Regularized Maximum Likelihood Estimation for the Random Coefficients Model in Python

Emil Mendoza*† Fabian Dunker‡ Marco Reale§
University of Canterbury University of Canterbury University of Canterbury

August 3, 2021

Abstract

We present **PyRMLE**, a Python module that implements Regularized Maximum Likelihood Estimation for the analysis of Random Coefficient models. **PyRMLE** is simple to use and readily works with data formats that are typical to Random Coefficient problems. The module makes use of Python's scientific libraries **NumPy** and **SciPy** for computational efficiency. The main implementation of the algorithm is executed purely in Python code which takes advantage of Python's high-level features.

1 Introduction

The Random Coefficients model is often used to model unobserved heterogeneity in a population, an important problem in econometrics and statistics. The model is given by

$$Y_i = \beta_{0i} + \beta_{1i}X_{1i} + \beta_{2i}X_{2i} + \ldots + \beta_{di}X_{di}. \tag{1}$$

Where $\mathbf{X}_i = (1, X_{1i}, X_{2i}, \ldots X_{di})^\top$ and $\boldsymbol{\beta}_i = (\beta_{0i}, \beta_{1i}, \beta_{2i}, \ldots, \beta_{di})^\top$ are the regressors, and regression coefficients respectively. It is assumed that $\mathbf{X}_i$, Y_i,

*School of Mathematics and Statistics, University of Canterbury, Private Bag 4800, Christchurch 8140, New Zealand, emil.mendoza@pg.canterbury.ac.nz
†Corresponding author
‡School of Mathematics and Statistics, University of Canterbury, Private Bag 4800, Christchurch 8140, New Zealand, fabian.dunker@canterbury.ac.nz
§School of Mathematics and Statistics, University of Canterbury, Private Bag 4800, Christchurch 8140, New Zealand, marco.reale@canterbury.ac.nz

$\boldsymbol{\beta}_i$ are i.i.d random variables with $\boldsymbol{\beta}_i$ and $\mathbf{X}_i$ being independent, and that Y_i and $\mathbf{X}_i$ are observed while $\boldsymbol{\beta}_i$ is unknown.

In this paper we introduce an open source `Python` module named **`PyRMLE`** which implemenents regularized maximum likelihood estimation (RMLE) to nonparametrically estimate the density $f_{\boldsymbol{\beta}}$.

Nonparametric estimation of the random coefficients model was first established by Beran and Hall (1992) for a single regressor. A kernel method for estimation was developed by Beran et al. (1996). The optimal rate for design densities with Cauchy-type tails is derived in Hoderlein et al. (2010) for a kernel method.

Equality constrained linear regression methods were developed by Fox et al. (2011); Heiss et al. (2021). Regularized maximum likelihood methods have been considered for problems other than the random coefficients model specified in (1) in Werner and Hohage (2012); Hohage and Werner (2013, 2016); Dunker and Hohage (2014). The method implemented by the **`PyRMLE`** module is the one developed in Dunker et al. (2021).

`Python` was chosen as the programming language mainly for the extensive scientific and computational libraries at its disposal, namely: **`NumPy, SciPy`** by Harris et al. (2020); Jones et al. (2001). The module takes advantage of the benefits of working with **`NumPy`** arrays in terms of computational efficiency achieved by doing array-wise computation. Another advantage is that there are no software imposed limits in terms of array size. The maximum array size in `Python` is solely determined by the amount of RAM available to the user, which allows the user the flexibility to increase the computational complexity of the method to the level that their system allows. The module also uses a trust-region constrained minimization algorithm developed by Byrd et al. (1999) which is implemented in `SciPy`.

The paper is organized as follows: Section 2 briefly describes the regularized maximum likelihood method developed in Dunker et al. (2021), Section 3 discusses the classes and functions available to the module, and Section 4 discusses examples of the modules usage for general cases.

2 Regularized Maximum Likelihood

It is assumed that the random coefficients, $\boldsymbol{\beta}$, have a Lebesgue density $f_{\boldsymbol{\beta}}$. If the conditional density $f_{Y|\mathbf{X}}$ exists, the two densities are connected by the integral equation

$$f_{Y|\mathbf{X}}(y|\mathbf{X}=\mathbf{x}) = \int_{\mathbb{R}^d} \mathbb{1}\{\mathbf{b}^\top \mathbf{x} = y\} f_{\boldsymbol{\beta}}(\mathbf{b}) d\mu_d(\mathbf{b}) = \int_{\mathbf{b}^\top \mathbf{x}=y} f_{\boldsymbol{\beta}}(\mathbf{b}) d\mu_d(\mathbf{b})$$

This connection allows us to employ maximum likelihood estimation to nonparametrically identify $f_{\boldsymbol{\beta}}$ as seen in the following expression of the log-likelihood

$$\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X}) = \frac{1}{n}\sum_{i=1}^{n}\log\left[\int_{\mathbb{R}} \mathbb{1}\left\{\boldsymbol{\beta}_i^\top \mathbf{X}_i = Y_i\right\} f_{\boldsymbol{\beta}}(b)d\mu(\mathbf{b})\right].$$

Direct maximization of $\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})$ over all densities it not feasible due to ill-posedness and will lead to overfitting. We stabilize the problem by adding a penalty term $\alpha\mathcal{R}(f_{\boldsymbol{\beta}})$ and state the estimator as a minimization problem with negative log-likelihood

$$\hat{f}_{\boldsymbol{\beta}_\alpha} = \underset{f\geq 0,\, \|f\|_{L^1}=1}{\arg\min} \; -\bar{\ell}(f|Y,\mathbf{X}) + \alpha\mathcal{R}(f). \tag{2}$$

Here $\alpha \geq 0$ is a regularization parameter that controls a bias variance trade-off. It was poined out in Heiss et al. (2021) that the constraint $\|f_{\boldsymbol{\beta}}\|_{L^1} = 1$ together with a finite difference discretization of $f_{\boldsymbol{\beta}}$ is equivalent to an ℓ^1 penalty on the discretized values of $f_{\boldsymbol{\beta}}$, which could cause unwanted shrinkage of the estimate. To reduce this LASSO effect, Heiss et al. (2021) introduced an additional quadratic constraint which turns the method into an elastic net. In Dunker et al. (2021) it was stated that the regularization term $\alpha\mathcal{R}(f_{\boldsymbol{\beta}})$ is analogous to this additional constraint but is more flexible as the method is not limited to quadratic $\mathcal{R}$.

The implemented regularization terms $\mathcal{R}$ in this module are: (1) squared L^2 norm $\mathcal{R}(f) = \|f\|_{L^2}^2 = \|f\|_2^2$, (2) the Sobolev Norm for H^1 $\mathcal{R}(f) = \|f_\beta\|_2^2 + \|f'_\beta\|_2^2$, and (3) entropy $\mathcal{R}(f) = \int f(\mathbf{b})\ln f(\mathbf{b})d\mathbf{b}$.

In addition to the regularization functional, a regularization parameter α also needs to be chosen. In this module we implement two methods of estimating α: Lepskii's Balancing principle, and K-fold cross validation.

3 Python Implementation

The `PyRMLE` module's implementation of regularized maximum likelihood is limited to applications with up to two regressors for the random coefficients model with intercept, and up to three regressors for a model without intercept.

There are two main functions used to implement regularized maximum likelihood estimation using `PyRMLE`, namely: `transmatrix()` and `rmle()`. There are other sub-functions necessary to the implementation, these will be discussed under the appropriate subsections when relevant.

3.1 The `transmatrix()` Function

The purpose of the function `transmatrix()` is to construct the discrete version of the linear operator given by

$$Tf_\beta = \int_{\mathbf{b}^\top \mathbf{x}=y} f_{\boldsymbol{\beta}}(\mathbf{b}) d\mu_d(\mathbf{b}). \tag{3}$$

The linear operator above describes the integral of $f_{\boldsymbol{\beta}}$ over the hyperplanes parametrized by the sample points $\mathbf{X}$, and Y. The function makes use of a finite-volume method as a discrete analog in evaluating the integral. The function is used as follows

```
trans_matrix = transmatrix(sample, grid)
```

The argument `sample` corresponds to the sample observations. The sample data should be in the following format:

$$\begin{bmatrix} X_{0,1} & X_{1,1} & \dots & Y_1 \\ X_{0,2} & X_{1,2} & \dots & Y_2 \\ \vdots & \vdots & \ddots & \vdots \\ X_{0,n} & X_{1,n} & \dots & Y_n \end{bmatrix}$$

In the case of a random coefficients model with intercept the first column would simply be $\mathbf{X}_0 = (1, 1, \dots, 1)^T$.

