7
Linear Models

The family of linear models represents one of the most useful hypothesis classes. Many learning algorithms that are widely applied in algorithmic trading rely on linear predictors because they can be efficiently trained in many cases, they are relatively robust to noisy financial data, and they have strong links to the theory of finance. Linear predictors are also intuitive, easy to interpret, and often fit the data reasonably well or at least provide a good baseline.

Linear regression has been known for over 200 years when Legendre and Gauss applied it to astronomy and began to analyze its statistical properties. Numerous extensions have since adapted the linear regression model and the baseline **ordinary least squares** (**OLS**) method to learn its parameters:

- **Generalized linear models** (**GLM**) expand the scope of applications by allowing for response variables that imply an error distribution other than the normal distribution. GLM include the probit or logistic models for **categorical response variables** that appear in classification problems.
- More **robust estimation methods** enable statistical inference where the data violates baseline assumptions due to, for example, correlation over time or across observations. This is often the case with panel data that contains repeated observations on the same units such as historical returns on a universe of assets.
- **Shrinkage methods** aim to improve the predictive performance of linear models. They use a complexity penalty that biases the coefficients learned by the model with the goal of reducing the model's variance and improving out-of-sample predictive performance.

In practice, linear models are applied to regression and classification problems with the goals of inference and prediction. Numerous asset pricing models that have been developed by academic and industry researchers leverage linear regression. Applications include the identification of significant factors that drive asset returns, for example, as a basis for risk management, as well as the prediction of returns over various time horizons. Classification problems, on the other hand, include directional price forecasts.

In this chapter, we will cover the following topics:

- How linear regression works and which assumptions it makes
- How to train and diagnose linear regression models
- How to use linear regression to predict future returns
- How use regularization to improve the predictive performance
- How logistic regression works
- How to convert a regression into a classification problem

For code examples, additional resources, and references, see the directory for this chapter in the online GitHub repository.

Linear regression for inference and prediction

As the name suggests, linear regression models assume that the output is the result of a linear combination of the inputs. The model also assumes a random error that allows for each observation to deviate from the expected linear relationship. The reasons that the model does not perfectly describe the relationship between inputs and output in a deterministic way include, for example, missing variables, measurement, or data collection issues.

If we want to draw statistical conclusions about the true (but not observed) linear relationship in the population based on the regression parameters estimated from the sample, we need to add assumptions about the statistical nature of these errors. The baseline regression model makes the strong assumption that the distribution of the errors is identical across errors and that errors are independent of each other, that is, knowing one error does not help to forecast the next error. The assumption of **independent and identically distributed** (**iid**) errors implies that their covariance matrix is the identity matrix multiplied by a constant representing the error variance.

These assumptions guarantee that the OLS method delivers estimates that are not only unbiased but also efficient, that is, they have the lowest sampling error learning algorithms. However, these assumptions are rarely met in practice. In finance, we often encounter panel data with repeated observations on a given cross-section. The attempt to estimate the systematic exposure of a universe of assets to a set of risk factors over time typically surfaces correlation in the time or cross-sectional dimension, or both. Hence, alternative learning algorithms have emerged that assume more error covariance matrices that differ from multiples of the identity matrix.

On the other hand, methods that learn biased parameters for a linear model may yield estimates with a lower variance and, hence, improve the predictive performance. **Shrinkage methods** reduce the model complexity by applying regularization that adds a penalty term to the linear objective function. The penalty is positively related to the absolute size of the coefficients so that these are shrunk relative to the baseline case. Larger coefficients imply a more complex model that reacts more strongly to variations in the inputs. Properly calibrated, the penalty can limit the growth of the model's coefficients beyond what an optimal bias-variance trade-off would suggest.

We will introduce the baseline cross-section and panel techniques for linear models and important enhancements that produce accurate estimates when key assumptions are violated. We will then illustrate these methods by estimating factor models that are ubiquitous in the development of algorithmic trading strategies. Lastly, we will focus on regularization methods.

The multiple linear regression model

We will introduce the model's specification and objective function, methods to learn its parameters, statistical assumptions that allow for inference and diagnostics of these assumptions, as well as extensions to adapt the model to situations where these assumptions fail.

How to formulate the model

The multiple regression model defines a linear functional relationship between one continuous outcome variable and p input variables that can be of any type but may require preprocessing. Multivariate regression, in contrast, refers to the regression of multiple outputs on multiple input variables.

In the population, the linear regression model has the following form for a single instance of the output y, an input vector $\mathbf{x}^T = (x_1, \ldots, x_p)$, and the error ε:

$$y = f(\mathbf{x}) + \epsilon = \beta_0 + \beta_1 x_1 \ldots + \beta_p x_p + \epsilon = \beta_0 + \sum_{j=1}^{p} \beta_j x_j + \epsilon$$

The interpretation of the coefficients is straightforward: the value of a coefficient β_i is the partial, average effect of the variable x_i on the output, holding all other variables constant.

The model can also be written more compactly in matrix form. In this case, y is a vector of N output observations, X is the design matrix with N rows of observations on the p variables plus a column of 1s for the intercept, and $\boldsymbol{\beta}$ is the vector containing the $P = p+1$ coefficients $\beta_0, \ldots, \beta_p$:

$$\underset{(N\times 1)}{\boldsymbol{y}} = \underset{(N\times P)}{\boldsymbol{X}} \underset{(P\times 1)}{\boldsymbol{\beta}} + \underset{(N\times 1)}{\boldsymbol{\epsilon}}$$

The model is linear in its $p+1$ parameters but can model non-linear relationships by choosing or transforming variables accordingly, for example by including a polynomial basis expansion or logarithmic terms. It can also use categorical variables with dummy encoding, and interactions between variables by creating new inputs of the form $x_i \cdot x_j$.

To complete the formulation of the model from a statistical point of view so that we can test a hypothesis about the parameters, we need to make specific assumptions about the error term. We'll do this after first introducing the alternative methods to learn the parameters.

How to train the model

There are several methods to learn the model parameters $\boldsymbol{\beta}$ from the data: **ordinary least squares** (**OLS**), **maximum likelihood estimation** (**MLE**), and **stochastic gradient descent** (**SGD**).

Least squares

The least squares method is the original method to learn the parameters of the hyperplane that best approximates the output from the input data. As the name suggests, the best approximation minimizes the sum of the squared distances between the output value and the hyperplane represented by the model.

The difference between the model's prediction and the actual outcome for a given data point is the residual (whereas the deviation of the true model from the true output in the population is called **error**). Hence, in formal terms, the least squares estimation method chooses the coefficient vector $\boldsymbol{\beta}$ to minimize the **residual sum of squares** (**RSS**):

$$RSS(\boldsymbol{\beta}) = \sum_{i=1}^{N} \epsilon_i^2 = \sum_{i=1}^{N} (y_i - f(x_i))^2 = \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 = (\boldsymbol{y} - \boldsymbol{X}\beta)^T (\boldsymbol{y} - \boldsymbol{X}\beta)$$

Hence, the least-squares coefficients $\boldsymbol{\beta}^{LS}$ are computed as:

$$\underset{\boldsymbol{\beta}^{LS}}{\operatorname{argmin}} = \text{RSS} = (\boldsymbol{y} - \boldsymbol{X}\beta)^T(\boldsymbol{y} - \boldsymbol{X}\beta)$$

The optimal parameter vector that minimizes RSS results from setting the derivatives of the preceding expression with respect to $\boldsymbol{\beta}$ to zero. This produces a unique solution, assuming X has full column rank, that is, the input variables are not linearly dependent, as follows:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T X)^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

When *y* and *X* have been de-meaned by subtracting their respective means, $\boldsymbol{\beta}$ represents the ratio of the covariance between the inputs and the outputs $\boldsymbol{X}^T\boldsymbol{y}$ and the output variance $(\boldsymbol{X}^T X)^{-1}$. There is also a geometric interpretation: the coefficients that minimize RSS ensure that the vector of residuals $\boldsymbol{y} - \hat{\boldsymbol{y}}$ is orthogonal to the subspace of $\mathbb{R}^N$ spanned by the columns of *X*, and the estimates $\hat{\boldsymbol{y}}$ are orthogonal projections into that subspace.

Maximum likelihood estimation

MLE is an important general method to estimate the parameters of a statistical model. It relies on the likelihood function that computes how likely it is to observe the sample of output values for a given set of both input data as a function of the model parameters. The likelihood differs from probabilities in that it is not normalized to range from 0 to 1.

We can set up the likelihood function for the linear regression example by assuming a distribution for the error term, such as the standard normal distribution:

$$\epsilon_i \sim N(0,1) \quad \forall\, i = 1, \ldots, n.$$

This allows us to compute the conditional probability of observing a given output y_i given the corresponding input vector x_i and the parameters, $p(y_i|\boldsymbol{x}_i, \boldsymbol{\beta})$:

$$p(y_i|\boldsymbol{x}_i, \boldsymbol{\beta}) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{\epsilon_i^2}{2\sigma}} = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(y_i - \boldsymbol{x}_i\beta)^2}{2\sigma}}$$

Assuming the output values are conditionally independent given the inputs, the likelihood of the sample is proportional to the product of the conditional probabilities of the individual output data points. Since it is easier to work with sums than with products, we apply the logarithm to obtain the log-likelihood function:

$$\log \mathcal{L}(\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{\beta}) = \sum_{i=i}^{n} \log \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - \boldsymbol{x}_i \boldsymbol{\beta})^2}{2\sigma}}$$

The goal of MLE is to maximize the probability of the output sample that has in fact been observed by choosing model parameters, taking the observed inputs as given. Hence, the MLE parameter estimate results from maximizing the (log) likelihood function:

$$\beta_{\text{MLE}} = \underset{\beta}{\operatorname{argmin}} \mathcal{L}$$

Due to the assumption of normal distribution, maximizing the log-likelihood function produces the same parameter solution as least squares because the only expression that depends on the parameters is squared residual in the exponent. For other distributional assumptions and models, MLE will produce different results, and in many cases, least squares is not applicable, as we will see later for logistic regression.

Gradient descent

Gradient descent is a general-purpose optimization algorithm that will find stationary points of smooth functions. The solution will be a global optimum if the objective function is convex. Variations of gradient descent are widely used in the training of complex neural networks, but also to compute solutions for MLE problems.

The algorithm uses the gradient of the objective function that contains its partial derivatives with respect to the parameters. These derivatives indicate how much the objective changes for infinitesimal steps in the direction of the corresponding parameters. It turns out that the maximal change of the function value results from a step in the direction of the gradient itself.

Hence, when minimizing a function that describes, for example, the cost of a prediction error, the algorithm computes the gradient for the current parameter values using the training data and modifies each parameter according to the negative value of its corresponding gradient component. As a result, the objective function will assume a lower value and move the parameters move closer to the solution. The optimization stops when the gradient becomes small, and the parameter values change very little.

The size of these steps is the learning rate, which is a critical parameter that may require tuning; many implementations include the option for this learning rate to increase with the number of iterations gradually. Depending on the size of the data, the algorithm may iterate many times over the entire dataset. Each such iteration is called an **epoch.** The number of epochs and the tolerance used to stop further iterations are hyperparameters you can tune.

Stochastic gradient descent randomly selects a data point and computes the gradient for this data point as opposed to an average over a larger sample to achieve a speedup. There are also batch versions that use a certain number of data points for each step.

The Gauss—Markov theorem

To assess the statistical of the model and conduct inference, we need to make assumptions about the residuals, that is, the properties of the unexplained part of the input. The **Gauss—Markov theorem** (**GMT**) defines the assumptions required for OLS to produce unbiased estimates of the model parameters $\boldsymbol{\beta}$, and when these estimates have the lowest standard error among all linear models for cross-sectional data.

