\_\_Init\_\_explain

# def fn\_gmm\_mixtools(vals):

The function first adds a small amount of random noise to the input data vals, using np.random.uniform (0, 0.0001, size=len(vals)). This is done to avoid potential numerical issues when fitting the Gaussian mixture model, which can occur when the data contains repeated values or when the covariance matrix of the components is close to singular.

After adding the noise, the function fits a Gaussian mixture model with three components using the GaussianMixture class from scikit-learn. The n\_components parameter specifies the number of components in the model, and the covariance\_type parameter specifies the type of covariance matrix used by each component (in this case, a full covariance matrix). The max\_iter parameter specifies the maximum number of iterations to perform during the EM algorithm.

***The EM algorithm, short for Expectation-Maximization algorithm, is an iterative optimization algorithm used to estimate the parameters of statistical models that involve latent variables. It is commonly used to fit mixture models, including Gaussian mixture models.***

***In essence, the EM algorithm is a two-step process that iteratively updates estimates of the model parameters until convergence. The two steps are:***

* ***Expectation (E) step: Compute the expected value of the latent variables (i.e., the responsibilities of each component) given the current parameter estimates.***
* ***Maximization (M) step: Update the parameter estimates to maximize the likelihood of the observed data, using the expected values of the latent variables computed in the E step.***

***The EM algorithm is guaranteed to converge to a local maximum of the likelihood function, and often works well in practice for fitting mixture models. However, the algorithm can be sensitive to the initial parameter values and may get stuck in local maxima if the number of components in the model is not well-chosen.***

***Overall, the EM algorithm is a powerful tool for estimating the parameters of complex statistical models, and is widely used in machine learning and statistics.***

Finally, the function returns a dictionary containing the estimated parameters of the Gaussian mixture model: the means of each component (mu), the standard deviations of each component (sigma), and the mixing coefficients of each component (lambda).

# def fn\_gmm\_mixtools\_v2(vals):

1. We import necessary libraries - NumPy, gaussian\_kde from SciPy and GaussianMixture from Scikit-learn.
2. The function takes in a vector of values vals as input.
3. We add random noise to the input data using np.random.uniform(0, 0.0001, size=len(vals)), similar to the R implementation.
4. We use gaussian\_kde to estimate the probability density function (PDF) of the data. The resulting PDF is stored in kde.
5. We create an array of x-values using np.linspace, then evaluate the PDF at those x-values to get the corresponding y-values.
6. We stack the x and y values into a single 2D array df, then use np.diff to find where the second derivative of the y-values changes sign. This indicates the location of any peaks in the PDF.
7. We select only the rows of df corresponding to the peaks, which we define as any y-value greater than 20% of the maximum y-value.
8. We initialize a GaussianMixture model with three components, using the mean of the peak x-values as the initial guess for the means of each component (means\_init).
9. We fit the model to the corrected data using gmm.fit.
10. We return a dictionary containing the means, standard deviations (sigma), and weights (lambda) of the fitted components.
11. If an error occurs during the function execution, None is returned.

# def fn\_gmm\_mclust(vals):

The function takes a single input vals, which is the array of values to be fitted to a Gaussian mixture model.

The input values are first corrected by adding a small amount of random noise using np.random.uniform.

The GaussianMixture function from the sklearn.mixture module is used to fit the input data to a Gaussian mixture model with a single component (n\_components=1).

The estimated mean and standard deviation of the fitted Gaussian distribution are extracted using gmm\_fit.means\_ and np.sqrt(gmm\_fit.covariances\_), respectively.

If the number of estimated means is 1, then lambda\_est is set to 1. Otherwise, the estimated mixture weights are extracted using gmm\_fit.weights\_.

The function returns a dictionary containing the estimated mean, standard deviation, and mixture weights as 'mu', 'sigma', and 'lambda', respectively.

# def fn\_gmm\_kclust(vals):

1. The input data vals is corrected by adding a small amount of uniform noise to avoid singularities.
2. The number of clusters is determined using the KMeans algorithm and the silhouette\_score metric. This is done by fitting KMeans models with 2, 3, 4, and 5 clusters, and choosing the number of clusters that maximizes the silhouette score.
3. If the KMeans fitting fails (i.e., an exception is raised), then only one cluster is assumed.
4. If only one cluster is assumed, then the mean and standard deviation of the input data are computed directly.
5. If more than one cluster is assumed, then a Gaussian mixture model is fit to the input data using the GaussianMixture algorithm.
6. For each cluster, the lambda (proportion of data points in the cluster), mu (mean of the data points in the cluster), and sigma (standard deviation of the data points in the cluster) are computed.
7. The lambda, mu, and sigma values for each cluster are returned as lists.

Note: The fn\_gmm\_kclust function is different from the other GMM functions in that it uses the KMeans algorithm to determine the number of clusters, instead of using the EM algorithm.

# def fn\_utility(utility\_func, \*args):

This function takes an arbitrary utility function and its arguments, and returns the result of calling the function with the given arguments.

The \*args parameter in Python allows us to pass a variable number of arguments to a function. The utility\_func parameter is the function we want to call, and \*args is a tuple of arguments that will be passed to utility\_func.

The fn\_utility\_ei function calculates the expected improvement (EI) for a set of traits and an optimal response. The EI is a commonly used acquisition function in Bayesian optimization, which is a sequential process for optimizing black-box functions.

The function takes the following arguments:

* traits: a list with three vectors (mu, sigma, and lambda) that describe the Gaussian mixture model (GMM) of the objective function
* optimal\_response: the best response observed so far
* complement\_u: a boolean indicating whether to calculate other acquisition functions in addition to EI
* estimated\_sigma: an optional vector of estimated standard deviations for the GMM components

The function first retrieves the mu and sigma vectors from the traits list, which represent the mean and standard deviation of each GMM component, respectively. If an estimated\_sigma vector is provided, it is used instead of the standard deviation from the data.

Next, the function calculates the gamma vector, which is the difference between the optimal response and the mean of each GMM component, divided by the standard deviation of each component. This is a normalized measure of how far each GMM component is from the optimal response.

Using gamma, the function then calculates the expected improvement for each GMM component using the formula u\_val\_mat <- s \* (gamma \* pnorm(gamma) + dnorm(gamma) ), where u\_val\_mat is a matrix with a column for EI values. The pnorm and dnorm functions are used to calculate the probability density and distribution functions of the standard normal distribution, respectively.

If complement\_u is set to TRUE, the function also calculates several other acquisition functions using other functions that are defined later in the code. These functions include lower confidence bound (LCB) and various versions of the GMM aggregation acquisition function, which is used to combine the predictions of multiple GMM models.

