projSplitFit

Release 1.0

Patrick R. Johnstone and Jonathan Eckstein

Aug 20, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

`ProjSplitFit` is a Python package for solving general linear data fitting problems involving multiple regularizers and compositions with linear operators. The solver is the *projective splitting* algorithm, a highly flexible and scalable first-order solver framework. This package implements most variants of projective splitting including *backward steps* (proximal steps), various kinds of *forward steps* (gradient steps), and *block-iterative operation.* The implementation is based on `numpy`.

The basic optimization problem that this code solves is the following:

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \left\{ \frac{1}{n}\sum_{i=1}^{n} \ell(z_0 + a_i^\top Hz, r_i) + \sum_{j=1}^{n_r} \nu_j h_j(G_j z) \right\} \tag{1.1}$$

where

- $z_0 \in \mathbb{R}$ is the intercept variable (which may be optionally fixed to zero)
- $z \in \mathbb{R}^d$ is the regression parameter vector
- $\ell : \mathbb{R}\times\mathbb{R} \to \mathbb{R}_+$ is the loss
- r_i for $i = 1, \dots, n$ are the responses (or labels)
- $H \in \mathbb{R}^{d'\times d}$ is a matrix (typically the identity)
- $a_i \in \mathbb{R}^{d'}$ are the observations, forming the rows of the $n \times d'$ observation/data matrix A
- h_j for $j = 1, \dots, n_r$ are convex functions which are *regularizers*, typically nonsmooth
- G_j for $j = 1, \dots, n_r$ are matrices, typically the identity.
- ν_j are positive scalar penalty parameters that multiply the regularizer functions.

The first summation in this formulation is the *loss*, measuring how well the predictions $z_0 + a_i^\top Hz$ obtained from the dataset using the regression parameters (z_0, z) match the observed responses r_i. `ProjSplitFit` supports the following choices for the loss ℓ:

- ℓ_p^p, that is, $\ell(a, b) = \frac{1}{p}|a - b|^p$ for any $p > 1$
- logistic, that is, $\ell(a, b) = \log(1 + \exp(-ab))$
- Any user-defined convex loss.

The second summation consists of regularizers that encourage specific structural properties in the z vector, most typically some form of sparsity. `ProjSplitFit` supports the following choices for the regularizers:

- The ℓ_1 norm, that is, $\|x\|_1 = \sum_i |x_i|$
- The ℓ_2^2 squared norm, that is, $\|x\|_2^2$
- The ℓ_2 norm that is, $\|x\|_2$

- Any user-defined convex regularizer.

The package does not impose any limits on the number of regularizers present in a single problem formulation.

The linear transformations H and G_j may be any linear operators. They may be passed to `projSplitFit` as 2D `NumPy` arrays, abstract linear opertors as defined by the `scipy.sparse.linalg.LinearOperator` class, or sparse matrices deriving from the `scipy.sparse.spmatrix` class. The data matrix A may be passed in as a 2D `NumPy` array or a sparse matrix deriving from the `scipy.sparse.spmatrix` class.

1.1 Brief technical overview

The projective splitting algorithm is a primal-dual algorithm based on separating hyperplanes. A *dual solution* is a tuple of vectors $\mathbf{w} = (w_1, \ldots, w_d)$ that certify the optimality of the "primal" vector z for (1.1). At each iteration, the algorithm maintains an estimate $(z, \mathbf{w})$ of primal and dual solutions. Each iteration has two phases: first, the algorithm "processes" some of the summation terms in the problem formulation. The results of the processing step allow the algorithm to construct a hyperplane that separates the current primal-dual solution estimate from the set of optimal primal-dual pairs. The next iterate is then obtained by projecting the current solution pair estimate onto this hyperplane.

Within this overall framework, there are many alternatives for processing the various summation terms in the formulation. `ProjSplitFit` processes all the regularizer terms at every iteration, using a standard proximal step (see below for more information). For the loss terms, however, it provides considerable flexibility: the terms in the loss summation may be divided into blocks, and only a subset of these blocks need be processed at each iteration – this mode of operation is called *block iterative*. The subset of blocks processed in each iteration may be chosen at random, cyclically, or using a greedy heuristic which selects those blocks most likely to yield the best separating hyperplane. Furthermore, there are numerous options for processing each block, including approximate backward (proximal) steps and various kinds of forward steps.

Projective splitting, generally, is an *operator splitting* method that is defined for "monotone inclusion" problems. This problem class includes all convex optimization problems, but also other problems not representable as convex optimization, and which do not have objective functions. For this reason, `projSplitFit` does not need to calculate the value of the objective function in (1.1) while solving the problem. Instead, it monitors how closely the current primal and dual solutions estimates come to certifying their joint optimality. However, if you call the `getObjective` method (see below) or elect to keep a history of the solution trajectory, `projSplitFit` will attempt to compute objective function values.

CHAPTER TWO

INSTALLATION

`ProjSplitFit` depends on the standard `numpy` and `scipy` packages, and has only been tested with Python 3.7. It is not compatible with Python 2.7.

2.1 Installing from the Linux/Unix Command Line

Using Git, navigate to the directory of the desired location, type:

```
$ git clone https://github.com/1austrartsua1/projSplitFit.git
```

To use the `projSplitFit` module, make sure the project root directory is in your Python path (given by the `PYTHONPATH` environment variable on Unix and Linux systems). Alternatively, run Python from the project root directory.

2.2 Installing Directly into Pycharm

If you wish to use `projSplitFit` from within PyCharm, you should be able to use Pycharm's VCS (Version Control System) integration.

Click VCS->enable VCS. Then click VCS->Clone and enter the URL https://github.com/1austrartsua1/projSplitFit.git.

2.3 Running the Tests

You may verify that `projSplitFit` is correctly installed and operating by running its test suite, located in the `tests` subdirectory. To run these tests, you need to have the pytest module installed (in addition to `numpy` and `scipy`). To initiate the tests from the command line, descend into the `tests` subdirectory and enter:

```
$ pytest
```

This command will run all the tests. On systems in which the `python` command defaults to Python 2.7 and later versions of Python use the `python3` command, instead enter the command:

```
$ python3 -m pytest
```

Depending on your CPU speed, it may take 5 to 10 minutes to run all the tests.

Specific tests can be run by specifying an individual test file. For example:

```
$ pytest test_multiple_norms.py
```

will only run the tests in the file `test_multiple_norms.py`. To accomplish the same thing on systems defaulting to Python 2.7, you would instead enter:

```
$ python3 -m pytest test_multiple_norms.py
```

To run tests from within PyCharm, issue `pytest` commands as above within PyCharm's Python Console tool pane, from the tests folder.

Most of the tests operate by running the algorithm on an optimization problem and checking that `projSplitFit` finds the optimal value of this problem to some desired accuracy. The optimal values are stored in the `tests/results` subdirectory that is downloaded with the distribution.

If you wish, you may refresh these optimal values by creating new random optimization problems with randomly drawn data. Code at top of each test file creates a boolean variable called `getNewOptVals`, set to `False`. If you change this assignment to `True`, the tests will create new optimization problems with randomly drawn data, and store their optimal values in the `tests/results` subdirectory. In order to use this feature, however, you must have the cvxpy package installed, since the target optimal values are computed with `cvxpy`. Using this feature will also slow down the testing process.

CHAPTER THREE

TUTORIAL

Solving a problem with `projSplitFit` requires the following fundamental steps:

1. Create an empty object from the `ProjSplitFit` class
2. Add data to set up the object's data/loss term
3. Add regularizers to the object
4. Run the algorithm to solve the optimization problem
5. Retrieve the solution and/or optimal value.

This chapter gives simple examples of each of these operations. Full descriptions of the methods used are in the following chapter. Complete running programs using the operations to solve problems may be found in the `examples` subdirectory of the package. Most of these examples solve synthetic, randomly generated problems, but one solves the (very difficult) problem posed in [YB18].

Note that `projSplitFit` with a lower-case initial 'p' denotes the name of the `projSplitFit` Python package, whereas `ProjSplitFit` with an upper-case initial 'P' denotes the primary class defined in that package.

3.1 Basic Setup with a Quadratic Loss Term

Assume that the matrix A is a 2D `NumPy` array whose rows are the observations of some dataset and y is a list or 1D `NumPy` array containing the corresponding response values. Consider the classical least-squares problem defined as

$$\min_{z\in\mathbb{R}^d} \frac{1}{2n}\|Az-y\|_2^2 \tag{3.1}$$

to solve this problem with `projSplitFit`, one would use the following code

```
import projSplitFit as ps
projSplit = ps.ProjSplitFit()
projSplit.addData(A,y,loss=2,intercept=False)
projSplit.run()
optimalVal = projSplit.getObjective()
z = projSplit.getSolution()
```

The first line after the `import` statement calls the contructor to set up an empty `ProjSplitFit` object. Next, the invocation of the `addData` method provides the object with the model data and defines the loss term.

In the `addData` call, the argument `loss` is set to 2 in order to use the ℓ_2^2 loss. Other possible choices are any $p > 1$ for the ℓ_p^p loss and the string "logistic" for the logistic loss. The user may also define their own loss via the `losses.LossPlugIn` class (see below). The `intercept=False` argument specifies that the model does not have an intercept (constant) term.

