Group Lasso Documentation

Release 1.5.0

Yngve Mardal Moe

Apr 26, 2022

Contents:

This library provides efficient computation of sparse group lasso regularise linear and logistic regression.

CHAPTER 1

What is group lasso?

It is often the case that we have a dataset where the covariates form natural groups. These groups can represent biological function in gene expression data or maybe sensor location in climate data. We then wish to find a sparse subset of these covariate groups that describe the relationship in the data. Let us look at an example to crystalise the usefulness of this further.

Say that we work as data scientists for a large Norwegian food supplier and wish to make a prediction model for the amount of that will be sold based on weather data. We have weather data from cities in Norway and need to know how the fruit should be distributed across different warehouses. From each city, we have information about temperature, precipitation, wind strength, wind direction and how cloudy it is. Multiplying the number of cities with the number of covariates per city, we get 1500 different covariates in total. It is unlikely that we need all these covariates in our model, so we seek a sparse set of these to do our predictions with.

Let us now assume that the weather data API that we use charge money by the number of cities we query, but the amount of information we get per city. We therefore wish to create a regression model that predicts fruit demand based on a sparse set of city observations. One way to achieve such sparsity is through the framework of group lasso regularisation[1].

[1] Yuan M, Lin Y. Model selection and estimation in regression with grouped variables. Journal of the Royal Statistical Society: Series B (Statistical Methodology). 2006 Feb;68(1):49-67.

CHAPTER 2

What is sparse group lasso

The sparse group lasso regulariser[2] is an extension of the group lasso regulariser that also promotes parameter-wise sparsity. It is the combination of the group lasso penalty and the normal lasso penalty. If we consider the example above, then the sparse group lasso penalty will yield a sparse set of groups and also a sparse set of covariates in each selected group. An example of where this is useful is if each city query has a set price that increases based on the number of measurements we want from each city.

[2] Simon, N., Friedman, J., Hastie, T., & Tibshirani, R. (2013). A sparse-group lasso. Journal of Computational and Graphical Statistics, 22(2), 231-245.

CHAPTER 3

A quick mathematical interlude

Let us now briefly describe the mathematical problem solved in group lasso regularised machine learning problems. Originally, group lasso algorithm[1] was defined as regularised linear regression with the following loss function

$$\arg\frac{1}{n}\min_{\beta_g\in\mathbb{R}^{\mathfrak{d}}}||\sum_{g\in\mathcal{G}}[\mathbf{X}_g\beta_g]-\mathbf{y}||_2^2+\lambda_1||\beta||_1+\lambda_2\sum_{g\in\mathcal{G}}\sqrt{d_g}||\beta_g||_2,$$

where $\mathbf{X}_g \in \mathbb{R}^{n\times d_g}$ is the data matrix corresponding to the covariates in group g, β_g is the regression coefficients corresponding to group g, $\mathbf{y}\in\mathbf{R}^n$ is the regression target, n is the number of measurements, d_g is the dimensionality of group g, λ_1 is the parameter-wise regularisation penalty, λ_2 is the group-wise regularisation penalty and $\mathcal{G}$ is the set of all groups.

Notice, in the equation above, that the 2-norm is *not* squared. A consequence of this is that the regulariser has a "kink" at zero, uninformative covariate groups to have zero-valued regression coefficients. Later, it has been popular to use this methodology to regularise other machine learning algorithms, such as logistic regression. The "only" thing neccesary to do this is to exchange the squared norm term, $||\sum_{g\in\mathcal{G}}[\mathbf{X}_g\beta_g]-\mathbf{y}||_2^2$, with a general loss term, $L(\beta;\mathbf{X},\mathbf{y})$, where β and $\mathbf{X}$ is the concatenation of all group coefficients and group data matrices, respectively.

CHAPTER 4

API design

The `group-lasso` python library is modelled after the `scikit-learn` API and should be fully compliant with the `scikit-learn` ecosystem. Consequently, the `group-lasso` library depends on `numpy`, `scipy` and `scikit-learn`.

Currently, the only supported algorithm is group-lasso regularised linear and multiple regression, which is available in the `group_lasso.GroupLasso` class. However, I am working on an experimental class with group lasso regularised logistic regression, which is available in the `group_lasso.LogisticGroupLasso` class. Currently, this class only supports binary classification problems through a sigmoidal transformation, but I am working on a multiple classification algorithm with the softmax transformation.

All classes in this library is implemented as both `scikit-learn` transformers and their regressors or classifiers (dependent on their use case). The reason for this is that to use lasso based models for variable selection, the regularisation coefficient should be quite high, resulting in sub-par performance on the actual task of interest. Therefore, it is common to first use a lasso-like algorithm to select the relevant features before using another another algorithm (say ridge regression) for the task at hand. Therefore, the `transform` method of `group_lasso.GroupLasso` to remove the columns of the input dataset corresponding to zero-valued coefficients.

4.1 Installation guide

4.1.1 Dependencies

`group-lasso` support Python 3.5+, Additionally, you will need `numpy`, `scikit-learn` and `scipy`. Hosever, these packages should be installed automatically when installing this codebase.

4.1.2 Installing group-lasso

`group-lasso` is available through Pypi and can easily be installed with a pip install:

```
pip install group-lasso
```

The Pypi version is updated regularly, however for the latest update, you should clone from GitHub and install it directly.:

```
git clone https://github.com/yngvem/group-lasso.git
cd group-lasso
python setup.py
```

4.2 Examples

Below is a gallery of examples

4.2.1 Warm start to choose regularisation strength

Setup

```
import matplotlib.pyplot as plt
import numpy as np

from group_lasso import GroupLasso

np.random.seed(0)
GroupLasso.LOG_LOSSES = True
```

Set dataset parameters

```
group_sizes = [np.random.randint(10, 20) for i in range(50)]
active_groups = [np.random.randint(2) for _ in group_sizes]
groups = np.concatenate(
    [size * [i] for i, size in enumerate(group_sizes)]
).reshape(-1, 1)
num_coeffs = sum(group_sizes)
num_datapoints = 10000
noise_std = 20
```

Generate data matrix

```
X = np.random.standard_normal((num_datapoints, num_coeffs))
```

Generate coefficients

```
w = np.concatenate(
    [
        np.random.standard_normal(group_size) * is_active
        for group_size, is_active in zip(group_sizes, active_groups)
    ]
)
w = w.reshape(-1, 1)
```

(continues on next page)

(continued from previous page)

```
true_coefficient_mask = w != 0
intercept = 2
```

Generate regression targets

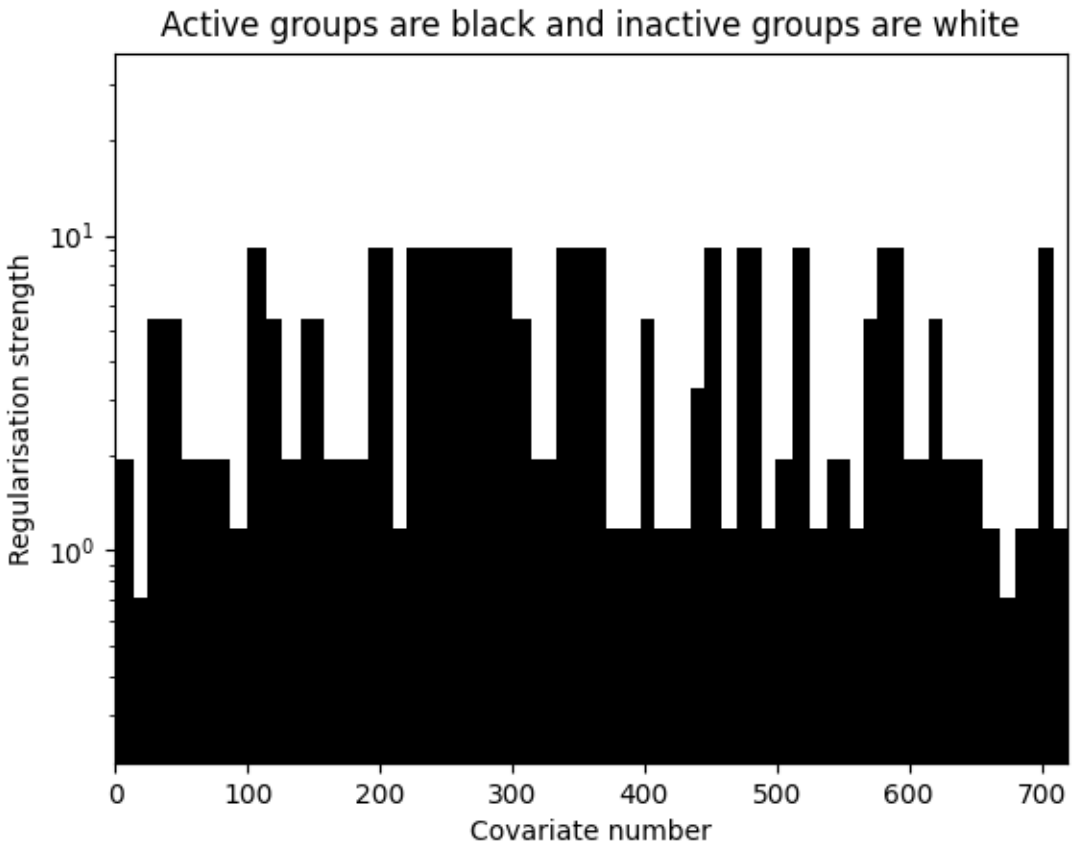
```
y_true = X @ w + intercept
y = y_true + np.random.randn(*y_true.shape) * noise_std
```

Generate estimator and train it

```
num_regs = 10
regularisations = np.logspace(-0.5, 1.5, num_regs)
weights = np.empty((num_regs, w.shape[0],))
gl = GroupLasso(
    groups=groups,
    group_reg=5,
    l1_reg=0,
    frobenius_lipschitz=True,
    scale_reg="inverse_group_size",
    subsampling_scheme=1,
    supress_warning=True,
    n_iter=1000,
    tol=1e-3,
    warm_start=True,  # Warm start to start each subsequent fit with previous weights
)

for i, group_reg in enumerate(regularisations[::-1]):
    gl.group_reg = group_reg
    gl.fit(X, y)
    weights[-(i + 1)] = gl.sparsity_mask_.squeeze()
```

