == None or new_metric < min_metric):
                best_lambda = lam
                min_metric = new_metric

        Z = np.random.normal(0, best_lambda**2, size=X.shape)
        new_X = X + Z
        model = algo
        tuned_model = model.fit(new_X, y)

    elif regularization == "Robust":
        tol = False #Are we in our tolerance range
        toler = 5e-4
        wts = [x/2 for x in c] #Initial weights are going to be c/2
        oerror = np.inf
        merror = np.inf
        itera = 0
        while not tol:
            #Create a bunch of matrices and choose the best by a score
            maxmatrix = None
            maxnorm = -np.inf
            for i in range(1000):
```

```
                matrix = np.random.rand(X.shape[0], X.shape[1])
                for m in range(matrix.shape[1]):
                    matrix[:,m] = (wts[m] / np.linalg.norm(matrix[:,m], 2)) * matrix[:,m]
                fnorm = np.linalg.norm(matrix, 2) #The criteria I'm using here is the two-norm
                if fnorm > maxnorm:
                    maxnorm = fnorm
                    maxmatrix = matrix
            new_X = X + maxmatrix #We add the permuted matrix to our design matrix

            #kfold cross validation
            errors = np.abs(cross_val_score(algo, new_X, y, cv=K, scoring=criterion))
            merror = np.mean(errors)
            if abs(oerror - merror) > toler:
                oerror = merror
                #Set our new weights for the next iteration
                wts = np.minimum(wts + (np.random.normal(size = len(wts)) * math.exp(-itera/2)), c)
                wts = np.maximum(0.1, wts) #weights can't be negative
                itera += 1
            else:
                tol = True

        #Once we have our best c bounds, let's use them exactly to construct the best model
        maxmatrix = None
        maxnorm = -np.inf
        for i in range(10000):
            matrix = np.random.rand(X.shape[0], X.shape[1])
            for m in range(matrix.shape[1]):
                matrix[:,m] = (wts[m] / np.linalg.norm(matrix[:,m], 2)) * matrix[:,m]
            fnorm = np.linalg.norm(matrix, 2) #The criteria I'm using here is the two-norm
            if fnorm > maxnorm:
                maxnorm = fnorm
                maxmatrix = matrix
        new_X = X + maxmatrix #We add the permuted matrix to our design matrix

        model = algo
        tuned_model = model.fit(new_X, y)
    else:
```

```
        raise ValueError('Please input one of of "Dropout", "NoiseAddition", or␣
↪"Robust"')

    return tuned_model
```

Function Demonstration

Below we fit our function on the iris data and show that the three regularization methods give identical answers for the coefficients.

[3]:
```
# use this to view doc string
?tune_bb
```

```
Signature:
tune_bb(
    algo,
    X,
    y,
    regularization='Dropout',
    M=10,
    c=None,
    K=5,
    criterion='MSE',
)
Docstring:
function to automatically tune blackbox regression model

Parameters:
-----------

algo : callable
    A learning algorithm that takes as input a matrix X in R nxp
    and a vector of responses Y in Rn and returns a function that
    maps inputs to outputs. Must have methods like .fit() and .predict()
X : array-like of shape (n,p)
    training data X in R nxp
y : array-like of shape (n,)
    training labels, Y, in Rn
regularization : str, default="Dropout"
    regularization method, can be any of "Dropout",
    "NoiseAddition", or "Robust"
M : int, default=
    A positive integer indicating the number of Monte Carlo
```

```
    replicates to be used if the method specified is Dropout or
    NoiseAddition
c : default=None
    A vector of column bounds to be used if method specified is "Robust"
K : int, default=5
    A positive integer indicating the number of CV-folds to be used to
    tune the amount of regularization, e.g., K = 5 indicates five-fold CV
criterion : str, default="MSE"
    A criterion to be used to evaluate the method that belongs to the set
    {MSE, MAD} where MSE encodes mean square error and MAD encodes mean
    absolute deviation.