Finally, the function returns a matrix with the EI values for each GMM component and the other acquisition functions if complement\_u is set to TRUE.

# def fn\_utility\_ei(traits, optimal\_response, complement\_u=False, estimated\_sigma=None):

The function fn\_utility\_ei computes the expected improvement for a set of traits and an optimal response.

Inputs:

1. traits
2. optimal response
3. complement\_u :(default to False) a flag indicating whether to compute complement utilities
4. estimated\_sigma: an optional input that specifies the estimated value of sigma (if any other than empirical estimation).

The function first extracts the mean (m) and standard deviation (s) of the traits. If estimated\_sigma is not None, it uses this value instead of the standard deviation s.

Next, the function computes the value of gamma as (optimal\_response - m) / s. The expected improvement is then computed as s \* (gamma \* norm.cdf(gamma) + norm.pdf(gamma)), where norm.cdf and norm.pdf are the cumulative distribution function and probability density function of the standard normal distribution, respectively.

If complement\_u is True, the function calls fn\_utility\_global\_gmix to get a list of global model mixtures (list\_gMix) and then computes the complement utilities using several functions (fn\_utility\_lcb, fn\_utility\_gmix\_agg\_ei, fn\_utility\_gmix\_wmax\_ei, etc.). The complement utilities are stacked in a value matrix.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Random\_Forest\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Infjack\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

def rInfJack(pred, inbag, calibrate=True, used\_trees=None):

The infinitesimal jackknife (IJ) is a variance estimation technique for random forests that computes the variance of the predictions at the individual level. The IJ is based on the idea of leaving one observation out at a time and computing the variance of the resulting predictions across all trees. The resulting estimates are then averaged to obtain an estimate of the overall variance. The IJ can be computationally expensive, particularly for large datasets and random forests with many trees, but it has been shown to provide accurate and reliable estimates of the prediction error for random forests. The IJ can also be combined with empirical Bayes calibration to further improve the accuracy of the variance estimates.

Here is an explanation of the rInfJack function with bullet points:

Inputs:

* pred: an n x B matrix of tree-wise predictions for n observations and B trees.
* inbag: an n x B matrix indicating which observations were included in each bootstrap sample.
* calibrate: a boolean indicating whether to calibrate the variance estimates using empirical Bayes. Default is True.
* used\_trees: a list of indices indicating which trees to use in the variance estimation. Default is all trees.
* Outputs:

An n x 2 array of estimated mean and variance for each observation.

Steps:

1. Select the subset of tree-wise predictions based on the used\_trees list.
2. Check if the bootstrap sampling was done with or without replacement.
3. Compute the raw infinitesimal jackknife variance estimator using two different formulas depending on the size of B.
4. Apply Monte Carlo bias correction to the raw estimator.
5. Apply finite sample correction if the bootstrap sampling was done without replacement.
6. If calibrate is True and the sample size is greater than 20, compute variance estimates using half of the trees, estimate the scale of Monte Carlo noise, and use it for empirical Bayes calibration of the variance estimates. If calibration fails, fall back to non-calibrated variance estimates.
7. Return an n x 2 array of estimated mean and variance for each observation.

# def gfit

This function estimates a probability density function (PDF) from a set of observations X using a non-parametric method. The algorithm uses a generalized linear model (GLM) framework to fit a polynomial function to the data, and adds a Gaussian kernel density estimator to smooth the result.

The gfit function takes the following input parameters:

* X: an array of observations
* sigma: a parameter that controls the width of the Gaussian kernel
* p: the order of the polynomial function to be fit to the data
* nbin: the number of bins used to discretize the domain

unif\_fraction: the fraction of the uniform density to be added to the estimated PDF

The function first generates a set of equally spaced bins (xvals) covering the range of the data, using the linspace function from the numpy library. The bin width is then computed as the distance between two consecutive bins.

The noise kernel is defined as a Gaussian distribution centered at zero with a standard deviation of sigma, and its values are normalized by the bin width and the standard deviation to ensure that the area under the curve is one. The roll function from numpy is then used to shift the kernel so that its peak aligns with the zero point of the domain.

The design matrix XX is generated by raising xvals to the powers from 1 to p and multiplying the result by an indicator function that is 1 for positive values and 0 otherwise. This creates a set of predictors that are combined linearly to fit the data.

The negative log-likelihood function neg\_loglik takes the parameter vector eta as input and computes the value of the objective function that needs to be minimized to obtain the maximum likelihood estimate of eta. The function first computes the predicted density g\_eta\_raw by exponentiating the product of XX and eta and multiplying it by the indicator function for positive values. The function then checks if the sum of g\_eta\_raw is infinite or too small to avoid numerical instability, and returns a large value if this is the case. The predicted density is then normalized by its sum to obtain the main estimate of the PDF. The uniform density is added to this estimate and the resulting density is convolved with the rotated noise kernel using the convolve function from scipy with the mode='same' parameter to ensure that the output has the same size as the input. Finally, the negative log-likelihood is computed as the sum of the logarithms of the estimated density values at the observation points, interpolated using the interp function from numpy.

The function uses the minimize function from scipy.optimize to find the optimal value of eta that minimizes the negative log-likelihood. The algorithm used is the Nelder-Mead simplex method, which does not require the computation of derivatives. The estimated PDF is then computed using the same steps as before, using the estimated value of eta.

The output of the function is a dictionary with two keys: x and g, containing the values of the bin boundaries and the estimated PDF, respectively.

# def gbayes(x0, gest, sigma):

The function takes three inputs: x0 is the observed value, gest is a dictionary containing two keys x and g, which are arrays representing a prior density (as returned by the function gfit), and sigma is the noise estimate.The function first calculates the kernel density estimate Kx of the prior density evaluated at the points (gest['x'] - x0) / sigma using the Gaussian kernel with standard deviation sigma. This is done using the np.exp() function to exponentiate the kernel function and the np.sqrt() function to take the square root of the denominator.

The function then calculates the posterior density by multiplying the prior density g.est$g with the kernel density estimate Kx. This is stored in the variable post.

The posterior density is then normalized to obtain a valid probability density function by dividing it by its sum, which is done by dividing post by np.sum(post). Finally, the function computes the posterior mean by taking the dot product of the normalized posterior density with the array gest['x'], which represents the grid points at which the prior density is evaluated.

The function returns the posterior mean E[mu | x0], which is the expected value of the true underlying mean parameter given the observed value x0 and the prior information.