Visualise chosen covariate groups

```
plt.figure()
plt.pcolormesh(np.arange(w.shape[0]), regularisations, -weights, cmap="gray")
plt.yscale("log")
plt.xlabel("Covariate number")
plt.ylabel("Regularisation strength")
plt.title("Active groups are black and inactive groups are white")
plt.show()
```



Total running time of the script: (0 minutes 20.652 seconds)

4.2.2 GroupLasso for logistic regression

A sample script for group lasso regression

Setup

```
import matplotlib.pyplot as plt
import numpy as np

from group_lasso import LogisticGroupLasso

np.random.seed(0)
LogisticGroupLasso.LOG_LOSSES = True
```

Set dataset parameters

```
group_sizes = [np.random.randint(10, 20) for i in range(50)]
active_groups = [np.random.randint(2) for _ in group_sizes]
groups = np.concatenate([size * [i] for i, size in enumerate(group_sizes)])
```

(continues on next page)

(continued from previous page)

```
num_coeffs = sum(group_sizes)
num_datapoints = 10000
noise_std = 1
```

Generate data matrix

```
X = np.random.standard_normal((num_datapoints, num_coeffs))
```

Generate coefficients

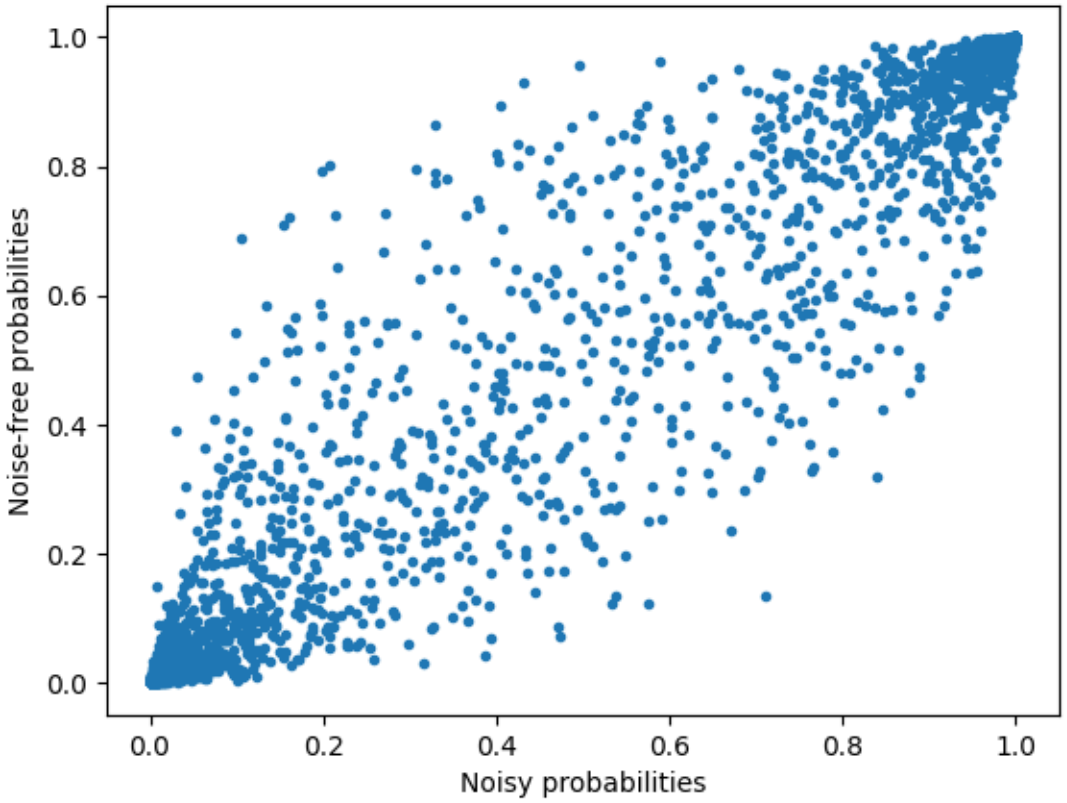
```
w = np.concatenate(
    [
        np.random.standard_normal(group_size) * is_active
        for group_size, is_active in zip(group_sizes, active_groups)
    ]
)
w = w.reshape(-1, 1)
true_coefficient_mask = w != 0
intercept = 2
```

Generate regression targets

```
y_true = X @ w + intercept
y = y_true + np.random.randn(*y_true.shape) * noise_std
p = 1 / (1 + np.exp(-y))
p_true = 1 / (1 + np.exp(-y_true))
c = np.random.binomial(1, p_true)
```

View noisy data and compute maximum accuracy

```
plt.figure()
plt.plot(p, p_true, ".")
plt.xlabel("Noisy probabilities")
plt.ylabel("Noise-free probabilities")
# Use noisy y as true because that is what we would have access
# to in a real-life setting.
best_accuracy = ((p_true > 0.5) == c).mean()
```



Generate estimator and train it

```
gl = LogisticGroupLasso(
    groups=groups,
    group_reg=0.05,
    l1_reg=0,
    scale_reg="inverse_group_size",
    subsampling_scheme=1,
    supress_warning=True,
)

gl.fit(X, c)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/group-lasso/envs/latest/lib/python3.
↪7/site-packages/group_lasso-1.5.0-py3.7.egg/group_lasso/_group_lasso.py:839:␣
↪UserWarning: Subsampling is not stable for logistic regression group lasso.
/home/docs/checkouts/readthedocs.org/user_builds/group-lasso/envs/latest/lib/python3.
↪7/site-packages/group_lasso-1.5.0-py3.7.egg/group_lasso/_fista.py:119:␣
↪ConvergenceWarning: The FISTA iterations did not converge to a sufficient minimum.
You used subsampling then this is expected, otherwise, try increasing the number of␣
↪iterations or decreasing the tolerance.
```

(continues on next page)

(continued from previous page)

```
LogisticGroupLasso(groups=array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, ␣
↪0,  0,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
        4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
        5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,
        5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  7,  7,
        7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,
        8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9,  9,
        9,  9,  9,  9,  9, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10...
       43, 43, 43, 43, 43, 43, 43, 43, 43, 43, 44, 44, 44, 44, 44, 44, 44,
       44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 45, 45, 45, 45, 45, 45, 45,
       45, 45, 45, 45, 45, 45, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46,
       46, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47,
       47, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 49, 49, 49, 49,
       49, 49, 49, 49, 49, 49]),
                    l1_reg=0, scale_reg='inverse_group_size',
                    subsampling_scheme=1, supress_warning=True)
```

Extract results and compute performance metrics

```
# Extract info from estimator
pred_c = gl.predict(X)
sparsity_mask = gl.sparsity_mask_
w_hat = gl.coef_

# Compute performance metrics
accuracy = (pred_c == c).mean()

# Print results
print(f"Number variables: {len(sparsity_mask)}")
print(f"Number of chosen variables: {sparsity_mask.sum()}")
print(f"Accuracy: {accuracy}, best possible accuracy = {best_accuracy}")
```

Out:

```
Number variables: 720
Number of chosen variables: 292
Accuracy: 0.504607, best possible accuracy = 0.9698
```

