1.1. Generalized Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the input variables. In mathematical notion, if $\hat{y}$ is the predicted value.

»

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

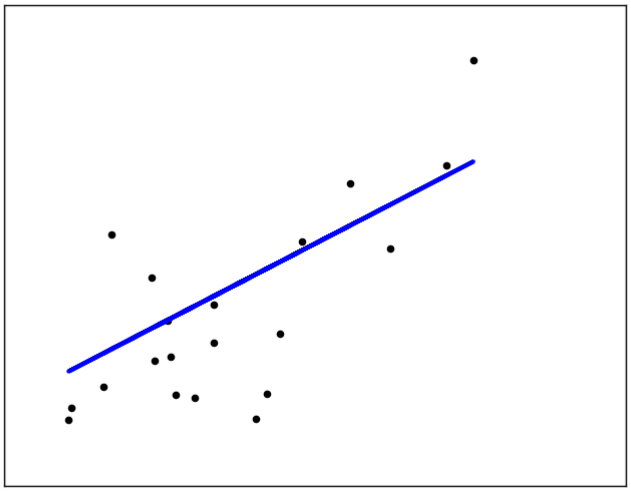
Across the module, we designate the vector $w = (w_1, ..., w_p)$ as `coef_` and w_0 as `intercept_`.

To perform classification with generalized linear models, see Logistic regression.

1.1.1. Ordinary Least Squares

`LinearRegression` fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed responses in the dataset, and the responses predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\underset{w}{min\,} {|| X w - y||_2}^2$$



`LinearRegression` will take in its `fit` method arrays X, y and will store the coefficients w of the linear model in its `coef_` member:

```
>>> from sklearn import linear_model
>>> reg = linear_model.LinearRegression()
>>> reg.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
>>> reg.coef_
array([ 0.5,  0.5])
```

However, coefficient estimates for Ordinary Least Squares rely on the independence of the model terms. When terms are correlated and the columns of the design matrix X have an approximate linear dependence, the design matrix becomes close to singular and as a result, the least-squares estimate becomes highly sensitive to random errors in the observed response, producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

»

Examples:

- Linear Regression Example

1.1.1.1. Ordinary Least Squares Complexity

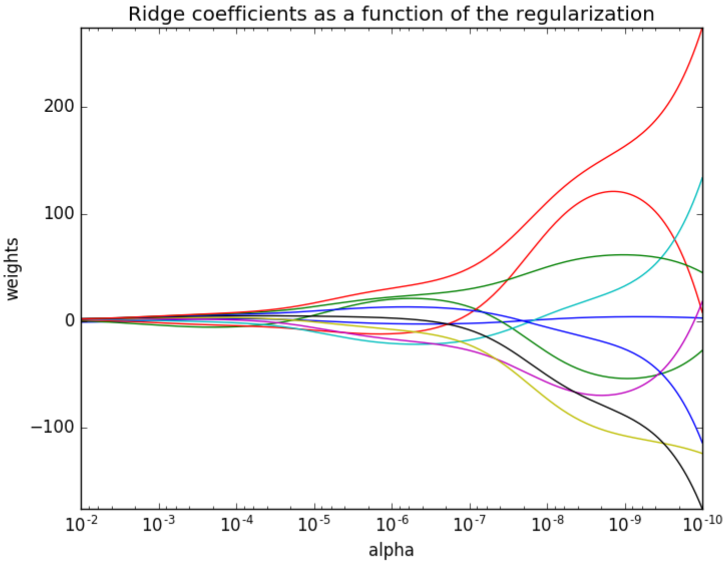
This method computes the least squares solution using a singular value decomposition of X. If X is a matrix of size (n, p) this method has a cost of $O(np^2)$, assuming that $n \geq p$.

1.1.2. Ridge Regression

`Ridge` regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of coefficients. The ridge coefficients minimize a penalized residual sum of squares,

$$\underset{w}{min\,} {{|| X w - y||_2}^2 + \alpha {||w||_2}^2}$$

Here, $\alpha \geq 0$ is a complexity parameter that controls the amount of shrinkage: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.



As with other linear models, `Ridge` will take in its `fit` method arrays X, y and will store the coefficients w of the linear model in its `coef_` member:

```
>>> from sklearn import linear_model
>>> reg = linear_model.Ridge (alpha = .5)
>>> reg.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
>>> reg.coef_
array([ 0.34545455,  0.34545455])
>>> reg.intercept_
0.13636...
```

Examples:

- Plot Ridge coefficients as a function of the regularization
- Classification of text documents using sparse features

1.1.2.1. Ridge Complexity

This method has the same order of complexity than an Ordinary Least Squares.

1.1.2.2. Setting the regularization parameter: generalized Cross-Validation

`RidgeCV` implements ridge regression with built-in cross-validation of the alpha parameter. The object works in the same way as GridSearchCV except that it defaults to Generalized Cross-Validation (GCV), an efficient form of leave-one-out cross-validation:

```
>>> from sklearn import linear_model
>>> reg = linear_model.RidgeCV(alphas=[0.1, 1.0, 10.0])
>>> reg.fit([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
RidgeCV(alphas=[0.1, 1.0, 10.0], cv=None, fit_intercept=True, scoring=None,
    normalize=False)
>>> reg.alpha_
0.1
```

References

- “Notes on Regularized Least Squares”, Rifkin & Lippert (technical report, course slides).

1.1.3. Lasso

The `Lasso` is a linear model that estimates sparse coefficients. It is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights (see Compressive sensing: tomography reconstruction with L1 prior (Lasso)).

Mathematically, it consists of a linear model trained with ℓ_1 prior as regularizer. The objective function to minimize is:

$$\underset{w}{min} \frac{1}{2n_{samples}} ||Xw - y||_2^2 + \alpha ||w||_1$$

The lasso estimate thus solves the minimization of the least-squares penalty with $\alpha ||w||_1$ added, where α is a constant and $||w||_1$ is the ℓ_1-norm of the parameter vector.

