04_logistic_regression

September 29, 2021

1 How to use logistic regression for prediction

The lasso L1 penalty and the ridge L2 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.

1.1 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

1.2 Imports

[66]:
```
import pandas as pd
import numpy as np
from time import time
import talib
import re
from statsmodels.api import OLS
from sklearn.metrics import mean_squared_error
from scipy.stats import spearmanr
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, Lasso,␣
↪LassoCV, LogisticRegression
from sklearn.preprocessing import StandardScaler

from quantopian.research import run_pipeline
from quantopian.pipeline import Pipeline, factors, filters, classifiers
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import (Latest,
                                         Returns,
                                         AverageDollarVolume,
                                         SimpleMovingAverage,
                                         EWMA,
                                         BollingerBands,
```

```
                                          CustomFactor,
                                          MarketCap,
                                         SimpleBeta)
from quantopian.pipeline.filters import QTradableStocksUS, StaticAssets
from quantopian.pipeline.data.quandl import fred_usdontd156n as libor
from empyrical import max_drawdown, sortino_ratio

import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score
```

1.3 Data Sources

[2]:
```
################
# Fundamentals #
################

# Morningstar fundamentals (2002 - Ongoing)
# https://www.quantopian.com/help/fundamentals
from quantopian.pipeline.data import Fundamentals

######################
# Analyst Estimates #
######################

# Earnings Surprises - Zacks (27 May 2006 - Ongoing)
# https://www.quantopian.com/data/zacks/earnings_surprises
from quantopian.pipeline.data.zacks import EarningsSurprises
from quantopian.pipeline.factors.zacks import␣
 ↪BusinessDaysSinceEarningsSurprisesAnnouncement

##########
# Events #
##########

# Buyback Announcements - EventVestor (01 Jun 2007 - Ongoing)
# https://www.quantopian.com/data/eventvestor/buyback_auth
from quantopian.pipeline.data.eventvestor import BuybackAuthorizations
from quantopian.pipeline.factors.eventvestor import BusinessDaysSinceBuybackAuth

# CEO Changes - EventVestor (01 Jan 2007 - Ongoing)
# https://www.quantopian.com/data/eventvestor/ceo_change
from quantopian.pipeline.data.eventvestor import CEOChangeAnnouncements

# Dividends - EventVestor (01 Jan 2007 - Ongoing)
# https://www.quantopian.com/data/eventvestor/dividends
from quantopian.pipeline.data.eventvestor import (
```

```
    DividendsByExDate,
    DividendsByPayDate,
    DividendsByAnnouncementDate,
)
from quantopian.pipeline.factors.eventvestor import (
    BusinessDaysSincePreviousExDate,
    BusinessDaysUntilNextExDate,
    BusinessDaysSinceDividendAnnouncement,
)

# Earnings Calendar - EventVestor (01 Jan 2007 - Ongoing)
# https://www.quantopian.com/data/eventvestor/earnings_calendar
from quantopian.pipeline.data.eventvestor import EarningsCalendar
from quantopian.pipeline.factors.eventvestor import (
    BusinessDaysUntilNextEarnings,
    BusinessDaysSincePreviousEarnings
)

# 13D Filings - EventVestor (01 Jan 2007 - Ongoing)
# https://www.quantopian.com/data/eventvestor/_13d_filings
from quantopian.pipeline.data.eventvestor import _13DFilings
from quantopian.pipeline.factors.eventvestor import␣
 ↪BusinessDaysSince13DFilingsDate

##############
# Sentiment #
##############

# News Sentiment - Sentdex Sentiment Analysis (15 Oct 2012 - Ongoing)
# https://www.quantopian.com/data/sentdex/sentiment
from quantopian.pipeline.data.sentdex import sentiment
```

1.4 Setup

1.4.1 Time Horizon

[3]:
```
# trading days per period
MONTH = 21
YEAR = 12 * MONTH
```

[4]:
```
START = '2014-01-01'
END = '2015-12-31'
```

1.4.2 Universe

[5]:
```
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'),
    )
```

[6]:
```
# UNIVERSE = StaticAssets(symbols(['MSFT', 'AAPL']))
UNIVERSE = Q100US()
```

