projSplitFit

Release 1

Patrick R. Johnstone and Jonathan Eckstein

Jul 22, 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:

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

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
- y_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 first summation in this formulation is the *loss*, measuring how well the predictions $z_0 + a_i^\top H z$ obtained from the dataset using the regression parameters (z_0, z) match the observed responses y_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 presepunt 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 or abstract linear opertors as defined by the `scipy.sparse.linalg.LinearOperator` class.

1.1 Brief technical overview

The project splitting algorithm is a primal-dual algorithm based on separating hyperplanes. A *dual solution* is a tuple of vectors $w = (w_1, \ldots, w_d)$ that certify the optimality of the "primal" vector z for the problem above. At each iteration, the algorithm maintains an estimate (z, 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*. Furthermore, there are numerous options for processing each block, including approximate backward (proximal) steps and various kinds of forward steps.

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 minuetes 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.

Most of the tests operate by running the algorithm on an optimization problem and checking that `projSplitFit` find 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

3.1 Adding Data

Consider the least-squares problem defined as

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

Assuming the matrix A is a 2D *NumPy* array, and y is a 1D *NumPy* array, or list, then to solve this problem with projSplitFit, use the following code

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

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).

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. Set it when calling the constructor:

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

(the default value is 1), or via the `setDualScaling` method:

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

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. To do this, set the `intercept` argument to the `addData` method to True (which is the default). Note that added 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 unit ℓ_2 norm.

3.5 Adding a Regularizer

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

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

where $\|z\|_1 = \sum_i |z_i|$. To solve this model instead, before calling `run()` we can invoke the `addRegularizer` method:

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

The built-in method `L1` returns an object of class `regularizers.Regularizer` which 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 $\|x\|_2$.

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 wants to calculate function values for performance tracking. The *prox* method returns the proximal operator for the function scaled by some amount. That is

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

The value function simply returns the value $f(x)$. Both of these functions must handle NumPy arrays. Value must return a float and prox must return a NumPy array with the same length as the input.

Adding multiple regularizers in *projSplitFit* is easy. Suppose one wants to solve the lasso with an additional constraint that each component of the solution must be non-negative. That is solve

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

The non-negativity constraint can be thought of as another regularizer. That is

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

where

$$g(z) = \begin{cases} \infty & \text{if some } z_i < 0 \\ 0 & \text{else} \end{cases}$$

To solve this problem with *projSplitFit* the user must define the regularizer object for g and then add it to the model with `addRegularizer`. This is done as follows:

```
from regularizers import Regularizer
def prox_g(z,sigma):
  return (z>=0)*z
regObj = Regularizer(prox_g)
projSplit.addRegularizer(regObj)
projSplit.run()
```

The proximal operator is just the projection onto the constraint set. Note that `prox_g` must still have a second argument for the scaling even though for this particular function it is not used.

3.7 Linear Operator Composed with a Regularizer

Sometimes, one would like to compose a regularizer with a linear operator. This occurs in Total Variation deblurring for example. *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 (matrix) G. The linear operator can be added as an argument to the `addRegularizer` method as follows:

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

G must be a 2D NumPy array (or similar). The number of columns of G must equal the number of primal variables, as defined by the matrix A which is input to `addData`. If not, *ProjSplitFit* will raise an Exception.

3.8 User-Defined Losses

Just as the user may define their own regularizers, they may define their own loss. This is achieved via the `losses.LossPlugIn` class. Objects of this class can be passed into `addData` as the `process` argument. To define a loss, one needs to define its derivative method. Optionally, one may also define its value method if one would like to compute function values for performance tracking.

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

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

To use this loss:

```
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(deriv,val)
projSplit.addData(A,y,loss=loss)
```

3.9 Complete Example: Rare Feature Selection

Let's look at a complete example from page 34 of our paper [JE19]. The problem of interest is

$$\min_{\substack{\gamma_0 \in \mathbb{R} \\ \gamma \in \mathbb{R}^{|\mathcal{T}|}}} \left\{ \frac{1}{2n} \|\gamma_0 e + XH\gamma - y\|_2^2 + \lambda\big(\mu\|\gamma_{-r}\|_1 + (1-\mu)\|H\gamma\|_1\big) \right\}$$

First let's deal with the loss. The loss is the ℓ_2^2 loss. Note that it is composed with a linear operator H. There are two ways to deal with this. If the size of the matrices is not too much of a concern, one may pre-compute a new observation matrix as `Xnew = X*H`. If this is prohibitive, the linear operator can be composed with the loss, meaning the *ProjSplitFit* handles it internally and does not explicitly compute the matrix product. This option is controlled via the `linearOp` argument to `addData`.