# def calibrateEB(vars, sigma2):

The function takes two inputs: vars is a one-dimensional array of p-values to be calibrated, and sigma2 is the variance parameter for the normal prior distribution used in the empirical Bayes procedure.

The function first checks if the variance parameter is positive and if the minimum and maximum values of the input p-values are the same. If either of these conditions is true, the function returns the input p-values clipped at zero, which are returned by np.maximum(vars, 0).

If the variance parameter is positive and the input p-values have different minimum and maximum values, the function calculates the empirical Bayes prior density using the gfit() function, which returns a dictionary with keys 'x' and 'g' representing the grid points and the estimated prior density, respectively.

If there are many test points (i.e., len(vars) >= 200), the function uses interpolation to speed up the computations. It first generates a set of equidistant quantiles of the input p-values between 0 and 1, inclusive, using the np.quantile() function. It then evaluates the empirical Bayes posterior mean at these quantiles using the gbayes() function, and interpolates the resulting curve using the interp1d() function from the SciPy library with cubic interpolation and extrapolation to handle values outside the quantiles. Finally, the calibrated p-values are obtained by evaluating the interpolated curve at the input p-values using the f() function.

If there are few test points, the function evaluates the empirical Bayes posterior mean using the gbayes() function at each input p-value using a list comprehension, which is stored in calib\_all.

The function returns the calibrated p-values, which are either obtained by interpolation or by direct evaluation of the empirical Bayes posterior mean.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Random\_Forest\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Infjack\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# def fn\_utility\_ei\_ij(traits, optimal\_response, complement\_u=False):

This Python function fn\_utility\_ei\_ij computes the expected improvement (EI) utility values for a set of candidate points, given the observed data and an optimal response value. It takes the following inputs:

* traits: a dictionary containing the mean and standard deviation of the candidate points, as well as any custom data needed for the computation.
* optimal\_response: the optimal response value that is used as a reference to compute the utility values.
* complement\_u: a boolean indicating whether to compute the utility values for complement criteria in addition to the standard EI utility.

The function first computes the Infinitesimal Jackknife (IJ) estimator for the variance of the observations, using the rInfJack function. If sigma\_IJ is not None, it is used as the standard deviation value for the computation, otherwise the empirical standard deviation is used. The function then computes the standard EI utility value for each candidate point using the formula:

u = s \* (gamma \* norm.cdf(gamma) + norm.pdf(gamma))

where s is the standard deviation value, gamma is the ratio of the difference between the optimal response and the candidate point mean to the standard deviation. The function then adds a column of zeros to the utility matrix to accommodate complement criteria, if specified.

If complement\_u is True, the function also computes the utility values for several complement criteria by calling other utility functions, including lower confidence bound (LCB), and various aggregation, weighted max, and max methods of Gaussian mixture models. The utility values for complement criteria are computed and added as additional columns to the utility matrix.

The function returns a matrix containing the EI utility values, with each row representing a candidate point and each column representing a different utility criterion.

# def fn\_utility\_lcb(traits: Dict[str, np.ndarray], optimal\_response: np.ndarray, kappa: float = 2.0, complement\_u: bool = False, estimated\_sigma: Optional[np.ndarray] = None) -> np.ndarray:

The function fn\_utility\_lcb computes the lower confidence bound (LCB) utility for a given set of traits and an optimal response. The LCB is defined as the minimum expected utility that a decision-maker can obtain with a given level of confidence (in this case, the confidence level is determined by the parameter kappa).

The function takes the following arguments:

* traits: A dictionary with two keys: 'mu' and 'sigma'. The value of the 'mu' key is an array of mean values for each trait, and the value of the 'sigma' key is an array of standard deviation values for each trait.
* optimal\_response: An array of optimal response values for each trait.
* kappa: A float value that determines the level of confidence for the LCB utility. The default value is 2.0.
* complement\_u: A boolean flag that determines whether to compute the complement of the LCB utility (i.e., the maximum expected utility minus the LCB utility). The default value is False.
* estimated\_sigma: An optional array of estimated standard deviation values for each trait. If this argument is provided, the standard deviation values in the `traits

# def fn\_utility\_global\_gmix(traits, optimal\_response, byrow=True):

Inputs:

1. Traits
2. Optimal response
3. Byrow(by row )

This function first determines the margin along which to apply the function (apply\_margin) based on whether the byrow parameter is set to True (margin 1) or False (margin 0). It then applies a function called gmix\_fn to each row or column of traits['values'], depending on the value of byrow.

The gmix\_fn function first fits a Gaussian mixture model to the input data using scikit-learn's GaussianMixture class. It then returns a dictionary containing the mean (mu), standard deviation (sigma), and mixing coefficient (lambda) of the fitted mixture model.

The np.apply\_along\_axis function is then used to apply the gmix\_fn function to each row or column of traits['values'], and the resulting dictionary for each row or column is returned as an array.

# def fn\_utility\_gmix\_agg\_ei(traits, optimal\_response, byrow=True, complement\_u=False, list\_MM=None):

The fn\_utility\_gmix\_agg\_ei function calculates the expected improvement over a mixture of Gaussian models, which are specified in the list\_gMix input. If list\_gMix is not provided, the function calls fn\_utility\_global\_gmix to generate the list of Gaussian mixture models.

The function first calculates the integral part of the expected improvement calculation for each Gaussian mixture model in list\_gMix using the fn\_utility\_ei function, which is then weighted by the lambda values of the model to obtain a single expected improvement value for each model.

If the complement\_u flag is True, the function also calculates the complement of the expected improvement for several other utility functions, concatenates the results, and returns the combined utilities.

# def fn\_utility\_gmix\_wmax\_ei(traits, optimal\_response, byrow=True, complement\_u=False, list\_MM=None):

The code starts by assigning the list\_MM argument to list\_gMix, or calls fn\_utility\_global\_gmix if list\_gMix is None.

The integral part of the utility calculation is performed using a list comprehension that applies fn\_utility\_ei on each element of list\_gMix, multiplies the result by the corresponding lambda, and stacks the resulting vectors vertically using np.vstack.

Then, the row index of the maximum value in utilities\_gMix is found using np.argmax, and utilities\_gMix is updated to contain only the row with the maximum value, multiplied by its lambda, using np.multiply.

The resulting 1D array is then reshaped to a 2D array with one column, and assigned the column name "gmix\_wmax\_ei".