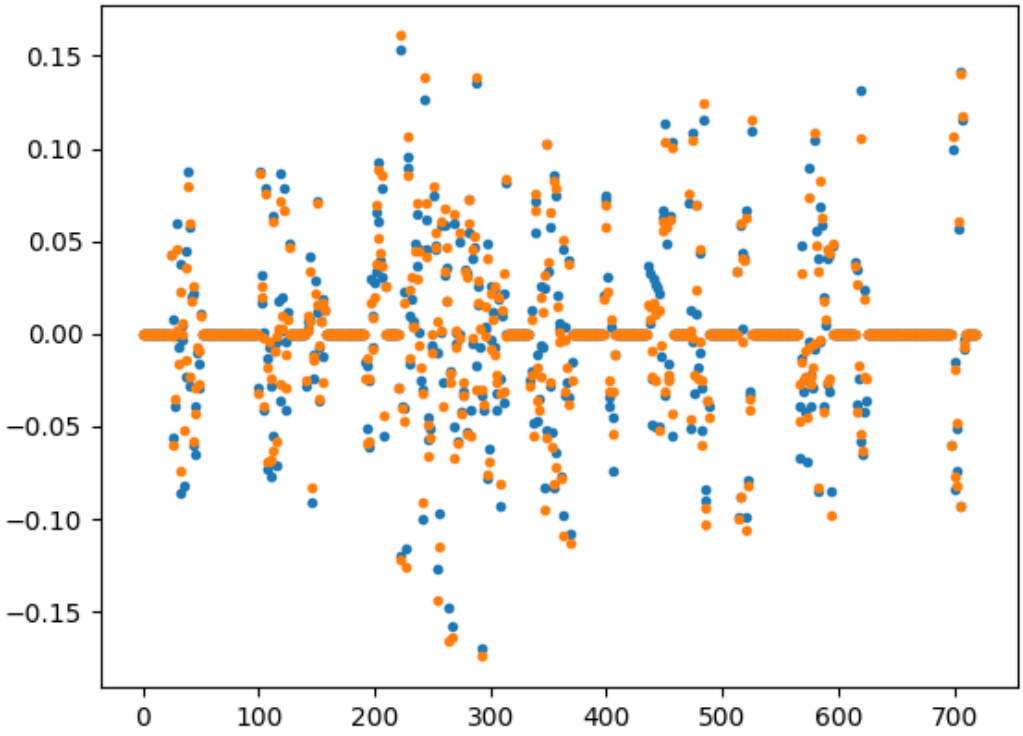
Visualise regression coefficients

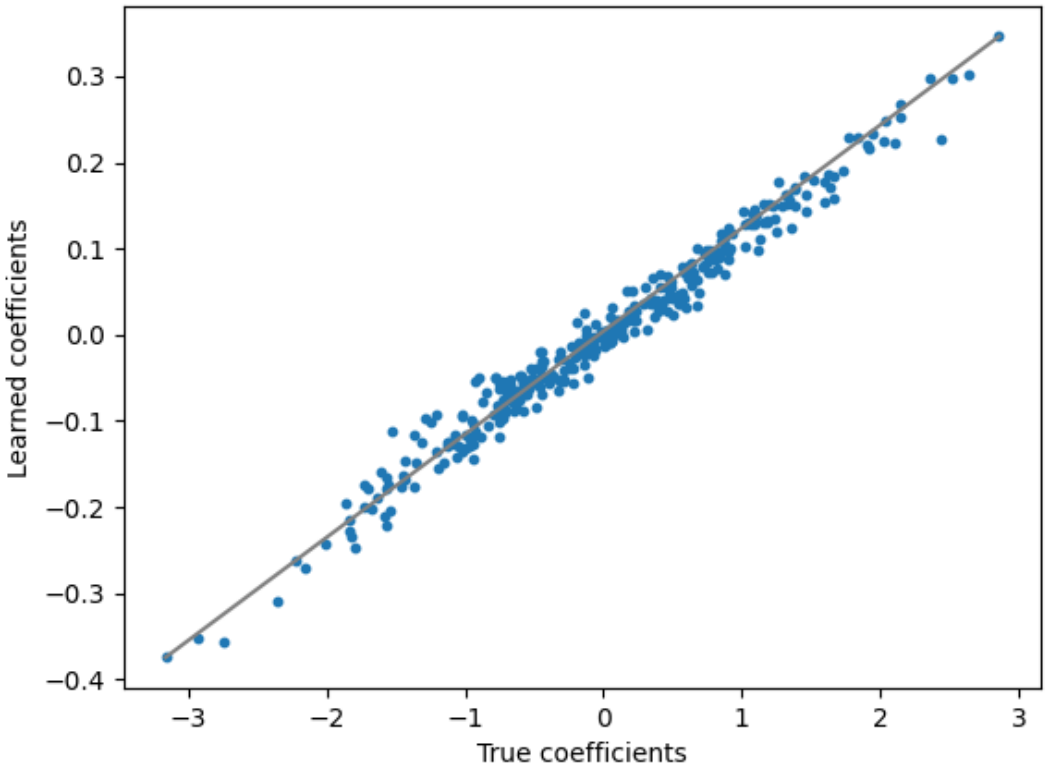
```
coef = gl.coef_[:, 1] - gl.coef_[:, 0]
plt.figure()
plt.plot(w / np.linalg.norm(w), ".", label="True weights")
plt.plot(
    coef / np.linalg.norm(coef), ".", label="Estimated weights",
)

plt.figure()
plt.plot([w.min(), w.max()], [coef.min(), coef.max()], "gray")
plt.scatter(w, coef, s=10)
plt.ylabel("Learned coefficients")
```

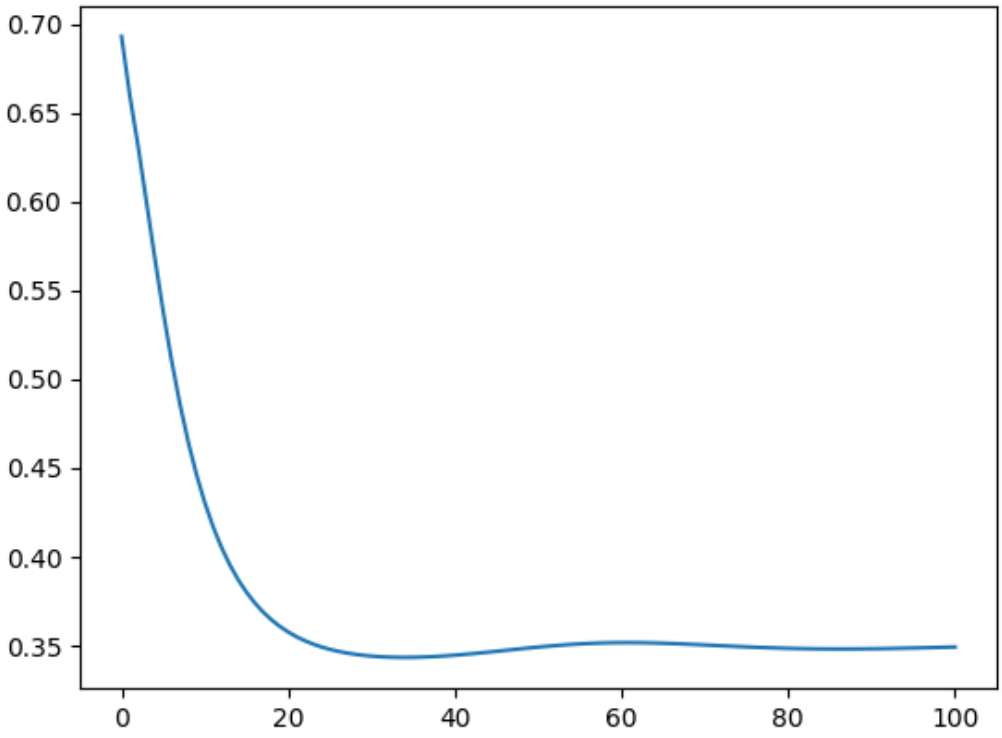
(continues on next page)

(continued from previous page)

```
plt.xlabel("True coefficients")

plt.figure()
plt.plot(gl.losses_)

plt.show()
```




Learned coefficients
True coefficients
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−3
−2
−1
0
1
2
3



-

Total running time of the script: (0 minutes 12.679 seconds)

4.2.3 GroupLasso for linear regression

A sample script for group lasso regression

Setup

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import r2_score

from group_lasso import GroupLasso

np.random.seed(0)
GroupLasso.LOG_LOSSES = True
```

Set dataset parameters

```
group_sizes = [np.random.randint(10, 20) for i in range(50)]
active_groups = [np.random.randint(2) for _ in group_sizes]
groups = np.concatenate(
```

(continues on next page)

(continued from previous page)

```
    [size * [i] for i, size in enumerate(group_sizes)]
).reshape(-1, 1)
num_coeffs = sum(group_sizes)
num_datapoints = 10000
noise_std = 20
```

Generate data matrix

```
X = np.random.standard_normal((num_datapoints, num_coeffs))
```

Generate coefficients

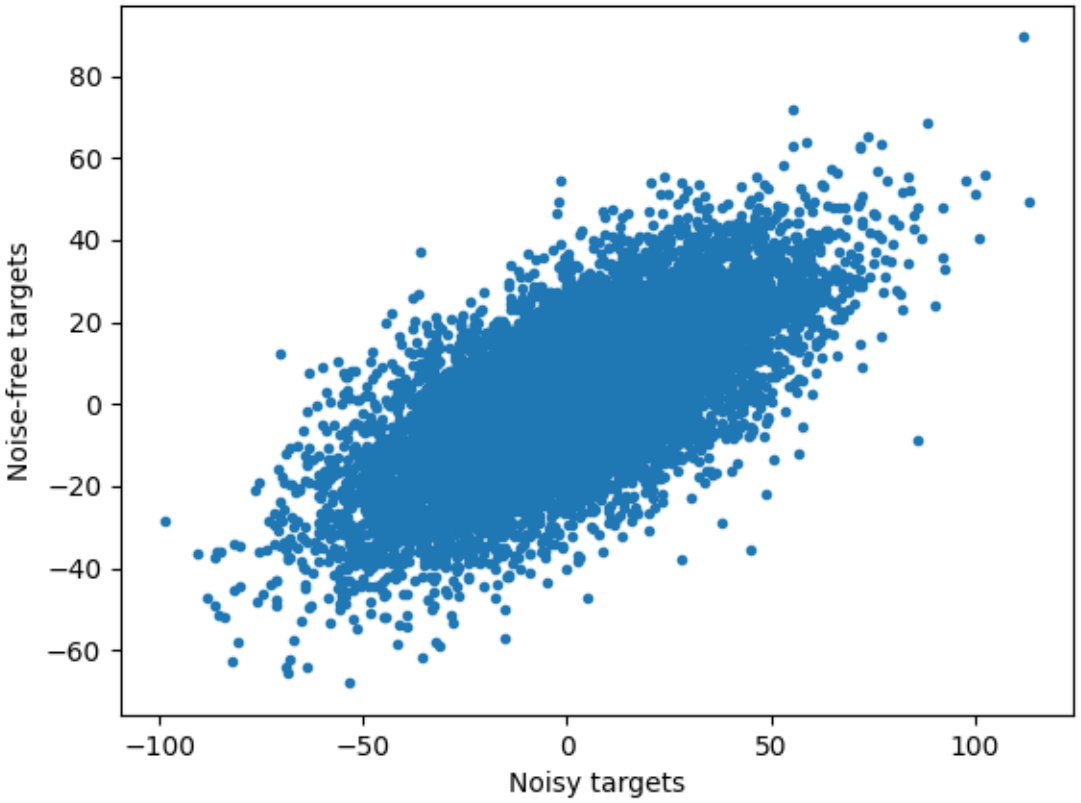
```
w = np.concatenate(
    [
        np.random.standard_normal(group_size) * is_active
        for group_size, is_active in zip(group_sizes, active_groups)
    ]
)
w = w.reshape(-1, 1)
true_coefficient_mask = w != 0
intercept = 2
```

Generate regression targets

```
y_true = X @ w + intercept
y = y_true + np.random.randn(*y_true.shape) * noise_std
```

View noisy data and compute maximum R^2

```
plt.figure()
plt.plot(y, y_true, ".")
plt.xlabel("Noisy targets")
plt.ylabel("Noise-free targets")
# Use noisy y as true because that is what we would have access
# to in a real-life setting.
R2_best = r2_score(y, y_true)
```



Generate estimator and train it

```
gl = GroupLasso(
    groups=groups,
    group_reg=5,
    l1_reg=0,
    frobenius_lipschitz=True,
    scale_reg="inverse_group_size",
    subsampling_scheme=1,
    supress_warning=True,
    n_iter=1000,
    tol=1e-3,
)
gl.fit(X, y)
```

Out:

```
GroupLasso(frobenius_lipschitz=True, group_reg=5,
           groups=array([[ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
```

(continues on next page)

(continued from previous page)

```
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
```

(continues on next page)

(continued from previous page)