We assumed A was a 2D `NumPy` array. However, `ProjSplitFit` also supports *sparse* data matrices of a class derived from `scipy.sparse.spmatrix`. See here for documentation on sparse matrices in `scipy`.

This classical model has no regularizers, so it is not necessary to add regularizers. The `run` method then solves the optimization problem. After solving the problem, the `getObjective` method returns the optimal solution value and the `getSolution` value returns the solution vector z.

3.2 Dual Scaling

The dual scaling parameter, called γ in most projective splitting papers, plays an important role in the empirical convergence rate of the method. It must be selected carefully. There are two ways to set γ. It may be set when calling the `projSplitfit` constructor, as in:

```
projSplit = ps.ProjSplitFit(dualScaling=gamma)
```

The default value is 1. The parameter may also be modified later through the `setDualScaling` method:

```
projSplit.setDualScaling(gamma)
```

Tuning this parameter is currently of paramount importance in the practical performance of the algorithm. In future, we hope to provide automated tools for tuning γ.

3.3 Including an Intercept Variable

It is common in machine learning to fit an intercept for a linear model. That is, instead of solving (3.1) solve

$$\min_{z_0\in\mathbb{R},z\in\mathbb{R}^d} \frac{1}{2n}\|z_0 e + Az - y\|^2$$

where e is a vector of all ones of the same length as y. To do this, set the `intercept` argument to the `addData` method to `True` (which is the default). Note that regularizers never apply to the intercept variable.

3.4 Normalization

The performance of first-order methods is effected by the scaling of the features. A common tactic to improve performance is to scale the features so that they have commensurate size. This is controlled by setting the `normalize` argument of `addData` to `True` (which is the default). If this is done, then the observations matrix A is copied and the columns of the copy are normalized to have ℓ_2 norm equal to $\sqrt{n}$ where n is the number of rows of A.

3.5 Adding a Regularizer

A common strategy in machine learning is to add a regularizer to the model. Consider the lasso

$$\min_{z\in\mathbb{R}^d} \frac{1}{2n}\|Az - y\|^2 + \lambda_1\|z\|_1, \tag{3.2}$$

where $\|z\|_1 = \sum_i |z_i|$. To solve this model instead, we call the `addRegularizer` method of the `ProjSplitFit` object before invoking `run()`:

```
from regularizers import L1
regObj = L1(scaling=lam1)
projSplit.addRegularizer(regObj)
```

The built-in method `L1` returns an object derived from the class `regularizers.Regularizer` The `regularizers.Regularizer` class may be used to describe any convex function to be used as a regularizer. Other built-in regularizers include `regularizers.L2sq`, which creates the regularizer $0.5\|x\|_2^2$, and `regularizers.L2`, which creates the regularizer $\|x\|_2$. A group L2 regularizer is also available.

To recap, the entire code to solve (3.2) with $\lambda_1 = 0.1$ and the default dual scaling of $\gamma = 1$ is

```
import projSplitFit as ps
from regularizers import L1
lam1 = 0.1
projSplit = ps.ProjSplitFit()
projSplit.addData(A,y,loss=2,intercept=False,normalize=False)
regObj = L1(scaling=lam1)
projSplit.addRegularizer(regObj)
projSplit.run()
optimalVal = projSplit.getObjective()
z = projSplit.getSolution()
```

If an intercept variable is desired, the keyword argument `intercept` should be set to `True` or omitted.

A complete example program solving both the LASSO problem and simple least-squares regression problem mentioned above may be found in `examples/LeastSquaresAndLASSO.py`.

3.6 User-Defined and Multiple Regularizers

In addition to these built-in regularizers, the user may define their own. In `projSplitFit`, a regularizer is defined by a `prox` method and a `value` method. The `prox` method must be defined. The `value` method is optional and is only used if the user specifies calculation of function values for performance tracking, or uses the `getObjective` method. The `prox` method returns the proximal operator of σf, where f is the regularizer function and σ is a positive scaling factor. That is, the `prox` method should be defined so that

$$f.\texttt{prox}(t,\sigma) = \text{prox}_{\sigma f}(t) = \arg\min_x \left\{ \sigma f(x) + \frac{1}{2}\|x-t\|_2^2 \right\}. \tag{3.3}$$

The `prox` method should expect its first argument to be a 1D `numpy` array and its second argument to be a positive `float`; it should return a `numpy` array of the same dimensions as the first argument.

The `value` method f`.value(`x`)`, if defined, should simply returns the function value $f(x)$; it should expect its argument to be a 1D `numpy` array and return a `float`.

Using multiple regularizers in `projSplitFit` is straightforward: one simply calls `addRegularizer` multiple times before calling `run`. Suppose one wants to solve the lasso with an additional constraint that each component of the solution must be nonnegative. That is, one wishes to solve

$$\min_{z\in\mathbb{R}^d, z\geq 0} \frac{1}{2n}\|Az-y\|^2 + \lambda_1\|z\|_1. \tag{3.4}$$

One possible approach to solving this problem is to formulate the nonnegativity constraint as a second regularizer. That is, one may rewrite (3.4) as

$$\min_{z\in\mathbb{R}^d} \frac{1}{2n}\|Az-y\|^2 + \lambda_1\|z\|_1 + g(z),$$

where

$$g(z) = \begin{cases} +\infty & \text{if } z_i < 0 \text{ for any } i \\ 0 & \text{otherwise.} \end{cases}$$

The proximal operator (3.3) for this function is simply projection onto the nonnegative orthant, and is independent of σ. To include this regularizer in `projSplitFit` object, one defines the regularizer object for g and then adds it to the model with `addRegularizer`. These operations may be accomplished as follows:

```
from regularizers import Regularizer
def prox_g(z,sigma):
  return (z>=0)*z
def value_g(x):
  if any(x < 0):
     return float('Inf')
  return 0.0
regObjNonneg = Regularizer(prox=prox_g, value=value_g)
projSplit.addRegularizer(regObjNonneg)
```

Note that `prox` function must still have a second argument `sigma` even in cases, like this one, where the returned value is independent of `sigma`.

In summary, the entire code to solve (3.4) with (for example) $\lambda_1 = 0.1$ and the default dual scaling of $\gamma = 1$ would be

```
import projSplitFit as ps
from regularizers import L1, Regularizer

def prox_g(z,sigma):
  return (z>=0)*z

def value_g(x):
  if any(x < -1e-7):
     return float('Inf')
  return 0.0

lam1 = 0.1

projSplit = ps.ProjSplitFit()
projSplit.addData(A,y,loss=2,intercept=False,normalize=False)
regObj = L1(scaling=lam1)
projSplit.addRegularizer(regObj)
regObjNonneg = Regularizer(prox=prox_g, value=value_g)
projSplit.addRegularizer(regObjNonneg)
projSplit.run()
optimalVal = projSplit.getObjective()
z = projSplit.getSolution()
```

Here, for numerical reasons, we have slightly modified the `value_g` function to treat very small-magnitude negative numbers as if they were zero. A complete example program creating a customized regularizer may be found in `examples/UserDefinedRegularizer.py`.

Note that we present the code above mainly for purposes of example. A potentially more efficient approach to solving the nonnegative lasso problem would be use a single user-defined regularizer of the form

$$h(x) = \begin{cases} x, & \text{if } x \geq 0 \\ +\infty, & \text{otherwise.} \end{cases}$$

This regularizer imposes both ℓ_1 regularization and the nonnegativity constraint, while having a proximal operation that is still easily evaluated.

3.7 Linear Operator Composed with a Regularizer

Sometimes, one would like to compose a regularizer with a linear operator. Total variation deblurring is an example of such a situation. `ProjSplitFit` handles this with ease. Consider the problem

$$\min_{z\in\mathbb{R}^d} \frac{1}{2n}\|Az-y\|^2 + \lambda_1\|Gz\|_1$$

for some linear operator or matrix G. The linear operator can be added as an argument to the `addRegularizer` method as follows, assuming the matrix variable `G` has been defined:

```
regObj = L1(scaling=lam1)
projSplit.addRegularizer(regObj,linearOp=G)
```

G must be a 2D `numpy` array, a `scipy` linear operator, or a `scipy` sparse matrix. If G is an array, the number of columns of G must equal the dimension of the solution vector z.

