Sample
Ridge Regression

The purpose of regression is to minimize a loss function, the **residual sum of squares** (RSS) being one commonly used. The problem of overfitting described in *Chapter 2 §Accessing a model – Overfitting* can be addressed by adding a penalty term to the loss function. The **penalty term** is an element of the larger concept of **regularization**.

L_n roughness penalty

Regularization consists of adding a penalty function **J(w)** to the loss function (or RSS in the case of a regressive classifier) in order to prevent the model parameters (or weights) from reaching high values. A model that fits a training set very well tends to have many features variable with relatively large weights. This process is known as **shrinkage**. Practically, shrinkage consists of adding a function with model parameters as an argument to the loss function.

$$\hat{\boldsymbol{w}} = \underset{w_d}{arg\,min} \left\{ \sum_{i=0}^{n-1} \left(y_i - f(\boldsymbol{x_i}|\boldsymbol{w}) \right)^2 + \lambda J(\boldsymbol{w}) \right\}$$

The penalty function is completely independent from the training set `{x,y}`. The penalty term is usually expressed as a power to function of the norm of the model parameters (or weights) w_d. For a model of D dimension the generic **Lp-norm** is defined as

$$J_{pq}(w) = \|w\|_p^q = \left[\sum_{d=1}^{D-1} |w_d|^p \right]^{q/p}$$

Notation:
Regularization applies to parameters or weights associated to an observations. In order to be consistent with our notation w_0 being the intercept value, the regularization applies to the parameters $w_1 \ldots w_d$.

The two most commonly used penalty functions for regularization are L_1 and L_2.

Regularization in machine learning:
The regularization technique is not specific to the linear or logistic regression. Any algorithm that minimizes the residual sum of squares, such as support vector machine or feed-forward neural network, can be regularized by adding a roughness penalty function to the RSS.

The L_1 regularization applied to the linear regression is known as the **Lasso regularization**. The **Ridge regression** is a linear regression that uses the L_2 regularization penalty.

You may wonder which regularization makes sense for a given training set. In a nutshell L_2 and L_1 regularization differs in terms of computation efficiency, estimation and features selection [*Machine Learning: A Probabilistic Perspective §13.1 L_1 Regularization*

basics 6:10] [*Feature selection, L_1 vs. L_2 regularization, and rotational invariance* 6:11]

- Model estimation: L_1 generates a sparser estimation of the regression parameters than L_2. For large non-sparse dataset, L_2 has a smaller estimation error than L_1
- Feature selection: L_1 is more effective in reducing the regression weights for features with high value than L_2.Therefore L_1 a reliable features selection tool.
- Overfitting: Both L_1 and L_2 reduce the impact of overfitting. However, L_1 has a significant advantage in overcoming overfitting (or excessive complexity of a model) for the same reason it is more appropriate for selecting features.
- Computation: L_2 is conducive to a more efficient computation model. The summation of the loss function and L_2 penalty $\mathbf{w^2}$ is a continuous and differentiable function for which the first and second derivative can be computed (**convex minimization**). The L_1 term is the summation of $|\mathbf{w_i}|$ and therefore not differentiable.

> Terminology:
> The ridge regression is sometimes called the **penalized least squares** regression. The L_2 regularization is also known as the **weight decay**.

Let us implement the ridge regression, and then evaluate the impact of the L_2-norm penalty factor.

Ridge Regression

The ridge regression is a multivariate linear regression with a L_2 norm penalty term.

$$\hat{w}_{Ridge} = \arg\min_{w_d} \sum_{j=0}^{n-1} (y - w_0 - w^T x)^2 + \lambda \|w\|_2^2 \quad \|w\|_2^2 = \sum_{d=1}^{D} {w_d}^2$$

The computation of the ridge regression parameters requires the resolution of the system of linear equations similar to the linear regression.

> Matrix representation of ridge regression closed form
>
> $$(X^T X - \lambda . I)\hat{w}_{Ridge} = X^T y$$
>
> I is the identity matrix and using the QR decomposition
>
> $$(X^T X - \lambda . I) = Q \begin{vmatrix} R \\ 0 \end{vmatrix} \quad w_{Ridge} = Q^T y \begin{bmatrix} R \\ 0 \end{bmatrix}^{-1}$$

Implementation

The implementation of the ridge regression add L_2 regularization term to the multiple linear regression computation of the Apache Commons Math Library.