Taking this option, the loss is dealt with as follows:

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

Note that, by default, the intercept term γ_0 is added.

The first regularizer needs to be custom-coded, as it leaves out the first variable, which is the root of the tree. It is dealt with as follows:

```
from regularizers import Regularizer
def prox(gamma,sigma):
  temp = numpy.zeros(gamma.shape)
  temp[1:] = (gamma[1:]>sigma)*(gamma[1:]-sigma)
  temp[1:] += (gamma[1:]<-sigma)*(gamma[1:]+sigma)
  temp[0]=gamma[0]
  return temp
regObj = Regularizer(prox,scaling=lam*mu)
projSplit.addRegularizer(regObj)
```

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:

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

Finally we are ready to run the method via:

```
projSplit.run()
```

One can obtain the final objective value and solution via:

```
optimalVal = projSplit.getObjective()
gammastar = projSplit.getSolution()
```

3.10 Loss Process Objects

Projective splitting comes with a rich array of ways to update the hyperplane at each iteration. In the original paper [ES08], the computation was based on the *prox*. Since then, several new calculations have been devised based on *forward steps*, i.e. *gradient* calculations, making projective splitting a true first-order method [JE18], [JE19].

In *ProjSplitFit*, there are a large number of options for which update method to use with respect to the blocks of variables associated with the *loss*. This is controlled by the `process` argument to the `addData` method. This argument must be a class derived from `lossProcessors.LossProcessor`. *ProjSplitFit* supports the following built-in loss processing classes defined in `lossProcessors.py`:

- `Forward2Fixed` two-forward-step update with fixed stepsize, see [JE18]
- `Forward2Backtrack` two-forward-step update with backtracking stepsize, see [JE18]. Note this is the *default* loss processor if the *process* argument is ommitted from `addData`
- `Forward2Affine` two-forward-step with the affine trick, see [JE18]. Only available when `loss=2`
- `Forward1Fixed` one-forward-step with fixed stepsize, see [JE19]
- `Forward1Backtrack` one-forward-step with backtracking stepsize, see [JE19]
- `BackwardExact` Exact backward step for ℓ_2^2 loss via matrix inversion. Only available with `loss=2`
- `BackwardCG` Backward step via conjugate gradient, only available when `loss=2`
- `BackwardLBFGS` Backward step via LBFGS solver.

To select a loss processor, one creates an object of the appropriate class from above, calling the constructor with the desired parameters, and then passes the object into `addData` as the `process` argument. For example, to use `BackwardLBFGS`:

```
import lossProcessors as lp
processObj = lp.BackwardLBFGS()
projSplit.addData(A,y,loss=2,process=processObj)
```

This will use BackwardLBFGS with all of the default parameters. See the detailed documentation for all of the possible parameters and settings for each loss process class.

The user may wish to define their own loss process classes. They must derive from `lossProcessors.LossProcessor` and they must implement the `initialize` and `update` methods. Of course, convergence cannot be guaranteed unless the user knows of a supporting mathematical theory for their process update method.

3.11 Embedding Regularizers

Projective splitting handles regularizers via their proxes. A regularizer is typically handled by including a new block of variables. However, it is possible to embed one regularizer into the block that handles the loss. In this case, the loss is handled in a forward-backward manner, with the forward step calculated, and then the backward step on the same block of variables. For example, with `Forward2Fixed` and embedding the update would be

$$x_i^k = \text{prox}_{\rho g}(z^k - \rho(\nabla f_i(z^k) - w_i^k))$$

Note that the prox is computed in-line with the forward step.

To enable this option, use the `embed` argument to the `addRegularizer` call, when adding the regularizer to the method.

If `nblocks` is greater than 1, the prox is performed on each block.

3.12 Options for the `run()` Method

The `run` method has several important options which we briefly discuss. The first is `nblocks`. This controls how many blocks projective splitting breaks the loss into for processing. Recall the loss is

$$\frac{1}{n}\sum_{i=1}^{n} \ell(z_0 + a_i^\top H z, y_i)$$

An important property of projective splitting is *block iterativeness*: It does not need to process every observation at each iteration. Instead, it may break the n observations into `nblocks` and process as few as one block at a time. `nblocks` may be anything from `1`, meaning all observations are processed at each iteration, to `n`, meaning every observation is treated as a block. `nblocks` defaults to 1.

