**import** **xgboost** **as** **xgb**

**from** **sklearn.datasets** **import** load\_boston

**from** **sklearn.model\_selection** **import** train\_test\_split

**from** **sklearn.model\_selection** **import** cross\_val\_score, KFold

**from** **sklearn.metrics** **import** mean\_squared\_error

**import** **matplotlib.pyplot** **as** **plt**

boston = load\_boston()

x, y = boston.data, boston.target

xtrain, xtest, ytrain, ytest=train\_test\_split(x, y, test\_size=0.15)

**Defining and fitting the model**  
  
For the regression problem, we'll use the XGBRegressor class of the xgboost package and we can define it with its default parameters. You can also set the new parameter values according to your data characteristics.

xgbr = xgb.XGBRegressor(verbosity=0)

print(xgbr)

XGBRegressor(base\_score=0.5, booster='gbtree', colsample\_bylevel=1,

colsample\_bynode=1, colsample\_bytree=1, gamma=0,

importance\_type='gain', learning\_rate=0.1, max\_delta\_step=0,

max\_depth=3, min\_child\_weight=1, missing=None, n\_estimators=100,

n\_jobs=1, nthread=None, objective='reg:linear', random\_state=0,

reg\_alpha=0, reg\_lambda=1, scale\_pos\_weight=1, seed=None,

silent=None, subsample=1, verbosity=1)

Next, we'll fit the model with train data.

xgbr.fit(xtrain, ytrain)

**Predicting and checking the results**

After training the model, we'll check the model training score.  
  
score = xgbr.score(xtrain, ytrain)

print("Training score: ", score)

Training score: 0.9738225090795732

We can also apply the cross-validation method to evaluate the training score.

scores = cross\_val\_score(xgbr, xtrain, ytrain,cv=10)

print("Mean cross-validation score: %.2f" % scores.mean())

Mean cross-validataion score: 0.87

Or if you want to use the KFlold method in cross-validation it goes as below.  
  
kfold = KFold(n\_splits=10, shuffle=**True**)

kf\_cv\_scores = cross\_val\_score(xgbr, xtrain, ytrain, cv=kfold )

print("K-fold CV average score: %.2f" % kf\_cv\_scores.mean())

K-fold CV average score: 0.87

Both methods show that the model is around 87 %  accurate on average.  
  
Next, we can predict test data, then check the prediction accuracy. Here, we'll use MSE and RMSE as accuracy metrics.

ypred = xgbr.predict(xtest)

mse = mean\_squared\_error(ytest, ypred)

print("MSE: %.2f" % mse)

MSE: 3.35

print("RMSE: %.2f" % (mse\*\*(1/2.0)))

RMSE: 1.83

Finally, we'll visualize the original and predicted test data in a plot to compare visually.

x\_ax = range(len(ytest))

plt.plot(x\_ax, ytest, label="original")

plt.plot(x\_ax, ypred, label="predicted")

plt.title("Boston test and predicted data")

plt.legend()

plt.show()

Compare with linear regression

LINEAR

from sklearn.linear\_model import LinearRegression

linear\_model = LinearRegression()

linear\_model.fit(X\_train, y\_train)

print(linear\_model.coef\_)

**XGBoost Regression with gblinear:**

from xgboost import XGBRegressor

xgb\_model = XGBRegressor(**n\_estimators=100, learning\_rate=0.06, gamma=1, subsample=0.8, objective='reg:squarederror', booster='gblinear', n\_jobs=-1**)

xgb\_model.fit(X\_train, y\_train)

print(xgb\_model.coef\_)

Output:

|  |  |  |
| --- | --- | --- |
| Linear  print(linear\_model.coef\_) | XGBRegression  print(xgb\_model.coef\_) | Diferencia |
| -1.31E-01 | -0.192631 | 0.06 |
| 4.94E-02 | 0.0966579 | -0.05 |
| 1.10E-03 | -0.00972393 | 0.01 |
| 2.71E+00 | 0.34198 | **2.36** |
| -1.60E+01 | 0.159105 | **-16.12** |
| 3.41E+00 | 1.09779 | **2.32** |
| 1.12E-03 | 0.039317 | -0.04 |
| -1.49E+00 | 0.289027 | -1.78 |
| 3.64E-01 | -0.00622574 | 0.37 |
| -1.32E-02 | 0.00236915 | -0.02 |
| -9.52E-01 | 0.171237 | -1.12 |
| 1.17E-02 | 0.0164343 | 0.00 |
| -5.94E-01 | -0.398639 | -0.20 |

While indeed the linear booster is using a linear routine and not a tree-based routine as a base learner, most of the similarities between a GBM with a linear booster and a linear regression model end there.

For starters, as you correctly note, within XGBoost we use (usually parallel) [coordinate descent](https://en.wikipedia.org/wiki/Coordinate_descent) optimization instead of "standard" gradient descent. This will likely give us, different results. In addition to that, the learning rate, ηη, will itself regularise the estimates inadvertently so there is not direct analogy to "no regularisation" even if we assume reg\_lambda, reg\_lambda\_bias and reg\_alpha (weights L2L2, baseline L2L2 and weights L1L1 regularisation respectively) to equate to zero. Finally, an XGBoost model and a linear regression model will not have the same intercept, β0β0: while in the case of a standard linear regression the intercept is calculated as part of the overall design matrix XX, the intercept used by XGBoost will depend on the learning rate, ηη as well as the mean of the response variable (i.e. we start boosting using as baseline the mean of our response variable or a regularised version of that mean to get our first estimates).

I do not even touch on the case of multithreading multiple estimators because this concerns the reproducibility of your results and not their interpretation per se. I also do not examine the case of multiple iterations (n\_estimators>1) as again, our estimates in this case might be affected by variation due to bagging. (Please note, that in the example code provided the subsample ratio of the training instances is set to 0.8, theoretically it should be set to 1 for the purposes of this comparison.) Finally, we should be aware that if we optimise for a sufficient large number of iterations, we will get the slope estimates that we would get from linear regression. After all, both XGBoost and LR will minimize the same cost function for the same data using the same slope estimates! :)

And to address your final question: yes, the interpretation of the XGBoost slope coefficient β1β1 as the "mean change in the response variable for one unit of change in the predictor variable while holding other predictors in the model constant." is correct. Putted differently: if two instances sAsA and sBsB only different by a unit change in a particular response variable x1x1 and are otherwise the same, the difference in our predictions for them will be the one quantified in the XGBoost β1β1 coefficient. Just the regularisation steps within XGBoost make the derivation of that intercept and slope coefficients totally incomparable with the derivation of the same intercept and slope coefficients within a linear regression framework.