[7]:
```
class AnnualizedData(CustomFactor):
    # Get the sum of the last 4 reported values
    window_length = 260

    def compute(self, today, assets, out, asof_date, values):
        for asset in range(len(assets)):
            # unique asof dates indicate availability of new figures
            _, filing_dates = np.unique(asof_date[:, asset], return_index=True)
            quarterly_values = values[filing_dates[-4:], asset]
            # ignore annual windows with <4 quarterly data points
            if len(~np.isnan(quarterly_values)) != 4:
                out[asset] = np.nan
            else:
                out[asset] = np.sum(quarterly_values)
```

[8]:
```
class AnnualAvg(CustomFactor):
    window_length = 252

    def compute(self, today, assets, out, values):
        out[:] = (values[0] + values[-1])/2
```

[9]:
```
def factor_pipeline(factors):
    start = time()
    pipe = Pipeline({k: v(mask=UNIVERSE).rank() for k, v in factors.items()},
                    screen=UNIVERSE)
    result = run_pipeline(pipe, start_date=START, end_date=END)
    return result, time() - start
```

1.5 Factor Library

1.5.1 Value Factors

[10]:
```
class ValueFactors:
    """Definitions of factors for cross-sectional trading algorithms"""

    @staticmethod
    def PriceToSalesTTM(**kwargs):
        """Last closing price divided by sales per share"""
        return Fundamentals.ps_ratio.latest

    @staticmethod
    def PriceToEarningsTTM(**kwargs):
        """Closing price divided by earnings per share (EPS)"""
        return Fundamentals.pe_ratio.latest

    @staticmethod
    def PriceToDilutedEarningsTTM(mask):
        """Closing price divided by diluted EPS"""
        last_close = USEquityPricing.close.latest
        diluted_eps = AnnualizedData(inputs = [Fundamentals.
↪diluted_eps_earnings_reports_asof_date,
                                               Fundamentals.
↪diluted_eps_earnings_reports],
                                     mask=mask)
        return last_close / diluted_eps

    @staticmethod
    def PriceToForwardEarnings(**kwargs):
        """Price to Forward Earnings"""
        return Fundamentals.forward_pe_ratio.latest

    @staticmethod
    def DividendYield(**kwargs):
        """Dividends per share divided by closing price"""
        return Fundamentals.trailing_dividend_yield.latest

    @staticmethod
    def PriceToFCF(mask):
        """Price to Free Cash Flow"""
        last_close = USEquityPricing.close.latest
        fcf_share = AnnualizedData(inputs = [Fundamentals.
↪fcf_per_share_asof_date,
                                             Fundamentals.fcf_per_share],
                                   mask=mask)
        return last_close / fcf_share
```

```
    @staticmethod
    def PriceToOperatingCashflow(mask):
        """Last Close divided by Operating Cash Flows"""
        last_close = USEquityPricing.close.latest
        cfo_per_share = AnnualizedData(inputs = [Fundamentals.
↪cfo_per_share_asof_date,
                                                 Fundamentals.cfo_per_share],
                                       mask=mask)
        return last_close / cfo_per_share

    @staticmethod
    def PriceToBook(mask):
        """Closing price divided by book value"""
        last_close = USEquityPricing.close.latest
        book_value_per_share = AnnualizedData(inputs = [Fundamentals.
↪book_value_per_share_asof_date,
                                                        Fundamentals.
↪book_value_per_share],
                                              mask=mask)
        return last_close / book_value_per_share


    @staticmethod
    def EVToFCF(mask):
        """Enterprise Value divided by Free Cash Flows"""
        fcf = AnnualizedData(inputs = [Fundamentals.free_cash_flow_asof_date,
                                       Fundamentals.free_cash_flow],
                             mask=mask)
        return Fundamentals.enterprise_value.latest / fcf

    @staticmethod
    def EVToEBITDA(mask):
        """Enterprise Value to Earnings Before Interest, Taxes, Deprecation and␣
↪Amortization (EBITDA)"""
        ebitda = AnnualizedData(inputs = [Fundamentals.ebitda_asof_date,
                                          Fundamentals.ebitda],
                                mask=mask)

        return Fundamentals.enterprise_value.latest / ebitda

    @staticmethod
    def EBITDAYield(mask):
        """EBITDA divided by latest close"""
        ebitda = AnnualizedData(inputs = [Fundamentals.ebitda_asof_date,
                                          Fundamentals.ebitda],
                                mask=mask)
        return USEquityPricing.close.latest / ebitda
```