The blocks are contiguous runs of indices. If $nblocks$ does not divide the number of rows/observations, then we use the formula

$$n = \lceil n/n_b \rceil n\%n_b + \lfloor n/n_b \rfloor (n_b - n\%n_b).$$

so that there are two groups of blocks, those with $\lceil n/n_b \rceil$ number of indices and those with $\lfloor n/n_b \rfloor$. That way, the number of indices in any two blocks differs by at most 1.

The number of blocks processed per iteration is controlled via the argument `blocksPerIteration` which defaults to 1.

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

- "random", randomly selected block
- "cyclic", cycle through the blocks
- "greedy", (default) use the greedy heuristic of [JE18] page 24 to select blocks.

3.13 Other Important Methods of ProjSplitFit

The `keepHistory` and `historyFreq` arguments to `run()` allow you to choose 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.

`getObjective()` simply returns the objective value at the current primal iterate.

`getSolution()` returns the primal iterate z^k. If the `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. That way, the coefficient vector can be used with unnormalized data such as new test data. However the method `getScaling()` returns this scaling vector. This scaling vector can then be applied to normalize new test data. To normalize a new test datapoint `xtest`:

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

CHAPTER

FOUR

DETAILED DOCUMENTATION

4.1 ProjSplitFit Class

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

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

Please refer to

- arxiv.org/abs/1803.07043 (algorithm definition page 9)
- arxiv.org/abs/1902.09025 (algorithm definiteion pages 10-11)

To create an object, call:

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

dualScaling (defaults to 1.0) is gamma in the algorithm definitions from the above papers.

The general optimization objective this can solve is

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

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
- y_i for $i = 1, \ldots, n$ are the 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.

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

regularizers are added via the `addRegularizer` method.

The algorithm is run via the `run` method.

__init__(*dualScaling=1.0*)

Parameters **dualScaling** (float, optional) – the primal-dual scaling parameter which is gamma in arxiv.org/abs/1803.07043 (algorithm definition page 9) and arxiv.org/abs/1902.09025 (algorithm definiteion pages 10-11). dualScaling must be > 0 and defaults to 1.0.

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

Adds data for the data fitting model.

Recall that the general optimization objective solved by this package is

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

Parameters

- **observations** (*2d ndarray or matrix*) – each row of the observations matrix being a_i above
- **responses** (1d ndarray or list or numpy array) – each element equal to y_i above
- **loss** (float or string or *losses.LossPlugIn*) – May be a float greater than 1, the string 'logistic', 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. Default is True.
- **normalize** (bool,optional) – whether to normalize columns of the data matrix to have unit norm. If True, data matrix will be copied. Default is True.
- **linearOp** (scipy.sparse.linalg.LinearOperator or similar, optional) – adds matrix H in Eq. (1). Defaults to the identity.

addRegularizer(*regObj*, *linearOp=None*, *embed=False*)

adds a regularizer to the optimization problem.

Recall the optimization problem

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

This method adds each h_j and G_j above

Parameters

- **regObj** (*regularizers.Regularizer*) – object of class *regularizers.Regularizer*
- **linearOp** (scipy.sparse.linalg.LinearOperator or similar,optional) – adds matrix G_j in above
- **embed** (bool,optional) – internal option in projective splitting. For forward-type loss process updates, perform the "prox" of this regularizer in a forward-backward style update. Defaults to False

getDualScaling()

Returns the current setting of dualScaling

Returns the dualScaling parameter

Return type float

getDualViolation()

Returns the current dual violation.

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

The dual violation is

$$\max_i \|y_i^k - w_i^k\|_2$$

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

Returns dualErr – Dual Violation.

Return type float

getHistory()

Returns array of history data on most recent run().

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 num 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 $\phi(p^k)$ 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()

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 a value method, calling getObjective raises an Exception.

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

Return type float

getPrimalViolation()

Returns the current primal violation.

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

The primal violation is

$$\max_i \|G_i z^k - x_i^k\|_2$$

where, with some abuse of notation, G_i is the linear operator associated with the ith block.

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 for each feature. This scaling vector can be used with new test data to normalize the features. If the `normalize` argument to `addData` was set to False, then an exception will be raised.

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

Returns **scaling** – scaling vector or None if `normalize` set to False in

Return type 1D NumPy array

`getSolution`(*descale=False*)

Returns the current primal solution vector. Recall the objective function

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

Returns the current primal solution z^k.