The `grid` argument is a class object generated by the `grid_set()` function. It has the following attributes and methods: { `scale`, `shifts`, `interval`, `dim`, `step`, `start`, `end`, `ks()`, `numgridpoints()`}. The `grid_set()` function is used as follows:

```
grid_beta = grid_set(num_grid_points=20,dim = 2)
```

The base grid that is generated by the `grid_set()` function is a symmetric grid that spans $[-5, 5]$ in each axis. The user inputs the number of grid points by passing an integer value to the function as **`num_grid_points`**. This specifies the step size of the grid as $\frac{10}{k}$ where k is the number of grid points along each axis. Additionally, the user can change the range over which each axis is defined by supplying new axes ranges through the arguments: **`B0_range, B1_range, B2_range`** which are passed as lists or arrays that contain the end points of the new range (e.g. **`B0_range = [0,10]`**). This is especially useful if the user expects a random coefficient to be significantly larger or smaller than the other random coefficients.

```
grid_beta_shifted = grid_set(num_grid_points = 20, \
dim = 2, B0_range = [0,10])
print(grid_beta_shifted.shifts)
[-5,0,0]
```

The output of the `transmatrix()` function is the '`tmatrix`' class object that has the following attributes and methods:

`Tmat`: returns a $n \times m$ `NumPy`-array that is produced by the function `transmatrix_2d()` or `transmatrix_3d()`.
`grid`: returns the **`class`** `grid_obj`.
`scaled_sample`: returns the scaled and shifted sample.
`sample`: returns the original sample.
`n()`: returns the number of sample observations.
m(): returns the number of grid points $\hat{f}_{\boldsymbol{\beta}}$ is to be estimated over.

3.1.1 `transmatrix_2d()`

The 2-dimensional implementation of this method works for the random coefficients model with a single regressor and random intercept

$$y_i = \beta_{0_i} + \beta_{1_i} x_{1i}, \tag{4}$$

and the model with two regressors and no intercept.

$$y_i = \beta_{1_i} x_{1i} + \beta_{2_i} x_{2i} + \epsilon_i \tag{5}$$

The function first initializes an $n \times m$-dimensional array of zeros, where n is the sample size, and m is the number of grid points $f_{\boldsymbol{\beta}}$ is to be estimated over. In the 1-dimensional case, the hyperplanes which $f_{\boldsymbol{\beta}}$ is integrated over reduce to lines which simplifies the problem. A finite-volume type method of estimation is employed to approximate the integral expression as seen in 3. This method is akin to the algebraic reconstruction methods used in Computed Tomography. Specifically, it is reminiscent to the discrete Radon Transform methodology laid out by Beylkin (1987).

To implement this finite-volume method, the lines parametrized by the sample points given by equations (4) or (5) are made to intersect with the grid. The length of each line segment that intersects with the grid is then calculated and stored in an object. The intersection points of these lines with the grid are retrieved and subsequently mapped to their respective array indices. These indices are used to map the length of the line segments to the appropriate entry in the initialized array forming the discretized linear

operator T. The algorithm is outlined as follows:

Algorithm 1: `transmatrix_2d()`

Input: *sample, grid*
Initialization *Initialize* ***NumPy*** *Array of Zeros*
for *s in sample* **do**
  1. get intersection points;
  2. get line segment lengths;
  3. map intersection points to their array indices;
  for *i in indices* **do**
    map line segment lengths to initialized array using index i
  end
end

The result of this function is a large, sparse array, $\mathbf{T}$ where each row corresponds to a collection of all the lengths of the line segments intersecting the grid for a line parametrized by a sample point, i.e. each l_{ij} corresponds to the length of the line segment intersecting the grid at section i, j. The resulting array is sparse because l_{ij} is equal to zero unless a line passes through a section of the grid. The algorithm's implementation is illustrated in figure 1.

The resulting array is then used to evaluate the log-likelihood functional to be optimized

$$\bar{\ell}(f_{\boldsymbol{\beta}}|Y, \mathbf{X}) = \frac{1}{n}\sum_{i=1}^{n} \log \mathbf{T} f^*_{\boldsymbol{\beta}}, \tag{6}$$

where $f^*_{\boldsymbol{\beta}} = (f_{\boldsymbol{\beta}_1}, f_{\boldsymbol{\beta}_2}, \ldots, f_{\boldsymbol{\beta}_m})^T$ is a $m \times 1$-array or vector that serves as the discrete approximation of $f_{\boldsymbol{\beta}}$.

3.1.2 `transmatrix_3d()`

The implementation of the function for the 3-dimensional case works for problems with two regressors and a random intercept,

$$y_i = \beta_{0i} + \beta_{1i}x_{1i} + \beta_{2i}x_{2i}, \tag{7}$$

and the model with three regressors and no intercept.

$$y_i = \beta_{1i}x_{1i} + \beta_{2i}x_{2i} + \beta_{3i}x_{3i} + \epsilon_i \tag{8}$$

Note: for the three-dimensional implementation of the algorithm numerical instabilities occur when the underlying joint density, $f_{\boldsymbol{\beta}}$, has a single

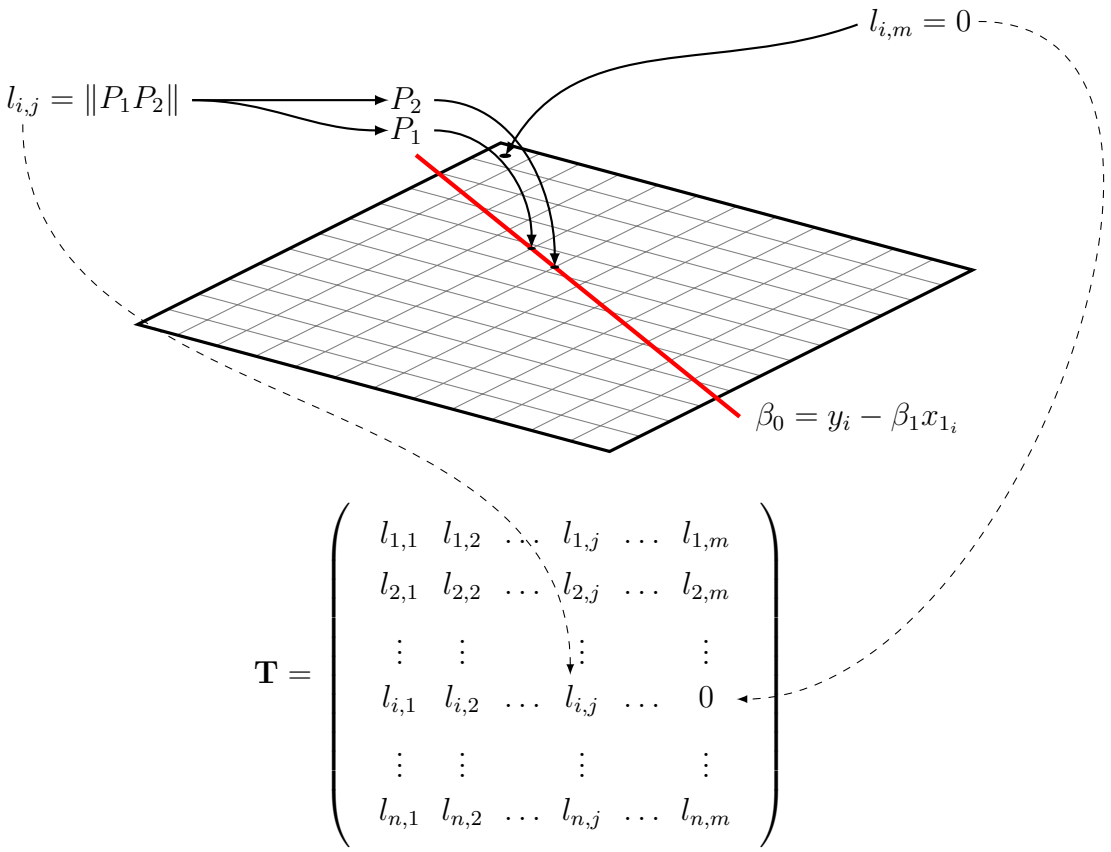

Figure 1: 2-D Transformation Matrix Algorithm

mode located at the center of the grid. This problem can occur in two ways: (1) it can occur organically if the mode of the density is located close to or at $(0, 0, 0)$ with the grid axes ranges at their default values [-5,5], or (2) artificially if the user provides grid ranges such that the mode of $f_{\boldsymbol{\beta}}$ is located at its center. This problem is overcome by simply imposing a shift onto the resulting density by applying a linear transformation to the sample data, as below:

$$Y_i = (\beta_{0_i} + c) + \beta_{1_i} X_1 + \beta_{2_i} X_2. \tag{9}$$

This can be achieved in two ways: (1) the more computationally efficient method is to simply supply a grid range for β_0 that offsets the mode of $\hat{f}_{\boldsymbol{\beta}}$ from the center, or (2) apply the shifting algorithm described in (9) which allows the user to supply any grid range but with an additional computational cost.