The implementation in the class **Lasso** uses coordinate descent as the algorithm to fit the coefficients. See Least Angle Regression for another implementation:

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lasso(alpha = 0.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
>>> reg.predict([[1, 1]])
array([ 0.8])
```

Also useful for lower-level tasks is the function **lasso_path** that computes the coefficients along the full path of possible values.

Examples:

- Lasso and Elastic Net for Sparse Signals
- Compressive sensing: tomography reconstruction with L1 prior (Lasso)

Note: Feature selection with Lasso

As the Lasso regression yields sparse models, it can thus be used to perform feature selection, as detailed in L1-based feature selection.

Note: Randomized sparsity

For feature selection or sparse recovery, it may be interesting to use Randomized sparse models.

1.1.3.1. Setting regularization parameter

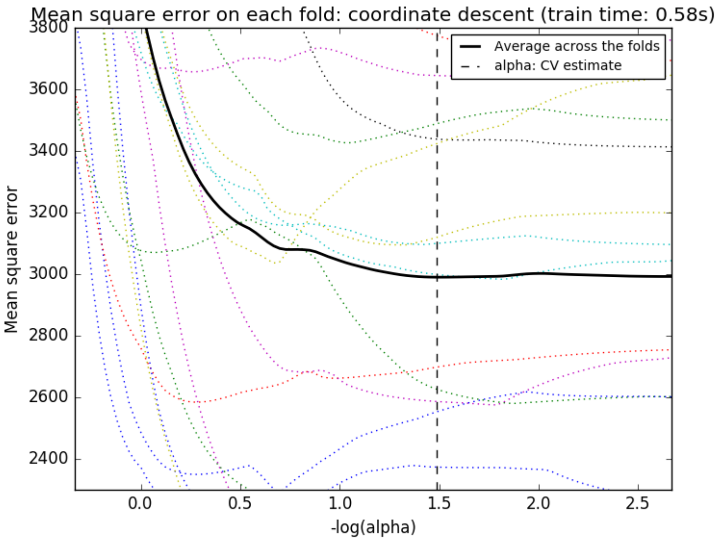
The `alpha` parameter controls the degree of sparsity of the coefficients estimated.

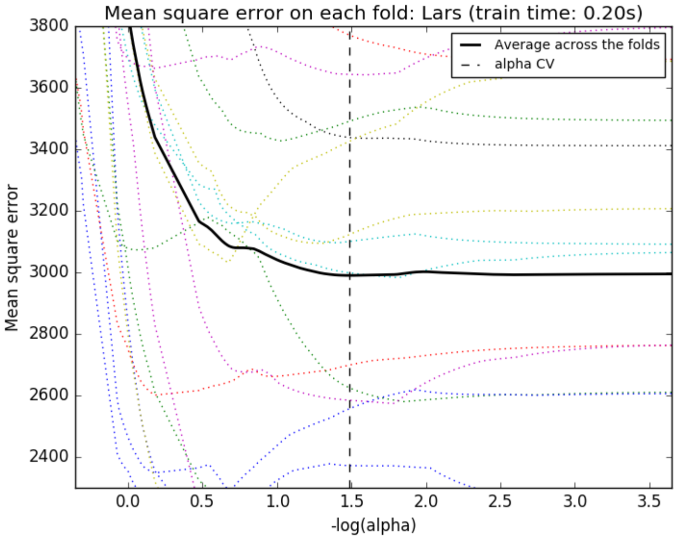
1.1.3.1.1. Using cross-validation

scikit-learn exposes objects that set the Lasso `alpha` parameter by cross-validation: **LassoCV** and **LassoLarsCV**. **LassoLarsCV** is based on the Least Angle Regression algorithm explained below.

For high-dimensional datasets with many collinear regressors, **LassoCV** is most often preferable. However, **LassoLarsCV** has the advantage of exploring more relevant values of *alpha* parameter, and if the number of samples is very small compared to the number of observations, it is often faster than **LassoCV**.

»

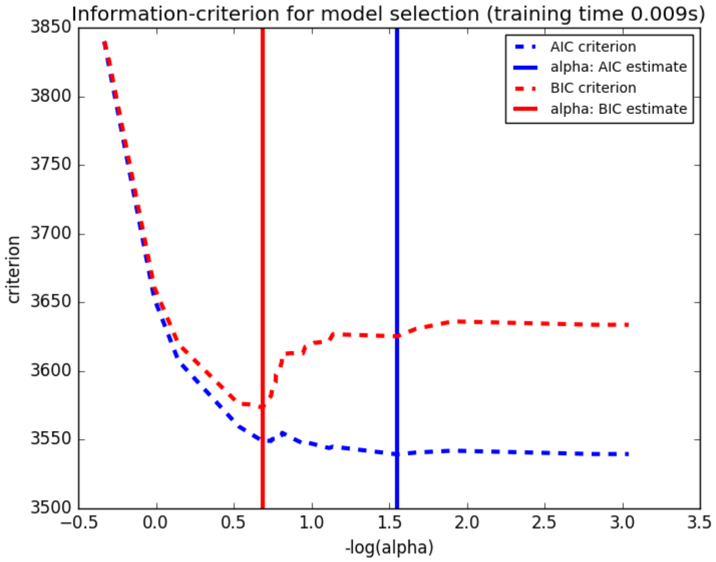




1.1.3.1.2. Information-criteria based model selection

Alternatively, the estimator `LassoLarsIC` proposes to use the Akaike information criterion (AIC) and the Bayes Information criterion (BIC). It is a computationally cheaper alternative to find the optimal value of alpha as the regularization path is computed only once instead of k+1 times when using k-fold cross-validation. However, such criteria needs a proper estimation of the degrees of freedom of the solution, are derived for large samples (asymptotic results) and assume the model is correct, i.e. that the data are actually generated by this model. They also tend to break when the problem is badly conditioned (more features than samples).

»



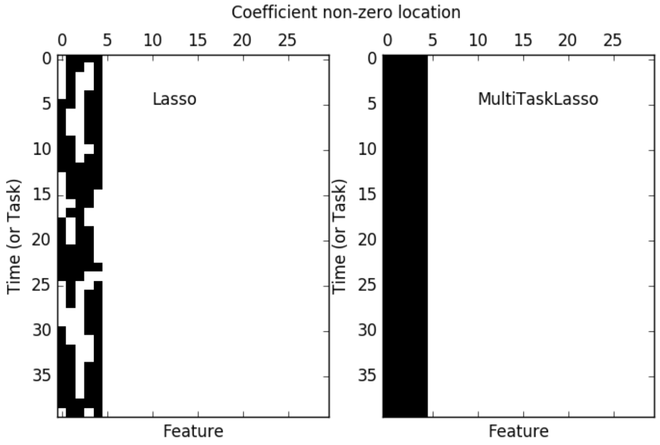
Examples:

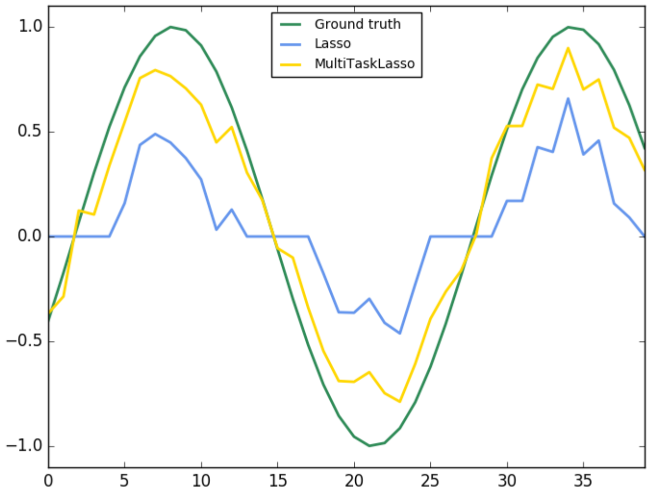
- Lasso model selection: Cross-Validation / AIC / BIC

1.1.4. Multi-task Lasso

The `MultiTaskLasso` is a linear model that estimates sparse coefficients for multiple regression problems jointly: `y` is a 2D array, of shape `(n_samples, n_tasks)`. The constraint is that the selected features are the same for all the regression problems, also called tasks.

The following figure compares the location of the non-zeros in W obtained with a simple Lasso or a MultiTaskLasso. The Lasso estimates yields scattered non-zeros while the non-zeros of the MultiTaskLasso are full columns.





Fitting a time-series model, imposing that any active feature be active at all times.

Examples:

- Joint feature selection with multi-task Lasso

Mathematically, it consists of a linear model trained with a mixed ℓ_1 ℓ_2 prior as regularizer. The objective function to minimize is:

$$\underset{w}{min\,} { \frac{1}{2n_{samples}} ||X W - Y||_{Fro} ^ 2 + \alpha ||W||_{21}}$$

where Fro indicates the Frobenius norm:

$$||A||_{Fro} = \sqrt{\sum_{ij} a_{ij}^2}$$

and ℓ_1 ℓ_2 reads:

$$||A||_{21} = \sum_i \sqrt{\sum_j a_{ij}^2}$$