Documentation for `scipy` linear operators may be found in the package `scipy.sparse.linalg`. When used with `projSplitFit`, such operators should have a `shape` (m, n) and define the methods `matvec` and `rmatvec`, which respectively compute the actions of the linear operator and its adjoint (the equivalent of multiplication by the matrix transpose). Consider the 1D total variation operator $\mathbb{R}^n \to \mathbb{R}^{n-1}$ given by

$$[x_1 \quad x_2 \quad \cdots \quad x_n] \quad \mapsto \quad [x_1 - x_2 \quad x_2 - x_3 \quad \cdots \quad x_{n-1} - x_n].$$

This map is equivalent to the action of $n-1 \times n$ matrix

$$V = \begin{bmatrix} 1 & -1 & & & & \\ & 1 & -1 & & & \\ & & 1 & -1 & & \\ & & & \ddots & \ddots & \\ & & & & 1 & -1 \end{bmatrix}.$$

The adjoint of this operator is the map, equivalent to multiplication by the transpose $V^\top$ of V, is therefore

$$[u_1 \quad u_2 \quad \cdots \quad u_{n-1}] \quad \mapsto \quad [u_1 \quad u_2 - u_1 \quad u_3 - u_2 \quad \cdots \quad u_{n-1} - u_{n-2} \quad -u_{n-1}].$$

Calling `varop1d(n)` as defined in the code below will create such an operator:

```
import numpy
import scipy

def applyOperator(x):
   return x[:(len(x)-1)] - x[1:]

def applyAdjoint(u):
   return numpy.pad(u,(0,1)) - numpy.pad(u,(1,0))

def varop1d(n):
   return scipy.sparse.linalg.LinearOperator(shape=(n-1,n),
                                             matvec=applyOperator,
                                             rmatvec=applyAdjoint)
```

A compete example program using the L1 regularizer composed with the above customized linear operator may be found in `examples/LinearOpComposedWithReg.py`.

3.8 User-Defined Losses

Just as you may define your own regularizers, you may define your own loss function, using the class `losses.LossPlugIn`. Objects of this class can be passed into `addData` as the `loss` argument. To define a loss, you need to define its `derivative` method. Optionally, you may also define its `value` method if you would like to compute function values (either for performance tracking or to call the `getObjective` method).

For example, consider the one-sided ℓ_2^2 loss:

$$\ell(x,y) = \begin{cases} 0 & \text{if } x \le y \\ \frac{1}{2}(x-y)^2 & \text{otherwise.} \end{cases}$$

To use this loss, you would proceed as follows:

```python
import losses as ls

def deriv(x,y):
  return (x>=y)*(x-y)
def val(x,y):
  return (x>=y)*(x-y)**2

loss = ls.LossPlugIn(derivative=deriv, value=val)
projSplit.addData(A,y,loss=loss)
```

A complete example program employing this user-defined loss function may be found in `examples/UserDefinedLoss.py`.

3.9 A More Complicated Example: Rare Feature Selection

We now consider a more complicated, complete example, the rare feature selection problem in the paper [JE19a]. The basic form of this problem originated with [YB18], and involves predicting TripAdvisor ratings of hotels from the adjectives present in the corresponding review text. The experiments in [JE19a] involve predicting whether or not the rating is a "5" (the highest rating), and use the logistic loss function. With the large dataset described in [YB18], this problem is very difficult to solve, and the experiments in [JE19a] suggest that projective splitting is the best available method for obtaining solutions of reasonable quality.

The problem takes the form (substituting v for γ as the decision variables)

$$\min_{\substack{v_0\in\mathbb{R}\\ v\in\mathbb{R}^d}} \left\{ \frac{1}{n}\ell(v_0 e + XHv, r) + \lambda\big(\mu\|v_{-r}\|_1 + (1-\mu)\|Hv\|_1\big) \right\},$$

where ℓ is the logistic loss function. In this formulation, the regression coefficients are composed with a linear operator H. There are two ways to deal with such situations: first, if the size and density of the matrices is not of great concern concern, one may pre-compute a new matrix through `Xnew = X*H`, and use `Xnew` as the observation matrix passed to `projSplitFit`. Second, if forming XH directly in this manner is somehow prohibitive or causes an unacceptable increase in the number of nonzero matrix elements, the linear operator can be instead composed with the loss, meaning that `projSplitFit` handles the composition internally and does not explicitly compute the matrix product. This option is controlled via the `linearOp` argument to `addData`.

Taking the second approach and electing not to normalize the input data, one may set up the loss term as follows:

```python
import projSplitFit as ps
projSplit = ps.ProjSplitFit()
projSplit.addData(X,y,loss=2,linearOp=H,normalize=False)
```

Note that, by default, the intercept term v_0 is incorporated into the loss. The complete code for the example, including reading input data from files, maybe found in `examples/RareFeatureSelection.py`. The example is configured to solve the instance with $\lambda = 10^{-4}$ and $\mu = 0.5$, and the data files are in the directory `examples/data`.

We now consider the two regularization terms. In the first regularization term, the notation v_{-r}, as introduced in [YB18], specifies that the regularizer applies to all but the last coefficient in v, which corresponds to the root node of the adjective tree described by the matrix H. A simple way to encode this regularization term is to treat it as the ℓ_1 norm composed with a linear operator which simply drops the last entry of a vector. That is, we write the regularizer as $\|Gv\|_1$, where

$$G : [v_1 \;\; v_2 \;\; \cdots \;\; v_{d-1} \;\; v_d] \mapsto [v_1 \;\; v_2 \;\; \cdots \;\; v_{d-1}].$$

Writing G as a matrix, we have

$$G = \begin{bmatrix} 1 & & & & 0 \\ & 1 & & & 0 \\ & & \ddots & & \vdots \\ & & & 1 & 0 \end{bmatrix} \quad \text{and therefore} \quad G^\top = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ 0 & 0 & \dots & 0 \end{bmatrix}.$$

We may create such a linear operator using the `scipy.sparse.linalg.LinearOperator` class and incorporate it into the regularizer as follows:

```
from scipy.sparse.linalg import LinearOperator
import numpy as np
import regularizers

def applyG(x):
  return x[:-1]

def applyGtranspose(v):
  return np.append(v,0.0)

(_,nv) = H.shape
shape = (nv-1,nv)
G = LinearOperator(shape,matvec=applyG,rmatvec=applyGtranspose)
projSplit.addRegularizer(regularizers.L1(scaling=mu*lam),linearOp=G)
```

The second regularizer is more straightforward and may be dealt with via the built-in `L1` function and composing with the linear operator H as follows:

```
regObj2 = regularizers.L1(scaling=lam*(1-mu))
projSplit.addRegularizer(regObj2,linearOp=H)
```

We set the dual scaling factor γ to 0.0001:

```
projSplit.setDualScaling(1e-4)
```

Finally we are ready to run the method with:

```
projSplit.run(nblocks=10, maxIterations=20000, verbose=True, keepHistory=True)
```

The limit of 20,000 iterations is sufficient to reproduce the results in [JE19a], reaching an objective level of approximately 0.525. The `keepHistory=True` option records the trajectory of the run.

One can obtain the final objective value and solution via:

```
objVal = projSplit.getObjective()
solVector = projSplit.getSolution()
```

The `projSplitFit` package currently only uses parallelism to the degree that `numpy` uses parallelism on your computer configuration. Other applications of parallelism hold the potential to improve performance, but are not addressed in this software at present.

3.10 Loss Processor Objects

Projective splitting offers numerous choices as to how to process the various operators making up a problem — in the current setting, "operators" corresponding to various elements in the summation in (1.1) — so as to construct a separating hyperplane. In the original papers [ES08][ES09], all operators were processed with some form of proximal step, that is, essentially the calculation (3.3) or some approximation thereof. Such calculations are also called *backward steps*. This feature persisted in later work such as [ACS14][CE18]. More recently, however, new ways of processing operators have been devised, based on *forward steps*, that is, simple gradient calculations [JE19a], [JE19b]. These innovations make projective splitting into a true first-order method.

`ProjSplitFit` assumes that all regularizers employed have a computationally efficient proximal operation. It invokes the proximal operation of every regularizer at every iteration. For the loss function terms, however, `projSplitFit` affords a large number of options. First, it permits the loss function to be divided into an arbitrary number of blocks, each containing the same number of observations (give or take one observation). You may determine how many of these blocks to process at each iteration, and among several rules to select blocks for processing. Second, it provides eight different options for processing each block.

The number of loss blocks and their activation scheme are controlled by keyword arguments to the `run` method, as described in Section 3.11 below. The procedure used to process each block is determined by the optional `process` argument to the `addData` method. This argument must be an object whose class is derived from `lossProcessors.LossProcessor`. The file `lossProcessors.py` pre-defines the following eight classes that may be used for this purpose :

- `Forward2Fixed`: two-forward-step update with fixed stepsize, see [JE19a]
- `Forward2Backtrack`: two-forward-step update with backtracking stepsize, see [JE19a]. This is the default loss processor if the `process` argument is ommitted from `addData`
- `Forward2Affine`: a specialized two-forward-step update for quadratic loss functions, automatically selecting a valid stepsize without backtracking, see [JE19a]. Only available when `loss=2`
- `Forward1Fixed`: one-forward-step update with fixed stepsize, see [JE19b]
- `Forward1Backtrack`: one-forward-step update with backtracking stepsize, see [JE19b]
- `BackwardExact`: Exact proximal/backward step for ℓ_2^2 loss via matrix factoring. Only available with `loss=2`
- `BackwardCG`: approximate proximal/backward step computed by a conjugate gradient method, only available when `loss=2`
- `BackwardLBFGS`: approximate backward/proximal step computed by a limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) solver.

To select a loss processor, you call the constructor of the desired class with any desired parameters, and then pass the resulting object into `addData` as the `process` argument. For example, to use `BackwardLBFGS` with its default parameters on the $\ell_{1.5}^{1.5}$ loss, you would use the code fragment

```
import lossProcessors as lp
processObj = lp.BackwardLBFGS()
projSplit.addData(A,y, loss=1.5, process=processObj)
```

See the detailed documentation section below for a complete listing of the parameters for each loss processing class.