If the `intercept` argument was True in `addData`, the intercept coefficient is 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 the `descale` argument here is True, the normalization that was applied to the columns of the data matrix is applied to the entries of z^k, meaning that the user may use the original unnormalized data matrix with this new feature, and also may use it on new 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.

Returns **z** – z^k

Return type 1D numpy array

`numObservations`()

Retrieve the number of observations.

After the `addData` method has been called, one may call this method.

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

Returns **nrowsOfA** – Number of observations

Return type `int`

`numPrimalVars`()

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

After the `addData` method has been called, one may call this method.

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

Returns nPrimalVars – Number of primal variables including the intercept if that option is taken

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*)
Run projective splitting.

Parameters

- **primalTol** (float,optional) – Continue running algorithm if primal error is greater than primalTol. The primal error is

$$\max_i \|G_i z^k - x_i^k\|_2$$

where, with some abuse of notation, G_i is the linear operator associated with the ith block. Note that to terminate the method, both primal error AND dual error must be smaller than their respective tolerances. Or the number of iterations exceeds the maximum number. Default 1e-6.

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

$$\max_i \|y_i^k - w_i^k\|_2$$

Note that to terminate the method, both primal error AND dual error must be smaller than their respective tolerances. Or the number of iterations exceeds the maximum number. Default 1e-6.

- **maxIterations** (int,optional) – Terminate algorithm if ran for more than maxIterations iterations. Default is None meaning do not terminate until primalTol and dualTol are reached.

- **keepHistory** (bool,optional) – If True, record the history (see getHistory method). Default False.

- **historyFreq** (int,optional) – Frequency to keep history, defaults to every 10 iterations. Note that to keep history requires computing the objective which may be slow for large problems.

- **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.

  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 was 105 and nblocks is set to 10, then the blocks would be 5 blocks of 11 and 5 blocks of 10, i.e.

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

  This uses the formula

$$n = \lceil n/n_b \rceil n\%n_b + \lfloor n/n_b \rfloor (n_b - n\%n_b).$$

- **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".

- **blocksPerIteration** (int,optional) – Number of blocks to update in each iteration. Defaults to 1.
- **resetIterate** (bool,optional) – If True, the current values of all variables (if run has been called before) in projective splitting (eg: z^k, w_i^k etc) are erased and initialized to 0. Defaults to False.
- **verbose** (bool,optional) – Verbose as in printing iteration counts etc. Defaults to False.

setDualScaling (*dualScaling*)

Changes the dual scaling parameter (gamma)

Parameters **dualScaling** (float, optional) – the primal-dual scaling parameter which is gamma in arxiv.org/abs/1803.07043 (algorithm definition page 9). dualScaling must be > 0 and defaults to 1.0.

4.2 Regularizer Class

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

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

Recall the objective function

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

The regularizer class essentially defines each $h_i(G_i z)$ term via methods for evaluating the prox of h_i and the function itself. Note the matrix G_i is added in the addRegularizer method of projSplitFit.

The user may use objects of this class to define regularizers, or may use one of the built-in regularizers.

To use this class, one must define a function for computing the prox of the regularizer and can then use that as an input to the constructor to create a Regularizer object.

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

Only define *value* if you wish to compute objective function values within ProjSplitFit to monitor progress, as its not necessary for the actual operation of ProjSplitFit. However, if the value function is set to None, but then the ProjSplit.getObjective() method is called, then it will raise an Exception.

Parameters

- **prox** (*function*) – must be a function of two parameters: a numpy-style array and a float which is the multiple applied to the function. That is, this function must return $\text{prox}_{\eta h}(x)$ for arbitrary inputs x and η.
- **value** (*function, optional*) – must be a function of one parameter: a numpy-style array. Must returns a float which is the value of $h(x)$. Default is None, meaning not defined. Note that this is the value of the *unscaled* function. In other words, with a scaling of 1.
- **scaling** (float,optional) – Scaling to use with this regularizer in the objective. The function will appear in the objective as

$$\nu h(x)$$

for a scaling ν. Defaults to 1.0

- **step** (float,optional) – Stepsize to use in the proximal steps of projective splitting with this regularizer. Defaults to 1.0

getScaling()
Get the scaling being used for this regularizer in the objective.

Returns scaling

Return type float

getStepsize()
get the stepsize being used in the proximal steps for this regularizer by projective splitting.

Returns stepsize

Return type float

setScaling(*scaling*)
Set the scaling. That is, in the objective, the regularizer will be scaled by scaling=nu. It will appear as

$$\nu h(x)$$

Parameters **scaling** (float) – scaling

setStep(*step*)
Set the stepsize being used in the proximal steps for this regularizer by projective splitting.