The methods of `RidgeRegression`, have the same signature as its Ordinary Least Squares counterpart. However, the class has to inherit the abstract base class `AbstractMultipleLinearRegression` in the Apache Commons Math and override the generation of the QR decomposition to include the penalty term

```
class RidgeRegression[T <% Double](val xt: XTSeries[Array[T]],
                                   val y: DblVector,
                                   val lambda: Double) {
                    extends AbstractMultipleLinearRegression
                    with PipeOperator[Array[T], Double] {
   private var qr: QRDecomposition = null
   private[this] val model: Option[RegressionModel] = …
   …
}
```

Besides the input time series `xt` and the labels `y`, the ridge regression requires the `lambda` factor of the L_2 penalty term. The instantiation of the class train the `model`. The steps to create the ridge regression models are to

- Extract the Q and R matrices for the input values, newXSampleData (1)
- Compute the weights using the `calculateBeta` defined in the base class (2)
- Return the tuple regression weights `calculateBeta` and the residuals `calculateResiduals`

```
private val model: Option[(DblVector, Double)] = {
  this.newXSampleData(xt.toDblMatrix)  //1
  newYSampleData(y)
  val _rss = calculateResiduals.toArray.map(x => x*x).sum
  val wRss = (calculateBeta.toArray, _rss) //2
  Some(RegressionModel(wRss._1, wRss._2))
}
```

The QR decomposition in the base class `AbstractMultipleLinearRegression` does not include the penalty term (3); the identity matrix with `lambda` factor in the diagonal has to be added to the matrix to be decomposed (4).

```
override protected def newXSampleData(x: DblMatrix): Unit = {
  super.newXSampleData(x)   //3
  val xtx: RealMatrix = getX
  val nFeatures = xt(0).size
  Range(0, nFeatures).foreach(i =>
xtx.setEntry(i,i,xtx.getEntry(i,i) + lambda)) //4
  qr = new QRDecomposition(xtx)
}
```

The regression weights are computed by resolving the system of linear equations using substitution on the Q.R matrices. It overrides the `calculateBeta` from the base class.

```
override protected def calculateBeta: RealVector =
  qr.getSolver().solve(getY())
```

Test case

The objective of the test case is to identify the impact of the L_2 penalization on the RSS value then compare the predicted values with original values.

Let us consider the first test case related to the regression on the daily price variation of the Copper ETF (symbol: CU) using the stock daily volatility and volume as feature. The

implementation of the extraction of observations is identical as with the least squares regression.

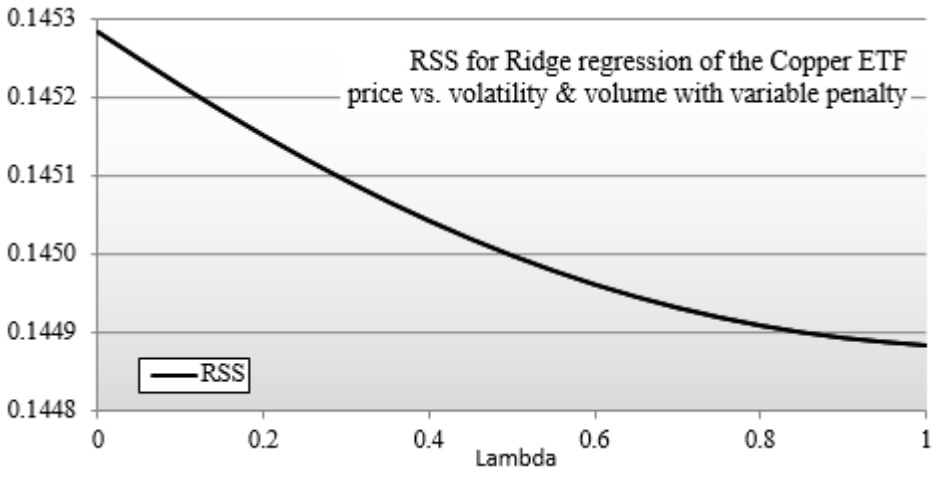
```
val src = DataSource(path, true, true, 1)
val price = src |> YahooFinancials.adjClose
val volatility = src |> YahooFinancials.volatility
val volume = src |> YahooFinancials.volume //1

val _price = price.get.toArray
val deltaPrice = XTSeries[Double](_price
                                  .drop(1)
                                  .zip(_price.take(_price.size -1))
                                  .map( z => z._1 - z._2)) //2
val data =  volatility.get
                      .zip(volume.get)
                      .map(z => Array[Double](z._1, z._2)) //3
val features = XTSeries[DblVector](data.take(data.size-1))
val regression = new RidgeRegression[Double](features, deltaPrice,
lambda) //4

regression.rss match {
  case Some(rss) => Display.show(rss, logger)
….
```

The observed data, ETF daily `price`, and the features (`volatility` and `volume`) are extracted from the source `src` (1). The daily price change `deltaPrice` is computed using a combination of Scala `take` and `drop` methods (2). The features vector is created by zipping volatility and volume (3). The model is created by instantiating the `RidgeRegression` class (4). The RSS value, `rss`, is finally displayed (5).