[11]:
```
VALUE_FACTORS = {
    'DividendYield'            : ValueFactors.DividendYield,
    'EBITDAYield'              : ValueFactors.EBITDAYield,
    'EVToEBITDA'               : ValueFactors.EVToEBITDA,
    'EVToFCF'                  : ValueFactors.EVToFCF,
    'PriceToBook'              : ValueFactors.PriceToBook,
    'PriceToDilutedEarningsTTM': ValueFactors.PriceToDilutedEarningsTTM,
    'PriceToEarningsTTM'       : ValueFactors.PriceToEarningsTTM,
    'PriceToFCF'               : ValueFactors.PriceToFCF,
    'PriceToForwardEarnings'   : ValueFactors.PriceToForwardEarnings,
    'PriceToOperatingCashflow' : ValueFactors.PriceToOperatingCashflow,
    'PriceToSalesTTM'          : ValueFactors.PriceToSalesTTM,
}
```

[12]:
```
value_result, t = factor_pipeline(VALUE_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
value_result.info()
```

```
/usr/local/lib/python2.7/dist-packages/numpy/lib/arraysetops.py:200:
FutureWarning: In the future, NAT != NAT will be True rather than False.
  flag = np.concatenate(([True], aux[1:] != aux[:-1]))

Pipeline run time 96.25 secs
<class 'pandas.core.frame.DataFrame'>
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 11 columns):
DividendYield                40772 non-null float64
EBITDAYield                  49823 non-null float64
EVToEBITDA                   49823 non-null float64
EVToFCF                      46400 non-null float64
PriceToBook                  50343 non-null float64
PriceToDilutedEarningsTTM    50215 non-null float64
PriceToEarningsTTM           48956 non-null float64
PriceToFCF                   49133 non-null float64
PriceToForwardEarnings       39607 non-null float64
PriceToOperatingCashflow     50343 non-null float64
PriceToSalesTTM              50362 non-null float64
dtypes: float64(11)
memory usage: 4.6+ MB
```

1.5.2 Momentum

[13]:
```
class MomentumFactors:
    """Custom Momentum Factors"""
    class PercentAboveLow(CustomFactor):
        """Percentage of current close above low
        in lookback window of window_length days
```

```
        """
        inputs = [USEquityPricing.close]
        window_length = 252

        def compute(self, today, assets, out, close):
            out[:] = close[-1] / np.min(close, axis=0) - 1

    class PercentBelowHigh(CustomFactor):
        """Percentage of current close below high
        in lookback window of window_length days
        """

        inputs = [USEquityPricing.close]
        window_length = 252

        def compute(self, today, assets, out, close):
            out[:] = close[-1] / np.max(close, axis=0) - 1

    @staticmethod
    def make_dx(timeperiod=14):
        class DX(CustomFactor):
            """Directional Movement Index"""
            inputs = [USEquityPricing.high,
                      USEquityPricing.low,
                      USEquityPricing.close]
            window_length = timeperiod + 1

            def compute(self, today, assets, out, high, low, close):
                out[:] = [talib.DX(high[:, i],
                                   low[:, i],
                                   close[:, i],
                                   timeperiod=timeperiod)[-1]
                          for i in range(len(assets))]
        return DX

    @staticmethod
    def make_mfi(timeperiod=14):
        class MFI(CustomFactor):
            """Money Flow Index"""
            inputs = [USEquityPricing.high,
                      USEquityPricing.low,
                      USEquityPricing.close,
                      USEquityPricing.volume]
            window_length = timeperiod + 1

            def compute(self, today, assets, out, high, low, close, vol):
                out[:] = [talib.MFI(high[:, i],
```

```
                                    low[:, i],
                                    close[:, i],
                                    vol[:, i],
                                    timeperiod=timeperiod)[-1]
                            for i in range(len(assets))]
        return MFI

    @staticmethod
    def make_oscillator(fastperiod=12, slowperiod=26, matype=0):
        class PPO(CustomFactor):
            """12/26-Day Percent Price Oscillator"""
            inputs = [USEquityPricing.close]
            window_length = slowperiod

            def compute(self, today, assets, out, close_prices):
                out[:] = [talib.PPO(close,
                                    fastperiod=fastperiod,
                                    slowperiod=slowperiod,
                                    matype=matype)[-1]
                          for close in close_prices.T]
        return PPO

    @staticmethod
    def make_stochastic_oscillator(fastk_period=5, slowk_period=3,␣
↪slowd_period=3,
                                   slowk_matype=0, slowd_matype=0):
        class StochasticOscillator(CustomFactor):
            """20-day Stochastic Oscillator """
            inputs = [USEquityPricing.high,
                      USEquityPricing.low,
                      USEquityPricing.close]
            outputs = ['slowk', 'slowd']
            window_length = fastk_period * 2

            def compute(self, today, assets, out, high, low, close):
                slowk, slowd = [talib.STOCH(high[:, i],
                                            low[:, i],
                                            close[:, i],
                                            fastk_period=fastk_period,
                                            slowk_period=slowk_period,
                                            slowk_matype=slowk_matype,
                                            slowd_period=slowd_period,
                                            slowd_matype=slowd_matype)[-1]
                                for i in range(len(assets))]

                out.slowk[:], out.slowd[:] = slowk[-1], slowd[-1]
        return StochasticOscillator
```

```
    @staticmethod
    def make_trendline(timeperiod=252):
        class Trendline(CustomFactor):
            inputs = [USEquityPricing.close]
            """52-Week Trendline"""
            window_length = timeperiod

            def compute(self, today, assets, out, close_prices):
                out[:] = [talib.LINEARREG_SLOPE(close,
                                                timeperiod=timeperiod)[-1]
                          for close in close_prices.T]
        return Trendline
```