The implementation in the class `MultiTaskLasso` uses coordinate descent as the algorithm to fit the coefficients.

1.1.5. Elastic Net

»

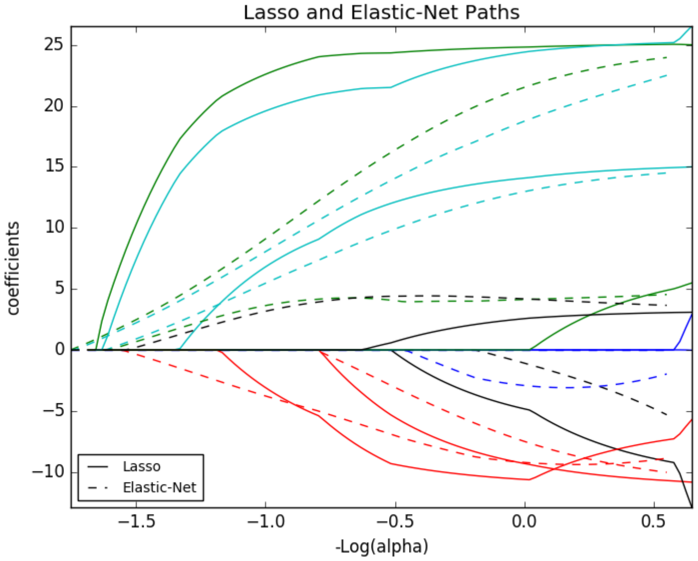
`ElasticNet` is a linear regression model trained with L1 and L2 prior as regularizer. This combination allows for learning a sparse model where few of the weights are non-zero like `Lasso`, while still maintaining the regularization properties of `Ridge`. We control the convex combination of L1 and L2 using the `l1_ratio` parameter.

Elastic-net is useful when there are multiple features which are correlated with one another. Lasso is likely to pick one of these at random, while elastic-net is likely to pick both.

A practical advantage of trading-off between Lasso and Ridge is it allows Elastic-Net to inherit some of Ridge's stability under rotation.

The objective function to minimize is in this case

$$\underset{w}{min\,} { \frac{1}{2n_{samples}} ||X w - y||_2 ^ 2 + \alpha \rho ||w||_1 + \frac{\alpha(1-\rho)}{2} ||w||_2 ^ 2}$$



The class `ElasticNetCV` can be used to set the parameters `alpha` (α) and `l1_ratio` (ρ) by cross-validation.

Examples:

- Lasso and Elastic Net for Sparse Signals
- Lasso and Elastic Net

1.1.6. Multi-task Elastic Net

The `MultiTaskElasticNet` is an elastic-net model that estimates sparse coefficients for multiple regression problems jointly: `Y` is a 2D array, of shape `(n_samples, n_tasks)`. The constraint is that the selected features are the same for all the regression problems, also called tasks.

Mathematically, it consists of a linear model trained with a mixed ℓ_1 ℓ_2 prior and ℓ_2 prior as regularizer. The objective function to minimize is:

$$\underset{W}{min} \frac{1}{2n_{samples}} ||XW - Y||_{Fro}^2 + \alpha\rho||W||_{21} + \frac{\alpha(1-\rho)}{2} ||W||_{Fro}^2$$

The implementation in the class `MultiTaskElasticNet` uses coordinate descent as the algorithm to fit the coefficients.

The class `MultiTaskElasticNetCV` can be used to set the parameters `alpha` (α) and `l1_ratio` (ρ) by cross-validation.

1.1.7. Least Angle Regression

Least-angle regression (LARS) is a regression algorithm for high-dimensional data, developed by Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani.

The advantages of LARS are:

- It is numerically efficient in contexts where p >> n (i.e., when the number of dimensions is significantly greater than the number of points)
- It is computationally just as fast as forward selection and has the same order of complexity as an ordinary least squares.
- It produces a full piecewise linear solution path, which is useful in cross-validation or similar attempts to tune the model.
- If two variables are almost equally correlated with the response, then their coefficients should increase at approximately the same rate. The algorithm thus behaves as intuition would expect, and also is more stable.
- It is easily modified to produce solutions for other estimators, like the Lasso.

The disadvantages of the LARS method include:

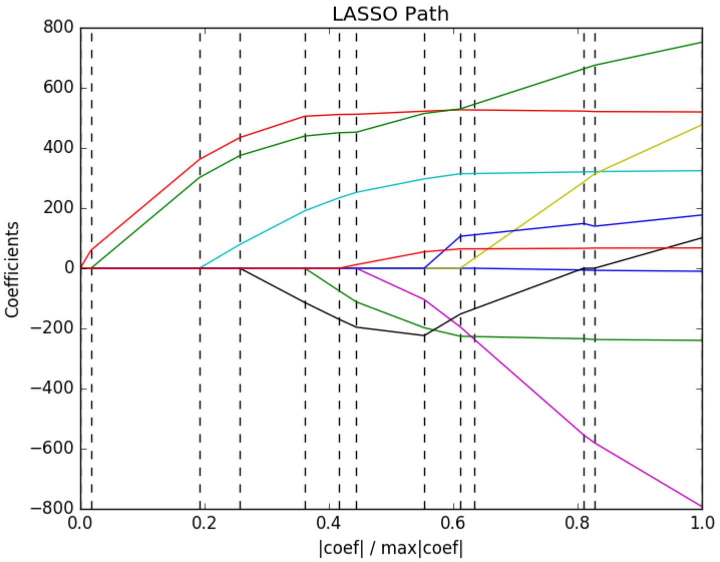
- Because LARS is based upon an iterative refitting of the residuals, it would appear to be especially sensitive to the effects of noise. This problem is discussed in detail by Weisberg in the discussion section of the Efron et al. (2004) Annals of Statistics article.

The LARS model can be used using estimator `Lars`, or its low-level implementation `lars_path`.

1.1.8. LARS Lasso

`LassoLars` is a lasso model implemented using the LARS algorithm, and unlike the implementation based on coordinate_descent, this yields the exact solution, which is piecewise linear as a function of the norm of its coefficients.

»



>>>