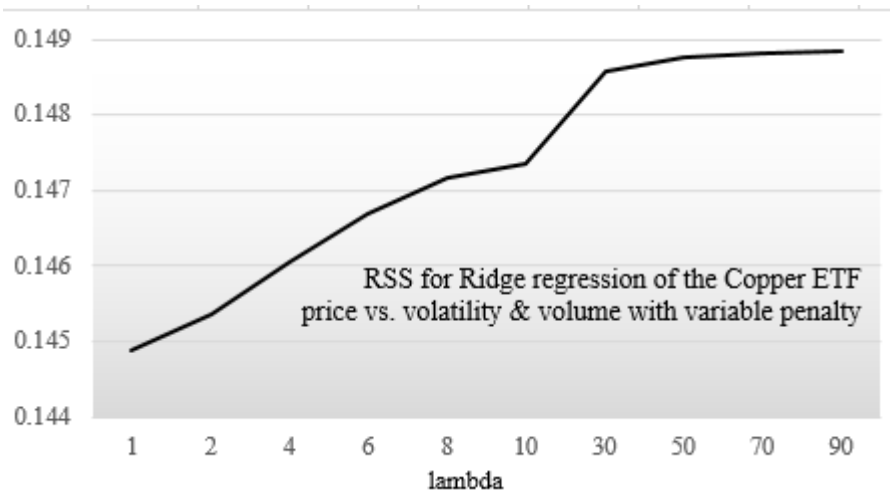
The RSS value, `rss` is plotted for different values of lambda <= 1.0



Graph of RSS versus Lambda for Copper ETF

The residual sum of squares decreased as λ increases. The curve seems reaching for a minimum around λ=1. The case of λ = 0 correspond to the least squares regression.

Next let us plot the RSS value for λ varying between 1 and 100



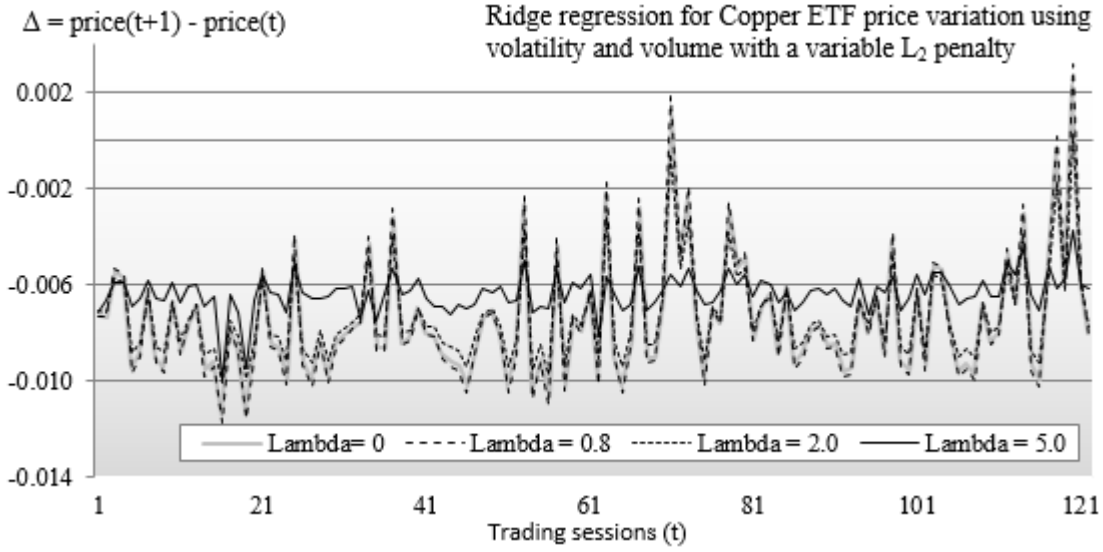
Graph RSS versus large value Lambda for Copper ETF

This time around RSS increases with λ before reaching a maximum for λ > 60. This behavior is consistent with other findings [*Model selection and assessment* 6:12]. *As λ increases, the over-fitting gets more expensive and therefore the RSS value increases.*

The regression `weights` can by simply output as follows

```
regression.weights.get
```

Let's plot the predicted price variation of the Copper ETF using the ridge regression with different value of lambda (λ).



Graph of Ridge regression on Copper ETF price variation with variable Lambda

The original price variation of the Copper ETF `Δ = price(t+1)-price(t)` is plotted as λ =0. The predicted values for λ = 0.8 is very similar to the original data. The predicted values for λ = 0.8 follows the pattern of the original data with reduction of large variations (peaks and troves). The predicted values for λ = 5 corresponds to a smoothed dataset. The pattern of the original data is preserved but the magnitude of the price variation is significantly reduced.