MLR_CV(***X***, ***y***, ***cross_validation***='resample_split', ***folds***=5, ***n_resamples***=10, ***resample_train_fraction***=0.8, ***bounds***=None, ***x0***=None, ***loss_func***=None, ***tol***=None, ***solver***='Nelder-Mead', ***return_xVals***=False, ***max_PLS_components***=25, ***plot_PLS***=False, ***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 cross-validation to set the hyperparameters required by the methods supported by **MLR_set**. Makes the basic assumption that the system can be described linearly as **y=X@dy_dX + ε**. Returns the optimized solution(s) and the hyperparameter(s) used to produce those solution(s).

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.**

cross_validation : ***{'resample_split', 'k-fold'}, default='resample_split'***

The supported cross-validation approaches are:

- 'resample-split' : This approach randomly divides **X** and **y** into a training subset and a validation subset. The fraction of the data included in the training subset is set by **resample_train_fraction**. Hyperparameter(s) are set by training on the training subsample and minimizing the loss (as defined by **loss_func**) against the validation subsample. If multiple iterations of the resampling procedure are required via **n_resamples**, the training and validation subsets will be selected with a different random seed, meaning overlap in validation subsets across iterations is expected. The optimal solution **dy_dX** is the mean across all n=**n_resamples** solutions.

- ‘k-fold’ : This approach divides the **X** and **y** data randomly and evenly into ‘k’ groups or ‘folds’, with ‘k’ set by the parameter **folds**. For example, this animation. The process is repeated ‘k’ times, with a different group held out as the validation subset in each iteration. In each iteration, hyperparameter(s) are set by minimizing the loss (as defined by **loss_func**) in the held-out validation group, training on the remaining data. Unlike ‘resample-split’, there will be no overlap in validation subsets across iterations. The optimal solution **dy_dX** is the mean across all k=**folds** solutions.

folds : ***int, default=5***

Number of groups that the data are divided among in the ‘k-fold’ cross-validation. This sets the number of cross-validation iterations, each with a different solution coefficient vector. The optimal solution **dy_dX** is taken as the mean across all iterations. Optimal hyperparameter(s) are saved and returned for each iteration.

n_resamples : ***int, default=10***

Number of times the data are randomly shuffled between training and validation subsets in the ‘resample-split’ cross-validation. These resamples are done with replacement, meaning each division has no knowledge of how the data were divided in previous iterations. This sets the number of cross-validation iterations, each with a different solution coefficient vector. The optimal solution **dy_dX** is taken as the mean across all iterations. Optimal hyperparameter(s) are saved and returned for each iteration.

resample_train_fraction : ***float, default=0.8***

Fraction of the data (0-1) that will be used as the training subset in the ‘resample-split’ cross-validation. The remaining fraction (1-**resample_train_fraction**) are reserved as a validation subset.

bounds : ***List[Tuple[float, float]], default=None***

Bounds for the hyperparameter(s). Should be in the form of a list with a tuple for each hyperparameter with low and high bounds for that parameter. If left ‘None’, the following defaults are applied by method:

- ‘RIDGE’ : [(1e-7, 1e6)] *for* ***[alpha]***

- ‘EN_RIDGE’ : [(1e-7,1e3)] *for **[alpha]***
- ‘LASSO’ : [(1e-9,1e2)] *for **[alpha]***
- ‘EN’ : [(1e-9,1e3), (1e-9, 0.99)] *for **[alpha, l1_ratio]***

x0 : ***List[float,...], default=None***

Initial guesses for hyperparameter(s). Should be in the form of a list of floats. If left ‘None’, the following defaults are applied by method:

- ‘RIDGE’ : [10] *for **[alpha]***
- ‘EN_RIDGE’ : [10] *for **[alpha]***
- ‘LASSO’ : [0.1] *for **[alpha]***
- ‘EN’ : [0.1, 0.5] *for **[alpha, l1_ratio]***

loss_func : function***(dy_dX: np.ndarray (n_features), X: np.ndarray (n_samples,n_features), y: np.ndarray (n_samples)) -> float, default=None***

Loss function to evaluate performance of a candidate solution for **dy_dX**. If ‘None’, the default is RMSE. Functions should take three arguments, the candidate solution, the predictor testing data, and the predictand data against which the loss will be judged: (**dy_dX, X_test, y_test**). The function must create some objective measure to evaluate how ‘well’ the prediction **X_test@dy_dX** matches the truth **y_test**, and return it as a float. Lower numbers should indicate better performance.

tol : ***default=None***

Option passed to scipy.optimize.minimize, which is used to select the hyperparameter that minimizes **loss_func**. See documentation for details and options list.

solver : *default='Nelder-Mead'*

Option passed to scipy.optimize.minimize, which is used to select the hyperparameter that minimizes **loss_func**. See documentation for details and options list.

return_xVals : *bool, default=False*

When 'True', returns the optimal solution found for each iteration set by **folds** or **n_resamples** rather than just the mean across all iterations.

max_PLS_components : *int, default=25*

Maximum number of components (**n_PLS_components**) to test when optimizing the 'PLS' method. PLS optimization is performed by calculating the loss for each value of **n_PLS_components** from 1 up to **max_PLS_components**, then taking the minimum loss. The discrete nature of **n_PLS_components** makes this possible and much better behaved than the other optimizations.

plot_PLS : *bool, default=False*

When 'True', plots the loss for each value of **n_PLS_components** up to **max_PLS_components**. This helps to visualize whether **max_PLS_components** has been set high enough.

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.

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 single optimal solution vector of coefficients relating each predictor variable to the predictand via **y=X@dy_dX**. This value is achieved by taking the average across all iterations if multiple solutions are set by **folds** or **n_resamples**.

xVals: ***ndarray of shape (n_iterations, n_features)***

The optimal solution vector of coefficients for each iteration if multiple solutions are set by **folds** or **n_resamples**. Returned instead of **dy_dX** if **return_xVals** is 'True'.

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

Returned as a second array only if **return_dynorm_dxnorm** is set to 'True'. The single optimal solution vector in units of standard deviation. This value is achieved by taking the average **dynorm_dXnorm** across all iterations if multiple solutions are set by **folds** or **n_resamples**.

xVals_norm: ***ndarray of shape (n_iterations, n_features)***

The optimal solution vector in units of standard deviation for each iteration if multiple solutions are set by **folds** or **n_resamples**. Returned instead of **dynorm_dXnorm** if **return_xVals** is 'True'.

hyper_params: ***List[ArrayLike] of shape (n_iterations)***

The hyperparameter(s) for each iteration.

Usage: .

```
X = np.random.random((500,1000))
y = np.random.random(500)

dy_dX, hyper_params                 = MLR_CV(X,y,method='PLS')

dy_dX, dynorm_dXnorm, hyper_params  =
MLR_CV(X,y,method='PLS',return_dynorm_dxnorm=True)

xVals, hyper_params                 = MLR_CV(X,y,method='PLS')

xVals, xVals_norm, hyper_params     =
MLR_CV(X,y,method='PLS',return_dynorm_dxnorm=True, return_xVals=True)
```