It is possible to create your own loss processing classes, although guaranteeing convergence may requires significant mathematical analysis. Please contact the authors for more information on extending `projSplitFit` in this manner.

3.11 Blocks of Observations

The `run` method of class `ProjSplitFit` has three important options which control the division of the loss function into blocks, and how these blocks are processed at each iteration. The first is `nblocks`. This controls how many blocks projective splitting breaks the loss into for processing. Recall the loss is

$$\frac{1}{n}\sum_{i=1}^{n}\ell(z_0 + a_i^\top Hz, r_i)$$

An important property of projective splitting is *block iterativeness*: the method does not need to process every observation at each iteration. Instead, it may break the n observations into `nblocks` blocks and process as few as one block at a time. `nblocks` may be any integer ranging from `1`, meaning all observations are processed at each iteration, up to `n`, meaning every individual observation is treated as a block. `nblocks` currently defaults to 1, but better performance is often observed for larger values.

At present, blocks may only be contiguous spans of observation indices. Suppose that `nblocks` is set to some value b. If n is divisible by b, then each block simply contains n/b contiguous indices. If b does not divide the number of observations, then the first $n \bmod b$ blocks have $\lceil n/b \rceil$ observations and the remaining blocks have $\lfloor n/b \rfloor$ observations.

The number of blocks processed per iteration is controlled via the argument `blocksPerIteration`, which defaults to 1. It can take any integer value between 1 and `nblocks`.

There are three ways to choose *which* blocks are processed at each iteration. The selection of blocks is controlled with the `blockActivation` argument, which may be set to

- `'random'`: select blocks at random, with equal probabilities
- `'cyclic'`: cycle through the blocks in a round-robin manner
- `'greedy'` (the default): use the "greedy" heuristic of [JE19a], page 24 to select blocks. This heuristic estimates which blocks are most important to process to make progress toward the optimal solution.

For example, to use 10 blocks and evaluate one block per iteration using a greedy selection scheme, one would run the optimization by (assuming that `projSplit` is a `projSplitFit` object)

```
projSplit.run(nblocks=10, blockActivation='greedy', blocksPerIteration=1)
```

However, greedy activation and one block per iteration being the defaults, the above could be shortened to

```
projSplit.run(nblocks=10)
```

For some problem classes, it has been empirically been observed that processing one or two blocks per iteration, selected in this greedy manner, yields similar convergence to processing the entire loss term, but with much lower time required per iteration.

3.12 Embedding Regularizers

Projective splitting handles regularizers through their proximal operations (3.3). Regularizers added to a ProjSplitFit object are processed at every iteration. Such regularizers cause projSplitFit to allocate three internal vector variables whose dimension matches the regularizer argument.

However, the "forward" loss processors also have the option to "embed" a single regularizer into each loss block; please see Section 3.11 above for a discussion of dividing the loss function into blocks. Each time a loss block is processed, the loss processor also performs a backward (proximal) step on the embedded regularizer, and no additional working memory needs to allocated to the regularizer.

The embedding feature is controlled by the embed keyword argument of the addData method. To solve a standard lasso problem with this technique, using 10 loss blocks, one would proceed as follows:

```
import projSplitFit as ps
from regularizers import L1
lam1 = 0.1
projSplit = ps.ProjSplitFit()
regObj = L1(scaling=lam1)
projSplit.addData(A, y, loss=2, intercept=False, normalize=False, embed=regObj)
projSplit.run(nblocks=10)
optimalVal = projSplit.getObjective()
z = projSplit.getSolution()
```

Note that when a regularizer is embedded in the loss function, it should not also be added to the problem with addRegularizer. But only one regularizer can be embedded in the loss term; if further regularizers are needed, then those should be introduced into the problem with addRegularizer. If the loss term also contains a linear operator, that linear operator applies to both the loss term and regularizer.

The embedded regularizer and the loss processor must use the same stepsize. If they are different, a warning is printed and the stepsize for the regularizer is set to be the stepsize of the loss processor. For backtracking loss processors which modify the stepsize as the algorithm runs, the embedded regularizer's stepsize will be automatically set to the correct stepsize before it's prox operator is applied.

The embed feature cannot be used with the backward loss processors nor with Forward2Affine.

3.13 Other Features

The getObjective() method of the ProjSplitFit class simply returns the objective value at the current primal iterate.

The keepHistory and historyFreq arguments to run() allow you to record the progress of the algorithm in terms of objective function values, running time, primal and dual residuals, and hyperplane values. These may be extracted later via the getHistory() method. Set keepHistory=True to record history information. The historyFreq parameter controls how often information is recorded: for example, setting historyFreq=1 causes the information to be recorded every iteration, while setting historyFreq=10 causes it to be recorded once every ten iterations.

The code at the end of examples/RareFeatureSelection.py shows how to use the data structure returned by getHistory to plot the progress of the objective function over the course of the run. This data structure is described in detail in the next section of this document.

If you use either the keepHistory feature or the getObjective function in conjunction with a user-defined loss function, then that loss function must have a value method. Similarly, using either the keepHistory feature or the getObjective function in conjunction with a user-defined regularizer requires that the regularizer have value method.

After using `run()`, the `getSolution()` method of the `ProjSplitFit` class returns the primal iterate z^k. If its `descale` argument is set to `True`, then the scaling vector used to scale each column of the data matrix is applied to the elements of z^k, so that the returned vector of coefficients is in the coordinate system of the original data. Thus, the returned coefficient vector may be directly used to make predictions using unnormalized data, such as new test data. The `descale` option is not available when the loss term is composed with a linear operator.

The `ProjSplitFit` method `getScaling()` returns the scaling vector used in normalization. This scaling vector can then be applied to normalize new test data. For example, to normalize a new test datapoint `xtest`, one could write:

```
scaling = projSplit.getScaling()
x_test_normalized = xtest/scaling
```

If the model was formulated with an intercept term, then the intercept term is the first element of the vector returned by `getSolution`.

CHAPTER

FOUR

DETAILED DOCUMENTATION

4.1 ProjSplitFit Class

class projSplitFit.**ProjSplitFit**(*dualScaling=1.0*)

ProjSplitFit is the class used for creating a data-fitting problem and solving it with projective splitting.

Please refer to

- [JE19a], arxiv.org/abs/1803.07043 (algorithm definition page 9)
- [JE19b], arxiv.org/abs/1902.09025 (algorithm definition pages 10-11)

To create an object, call:

```
psobj = ProjSplitFit(dualScaling)
```

`dualScaling` (which defaults to 1.0) is γ in the algorithm definitions from the above papers.

The general optimization objective this can solve is

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\ell(z_0 + a_i^\top H z, r_i) + \sum_{j=1}^{n_r}\nu_j h_j(G_j z)$$

where

- $z_0 \in \mathbb{R}$ is the intercept variable
- $z \in \mathbb{R}^d$ is the parameter vector
- $\ell : \mathbb{R}\times\mathbb{R} \to \mathbb{R}_+$ is the loss
- r_i for $i = 1, \ldots, n$ are the responses (or labels)
- $H \in \mathbb{R}^{d'\times d}$ is a matrix (typically the identity)
- $a_i \in \mathbb{R}^{d'}$ are the observations, forming the rows of the $n \times d'$ observation/data matrix A
- h_j for $j = 1, \ldots, n_r$ are convex functions which are *regularizers*, typically nonsmooth
- G_j for $j = 1, \ldots, n_r$ are matrices, typically the identity.
- ν_j are positive scalar penalty parameters that multiply the regularizer functions.

The data A and y are introduced via the `addData` method.

Regularizers are introduced through the `addRegularizer` method.

The `run` method solves the problem.

__init__(*dualScaling=1.0*)

Parameters **`dualScaling`** (`float`, optional) – the primal-dual scaling parameter which is γ in [JE19a] (algorithm definition on page 9) and [JE19b] (algorithm definition on pages 10-11). `dualScaling` must be positive, and defaults to 1.0.

`addData` (*observations*, *responses*, *loss*, *process=<lossProcessors.Forward2Backtrack object>*, *intercept=True*, *normalize=True*, *linearOp=None*, *embed=None*)

Introduces the data for the fitting model, and configures the loss function.