Finally, if complement\_u is True, a list of complement utility functions and their parameters are defined

# Summary of fn\_utilities

1. **fn\_utility\_ei**: This function computes the Expected Improvement (EI) acquisition function, which is commonly used in Bayesian optimization to guide the search for the maximum of an unknown function. EI is defined as the expected improvement over the current best value of the function, based on a probabilistic surrogate model that approximates the true function.
2. **fn\_utility\_lcb**: This function computes the Lower Confidence Bound (LCB) acquisition function, which is another commonly used acquisition function in Bayesian optimization. LCB trades off exploration and exploitation by balancing the expected improvement with the uncertainty of the surrogate model.
3. **fn\_utility\_gmix\_agg\_ei**: This function computes the Expected Improvement acquisition function over a mixture of Gaussian distributions, where the mixture weights and parameters are learned from the data. This allows the surrogate model to capture complex, multi-modal functions.
4. **fn\_utility\_gmix\_wmax\_ei**: This function computes the Expected Improvement acquisition function over a mixture of Gaussian distributions, but instead of taking the expectation over the entire mixture, it selects the component with the highest weight and computes the EI for that component only. This allows the search to focus on the most likely mode of the function.
5. **fn\_utility\_gmix\_max\_ei**: This function is similar to fn\_utility\_gmix\_wmax\_ei, but instead of selecting the component with the highest weight, it selects the component with the highest expected improvement.
6. **fn\_utility\_gmix\_wmax\_lcb**: This function computes the Lower Confidence Bound acquisition function over a mixture of Gaussian distributions, selecting the component with the highest weight only.
7. **fn\_utility\_gmix\_max\_lcb:** This function is similar to fn\_utility\_gmix\_wmax\_lcb, but selects the component with the highest LCB.

The main differences between these functions lie in the way they model the surrogate function and the way they balance exploration and exploitation. fn\_utility\_ei and fn\_utility\_lcb use a single Gaussian distribution to model the surrogate function, while the fn\_utility\_gmix\_\* functions use a mixture of Gaussians. fn\_utility\_ei and fn\_utility\_gmix\_agg\_ei focus on maximizing the expected improvement, while fn\_utility\_lcb and fn\_utility\_gmix\_\*\_lcb balance exploration and exploitation by taking the uncertainty of the surrogate function into account. Finally, the fn\_utility\_gmix\_wmax\_\* and fn\_utility\_gmix\_max\_\* functions use the mixture of Gaussians to focus on the most promising regions of the search space.

# def fn\_eval\_utility(ds, surrogate\_predict\_fn, surrogate\_model, surrogate\_utility\_fn, optimal\_value, col\_names, direction=1, is\_normalization\_required=False, param\_def=None, eval\_complement=False):

Inputs:

* Ds: a dataset to evaluate
* surrogate\_model
* surrogate\_predict\_fn: the surrogate model's prediction on, along with the surrogate model itself surrogate\_utility\_fn: function to evaluate the utility of the predictions.
* optimal\_value: is the optimal response value (e.g., maximum or minimum) that the surrogate model is trying to predict, and col\_names are the names of the columns in ds. direction is a parameter that determines whether the utility should be maximized (1) or minimized (-1).
* is\_normalization\_required: If it is True, the ds dataset will be normalized before being passed to the surrogate model.
* param\_def : a dictionary of the parameter definitions, which is used to normalize the data.
* eval\_complement: If is True, the function will evaluate the complement of the utility function.

The function returns the estimated utility value for the given dataset.

# def fn\_sampling\_lhs(vals=None, param\_space=None, sample\_size=None):

This function generates Latin Hypercube samples (LHS) from a parameter space. The function takes several arguments:

Inputs:

* vals: unused argument.
* param\_space: dictionary containing the definition of the parameter space to sample from. It includes:
* definition: a pandas DataFrame with columns 'name', 'type', 'min', and 'max', where 'name' corresponds to the name of the parameter, 'type' is the type of the parameter ('continuous' or 'discrete'), 'min' and 'max' are the minimum and maximum values the parameter can take, respectively.
* sample\_size: integer representing the number of samples to generate.

The output of the function is a dictionary where the keys are the parameter names and the values are arrays with the sampled values for each parameter. The LHS samples are generated using the lhs() function from the pyDOE package, which returns a normalized LHS matrix. The function then scales the normalized samples to the actual parameter range and returns a dictionary with the scaled samples.

The function returns the generated LHS samples as a matrix. Each row of the matrix corresponds to a sample, and the columns correspond to the dimensions of the parameter space.

# def fn\_sampling\_normal\_optimal(vals, param\_space, sample\_size, \*\*kwargs):

This function generates a sample of parameter values based on a set of previously evaluated parameter sets. It uses normal distribution to sample parameter values around the current optimal parameter set. The function takes the following inputs:

* vals: a matrix of previously evaluated parameter sets
* param\_space: a list containing the definition of the parameter space
* sample\_size: the number of parameter sets to be generated
* ...: additional arguments (not used in the function)

The function first extracts the names of the parameters and the number of parameters to be sampled from the parameter space definition. It then calculates the sampling intensity and deviation based on the mutation rate and mutation standard deviation specified in the settings. If these values are not present in the settings, default values of 0.5 and 1 are used, respectively.

The function then checks if the number of optimization cores is greater than 1. If so, it uses the mclapply function to generate the parameter sets in parallel. Otherwise, it uses the apply function to generate the parameter sets sequentially.

For each parameter, the function selects the current optimal value from the previously evaluated parameter sets and samples new values based on a normal distribution centered around the optimal value with a standard deviation equal to the sampling deviation. The number of new values to be sampled is determined by the sampling intensity, which is the proportion of the sample size to be replaced by new values. The function then returns the generated parameter sets as a matrix with column names corresponding to the parameter names.

# def fn\_optimisation\_uniform\_prior(vals, sampling\_fn, surrogate\_predict\_fn, surrogate\_model, surrogate\_utility\_fn, iteration\_space, param\_range, param\_space, omit, target, sample\_size=1, variation\_rate=0.3, opt\_parallel=True, opt\_cores=cpu\_count()-2, opt\_store\_intermediate=True, is\_normalization\_required=False, eval\_complement=False, \*\*kwargs):

This is a function for performing optimization with a uniform prior distribution.