```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLars(alpha=.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
LassoLars(alpha=0.1, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, positive=False,
     precompute='auto', verbose=False)
>>> reg.coef_
array([ 0.717157...,  0.        ])
```

Examples:

- Lasso path using LARS

The Lars algorithm provides the full path of the coefficients along the regularization parameter almost for free, thus a common operation consist of retrieving the path with function `lars_path`

1.1.8.1. Mathematical formulation

The algorithm is similar to forward stepwise regression, but instead of including variables at each step, the estimated parameters are increased in a direction equiangular to each one's correlations with the residual.

Instead of giving a vector result, the LARS solution consists of a curve denoting the solution for each value of the L1 norm of the parameter vector. The full coefficients path is stored in the array `coef_path_`, which has size (n_features, max_features+1). The first column is always zero.

References:

- Original Algorithm is detailed in the paper Least Angle Regression by Hastie et al.

1.1.9. Orthogonal Matching Pursuit (OMP)

`OrthogonalMatchingPursuit` and `orthogonal_mp` implements the OMP algorithm for approximating the fit of a linear model with constraints imposed on the number of non-zero coefficients (ie. the L $_0$ pseudo-norm).

Being a forward feature selection method like Least Angle Regression, orthogonal matching pursuit can approximate the optimum solution vector with a fixed number of non-zero elements:

$$\arg\min ||y - X\gamma||_2^2 \text{ subject to } \quad ||\gamma||_0 \leq n_{nonzero\_coefs}$$

Alternatively, orthogonal matching pursuit can target a specific error instead of a specific number of non-zero coefficients. This can be expressed as:

$$\arg\min ||\gamma||_0 \text{ subject to } ||y - X\gamma||_2^2 \quad \leq \text{tol}$$

OMP is based on a greedy algorithm that includes at each step the atom most highly correlated with the current residual. It is similar to the simpler matching pursuit (MP) method, but better in that at each iteration, the residual is recomputed using an orthogonal projection on the space of the previously chosen dictionary elements.

Examples:

- Orthogonal Matching Pursuit

References:

- http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf
- Matching pursuits with time-frequency dictionaries, S. G. Mallat, Z. Zhang,

1.1.10. Bayesian Regression

Bayesian regression techniques can be used to include regularization parameters in the estimation procedure: the regularization parameter is not set in a hard sense but tuned to the data at hand.

This can be done by introducing uninformative priors over the hyper parameters of the model. The ℓ_2 regularization used in Ridge Regression is equivalent to finding a maximum a-postiori solution under a Gaussian prior over the parameters w with precision λ^{-1}. Instead of setting *lambda* manually, it is possible to treat it as a random variable to be estimated from the data.

To obtain a fully probabilistic model, the output y is assumed to be Gaussian distributed around Xw:

$$p(y|X, w, \alpha) = \mathcal{N}(y|Xw, \alpha)$$

Alpha is again treated as a random variable that is to be estimated from the data.

The advantages of Bayesian Regression are:

- It adapts to the data at hand.
- It can be used to include regularization parameters in the estimation procedure.

The disadvantages of Bayesian regression include:

- Inference of the model can be time consuming.

References

- A good introduction to Bayesian methods is given in C. Bishop: Pattern Recognition and Machine learning
- Original Algorithm is detailed in the book *Bayesian learning for neural networks* by Radford M. Neal

1.1.10.1. Bayesian Ridge Regression

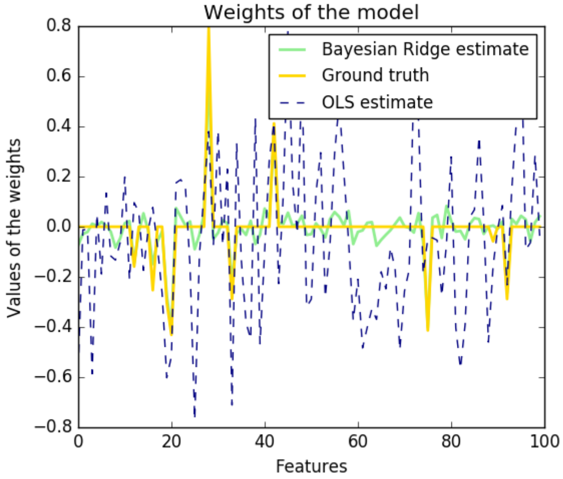
`BayesianRidge` estimates a probabilistic model of the regression problem as described above. The prior for the parameter w is given by a spherical Gaussian:

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I_p})$$

The priors over α and λ are chosen to be gamma distributions, the conjugate prior for the precision of the Gaussian.

The resulting model is called *Bayesian Ridge Regression*, and is similar to the classical **`Ridge`**. The parameters w, α and λ are estimated jointly during the fit of the model. The remaining hyperparameters are the parameters of the gamma priors over α and λ. These are usually chosen to be *non-informative*. The parameters are estimated by maximizing the *marginal log likelihood*.

By default $\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1.e^{-6}$.



Bayesian Ridge Regression is used for regression:

```
>>> from sklearn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> reg = linear_model.BayesianRidge()
>>> reg.fit(X, Y)
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False, copy_X=True,
       fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06, n_iter=300,
       normalize=False, tol=0.001, verbose=False)
```

After being fitted, the model can then be used to predict new values:

```
>>> reg.predict ([[1, 0.]])
array([ 0.50000013])
```

The weights w of the model can be access:

```
>>> reg.coef_
array([ 0.49999993,  0.49999993])
```

Due to the Bayesian framework, the weights found are slightly different to the ones found by Ordinary Least Squares. However, Bayesian Ridge Regression is more robust to ill-posed problem.

Examples:

- Bayesian Ridge Regression

References

- More details can be found in the article Bayesian Interpolation by MacKay, David J. C.

1.1.10.2. Automatic Relevance Determination - ARD

`ARDRegression` is very similar to Bayesian Ridge Regression, but can lead to sparser weights w [1] [2]. **`ARDRegression`** poses a different prior over w, by dropping the assumption of the Gaussian being spherical.

Instead, the distribution over w is assumed to be an axis-parallel, elliptical Gaussian distribution.

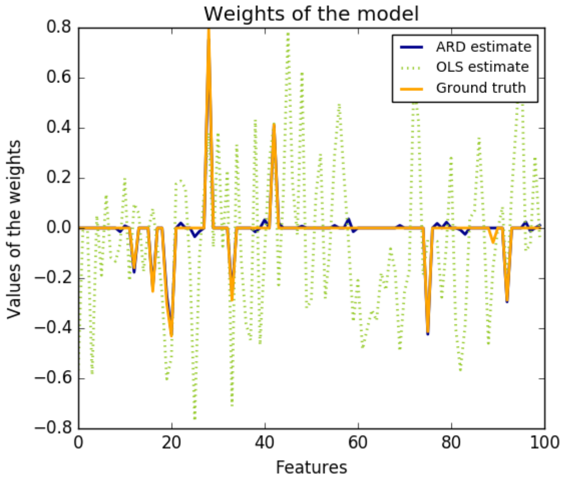
This means each weight w_i is drawn from a Gaussian distribution, centered on zero and with a precision λ_i:

$$p(w|\lambda) = \mathcal{N}(w|0, A^{-1})$$

with $diag\ (A) = \lambda = \{\lambda_1, ..., \lambda_p\}$.

In contrast to Bayesian Ridge Regression, each coordinate of w_i has its own standard deviation λ_i. The prior over all λ_i is chosen to be the same gamma distribution given by hyperparameters λ_1 and λ_2.

»



ARD is also known in the literature as *Sparse Bayesian Learning* and *Relevance Vector Machine* [3] [4].

Examples:

- Automatic Relevance Determination Regression (ARD)

References:

[1] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 7.2.1
[2] David Wipf and Srikantan Nagarajan: A new view of automatic relevance determination
[3] Michael E. Tipping: Sparse Bayesian Learning and the Relevance Vector Machine
[4] Tristan Fletcher: Relevance Vector Machines explained

1.1.11. Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

The implementation of logistic regression in scikit-learn can be accessed from class `LogisticRegression`. This implementation can fit binary, One-vs- Rest, or multinomial logistic regression with optional L2 or L1 regularization.

As an optimization problem, binary class L2 penalized logistic regression minimizes the following cost function:

$$\underset{w, c}{min\,} \frac{1}{2}w^T w + C \sum_{i=1}^n \log(\exp(- y_i (X_i^T w + c)) + 1) .$$

Similarly, L1 regularized logistic regression solves the following optimization problem

$$\underset{w, c}{min\,} \|w\|_1 + C \sum_{i=1}^n \log(\exp(- y_i (X_i^T w + c)) + 1) .$$

The solvers implemented in the class `LogisticRegression` are “liblinear”, “newton-cg”, “lbfgs” and “sag”:

The solver “liblinear” uses a coordinate descent (CD) algorithm, and relies on the excellent C++ LIBLINEAR library, which is shipped with scikit-learn. However, the CD algorithm implemented in liblinear cannot learn a true multinomial (multiclass) model; instead, the optimization problem is decomposed in a “one-vs-rest” fashion so separate binary classifiers are trained for all classes. This happens under the hood, so `LogisticRegression` instances using this solver behave as multiclass classifiers. For L1 penalization `sklearn.svm.l1_min_c` allows to calculate the lower bound for C in order to get a non “null” (all feature weights to zero) model.

»

The “lbfgs”, “sag” and “newton-cg” solvers only support L2 penalization and are found to converge faster for some high dimensional data. Setting *multi_class* to “multinomial” with these solvers learns a true multinomial logistic regression model [5], which means that its probability estimates should be better calibrated than the default “one-vs-rest” setting. The “lbfgs”, “sag” and “newton-cg”” solvers cannot optimize L1-penalized models, therefore the “multinomial” setting does not learn sparse models.

The solver “sag” uses a Stochastic Average Gradient descent [6]. It is faster than other solvers for large datasets, when both the number of samples and the number of features are large.

In a nutshell, one may choose the solver with the following rules:

Case	Solver
Small dataset or L1 penalty	“liblinear”
Multinomial loss or large dataset	“lbfgs”, “sag” or “newton-cg”
Very Large dataset	“sag”

For large dataset, you may also consider using `SGDClassifier` with ‘log’ loss.

Examples:

- L1 Penalty and Sparsity in Logistic Regression
- Path with L1- Logistic Regression
- Plot multinomial and One-vs-Rest Logistic Regression

Differences from liblinear:

There might be a difference in the scores obtained between `LogisticRegression` with `solver=liblinear` or `LinearSVC` and the external liblinear library directly, when `fit_intercept=False` and the fit `coef_` (or) the data to be predicted are zeroes. This is because for the sample(s) with `decision_function` zero, `LogisticRegression` and `LinearSVC` predict the negative class, while liblinear predicts the positive class. Note that a model with `fit_intercept=False` and having many samples with `decision_function` zero, is likely to be a underfit, bad model and you are advised to set `fit_intercept=True` and increase the intercept_scaling.

Note: Feature selection with sparse logistic regression

A logistic regression with L1 penalty yields sparse models, and can thus be used to perform feature selection, as detailed in L1-based feature selection.

`LogisticRegressionCV` implements Logistic Regression with builtin cross-validation to find out the optimal C parameter. “newton-cg”, “sag” and “lbfgs” solvers are found to be faster for high-dimensional dense data, due to warm-starting. For the multiclass case, if *multi_class* option is set to “ovr”, an optimal C is obtained for each class and if the *multi_class* option is set to “multinomial”, an optimal C is obtained by minimizing the cross- entropy loss.

References:

[5] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4

[6] Mark Schmidt, Nicolas Le Roux, and Francis Bach: Minimizing Finite Sums with the Stochastic Average Gradient.

1.1.12. Stochastic Gradient Descent - SGD

Stochastic gradient descent is a simple yet very efficient approach to fit linear models. It is particularly useful when the number of samples (and the number of features) is very large. The `partial_fit` method allows only/out-of-core learning.