Returns:
-----------
tuned_model : callable
    A tuned predictive model that optimizes the specific criterion using
    the specified method
File:      c:\users\elign\appdata\local\temp\ipykernel_32024\1101274168.py
Type:      function
```

[4]:
```
# getting full iris data set to train our model
X, y = datasets.load_iris(return_X_y=True)
```

[5]:
```
# An example of all 3 regularization types with Ridge() and MSE
Robust_Ridge = tune_bb(Ridge(),
                       X,
                       y,
                       regularization="Robust",
                       c = [4,5,4,3],
                       criterion="MSE")

Dropout_Ridge = tune_bb(Ridge(),
                        X,
                        y,
                        regularization="Dropout",
                        criterion="MSE")

Noise_Ridge = tune_bb(Ridge(),
                      X,
                      y,
                      regularization="NoiseAddition",
                      criterion="MSE")

print("Robust_Ridge Regression Coefficients : ", Robust_Ridge.coef_)
print("Dropout_Ridge Regression Coefficients : ", Dropout_Ridge.coef_)
print("Noise_Ridge Regression Coefficients : ", Noise_Ridge.coef_)
```

```
rr = Robust_Ridge.predict(X)
dr = Dropout_Ridge.predict(X)
nr = Noise_Ridge.predict(X)

print()
print("Robust_Ridge MSE : ", mean_squared_error(y, rr))
print("Dropout_Ridge MSE : ", mean_squared_error(y, dr))
print("Noise_Ridge MSE : ", mean_squared_error(y, nr))
```

```
Robust_Ridge Regression Coefficients :  [-0.07314516 -0.02833511  0.41251747
0.42918069]
Dropout_Ridge Regression Coefficients :  [-0.07346142 -0.02451997  0.37922144
0.46389549]
Noise_Ridge Regression Coefficients :  [-0.07381698 -0.02439013  0.38086222
0.46235169]

Robust_Ridge MSE :  2.214925565757384
Dropout_Ridge MSE :  2.263664317895736
Noise_Ridge MSE :  2.271619768722753
```

[6]:

```
# An example of all 3 regularization types with LinearRegression() and MAE
Robust_Linear = tune_bb(LinearRegression(),
                        X,
                        y,
                        regularization="Robust",
                        c = [4,5,4,3],
                        criterion="MAE")

Dropout_Linear = tune_bb(LinearRegression(),
                         X,
                         y,
                         regularization="Dropout",
                         criterion="MAE")

Noise_Linear = tune_bb(LinearRegression(),
                       X,
                       y,
                       regularization="NoiseAddition",
                       criterion="MAE")

print("Robust_Linear Regression Coefficients : ", Robust_Linear.coef_)
print("Dropout_Linear Regression Coefficients : ", Dropout_Linear.coef_)
print("Noise_Linear Regression Coefficients : ", Noise_Linear.coef_)

rl = Robust_Linear.predict(X)
dl = Dropout_Linear.predict(X)
```

```
nl = Noise_Linear.predict(X)

print()
print("Robust_Linear MAE : ", mean_absolute_error(y, rl))
print("Dropout_Linear MAE : ", mean_absolute_error(y, dl))
print("Noise_Linear MAE : ", mean_absolute_error(y, nl))
```

```
Robust_Linear Regression Coefficients :  [-0.0896758  -0.01689165  0.42533297
0.43754402]
Dropout_Linear Regression Coefficients :  [-0.09235605 -0.01741097  0.40227899
0.46284429]
Noise_Linear Regression Coefficients :  [-0.09127121 -0.01827394  0.40174911
0.46173039]

Robust_Linear MAE :  1.50591354209955
Dropout_Linear MAE :  1.4739707410920417
Noise_Linear MAE :  1.474168983177445
```

Resources and Notes:

1. https://www.statology.org/k-fold-cross-validation-in-python/