Parameters **step** (float) – stepsize

4.3 Built-in Regularizers

regularizers.**L1**(*scaling=1.0*, *step=1.0*)
Create the L1 regularizer. The output is an object of class regularizers.Regularizer which may be input to ProjSplitFit.addRegularizer.

Scaling is the coefficient ν that will be applied to the function in the objective. That is, it will appear as

$$\nu\|z\|_1$$

step is the stepsize that projective splitting will use for the proximal steps w.r.t. this regularizer.

Parameters

- **Scaling** (float,optional) – Defaults to 1.0
- **Stepsize** (float,optional) – Defaluts to 1.0

Returns regObj

Return type *regularizers.Regularizer* object

regularizers.**L2sq**(*scaling=1.0*, *step=1.0*)
Create the L2 squared regularizer. The output is an object of class regularizers.Regularizer which may be input to ProjSplitFit.addRegularizer.

Scaling is the coefficient ν that will be applied to the function in the objective. That is, it will appear as

$$\frac{\nu}{2}\|z\|_2^2.$$

Note the factor of 0.5.

step is the stepsize that projective splitting will use for the proximal steps w.r.t. this regularizer.

Parameters

- **Scaling** (float,optional) – Defaults to 1.0
- **Stepsize** (float,optional) – Defaluts to 1.0

Returns **regObj**

Return type *regularizers.Regularizer* object

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

Create the L2 norm regularizer. Not to be confused with the L2sq regularizer, which is this function *squared*.

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

Scaling is the coefficient ν that will be applied to the function in the objective. That is, it will appear as

$$\nu\|z\|_2$$

step is the stepsize that projective splitting will use for the proximal steps w.r.t. this regularizer.

Parameters

- **Scaling** (float,optional) – Defaults to 1.0
- **Stepsize** (float,optional) – Defaluts to 1.0

Returns **regObj**

Return type *regularizers.Regularizer* object

4.4 User-Defined Losses (Loss PlugIn Class)

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

Objects of this class may be used as the input loss to the ProjSplitFit.addData method to define custom losses.

The user may set the argument loss to ProjSplitFit.addData to an integer $p \geq 1$ to use the ℓ_p^p loss, or they may set it to "logistic" to use the logistic loss.

However, if the user would like to define their own loss, then they must write a function for computing the derivative of the loss and pass it into the constructor to get an object of this class. This can then be used as the input loss to ProjSplitFit.addData.

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

Only implement value if you wish to compute objective function values of the outputs of ProjSplitFit to monitor progress. It is not necessary for the operation of ProjSplitFit. However, if the value function is set to None, but then the ProjSplitFit.getObjective method is called, then it will raise an Exception. Similarly if ProjSplitFit.run is called with the keepHistory argument set to True.

Parameters

- **derivative** (*function*) – Function of two 1D NumPy arrays of the same length. Must output an array of the same length as the two inputs which is the derivative wrt the first argument of the loss evaluated at each pair of elements in the input arrays. That is, for inputs:

```
[x_1,x_2,...,x_n], [y_1,y_2,...,y_n]
```

output:

```
[z_1,z_2,...,z_n]
```

where

$$z_i = \frac{\partial}{\partial x}\ell(x_i, y_i)$$

and the partial derivative is w.r.t. the first argument to ℓ.

- **value** (*function, optional*) – Must handle two float inputs and output a float. Defaults to None, not supported. Outputs

$$\ell(x, y)$$

for inputs x and y.

4.5 Loss Processors

These classes instruct projective splitting how to process the blocks of variables associated with the loss, and are added in the `projSplitFit.addData` method as the `process` input.

If you're not interested in playing with different loss processors, then just leave the `process` argument to `ProjSplitFit.addData` unused and `ProjSplitFit` uses the default loss processor, `Forward2Backtrack`.

4.5.1 Forward-step (Gradient) Based Loss Processors

Forward2Fixed

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

Two forward steps with a fixed stepsize. Updates of 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

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

See 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) – stepsize, defaults to 1.0

Forward2Backtrack

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

Two forward steps with backtracking linesearch stepsize.

Updates of 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 discovered by backtracking and

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

See 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

- **step** (`float`,optional) – stepsize, defaults to 1.0
- **Delta** (`float`,optional) – parameter in backtracking line search check condition. Defaults to 1.0
- **backtrackFactor** (`float`,optional) – How much to decrement the stepsize by at each iteration of backtracking. Must be 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 never grow the stepsize. Must be at least one.

