MLR_set(***X***, ***y***, ***method***=*'OLS'*, ***detrend***=*True*, ***standardize***=*True*, ***weights***=*None*, ***calibrate***=*True*, ***calibration_X***=*None*, ***calibration_y***=*None*, ***fit_intercept***=*False*, ***EN_selection***=*'random'*, ***ridge_solver***=*'svd'*, ***l1_ratio***=*0.5*, ***alpha***=*1*, ***n_PLS_components***=*5*, ***return_dynorm_dxnorm***=*False*, ***random_seed***=*42*)

Performs multilinear regression (MLR) using one of the supported methods on the training data with the hyperparameters set by the user. Makes the basic assumption that the system can be described linearly as **y=X@dy_dX + ε**. Returns the vector of predictor coefficients **dy_dX**.

Parameters: .

X : ***ndarray of shape (n_samples,n_features)***

The array that contains the training predictor variables [for example, could be in the shape (time x space)]. The first dimension should match the length of **y**, and the second dimension will be the shape of the MLR coefficient solution vector **dy_dX.**

y : ***ndarray of shape (n_samples)***

The 1-Dimensional vector that contains the training predictand variable over samples (e.g. timeseries). The length should match the first dimension of **X.**

method : ***{'OLS', 'MCA', 'RIDGE', 'EN', 'EN_RIDGE', 'LASSO', 'PLS'}, default='OLS'***

The multilinear regression method used:

- 'OLS': Ordinary least squares regression implemented using np.linalg.lstsq. This version either minimizes the Euclidean 2-norm ||y - X@dy_dX||^2_2 (the definition of least-squares), or if there are multiple minimizing solutions, the one with the smallest 2-norm ||dy_dX||^2_2 is returned (the definition of the minimum-norm solution). This method requires no hyperparameters.

- 'MCA': Maximum covariance analysis (identical to PLS with 1 component). This is the pattern in spatial variable that explains the maximum fraction of the covariance between the spatial variable and the scalar variable. This is as implemented in Bretherton et al., 1992. As an aside, the relative

magnitudes of solution coefficients is identical to that of ridge regression as the ridge penalty approaches infinity. This method requires no hyperparameters.

- 'RIDGE': Ridge regression as implemented using sklearn.linear_model.Ridge. Ridge regression minimizes the objective function ||y - X@dy_dX||^2_2 + alpha * ||dy_dX||^2_2. This method requires the hyperparameter **alpha**, the penalty on the squared 2-norm.

- 'EN': Elastic net (applying both LASSO and ridge regularization penalties) implemented using sklearn.linear_model.ElasticNet. Elastic net minimizes the objective function ||y - X@dy_dX||^2_2 + a * ||dy_dX||_1 + 0.5 * b * ||dy_dX||^2_2. The LASSO and ridge penalties (*a* and *b* respectively) are related to the hyperparameters **alpha** and **l1_ratio** by alpha = a + b and l1_ratio = a / (a + b). Hence, this method requires the hyperparameters **alpha** and **l1_ratio.**

- 'EN_RIDGE': Ridge regression implemented by setting the **l1_ratio** in sklearn.linear_model.ElasticNet equal to zero (0). This results in essentially the same solution as the option 'RIDGE', but has slightly different optimization behavior so may be better for small sample sizes. This method requires the hyperparameter **alpha**.

- 'LASSO': LASSO regression implemented by setting the **l1_ratio** in sklearn.linear_model.ElasticNet equal to one (1). This method requires the hyperparameter **alpha**.