Inputs:

* Vals: the current set of parameter values
* sampling\_fn :a sampling function,
* surrogate\_predict\_fn :a function for predicting the surrogate model,
* surrogate\_model: the surrogate model itself,
* surrogate\_utility\_fn: a utility function for the surrogate model
* param\_range: the range of parameter values,
* param\_space : the parameter space,
* omit parameter
* target parameter
* sample\_size : number of samples
* a variation.rate parameter
* opt\_parallel :a boolean flag for parallelization
* opt\_cores : the number of cores to use for parallelization
* opt\_store\_intermediate : a flag for storing intermediate results
* is\_normalization\_required : a flag for whether normalization is required
* eval\_complement : a flag for evaluating the complement of the acquisition function
* ...: additional parameters to pass to the sampling\_fn

The function first generates a prior sample of parameter values using the sampling\_fn, and then evaluates the utility of each parameter set using the surrogate model and utility function. If opt\_parallel is set to TRUE, the evaluation is parallelized across multiple cores using the mclapply function. The function then returns the optimal set of parameters based on the highest utility value, as well as the prior sample and its corresponding utilities. If opt\_store\_intermediate is set to TRUE, the function also stores the prior sample and its corresponding utilities. If eval\_complement is set to TRUE, the function evaluates the complement of the acquisition function and stores the corresponding utility values.

Outputs:

* output\_obj: a list containing the optimal parameter set and its corresponding utility
* output\_obj$space: a matrix containing the optimal parameter set
* output\_obj$utility: the utility of the optimal parameter set
* output\_obj$pop: a data frame containing all parameter sets and their corresponding utilities generated during the optimization process (if opt\_store\_intermediate is set to TRUE)
* output\_obj$pop\_utility: a vector containing the utilities of all parameter sets generated during the optimization process (if opt\_store\_intermediate is set to TRUE)
* output\_obj$complementary\_utility\_fn: a vector containing the names of the complementary acquisition functions (if eval\_complement is set to TRUE)

# def fn\_optimisation\_quasi\_newthon (vals, sampling\_fn, surrogate\_predict\_fn,surrogate\_model, surrogate\_utility\_fn, iteration\_space, param\_range, param\_space, sample\_size=1, opt\_parallel=True, opt\_cores=cpu\_count()-2, opt\_best\_init=False, opt\_init\_pop\_shufle\_rate=0.1, opt\_store\_intermediate=False,is\_normalization\_required=False, \*\*kwargs):

The function fn\_optimisation\_quasi\_newthon performs optimization using quasi-Newton methods.

* vals: A vector of numeric values representing the target variable for the optimization problem.
* sampling\_fn: A function that generates samples from the parameter space for optimization. This function is called to generate an initial set of points to evaluate the utility function.
* surrogate\_predict\_fn: A function that predicts the target variable using the surrogate model.
* surrogate\_model: The surrogate model used to approximate the target function. The model is updated as new data is collected during optimization.
* surrogate\_utility\_fn: A function that calculates the utility of a candidate set of parameters. The utility function is used to guide the search for optimal parameters.
* iteration\_space: A named list that contains the target value (value) and the tolerance (tol) for the optimization problem.
* param\_range: A named list that contains the upper and lower bounds for the parameters being optimized.
* param\_space: A list that contains the definition of the parameter space being optimized. It includes the parameter names, initial values, and bounds for each parameter.
* sample\_size: The number of samples to generate from the parameter space for optimization.
* opt\_parallel: A logical value indicating whether to use parallel processing during optimization.
* opt\_cores: The number of processor cores to use for parallel processing.
* opt\_best\_init: A logical value indicating whether to use the best initial value found during optimization.
* opt\_init\_pop\_shufle\_rate: A numeric value between 0 and 1 indicating the rate of shuffling of the initial population for optimization.
* opt\_store\_intermediate: A logical value indicating whether to store intermediate results during optimization.
* is\_normalization\_required: A logical value indicating whether normalization is required for the parameters being optimized.
* ...: Additional arguments that can be passed to the sampling\_fn.

The function first reduces the parameter space to only the enabled dimensions, and then generates an initial set of points to evaluate the surrogate model. It then optimizes the surrogate model using quasi-Newton methods for each initial point and returns the set of parameters that maximizes the utility function. If opt\_store\_intermediate is set to TRUE, the function also returns the intermediate populations evaluated during the optimization process. The function returns a list object containing the optimized parameters and the corresponding utility value.

"Singleton optimization" refers to the optimization of a single point or instance. In the context of the fn\_optimisation\_quasi\_newthon function, it is the optimization of a single set of parameters (a "singleton") using the quasi-Newton method with limited memory (L-BFGS-B). The function generates an initial grid of parameter values using a sampling function, then performs optimization on each of these initial points using the L-BFGS-B method. The function returns the optimal parameter set and corresponding utility value.

# def fn\_optimisation\_differential\_evolution(vals, surrogate\_predict\_fn, surrogate\_model, surrogate\_utility\_fn, iteration\_space, param\_range, param\_space, sample\_size = 1, opt\_parallel = True, opt\_cores = multiprocessing.cpu\_count()-2, opt\_best\_init = False, opt\_init\_pop\_shufle\_rate = 0.1, opt\_store\_intermediate = False, is\_normalization\_required = False, \*\*kwargs):

The fn\_optimisation\_differential\_evolution function takes several arguments including

* vals :the initial set of parameter values
* surrogate\_predict\_fn :a function to predict the output of the simulation based on the parameter values,
* surrogate\_model : the model used to predict the output
* surrogate\_utility\_fn: a function to evaluate the utility of the predicted output
* iteration\_space : the space of possible iterations,
* param\_range : the range of each parameter
* param\_space : the space of possible parameter values), and various optimization options.

The function first identifies the indices of the enabled parameters in the param\_space definition. If any parameters are disabled, they are excluded from the optimization. The parameter values and space are then reduced to only include the enabled parameters.

The function then sets the parameters for the differential evolution algorithm, including the mutation and recombination rates, population size, and iteration limit. If opt\_store\_intermediate is True, the function also sets options to store the intermediate populations during the optimization.

The function then runs the differential evolution algorithm sample\_size times, each time optimizing the enabled parameters based on the utility function. The optimal parameter sets and their corresponding utility values are stored in optimal\_sets.

The function identifies the index of the optimal set with the highest utility value and returns the parameter set associated with that index. If opt\_store\_intermediate is True, the function also returns the intermediate populations during the optimization.

Differential evolution (DE) is a population-based metaheuristic search algorithm that optimizes a problem by iteratively improving a candidate solution based on an evolutionary process. Such algorithms make few or no assumptions about the underlying optimization problem and can quickly explore very large design spaces.

# def fn\_reference\_sample(sps, ref\_type=0, target=None, \*\*kwargs):

This function fn\_reference\_sample returns a reference sample point from the provided parameter space. The reference sample point is used for mutation in later iterations of the optimization process.