Similar to the 2-dimensional implementation, a finite volume type ap-

proach is used to create the discrete version of the linear operator T. In this higher-dimensional case the hyper-planes parametrized by the sample observations are now 2-dimensional planes, and the grid that $f_{\boldsymbol{\beta}}$ is estimated over is comprised of three axes and is therefore in the shape of a 3-dimensional cube. In this case, the intersection of the plane with the grid are characterized by a collection of points that define a polygon whose areas are used as the 2-dimensional analog of the line segments in the lower dimensional implementation of this finite volume estimation process. The algorithm is outlined below. Figure 2 also illustrates the algorithm for a single cuboidal grid section.

Algorithm 2: `transmatrix_3d()`

Input: *sample, grid*
Initialization *Initialize* ***NumPy*** *Array of Zeros*
for *s in sample* **do**
  get intersection points;
  for *p in intersection points* **do**
    map intersection points to array indices;
    for *i in indices* **do**
      assign intersection point to grid location (a point can belong to more than one grid box);
    end
  end
  for *m in assigned points* **do**
    sort points to form polygon and generate the area
  end
  for *i in indices* **do**
    map each polygon's area to the initialized array
  end
end

The resulting array is in the same form as the one obtained using the 2-dimensional implementation `transmatrix_2d()`, and can be used in the following mannner to evaluate the log-likelihood functional in (6)

$$\mathbf{T}f^*_{\boldsymbol{\beta}} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,m} \\ a_{3,1} & a_{3,2} & a_{3,3} & \dots & l_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,m} \end{bmatrix} \begin{bmatrix} f_{\beta_1} \\ f_{\beta_2} \\ f_{\beta_3} \\ \vdots \\ f_{\beta_m} \end{bmatrix},$$

where in this case each a_{ij} corresponds to the area of the polygonal in-

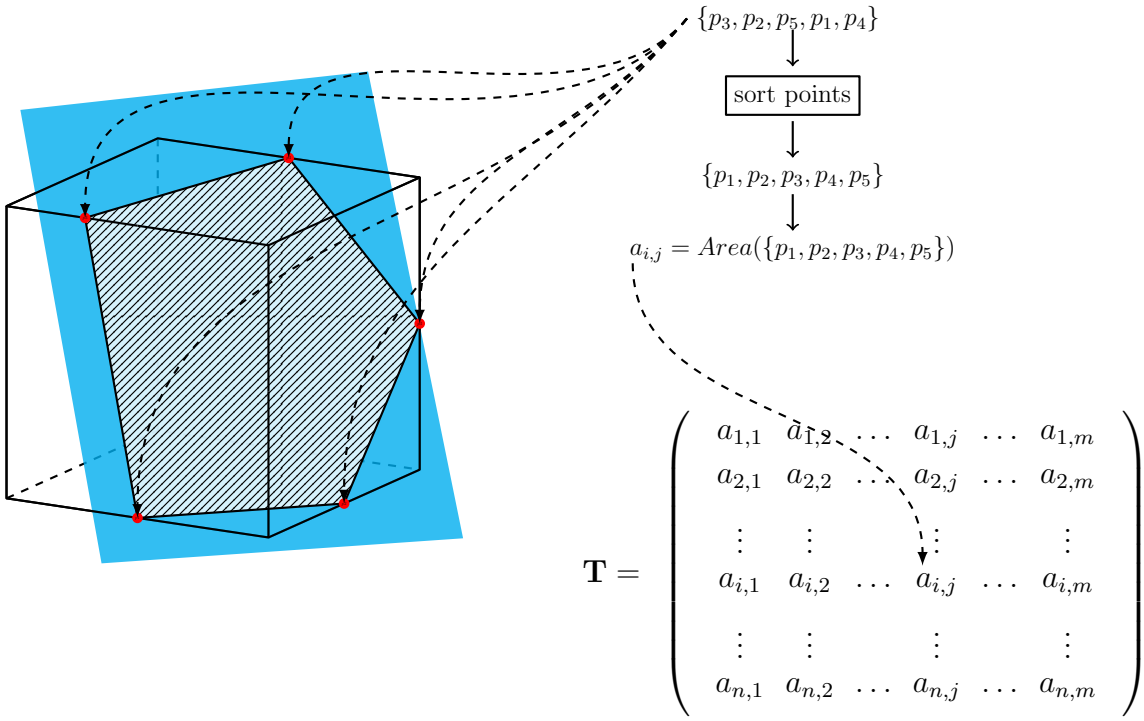

Figure 2: 3-D Transformation Matrix Algorithm

tersection of the plane with the discrete estimation grid. The resulting product of this matrix and vector is an $n \times 1$ array. Applying an element-wise log-transformation, then taking the sum of this $n \times 1$ array results in the expression in (6).

It is important to note the computational performance of this algorithm. Matrix multiplication parallelizes the computation and eliminates the need to evaluate the log-likelihood functional sequentially for each sample point. However, the process of creating the transformation matrix $\mathbf{T}$ scales linearly with the sample size n and exponentially with the dimensionality of the problem. The function was written in a way that maximizes efficiency as much as possible by making use of Python's parallel computation through array-wise computation when possible.

Solving for the transformation matrix, $\mathbf{T}$ is only one of two steps in the process of estimating the density $f_{\boldsymbol{\beta}}$. The second step is the process of optimizing the regularized log-likelihood functional, the run-time of which scales multiplicatively with respect to the sample size, n, and the number of discretezation points, m. Therefore, choosing an appropriate sample size and level of discretezation is an important consideration as both primarily

determine the total cost of running the algorithm.

3.2 The `rmle()` Function

The `rmle()` function returns a class named `'RMLEResult'`. This class stores the solution to the optimization problem, $f^*_{\boldsymbol{\beta}}$, and metadata about the solution and the process of optimization. An instance of `'RMLEResult'` has the following accessible attributes and methods:
`f`: returns a $m \times 1$ array containing all the estimated function values. It is necessary to reshape the solution before visual representation.
`f_shaped`: returns the reshaped array of f.
`dim`: returns an integer which represents how many dimensions $f_{\boldsymbol{\beta}}$ is estimated over.
`maxval()`: returns a list containing the maximum value of $f^*_{\boldsymbol{\beta}}$ and its location.
`mode()`: returns a list containing possible modes of the density and their locations.
`ev()`: returns atuple containing the expected value of each β_i.
`alpha`: returns a floating-point that specifies the regularization parameter, α, used for estimation.
`alpmth`: returns a string that specifies the method by which the regularization parameter, α was chosen. It can take on the following values: {'Lepskii', 'CV', 'User'}.
`T`: returns a class object that is created using the `transmatrix()` function. It contains attributes, methods and subclasses that contain information about the transformation matrix $\mathbf{T}$ and meta-data about it.
`Tmat`: returns an attribute of the sublcass `T` but also accessible from the `RMLEResult` class. It returns the transformation matrix $\mathbf{T}$.
`grid`: returns a class object that is created using the `grid_set()` function. It has attributes and methods that contain information about the grid $\hat{f}_{\boldsymbol{\beta}}$ is estimated over.
`details`: returns a dictionary containing metadata about the minimization process.