Recall that the general optimization objective solved by this package is

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\ell(z_0 + a_i^\top Hz, r_i) + \sum_{j=1}^{n_r}\nu_j h_j(G_j z)$$

Parameters

- **`observations`** (2d `numpy.ndarray` or `scipy.sparse.spmatrix`) – A 2D numpy array or scipy sparse matrix. The rows of this matrix are the vectors a_i above. All `scipy.sparse.spmatrix` subclasses are supported. Internally, the matrix is converted to `scipy.sparse.csr_matrix` format, since this format is the most convenient for the row slicing and arithmetic operations required by the solution algorithm.
- **`responses`** (1d `numpy.ndarray` or `list`) – the elements within this object comprise the response values r_i above. The number of elements should equal the number of rows in `observations`.
- **`loss`** (`float` or `string` or *`losses.LossPlugIn`*) – Specifies the loss function ℓ. May be a `float` $p > 1$ to indicate the ℓ_p^p loss, the `string` 'logistic' to specify the logistic loss, function, or an object of class *`losses.LossPlugIn`*.
- **`process`** (`lossProcessors.LossProcessor`, optional) – An object of a class derived from `lossProcessors.LossProcessor`. Default is `Forward2Backtrack()`
- **`intercept`** (`bool`, optional) – whether to include an intercept/constant term in the linear model. The default value is `True`.
- **`normalize`** (`bool`, optional) – whether to normalize columns of the data matrix to have square norm equal to num rows. If True, data matrix will be copied. Default is True.
- **`linearOp`** (`scipy.sparse.linalg.LinearOperator` or 2D `numpy.ndarray` or 2D `scipy.sparse.spmatrix`, optional) – Introduces the matrix H in the above problem formulation. Defaults to the identity. If this argument is a sparse matrix, it will be converted to `scipy.sparse.csr_matrix` format, as this format is the most convenient for the arithmetic operations required in the solution algorithm.
- **`embed`** (*`regularizers.Regularizer`*, optional) – Embeds a regularizer into the loss, meaning that the proximal operator is evaluated in-line with the loss processing update. Only available for the following forward-type loss processors: `Forward1Fixed`, `Forward1Backtrack`, `Forward2Fixed`, `Forward2Backtrack`. If embed is used with any other loss processor, a warning is printed and the regularizer is added as an ordinary regularizer instead.

`addRegularizer` (*regObj*, *linearOp=None*)

Introduces a regularizer term into the optimization problem.

Recall the optimization problem

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\ell(z_0 + a_i^\top Hz, r_i) + \sum_{j=1}^{n_r}\nu_j h_j(G_j z)$$

This method adds each h_j, ν_j, and G_j above

Parameters

- **regObj** (*regularizers.Regularizer*) – object of class *regularizers.Regularizer*
- **linearOp** (`scipy.sparse.linalg.LinearOperator` or 2D `numpy.ndarray` or 2D `scipy.sparse.spmatrix`, optional) – Introduces the matrix G_j above, which otherwise defaults to an identity matrix. If a sparse matrix is supplied, it is internally converted to the `scipy.sparse.csr_matrix` format.

getDualScaling()

Returns the current setting of `dualScaling`

Returns the `dualScaling` parameter

Return type `float`

getDualViolation()

Returns the current dual violation. A solution is exactly optimal if both its primal and dual violation are zero.

After at least one call to the method run(), returns a float equal to the dual violation.

Recall the objective

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\ell(z_0 + a_i^\top Hz, r_i) + \sum_{j=1}^{n_r}\nu_j h_j(G_j z)$$

In the notation of [JE19a], dual violation is

$$\max\left\{\max_{i=1,..,n_b}\|y_i^k - w_i^k\|_2, \max_{j=1,..,n_r}\|y_{j+n_b} - w_j^k\|_2\right\}$$

where, n_b is the number of blocks in the loss (controlled by `nblocks` argument to `run`).

If run has not been called yet, raises an exception.

Returns **dualErr** – Dual Violation.

Return type `float`

getHistory()

Returns array of history data from most recent invocation of `run` for which the `keepHistory` was set to `True`.

After at least one call to run with keepHistory set to `True`, the function call:

```
historyArray = psfObj.getHistory()
```

returns a two-dimensional, five-row NumPy array with each column corresponding to an iteration for which the history statistics were recorded. The total number of columns is the number of iterations divided by the `historyFreq` parameter, which can be set as an argument to `run` and defaults to 10. In each row of this array, the rows have the following interpretation:

0. Objective value
1. Cumulative run time
2. Primal violation
3. Dual violation
4. Value of the $\phi(p^k)$ quantity used in hyperplane construction

If `run` has not yet been called with `keepHistory` set to True, this function will raise an Exception when called.

If `keepHistory` is set to True and a regularizer or the loss is added without implementing its value method, an exception will be raised.

Returns **historyArray** – ndarray with 5 rows.

Return type ndarray

`getObjective`(*ergodic=False*)

Returns the current objective value evaluated at the current primal iterate z^k. If the method has not been run yet, raises an exception.

If a loss or regularizer was added without defining its value method, calling `getObjective` raises an exception.

Parameters **`ergodic`** (`bool` or `string`, optional) – Whether to compute objective at the primal iterate z^k, or one of its two averaged versions. If `False` (the default), uses the primal iterate. If "simple", evaluate at $\frac{1}{k}\sum_{t=1}^{k} z^t$; if "weighted", evaluate at

$$\frac{\sum_{t=1}^{k}\tau_t z^t}{\sum_{t=1}^{k}\tau_t}$$

where the τ_t are the stepsizes used in the hyperplane projections.

Returns **currentLoss** – the current objective value evaluated at the current iterate

Return type `float`

`getPrimalViolation`()

Returns the current primal violation. A solution is exactly optimal if both its primal and dual violation are zero.

After at least one call to the method `run`, this method returns a `float` equal to the primal violation.

Recall the objective

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\ell(z_0 + a_i^\top Hz, r_i) + \sum_{j=1}^{n_r}\nu_j h_j(G_j z)$$

In the notation of [JE19a], the primal violation is

$$\max\{\max_{i=1,..,n_b}\|Hz^k - x_i^k\|_2, \max_{j=1,..,n_r}\|G_j z^k - x_{j+n_b}^k\|_2\}$$

where, n_b is the number of blocks in the loss (controlled by `nblocks` argument to `run`).

If `run` has not been called yet, raises an exception.

Returns **primalErr** – Primal Violation.

Return type `float`

`getScaling`()

Returns the scaling vector. For the $n \times d'$ data matrix A, the scaling vector is $d' \times 1$ vector containing the scaling factors used to normalize new test data. If the `normalize` argument to `addData` was `False`, then the method simply returns a vector of ones.

If no data have been added yet, raises an exception.

Returns **scaling** – scaling vector

Return type 1D NumPy array

getSolution(*descale=False*, *ergodic=False*)

Returns the current primal solution z^k.

If the `intercept` argument was True in `addData`, the intercept coefficient is returned as the first entry of z^k.

If the `run` method has not been called yet, raises an exception.

Parameters

- **descale** (`bool`, optional) – Defaults to False. If the `normalize` argument to `addData` was set to True and `descale` is True, the normalization that was applied to the columns of the data matrix is applied to the entries of z^k, meaning that one may use it to make predictions using unnormalized data. However, if a linear operator was added with `addData` via argument `linOp`, then a warning message will be printed and the solution vector will not be descaled.
- **ergodic** (`bool` or `string`, optional) – Whether to return the primal iterate z^k, or one of its two averaged versions. If `False`, return the primal iterate. If "simple", return $\frac{1}{k}\sum_{t=1}^{k} z^k$; if "weighted", return

$$\frac{\sum_{t=1}^{k} \tau_t z^t}{\sum_{t=1}^{k} \tau_t}$$

where τ_t are the stepsizes used in the hyperplane projections.

Returns **z** – z^k

Return type 1D numpy array

numObservations()

Retrieve the number of observations.

Should only be invoked after calling the `addData` method; otherwise, calling this method raises an exception.

Returns **nrowsOfA** – Number of observations

Return type `int`

numPrimalVars()

Retrieve the number of primal variables (possibly including the intercept).

Should only be invoked after calling the `addData` method; otherwise, calling this method raises an exception.

Returns **nPrimalVars** – Number of primal variables, including the intercept if present

Return type `int`

run(*primalTol=1e-06*, *dualTol=1e-06*, *maxIterations=None*, *keepHistory=False*, *historyFreq=10*, *nblocks=1*, *blockActivation='greedy'*, *blocksPerIteration=1*, *resetIterate=False*, *verbose=False*, *ergodic=None*, *equalizeStepsizes=False*)

Run projective splitting.

Parameters

- **primalTol** (`float`, optional) – Continue running algorithm if primal error is greater than `primalTol`. In the notation of [JE19a], the primal violation is

$$\max\{\max_{i=1,..,n_b} \|Hz^k - x_i^k\|_2, \max_{j=1,..,n_r} \|G_j z^k - x_{j+n_b}^k\|_2\}$$

where, n_b is the number of blocks in the loss (controlled by `nblocks` argument to `run`) and n_r is the number of regularizers. To terminate the method, both primal error and dual error must be smaller than their respective tolerances, or the number of iterations must exceed `maxIteration`. Default 1e-6.

- **`dualTol`** (`float`, optional) – Continue running algorithm if dual error is greater than dualTol. The dual error is

  $$\max\{\max_{i=1,..,n_b} \|y_i^k - w_i^k\|_2, \max_{j=1,..,n_r} \|y_{j+n_b} - w_j^k\|_2\}$$

  where, n_b is the number of blocks in the loss (controlled by `nblocks` argument to `run`) and n_r is the number of regularizers. To terminate the method, both primal error and dual error must be smaller than their respective tolerances, or the number of iterations must exceed `maxIteration`. Default 1e-6.

- **`maxIterations`** (`int`, optional) – Terminate algorithm as soon as it has run for more than `maxIterations` iterations. Default is `None`, which means not to terminate until the `primalTol` and `dualTol` conditions are reached.