The function takes the following arguments:

* sps: a data frame with the parameter space, where each row represents a parameter configuration and each column represents a different parameter.
* ref\_type: an integer that determines the type of reference sample point to be returned. It can be either 0 (for the optimal point) or 1 (for the last point in the space).
* target: a character string specifying the name of the target variable to optimize. It is used only if ref\_type = 0.

If the parameter space sps is not empty, the function calculates the reference sample point based on the ref\_type argument. If ref\_type = 0, it returns the parameter configuration with the minimum value of the target variable specified by target. If ref\_type = 1, it returns the last row of the data frame sps. If the parameter space sps is empty, the function returns NULL. The function uses the assert\_that function from the assertthat package to check if ref\_type is either 0 or 1.

# def fn\_optimal\_acquisition(param\_space, iter\_space = None, sample\_size = 1, omit = [], inner\_optimisation\_fn = fn\_optimisation\_uniform\_prior, sampling\_fn = fn\_sampling\_lhs, surrogate\_predict\_fn = None, surrogate\_utility\_fn = None, \*\*kwargs):

This function is used to find the optimal acquisition point for Bayesian optimization.

Input arguments:

* param\_space
* iter\_space
* sample\_size
* omit
* inner\_optimisation\_fn
* sampling\_fn
* surrogate\_predict\_fn
* surrogate\_utility\_fn
* and .... (\*\*kwargs)

The arguments param\_space and iter\_space are required, while the others have default values.

1. The code first checks that the inner\_optimisation\_fn and sampling\_fn arguments are valid functions using the assert\_that() function from the assertthat package.
2. The code extracts the space object from the param\_space argument, which defines the search space for the optimization problem. It also sets default lower and upper limits for the search space if the param\_space object doesn't have any predefined parameter definitions.
3. The fn\_reference\_sample function is called to generate a reference sample point for future mutation. This is essentially an initial starting point for the optimization algorithm.
4. Any parameters that are not necessary are omitted from the reference sample point using the omit argument. This can be useful when certain parameters are not important to the optimization problem.
5. The inner\_optimisation\_fn function is called with several arguments, including the reference sample point, the sampling\_fn function, the surrogate\_predict\_fn function, the surrogate\_model, and other relevant parameters. This function is responsible for actually finding the optimal acquisition point based on the input parameters.
6. The function returns the optimal acquisition point as a data frame.

Overall, this function is an important part of the Bayesian optimization process, as it determines the next point to be sampled based on the results of previous samples.

# def fn\_enforce\_bounds(value, param, param\_def):

The function fn\_enforce\_bounds takes three arguments:

* value (the value to be checked),
* param (the parameter to which the value belongs)
* param\_def (a data frame that defines the bounds for each parameter).

The function first filters the param\_def data frame to get the row corresponding to the param parameter. If the parameter is not defined in the data frame, the function simply returns the input value. If the parameter is defined, the function checks if the input value is within the lower and upper bounds specified in the param\_def data frame. If the value is outside the bounds, the function returns the corresponding bound. Otherwise, the function returns the input value. This function is used to enforce the parameter bounds during the optimization process, so that the optimizer does not produce values that are outside the allowed range.

# def fn\_normalize\_min\_max(ds, param\_def):

The fn\_normalize\_min\_max function takes a dataset ds and a parameter definition param\_def and normalizes the columns in ds that correspond to parameters in the param\_def based on their minimum and maximum values. The function first checks if ds is a 1-dimensional array and if so, converts it to a DataFrame with a single row. Then, it creates a param\_mat DataFrame from the param\_def with columns for the parameter name, lower limit, and upper limit. The function then identifies the subset of columns in ds that correspond to parameters in param\_def using params\_used, and the subset of columns in ds that are not used in params\_omitted.

Next, for each column in params\_used, the function calculates the minimum and maximum values of the corresponding parameter from param\_mat and uses these to normalize the column values. The normalized columns are then concatenated into a new DataFrame input\_space with the same columns as params\_used. If there are any columns in params\_omitted, the function concatenates these onto the end of input\_space as well. The normalized DataFrame input\_space is then returned.

# def fn\_scaleup\_standard(ds, param\_def):

This function takes a dataset ds and param\_def dataframe which defines the parameters and their bounds. It scales up the values in ds to the original parameter ranges defined by param\_def.

First, it checks if ds is a 1D array, if it is, it converts it to a row vector dataframe. Then, it extracts the parameter information from param\_def dataframe and determines which columns of ds correspond to the parameters. It then scales up the values of each parameter to the original range and combines them into a matrix. Finally, it appends any omitted columns in ds to the end of the matrix and returns it as a dataframe with column names.

# def fn\_fit\_gp(ds, target, method\_params = {'KERNEL': 'exponential', 'POWER': 1.95}, test\_ds = None):

The function takes in :

* a dataset ds
* the name of the target variable target
* optional parameters for the Gaussian process model method\_params

The function first extracts the design matrix (i.e., the matrix of predictor variables) from the dataset ds.

It then creates a GPRegression object from the GPy library using the design matrix and target variable, as well as the kernel specified by the method\_params parameter.

The function optimizes the hyperparameters of the Gaussian process model using the optimize() method of the GPRegression object.

Finally, if a test dataset is provided, the function computes the root mean squared error (RMSE) of the model's predictions on the test dataset and stores it in the output dictionary under the key 'performance'.

# def fn\_fit\_mlegp(ds, target, method\_params = {}, test\_ds = None):

Note that in the Python version, we are using the GPy.kern.RBF kernel function instead of the default kernel used in mlegp function, which is the squared exponential kernel. We are also optimizing the model parameters using the optimize() method of the GPRegression class. Finally, we are computing the RMSE (root mean squared error) instead of the MSE (mean squared error) for the performance metric.

# def fn\_fit\_rf(ds, target, method\_params={"RF\_NSIZE": 1, "RF\_NTREE": 500}, test\_ds=None, keep\_inbag=True):

This function fits a Random Forest model to a dataset using the scikit-learn library. Here is a breakdown of the code:

The function takes in the following arguments:

* ds: a Pandas DataFrame containing the training data.
* target: a string representing the name of the target variable in the training data.
* method\_params: a dictionary containing hyperparameters for the Random Forest model. The default values are RF\_NSIZE=1 (minimum number of samples required to split an internal node), RF\_NTREE=500 (number of trees in the forest), and RF\_MTRY (number of features to consider when looking for the best split at each node), which is set to the square root of the number of features by default.
* test\_ds: a Pandas DataFrame containing the test data. This argument is optional.
* keep\_inbag: a boolean indicating whether to compute and store the "in-bag" score (i.e., the prediction error on the training set).