The function `rmle()` serves as a wrapper function for **SciPy**'s `minimize` function with `'trust-const'` set as the minimization algorithm. The choice of this algorithm is crucial as it specializes in large-scale constrained minimization problems, for further details we refer to Byrd et al. (1999). The ability to handle large-scale problems is important because depending on the size of the sample, and level of discretezation, the functional evaluations as well as the gradient evaluations could become exceedingly expensive, as it

scales with both in a multiplicative manner ($n \times m$). The option to set constraints was also an important consideration. As in equation (2) there are two important constraints in estimating $f_{\boldsymbol{\beta}}$, namely: $f \geq 0$, and $\|f\|_{L^1} = 1$. These two constraints ensure the resulting solution satisfies the definition of a density.

Table 1 contains all the arguments for the function `rmle()` followed by a short description of what they pertain to. More important arguments will be discussed in following subsections.

Table 1: `rmle()` arguments

Argument	**Description**
functional	Negative likelihood functional with corresponding regularization term
alpha	constant ≥ 0 that serves as the regularization parameter, or a string matching: 'cv' or 'lepskii'.
tmat	Class object returned by `transmatrix()` which contains information about the transformation matrix $\mathbf{T}$ and the grid it is estimated over.
k	Optional argument: integer which specifies how many folds for modified k-fold cross-validation. Default value is $k = 10$
initial_guess	Optional argument from the **SciPy Optimize** minimize function. Used to supply an initial value for the minimization algorithm to start. Default value is set to a **NumPy** array of values close to zero.
hessian_method	Optional argument from **SciPy Optimize** minimize function. Default value is set to '2-point'.
constraints	Refers to the linear constraints imposed onto the problem. It is set as an optional argument, the default value is set to $\|f\|_{L^1} = 1$
tolerance	Optional argument for the tolerance criteria for the optimization algorithm's termination. Default value is set to `1e-6`.
max_iter	Optional argument for the maximum number of iterations. Default value is set to 100.
bounds	Refers to the bound constraints of the optimization problem. The default is expressed as $f \geq 0$

3.2.1 Functionals and Regularization Terms

Recall the average log-likelihoood functional to be minimized as in equation (2). As specified in section 2 the regularization terms implemented in this module are: the Sobolev norm for H^1, the squared L^2 norm, and Entropy. The module also includes an option for a functional that has no regularization term if the user wishes to produce a reconstruction of $f_{\boldsymbol{\beta}}$ without any form of regularization. This option will often lead to overfitting, and produce a highly unstable solution.

3.2.2 Sobolev Norm for H^1

The functional incorporating the Sobolev norm for H^1 has the following form,

$$-\bar{\ell}(f_{\boldsymbol{\beta}}|Y, \mathbf{X}) + \alpha(\|f\|_2^2 + \|f'\|_2^2) \tag{10}$$

where $\|\cdot\|_2^2$ indicates the squared L^2 norm. It is important to note that the choice of the regularization term would typically depend on knowledge about the true solution, as the choice of the regularization term imposes certain assumptions on $f_{\boldsymbol{\beta}}$. In the case of the H^1 penalty term, the solution has a square-integrable first derivative in addition to the assumptions of non-negativity and integrability imposed by the constraints of the minimization problem.

The function `sobolev()`, or `sobolev_3d` for the 3-dimensional application returns the value of the discrete implementation of the functional in (10). Table 2 provides details on the arguments required for this function. The **SciPy Optimize** minimize function does not accept array arguments that are greater than one dimension. A necessary step is to unravel the transformation matrix, $\mathbf{T}$, into a one-dimensional array which is passed to the function as `tm_long` and simply reshaped into the proper array dimensions. The term $\mathbf{T}f$ is calculated using **NumPy's** matrix multiplication and sum functions which are much faster alternatives than their non-**Numpy** counterparts. The regularization term is calculated in (10), with the function `norm_fprime()` computes for $\|f'\|_2^2$ where f' is treated as a total derivative.

The underlying minimization function is able to approximate the Jacobian of the functional with an additional computational cost. For computational efficiency, we supply an explicit form for the Jacobian of the functional: $-\frac{\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})'}{\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})} + 2\alpha(f - f'')$

The form of this Jacobian implies an additional smoothness assumption on the solution, as it requires f_β to be twice differentiable.

Table 2: `sobolev()`, `sobolev_3d()` arguments

Argument	Notation	Description
f	$f^*_{\boldsymbol{\beta}}$	current value of the solution f^*_β
a	α	constant that serves as the regularization parameter
tm_long	$\mathbf{T}$	unraveled form of the transformation matrix, $\mathbf{T}$
n	n	the sample size
s	Δb	the step size of the grid

3.2.3 Squared L^2 Norm

The form of the functional in (2) that incorporates the squared L^2 norm as the regularization term is:

$$-\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})f + \alpha\|f\|_2^2$$

The arguments for this function are indentical to those listed in Table 2, likewise is true for the Jacobian associated with this regularization term. The Jacobian has the following form: $-\frac{\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})'}{\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})} + 2\alpha f$

Choosing this regularization functional imposes less smoothness assumptions on the solution, as the only additional assumption in place is square-integrability. This leads to a typically less smooth reconstruction as compared to using the Sobolev norm for H^1 as the regularization functional. The functions in python are coded similarly as with the H^1 regularization functional.

3.2.4 Entropy

The form of the functional in (2) that incorporates the entropy of the function has the following form: $-\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X}) + \alpha\int f\log(f)db$ This functional has the least amount of assumptions on the solution, $f_{\boldsymbol{\beta}}$. It only requires finite entropy which is a weak assumption in addition to the non-negativity and L^1 constraints of the minimization problem.

The Jacobian of the entropy functional also does not impose any additional assumptions on the solution, and has the following form: $-\frac{\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})'}{\bar{\ell}(f_{\boldsymbol{\beta}}|Y,\mathbf{X})} + \alpha(\log f + 1)$

3.2.5 Parameter Selection

Recall the minimization problem as in (2) where a constant $\alpha \geq 0$ controls the size of the effect of the regularization term. The user can provide the

α value directly if the user has a general idea of the level of smoothing necessary for the solution. If the user has no best-guess for the value of the regularization parameter alpha, the module has two options to automatically select the parameter α, namely: Lepskii's balancing principle, and k-fold cross-validation. The Lepskii method typically yields a less accurate result relative to k-fold cross-validation; however, its advantage lies in significantly less computational cost.

3.2.6 Lepskii's Balancing Principle

Lepskii's principle is an adaptive form of estimation which is popular for inverse problems (insert citations). It is significantly computationally less expensive than other parameter selection methods.

The method works as follows: we compute $\hat{f}_{\boldsymbol{\beta}_{\alpha_1}}, \ldots, \hat{f}_{\boldsymbol{\beta}_{\alpha_m}}$ for $\alpha_1 = c_{L_n} \frac{\ln(n)}{\sqrt{n}}$ and $\alpha_{i+1} = r\alpha_i$ with some constants $c_{L_n} > 0, r > 1$. We then select α_j as the optimal parameter choice where:

$$j_{bal} := \max\{j \leq m, \|\hat{f}_{\boldsymbol{\beta}_{\alpha_i}} - \hat{f}_{\boldsymbol{\beta}_{\alpha_j}}\| \leq 8r^{\frac{1-i}{2}}, \text{ for all } i < j\}.$$

The algorithm is implemented in python as follows:

Algorithm 3: Lepskii's Balancing Principle

```
Input: {α_1, α_2, ..., α_m}
Initialization Generate the transformation matrix T
for α_i in {α_1, α_2, ..., α_m} do
    Compute for f̂_{β_{α_i}} using T;
end
for j in {1, 2, ..., m} do
    Set i = 0 ;
    while i < j do
        Check if ||f̂_{β_{α_i}} − f̂_{β_{α_j}}|| ≤ 8r^{(1−i)/2} ;
        if ||f̂_{β_{α_i}} − f̂_{β_{α_j}}|| > 8r^{(1−i)/2} then
            j_bal = j ;
            break
        end
        i+ = 1
    end
end
```

The bulk of the computational cost of the Lepskii algorithm implementation can be broken down into two components: the fixed cost of generating

$\mathbf{T}$, and the variable cost of computing $\hat{f}_{\boldsymbol{\beta}_{\alpha_1}}, \ldots, \hat{f}_{\boldsymbol{\beta}_{\alpha_m}}$ as the number of α values to be used depends on $c_{L_n} > 0$, $r > 1$, and also the sample size of the data. This implementation of Lepskii algorithm's scales linearly in terms of runtime with the number of α values being tested, m.