- **`keepHistory`** (`bool`, optional) – If `True`, record the algorithm history (see the `getHistory` method). Default is `False`. Note that to keep history requires computing the objective value, which may be slow for large problems.

- **`historyFreq`** (`int`, optional) – If `keepHistory` is `True`, history information is recorded every `historyFreq` iterations. Defaults to 10.

- **`nblocks`** (`int`, optional) – Number of blocks in the projective splitting decomposition of the loss. Defaults to 1. Blocks are contiguous indices and the number of indices in each block varies by at most one.

  `nblocks` must be an integer in the range 1 to n, where n is the number of observations.

  In conjunction with the greedy activation method (see below), choosing `nblocks` larger than 1 has been shown to greatly improve algorithm performance for some problem classes.

  Suppose `nblocks` is set to b and the number of observations is n. Then the first $n \bmod b$ blocks have $\lceil n/b \rceil$ observations and the remainder have $\lfloor n/b \rfloor$ observations. If b divides n, this means that all blocks have n/b observations.

  For example, if number of observations is 100 and nblocks is set to 10 then the blocks would be

  [[0,1,. . . ,9], [10,11,. . . ,19], . . . [90,91,. . . ,99]]

  If the number of observations is 105 and nblocks is set to 10, then the blocks would be 5 blocks of size 11 and 5 blocks of 10, that is,

  [[0,1,. . . ,10], [11,12,..22], . . . [44,45,. . . ,54], [55,56,. . . ,64], . . . [95,96,. . . ,104]]

- **`blockActivation`** (`string`, optional) – Strategy for selecting blocks of the loss to process at each iteration. Defaults to "greedy". Other valid choices are "random" and "cyclic". If there is only one block, all these choices are equivalent.

- **`blocksPerIteration`** (`int`, optional) – Number of blocks to update in each iteration. Defaults to 1. Must be a positive integer in the range 1 to `nblocks`.

- **`resetIterate`** (`bool`, optional) – If `True`, the current values of all working variables (if `run` has been called before) in the projective splitting algorithm (eg: z^k, w_i^k etc) are overwritten with zero vectors before starting the run. Defaults to `False`, meaning that the algorithm starts from its previous state.

- **`verbose`** (`bool`, optional) – If `True`, will print iteration counts every 100 iterations. Defaults to `False`.

- **ergodic** (`bool` or `string`, optional) – If `keepHistory=True`, whether to compute the objective at the primal iterate z^k, or one of its two averaged versions. If `False`, use the primal iterate. If "simple", evaluate at $\frac{1}{k}\sum_{t=1}^{k} z^t$; if "weighted", evaluate at

$$\frac{\sum_{t=1}^{k} \tau_t z^t}{\sum_{t=1}^{k} \tau_t}$$

  where τ_t are the stepsizes used in the hyperplane projections.

- **equalizeStepsizes** (`bool`, optional) – Applies only when using backtracking loss processors (`Forward2Backtrack` and `Forward1Backtrack`). If `True`, set the regularizer stepsizes according to the stepsizes returned by backtracking. Defaults to `False`.

setDualScaling (*dualScaling*)

Changes the dual scaling parameter (gamma)

Parameters **dualScaling** (`float`, optional) – the primal-dual scaling parameter, which is gamma in [JE19a] (algorithm definition on page 9). Must be positive and defaults to 1.0.

4.2 Regularizer Class

class regularizers.**Regularizer** (*prox*, *value=None*, *scaling=1.0*, *step=1.0*, *testLength=100*)

Regularizer class to use as an input to the `ProjSplitFit.addRegularizer` method.

Recall the objective function

$$\min_{z\in\mathbb{R}^d, z_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n} \ell(z_0 + a_i^\top Hz, r_i) + \sum_{j=1}^{n_r} \nu_j h_j(G_j z)$$

The regularizer class is used to define each $\nu_j h_j(G_j z)$ term, with the exception of the optional linear operator G_j, which is supplied when calling `addRegularizer` to introduce the regularizer to the formulation.

You may use standard built-in regularizers, or create objects of this class to define new regularizers. When defining your own regularizers, you must provide a function implementing the regularizer's proximal operator ("prox"). If you wish to compute objective values, you must also supply a function to compute the regularizer value.

__init__ (*prox*, *value=None*, *scaling=1.0*, *step=1.0*, *testLength=100*)

It is only necessary to define *value* if you wish to compute objective function values, either by calling `getObjective` or by using the `keepHistory` option of the `ProjSplitFit.run` method.

Parameters

- **prox** (`function`) – must be a function of two parameters: a `numpy`-style array s and a positive `float` η. This function must return the vector $\text{prox}_{\eta h_j}(s) = \arg\min_x \left\{\eta h_j(x) + (1/2)\|x - s\|^2\right\}$ for arbitrary inputs s and $0 < \eta < \infty$.
- **value** (`function`, optional) – must be a function of one parameter: a numpy-style array. Must returns a float which is the value of $h(x)$. The default is `None`, meaning undefined. The returned value should not include the scaling factor ν_j.
- **scaling** (`float`, optional) – the objective scaling factor ν_j to use with this regularizer. The function will appear in the objective as $\nu_j h_j(G_j z)$. Must be positive and defaults to 1.0.

- **`step`** (`float`, optional) – the stepsize η to use in the proximal steps of projective splitting with this regularizer. Must be positive and defaults to 1.0. Will be overridden on an iteration-by-iteration basis if the `equalizeStepsizes` option is enabled in the `run` method of `ProjSplitFit`.

`getScaling`()

Get the scaling ν_j being used for this regularizer.

Returns **scaling**

Return type `float`

`getStep`()

get the stepsize η being used in the proximal steps for this regularizer.

Returns **stepsize**

Return type `float`

`setScaling`(*scaling*)

Set the scaling factor ν_j.

Parameters **`scaling`** (`float`) – scaling factor; must be positive and finite

`setStep`(*step*)

Set the stepsize η for proximal steps for this regularizer.

Parameters **`step`** (`float`) – stepsize; must be positive and finite

4.3 Built-in Regularizers

`regularizers.`**`L1`**(*scaling=1.0*, *step=1.0*)

Returns an L1 regularizer. The output is an object of class `regularizers.Regularizer` suitable as input to `ProjSplitFit.addRegularizer`.

scaling is the coefficient ν_j that will be applied to the function. That is, the regularizer will appear as $\nu_j\|z\|_1$ or $\nu_j\|G_j z\|_1$ in the objective formulation, depending on whether a linear operator G_j is is supplied when it is introduced into the formulation with `addRegularizer`.

step is the stepsize η that projective splitting will use for proximal steps using this regularizer, unless overridden by the `equalizeStepsizes` option of the `run` method of `ProjSplitFit`.

Parameters

- **`scaling`** (`float`, optional) – Defaults to 1.0. Must be positive and finite
- **`step`** (`float`, optional) – Defaults to 1.0. Must be positive and finite

Returns **regObj**

Return type *`regularizers.Regularizer`* object

`regularizers.`**`L2sq`**(*scaling=1.0*, *step=1.0*)

Create an L2 squared regularizer. The output is an object of class `regularizers.Regularizer` which may be passed to `ProjSplitFit.addRegularizer`.

scaling is the coefficient ν_j that will be applied to the regularizer in the objective. That is, the regularizer will appear as $(\nu_j/2)\|\cdot\|_2^2$. Note the factor of 0.5.

step is the stepsize that projective splitting will use for the proximal steps performed on this regularizer.

Parameters

- **scaling** (float, optional) – Defaults to 1.0. Must be positive and finite.
- **step** (float, optional) – Defaluts to 1.0. Must be positive and finite.

Returns **regObj**

Return type *regularizers.Regularizer* object

regularizers.**L2**(*scaling=1.0*, *step=1.0*)

Create an L2-norm regularizer. Not to be confused with the L2sq regularizer, which is the same function squared and divided by 2.

The output is an object of class regularizers.Regularizer, which may be passed to ProjSplitFit.addRegularizer.

scaling is the coefficient ν_j that will be applied to the function in the objective. That is, the regularizer will appear as $\nu_j\|\cdot\|_2$.

step is the stepsize η that projective splitting will use in proximal steps with respect to this regularizer.

Parameters

- **scaling** (float, optional) – Defaults to 1.0. Must be positive and finite.
- **step** (float, optional) – Defaluts to 1.0. Must be positive and finite.

Returns **regObj**

Return type *regularizers.Regularizer* object

regularizers.**groupL2**(*dimension*, *groups*, *scaling=1.0*, *step=1.0*)

Create a group L2-norm regularizer.

The output is an object of class regularizers.Regularizer, which may be passed to ProjSplitFit.addRegularizer.

The regularizer takes the form

$$h(z) = \nu_j \sum_{G \in \mathcal{G}} \|z_G\|,$$

where $\mathcal{G}$ are the groups and z_G denotes the subvector of z consisting of the elements whose indices are in G. The groups G may not overlap.

dimension is the size of vectors that will be passed to the regularizer in future.

group is an iterable consisting of iterables of integers. Each inner iterable represents the indices in one of the groups.

scaling is the coefficient ν_j that will be applied to the function in the objective.

step is the stepsize η that projective splitting will use in proximal steps with respect to this regularizer.