The classes `SGDClassifier` and `SGDRegressor` provide functionality to fit linear models for classification and regression using different (convex) loss functions and different penalties. E.g., with `loss="log"`, `SGDClassifier` fits a logistic regression model, while with `loss="hinge"` it fits a linear support vector machine (SVM).

References

- Stochastic Gradient Descent

1.1.13. Perceptron

The `Perceptron` is another simple algorithm suitable for large scale learning. By default:

- It does not require a learning rate.
- It is not regularized (penalized).
- It updates its model only on mistakes.

The last characteristic implies that the Perceptron is slightly faster to train than SGD with the hinge loss and that the resulting models are sparser.

1.1.14. Passive Aggressive Algorithms

The passive-aggressive algorithms are a family of algorithms for large-scale learning. They are similar to the Perceptron in that they do not require a learning rate. However, contrary to the Perceptron, they include a regularization parameter `C`.

For classification, **`PassiveAggressiveClassifier`** can be used with `loss='hinge'` (PA-I) or `loss='squared_hinge'` (PA-II). For regression, **`PassiveAggressiveRegressor`** can be used with `loss='epsilon_insensitive'` (PA-I) or `loss='squared_epsilon_insensitive'` (PA-II).

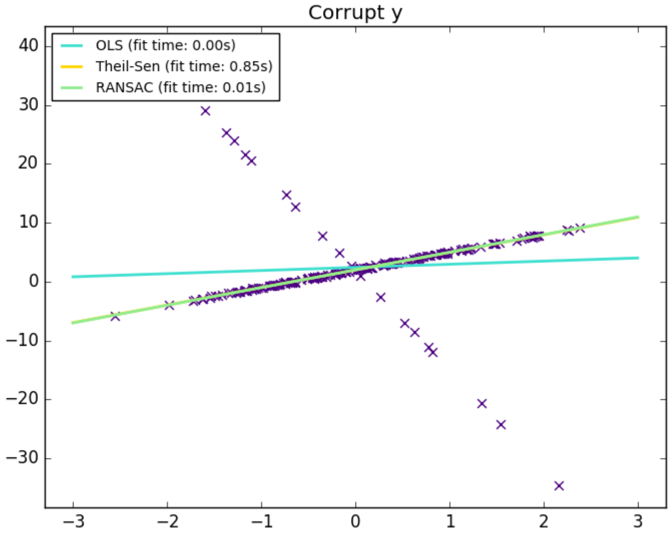
References:

- "Online Passive-Aggressive Algorithms" K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR 7 (2006)

»

1.1.15. Robustness regression: outliers and modeling errors

Robust regression is interested in fitting a regression model in the presence of corrupt data: either outliers, or error in the model.

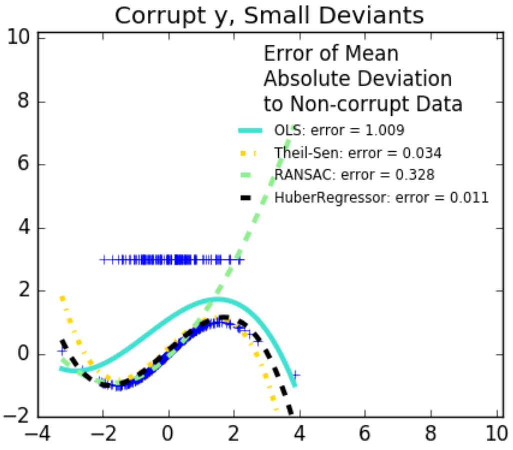


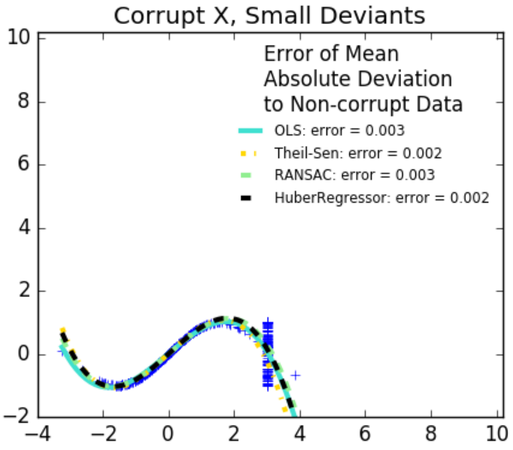
1.1.15.1. Different scenario and useful concepts

There are different things to keep in mind when dealing with data corrupted by outliers:

- **Outliers in X or in y**?

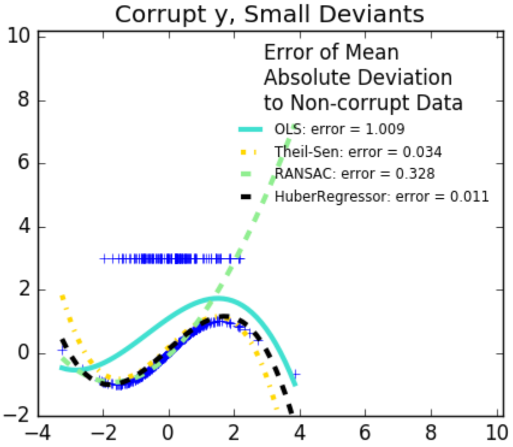
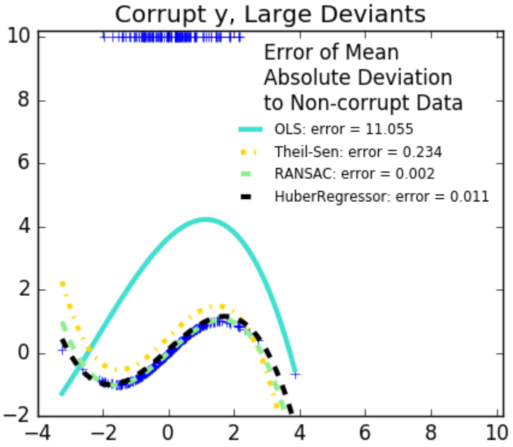
Outliers in the y direction	Outliers in the X direction





- **Fraction of outliers versus amplitude of error**

  The number of outlying points matters, but also how much they are outliers.

Small outliers	Large outliers
	

An important notion of robust fitting is that of breakdown point: the fraction of data that can be outlying for the fit to start missing the inlying data.

Note that in general, robust fitting in high-dimensional setting (large *n_features*) is very hard. The robust models here will probably not work in these settings.

Trade-offs: which estimator?

Scikit-learn provides 3 robust regression estimators: RANSAC, Theil Sen and HuberRegressor

- HuberRegressor should be faster than RANSAC and Theil Sen unless the number of samples are very large, i.e `n_samples` >> `n_features`. This is because RANSAC and Theil Sen fit on smaller subsets of the data. However, both Theil Sen and RANSAC are unlikely to be as robust as HuberRegressor for the default parameters.
- RANSAC is faster than Theil Sen and scales much better with the number of samples
- RANSAC will deal better with large outliers in the y direction (most common situation)
- Theil Sen will cope better with medium-size outliers in the X direction, but this property will disappear in large dimensional settings.

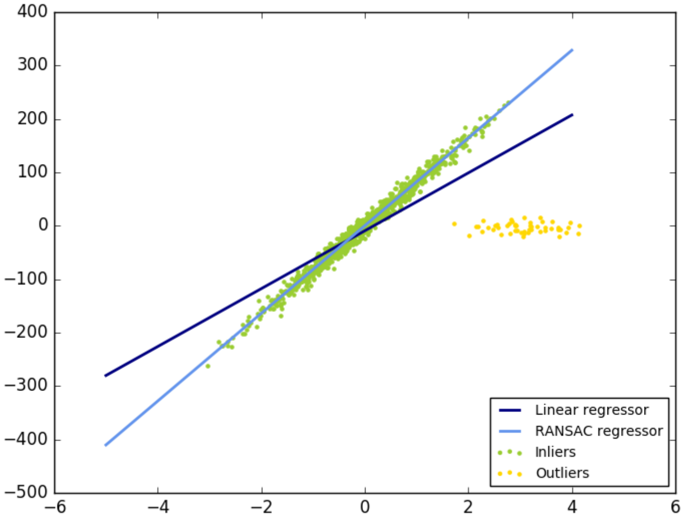
When in doubt, use RANSAC

1.1.15.2. RANSAC: RANdom SAmple Consensus

RANSAC (RANdom SAmple Consensus) fits a model from random subsets of inliers from the complete data set.

RANSAC is a non-deterministic algorithm producing only a reasonable result with a certain probability, which is dependent on the number of iterations (see *max_trials* parameter). It is typically used for linear and non-linear regression problems and is especially popular in the fields of photogrammetric computer vision.

The algorithm splits the complete input sample data into a set of inliers, which may be subject to noise, and outliers, which are e.g. caused by erroneous measurements or invalid hypotheses about the data. The resulting model is then estimated only from the determined inliers.



1.1.15.2.1. Details of the algorithm

Each iteration performs the following steps:

1. Select `min_samples` random samples from the original data and check whether the set of data is valid (see `is_data_valid`).
2. Fit a model to the random subset (`base_estimator.fit`) and check whether the estimated model is valid (see `is_model_valid`).
3. Classify all data as inliers or outliers by calculating the residuals to the estimated model (`base_estimator.predict(X) - y`) - all data samples with absolute residuals smaller than the `residual_threshold` are considered as inliers.
4. Save fitted model as best model if number of inlier samples is maximal. In case the current estimated model has the same number of inliers, it is only considered as the best model if it has better score.

These steps are performed either a maximum number of times (`max_trials`) or until one of the special stop criteria are met (see `stop_n_inliers` and `stop_score`). The final model is estimated using all inlier samples (consensus set) of the previously determined best model.

The `is_data_valid` and `is_model_valid` functions allow to identify and reject degenerate combinations of random sub-samples. If the estimated model is not needed for identifying degenerate cases, `is_data_valid` should be used as it is called prior to fitting the model and thus leading to better computational performance.

Examples:

- Robust linear model estimation using RANSAC
- Robust linear estimator fitting

References:

- https://en.wikipedia.org/wiki/RANSAC
- "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography" Martin A. Fischler and Robert C. Bolles - SRI International (1981)
- "Performance Evaluation of RANSAC Family" Sunglok Choi, Taemin Kim and Wonpil Yu - BMVC (2009)