[14]:
```
MOMENTUM_FACTORS = {
    'Percent Above Low'            : MomentumFactors.PercentAboveLow,
    'Percent Below High'           : MomentumFactors.PercentBelowHigh,
    'Price Oscillator'             : MomentumFactors.make_oscillator(),
    'Money Flow Index'             : MomentumFactors.make_mfi(),
    'Directional Movement Index'   : MomentumFactors.make_dx(),
    'Trendline'                    : MomentumFactors.make_trendline()
}
```

[15]:
```
momentum_result, t = factor_pipeline(MOMENTUM_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
momentum_result.info()
```

```
Pipeline run time 21.78 secs
<class 'pandas.core.frame.DataFrame'>
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 6 columns):
Directional Movement Index    50362 non-null float64
Money Flow Index              50362 non-null float64
Percent Above Low             49536 non-null float64
Percent Below High            49536 non-null float64
Price Oscillator              50355 non-null float64
Trendline                     49536 non-null float64
dtypes: float64(6)
memory usage: 2.7+ MB
```

1.5.3 Efficiency

[16]:
```
class EfficiencyFactors:

    @staticmethod
    def CapexToAssets(mask):
```

```
        """Capital Expenditure divided by Total Assets"""
        capex = AnnualizedData(inputs = [Fundamentals.
↪capital_expenditure_asof_date,
                                         Fundamentals.capital_expenditure],
                               mask=mask)
        assets = Fundamentals.total_assets.latest
        return - capex / assets

    @staticmethod
    def CapexToSales(mask):
        """Capital Expenditure divided by Total Revenue"""
        capex = AnnualizedData(inputs = [Fundamentals.
↪capital_expenditure_asof_date,
                                         Fundamentals.capital_expenditure],
                               mask=mask)
        revenue = AnnualizedData(inputs = [Fundamentals.total_revenue_asof_date,
                                         Fundamentals.total_revenue],
                               mask=mask)
        return - capex / revenue

    @staticmethod
    def CapexToFCF(mask):
        """Capital Expenditure divided by Free Cash Flows"""
        capex = AnnualizedData(inputs = [Fundamentals.
↪capital_expenditure_asof_date,
                                         Fundamentals.capital_expenditure],
                               mask=mask)
        free_cash_flow = AnnualizedData(inputs = [Fundamentals.
↪free_cash_flow_asof_date,
                                         Fundamentals.free_cash_flow],
                               mask=mask)
        return - capex / free_cash_flow

    @staticmethod
    def EBITToAssets(mask):
        """Earnings Before Interest and Taxes (EBIT) divided by Total Assets"""
        ebit = AnnualizedData(inputs = [Fundamentals.ebit_asof_date,
                                         Fundamentals.ebit],
                               mask=mask)
        assets = Fundamentals.total_assets.latest
        return ebit / assets

    @staticmethod
    def CFOToAssets(mask):
        """Operating Cash Flows divided by Total Assets"""
        cfo = AnnualizedData(inputs = [Fundamentals.
↪operating_cash_flow_asof_date,
```

```
                                        Fundamentals.operating_cash_flow],
                                    mask=mask)
        assets = Fundamentals.total_assets.latest
        return cfo / assets

    @staticmethod
    def RetainedEarningsToAssets(mask):
        """Retained Earnings divided by Total Assets"""
        retained_earnings = AnnualizedData(inputs = [Fundamentals.
↪retained_earnings_asof_date,
                                        Fundamentals.retained_earnings],
                                    mask=mask)
        assets = Fundamentals.total_assets.latest
        return retained_earnings / assets
```