3.2.7 K-Fold Cross Validation

Cross-validation is another popular parameter choice rule. Here we present a modified implementation of k-fold cross-validation with a cost-function that is applicable to our problem. The modification we apply is an algorithm to lessen the computation time by reducing the number of α values it needs to iterate through. The loss function we considered was,

$$J_{\alpha_i} = -\sum_{j=1}^{k} \log \mathbf{T}_j \hat{f}_{\boldsymbol{\beta}_{-j}}$$

Where $\mathbf{T}_j$ is the transformation matrix generated from a subsample of the observations, which can be interpreted as the j-th fold that is left out in the current iteration, and $\hat{f}_{\boldsymbol{\beta}_{-j}}$ is the estimate for $f_{\boldsymbol{\beta}}$ using $\mathbf{T}_{-j}$. The loss function can be interpreted as the negative of the likelihood that the j-th fold of the sample used to generate $\mathbf{T}_{-j}$ was drawn from the distribution, as the subsample fold used to produce $\mathbf{T}_j$. We aim to choose the α that minimizes this loss function.

The search method for the optimal α value reduces the number of α values tested. The algorithm involves separating the range of α values into two sections $\{\alpha_1, \ldots, \alpha_j\}$ and $\{\alpha_{j+1}, \ldots, \alpha_m\}$. Two alpha values α_a, and α_b are randomly selected from the respective sections and are used to compute for the corresponding loss function values, J_{α_a} and J_{α_b}. The section from which the α value that produces the smaller J_α was drawn from is kept, while the other is discarded. This is repeated until there is a sufficiently small range of α values. Once this range of α values is obtained, the loss function is evaluated over all the remaining α values and the optimal α is chosen as the one which minimizes J_α. The complete algorithm is implemented as follows:

Algorithm 4: Modified K-fold Cross Validation

Input: $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$

Initialization *Generate the transformation matrix* $\mathbf{T}$ *and apply a random shuffle, set* $\boldsymbol{\alpha} = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$

while *len*$(\boldsymbol{\alpha}) > 3$ **do**

1. Set $\boldsymbol{\alpha_a} = \{\alpha_1, \alpha_2, \ldots, \alpha_j\}$;
2. Set $\boldsymbol{\alpha_b} = \{\alpha_{j+1}, \alpha_{j+2}, \ldots, \alpha_m\}$;
3. Randomly select α_a and α_b from $\boldsymbol{\alpha_a}$ and $\boldsymbol{\alpha_b}$ respectively. ;
4. Evaluate J_{α_a} and J_{α_b} ;

if $J_{\alpha_a} < J_{\alpha_b}$ **then**
  Set $\boldsymbol{\alpha} = \boldsymbol{\alpha_a}$
end
else
  Set $\boldsymbol{\alpha} = \boldsymbol{\alpha_b}$
end

end

for α_i *in* $\boldsymbol{\alpha}$ **do**
  Compute for J_{α_i} ;
end

Choose $\alpha_{cv} = \arg\min J_{\alpha_i}$

The runtime of the unmodified version of k-fold cross-validation scales linearly with the product $k \times m$ where k is the number of folds and m is the number of α values being tested. Applying the modified version reduces the number of α values being tested, m, by some logarithmic factor, which is a significant reduction in computational cost which makes cross-validation more computationally feasible.

4 Examples

The following examples will be demonstrated in this section: the random coefficients model with a single regressor and random intercept for the two-dimensional case, and the two regressors and random intercept for the three-dimensional case. This section will also show how to plot the estimated density using the built in plotting function `plot_rmle()` which makes use of functions from the `matplotlib` library. It will also show how to plot the density without the use of the built-in function in case the user wishes to explore different plotting options.

4.1 Example 1: 2-D Case Single Regressor with Random Intercept

The first example is the case described by (4). The example is demonstrated with simulated data using the function `sim_sample()`. This function simulates the regressor $X_1 \sim \mathcal{U}(-2, 2)$ and the random coefficients β_0, β_1 from a bimodal multivariate normal mixture as follows,

$$0.5\mathcal{N}([-0.5, -0.5], 0.01\mathbb{I}_2) + 0.5\mathcal{N}([0.5, 0.5], 0.01\mathbb{I}_2).$$

The general flow of the process of using the module can be broken down in five steps:

1. Import the necessary modules and functions.
2. Establish the dataset to be used (either real or simulated data).
3. Specify the grid over which $\hat{f}_{\boldsymbol{\beta}}$ is to be estimated over.
4. Generate the transformation matrix $\mathbf{T}$.
5. Run the `rmle()` function.

```
from pyrmle import *

sample = sim_sample(n = 10000,dim = 2)
```

The program begins by importing the necessary functions from `pyrmle.py` which contains the high-level functions that the user interacts with. The module `pyrmle_funcs` contains the underlying functions necessary for functions in the main module `pyrmle` to run. The next step is to define the sample to be used in creating the transformation matrix $\mathbf{T}$. The sample has the same form as described in subsection 3.1, where the sample has the form $[\mathbf{X}_0, \mathbf{X}_1, \mathbf{Y}]$. In Python it takes the shape of 10000×3 `NumPy` array as seen below.

```
[[ 1.         , -0.76765455,  2.10802347],
 [ 1.         ,  1.51774991, -0.11337413],
 ...,
 [ 1.         , -1.80486996, -3.08120151]]
```

The next step is to generate the grid over which $\hat{f}_{\boldsymbol{\beta}}$ is estimated over. This is done using the `grid_set()` function, as discussed briefly in 3.1. In this example we set the ranges of β_0 and β_1 to $[-10, 10]$ which defines a two-dimensional grid spanning that range in each axis. This function creates an

instance of the **class** `grid_obj` which has attributes and methods enumerated and described in subsection 3.1. When the `grid` class instance has been created, the user can proceed to generate an instance of the **class** `tmatrix` using the `transmatrix()` function.

```
grid_beta = grid_set(num_grid_points = 40, dim = 2)
print(grid_beta.numgridpoints())
1600
T = transmatrix(sample, grid_beta)
```

After the instance of the transformation matrix is produced, the user can then run the `rmle()` function. As stated in subsection 3.2, the `rmle()` function has three essential arguments: {`'functional','alpha','tmat'`}.

```
result = rmle(sobolev,0.25,T)
print(result.ev())
[-0.002865326246726808, -0.010416375635973668]
print(result.mode()[:2])
[[0.39681799850755783, [-0.625, -0.375]],
[0.38831830923870914, [0.625, 0.375]]]
plot_rmle(result)
plot_rmle(result,plt_type='surface')
```

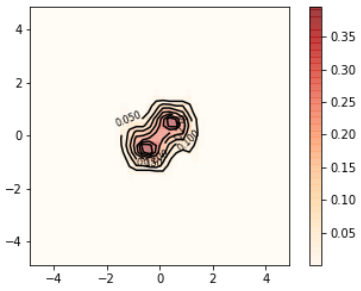

Figure 3: $\hat{f}_\beta$ contour plot with 40 grid points

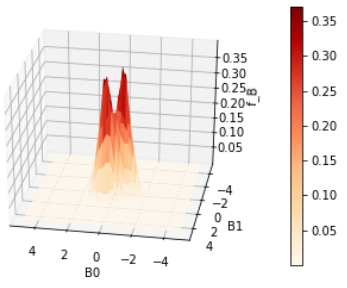

Figure 4: $\hat{f}_\beta$ surface plot with 40 grid points

As stated in subsection 3.2, the `rmle()` function generates an instance of **class** `RMLEResult`. This class has a number of useful and informative attributes and methods that describe the estimated density. The `ev()` method

returns the expected value of each $\boldsymbol{\beta}_j$, while the `mode()` method returns possible maxima of the estimated density. The `mode()` method relies on a naive search method for maxima with no underlying statistical tests.