```
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 5],
       [ 5],
       [ 5],
       [ 5],...
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
```

(continues on next page)

(continued from previous page)

```
       [49]]),
           l1_reg=0, n_iter=1000, scale_reg='inverse_group_size',
           subsampling_scheme=1, supress_warning=True, tol=0.001)
```

Extract results and compute performance metrics

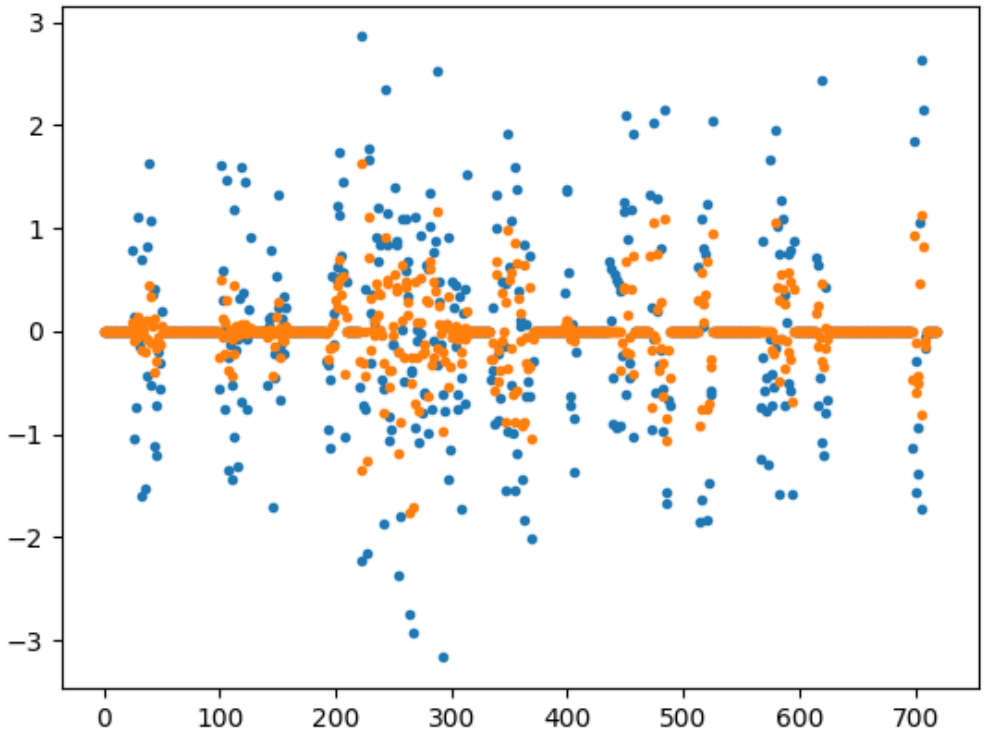
```
# Extract info from estimator
yhat = gl.predict(X)
sparsity_mask = gl.sparsity_mask_
w_hat = gl.coef_

# Compute performance metrics
R2 = r2_score(y, yhat)

# Print results
print(f"Number variables: {len(sparsity_mask)}")
print(f"Number of chosen variables: {sparsity_mask.sum()}")
print(f"R^2: {R2}, best possible R^2 = {R2_best}")
```

Out:

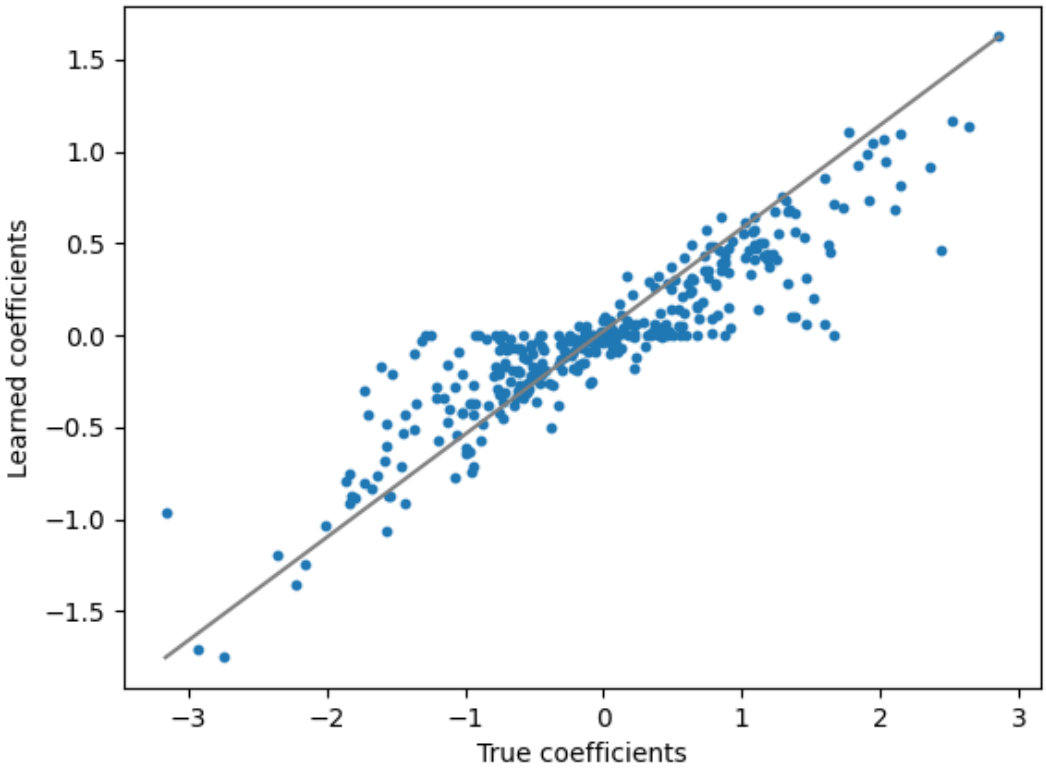
```
Number variables: 720
Number of chosen variables: 313
R^2: 0.29097931452380443, best possible R^2 = 0.46262785225190173
```

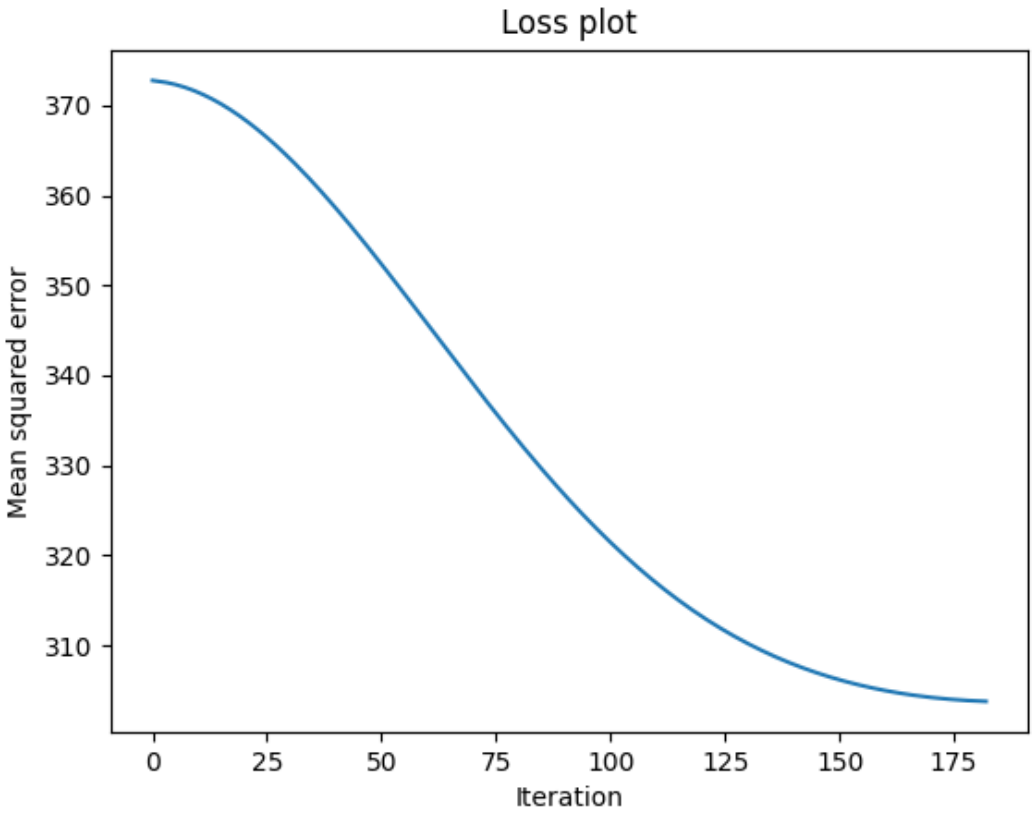
Visualise regression coefficients

```
plt.figure()
plt.plot(w, ".", label="True weights")
plt.plot(w_hat, ".", label="Estimated weights")

plt.figure()
plt.plot([w.min(), w.max()], [w_hat.min(), w_hat.max()], "gray")
plt.scatter(w, w_hat, s=10)
plt.ylabel("Learned coefficients")
plt.xlabel("True coefficients")

plt.figure()
plt.plot(gl.losses_)
plt.title("Loss plot")
plt.ylabel("Mean squared error")
plt.xlabel("Iteration")

print("X shape: {X.shape}".format(X=X))
print("True intercept: {intercept}".format(intercept=intercept))
print("Estimated intercept: {intercept}".format(intercept=gl.intercept_))
plt.show()
```


3
2
1
0
−1
−2
−3
0
100
200
300
400
500
600
700

•


Learned coefficients
True coefficients
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3



Out:

```
X shape: (10000, 720)
True intercept: 2
Estimated intercept: [2.08271211]
```

Total running time of the script: (0 minutes 6.999 seconds)

4.2.4 GroupLasso as a transformer

A sample script to demonstrate how the group lasso estimators can be used for variable selection in a scikit-learn pipeline.

Setup

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score
from sklearn.pipeline import Pipeline

from group_lasso import GroupLasso

np.random.seed(0)
```

Set dataset parameters

```
group_sizes = [np.random.randint(10, 20) for i in range(50)]
active_groups = [np.random.randint(2) for _ in group_sizes]
groups = np.concatenate(
    [size * [i] for i, size in enumerate(group_sizes)]
).reshape(-1, 1)
num_coeffs = sum(group_sizes)
num_datapoints = 10000
noise_std = 20
```