Parameters

- **dimension** (int) – The size of the vectors to which the regularizer will be applied.
- **groups** (iterable of iterables of int) – An iterable specifying the groups, playing the role of $\mathcal{G}$ in the above formula. Each member of the iterable must itself be an iterable consisting of nonnegative integers whose value is less than dimension. If objects of any other type are encountered, an exception is raised. Each of the inner iterables specifies the indices in a single group G. If any index appears in more than one group, an exception is raised.
- **scaling** (float, optional) – Defaults to 1.0. Specifies ν_j. Must be positive and finite.
- **step** (float, optional) – Defaluts to 1.0. Specifies the proximal stepsize to be applied to the regularizer. Must be positive and finite.

Returns regObj

Return type `regularizers.Regularizer` object

4.4 User-Defined Losses (LossPlugIn Class)

class `losses.LossPlugIn`(*derivative*, *value=None*)

Objects of this class may be used as the `loss` argument of the `ProjSplitFit.addData` method, to define customized loss functions. That argument also accepts `float` or `int` values $p > 1$, which are interpreted as specifying the ℓ_p^p loss, or the string “logistic” to specify the logistic loss function.

Other choices require creating a `LossPlugIn` object. This in turn requires supplying a function to compute the derivative of the loss function. If you plan to compute objective function values, you must also supply a function to compute the loss function value.

`__init__`(*derivative*, *value=None*)

You need only supply a *value* function if you wish to compute objective function values (either with `ProjSplitFit.getObjective` or by enabling history collection in `ProjSplitFit.run`).

Parameters

- **derivative** (`function`) – Function of two 1D `numpy` arrays of the same length, the first containing predicted values and the second containing actual response values. Must output an array of the same length as the two inputs, whose elements consists of partial derivatives with respect to the predicted values. Specifically, supposing that the two input arrays are $q = [q_0\ q_1\ \cdots q_k]$ and $q = [r_0\ r_1\ \cdots r_k]$, the returned array should be contain elements of the form $\frac{\partial}{\partial q_i}\ell(q_i, r_i)$ for each input index i.
- **value** (`function`, optional) – Must accept two `float` arguments and return a single `float`. If supplied the arguments q_i (for the prediction) and r_i (for the response), the function should return $\ell(q_i, r_i)$. Defaults to `None`. If the default is used, however, attempting to compute the objective value will raise an exception.

4.5 Loss Processors

The loss processor classes instruct projective splitting how to process the loss function. The loss processor is specified by the `process` argument of the `ProjSplitFit.addData`.

If you omit the `process` argument to `ProjSplitFit.addData`, then `ProjSplitFit` will use the default loss processor, `Forward2Backtrack`.

When a loss processor for block i is invoked within the projective splitting algorithm, it is provided with the vector Hz^k derived from the current primal solution estimate z^k (which just equals z^k if H was not specified) and the dual solution estimate w_i^k. It returns two vectors x_i^k and y_i^k, which should have the same dimension as w_i^k. These returned vectors must have specific properties in order to guarantee convergence of the algorithm; all the provided loss processor have these properties, with one caveat mentioned below.

4.5.1 Forward-step (Gradient) Loss Processors

Forward2Fixed

class `lossProcessors.`**`Forward2Fixed`**(*step=1.0*)

Two forward steps with a fixed stepsize. The returned vectors take the form

$$x_i^k = Hz^k - \rho(\nabla f_i(Hz^k) - w_i^k)$$
$$y_i^k = \nabla f_i(x_i^k)$$

where the stepsize ρ is fixed and

$$f_i(t) = \frac{1}{n} \sum_{j \in \text{block } i} \ell(t_0 + a_j^T t, r_j)$$

See [JE19a], https://arxiv.org/abs/1803.07043.

Objects of this class may be used as the `process` argument to `ProjSplitFit.addData`.

`__init__`(*step=1.0*)

Parameters **`step`** (`float`, optional) – the stepsize ρ, defaulting to 1.0. Should be positive. For convergence to be guaranteed, the stepsize should be less than $1/L_i$, where L_i is the Lipschitz continuity modulus of the gradient of the function f_i defined above. If this value is unknown or is infinite, use the `Forward2Backtrack` loss processor instead.

Forward2Backtrack

class `lossProcessors.`**`Forward2Backtrack`**(*initialStep=1.0*, *Delta=1.0*, *backtrackFactor=0.7*, *growFactor=1.0*, *growFreq=None*)

Two forward steps with a backtracking linesearch stepsize.

The returned pair of vectors takes the form

$$x_i^k = Hz^k - \rho_{ik}(\nabla f_i(Hz^k) - w_i^k)$$
$$y_i^k = \nabla f_i(x_i^k)$$

where the stepsize ρ_{ik} is discovered by a backtracking linesearch at each iteration and

$$f_i(t) = \frac{1}{n} \sum_{j \in \text{block } i} \ell(t_0 + a_j^T t, r_j)$$

See [JE19a], https://arxiv.org/abs/1803.07043.

Objects of this class may be used as the `process` argument to `ProjSplitFit.addData`.

`__init__`(*initialStep=1.0*, *Delta=1.0*, *backtrackFactor=0.7*, *growFactor=1.0*, *growFreq=None*)

Parameters

- **`initialStep`** (`float`, optional) – Initial trial choice of the stepsize ρ_{ik}, defaulting to 1.0
- **`Delta`** (`float`, optional) – the parameter Δ in backtracking linesearch termination condition of [JE19a]. Larger values make the condition more difficult to satisfy and result in more backtracking iterations and smaller accepted stepsizes. Defaults to 1.0.
- **`backtrackFactor`** (`float`, optional) – How much to shrink the stepsize by at each iteration of backtracking. Must be strictly between 0 and 1. Defaults to 0.7

- **growFactor** (float, optional) – How much to grow the stepsize by before backtracking. Must be at least 1.0. Defaults to 1.0
- **growFreq** (int, optional) – How often, in terms of iterations, to grow the stepsize, defaults to None, which means to never grow the stepsize. Must be at least 1.

Forward2Affine

class lossProcessors.**Forward2Affine**(*Delta=1.0*)

Two forward steps with stepsize automatically tuned for the ℓ_2^2 loss. This loss process is only applicable when the loss function has an affine gradient map, which occurs only in the ℓ_2^2 case. See [JE19a], https://arxiv.org/abs/1803.07043.

Objects of this class may be used as the process argument to ProjSplitFit.addData.

__init__(*Delta=1.0*)

Parameters Delta (float, optional) – parameter in stepsize calculation condition of [JE19a]. Larger values result in smaller stepsizes. Defaults to 1.0

Forward1Fixed

class lossProcessors.**Forward1Fixed**(*stepsize=1.0*, *blendFactor=0.1*)

One forward step with a fixed stepsize. See [JE19b], https://arxiv.org/abs/1902.09025.

The returned vectors are calculated by

$$\begin{aligned} x_i^k &= (1-\alpha)x_i^{k-1} + \alpha H z^k - \rho(y_i^{k-1} - w_i^k) \\ y_i^k &= \nabla f_i(x_i^k) \end{aligned}$$

where the stepsize ρ is constant and

$$f_i(t) = \frac{1}{n}\sum_{j\in\text{block } i} \ell(t_0 + a_j^T t, r_j).$$

See [JE19b], https://arxiv.org/abs/1902.09025.

Objects of this class may be used as the process argument to ProjSplitFit.addData.

Note that convergence has not been proven for this this loss processor in the case that blocksPerIteration is smaller than nBlocks, although it is suspected that it does indeed converge in this case.

__init__(*stepsize=1.0*, *blendFactor=0.1*)

Parameters

- **stepsize** (float, optional) – stepsize ρ, defaulting to 1.0. Must be positive. To guarantee convergence, should be less than $2(1-\alpha)/L_i$, where α is the blendFactor constant below and L_i is the modulus of Lipschitz continuity of the function f_i as defined above. If L_i is unknown or infinite, use the Forward2backtrack loss processor instead.
- **blendFactor** (float, optional) – The averaging parameter α in one-forward-step calculations above. Defaults to 0.1. Must be strictly between 0 and 1.

Forward1Backtrack

class `lossProcessors.`**`Forward1Backtrack`**(*initialStep=1.0*, *blendFactor=0.1*, *backTrackFactor=0.7*, *growFactor=1.0*, *growFreq=None*)

One forward step with stepsize determined by a backtracking line search. See [JE19b], https://arxiv.org/abs/1902.09025.

The returned vectors are of the form

$$
\begin{aligned}
x_i^k &= (1-\alpha)x_i^{k-1} + \alpha H z^k - \rho_{ik}(y_i^{k-1} - w_i^k) \\
y_i^k &= \nabla f_i(x_i^k)
\end{aligned}
$$

where the stepsize ρ_{ik} is discovered by a backtracking linesearch at each iteration and

$$
f_i(t) = \frac{1}{n} \sum_{j \in \text{block } i} \ell(t_0 + a_j^T t, r_j)
$$

See [JE19b], https://arxiv.org/abs/1902.09025.