[17]:
```
EFFICIENCY_FACTORS = {
    'CFO To Assets' :EfficiencyFactors.CFOToAssets,
    'Capex To Assets' :EfficiencyFactors.CapexToAssets,
    'Capex To FCF' :EfficiencyFactors.CapexToFCF,
    'Capex To Sales' :EfficiencyFactors.CapexToSales,
    'EBIT To Assets' :EfficiencyFactors.EBITToAssets,
    'Retained Earnings To Assets' :EfficiencyFactors.RetainedEarningsToAssets
    }
```

[18]:
```
efficiency_result, t = factor_pipeline(EFFICIENCY_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
efficiency_result.info()
```

```
Pipeline run time 37.93 secs
<class 'pandas.core.frame.DataFrame'>
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 6 columns):
CFO To Assets                  50351 non-null float64
Capex To Assets                46997 non-null float64
Capex To FCF                   45799 non-null float64
Capex To Sales                 46997 non-null float64
EBIT To Assets                 46635 non-null float64
Retained Earnings To Assets    50349 non-null float64
dtypes: float64(6)
memory usage: 2.7+ MB
```

1.5.4 Risk

[19]:
```
class RiskFactors:

    @staticmethod
    def LogMarketCap(mask):
        """Log of Market Capitalization log(Close Price * Shares Outstanding)"""
        return np.log(MarketCap(mask=mask))

    class DownsideRisk(CustomFactor):
        """Mean returns divided by std of 1yr daily losses (Sortino Ratio)"""
        inputs = [USEquityPricing.close]
        window_length = 252

        def compute(self, today, assets, out, close):
            ret = pd.DataFrame(close).pct_change()
            out[:] = ret.mean().div(ret.where(ret<0).std())

    @staticmethod
    def MarketBeta(**kwargs):
        """Slope of 1-yr regression of price returns against index returns"""
        return SimpleBeta(target=symbols('SPY'), regression_length=252)

    class DownsideBeta(CustomFactor):
        """Slope of 1yr regression of returns on negative index returns"""
        inputs = [USEquityPricing.close]
        window_length = 252

        def compute(self, today, assets, out, close):
            t = len(close)
            assets = pd.DataFrame(close).pct_change()

            start_date = (today - pd.DateOffset(years=1)).strftime('%Y-%m-%d')
            spy = get_pricing('SPY',
                              start_date=start_date,
                              end_date=today.strftime('%Y-%m-%d')).
↪reset_index(drop=True)
            spy_neg_ret = (spy
                           .close_price
                           .iloc[-t:]
                           .pct_change()
                           .pipe(lambda x: x.where(x<0)))

            out[:] = assets.apply(lambda x: x.cov(spy_neg_ret)).div(spy_neg_ret.
↪var())

    class Vol3M(CustomFactor):
```

```
        """3-month Volatility: Standard deviation of returns over 3 months"""

        inputs = [USEquityPricing.close]
        window_length = 63

        def compute(self, today, assets, out, close):
            out[:] = np.log1p(pd.DataFrame(close).pct_change()).std()
```

[20]:
```
RISK_FACTORS = {
    'Log Market Cap' : RiskFactors.LogMarketCap,
    'Downside Risk'  : RiskFactors.DownsideRisk,
    'Index Beta'     : RiskFactors.MarketBeta,
#     'Downside Beta'  : RiskFactors.DownsideBeta,
    'Volatility 3M'  : RiskFactors.Vol3M,
}
```