1.1.15.3. Theil-Sen estimator: generalized-median-based estimator

The **`TheilSenRegressor`** estimator uses a generalization of the median in multiple dimensions. It is thus robust to multivariate outliers. Note however that the robustness of the estimator decreases quickly with the dimensionality of the problem. It looses its robustness properties and becomes no better than an ordinary least squares in high dimension.

Examples:

- Theil-Sen Regression
- Robust linear estimator fitting

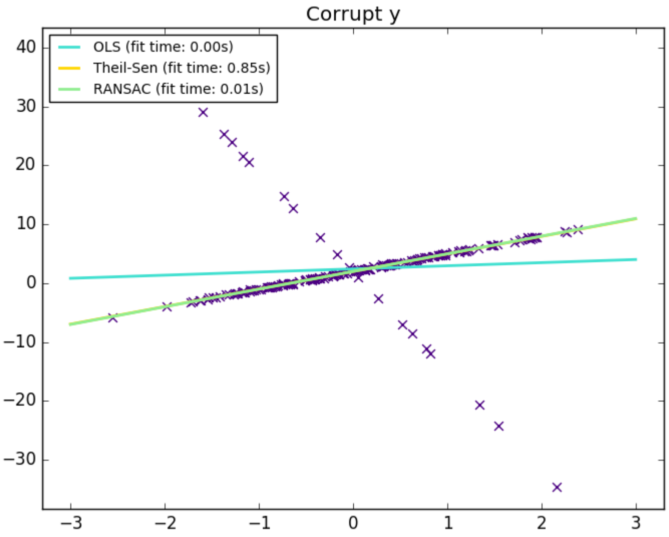
References:

- https://en.wikipedia.org/wiki/Theil%E2%80%93Sen_estimator

1.1.15.3.1. Theoretical considerations

`TheilSenRegressor` is comparable to the Ordinary Least Squares (OLS) in terms of asymptotic efficiency and as an unbiased estimator. In contrast to OLS, Theil-Sen is a non-parametric method which means it makes no assumption about the underlying distribution of the data. Since Theil-Sen is a median-based estimator, it is more robust against corrupted data aka outliers. In univariate setting, Theil-Sen has a breakdown point of about 29.3% in case of a simple linear regression which means that it can tolerate arbitrary corrupted data of up to 29.3%.

»



The implementation of `TheilSenRegressor` in scikit-learn follows a generalization to a multivariate linear regression model [#f1]_ using the spatial median which is a generalization of the median to multiple dimensions [8].

In terms of time and space complexity, Theil-Sen scales according to

$$\binom{n_{samples}}{n_{subsamples}}$$

which makes it infeasible to be applied exhaustively to problems with a large number of samples and features. Therefore, the magnitude of a subpopulation can be chosen to limit the time and space complexity by considering only a random subset of all possible combinations.

Examples:

- Theil-Sen Regression

References:

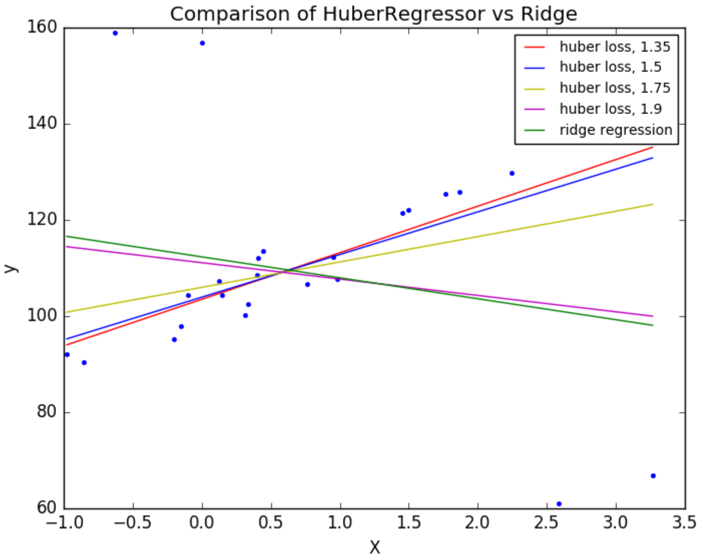
[7] Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang: Theil-Sen Estimators in a Multiple Linear Regression Model.

[8] T. Kärkkäinen and S. Äyrämö: On Computation of Spatial Median for Robust Data Mining.

1.1.15.4. Huber Regression

The `HuberRegressor` is different to `Ridge` because it applies a linear loss to samples that are classified as outliers. A sample is classified as an inlier if the absolute error of that sample is lesser than a certain threshold. It differs from `TheilSenRegressor` and `RANSACRegressor` because it does not ignore the effect of the outliers but gives a lesser weight to them.

»



The loss function that `HuberRegressor` minimizes is given by

$$\underset{w,\sigma}{min} \sum_{i=1}^{n} \left(\sigma + H_m\left(\frac{X_i w - y_i}{\sigma} \right) \sigma \right) + \alpha {||w||_2}^2$$

where

$$H_m(z) = \begin{cases} z^2, & \text{if } |z| < \epsilon, \\ 2\epsilon|z| - \epsilon^2, & \text{otherwise} \end{cases}$$

It is advised to set the parameter `epsilon` to 1.35 to achieve 95% statistical efficiency.

1.1.15.5. Notes

The `HuberRegressor` differs from using `SGDRegressor` with loss set to *huber* in the following ways.

- `HuberRegressor` is scaling invariant. Once `epsilon` is set, scaling `X` and `y` down or up by different values would produce the same robustness to outliers as before. as compared to `SGDRegressor` where `epsilon` has to be set again when `X` and `y` are scaled.
- `HuberRegressor` should be more efficient to use on data with small number of samples while `SGDRegressor` needs a number of passes on the training data to produce the same robustness.

Examples:

- HuberRegressor vs Ridge on dataset with strong outliers

References:

[9] Peter J. Huber, Elvezio M. Ronchetti: Robust Statistics, Concomitant scale estimates, pg 172

Also, this estimator is different from the R implementation of Robust Regression (http://www.ats.ucla.edu/stat/r/dae/rreg.htm) because the R implementation does a weighted least squares implementation with weights given to each sample on the basis of how much the residual is greater than a certain threshold.

1.1.16. Polynomial regression: extending linear models with basis functions

»

One common pattern within machine learning is to use linear models trained on nonlinear functions of the data. This approach maintains the generally fast performance of linear methods, while allowing them to fit a much wider range of data.

For example, a simple linear regression can be extended by constructing **polynomial features** from the coefficients. In the standard linear regression case, you might have a model that looks like this for two-dimensional data:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2$$

If we want to fit a paraboloid to the data instead of a plane, we can combine the features in second-order polynomials, so that the model looks like this:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

The (sometimes surprising) observation is that this is *still a linear model*: to see this, imagine creating a new variable

$$z = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]$$

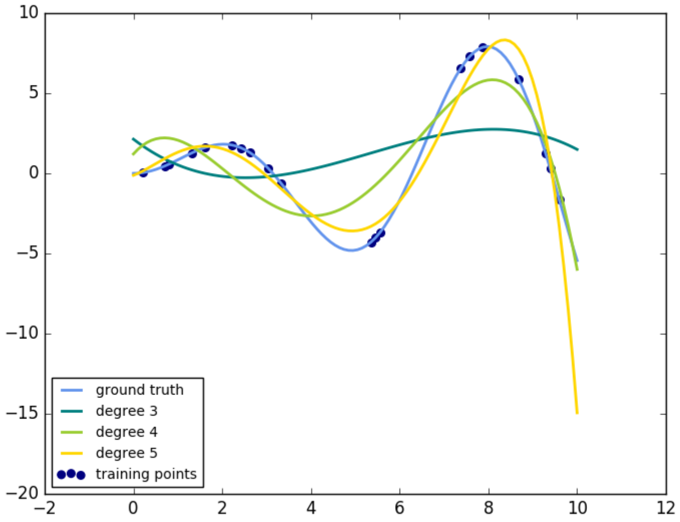
With this re-labeling of the data, our problem can be written

$$\hat{y}(w, x) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5$$

We see that the resulting *polynomial regression* is in the same class of linear models we'd considered above (i.e. the model is linear in w) and can be solved by the same techniques. By considering linear fits within a higher-dimensional space built with these basis functions, the model has the flexibility to fit a much broader range of data.

Here is an example of applying this idea to one-dimensional data, using polynomial features of varying degrees:

»



This figure is created using the **PolynomialFeatures** preprocessor. This preprocessor transforms an input data matrix into a new data matrix of a given degree. It can be used as follows:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   0.,   0.,   1.],
       [  1.,   2.,   3.,   4.,   6.,   9.],
       [  1.,   4.,   5.,  16.,  20.,  25.]])
```