Generate data matrix

```
X = np.random.standard_normal((num_datapoints, num_coeffs))
```

Generate coefficients

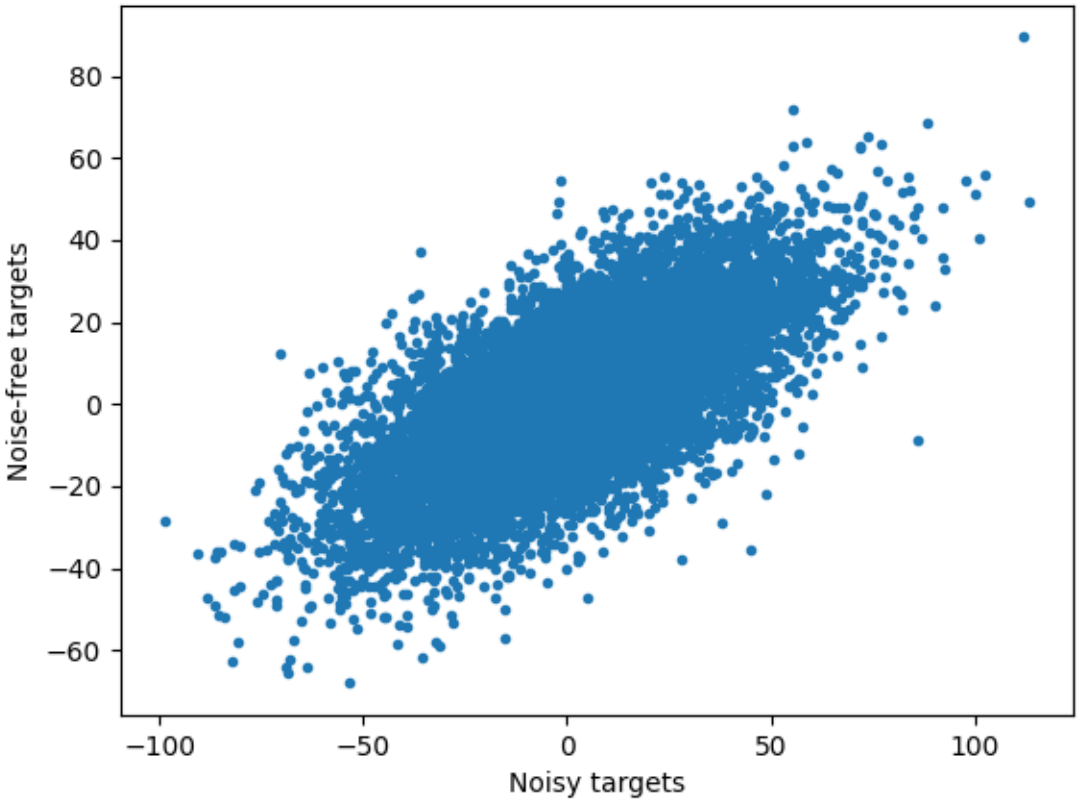
```
w = np.concatenate(
    [
        np.random.standard_normal(group_size) * is_active
        for group_size, is_active in zip(group_sizes, active_groups)
    ]
)
w = w.reshape(-1, 1)
true_coefficient_mask = w != 0
intercept = 2
```

Generate regression targets

```
y_true = X @ w + intercept
y = y_true + np.random.randn(*y_true.shape) * noise_std
```

View noisy data and compute maximum R^2

```
plt.figure()
plt.plot(y, y_true, ".")
plt.xlabel("Noisy targets")
plt.ylabel("Noise-free targets")
# Use noisy y as true because that is what we would have access
# to in a real-life setting.
R2_best = r2_score(y, y_true)
```



Generate pipeline and train it

```
pipe = Pipeline(
    memory=None,
    steps=[
        (
            "variable_selection",
            GroupLasso(
                groups=groups,
                group_reg=5,
                l1_reg=0,
                scale_reg="inverse_group_size",
                subsampling_scheme=1,
                supress_warning=True,
            ),
        ),
        ("regressor", Ridge(alpha=0.1)),
    ],
)
pipe.fit(X, y)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/group-lasso/envs/latest/lib/python3.
↪7/site-packages/group_lasso-1.5.0-py3.7.egg/group_lasso/_fista.py:119:␣
↪ConvergenceWarning: The FISTA iterations did not converge to a sufficient minimum.
You used subsampling then this is expected, otherwise, try increasing the number of␣
↪iterations or decreasing the tolerance.

Pipeline(steps=[('variable_selection',
                 GroupLasso(group_reg=5,
                            groups=array([[ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 0],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 1],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 2],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
       [ 3],
```

(continues on next page)

(continued from previous page)

```
       [ 3],
       [ 3],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 4],
       [ 5]...
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [46],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [47],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
       [48],
```

(continues on next page)

(continued from previous page)

```
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49],
       [49]]),
                           l1_reg=0, scale_reg='inverse_group_size',
                           subsampling_scheme=1, supress_warning=True)),
                ('regressor', Ridge(alpha=0.1))])
```

Extract results and compute performance metrics

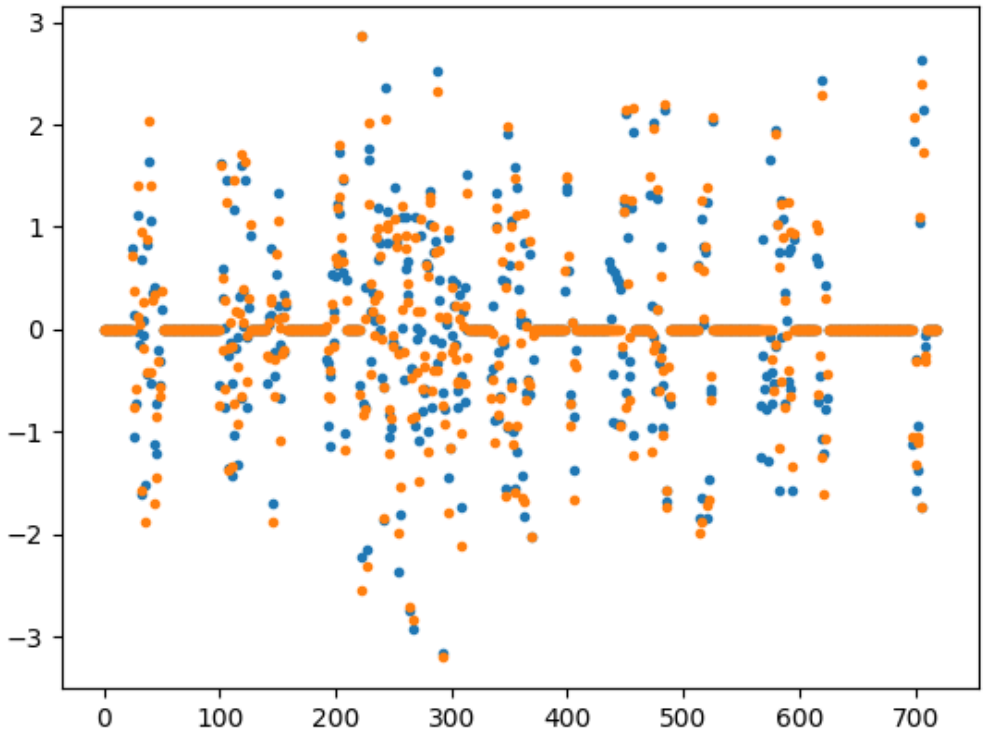
```
# Extract from pipeline
yhat = pipe.predict(X)
sparsity_mask = pipe["variable_selection"].sparsity_mask_
coef = pipe["regressor"].coef_.T

# Construct full coefficient vector
w_hat = np.zeros_like(w)
w_hat[sparsity_mask] = coef

R2 = r2_score(y, yhat)

# Print performance metrics
print(f"Number variables: {len(sparsity_mask)}")
print(f"Number of chosen variables: {sparsity_mask.sum()}")
print(f"R^2: {R2}, best possible R^2 = {R2_best}")
```

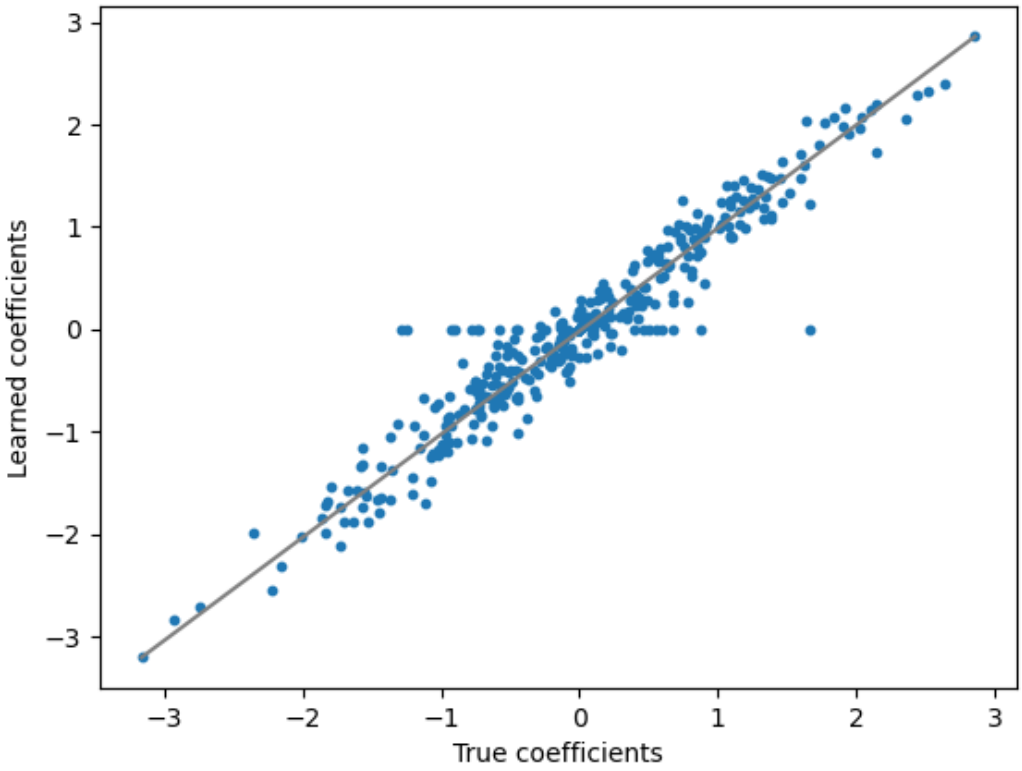
Out:

```
Number variables: 720
Number of chosen variables: 313
R^2: 0.46373947136901283, best possible R^2 = 0.46262785225190173
```