The baseline multiple regression model makes the following GMT assumptions:

1. In the population, **linearity** holds, $y = \beta_0 + \beta_1 x_1 \ldots + \beta_k x_k + \epsilon$ where β_i are unknown but constant and ϵ is a random error
2. The data for the input variables $x_1, \ldots, x_k$ are a **random sample** from the population
3. No perfect **collinearity**—there are no exact linear relationships among the input variables
4. The **error has a conditional mean of zero** given any of the inputs: $E[\epsilon | x_1, \ldots, x_k] = 0$
5. **Homoskedasticity**, the error term ϵ has constant variance given the inputs: $E[\epsilon | x_1, \ldots, x_k] = \sigma^2$

The fourth assumption implies that no missing variable exists that is correlated with any of the input variables. Under the first four assumptions, the OLS method delivers **unbiased** estimates: including an irrelevant variable does not bias the intercept and slope estimates, but omitting a relevant variable will bias the OLS estimates. OLS is then also **consistent**: as the sample size increases, the estimates converge to the true value as the standard errors become arbitrary. The converse is unfortunately also true: if the conditional expectation of the error is not zero because the model misses a relevant variable or the functional form is wrong (that is, quadratic or log terms are missing), then all parameter estimates are biased. If the error is correlated with any of the input variables then OLS is also not consistent, that is, adding more data will not remove the bias.

If we add the fifth assumptions, then OLS also produces the best linear, unbiased estimates (BLUE), where best means that the estimates have the lowest standard error among all linear estimators. Hence, if the five assumptions hold and statistical inference is the goal, then the OLS estimates is the way to go. If the goal, however, is to predict, then we will see that other estimators exist that trade off some bias for a lower variance to achieve superior predictive performance in many settings.

Now that we have introduced the basic OLS assumptions, we can take a look at inference in small and large samples.

How to conduct statistical inference

Inference in the linear regression context aims to draw conclusions about the true relationship in the population from the sample data. This includes tests of hypothesis about the significance of the overall relationship or the values of particular coefficients, as well as estimates of confidence intervals.

The key ingredient for statistical inference is a test statistic with a known distribution. We can use it to assume that the null hypothesis is true and compute the probability of observing the value for this statistic in the sample, familiar as the p-value. If the p-value drops below a significance threshold (typically five percent) then we reject the hypothesis because it makes the actual sample value very unlikely. At the same time, we accept that the p-value reflects the probability that we are wrong in rejecting what is, in fact, a correct hypothesis.

In addition to the five GMT assumptions, the classical linear model assumes **normality**—the population error is normally distributed and independent of the input variables. This assumption implies that the output variable is normally distributed, conditional on the input variables. This strong assumption permits the derivation of the exact distribution of the coefficients, which in turn implies exact distributions of the test statistics required for similarly exact hypotheses tests in small samples. This assumption often fails—asset returns, for instance, are not normally distributed—but, fortunately, the methods used under normality are also approximately valid.

We have the following distributional characteristics and test statistics, approximately under GMT assumptions 1–5, and exactly when normality holds:

- The parameter estimates follow a multivariate normal distribution: $\hat{\beta} \sim N(\beta, (\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}\sigma)$.
- Under GMT 1–5, the parameter estimates are already unbiased and we can get an unbiased estimate of σ, the constant error variance, using $\hat{\sigma} = \frac{1}{N-p-1}\sum_{i=i}^{N}(y_i - \hat{y}_i)^2$.
- The t statistic for a hypothesis tests about an individual coefficient β_j is $t_j = \frac{\hat{\beta}_j}{\hat{\sigma}\sqrt{v_j}} \sim t_{N-p-1}$ and follows a t distribution with *N-p-1* degrees of freedom where v_j is the j's element of the diagonal of $(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}$.
- The *t* distribution converges to the normal distribution and since the 97.5 quantile of the normal distribution is 1.96, a useful rule of thumb for a 95% confidence interval around a parameter estimate is $\hat{\beta} \pm 2 \cdot \text{se}(\hat{\beta})$. An interval that includes zero implies that we can't reject the null hypothesis that the true parameter is zero and, hence, irrelevant for the model.
- The *F* statistic allows for tests of restrictions on several parameters, including whether the entire regression is significant. It measures the change (reduction) in the RSS that results from additional variables.
- Finally, the **Lagrange Multiplier** (**LM**) test is an alternative to the *F* test to restrict multiple restrictions.

How to diagnose and remedy problems

Diagnostics validate the model assumptions and prevent wrong conclusions when interpreting the result and conducting statistical inference. They include measures of goodness of fit and various tests of the assumptions about the error term, including how closely the residuals match a normal distribution. Furthermore, diagnostics test whether the residual variance is indeed constant or exhibits heteroskedasticity, and if the errors are conditionally uncorrelated or exhibit serial correlation, that is, if knowing one error helps to predict consecutive errors.

In addition to the tests outlined as follows, it is always important to visually inspect the residuals to detect whether there are systematic patterns because these indicate that the model is missing one or more factors that drive the outcome.

Goodness of fit

Goodness-of-fit measures assess how well a model explains the variation in the outcome. They help to assess the quality of model specification, for instance, to select among different model designs. They differ in how they evaluate the fit. The measures discussed here provide in-sample information; we will use out-of-sample testing and cross-validation when we focus on predictive models in the next section.

Prominent goodness-of-fit measures include the (adjusted) R^2 that should be maximized and is based on the least-squares estimate:

- R^2 measures the share of the variation in the outcome data explained by the model and is computed as $R^2 = 1 - \frac{\text{RSS}}{\text{TSS}}$, where TSS is the sum of squared deviations of the outcome from its mean. It also corresponds to the squared correlation coefficient between the actual outcome values and those estimated (fitted) by the model. The goals is to maximize R^2 but it never decreases as the model adds more variables and, hence, encourages overfitting.
- The adjusted R^2 penalizes R^2 for adding more variables; each additional variable needs to reduce RSS significantly to produce better goodness of fit.

Alternatively, the Akaike (AIC) and the **Bayesian Information Criterion** (**BIC**) are to be minimized and are based on the maximum-likelihood estimate:

- $\text{AIC} = -2\log(\mathcal{L}^*) + 2k$, where $\mathcal{L}^*$ is the value of the maximized likelihood function, k is the number of parameters
- $\text{BIC} = -2\log(\mathcal{L}^*) + \log(N)k$ where N is the sample size

Both metrics penalize for complexity, with BIC imposing a higher penalty so that it might underfit whereas AIC might overfit in relative terms. Conceptually, AIC aims at finding the model that best describes an unknown data-generating process, whereas BIC tries to find the best model among the set of candidates. In practice, both criteria can be used jointly to guide model selection when the goal is in-sample fit; otherwise, cross-validation and selection based on estimates of generalization error are preferable.

Heteroskedasticity

GMT assumption 5 requires the residual covariance to take the shape $\Sigma = \sigma^2 \boldsymbol{I}$, that is, a diagonal matrix with entries equal to the constant variance of the error term. Heteroskedasticity occurs when the residual variance is not constant but differs across observations. If the residual variance is positively correlated with an input variable, that is, when errors are larger for input values that are far from their mean, then OLS standard error estimates will be too low, and, consequently, the t-statistic will be inflated leading to false discoveries of relationships where none actually exist.

Diagnostics starts with a visual inspection of the residuals. Systematic patterns in the (supposedly random) residuals suggest statistical tests of the null hypothesis that errors are homoscedastic against various alternatives. These tests include the Breusch—Pagan and White tests.

There are several ways to correct OLS estimates for heteroskedasticity:

- Robust standard errors (sometimes called white standard errors) take heteroskedasticity into account when computing the error variance using a so-called **sandwich estimator**.
- Clustered standard errors assume that there are distinct groups in your data that are homoskedastic but the error variance differs between groups. These groups could be different asset classes or equities from different industries.

Several alternatives to OLS estimate the error covariance matrix using different assumptions when $\Sigma \neq \sigma^2 \boldsymbol{I}$. The following are available in `statsmodels`:

- **Weighted least squares** (**WLS**): For heteroskedastic errors where the covariance matrix has only diagonal entries as for OLS, but now the entries are allowed to vary
- Feasible **generalized least squares** (**GLSAR**), for autocorrelated errors that follow an autoregressive AR (p) process (see the chapter on linear time series models)
- **Generalized least squares** (**GLS**) for arbitrary covariance matrix structure; yields efficient and unbiased estimates in the presence of heteroskedasticity or serial correlation

Serial correlation

Serial correlation means that consecutive residuals produced by linear regression are correlated, which violates the fourth GMT assumption. Positive serial correlation implies that the standard errors are underestimated and the t-statistics will be inflated, leading to false discoveries if ignored. However, there are procedures to correct for serial correlation when calculating standard errors.

The Durbin—Watson statistic diagnoses serial correlation. It tests the hypothesis that the OLS residuals are not autocorrelated against the alternative that they follow an autoregressive process (that we will explore in the next chapter). The test statistic ranges from 0 to 4, and values near 2 indicate non-autocorrelation, lower values suggest positive, and higher values indicate negative autocorrelation. The exact threshold values depend on the number of parameters and observations and need to be looked up in tables.

Multicollinearity

Multicollinearity occurs when two or more independent variables are highly correlated. This poses several challenges:

- It is difficult to determine which factors influence the dependent variable
- The individual p values can be misleading—a p-value can be high even if the variable is important
- The confidence intervals for the regression coefficients will be excessive, possibly even including zero, making it impossible to determine the effect of an independent variable on the outcome

There is no formal or theory-based solution that corrects for multicollinearity. Instead, try to remove one or more of the correlated input variables, or increase the sample size.

How to run linear regression in practice

The accompanying notebook `linear_regression_intro.ipynb` illustrates a simple and then a multiple linear regression, the latter using both OLS and gradient descent. For the multiple regression, we generate two random input variables x_1 and x_2 that range from -50 to +50, and an outcome variable calculated as a linear combination of the inputs plus random Gaussian noise to meet the normality assumption GMT 6:

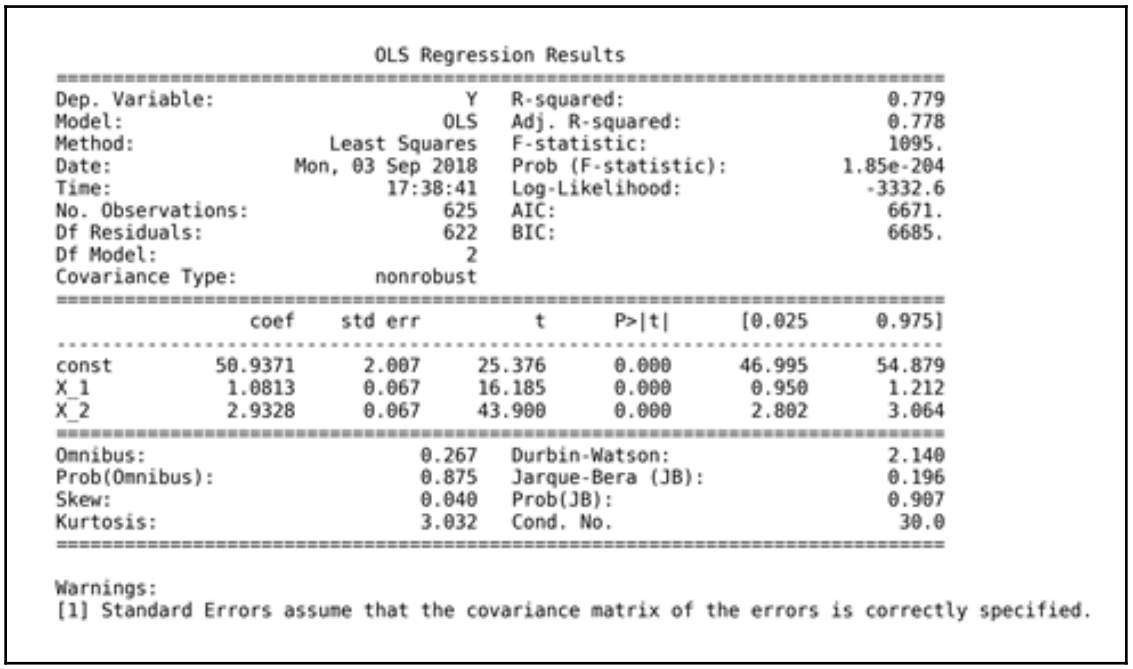
$$y = 50 + x_1 + 3x_2 + \epsilon, \qquad \epsilon \sim N(0, 50)$$