If RF\_MTRY is not specified in method\_params, it is set to the square root of the number of features in the dataset minus one.

The X and y variables are created by extracting the feature columns and target column from the training data, respectively.

A Random Forest regressor object is created using the RandomForestRegressor class from scikit-learn, with hyperparameters specified in method\_params.

The model is trained on the training data using the fit method of the regressor object.

The output is a dictionary containing the trained model and optionally the performance on the test data, which is the root mean squared error between the true target values and the predicted values.

# def fn\_fit\_rf\_ranger(ds, target, ethod\_params={"RF\_NSIZE": 1, "RF\_NTREE": 500}, test\_ds=None, keep\_inbag=True):

This function fits a random forest model using the ranger package in R.

The function takes as input:

* dataset (ds)
* target variable (target)
* optional arguments for method\_params
* test\_ds
* keep\_inbag

Inside the function, ranger is imported using the importr function from the rpy2 package. The formula for the random forest model is created using the Formula function from rpy2. The ranger.ranger function is then called with the formula, dataset, and various method parameters, and the resulting model is stored in the output dictionary under the key "model".

If test\_ds is not None, the function calculates the root mean squared error (RMSE) of the model's predictions on the test dataset and stores it in the output dictionary under the key "performance".

# def fn\_do\_folding(ds, folds=10, loo=False):

This is the Python equivalent of the R function fn\_do\_folding(), which performs k-fold cross-validation on a dataset. The function takes in three arguments:

* ds: the dataset to perform cross-validation on
* folds: the number of folds to use (default is 10)
* loo: whether to use leave-one-out cross-validation (default is False)

The function first checks if the number of rows in the dataset is less than or equal to the number of folds. If so, it sets folds to the number of rows and loo to True.

It then initializes an empty list test\_ds\_idx to hold the indices of the rows that will be used as test sets in each fold.

If loo is True, it randomly samples folds indices from the range 1 to the number of rows in the dataset, without replacement, using the np.random.choice() function.

If loo is False, it calculates the size of each fold as the floor division of the number of rows in the dataset by folds, and then randomly samples indices from the range 1 to the number of rows, with replacement, for each fold using a list comprehension.

The function then returns a list of dictionaries, with each dictionary containing two keys: "train" and "test". The "train" key holds the training dataset, which is the original dataset with the rows corresponding to the test set indices removed. The "test" key holds the test dataset, which is a subset of the original dataset consisting of the rows corresponding to the test set indices.

We use the pandas library to manipulate the dataset and numpy to generate the random indices.

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation.

# def fn\_fit(type, ds, target, param\_def, is\_normalization\_required=False, method\_params=np.nan, cv\_parallel=True, cv\_cores=multiprocessing.cpu\_count() - 2, \*args, \*\*kwargs):

The fn\_fit function takes as input several parameters including :

* type: type of model to be used
* ds: the dataset
* target: the target variable
* param\_def: parameter definition
* is\_normalization\_required : whether or not normalization is required
* method\_params :method parameters
* several optional arguments (\*args and \*\*kwargs).

The function first identifies the index of the target variable within the dataset and applies filtering if the param\_def dataframe contains an "enabled" column. If normalization is required, the function applies the fn\_normalize\_min\_max function to the dataset.

If method parameters are provided, the function creates a grid of all possible parameter combinations and splits the dataset into training and testing folds using the fn\_do\_folding function. It then fits the model on each training fold using either parallel processing or a serial loop and computes the performance of the resulting model on the corresponding testing fold. The performance metrics for all parameter combinations are stored in a list and the function identifies the best-performing model based on the mean performance across all testing folds. Finally, the best-performing model is trained on the entire dataset and returned as a list of models, each of which is associated with a subset of the input dataset.

If no method parameters are provided, the function splits the dataset into subspaces using the fn\_split\_space function and fits the model on each subspace, returning a list of models.

def fn\_split\_space(ds):

The function splits the dataset into a list of subspaces for fitting multiple models.

* The function takes a dataset ds as its input.
* The variable no\_subspaces is initialized to 1, and is updated to the value of shared.env['settings']['SUBSPACES\_NUMBER'] if this key exists in the shared.env['settings'] dictionary.
* The variable size\_subspace is initialized to the ceiling of (ds.shape[1] - 1) / no\_subspaces, and is updated to the ceiling of (ds.shape[1] - 1) \* shared.env['settings']['SUBSPACES\_SIZE'] if this key exists in the shared.env['settings'] dictionary.
* The variable overlap\_subspaces is initialized to False, and is updated to the value of shared.env['settings']['SUBSPACES\_OVERLAP'] if this key exists in the shared.env['settings'] dictionary. If the total number of features in ds minus 1 is less than the product of no\_subspaces and size\_subspace, overlap\_subspaces is set to True.
* The variable output is initialized as an empty list.
* If overlap\_subspaces is True, the lapply function is used to iterate no\_subspaces times and generate a list of subspaces. For each iteration, a col\_idx variable is generated by randomly sampling size\_subspace indices from the range (ds.shape[1] - 1) without replacement, and the resulting col\_idx list is sorted and appended to output.
* If overlap\_subspaces is False, a col\_set variable is generated as a list of indices from 1 to ds.shape[1] - 1. A for loop is used to iterate no\_subspaces times, and for each iteration, a gen\_vector variable is generated by randomly sampling size\_subspace indices from col\_set without replacement. The resulting gen\_vector list is sorted and appended to output, and col\_set is updated to be the set difference of col\_set and gen\_vector.

Finally, the function returns the output list of subspaces.

# def fn\_predict\_gp(surrogate\_model, ds):

This function takes in

* Gaussian process surrogate model (surrogate\_model)
* data set (ds) as input.

It uses the predict method of the surrogate\_model object to generate predictions for the data set, which returns a GPy.models.gp\_regression.Prediction object.

The function then returns a dictionary containing three keys:

* pred: the Prediction object returned by the predict method.
* mu: the mean predictions for the data set, computed by calling the mean method of the Prediction object.
* sigma: the standard deviation of the predictions for the data set, computed by calling the var method of the Prediction object and taking the square root.

This function can be used to generate predictions using a Gaussian process surrogate model trained on some data set, and returns both the predicted values and their associated uncertainty estimates.

# def fn\_predict\_mlegp(surrogate\_model, ds):

The function called takes two arguments:

* surrogate\_model: a trained Gaussian process (GP) model
* ds: a data frame of input points to be used for prediction.