Visualise regression coefficients

```
for i in range(w.shape[1]):
    plt.figure()
    plt.plot(w[:, i], ".", label="True weights")
    plt.plot(w_hat[:, i], ".", label="Estimated weights")

plt.figure()
plt.plot([w.min(), w.max()], [coef.min(), coef.max()], "gray")
plt.scatter(w, w_hat, s=10)
plt.ylabel("Learned coefficients")
plt.xlabel("True coefficients")
plt.show()
```


3
2
1
0
−1
−2
−3
0
100
200
300
400
500
600
700



-

Total running time of the script: (0 minutes 4.421 seconds)

4.2.5 GroupLasso for linear regression with dummy variables

A sample script for group lasso with dummy variables

Setup

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder

from group_lasso import GroupLasso
from group_lasso.utils import extract_ohe_groups

np.random.seed(42)
GroupLasso.LOG_LOSSES = True
```

Set dataset parameters

```
num_categories = 30
min_options = 2
max_options = 10
num_datapoints = 10000
noise_std = 1
```

Generate data matrix

```
X_cat = np.empty((num_datapoints, num_categories))
for i in range(num_categories):
    X_cat[:, i] = np.random.randint(min_options, max_options, num_datapoints)

ohe = OneHotEncoder()
X = ohe.fit_transform(X_cat)
groups = extract_ohe_groups(ohe)
group_sizes = [np.sum(groups == g) for g in np.unique(groups)]
active_groups = [np.random.randint(0, 2) for _ in np.unique(groups)]
```

Generate coefficients

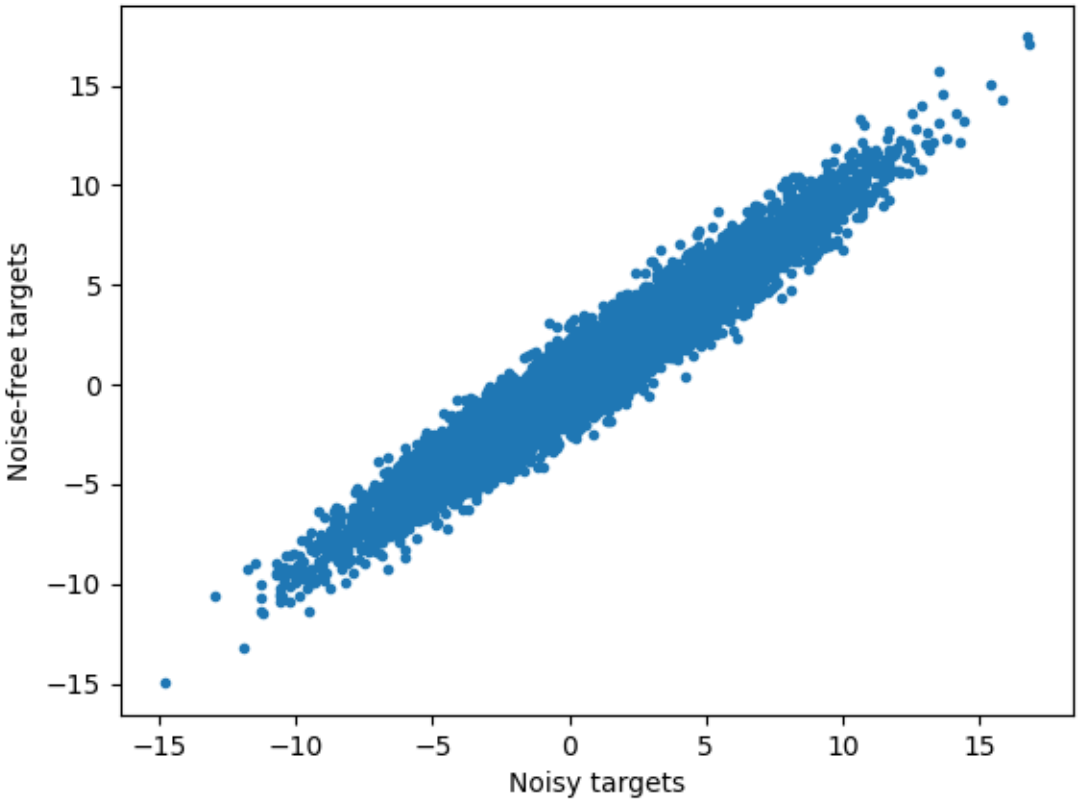
```
w = np.concatenate(
    [
        np.random.standard_normal(group_size) * is_active
        for group_size, is_active in zip(group_sizes, active_groups)
    ]
)
w = w.reshape(-1, 1)
true_coefficient_mask = w != 0
intercept = 2
```

Generate regression targets

```
y_true = X @ w + intercept
y = y_true + np.random.randn(*y_true.shape) * noise_std
```

View noisy data and compute maximum R^2

```
plt.figure()
plt.plot(y, y_true, ".")
plt.xlabel("Noisy targets")
plt.ylabel("Noise-free targets")
# Use noisy y as true because that is what we would have access
# to in a real-life setting.
R2_best = r2_score(y, y_true)
```



Generate pipeline and train it

```
pipe = pipe = Pipeline(
    memory=None,
    steps=[
        (
            "variable_selection",
            GroupLasso(
                groups=groups,
                group_reg=0.1,
                l1_reg=0,
                scale_reg=None,
                supress_warning=True,
                n_iter=100000,
                frobenius_lipschitz=False,
            ),
        ),
        ("regressor", Ridge(alpha=1)),
    ],
)
pipe.fit(X, y)
```

Out:

```
Pipeline(steps=[('variable_selection',
                 GroupLasso(group_reg=0.1,
                            groups=array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,
↪  1.,  1.,  1.,  1.,
       1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  3.,  3.,
       3.,  3.,  3.,  3.,  3.,  3.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,
       4.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  6.,  6.,  6.,  6.,
       6.,  6.,  6.,  6.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  8.,
       8.,  8.,  8.,  8.,  8.,  8.,  8.,  9.,  9.,  9.,  9.,  9.,  9.,
       9.,  9., 10., 10., 10., 10., 10., 10., 10., 10., 1...
      21., 21., 21., 21., 21., 21., 21., 22., 22., 22., 22., 22., 22.,
      22., 22., 23., 23., 23., 23., 23., 23., 23., 23., 24., 24., 24.,
      24., 24., 24., 24., 24., 25., 25., 25., 25., 25., 25., 25., 25.,
      26., 26., 26., 26., 26., 26., 26., 26., 27., 27., 27., 27., 27.,
      27., 27., 27., 28., 28., 28., 28., 28., 28., 28., 28., 29., 29.,
      29., 29., 29., 29., 29., 29.]),
                            l1_reg=0, n_iter=100000, scale_reg=None,
                            supress_warning=True)),
                ('regressor', Ridge(alpha=1))])
```

Extract results and compute performance metrics

```
# Extract from pipeline
yhat = pipe.predict(X)
sparsity_mask = pipe["variable_selection"].sparsity_mask_
coef = pipe["regressor"].coef_.T

# Construct full coefficient vector
w_hat = np.zeros_like(w)
w_hat[sparsity_mask] = coef

R2 = r2_score(y, yhat)

# Print performance metrics
print(f"Number variables: {len(sparsity_mask)}")
print(f"Number of chosen variables: {sparsity_mask.sum()}")
print(f"R^2: {R2}, best possible R^2 = {R2_best}")
```

Out:

```
Number variables: 240
Number of chosen variables: 144
R^2: 0.9278329616431523, best possible R^2 = 0.9394648554757948
```

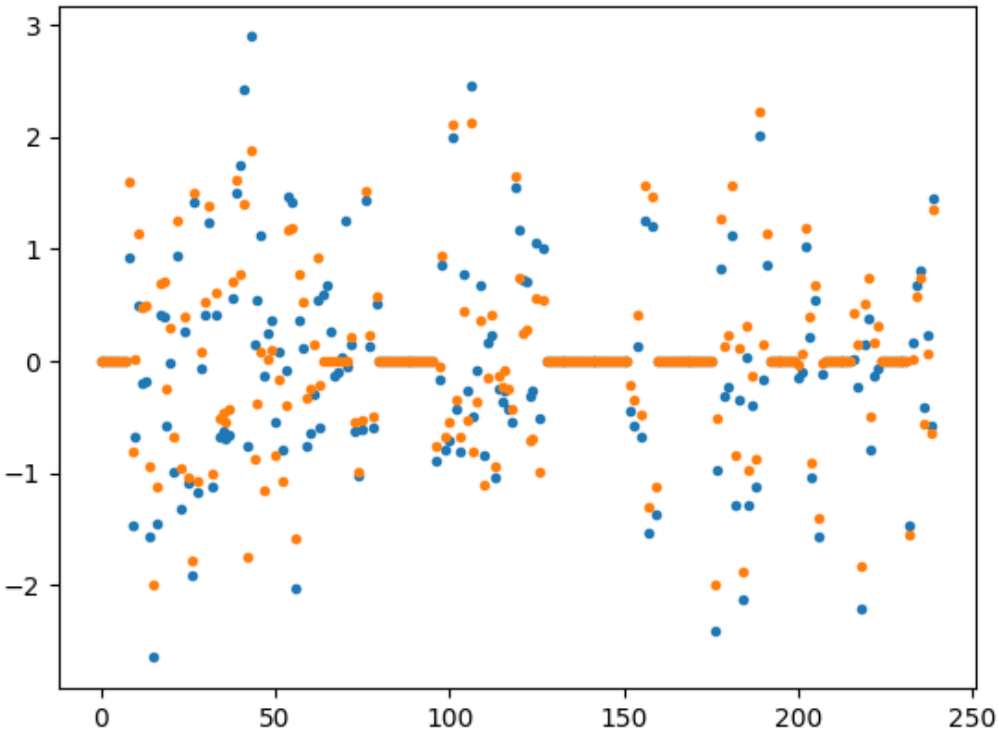
Visualise regression coefficients

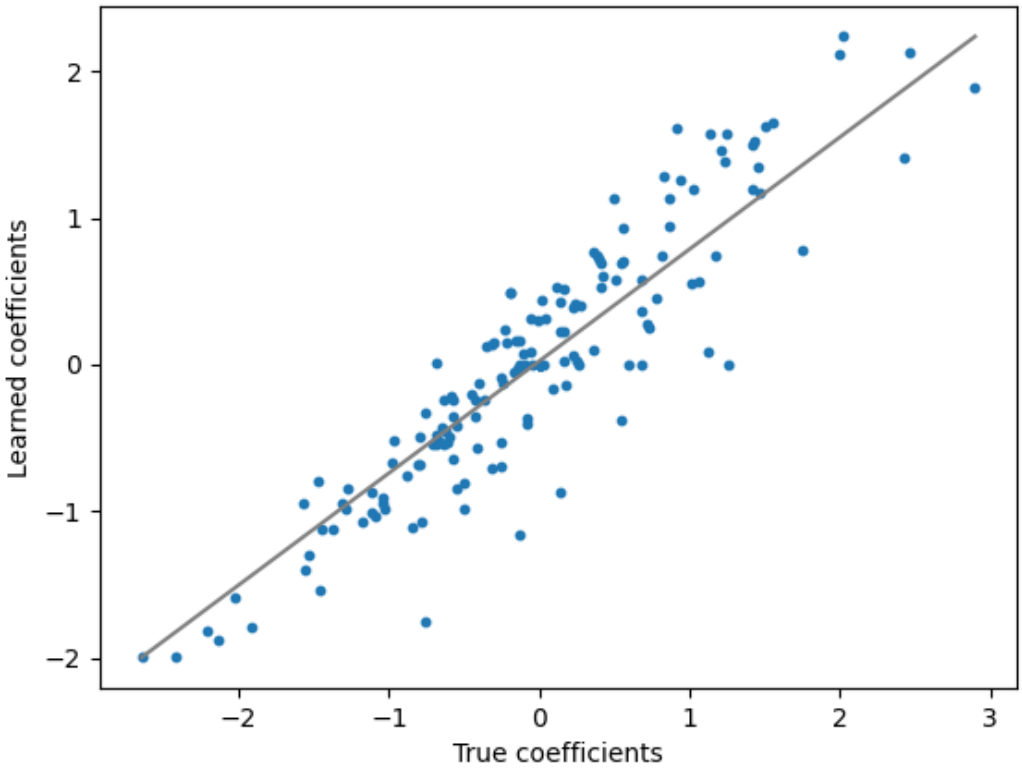
```
for i in range(w.shape[1]):
    plt.figure()
    plt.plot(w[:, i], ".", label="True weights")
    plt.plot(w_hat[:, i], ".", label="Estimated weights")