OLS with statsmodels

We use `statsmodels` to estimate a multiple regression model that accurately reflects the data generating process as follows:

```
from statsmodels.api import
X_ols = add_constant(X)
model = OLS(y, X_ols).fit()
model.summary()
```

This yields the following **OLS Regression Results** summary:



OLS Regression Results

Dep. Variable:	Y	R-squared:	0.779
Model:	OLS	Adj. R-squared:	0.778
Method:	Least Squares	F-statistic:	1095.
Date:	Mon, 03 Sep 2018	Prob (F-statistic):	1.85e-204
Time:	17:38:41	Log-Likelihood:	-3332.6
No. Observations:	625	AIC:	6671.
Df Residuals:	622	BIC:	6685.
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	50.9371	2.007	25.376	0.000	46.995	54.879
X_1	1.0813	0.067	16.185	0.000	0.950	1.212
X_2	2.9328	0.067	43.900	0.000	2.802	3.064

Omnibus:	0.267	Durbin-Watson:	2.140
Prob(Omnibus):	0.875	Jarque-Bera (JB):	0.196
Skew:	0.040	Prob(JB):	0.907
Kurtosis:	3.032	Cond. No.	30.0

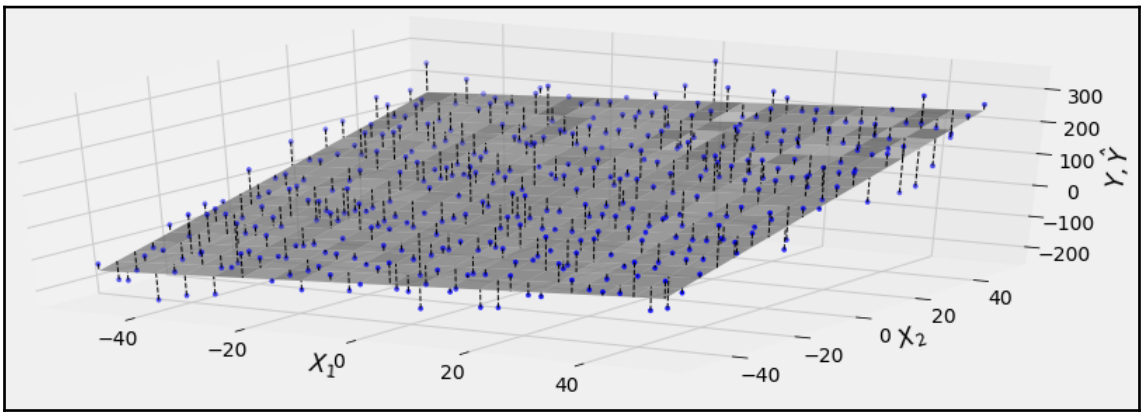
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Summary of OLS Regression Results

The upper part of the summary displays the dataset characteristics, namely the estimation method, the number of observations and parameters, and indicates that standard error estimates do not account for heteroskedasticity. The middle panel shows the coefficient values that closely reflect the artificial data generating process. We can confirm that the estimates displayed in the middle of the summary result can be obtained using the OLS formula derived previously:

```
beta = np.linalg.inv(X_ols.T.dot(X_ols)).dot(X_ols.T.dot(y))
pd.Series(beta, index=X_ols.columns)

const   50.94
X_1      1.08
X_2      2.93
```

The following diagram illustrates the hyperplane fitted by the model to the randomly generated data points:



Hyperplane

The upper right part of the panel displays the goodness-of-fit measures just discussed, alongside the F-test that rejects the hypothesis that all coefficients are zero and irrelevant. Similarly, the t-statistics indicate that intercept and both slope coefficients are, unsurprisingly, highly significant.

The bottom part of the summary contains the residual diagnostics. The left panel displays skew and kurtosis that are used to test the normality hypothesis. Both the Omnibus and the Jarque—Bera test fails to reject the null hypothesis that the residuals are normally distributed. The Durbin—Watson statistic tests for serial correlation in the residuals and has a value near 2 which, given 2 parameters and 625 observations, fails to reject the hypothesis of no serial correlation.

Lastly, the condition number provides evidence about multicollinearity: it is the ratio of the square roots of the largest and the smallest eigenvalue of the design matrix that contains the input data. A value above 30 suggests that the regression may have significant multicollinearity.

`statsmodels` includes additional diagnostic tests that are linked in the notebook.

Stochastic gradient descent with sklearn

The `sklearn` library includes an `SGDRegressor` model in its `linear_models` module. To learn the parameters for the same model using this method, we need to first standardize the data because the gradient is sensitive to the scale. We use `StandardScaler()` for this purpose that computes the mean and the standard deviation for each input variable during the fit step, and then subtracts the mean and divides by the standard deviation during the transform step that we can conveniently conduct in a single `fit_transform()` command:

```
scaler = StandardScaler()
X_ = scaler.fit_transform(X)
```

Then we instantiate the `SGDRegressor` using the default values except for a `random_state` setting to facilitate replication:

```
sgd = SGDRegressor(loss='squared_loss', fit_intercept=True,
                   shuffle=True, random_state=42,  # shuffle training data
for better gradient estimates
                   learning_rate='invscaling',     # reduce learning rate
over time
                   eta0=0.01, power_t=0.25)        # parameters for
learning rate path
```

Now we can fit the `sgd` model, create the in-sample predictions for both the OLS and the `sgd` models, and compute the root mean squared error for each:

```
sgd.fit(X=X_, y=y)
resids = pd.DataFrame({'sgd': y - sgd.predict(X_),
                       'ols': y - model.predict(sm.add_constant(X))})
resids.pow(2).sum().div(len(y)).pow(.5)

ols   50.06
sgd   50.06
```

As expected, both models yield the same result. We will now take on a more ambitious project using linear regression to estimate a multi-factor asset pricing model.

How to build a linear factor model

Algorithmic trading strategies use **linear factor models** to quantify the relationship between the return of an asset and the sources of risk that represent the main drivers of these returns. Each factor risk carries a premium, and the total asset return can be expected to correspond to a weighted average of these risk premia.

There are several practical applications of factor models across the portfolio management process from construction and asset selection to risk management and performance evaluation. The importance of factor models continues to grow as common risk factors are now tradeable:

- A summary of the returns of many assets by a much smaller number of factors reduces the amount of data required to estimate the covariance matrix when optimizing a portfolio
- An estimate of the exposure of an asset or a portfolio to these factors allows for the management of the resultant risk, for instance by entering suitable hedges when risk factors are themselves traded
- A factor model also permits the assessment of the incremental signal content of new alpha factors
- A factor model can also help assess whether a manager's performance relative to a benchmark is indeed due to skill in selecting assets and timing the market, or if instead, the performance can be explained by portfolio tilts towards known return drivers that can today be replicated as low-cost, passively managed funds without incurring active management fees

The following examples apply to equities, but risk factors have been identified for all asset classes (see references in the GitHub repository).

From the CAPM to the Fama—French five-factor model

Risk factors have been a key ingredient to quantitative models since the **Capital Asset Pricing Model** (**CAPM**) explained the expected returns of all N assets $r_i, \quad i = 1, \ldots, N$ using their respective exposure β_i to a single factor, the expected excess return of the overall market over the risk-free rate r_f. The model takes the following linear form:

$$E[r_i] = \alpha_i + r_f + \beta_i(E[r_m] - r_f)$$

This differs from classic fundamental analysis a la Dodd and Graham where returns depend on firm characteristics. The rationale is that, in the aggregate, investors cannot eliminate this so-called systematic risk through diversification. Hence, in equilibrium, they require compensation for holding an asset commensurate with its systematic risk. The model implies that, given efficient markets where prices immediately reflect all public information, there should be no superior risk-adjusted returns, that is, the value of α should be zero.

Empirical tests of the model use linear regression and have consistently failed, prompting a debate whether the efficient markets or the single factor aspect of the joint hypothesis is to blame. It turns out that both premises are probably wrong:

- Joseph Stiglitz earned the 2001 Nobel Prize in economics in part for showing that markets are generally not perfectly efficient: if markets are efficient, there is no value in collecting data because this information is already reflected in prices. However, if there is no incentive to gather information, it is hard to see how it should be already reflected in prices.
- On the other hand, theoretical and empirical improvements on the CAPM suggest that additional factors help explain some of the anomalies that consisted in superior risk-adjusted returns that do not depend on overall market exposure, such as higher returns for smaller firms.

Stephen Ross proposed the **Arbitrage Pricing Theory** (**APT**) in 1976 as an alternative that allows for several risk factors while eschewing market efficiency. In contrast to the CAPM, it assumes that opportunities for superior returns due to mispricing may exist but will quickly be arbitraged away. The theory does not specify the factors, but research by the author suggests that the most important are changes in inflation and industrial production, as well as changes in risk premia or the term structure of interest rates.

Kenneth French and Eugene Fama (who won the 2013 Nobel Prize) identified additional risk factors that depend on firm characteristics and are widely used today. In 1993, the Fama—French three-factor model added the relative size and value of firms to the single CAPM source of risk. In 2015, the five-factor model further expanded the set to include firm profitability and level of investment that had been shown to be significant in the intervening years. In addition, many factor models include a price momentum factor.

The Fama—French risk factors are computed as the return difference on diversified portfolios with high or low values according to metrics that reflect a given risk factor. These returns are obtained by sorting stocks according to these metrics and then going long stocks above a certain percentile while shorting stocks below a certain percentile. The metrics associated with the risk factors are defined as follows:

- **Size**: **Market Equity** (**ME**)
- **Value**: **Book Value of Equity** (**BE**) divided by ME
- **Operating Profitability (OP)**: Revenue minus cost of goods sold/assets
- **Investment**: Investment/assets

There are also unsupervised learning techniques for a data-driven discovery of risk factors using factors and principal component analysis that we will explore in `Chapter 12`, *Unsupervised Learning.*

Obtaining the risk factors

Fama and French make updated risk factor and research portfolio data available through their website, and you can use the `pandas_datareader` library to obtain the data. For this application, refer to the `fama_macbeth.ipynb` notebook for additional detail.