Figure 3 (reference) shows the contour plot produced by the `plot_rmle()` function. It is clear that there is a large portion of the grid that is essentially unused, and the user could benefit from a reduction in the grid-size in terms of computational costs. This can be done by reducing the number of grid points and shrinking the range of each axis. In terms of tuning the size of the grid, we suggest that the user tries a relatively large range to begin with to ensure that the grid contains the support of $\hat{f}_{\boldsymbol{\beta}}$, and then consider smaller grid ranges. Having a grid range that is too small has a negative effect on the estimate as the optimization algorithm enforces the constraint that $\|\hat{f}_{\boldsymbol{\beta}}\|_{L^1} = 1$.

```
grid_beta_alt = grid_set(20,2,B0_range=[-1.5,1.5],\
B1_range=[-1.5,1.5])
print(grid_obj_alt.numgridpoints)
400
T2 = transmatrix(sample, grid_beta_alt)
result2 = rmle(sobolev,0.15,T2)

print(result2.ev())
[-0.004977073000670898, -0.003964663139211258]
print(result2.mode()[:2])
[[0.3643228046077392, [0.5249999999999, 0.5249999999999]],
[0.3580092260598125, [-0.5249999999999, -0.5249999999999]]]
plot_rmle(result2)
plot_rmle(result2,plt_type='surface')
```

The reduction in the number of grid points resulted in a significant reduction in the run time of the algorithm while achieving a better estimate for $\hat{f}_{\boldsymbol{\beta}}$. With the reduction in the computational cost of the algorithm makes it more favorable to run an automatic parameter choice method.

```
result_cv = rmle(sobolev,'cv',T2)
print(result_cv.alpha)
0.07065193045869372
print(result_cv.ev())
[-0.004977073000670898, -0.003964663139211258]
print(result_cv.mode()[:2])
[[0.3643228046077392, [0.5249999999999, 0.5249999999999]],
[0.3580092260598125, [-0.5249999999999, -0.5249999999999]]]
plot_rmle(result_cv)
```

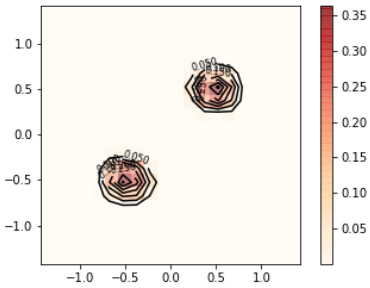

Figure 5: $\hat{f}_\beta$ contour plot with 20 grid points on a smaller grid

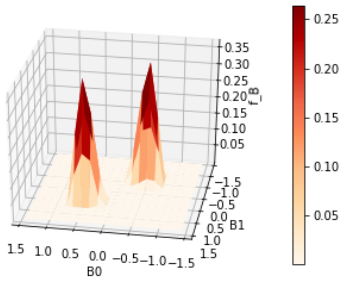

Figure 6: $\hat{f}_\beta$ surface plot with 20 grid points on a smaller grid

```
plot_rmle(result, plt_type='surface')
```

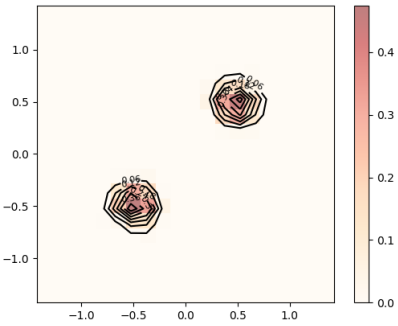

Figure 7: $\hat{f}_\beta$ contour plot with 20 grid points on a smaller grid using cross-validation

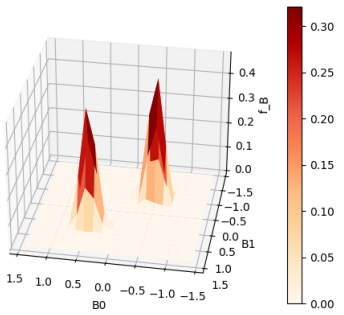

Figure 8: $\hat{f}_\beta$ surface plot with 20 grid points on a smaller grid using cross-validation

The general workflow that we suggest when tuning the parameters to be used in estimation is as follows:

1. Establish a relatively large grid range for estimation and generate the transformation matrix $\mathbf{T}$. This should be treated as an exploratory step in terms of analyzing the data.
2. Set α equal to the step size of the grid.
3. Run the `rmle()` function.
4. Plot $\hat{f}_{\boldsymbol{\beta}}$ using the `plot_rmle()` function and determine the necessary grid range.

5. Limit the grid range as well as the grid points to reduce computation costs and genrate the new matrix $\mathbf{T}^*$.
6. Run the `rmle()` function with $\mathbf{T}^*$, and optionally employ one of the two automatic parameter selection methods: {`'cv'`,`'lepskii'`}.

The following example will demonstrate usage of the `grid_set()` function in terms of supplying a different range β_1. The simulated data in this case will have modes for β_1 that are significantly larger than that of β_0 and are not encapsulated by the default range $[-5, 5]$. The betas are sampled from the following distribution: $0.5\mathcal{N}([-1.5, 6], \mathbb{I}_2) + 0.5\mathcal{N}([1.5, 9], \mathbb{I}_2)$.

```
cov = [[[1, 0], [0, 1]],[[1, 0], [0, 1]]]
mu = [[-1.5,6],[1.5,9]]
sample = sim_sample(10000,2,beta_mu = mu, beta_cov = cov))
grid_beta_shifted = grid_set(num_grid_points = 20, \
dim = 2, B1_range=[2,13])
T_shifted = transmatrix(sample, grid_beta_shifted)
result_shifted = rmle(sobolev_norm_penal,0.5,T_shifted)

print(result_shifted.ev())
[0.03520704073478552, 7.524001743037029]
print(result.mode()[0:2])
[[0.07133616078580148, [1.25, 8.875]],
[0.0652140364538059, [-1.75, 5.574999999999999]]]

plot_rmle(result_shifted)
plot_rmle(result_shifted,plt_type='surface')
```

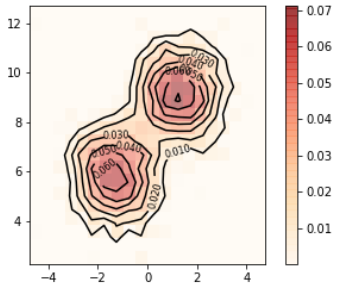

Figure 9: $\hat{f}_\beta$ contour plot with 20 grid points on shifted grid

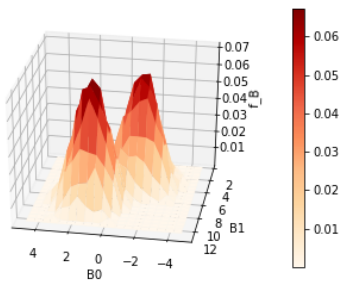

Figure 10: $\hat{f}_\beta$ surface plot with 20 grid points on a shifted grid

4.1.1 Example 2: 3-D Case Two Regressors with Random Intercept

This second example is the case described by (7). The example is demonstrated likewise with simulated data using the same function `sim_sample()`. The regressors are simulated as follows, X_1, X_2 are i.i.d $\mathcal{U}(-2, 2)$, and the random coefficients $\beta_0, \beta_1,$ and β_2 are simulated from $\mathcal{N}([2, 2, 2], 0.01\mathbb{I}_3)$

```
from pyrmle import *

sample = sim_sample(n = 10000,dim = 3)
```

As in the previous example, the program begins by importing the necessary modules and functions. The `sim_sample()` function generates sample observations based on the aforementioned distributions. This results in a $10,000 \times 4$ `NumPy` array.

```
grid_beta = grid_set(10,3,B0_range=[-2,4],\
B1_range=[-2,4],B2_range=[-2,4])
print(grid_beta.numgridpoints())
1000
T = transmatrix(sample,grid_beta)
```