- 'PLS': Partial least squares regression as implemented using sklearn.cross_decomposition.PLSRegression. This essentially applies the MCA method iteratively and keeps each subsequent iteration as a "component" in the final solution coefficient vector. This method requires the hyperparameter **n_PLS_components.**

detrend : ***bool, default=True***

When set to 'True', detrends the **X** and **y** data linearly along the n_samples axis. This removes both the slope and the mean from the series.

standardize : ***bool, default=True***

When set to ‘True’, standardizes the **X** and **y** data linearly along the n_samples axis. This removes the mean across the n_samples axis, then divides by the standard deviation across the n_samples axis. The resulting modified **X** and **y** are thus in units of sigma (standard deviation).

weights : ***ndarray of shape (n_features), default=None***

Weights to be applied to the predictor variables in **X** along the n_features axis. For example, cosine latitude weighting should be applied here if predictors are gridpoint locations in a rectilinear grid.

calibrate : ***bool, default=True***

When set to ‘True’, this option rescales the solution vector **dy_dX** to enforce that it recreates the variance in **y** when projected onto **X**. This is analogous to orthogonal regression in single variable regression. I.e., the solution is modified: **dy_dX * std(y) / std(X@dy_dX)**.

calibration_X : ***ndarray of shape (n_samples,n_features), default=None***

Always use in concert with **calibration_y**. When left as ‘None’, **calibrate** will simply use **X** and **y** to rescale the solution. When **calibration_X** and **calibration_y** are provided, the solution will instead rescale via **dy_dX * std(calibration_y) / std(calibration_X@dy_dX)**. This is a niche application but may be useful to apply a regression trained on one timescale to a different timescale.

calibration_y : ***ndarray of shape (n_samples), default=None***

Always use in concert with **calibration_X**. When left as ‘None’, **calibrate** will simply use **X** and **y** to rescale the solution. When **calibration_X** and **calibration_y** are provided, the solution will instead rescale via **dy_dX * std(calibration_y) / std(calibration_X@dy_dX)**. This is a niche application but may be useful to apply a regression trained on one timescale to a different timescale.

fit_intercept : *bool, default=True*

Literally makes no difference because I have not included a way to return the y-intercept. The structure is in place that I can eventually include it if needed, but for now this is a placeholder.

EN_selection : *default='random'*

Option passed to sklearn.linear_model.ElasticNet. See documentation for details and options list.

ridge_solver : *default='svd'*

Option passed to sklearn.linear_model.Ridge. See documentation for details and options list.

l1_ratio : *float, default=0.5*

Ratio of LASSO regularization relative to ridge in sklearn.linear_model.ElasticNet. l1_ratio = a / (a + b) in the objective function ||y - X@dy_dX||^2_2 + a * ||dy_dX||_1 + 0.5 * b * ||dy_dX||^2_2. Only relevant for 'EN' method.

alpha : *float, default=1*

Regularization strength relevant for methods 'RIDGE', 'EN', 'EN_RIDGE', and 'LASSO'. **alpha** has a slightly different meaning for sklearn.linear_model.Ridge than for the implementation in sklearn.linear_model.ElasticNet.

n_PLS_components : *int, default=5*

Number of partial least squares components to keep, relevant for 'PLS' only. See sklearn.cross_decomposition.PLSRegression for more detail.

return_dynorm_dxnorm* : *bool, default=False

When 'True', and only if **standardize** is also 'True', returns the solution coefficient vector without returning the solution to original units. I.e., the coefficients will be in units sigma/sigma, relating a standard deviation in one variable to a standard deviation in the other. In this case only, the function returns **dy_dX, dynorm_dXnorm**.

random_seed : ***int, default=42***

Random seed applied wherever an option for randomness exists for reproducibility.

Returns:

dy_dX: ***ndarray of shape (n_features)***

The solution vector of coefficients relating each predictor variable to the predictand via **y=X@dy_dX**. Solution will be in original data units and already account for weighting if applicable, so will apply directly to nominal predictor variables.

dynorm_dXnorm: ***ndarray of shape (n_features)***

Returned as a second array only if **return_dynorm_dxnorm** is set to 'True'. Incorporates any weighting but leaves solution in units of standard deviation. Thus, this vector relates a standard deviation in **y** to a standard deviation in each feature of **X**.

Usage:

```
X = np.random.random((100,10))
y = np.random.random(100)

dy_dX                   = MLR_set(X,y,method='PLS')
dy_dX, dynorm_dXnorm    = MLR_set(X,y,method='PLS',return_dynorm_dxnorm=True)
```