In particular, we will be using the five Fama—French factors that result from sorting stocks first into three size groups and then into two for each of the remaining three firm-specific factors. Hence, the factors involve three sets of value-weighted portfolios formed as 3 x 2 sorts on size and book-to-market, size and operating profitability, and size and investment. The risk factor values computed as the average returns of the **portfolios** (**PF**) as outlined in the following table:

Concept	Label	Name	Risk factor calculation
Size	SMB	Small minus big	Nine small stock PF minus nine large stock PF
Value	HML	High minus low	Two value PF minus two growth (with low BE/ME value) PF
Profitability	RMW	Robust minus weak	Two robust OP PF minus two weak OP PF
Investment	CMA	Conservative minus aggressive	Two conservative investment portfolios minus two aggressive investment portfolios
Market	Rm-Rf	Excess return on the market	Value-weight return of all firms incorporated in and listed on major US exchanges with good data minus the one-month Treasury bill rate

We will use returns at a monthly frequency that we obtain for the period 2010 – 2017 as follows:

```
import pandas_datareader.data as web
ff_factor = 'F-F_Research_Data_5_Factors_2x3'
ff_factor_data = web.DataReader(ff_factor, 'famafrench', start='2010',
end='2017-12')[0]
ff_factor_data.info()

PeriodIndex: 96 entries, 2010-01 to 2017-12
Freq: M
Data columns (total 6 columns):
Mkt-RF 96 non-null float64
SMB 96 non-null float64
HML 96 non-null float64
RMW 96 non-null float64
CMA 96 non-null float64
```

```
RF 96 non-null float64
```

Fama and French also make available numerous portfolios that we can illustrate the estimation of the factor exposures, as well as the value of the risk premia available in the market for a given time period. We will use a panel of the 17 industry portfolios at a monthly frequency. We will subtract the risk-free rate from the returns because the factor model works with excess returns:

```
ff_portfolio = '17_Industry_Portfolios'
ff_portfolio_data = web.DataReader(ff_portfolio, 'famafrench',
start='2010', end='2017-12')[0]
ff_portfolio_data = ff_portfolio_data.sub(ff_factor_data.RF, axis=0)
ff_factor_data = ff_factor_data.drop('RF', axis=1)
ff_portfolio_data.info()

PeriodIndex: 96 entries, 2010-01 to 2017-12
Freq: M
Data columns (total 17 columns):
Food     96 non-null float64
Mines    96 non-null float64
Oil      96 non-null float64
...
Rtail    96 non-null float64
Finan    96 non-null float64
Other    96 non-null float64
```

We will now build a linear factor model based on this panel data using a method that addresses the failure of some basic linear regression assumptions.

Fama—Macbeth regression

Given data on risk factors and portfolio returns, it is useful to estimate the portfolio's exposure, that is, how much the risk factors drive portfolio returns, as well as how much the exposure to a given factor is worth, that is, the what market's risk factor premium is. The risk premium then permits to estimate the return for any portfolio provided the factor exposure is known or can be assumed.

More formally, we will have *i=1, ..., N* asset or portfolio returns over *t=1, ..., T* periods and each asset's excess period return will be denoted r_{it}. The goals is to test whether the *j=1, ..., M* factors F_{jt} explain the excess returns and the risk premium associated with each factor. In our case, we have *N=17* portfolios and *M=5* factors, each with =96 periods of data.

Factor models are estimated for many stocks in a given period. Inference problems will likely arise in such cross-sectional regressions because the fundamental assumptions of classical linear regression may not hold. Potential violations include measurement errors, covariation of residuals due to heteroskedasticity and serial correlation, and multicollinearity.

To address the inference problem caused by the correlation of the residuals, Fama and MacBeth proposed a two-step methodology for a cross-sectional regression of returns on factors. The two-stage Fama—Macbeth regression is designed to estimate the premium rewarded for the exposure to a particular risk factor by the market. The two stages consist of:

- **First stage**: *N* time-series regression, one for each asset or portfolio, of its excess returns on the factors to estimate the factor loadings. In matrix form, for each asset:

$$\underset{T\times 1}{\boldsymbol{r}_i} = \underset{T\times(m+1)}{\boldsymbol{F}} \underset{(m+1)\times 1}{\boldsymbol{\beta}_i} + \underset{T\times 1}{\boldsymbol{\epsilon}_i}$$

- **Second stage**: T cross-sectional regression, one for each time period, to estimate the risk premium. In matrix form, we obtain a vector $\hat{\boldsymbol{\lambda}}_t$ of risk premia for each period:

$$\underset{N\times(M+1}{\boldsymbol{r}_t} = \underset{N\times(M+1)(M+1)\times 1}{\hat{\boldsymbol{\beta}}} \boldsymbol{\lambda}_t$$

Now we can compute the factor risk premia as the time average and get t-statistic to assess their individual significance, using the assumption that the risk premia estimates are independent over time: $t = \frac{\lambda_j}{\sigma(\lambda_j)/\sqrt{(T)}}$.

If we had a very large and representative data sample on traded risk factors we could use the sample mean as a risk premium estimate. However, we typically do not have a sufficiently long history to and the margin of error around the sample mean could be quite large. The Fama—Macbeth methodology leverages the covariance of the factors with other assets to determine the factor premia. The second moment of asset returns is easier to estimate than the first moment, and obtaining more granular data improves estimation considerably, which is not true of mean estimation.

We can implement the first stage to obtain the 17 factor loading estimates as follows:

```
betas = []
for industry in ff_portfolio_data:
    step1 = OLS(endog=ff_portfolio_data[industry],
                exog=add_constant(ff_factor_data)).fit()
    betas.append(step1.params.drop('const'))

betas = pd.DataFrame(betas,
                     columns=ff_factor_data.columns,
                     index=ff_portfolio_data.columns)
betas.info()
Index: 17 entries, Food  to Other
Data columns (total 5 columns):
Mkt-RF    17 non-null float64
SMB       17 non-null float64
HML       17 non-null float64
RMW       17 non-null float64
CMA       17 non-null float64
```

For the second stage, we run 96 regressions of the period returns for the cross section of portfolios on the factor loadings:

```
lambdas = []
for period in ff_portfolio_data.index:
    step2 = OLS(endog=ff_portfolio_data.loc[period, betas.index],
                exog=betas).fit()
    lambdas.append(step2.params)

lambdas = pd.DataFrame(lambdas,
                       index=ff_portfolio_data.index,
                       columns=betas.columns.tolist())
lambdas.info()
PeriodIndex: 96 entries, 2010-01 to 2017-12
Freq: M
Data columns (total 5 columns):
Mkt-RF    96 non-null float64
SMB       96 non-null float64
HML       96 non-null float64
RMW       96 non-null float64
CMA       96 non-null float64
```

Finally, we compute the average for the 96 periods to obtain our factor risk premium estimates:

```
lambdas.mean()
Mkt-RF    1.201304
SMB       0.190127
HML      -1.306792
RMW      -0.570817
CMA      -0.522821
```

The `linear_models` library extends `statsmodels` with various models for panel data and also implements the two-stage Fama—MacBeth procedure:

```
model = LinearFactorModel(portfolios=ff_portfolio_data,
                          factors=ff_factor_data)
res = model.fit()
```

This provides us with the same result:

```
                     LinearFactorModel Estimation Summary
================================================================================
No. Test Portfolios:                 17   R-squared:                      0.6943
No. Factors:                          6   J-statistic:                    19.155
No. Observations:                    95   P-value                         0.0584
Date:                  Wed, Oct 31 2018   Distribution:                chi2(11)
Time:                          15:15:52
Cov. Estimator:                  robust

                            Risk Premia Estimates
================================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
--------------------------------------------------------------------------------
Mkt-RF         1.2446     0.3928     3.1689     0.0015      0.4748      2.0144
SMB            0.0074     0.7055     0.0105     0.9917     -1.3753      1.3901
HML           -0.6970     0.5334    -1.3067     0.1913     -1.7424      0.3484
RMW           -0.2558     0.6888    -0.3713     0.7104     -1.6057      1.0942
CMA           -0.3086     0.4737    -0.6515     0.5147     -1.2371      0.6198
RF            -0.0133     0.0132    -1.0092     0.3129     -0.0393      0.0126
================================================================================

Covariance estimator:
HeteroskedasticCovariance
See full_summary for complete results
```

LinearFactorModel Estimation Summary

The accompanying notebook illustrates the use of categorical variables by using industry dummies when estimating risk premia for a larger panel of individual stocks.

Shrinkage methods: regularization for linear regression

The least squares methods to train a linear regression model will produce the best, linear, and unbiased coefficient estimates when the Gauss—Markov assumptions are met. Variations like GLS fare similarly well even when OLS assumptions about the error covariance matrix are violated. However, there are estimators that produce biased coefficients to reduce the variance to achieve a lower generalization error overall.

When a linear regression model contains many correlated variables, their coefficients will be poorly determined because the effect of a large positive coefficient on the RSS can be canceled by a similarly large negative coefficient on a correlated variable. Hence, the model will have a tendency for high variance due to this wiggle room of the coefficients that increases the risk that the model overfits to the sample.

How to hedge against overfitting

One popular technique to control overfitting is that of **regularization**, which involves the addition of a penalty term to the error function to discourage the coefficients from reaching large values. In other words, size constraints on the coefficients can alleviate the resultant potentially negative impact on out-of-sample predictions. We will encounter regularization methods for all models since overfitting is such a pervasive problem.

In this section, we will introduce shrinkage methods that address two motivations to improve on the approaches to linear models discussed so far:

- **Prediction accuracy**: The low bias but high variance of least squares estimates suggests that the generalization error could be reduced by shrinking or setting some coefficients to zero, thereby trading off a slightly higher bias for a reduction in the variance of the model.
- **Interpretation**: A large number of predictors may complicate the interpretation or communication of the big picture of the results. It may be preferable to sacrifice some detail to limit the model to a smaller subset of parameters with the strongest effects.

Shrinkage models restrict the regression coefficients by imposing a penalty on their size. These models achieve this goal by adding a term to the objective function so that the coefficients of a shrinkage model minimize the RSS plus a penalty that is positively related to the (absolute) size of the coefficients. The added penalty turns finding the linear regression coefficients into a constrained minimization problem that, in general, takes the following Lagrangian form:

$$\hat{\boldsymbol{\beta}}^{\boldsymbol{S}} = \underset{\boldsymbol{\beta}^S}{\operatorname{argmin}} \sum_{i=1}^{N} \left[(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_j)^2 + \lambda S(\boldsymbol{\beta}) \right] = \underset{\boldsymbol{\beta}^S}{\operatorname{argmin}}\, \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta} - \lambda S(\boldsymbol{\beta})$$

The regularization parameter λ determines the size of the penalty effect, that is, the strength of the regularization. As soon as λ is positive, the coefficients will differ from the unconstrained least squared parameters, which implies a biased estimate. The hyperparameter λ should be adaptively chosen using cross-validation to minimize an estimate of expected prediction error.

Shrinkage models differ by how they calculate the penalty, that is, the functional form of S. The most common versions are the ridge regression that uses the sum of the squared coefficients, whereas the lasso model bases the penalty on the sum of the absolute values of the coefficients.

How ridge regression works

The ridge regression shrinks the regression coefficients by adding a penalty to the objective function that equals the sum of the squared coefficients, which in turn corresponds to the L^2 norm of the coefficient vector:

$$S(\boldsymbol{\beta}) = \sum_{i=1}^{p} \beta_i^2 = \|\boldsymbol{\beta}\|^2$$

Hence, the ridge coefficients are defined as:

$$\hat{\boldsymbol{\beta}}^{\text{Ridge}} = \underset{\boldsymbol{\beta}^{\text{Ridge}}}{\operatorname{argmin}} \sum_{i=1}^{N} \left[(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_j)^2 + \lambda \sum_{j=1}^{p} \boldsymbol{\beta}_j^2 \right] = \underset{\boldsymbol{\beta}^{\text{Ridge}}}{\operatorname{argmin}} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda \boldsymbol{\beta}^T \boldsymbol{\beta}$$

The intercept β_0 has been excluded from the penalty to make the procedure independent of the origin chosen for the output variable—otherwise, adding a constant to all output values would change all slope parameters as opposed to a parallel shift.

It is important to standardize the inputs by subtracting from each input the corresponding mean and dividing the result by the input's standard deviation because the ridge solution is sensitive to the scale of the inputs. There is also a closed solution for the ridge estimator that resembles the OLS case:

$$\hat{\boldsymbol{\beta}}^{\text{Ridge}} = (\boldsymbol{X}^T \boldsymbol{X} + \lambda I)^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

The solution adds the scaled identity matrix λI to $X^T X$ before inversion, which guarantees that the problem is non-singular, even if $X^T X$ does not have full rank. This was one of the motivations for using this estimator when it was originally introduced.