The next step is to establish the number of grid points, and to generate the transformation using the simulated sample observations. In this case, we first consider ten grid points in each axis amounting to a total of 1000 grid points. If the user wishes to estimate $\hat{f}_{\boldsymbol{\beta}}$ over a finer grid it would be more efficient to first determine the smallest possible range of each axis that would fit almost all of the probability mass of $\hat{f}_{\boldsymbol{\beta}}$.

```
result = rmle(sobolev_norm_penal2d,0.6,T)
print(result.ev())
[2.458436489282234, 2.25500373629305, 1.9058740990043983]
print(result.mode())
[0.08926614291105403, [1.90000000, 1.90000000, 1.90000000]]
plot_rmle(result)
plot_rmle(result,plt_type='surface')
```

In the three dimensional application we use the regularization functional `sobolev_3d` and supply it as an argument to the `rmle()` function along with the transformation matrix and α. The results show the effect of the level of discretezation on the estimate. It is possible to achieve more accurate

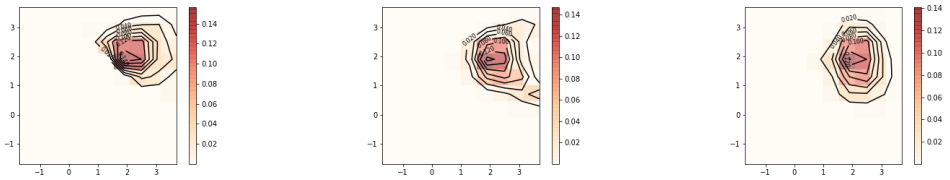

Figure 11: Contour plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

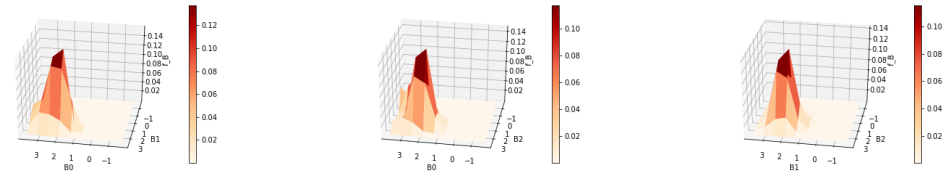

Figure 12: Surface plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

estimates with more grid points in conjunction with a narrower grid range, but with a significantly higher computational cost.

```
grid_beta_alt = grid_set(20,3,B0_range=[0,3],\
B1_range=[0,3],B2_range=[0,3])
print(grid_beta.numgridpoints())
8000
T2 = transmatrix(sample,grid_beta_alt)
result2 = rmle(sobolev_norm_penal2d,0.3,T2)
print(result2.ev())
[2.102855111361121, 2.077365765266784, 1.9697452677152947]
print(result2.mode()[0])
[[0.2008050879224736, [2.025, 2.025, 2.025]]
plot_rmle(result2)
plot_rmle(result2,plt_type='surface')
```

In this example, the number of grid points $\hat{f}_{\boldsymbol{\beta}}$ is estimated over is set to 20 on each axis which amounts to a total of 8,000 grid points. The additional level of discretezation due to the increased number of grid points and the smaller grid range provided resulted in an estimate $\hat{f}_{\boldsymbol{\beta}}$.

The next example will demonstrate the case when the underlying joint density being reconstructed has a single mode at the center of the grid. Both methods of dealing with this issue described in section 3.1 will be illustrated.

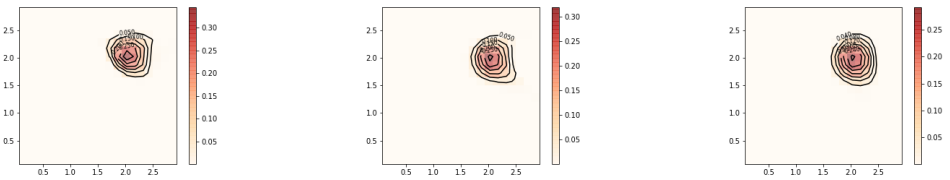

Figure 13: Contour plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

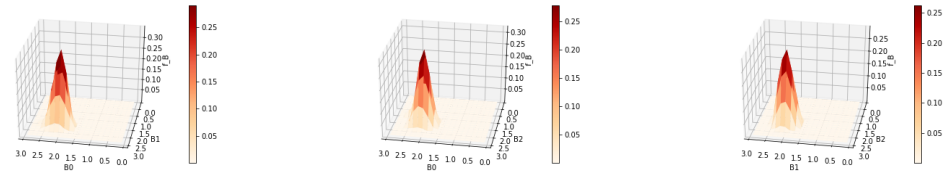

Figure 14: Surface plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

```
mu = [[0,0,0],[0,0,0],[0,0,0]]
sample = sim_sample(n = 5000,dim = 3, \
beta_mu = mu)
grid_beta = grid_set(20,3,B0_range=[-1,2],\
B1_range=[-1.5,1.5],B2_range=[-1.5,1.5])
T = transmatrix(sample,grid_beta)
result = rmle(sobolev_3d,0.15,T)
print(result.ev())
[-0.19254733276928582, 0.005554898823827626, -0.009788891289943112]
print(result.mode())
.14336637945812933, [-0.024999999999999967, 0.075, -0.075]]
plot_rmle(result)
plot_rmle(result,plt_type='surface')
```

The following example demonstrates how to apply the shifting algorithm briefly mentioned in section 3.1. The algorithm is implemented in python by adding $c \sim \mathcal{U}(a,b)$ to the intercept, where a and b are determined by the size of the grid. This applies a transformation that can be back-transformed after estimation by adjusting the grid over which $\hat{f}_{\boldsymbol{\beta}}$ is estimated over. This is repeated across ten different shifted samples, then a simple k-means clustering algorithm is applied using `sklearn.cluster.KMeans()` on the $L2$ penalties of $\{\hat{f}_{\boldsymbol{\beta}_1},\ldots,\hat{f}_{\boldsymbol{\beta}_{10}}\}$. The reconstruction, $\hat{f}_{\boldsymbol{\beta}_j}$, that is closest to the centroid of

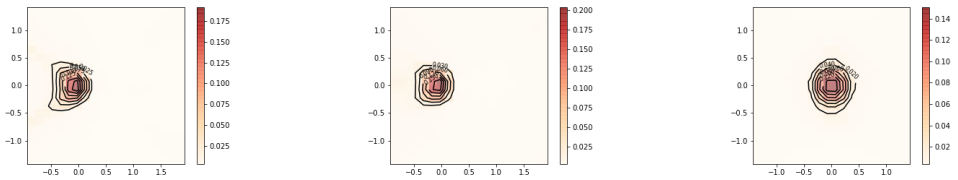

Figure 15: Contour plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

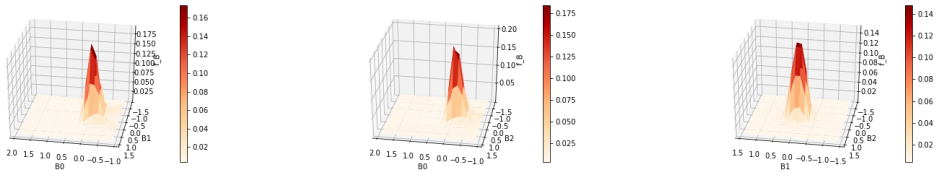

Figure 16: Surface plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

the largest cluster is then output as the solution.

```
grid_beta = grid_set(20,3,B0_range=[-1.5,1.5],\
B1_range=[-1.5,1.5],B2_range=[-1.5,1.5])
T = transmatrix(sample,grid_beta)
result_shift = rmle(sobolev_3d,0.15,T,shift=True)
print(result_shift.ev())
[-0.12974928815245057, -0.004493337754614205, -0.0052980597609372385]
print(result_shift.mode())
[[0.14624599807641833, [0.009524681155561987, -0.075, -0.075]]
plot_rmle(result)
plot_rmle(result_shift,plt_type='surface')
```

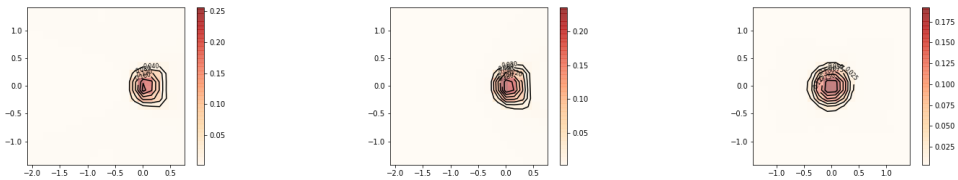

Figure 17: Contour plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

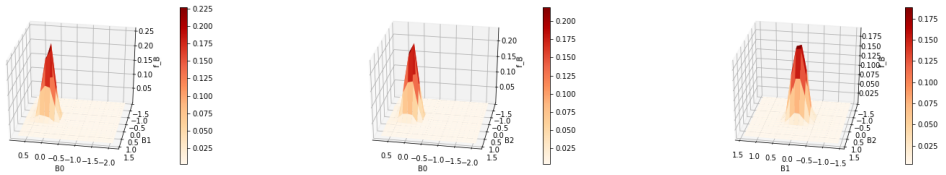

Figure 18: Surface plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

The final example demonstrates how to use the module on a real dataset, and shows the user how to load the data into Python from a comma-separated value (CSV) format and how to pre-process it, if necessary, into a usable form. The data used here are from the British Family Expenditure Surey that was used in Dunker et al. (2019, 2021).