Forward2Affine

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

Two forward steps with stepsize automatically tuned. Only works for affine gradients, i.e. when the loss is the squared loss, i.e. p = 2. See 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 backtracking line search check condition. Defaults to 1.0

Forward1Fixed

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

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

Updates of the form

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

where the stepsize ρ is constant and

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

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

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

Parameters

- **stepsize** (`float`,optional) – stepsize, defaults to 1.0
- **blendFactor** (`float`,optional) – Averaging parameter α in one forward step update. Defaults to 0.1. Must be 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 a backtracking line-search stepsize. See https://arxiv.org/abs/1902.09025.

Updates of the form

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

where the stepsize ρ is discovered by backtracking and

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

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) – Stepsize in first iteration, defaults to 1.0
- **blendFactor** (`float`,optional) – Averaging parameter α in one forward step update. Defaults to 0.1. Must be between 0 and 1.
- **backtrackFactor** (`float`,optional) – How much to decrement the stepsize by at each iteration of backtracking. Must be 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 never grow the stepsize. Must be at least one.

4.5.2 Backward-Step (Proximal) Based Loss Processors

Backward Exact

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

Exact backward step for quadratics via matrix inversion. Only works with the squared loss, i.e. p=2. Appropriate matrix inverses are cached before the first iteration.

If the involed matrices are wide (number of rows less than half number of cols), the matrix inversion lemma is used, see Sec. 4.2.4 of https://web.stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf.

See https://arxiv.org/abs/1902.09025.

Updates of the form

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

where

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

and the proximal operator is computed exactly by solving the appropriate linear equation. Only available when the loss is the ℓ_2^2 loss.

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

__init__(*stepsize=1.0*)

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

setStep(*step*)

Set the stepsize in use with this loss processor.

Parameters **step** (float) – stepsize

Backward Step with Conjugate Gradient

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

Backward step via conjugate gradient for quadratics. Only works for the squared loss, i.e. p=2.

See https://arxiv.org/abs/1902.09025.

Updates of the form

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

where

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

The proximal operator is only computed approximately via a conjugate gradient method. This only works for 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 critera of https://arxiv.org/abs/1902.09025 are met, or the max number of iterations is run.

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, defaults to 1.0
- **maxIter** (int,optional) – max number of iterations of conjugate gradient. Defaults to 100. Must be at least one.

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*)

Backward step via the L-BFGS solver.

See https://arxiv.org/abs/1902.09025.

Updates of the form

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

where

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

The proximal operator is computed approximately via the L-BFGS solver until the relative error criteria of https://arxiv.org/abs/1902.09025 are met, or a max number of iterations is run.

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, defaults 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 in L-BFGS. Defaults to 10. Must be at least one.
- **c1** (float,optional) – c_1 parameter in the Wolfe linesearch. Defaults to 1e-4. Must be between 0 and 1 and $c_1 < c_2$.
- **c2** (float,optional) – c_2 parameter in the Wolfe linesearch. Defaults to 0.9. Must be between 0 and 1 and $c_1 < c_2$.
- **shrinkFactor** (float,optional) – How much to shrink stepsize during Wolfe line-search. Must be between 0 and 1 and defaults to 0.7
- **growFactor** (float,optional) – How much to grow stepsize during Wolfe line-search. Must be > 1 and defaults to 1.1
- **maxiter** (int,optional) – max number of iterations of L-BFGS. Defaults to 100. Must be at least one.

- **lineSearchIter** (int,optional) – max number of iterations of Wolfe linesearch. Defaults to 20. Must be at least one.

4.5.3 Other Methods

Each loss processor object also inherits the following useful methods.

lossProcessors.LossProcessor.**getStep**(*self*)
Get the stepsize in use with this loss processor.

Returns **step** – stepsize

Return type float

lossProcessors.LossProcessor.**setStep**(*self*, *step*)
Set the stepsize in use with this loss processor.

Parameters **step** (float) – stepsize

BIBLIOGRAPHY

[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.

[JE18] Patrick R Johnstone and Jonathan Eckstein. Projective splitting with forward steps: asynchronous and block-iterative operator splitting. *arXiv preprint arXiv:1803.07043*, 2018.

[JE19] Patrick R Johnstone and Jonathan Eckstein. Single-forward-step projective splitting: exploiting cocoercivity. *arXiv preprint arXiv:1902.09025*, 2019.

INDEX

Symbols

A

B

F

G

L

N

P

R

S