Within the function, the predict.gp() function is called on the surrogate\_model using the newdata parameter set to ds and se.fit set to True. This function returns a named list containing the predicted mean and standard deviation at each input point in ds. The function then returns a list with three elements:

* Pred: the named list returned by predict.gp()
* mu: predicted mean
* sigma: standard deviation

They are converted to one-dimensional arrays using as.vector().

# def fn\_predict\_rf(surrogate\_model, ds):

The function is used to predict the response for new data using a trained random forest model. The newdata argument is used to provide the new data and predict.all argument is set to True to return individual predictions for all trees in the forest.

The function returns a dictionary with the following keys:

* "pred": individual predictions for all trees in the forest.
* "mu": predicted values for the new data obtained by aggregating individual predictions of all trees in the forest.
* "sigma": standard deviation of individual predictions for each data point obtained using np.apply\_along\_axis function.
* "inbag": proportion of times each data point is included in the bootstrap samples used to grow the trees.

# def fn\_predict(type, surrogate\_model, ds, param\_def=None, is\_normalization\_required=False, \*\*kwargs):

The function takes the following arguments:

* param\_def : a pandas DataFrame, and the "enabled" column is checked using the in operator instead of %in%.
* \*\*kwargs : variable keyword arguments to the type function, which allows for greater flexibility in specifying additional arguments.

The function works by first applying feature filtering to the input ds data frame, if the param\_def argument is not None and the "enabled" column is present. Then, if normalization is required and param\_def is not None, the fn\_normalize\_min\_max() function is called to normalize the input data.

Next, the function applies the surrogate models in the surrogate\_model list by calling the type function on each subspace of the input data. The results are collected in a list of dictionaries, where each dictionary contains the predicted values, means, standard deviations, and so on, for each subspace.

Finally, the function concatenates the results from each subspace into a single dictionary of lists, where the keys are the headers of the output. The dictionary is returned as the output of the function.

# def fn\_bayes\_optimization

Bayesian optimization is a sequential model-based optimization technique that uses a probabilistic model to approximate the objective function and finds the next best set of hyperparameters to evaluate using an acquisition function. The function you provided seems to be a template function for Bayesian optimization in R that uses several user-defined functions to define the search space, surrogate model, acquisition function, termination criterion, and other parameters.

The basic workflow of the function appears to be as follows:

1. Define the search space using a parameter definition and optionally add additional parameters.
2. Generate an initial set of parameter values to be evaluated using an initial sampling function.
3. Evaluate the initial set of parameter values using a simulation function and calculate the corresponding values of the objective function.
4. Fit a surrogate model to the evaluated parameter-objective function pairs.
5. Repeat the following steps until a termination criterion is met:
   1. Use the surrogate model to select the next best set of parameters to evaluate using an acquisition function.
   2. Evaluate the objective function for the selected parameter values.
   3. Add the evaluated parameter-objective function pair to the existing data and update the surrogate model.
6. Return the final set of evaluated parameters and corresponding values of the objective function.

The function allows the user to define various functions to customize the optimization process, including the surrogate model fitting and prediction functions, the acquisition function, the initial sampling function, and the termination criterion. The function also allows the user to specify various parameters, such as the number of initial samples, the number of samples to be evaluated in each iteration, and the number of potential parameter sets to be evaluated using the acquisition function.

# def fn\_assemble\_prediction\_traits(prediction\_obj):

This code uses the NumPy library to perform operations on matrices and arrays. It first defines an empty dictionary prediction\_traits to hold the output. Then, it computes the mean of the mu and sigma arrays for all rows, after stacking them into a single matrix with a column for each value in each array. Finally, it concatenates the pred array along the second axis (columns) to create a values array, and creates a custom dictionary containing all the entries from prediction\_obj that are not mu, sigma, or pred. The resulting dictionary is then returned.

# def fn\_assemble\_prediction\_traits\_single(prediction\_obj):

This function takes a dictionary prediction\_obj as input, creates a new dictionary prediction\_traits, and extracts some key-value pairs from the input dictionary to add to the output dictionary. The custom key in the output dictionary is populated with any remaining key-value pairs from the input dictionary that are not mu, sigma, or pred.

# def fn\_pull\_optimal\_samples(space, target, objective\_fn=min, selection\_fn=None, \*\*kwargs):

The function takes four parameters:

* space: a data frame or a list of dictionaries containing the search space, where each row or dictionary represents a set of hyperparameters to be evaluated.
* target: a string indicating the name of the column or key in the dictionary that represents the metric to optimize.
* objective\_fn: a function to use as the optimization objective. The default value is min, which means the function will try to minimize the target metric.
* selection\_fn: an optional function that takes the best performing hyperparameters and further refines the search space. The function should return a dictionary with two keys: space (a subset of the input search space containing the selected hyperparameters) and idx (the index of the selected hyperparameters in the original search space).

The function first finds the indices of the hyperparameters that achieve the best performance, based on the optimization objective specified by objective\_fn. It does this by creating a list of all indices where the target metric equals the result of applying the objective function to the list of target metric values in the search space. If there are multiple hyperparameter sets that achieve the same best performance, the function returns all of them.

Next, if a selection\_fn is specified, the function uses it to further refine the search space by selecting a subset of the best performing hyperparameters. The function calls selection\_fn with the best performing hyperparameters and any additional keyword arguments passed via the ... operator. The result is a dictionary containing the selected hyperparameters and their indices in the original search space. The function updates the best\_performer dictionary with the selected hyperparameters and their index.

Finally, the function computes the value of the target metric for the selected hyperparameters, based on the optimization objective specified by objective\_fn. It updates the best\_performer dictionary with the computed value and returns it.

# def fn\_pull\_optimal\_selection(space, randomized\_selection=False, selection\_size=0):

# def fn\_resolve\_fn\_params(fn, \*args, \*\*kwargs):

The function works as follows:

* It first uses the inspect module to get the list of argument names of the input function.
* It then gets all the parameters provided in the function call by combining the positional arguments (\*args) and keyword arguments (\*\*kwargs) into a list.
* The function then checks if any arguments are requested to be omitted by checking if the first argument is an integer (for positional arguments) or a list/tuple/set (for keyword arguments), and removes them from the argument list.
* If there are no arguments left in the argument list, the function returns an empty list.
* For each argument left in the argument list, the function checks if it is in the parameter names provided in the function call. If it is, the argument value is replaced with the corresponding parameter value.
* The function returns the updated argument list.

Overall, this function takes a function and a set of arguments and keyword arguments, and returns the arguments with updated values based on the provided keyword arguments.