plt.figure()
plt.plot([w.min(), w.max()], [coef.min(), coef.max()], "gray")
plt.scatter(w, w_hat, s=10)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Learned coefficients")
plt.xlabel("True coefficients")
plt.show()
```





Total running time of the script: (0 minutes 18.205 seconds)

4.3 Mathematical background

4.3.1 Quick overview

Let us recap the definition of a sparse group lasso regularised machine learning algorithm. Consider the unregularised loss function $L(\beta; \mathbf{X}, \mathbf{y})$, where β is the model coefficients, $\mathbf{X}$ is the data matrix and $\mathbf{y}$ is the target vector (or matrix in the case of multiple regression/classification algorithms). Furthermore, we assume that $\beta = \left[\beta_1^T, ..., \beta_G^T\right]^T$ and that $\mathbf{X} = \left[\mathbf{X}_1^T, ..., \mathbf{X}_G^T\right]^T$, where β_g and $\mathbf{X}_g$ is the coefficients and data matrices corresponding to covariate group g. In this case, we define the group lasso regularised loss function as

$$L(\beta; \mathbf{X}, \mathbf{y}) + \lambda_1 ||\beta||_1 + \lambda_2 \sum_{g \in \mathcal{G}} \sqrt{d_g} ||\beta_g||_2$$

where λ_1 is the parameter-wise regularisation penalty, λ_2 is the group-wise regularisation penalty, $\beta_g \in \mathbf{d_g}$ and $\mathcal{G}$ is the set of all groups.

The above regularisation penalty is nice in the sense that it promotes that a sparse set of groups are chosen for the regularisation coefficients[1]. However, the non-continuous derivative makes the optimisation procedure much more complicated than with say a Ridge penalty (i.e. squared 2-norm penalty). One common algorithm used to solve this optimisation problem is *group coordinate descent*, in which the optimisation problem is solved for each group

[1] Yuan M, Lin Y. Model selection and estimation in regression with grouped variables. Journal of the Royal Statistical Society: Series B (Statistical Methodology). 2006 Feb;68(1):49-67.

separately, in an alternating fashion. However, I decided to use the fast iterative soft thresholding (FISTA) algorithm[2] with the gradient-based restarting scheme given in[3]. This is regarded as one of the best algorithms to solve optimisation problems on the form

$$\arg\min_{\beta} L(\beta) + R(\beta),$$

where L is a convex, differentiable function with Lipschitz continuous gradient and R is a convex lower semicontinouous function.

4.3.2 Details on FISTA

There are three essential parts of having an efficient implementation of the FISTA algorithm. First and foremost, we need an efficient way to compute the gradient of the loss function. Next, and just as important, we need to be able to compute the *proximal map* of the regulariser efficiently. That is, we need to know how to compute

$$prox(\beta) = \arg\min_{\hat{\beta}} R(\hat{\beta}) + \frac{1}{2}||\hat{\beta} - \beta||_2^2$$

efficiently. To compute the proximal map for the sparse group lasso regulariser, we use the following identity from[4]:

$$prox_{\lambda_1||\cdot||_1 + \lambda_2 \sum_g w_g||\cdot||}(\beta) = prox_{\lambda_2 \sum_g w_g||\cdot||}(prox_{\lambda_1||\cdot||_1}(\beta),$$

where $prox_{\lambda_1||\cdot||_1 + \lambda_2 \sum_g w_g||\cdot||}$ is the proximal map for the sparse group lasso regulariser, $prox_{\lambda_2 \sum_g w_g||\cdot||}$ is the proximal map for the group lasso regulariser and $prox_{\lambda_1||\cdot||_1}$ is the proximal map for the lasso regulariser. For more information on the proximal map, see[5] or[6]. Finally, we need a Lipschitz bound for the gradient of the loss function, since this is used to compute the step-length of the optimisation procedure. Luckily, this can also be estimated using a line-search.

Unfortunately, the FISTA algorithm is not stable in the mini-batch case, making it inefficient for extremely large datasets. However, in my experiments, I have found that it still recovers the correct sparsity patterns in the data when used in a mini-batch fashion for the group lasso problem. At least so long as the mini-batches are relatively large.

4.3.3 Computing the Lipschitz coefficients

The Lipschitz coefficient of the gradient to the sum-of-squares loss is given by σ_1^2, where σ_1 is the largest singular value of the data matrix.

For logistic regression, we use the line search algorithm presented in[2] to estimate the Lipschitz bound, with an initial guess given by the Frobenius norm of the data matrix.

[2] Beck A, Teboulle M. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM journal on imaging sciences. 2009 Mar 4;2(1):183-202.

[3] O'Donoghue B, Candes E. Adaptive restart for accelerated gradient schemes. Foundations of computational mathematics. 2015 Jun 1;15(3):715-32.

[4] Yuan L, Liu J, Ye J. (2011), Efficient methods for overlapping group lasso. Advances in Neural Information Processing Systems (pp. 352-360).

[5] Parikh, N., & Boyd, S. (2014). Proximal algorithms. Foundations and Trends in Optimization, 1(3), 127-239.

[6] Beck, A. (2017). First-order methods in optimization (Vol. 25). SIAM.

4.3.4 References

4.4 API Reference

4.4.1 Group Lasso regularised estimators

class group_lasso.**GroupLasso**(*groups=None*, *group_reg=0.05*, *l1_reg=0.05*, *n_iter=100*, *tol=1e-05*, *scale_reg='group_size'*, *subsampling_scheme=None*, *fit_intercept=True*, *frobenius_lipschitz=False*, *random_state=None*, *warm_start=False*, *old_regularisation=False*, *supress_warning=False*)

Sparse group lasso regularised least squares linear regression.

This class implements the Sparse Group Lasso [1] regularisation for linear regression with the mean squared penalty.

This class is implemented as both a regressor and a transformation. If the `transform` method is called, then the columns of the input that correspond to zero-valued regression coefficients are dropped.

The loss is optimised using the FISTA algorithm proposed in [2] with the generalised gradient-based restarting scheme proposed in [3]. This algorithm is not as accurate as a few other optimisation algorithms, but it is extremely efficient and does recover the sparsity patterns. We therefore reccomend that this class is used as a transformer to select the viable features and that the output is fed into another regression algorithm, such as RidgeRegression in scikit-learn.

Parameters

- **groups** (*Iterable*) – Iterable that specifies which group each column corresponds to. For columns that should not be regularised, the corresponding group index should either be None or negative. For example, the list `[1, 1, 1, 2, 2, -1]` specifies that the first three columns of the data matrix belong to the first group, the next two columns belong to the second group and the last column should not be regularised.
- **group_reg** (*float or iterable [default=0.05]*) – The regularisation coefficient(s) for the group sparsity penalty. If `group_reg` is an iterable, then its length should be equal to the number of groups.
- **l1_reg** (*float or iterable [default=0.05]*) – The regularisation coefficient for the coefficient sparsity penalty.
- **n_iter** (*int [default=100]*) – The maximum number of iterations to perform
- **tol** (*float [default=1e-5]*) – The convergence tolerance. The optimisation algorithm will stop once ||x_{n+1} - x_n|| < `tol`.
- **scale_reg** (*str [in {"group_size", "none", "inverse_group_size"] or None*) – How to scale the group-wise regularisation coefficients. In the original group lasso paper scaled the regularisation by the square root of the elements in each group so that each variable has the same effect on the regularisation. This is not sensible for dummy encoded variables, as these always have either unit or zero norm. `scale_reg` should therefore be None if all variables are dummy variables. Finally, if the group size shouldn't be considered when choosing variables, then inverse_group_size should be used instead as that divide by the square root of the group size, removing the dependence of group size on the regularisation strength.
- **subsampling_scheme** (*None, float, int or str [default=None]*) – The subsampling rate used for the gradient and singular value computations. If it is a float, then it specifies the fraction of rows to use in the computations. If it is an int, it specifies the

number of rows to use in the computation and if it is a string, then it must be 'sqrt' and the number of rows used in the computations is the square root of the number of rows in X.

- **frobenius_lipschitz** (*bool [default=False]*) – Use the Frobenius norm to estimate the lipschitz coefficient of the MSE loss. This works well for systems whose power iterations converge slowly. If False, then subsampled power iterations are used. Using the Frobenius approximation for the Lipschitz coefficient might fail, and end up with all-zero weights.
- **fit_intercept** (*bool [default=True]*) – Whether to fit an intercept or not.
- **random_state** (*np.random.RandomState [default=None]*) – The random state used for initialisation of parameters.
- **warm_start** (*bool [default=False]*) – If true, then subsequent calls to fit will not re-initialise the model parameters. This can speed up the hyperparameter search

References

[1] Simon, N., Friedman, J., Hastie, T., & Tibshirani, R. (2013). A sparse-group lasso. Journal of Computational and Graphical Statistics, 22(2), 231-245.

[2] Beck A, Teboulle M. (2009). A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM journal on imaging sciences. 2009 Mar 4;2(1):183-202.

[3] O'Donoghue B, Candes E. (2015) Adaptive restart for accelerated gradient schemes. Foundations of computational mathematics. Jun 1;15(3):715-32

Attributes

chosen_groups_ A set of the coosen group ids.

sparsity_mask A boolean mask indicating whether features are used in prediction.

sparsity_mask_ A boolean mask indicating whether features are used in prediction.

Methods

fit(X, y[, lipschitz])	Fit a group lasso regularised linear regression model.
fit_transform(X, y[, lipschitz])	Fit a group lasso model to X and y and remove unused columns from X
get_params([deep])	Get parameters for this estimator.
loss(X, y)	The group-lasso regularised loss with the current coefficients
predict(X)	Predict using the linear model.
score(X, y[, sample_weight])	Return the coefficient of determination of the prediction.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Remove columns corresponding to zero-valued coefficients.

fit_predict	

chosen_groups_
A set of the coosen group ids.

fit (*X*, *y*, *lipschitz=None*)
Fit a group lasso regularised linear regression model.

Parameters

- **X** (*np.ndarray*) – Data matrix
- **y** (*np.ndarray*) – Target vector or matrix
- **lipschitz** (*float or None [default=None]*) – A Lipshitz bound for the mean squared loss with the given data and target matrices. If None, this is estimated.

fit_transform (*X*, *y*, *lipschitz=None*)
Fit a group lasso model to X and y and remove unused columns from X

get_params (*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

loss (*X*, *y*)
The group-lasso regularised loss with the current coefficients

Parameters

- **X** (*np.ndarray*) – Data matrix, `X.shape == (num_datapoints, num_features)`
- **y** (*np.ndarray*) – Target vector/matrix, `y.shape == (num_datapoints, num_targets)`, or `y.shape == (num_datapoints,)`

predict (*X*)
Predict using the linear model.

score (*X*, *y*, *sample_weight=None*)
Return the coefficient of determination of the prediction.

The coefficient of determination R^2 is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns **score** – R^2 of `self.predict(X)` wrt. *y*.

Return type float

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

sparsity_mask
A boolean mask indicating whether features are used in prediction.

sparsity_mask_
A boolean mask indicating whether features are used in prediction.

transform(*X*)
Remove columns corresponding to zero-valued coefficients.

class group_lasso.**LogisticGroupLasso**(*groups=None*, *group_reg=0.05*, *l1_reg=0.05*, *n_iter=100*, *tol=1e-05*, *scale_reg='group_size'*, *subsampling_scheme=None*, *fit_intercept=True*, *random_state=None*, *warm_start=False*, *old_regularisation=False*, *supress_warning=False*)
Sparse group lasso regularised multi-class logistic regression.

This class implements the Sparse Group Lasso [1] regularisation for multi-class logistic regression with a cross entropy penalty.

This class is implemented as both a regressor and a transformation. If the `transform` method is called, then the columns of the input that correspond to zero-valued regression coefficients are dropped.

The loss is optimised using the FISTA algorithm proposed in [2] with the generalised gradient-based restarting scheme proposed in [3]. This algorithm is not as accurate as a few other optimisation algorithms, but it is extremely efficient and does recover the sparsity patterns. We therefore reccomend that this class is used as a transformer to select the viable features and that the output is fed into another classification algorithm, such as LogisticRegression in scikit-learn.

Parameters

- **groups** (*Iterable*) – Iterable that specifies which group each column corresponds to. For columns that should not be regularised, the corresponding group index should either be None or negative. For example, the list `[1, 1, 1, 2, 2, -1]` specifies that the first three columns of the data matrix belong to the first group, the next two columns belong to the second group and the last column should not be regularised.
- **group_reg** (*float or iterable [default=0.05]*) – The regularisation coefficient(s) for the group sparsity penalty. If `group_reg` is an iterable, then its length should be equal to the number of groups.
- **l1_reg** (*float or iterable [default=0.05]*) – The regularisation coefficient for the coefficient sparsity penalty.
- **n_iter** (*int [default=100]*) – The maximum number of iterations to perform

- **tol** (*float [default=1e-5]*) – The convergence tolerance. The optimisation algorithm will stop once ||x_{n+1} - x_n|| < `tol`.
- **scale_reg** (*str [in {"group_size", "none", "inverse_group_size"] or None*) – How to scale the group-wise regularisation coefficients. In the original group lasso paper scaled the regularisation by the square root of the elements in each group so that each variable has the same effect on the regularisation. This is not sensible for dummy encoded variables, as these always have either unit or zero norm. `scale_reg` should therefore be None if all variables are dummy variables. Finally, if the group size shouldn't be considered when choosing variables, then inverse_group_size should be used instead as that divide by the square root of the group size, removing the dependence of group size on the regularisation strength.
- **subsampling_scheme** (*None, float, int or str [default=None]*) – The subsampling rate used for the gradient and singular value computations. If it is a float, then it specifies the fraction of rows to use in the computations. If it is an int, it specifies the number of rows to use in the computation and if it is a string, then it must be 'sqrt' and the number of rows used in the computations is the square root of the number of rows in X.
- **frobenius_lipschitz** (*bool [default=False]*) – Use the Frobenius norm to estimate the lipschitz coefficient of the MSE loss. This works well for systems whose power iterations converge slowly. If False, then subsampled power iterations are used. Using the Frobenius approximation for the Lipschitz coefficient might fail, and end up with all-zero weights.
- **fit_intercept** (*bool [default=True]*) – Whether to fit an intercept or not.
- **random_state** (*np.random.RandomState [default=None]*) – The random state used for initialisation of parameters.
- **warm_start** (*bool [default=False]*) – If true, then subsequent calls to fit will not re-initialise the model parameters. This can speed up the hyperparameter search

References

[1] Simon, N., Friedman, J., Hastie, T., & Tibshirani, R. (2013). A sparse-group lasso. Journal of Computational and Graphical Statistics, 22(2), 231-245.

[2] Beck A, Teboulle M. (2009). A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM journal on imaging sciences. 2009 Mar 4;2(1):183-202.

[3] O'Donoghue B, Candes E. (2015) Adaptive restart for accelerated gradient schemes. Foundations of computational mathematics. Jun 1;15(3):715-32

Attributes

chosen_groups_ A set of the coosen group ids.

sparsity_mask A boolean mask indicating whether features are used in prediction.

sparsity_mask_ A boolean mask indicating whether features are used in prediction.

Methods

fit(X, y[, lipschitz])	Fit a group-lasso regularised linear model.
fit_transform(X, y[, lipschitz])	Fit a group lasso model to X and y and remove unused columns from X

Continued on next page

Table 2 – continued from previous page

get_params([deep])	Get parameters for this estimator.
loss(X, y)	The group-lasso regularised loss with the current coefficients
predict(X)	Predict using the linear model.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Remove columns corresponding to zero-valued coefficients.

fit_predict	
predict_proba	

chosen_groups_
A set of the coosen group ids.

fit(*X*, *y*, *lipschitz=None*)
Fit a group-lasso regularised linear model.

fit_transform(*X*, *y*, *lipschitz=None*)
Fit a group lasso model to X and y and remove unused columns from X

get_params(*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

loss(*X*, *y*)
The group-lasso regularised loss with the current coefficients

Parameters

- **X** (*np.ndarray*) – Data matrix, `X.shape == (num_datapoints, num_features)`
- **y** (*np.ndarray*) – Target vector/matrix, `y.shape == (num_datapoints, num_targets)`, or `y.shape == (num_datapoints,)`

predict(*X*)
Predict using the linear model.

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.

- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns **score** – Mean accuracy of `self.predict(X)` wrt. *y*.

Return type float

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

sparsity_mask
A boolean mask indicating whether features are used in prediction.

sparsity_mask_
A boolean mask indicating whether features are used in prediction.

transform(*X*)
Remove columns corresponding to zero-valued coefficients.

4.4.2 Utilities for group lasso

group_lasso.utils.**extract_ohe_groups**(*onehot_encoder*)
Extract a vector with group indices from a scikit-learn OneHotEncoder

Parameters **onehot_encoder** (*sklearn.preprocessing.OneHotEncoder*) –

Returns A group-vector that can be used with the group lasso regularised linear models.

Return type np.ndarray

CHAPTER 5

References

Python Module Index

g

Index