The model is identified as follows:

$$BS_i = \beta_{0,i} + \beta_{1,i} \ln(TotalExpenditure_i) + \beta_{2,i} \ln(FoodPrices_i) + \epsilon_i$$

In the case of this dataset the regressors need to undergo a linear transformation before it generating the transformation matrix to be used in the algorithm. This is to ensure that the grid is sufficiently covered by the hyperplanes generated by the sample observations. For more details (refer to the section of our paper discussing this issue). The OLS estimate for the random coefficients suggests $f_{\boldsymbol{\beta}}$ is centralized at (0.262755,0.0048,-0.00069) for a subsample size of $n = 5000$, which requires the application of one of the two methods described above to circumvent the problem that arises from a mode close to $(0, 0, 0)$.

```
data = np.genfromtxt('filename.csv', delimiter=',')
data = data[1:-1,1:4]
data = data[np.random.randint(1,len(data),5000),:]
data = data[~np.isnan(data).any(axis=1)]
ones = np.repeat(1,len(data))
real_data_sample = np.c_[ones, \
25*data[:,2]-0.3,data[:,1]-5,data[:,0]]
```

The code block above loads the data from the a csv file, selects a random subsample of 5,000 observations, and applies the linear transformations on the data necessary.

```
grid_beta  = grid_set(20,3,B0_range=[-0.75,1.25],\
```

```
B1_range=[-1,1],B2_range=[-1,1])
T = transmatrix(real_data_sample,grid_beta)
result = rmle(sobolev_rmle_2d,0.25,T,shift=True)
print(result.ev())
[0.2139255519131858, -0.0022826617746834394, -0.10875379495199464]
print(result.mode()[0:1])
[[0.19533254824055002, [0.3, -0.05, -0.15000000000000002]]
plot_rmle(result)
plot_rmle(result,plt_type='surface')
```

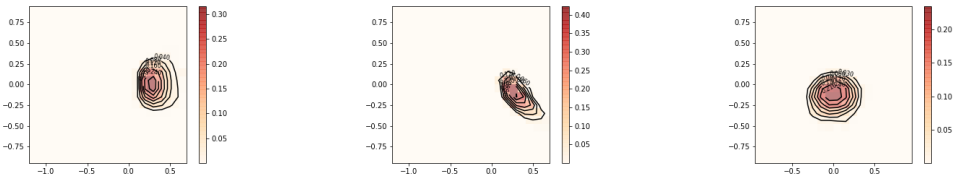

Figure 19: Contour plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

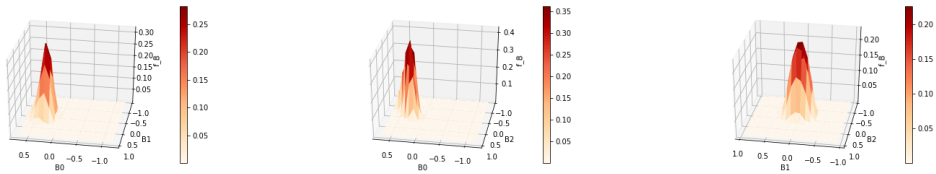

Figure 20: Surface plots of joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

Included in the module is an option to fit a spline on the estimate $\hat{f}_{\beta}$. This would be the most computationally feasible option if the user needs a finer grid. The `spline_fit()` function takes two essential arguments. The first positional argument is the **class** `RMLEResult` object, and second is the number of grid points on each axis.

```
spline = spline_fit(result,num_grid_points = 400)
plot_rmle(spline)
plot_rmle(spline,plt_type='surface')
```

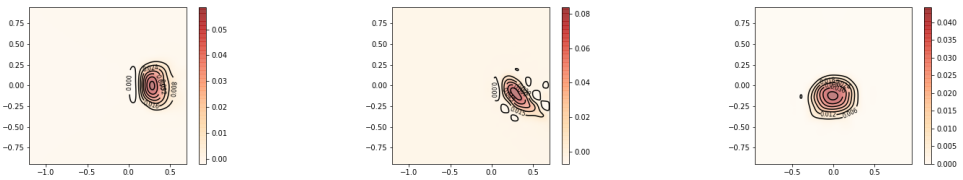

Figure 21: Contour plots of the spline interpolated joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

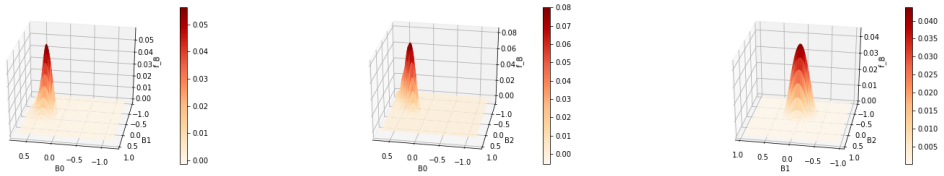

Figure 22: Surface plots of the spline interpolated joint bivariate marginal distributions of $\hat{f}_{\boldsymbol{\beta}}$: $\hat{f}_{\beta_0,\beta_1}$ (left), $\hat{f}_{\beta_0,\beta_2}$ (middle), and $\hat{f}_{\beta_0,\beta_1}$ (right).

References

Beran, R., Feuerverger, A., and Hall, P. (1996). On nonparametric estimation of intercept and slope distributions in random coefficient regression. *Ann. Statist.*, 24(6):2569–2592.

Beran, R. and Hall, P. (1992). Estimating coefficient distributions in random coefficient regressions. *Ann. Statist.*, 20(4):1970–1984.

Beylkin, G. (1987). Discrete radon transform. *IEEE transactions on acoustics, speech, and signal processing*, 35(2):162–172.

Byrd, R. H., Hribar, M. E., and Nocedal, J. (1999). An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900.

Dunker, F., Eckle, K., Proksch, K., and Schmidt-Hieber, J. (2019). Tests for qualitative features in the random coefficients model. *Electron. J. Statist.*, 13(2):2257–2306.

Dunker, F. and Hohage, T. (2014). On parameter identification in stochastic differential equations by penalized maximum likelihood. *Inverse Problems*, 30(9):095001.

Dunker, F., Mendoza, E., and Reale, M. (2021). Regularized maximum likelihood estimation for the random coefficients model.

Fox, J. T., Kim, K. I., Ryan, S. P., and Bajari, P. (2011). A simple estimator for the distribution of random coefficients. *Quantitative Economics*, 2(3):381–418.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Heiss, F., Hetzenecker, S., and Osterhaus, M. (2021). Nonparametric estimation of the random coefficients model: An elastic net approach. *Journal of Econometrics*.

Hoderlein, S., Klemelä, J., and Mammen, E. (2010). Analyzing the random coefficient model nonparametrically. *Econometric Theory*, 26(3):804–837.

Hohage, T. and Werner, F. (2013). Iteratively regularized Newton-type methods for general data misfit functionals and applications to Poisson data. *Numer. Math.*, 123(4):745–779.

Hohage, T. and Werner, F. (2016). Inverse problems with poisson data: statistical regularization theory, applications and algorithms. *Inverse Problems*, 32(9):093001.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). Scipy: Open source scientific tools for python.

Werner, F. and Hohage, T. (2012). Convergence rates in expectation for Tikhonov-type regularization of inverse problems with Poisson data. *Inverse Problems*, 28(10):104004, 15.