The features of X have been transformed from $[x_1, x_2]$ to $[1, x_1, x_2, x_1^2, x_1 x_2, x_2^2]$, and can now be used within any linear model.

This sort of preprocessing can be streamlined with the Pipeline tools. A single object representing a simple polynomial regression can be created and used as follows:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> model = Pipeline([('poly', PolynomialFeatures(degree=3)),
...                   ('linear', LinearRegression(fit_intercept=False))])
>>> # fit to an order-3 polynomial data
>>> x = np.arange(5)
>>> y = 3 - 2 * x + x ** 2 - x ** 3
>>> model = model.fit(x[:, np.newaxis], y)
>>> model.named_steps['linear'].coef_
array([ 3., -2.,  1., -1.])
```

The linear model trained on polynomial features is able to exactly recover the input polynomial coefficients.

In some cases it's not necessary to include higher powers of any single feature, but only the so-called *interaction features* that multiply together at most d distinct features. These can be gotten from **PolynomialFeatures** with the setting `interaction_only=True`.

For example, when dealing with boolean features, $x_i^n = x_i$ for all n and is therefore useless; but $x_i x_j$ represents the conjunction of two booleans. This way, we can solve the XOR problem with a linear classifier:

```
>>> from sklearn.linear_model import Perceptron
>>> from sklearn.preprocessing import PolynomialFeatures
>>> import numpy as np
>>> X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
>>> y = X[:, 0] ^ X[:, 1]
>>> y
array([0, 1, 1, 0])
>>> X = PolynomialFeatures(interaction_only=True).fit_transform(X).astype(int)
>>> X
array([[1, 0, 0, 0],
       [1, 0, 1, 0],
       [1, 1, 0, 0],
       [1, 1, 1, 1]])
>>> clf = Perceptron(fit_intercept=False, n_iter=10, shuffle=False).fit(X, y)
```

And the classifier “predictions” are perfect:

```
>>> clf.predict(X)
array([0, 1, 1, 0])
>>> clf.score(X, y)
1.0
```