Note that convergence has not been proven for this this loss processor in the case that `blocksPerIteration` is smaller than `nBlocks`, although it is suspected that it does indeed converge in this case.

Objects of this class may be used as the `process` argument to `ProjSplitFit.addData`.

`__init__`(*initialStep=1.0*, *blendFactor=0.1*, *backTrackFactor=0.7*, *growFactor=1.0*, *growFreq=None*)

Parameters

- **`initialStep`** (`float`, optional) – Initial trial stepsize in first iteration, defaults to 1.0
- **`blendFactor`** (`float`, optional) – The averaging parameter α in calculation above. Defaults to 0.1. Must be strictly between 0 and 1.
- **`backtrackFactor`** (`float`, optional) – How much to shrink the stepsize by at each iteration of backtracking. Must be strictly between 0 and 1. Defaults to 0.7
- **`growFactor`** (`float`, optional) – How much to grow the stepsize before backtracking. Must be at least 1.0. Defaults to 1.0
- **`growFreq`** (`int`, optional) – How often, in terms of iterations, to grow the stepsize, defaults to `None`, which means to never grow the stepsize. Must be at least 1.

4.5.2 Backward-Step (Proximal) Based Loss Processors

Backward Exact

class `lossProcessors.`**`BackwardExact`**(*stepsize=1.0*)

Exact backward step for quadratic loss functions, calculated via matrix inversion. Only applicable to the ℓ_2^2 loss function. Appropriate matrix inverses are cached before the first iteration.

The returned vectors are of the form

$$
\begin{aligned}
x_i^k &= \text{prox}_{\rho f_i}(H z^k + \rho w_i^k) \\
y_i^k &= \rho^{-1}(H z^k + \rho w_i^k - x_i^k)
\end{aligned}
$$

where

$$
f_i(t) = \frac{1}{n} \sum_{j \in \text{block } i} \ell(t_0 + a_j^T t, r_j)
$$

and the proximal operator is computed exactly by solving the appropriate system of linear equations. Only applicable when using the ℓ_2^2 loss.

If the involved matrices are wide (having a number of rows less than half the number of columns), the matrix inversion lemma is used to reduce the size of the inverted matrix, see Section 4.2.4 of https://web.stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf.

Objects of this class may be used as the `process` argument to `ProjSplitFit.addData`.

`__init__`(*stepsize=1.0*)

Parameters **`stepsize`** (`float`, optional) – Stepsize ρ, defaults to 1.0

`setStep`(*step*)

Set the stepsize for this loss processor.

Parameters **`step`** (`float`) – stepsize. Must be positive and finite

Backward Step with Conjugate Gradient

`class` `lossProcessors.`**`BackwardCG`**(*relativeErrorFactor=0.9*, *stepsize=1.0*, *maxIter=100*)

Approximate backward step for the ℓ_2^2 loss, computed by the conjugate gradient method for linear equations. Only applicable to the ℓ_2^2.

Updates are of the form

$$\begin{aligned} x_i^k &= \text{prox}_{\rho f_i}(Hz^k + \rho w_i^k) \\ y_i^k &= \rho^{-1}(Hz^k + \rho w_i^k - x_i^k) \end{aligned}$$

where

$$f_i(t) = \frac{1}{n} \sum_{j \in \text{block } i} \ell(t_0 + a_j^T t, r_j).$$

The proximal operator is only computed approximately via a conjugate gradient method. This method is only applicable the ℓ_2^2 loss, in which case computing the prox is equivalent to solving a linear system of equations.

The conjugate gradient method is iterated until the relative error criteria specified in [Eck17][CE18][JE19a], are met, or the maximum number of iterations is reached. Convergence is not guaranteed when the maximum number of conjugate gradient iterations is reached in more than a finite number of projective splitting iterations.

Objects of this class may be used as the `process` argument to `ProjSplitFit.addData`.

`__init__`(*relativeErrorFactor=0.9*, *stepsize=1.0*, *maxIter=100*)

Parameters

- **`relativeErrorFactor`** (`float`, optional) – σ, relative error factor. Must be in [0,1). Defaults to 0.9
- **`stepsize`** (`float`, optional) – stepsize ρ, defaultings to 1.0
- **`maxIter`** (`int`, optional) – Maximum number of iterations of conjugate gradient. Defaults to 100. Must be at least 1.

Backward Step with L-BFGS

class lossProcessors.**BackwardLBFGS**(*step=1.0*, *relativeErrorFactor=0.9*, *memory=10*, *c1=0.0001*, *c2=0.9*, *shrinkFactor=0.7*, *growFactor=1.1*, *maxiter=100*, *lineSearchIter=20*)

Approximate backward step computed by the limited-memory BFGS (L-BFGS) method.

The returned vectors are of the form

$$\begin{aligned} x_i^k &= \text{prox}_{\rho f_i}(Hz^k + \rho w_i^k) \\ y_i^k &= \rho^{-1}(Hz^k + \rho w_i^k - x_i^k) \end{aligned}$$

where

$$f_i(t) = \frac{1}{n} \sum_{j \in \text{block } i} \ell(t_0 + a_j^T t, r_j).$$

The proximal operator is computed approximately by the L-BFGS method, iterated until the relative error criteria specified in [Eck17][CE18][JE19a], are met, or the maximum number of iterations is reached. Convergence is not guaranteed when the maximum number L-BFGS of iterations is reached in more than a finite number of projective splitting iterations.

Objects of this class may be used as the `process` argument to `ProjSplitFit.addData`.

__init__(*step=1.0*, *relativeErrorFactor=0.9*, *memory=10*, *c1=0.0001*, *c2=0.9*, *shrinkFactor=0.7*, *growFactor=1.1*, *maxiter=100*, *lineSearchIter=20*)

Parameters

- **step** (`float`, optional) – Stepsize ρ, defaulting to 1.0
- **relativeErrorFactor** (`float`, optional) – σ, relative error factor. Must be in [0,1). Defaults to 0.9
- **memory** (`int`, optional) – how many iterations of memory are held by L-BFGS. Defaults to 10. Must be at least 1.
- **c1** (`float`, optional) – the c_1 parameter in the Wolfe linesearch used by L-BFGS. Defaults to 1e-4. Must be strictly between 0 and 1, with $c_1 < c_2$.
- **c2** (`float`, optional) – the c_2 parameter in the Wolfe linesearch used by L-BFGS. Defaults to 0.9. Must be strictly between 0 and 1, with $c_1 < c_2$.
- **shrinkFactor** (`float`, optional) – How much to shrink stepsize during the Wolfe linesearch. Must be strictly between 0 and 1 and defaults to 0.7
- **growFactor** (`float`, optional) – How much to grow stepsize at the outset of the Wolfe line-search. Must be greater than 1, and defaults to 1.1
- **maxiter** (`int`, optional) – maximum number of iterations of L-BFGS. Defaults to 100. Must be at least 1.
- **lineSearchIter** (`int`, optional) – maximum number of iterations of Wolfe line-search. Defaults to 20. Must be at least 1.

4.5.3 Other Methods

Each loss processor object also inherits the following useful methods.

`lossProcessors.LossProcessor.`**`getStep`**`(`*self*`)`

Return the stepsize in use with this loss processor.

Returns **step** – stepsize

Return type `float`

`lossProcessors.LossProcessor.`**`setStep`**`(`*self*, *step*`)`

Set the stepsize for this loss processor.

Parameters **`step`** (`float`) – stepsize. Must be positive and finite

BIBLIOGRAPHY

[ACS14] Abdullah Alotaibi, Patrick L Combettes, and Naseer Shahzad. Solving coupled composite monotone inclusions by successive Fejér approximations of their Kuhn–Tucker set. *SIAM Journal on Optimization*, 24(4):2076–2095, 2014.

[CE18] Patrick L. Combettes and Jonathan Eckstein. Asynchronous block-iterative primal-dual decomposition methods for monotone inclusions. *Mathematical Programming*, 168(1-2):645–672, 2018.

[Eck17] Jonathan Eckstein. A simplified form of block-iterative operator splitting and an asynchronous algorithm resembling the multi-block alternating direction method of multipliers. *Journal of Optimization Theory and Applications*, 173(1):155–182, 2017.

[ES08] Jonathan Eckstein and Benar Fux Svaiter. A family of projective splitting methods for the sum of two maximal monotone operators. *Mathematical Programming*, 111(1-2):173–199, 2008.

[ES09] Jonathan Eckstein and Benar Fux Svaiter. General projective splitting methods for sums of maximal monotone operators. *SIAM Journal on Control and Optimization*, 48(2):787–811, 2009.

[JE19a] Patrick R Johnstone and Jonathan Eckstein. Projective splitting with forward steps: asynchronous and block-iterative operator splitting. Preprint 1803.07043, arXiv, 2019.

[JE19b] Patrick R Johnstone and Jonathan Eckstein. Single-forward-step projective splitting: exploiting cocoercivity. Preprint 1902.09025, arXiv, 2019.

[YB18] Xiaohan Yan and Jacob Bien. Rare Feature Selection in High Dimensions. Preprint 1803.06675, arXiv, 2018.

INDEX