The ridge penalty results in proportional shrinkage of all parameters. In the case of **orthonormal inputs**, the ridge estimates are just a scaled version of the least squares estimates, that is:

$$\hat{\boldsymbol{\beta}}^{\text{Ridge}} = \frac{\hat{\boldsymbol{\beta}}^{\text{LS}}}{1 + \lambda}$$

Using the **singular value decomposition** (**SVD**) of the input matrix X, we can gain insight into how the shrinkage affects inputs in the more common case where they are not orthonormal. The SVD of a centered matrix represents the principal components of a matrix (refer to `Chapter 11`, *Gradient Boosting Machines*, on unsupervised learning) that capture uncorrelated directions in the column space of the data in descending order of variance.

Ridge regression shrinks coefficients on input variables that are associated with directions in the data that have less variance more than input variables that correlate with directions that exhibit more variance. Hence, the implicit assumption of ridge regression is that the directions in the data that vary the most will be most influential or most reliable when predicting the output.

How lasso regression works

The lasso, known as basis pursuit in signal processing, also shrinks the coefficients by adding a penalty to the sum of squares of the residuals, but the lasso penalty has a slightly different effect. The lasso penalty is the sum of the absolute values of the coefficient vector, which corresponds to its L^1 norm. Hence, the lasso estimate is defined by:

$$\hat{\boldsymbol{\beta}}^{\text{Lasso}} = \underset{\boldsymbol{\beta}^{\text{Lasso}}}{\operatorname{argmin}} \sum_{i=1}^{N} \left[(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right] = \underset{\boldsymbol{\beta}^{\text{Ridge}}}{\operatorname{argmin}} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda |\boldsymbol{\beta}|$$

Similarly to ridge regression, the inputs need to be standardized. The lasso penalty makes the solution nonlinear, and there is no closed-form expression for the coefficients as in ridge regression. Instead, the lasso solution is a quadratic programming problem and there are available efficient algorithms that compute the entire path of coefficients that result for different values of λ with the same computational cost as for ridge regression.

The lasso penalty had the effect of gradually reducing some coefficients to zero as the regularization increases. For this reason, the lasso can be used for the continuous selection of a subset of features.

How to use linear regression to predict returns

The notebook `linear_regression.ipynb` contains examples for the prediction of stock prices using OLS with `statsmodels` and `sklearn`, as well as ridge and lasso models. It is designed to run as a notebook on the Quantopian research platform and relies on the `factor_library` introduced in `Chapter 4`, *Alpha Factors Research*.

Prepare the data

We need to select a universe of equities and a time horizon, build and transform alpha factors that we will use as features, calculate forward returns that we aim to predict, and potentially clean our data.

Universe creation and time horizon

We will use equity data for the years 2014 and 2015 from a custom `Q100US` universe that uses built-in filters, factors, and classifiers to select the 100 stocks with the highest average dollar volume of the last 200 trading days filtered by additional default criteria (see Quantopian docs linked on GitHub for detail). The universe dynamically updates based on the filter criteria so that, while there are 100 stocks at any given point, there may be more than 100 distinct equities in the sample:

```
def Q100US():
    return filters.make_us_equity_universe(
        target_size=100,
        rankby=factors.AverageDollarVolume(window_length=200),
        mask=filters.default_us_equity_universe_mask(),
        groupby=classifiers.fundamentals.Sector(),
        max_group_weight=0.3,
        smoothing_func=lambda f: f.downsample('month_start'),
    )
```

Target return computation

We will test predictions for various `lookahead` periods to identify the best holding periods that generate the best predictability, measured by the information coefficient. More specifically, we compute returns for 1, 5, 10, and 20 days using the built-in `Returns` function, resulting in over 50,000 observations for the universe of 100 stocks over two years (that include approximately 252 trading days each):

```
lookahead = [1, 5, 10, 20]
returns = run_pipeline(Pipeline({'Returns{}D'.format(i):
Returns(inputs=[USEquityPricing.close],
                                            window_length=i+1, mask=UNIVERSE)
for i in lookahead},
                               screen=UNIVERSE),
                      start_date=START,
                      end_date=END)
return_cols = ['Returns{}D'.format(i) for i in lookahead]
returns.info()

MultiIndex: 50362 entries, (2014-01-02 00:00:00+00:00, Equity(24 [AAPL]))
to (2015-12-31 00:00:00+00:00, Equity(47208 [GPRO]))
Data columns (total 4 columns):
Returns10D    50362 non-null float64
Returns1D     50362 non-null float64
Returns20D    50360 non-null float64
Returns5D     50362 non-null float64
```

Alpha factor selection and transformation

We will use over 50 features that cover a broad range of factors based on market, fundamental, and alternative data. The notebook also includes custom transformations to convert fundamental data that is typically available in quarterly reporting frequency to rolling annual totals or averages to avoid excessive season fluctuations.

Once the factors have been computed through the various pipelines outlined in `Chapter 4`, *Alpha Factors Research*, we combine them using `pd.concat()`, assign index names, and create a categorical variable that identifies the asset for each data point:

```
data = pd.concat([returns, value_factors, momentum_factors,
                  quality_factors, payout_factors, growth_factors,
                  efficiency_factors, risk_factors], axis=1).sortlevel()
data.index.names = ['date', 'asset']
data['stock'] = data.index.get_level_values('asset').map(lambda x:
x.asset_name)
```

Data cleaning – missing data

In a next step, we remove rows and columns that lack more than 20 percent of the observations, resulting in a loss of six percent of the observations and three columns:

```
rows_before, cols_before = data.shape
data = (data
        .dropna(axis=1, thresh=int(len(data) * .8))
        .dropna(thresh=int(len(data.columns) * .8)))
data = data.fillna(data.median())
rows_after, cols_after = data.shape
print('{:,d} rows and {:,d} columns dropped'.format(rows_before -
rows_after, cols_before - cols_after))
2,985 rows and 3 columns dropped
```

At this point, we have 51 features and the categorical identifier of the stock:

```
data.sort_index(1).info()

MultiIndex: 47377 entries, (2014-01-02, Equity(24 [AAPL])) to (2015-12-
                           31, Equity(47208 [GPRO]))
Data columns (total 52 columns):
AssetToEquityRatio             47377 non-null float64
AssetTurnover                  47377 non-null float64
CFO To Assets                  47377 non-null float64
...
WorkingCapitalToAssets         47377 non-null float64
WorkingCapitalToSales          47377 non-null float64
```

```
stock                          47377 non-null object
dtypes: float64(51), object(1)
```

Data exploration

For linear regression models, it is important to explore the correlation among the features to identify multicollinearity issues, and to check the correlation between the features and the target. The notebook contains a seaborn clustermap that shows the hierarchical structure of the feature correlation matrix. It identifies a small number of highly correlated clusters.

Dummy encoding of categorical variables

We need to convert the categorical `stock` variable into a numeric format so that the linear regression can process it. For this purpose, we use dummy encoding that creates individual columns for each category level and flags the presence of this level in the original categorical column with an entry of `1`, and `0` otherwise. The pandas function `get_dummies()` automates dummy encoding. It detects and properly converts columns of type objects as illustrated next. If you need dummy variables for columns containing integers, for instance, you can identify them using the keyword `columns`:

```
df = pd.DataFrame({'categories': ['A','B', 'C']})

  categories
0          A
1          B
2          C

pd.get_dummies(df)

   categories_A  categories_B  categories_C
0             1             0             0
1             0             1             0
2             0             0             1
```

When converting all categories to dummy variables and estimating the model with an intercept (as you typically would), you inadvertently create multicollinearity: the matrix now contains redundant information and no longer has full rank, that is, becomes singular. It is simple to avoid this by removing one of the new indicator columns. The coefficient on the missing category level will now be captured by the intercept (which is always `1` when every other category dummy is `0`). Use the `drop_first` keyword to correct the dummy variables accordingly:

```
pd.get_dummies(df, drop_first=True)

   categories_B  categories_C
0             0             0
1             1             0
2             0             1
```

Applied to our combined features and returns, we obtain 181 columns because there are more than 100 stocks as the universe definition automatically updates the stock selection:

```
X = pd.get_dummies(data.drop(return_cols, axis=1), drop_first=True)
X.info()

MultiIndex: 47377 entries, (2014-01-02 00:00:00+00:00, Equity(24 [AAPL]))
to (2015-12-31 00:00:00+00:00, Equity(47208 [GPRO]))
Columns: 181 entries, DividendYield to stock_YELP INC
dtypes: float64(182)
memory usage: 66.1+ MB
```

Creating forward returns

The goal is to predict returns over a given holding period. Hence, we need to align the features with return values with the corresponding return data point 1, 5, 10, or 20 days into the future for each equity. We achieve this by combining the pandas `.groupby()` method with the `.shift()` method as follows:

```
y = data.loc[:, return_cols]
shifted_y = []
for col in y.columns:
    t = int(re.search(r'\d+', col).group(0))
shifted_y.append(y.groupby(level='asset')['Returns{}D'.format(t)].shift(-
t).to_frame(col))
y = pd.concat(shifted_y, axis=1)
y.info()

MultiIndex: 47377 entries, (2014-01-02, Equity(24 [AAPL])) to (2015-12-31,
Equity(47208 [GPRO]))
```

```
Data columns (total 4 columns):
Returns1D     47242 non-null float64
Returns5D     46706 non-null float64
Returns10D    46036 non-null float64
Returns20D    44696 non-null float64
dtypes: float64(4)
```

There are now different numbers of observations for each return series as the forward shift has created missing values at the tail end for each equity.

Linear OLS regression using statsmodels

We can estimate a linear regression model using OLS with `statsmodels` as demonstrated previously. We select a forward return, for example for a 10-day holding period, remove outliers below the 2.5% and above the 97.5% percentiles, and fit the model accordingly:

```
target = 'Returns10D'
model_data = pd.concat([y[[target]], X], axis=1).dropna()
model_data =
model_data[model_data[target].between(model_data[target].quantile(.025),
model_data[target].quantile(.975))]

model = OLS(endog=model_data[target], exog=model_data.drop(target, axis=1))
trained_model = model.fit()
trained_model.summary()
```

Diagnostic statistics

The summary is available in the notebook to save some space due to the large number of variables. The diagnostic statistics show that, given the high p-value on the Jarque—Bera statistic, the hypothesis that the residuals are normally distributed cannot be rejected.

However, the Durbin—Watson statistic is low at 1.5 so we can reject the null hypothesis of no autocorrelation comfortably at the 5% level. Hence, the standard errors are likely positively correlated. If our goal were to understand which factors are significantly associated with forward returns, we would need to rerun the regression using robust standard errors (a parameter in `statsmodels .fit()` method), or use a different method altogether such as a panel model that allows for more complex error covariance.

Linear OLS regression using sklearn

Since sklearn is tailored towards prediction, we will evaluate the linear regression model based on its predictive performance using cross-validation.

Custom time series cross-validation

Our data consists of grouped time series data that requires a custom cross-validation function to provide the train and test indices that ensure that the test data immediately follows the training data for each equity and we do not inadvertently create a look-ahead bias or leakage.

We can achieve this using the following function that returns a `generator` yielding pairs of train and test dates. The set of train dates that ensure a minimum length of the training periods. The number of pairs depends on the parameter `nfolds`. The distinct test periods do not overlap and are located at the end of the period available in the data. After a test period is used, it becomes part of the training data that grow in size accordingly:

```
def time_series_split(d=model_data, nfolds=5, min_train=21):
    """Generate train/test dates for nfolds
    with at least min_train train obs
    """
    train_dates = d[:min_train].tolist()
    n = int(len(dates)/(nfolds + 1)) + 1
    test_folds = [d[i:i + n] for i in range(min_train, len(d), n)]
    for test_dates in test_folds:
        if len(train_dates) > min_train:
            yield train_dates, test_dates
        train_dates.extend(test_dates)
```

Select features and target

We need to select the appropriate return series (we will again use a 10-day holding period) and remove outliers. We will also convert returns to log returns as follows:

```
target = 'Returns10D'
outliers = .01
model_data = pd.concat([y[[target]], X],
axis=1).dropna().reset_index('asset', drop=True)
model_data =
model_data[model_data[target].between(*model_data[target].quantile([outlier
s, 1-outliers]).values)]

model_data[target] = np.log1p(model_data[target])
```

```
features = model_data.drop(target, axis=1).columns
dates = model_data.index.unique()

DatetimeIndex: 45114 entries, 2014-01-02 to 2015-12-16
Columns: 183 entries, Returns10D to stock_YELP INC
dtypes: float64(183)
```

Cross-validating the model

We will use 250 folds to generally predict about 2 days of forward returns following the historical training data that will gradually increase in length. Each iteration obtains the appropriate training and test dates from our custom cross-validation function, selects the corresponding features and targets, and then trains and predicts accordingly. We capture the root mean squared error as well as the Spearman rank correlation between actual and predicted values:

```
nfolds = 250
lr = LinearRegression()

test_results, result_idx, preds = [], [], pd.DataFrame()
for train_dates, test_dates in time_series_split(dates, nfolds=nfolds):
    X_train = model_data.loc[idx[train_dates], features]
    y_train = model_data.loc[idx[train_dates], target]
    lr.fit(X=X_train, y=y_train)

    X_test = model_data.loc[idx[test_dates], features]
    y_test = model_data.loc[idx[test_dates], target]
    y_pred = lr.predict(X_test)

    rmse = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_test))
    ic, pval = spearmanr(y_pred, y_test)

    test_results.append([rmse, ic, pval])
    preds =
preds.append(y_test.to_frame('actuals').assign(predicted=y_pred))
    result_idx.append(train_dates[-1])
```

Test results – information coefficient and RMSE

We have captured the test predictions from the 250 folds and can compute both the overall and a 21-day rolling average:

```
fig, axes = plt.subplots(nrows=2)
rolling_result = test_result.rolling(21).mean()
rolling_result[['ic', 'pval']].plot(ax=axes[0], title='Information
Coefficient')
axes[0].axhline(test_result.ic.mean(), lw=1, ls='--', color='k')
rolling_result[['rmse']].plot(ax=axes[1], title='Root Mean Squared Error')
axes[1].axhline(test_result.rmse.mean(), lw=1, ls='--', color='k')
```

We obtain the following chart that highlights the negative correlation of IC and RMSE and their respective values:

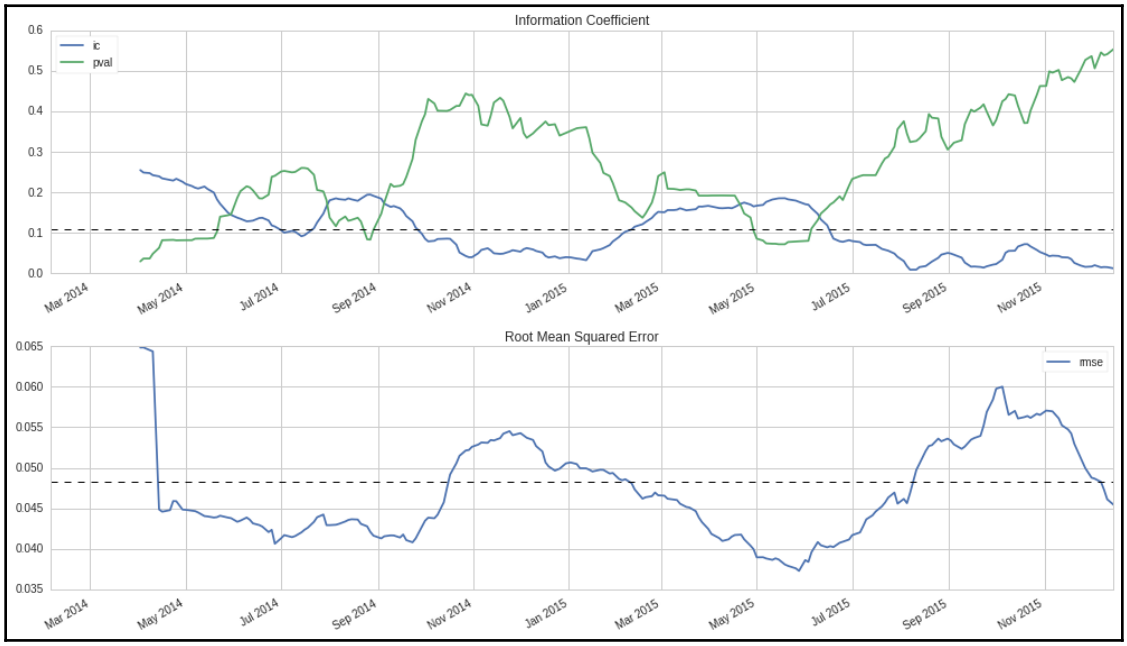
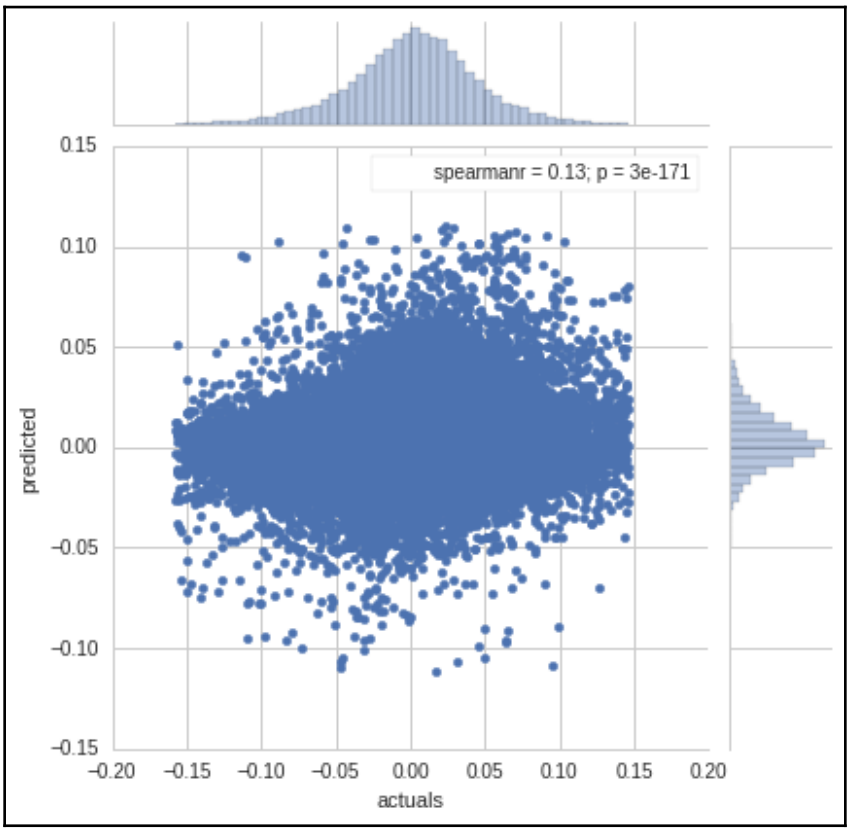


Chart highlighting the negative correlation of IC and RMSE

For the entire period, we see that the Information Coefficient measured by the rank correlation of actual and predicted returns is weakly positive and statistically significant:



Ridge regression using sklearn

For the ridge regression, we need to tune the regularization parameter with the keyword `alpha` that corresponds to the λ we used previously. We will try 21 values from 10^{-5} to 10^{5} in logarithmic steps.

The scale sensitivity of the ridge penalty requires us to standardize the inputs using the `StandardScaler`. Note that we always learn the mean and the standard deviation from the training set using the `.fit_transform()` method and then apply these learned parameters to the test set using the `.transform()` method.

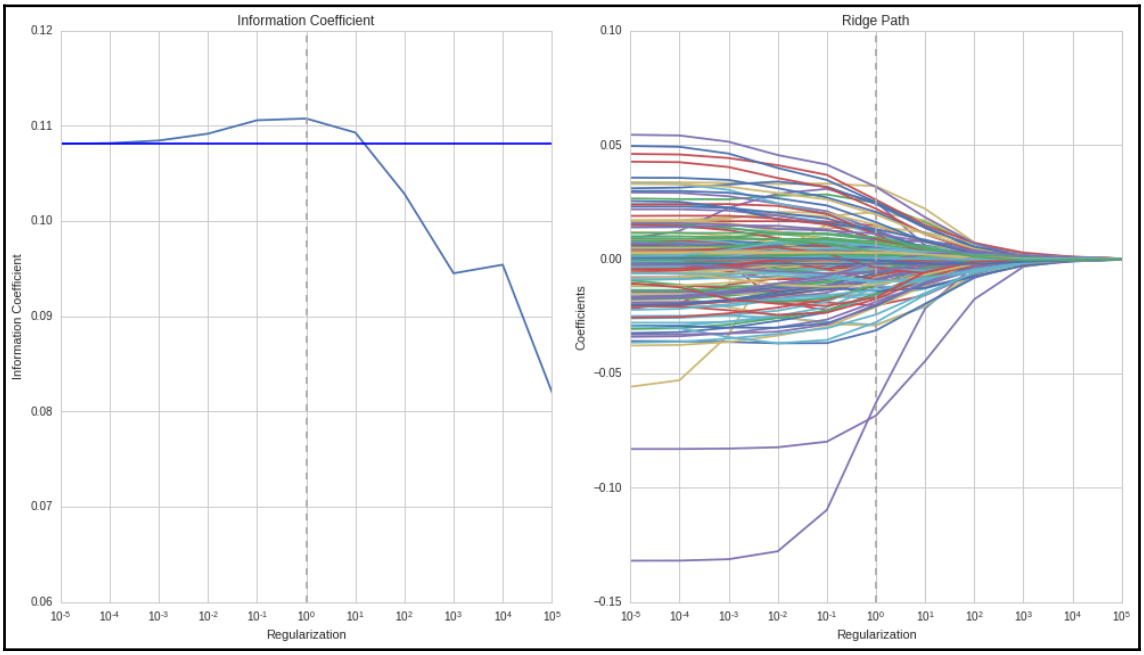
Tuning the regularization parameters using cross-validation

We then proceed to cross-validate the hyperparameter values again using `250` folds as follows:

```
nfolds = 250
alphas = np.logspace(-5, 5, 21)
scaler = StandardScaler()

ridge_result, ridge_coeffs = pd.DataFrame(), pd.DataFrame()
for i, alpha in enumerate(alphas):
    coeffs, test_results = [], []
    lr_ridge = Ridge(alpha=alpha)
    for train_dates, test_dates in time_series_split(dates, nfolds=nfolds):
        X_train = model_data.loc[idx[train_dates], features]
        y_train = model_data.loc[idx[train_dates], target]
        lr_ridge.fit(X=scaler.fit_transform(X_train), y=y_train)
        coeffs.append(lr_ridge.coef_)

        X_test = model_data.loc[idx[test_dates], features]
        y_test = model_data.loc[idx[test_dates], target]
        y_pred = lr_ridge.predict(scaler.transform(X_test))

        rmse = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_test))
        ic, pval = spearmanr(y_pred, y_test)

        test_results.append([train_dates[-1], rmse, ic, pval, alpha])
    test_results = pd.DataFrame(test_results, columns=['date', 'rmse',
'ic', 'pval', 'alpha'])
    ridge_result = ridge_result.append(test_results)
    ridge_coeffs[alpha] = np.mean(coeffs, axis=0)
```

Cross-validation results and ridge coefficient paths

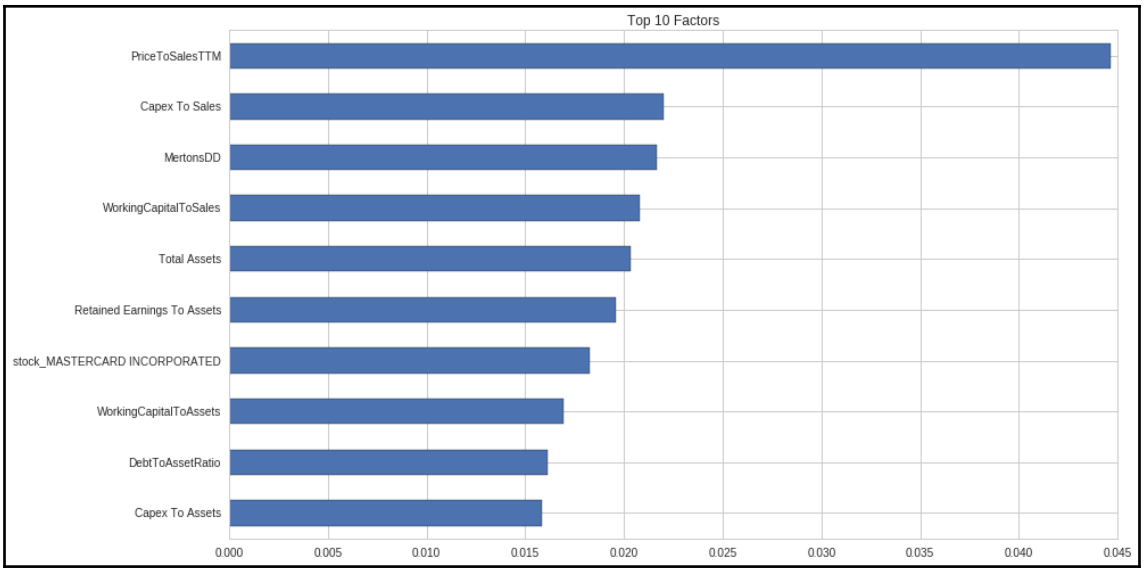
We can now plot the information coefficient obtained for each hyperparameter value and also visualize how the coefficient values evolve as the regularization increases. The results show that we get the highest IC value for a value of λ=10. For this level of regularization, the right-hand panel reveals that the coefficients have been already significantly shrunk compared to the (almost) unconstrained model with $\lambda=10^{-5}$:



Cross-validation results and ridge coefficient paths

Top 10 coefficients

The standardization of the coefficients allows us to draw conclusions about their relative importance by comparing their absolute magnitude. The 10 most relevant coefficients are:



Top 10 coefficients

Lasso regression using sklearn

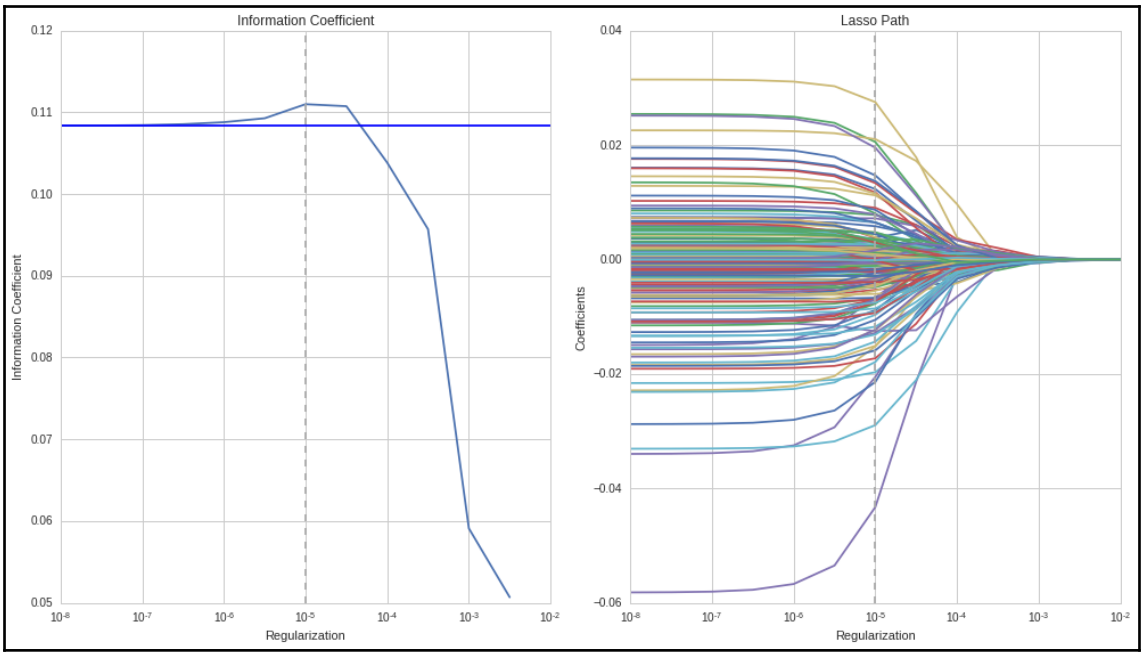
The lasso implementation looks very similar to the ridge model we just ran. The main difference is that lasso needs to arrive at a solution using iterative coordinate descent whereas ridge can rely on a closed-form solution:

```
nfolds = 250
alphas = np.logspace(-8, -2, 13)
scaler = StandardScaler()

lasso_results, lasso_coeffs = pd.DataFrame(), pd.DataFrame()
for i, alpha in enumerate(alphas):
    coeffs, test_results = [], []
    lr_lasso = Lasso(alpha=alpha)
    for i, (train_dates, test_dates) in enumerate(time_series_split(dates,
nfolds=nfolds)):
        X_train = model_data.loc[idx[train_dates], features]
        y_train = model_data.loc[idx[train_dates], target]
        lr_lasso.fit(X=scaler.fit_transform(X_train), y=y_train)

        X_test = model_data.loc[idx[test_dates], features]
        y_test = model_data.loc[idx[test_dates], target]
        y_pred = lr_lasso.predict(scaler.transform(X_test))
```

```
            rmse = np.sqrt(mean_squared_error(y_pred=y_pred, y_true=y_test))
            ic, pval = spearmanr(y_pred, y_test)

            coeffs.append(lr_lasso.coef_)
            test_results.append([train_dates[-1], rmse, ic, pval, alpha])
        test_results = pd.DataFrame(test_results, columns=['date', 'rmse',
'ic', 'pval', 'alpha'])
        lasso_results = lasso_results.append(test_results)
        lasso_coeffs[alpha] = np.mean(coeffs, axis=0)
```

Cross-validated information coefficient and Lasso Path

As before, we can plot the average information coefficient for all test sets used during cross-validation. We see again that regularization improves the IC over the unconstrained model, delivering the best out-of-sample result at a level of $\lambda=10^{-5}$. The optimal regularization value is quite different from ridge regression because the penalty consists of the sum of the absolute, not the squared values of the relatively small coefficient values. We can also see that for this regularization level, the coefficients have been similarly shrunk, as in the ridge regression case:



Cross-validated information coefficient and Lasso Path

In sum, ridge and lasso will produce similar results. Ridge often computes faster, but lasso also yields continuous features subset selection by gradually reducing coefficients to zero, hence eliminating features.

Linear classification

The linear regression model discussed so far assumes a quantitative response variable. In this section, we will focus on approaches to modeling qualitative output variables for inference and prediction, a process that is known as **classification** and that occurs even more frequently than regression in practice.

Predicting a qualitative response for a data point is called **classifying** that observation because it involves assigning the observation to a category, or class. In practice, classification methods often predict probabilities for each of the categories of a qualitative variable and then use this probability to decide on the proper classification.

We could approach the classification problem ignoring the fact that the output variable assumes discrete values, and apply the linear regression model to try to predict a categorical output using multiple input variables. However, it is easy to construct examples where this method performs very poorly. Furthermore, it doesn't make intuitive sense for the model to produce values larger than 1 or smaller than 0 when we know that $y \in [0, 1]$.

There are many different classification techniques, or classifiers, that are available to predict a qualitative response. In this section, we will introduce the widely used logistic regression which is closely related to linear regression. We will address more complex methods in the following chapters, on generalized additive models that include decision trees and random forests, as well as gradient boosting machines and neural networks.

The logistic regression model

The logistic regression model arises from the desire to model the probabilities of the output classes given a function that is linear in x, just like the linear regression model, while at the same time ensuring that they sum to one and remain in the [0, 1] as we would expect from probabilities.

In this section, we introduce the objective and functional form of the logistic regression model and describe the training method. We then illustrate how to use logistic regression for statistical inference with macro data using statsmodels, and how to predict price movements using the regularized logistic regression implemented by sklearn.

Objective function

For illustration, we'll use the output variable y that takes on the value 1 if a stock return is positive over a given time horizon d, and 0 otherwise:

$$y_t = \begin{cases} 1 & r_{t+d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

We could easily extend y to three categories, where 0 and 2 reflect negative and positive price moves beyond a certain threshold, and 1 otherwise. Rather than modeling the output variable *y*, however, logistic regression models the probability that y belongs to either of the categories given a vector of alpha factors or features $\boldsymbol{x}_t$. In other words, the logistic regression models the probability that the stock price goes up, conditional on the values of the variables included in the model:

$$p(\boldsymbol{x}_t) = Pr(y_t = 1|\boldsymbol{x}_t)$$

The logistic function

To prevent the model from producing values outside the [0, 1] interval, we must model *p(x)* using a function that only gives outputs between 0 and 1 over the entire domain of *x*. The logistic function meets this requirement and always produces an S-shaped curve (see notebook examples), and so, regardless of the value of X, we will obtain a sensible prediction:

$$p(\boldsymbol{x}) = \frac{e^{\beta_0 + \sum_{i=1}^{p} \beta_i x_i}}{1 + e^{\beta_0 + \sum_{i=1}^{p} \beta_i x_i}} = \frac{e^{\boldsymbol{x}\boldsymbol{\beta}}}{1 + e^{\boldsymbol{x}\boldsymbol{\beta}}}$$

Here, the vector *x* includes a 1 for the intercept captured by the first component of $\boldsymbol{\beta}$, β_0. We can transform this expression to isolate the part that looks like a linear regression to arrive at:

$$\underbrace{\frac{p(\boldsymbol{x})}{1 - p(\boldsymbol{x})}}_{\text{odds}} = e^{\beta_0 + \sum_{i=1}^{p} \beta_i} \iff \underbrace{\log\left(\frac{p(\boldsymbol{x})}{1 - p(\boldsymbol{x})}\right)}_{\text{logit}} = \beta_0 + \sum_{i=1}^{p} \beta_i$$

The quantity *p(x)/[1−p(x)]* is called the **odds**, an alternative way to express probabilities that may be familiar from gambling, and can take on any value odds between 0 and ∞, where low values also imply low probabilities and high values imply high probabilities.

The logit is also called log-odds (since it is the logarithm of the odds). Hence, the logistic regression represents a logit that is linear in *x* and looks a lot like the preceding linear regression.

Maximum likelihood estimation

The coefficient vector $\boldsymbol{\beta}$ must be estimated using the available training data. Although we could use (non-linear) least squares to fit the logistic regression model, the more general method of maximum likelihood is preferred, since it has better statistical properties. As we have just discussed, the basic intuition behind using maximum likelihood to fit a logistic regression model is to seek estimates for $\boldsymbol{\beta}$ such that the predicted probability $\hat{p}(\boldsymbol{x})$ corresponds as closely as possible to the actual outcome. In other words, we try to find $\hat{\boldsymbol{\beta}}$ such that these estimates yield a number close to 1 for all cases where the stock price went up, and a number close to 0 otherwise. More formally, we are seeking to maximize the likelihood function:

$$\max_{\boldsymbol{\beta}} \mathcal{L}(\boldsymbol{\beta}) = \prod_{i:y_i=1} p(\boldsymbol{x}_i) \prod_{i':y_{i'}=0} (1 - p(\boldsymbol{x}_{i'}))$$

It is easier to work with sums than with products, so let's take logs on both sides to get the log-likelihood function and the corresponding definition of the logistic regression coefficients:

$$\boldsymbol{\beta}^{\mathrm{ML}} = \operatorname{argmax} \log \mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{N} (y_i \log p(\boldsymbol{x}_i, \boldsymbol{\beta}) + (1 - y_i \log(1 - p(\boldsymbol{x}_i, \boldsymbol{\beta}))))$$

Maximizing this equation by setting the derivatives of $\mathcal{L}$ with respect to $\boldsymbol{\beta}$ to zero yields p+1 so-called score equations that are nonlinear in the parameters that can be solved using iterative numerical methods for the concave log-likelihood function.

How to conduct inference with statsmodels

We will illustrate how to use logistic regression with `statsmodels` based on a simple built-in dataset containing quarterly US macro data from 1959 – 2009 (see the notebook `logistic_regression_macro_data.ipynb` for detail).

The variables and their transformations are listed in the following table:

Variable	Description	Transformation
`realgdp`	Real gross domestic product	Annual Growth Rate
`realcons`	Real personal consumption expenditures	Annual Growth Rate
`realinv`	Real gross private domestic investment	Annual Growth Rate
`realgovt`	Real federal expenditures and gross investment	Annual Growth Rate
`realdpi`	Real private disposable income	Annual Growth Rate
`m1`	M1 nominal money stock	Annual Growth Rate
`tbilrate`	Monthly 3 treasury bill rate	Level
`unemp`	Seasonally adjusted unemployment rate (%)	Level
`infl`	Inflation rate	Level
`realint`	Real interest rate	Level

To obtain a binary target variable, we compute the 20-quarter rolling average of the annual growth rate of quarterly real GDP. We then assign 1 if current growth exceeds the moving average and 0 otherwise. Finally, we shift the indicator variables to align next quarter's outcome with the current quarter.

We use an intercept and convert the quarter values to dummy variables and train the logistic regression model as follows:

```
import statsmodels.api as sm

data = pd.get_dummies(data.drop(drop_cols, axis=1), columns=['quarter'],
drop_first=True).dropna()
model = sm.Logit(data.target, sm.add_constant(data.drop('target', axis=1)))
result = model.fit()
result.summary()
```

This produces the following summary for our model with 198 observations and 13 variables, including intercept:

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                 target   No. Observations:                  198
Model:                          Logit   Df Residuals:                      185
Method:                           MLE   Df Model:                           12
Date:                Mon, 10 Sep 2018   Pseudo R-squ.:                 0.5022
Time:                        20:27:53   Log-Likelihood:                -67.907
converged:                       True   LL-Null:                       -136.42
                                        LLR p-value:                 2.375e-23
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -8.5881      1.908     -4.502      0.000     -12.327      -4.849
realcons     130.1446     26.633      4.887      0.000      77.945     182.344
realinv       18.8414      4.053      4.648      0.000      10.897      26.786
realgovt     -19.0318      6.010     -3.166      0.002     -30.812      -7.252
realdpi      -52.2473     19.912     -2.624      0.009     -91.275     -13.220
m1            -1.3462      6.177     -0.218      0.827     -13.453      10.761
tbilrate      60.8607     44.350      1.372      0.170     -26.063     147.784
unemp          0.9487      0.249      3.818      0.000       0.462       1.436
infl         -60.9647     44.362     -1.374      0.169    -147.913      25.984
realint      -61.0453     44.359     -1.376      0.169    -147.987      25.896
quarter_2      0.1128      0.618      0.182      0.855      -1.099       1.325
quarter_3     -0.1991      0.609     -0.327      0.744      -1.393       0.995
quarter_4      0.0007      0.608      0.001      0.999      -1.191       1.192
==============================================================================
```

Logit Regression results

The summary indicates that the model has been trained using maximum likelihood and provides the maximized value of the log-likelihood function at -67.9.

The LL-Null value of -136.42 is the result of the maximized log-likelihood function when only an intercept is included. It forms the basis for the pseudo-R^2 statistic and the Log-**Likelihood Ratio** (**LLR**) test.

The pseudo-R^2 statistic is a substitute for the familiar R^2 available under least squares. It is computed based on the ratio of the maximized log-likelihood function for the null model m_0 and the full model m_1 as follows:

$$\rho^2 = 1 - \frac{\log \mathcal{L}(m_1^*)}{\log \mathcal{L}(m_0^*)}$$

The values vary from 0 (when the model does not improve the likelihood) to 1 where the model fits perfectly and the log-likelihood is maximized at 0. Consequently, higher values indicate a better fit.

The LLR test generally compares a more restricted model and is computed as:

$$\text{LLR} = -2\log(\mathcal{L}(m_0^*)/\mathcal{L}(m_1^*)) = 2(\log\mathcal{L}(m_1^*) - \log\mathcal{L}(m_0^*))$$

The null hypothesis is that the restricted model performs better but the low p-value suggests that we can reject this hypothesis and prefer the full model over the null model. This is similar to the F-test for linear regression (where we can also use the LLR test when we estimate the model using MLE).

The z-statistic plays the same role as the t-statistic in the linear regression output and is equally computed as the ratio of the coefficient estimate and its standard error. The p-values also indicate the probability of observing the test statistic assuming the null hypothesis $H_0 : \beta = 0$ that the population coefficient is zero. We can reject this hypothesis for the `intercept`, `realcons`, `realinv`, `realgovt`, `realdpi`, and `unemp`.

How to use logistic regression for prediction

The lasso L_1 penalty and the ridge L_2 penalty can both be used with logistic regression. They have the same shrinkage effect as we have just discussed, and the lasso can again be used for variable selection with any linear regression model.

Just as with linear regression, it is important to standardize the input variables as the regularized models are scale sensitive. The regularization hyperparameter also requires tuning using cross-validation as in the linear regression case.

How to predict price movements using sklearn

We continue the price prediction example but now we binarize the outcome variable so that it takes on the value 1 whenever the 10-day return is positive and 0 otherwise; see the notebook `logistic_regression.ipynb` in the sub directory `stock_price_prediction`:

```
target = 'Returns10D'
label = (y[target] > 0).astype(int).to_frame(target)
```

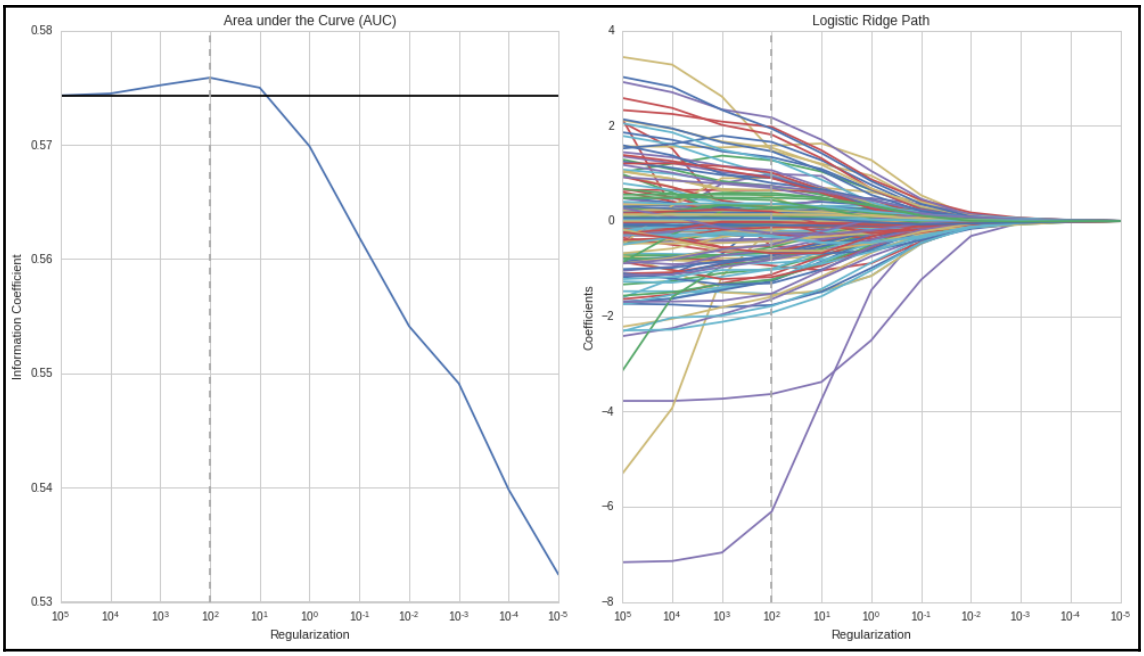
With this new categorical outcome variable, we can now train a logistic regression using the default L_2 regularization. For logistic regression, the regularization is formulated inversely to linear regression: higher values for λ imply less regularization and vice versa. We evaluate 11 parameter values using cross validation as follows:

```
nfolds = 250
Cs = np.logspace(-5, 5, 11)
scaler = StandardScaler()

logistic_results, logistic_coeffs = pd.DataFrame(), pd.DataFrame()
for C in Cs:
    coeffs = []
    log_reg = LogisticRegression(C=C)
    for i, (train_dates, test_dates) in enumerate(time_series_split(dates,
nfolds=nfolds)):
        X_train = model_data.loc[idx[train_dates], features]
        y_train = model_data.loc[idx[train_dates], target]
        log_reg.fit(X=scaler.fit_transform(X_train), y=y_train)

        X_test = model_data.loc[idx[test_dates], features]
        y_test = model_data.loc[idx[test_dates], target]
        y_pred = log_reg.predict_proba(scaler.transform(X_test))[:, 1]

        coeffs.append(log_reg.coef_.squeeze())
        logistic_results = (logistic_results
                            .append(y_test
                                    .to_frame('actuals')
                                    .assign(predicted=y_pred, C=C)))
    logistic_coeffs[C] = np.mean(coeffs, axis=0)
```

We then use the `roc_auc_score` discussed in the previous chapter to compare the predictive accuracy across the various regularization parameters:

```
auc_by_C = logistic_results.groupby('C').apply(lambda x:
roc_auc_score(y_true=x.actuals.astype(int),
y_score=x.predicted))
```

We can again plot the AUC result for the range of hyperparameter values alongside the coefficient path that shows the improvements in predictive accuracy as the coefficients are a bit shrunk at the optimal regularization value 10^2:



AUC and Logistic Ridge path

Summary

In this chapter, we introduced the first machine learning models using the important baseline case of linear models for regression and classification. We explored the formulation of the objective functions for both tasks, learned about various training methods, and learned how to use the model for both inference and prediction.

We applied these new machine learning techniques to estimate linear factor models that are very useful to manage risks, assess new alpha factors, and attribute performance. We also applied linear regression and classification to accomplish the first predictive task of predicting stock returns in absolute and directional terms.

In the next chapter, we will look at the important topic of linear time series models that are designed to capture serial correlation patterns in the univariate and multivariate case. We will also learn about new trading strategies as we explore pairs trading based on the concept of cointegration that captures dynamic correlation among two stock price series.