[21]:
```
risk_result, t = factor_pipeline(RISK_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
risk_result.info()
```

```
Pipeline run time 48.59 secs
<class 'pandas.core.frame.DataFrame'>
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):
Downside Risk     50362 non-null float64
Index Beta        50079 non-null float64
Log Market Cap    50362 non-null float64
Volatility 3M     50362 non-null float64
dtypes: float64(4)
memory usage: 1.9+ MB
```

1.5.5 Growth

[22]:
```
def growth_pipeline():
    revenue = AnnualizedData(inputs = [Fundamentals.total_revenue_asof_date,
                                       Fundamentals.total_revenue],
                             mask=UNIVERSE)
    eps = AnnualizedData(inputs = [Fundamentals.
 ↪diluted_eps_earnings_reports_asof_date,
                                       Fundamentals.
 ↪diluted_eps_earnings_reports],
                             mask=UNIVERSE)

    return Pipeline({'Sales': revenue,
                     'EPS': eps,
                     'Total Assets': Fundamentals.total_assets.latest,
```

```
                     'Net Debt': Fundamentals.net_debt.latest},
                    screen=UNIVERSE)
```

[23]:
```
start_timer = time()
growth_result = run_pipeline(growth_pipeline(), start_date=START, end_date=END)

for col in growth_result.columns:
    for month in [3, 12]:
        new_col = col + ' Growth {}M'.format(month)
        kwargs = {new_col: growth_result[col].pct_change(month*MONTH).
 ↪groupby(level=1).rank()}
        growth_result = growth_result.assign(**kwargs)
print('Pipeline run time {:.2f} secs'.format(time() - start_timer))
growth_result.info()
```

```
Pipeline run time 23.48 secs
<class 'pandas.core.frame.DataFrame'>
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 12 columns):
EPS                         50215 non-null float64
Net Debt                    47413 non-null float64
Sales                       50351 non-null float64
Total Assets                50362 non-null float64
EPS Growth 3M               50152 non-null float64
EPS Growth 12M              49963 non-null float64
Net Debt Growth 3M          47350 non-null float64
Net Debt Growth 12M         47171 non-null float64
Sales Growth 3M             50288 non-null float64
Sales Growth 12M            50099 non-null float64
Total Assets Growth 3M      50299 non-null float64
Total Assets Growth 12M     50110 non-null float64
dtypes: float64(12)
memory usage: 5.0+ MB
```

1.5.6 Quality

[24]:
```
class QualityFactors:

    @staticmethod
    def AssetTurnover(mask):
        """Sales divided by average of year beginning and year end assets"""

        assets = AnnualAvg(inputs=[Fundamentals.total_assets],
                           mask=mask)
        sales = AnnualizedData([Fundamentals.total_revenue_asof_date,
                                Fundamentals.total_revenue], mask=mask)
```

```
        return sales / assets

    @staticmethod
    def CurrentRatio(mask):
        """Total current assets divided by total current liabilities"""

        assets = Fundamentals.current_assets.latest
        liabilities = Fundamentals.current_liabilities.latest
        return assets / liabilities

    @staticmethod
    def AssetToEquityRatio(mask):
        """Total current assets divided by common equity"""

        assets = Fundamentals.current_assets.latest
        equity = Fundamentals.common_stock.latest
        return assets / equity


    @staticmethod
    def InterestCoverage(mask):
        """EBIT divided by interest expense"""

        ebit = AnnualizedData(inputs = [Fundamentals.ebit_asof_date,
                                        Fundamentals.ebit], mask=mask)

        interest_expense = AnnualizedData(inputs = [Fundamentals.
↪interest_expense_asof_date,
                                        Fundamentals.interest_expense],␣
↪mask=mask)
        return ebit / interest_expense

    @staticmethod
    def DebtToAssetRatio(mask):
        """Total Debts divided by Total Assets"""

        debt = Fundamentals.total_debt.latest
        assets = Fundamentals.total_assets.latest
        return debt / assets

    @staticmethod
    def DebtToEquityRatio(mask):
        """Total Debts divided by Common Stock Equity"""

        debt = Fundamentals.total_debt.latest
        equity = Fundamentals.common_stock.latest
        return debt / equity
```

```
    @staticmethod
    def WorkingCapitalToAssets(mask):
        """Current Assets less Current liabilities (Working Capital) divided by␣
↪Assets"""

        working_capital = Fundamentals.working_capital.latest
        assets = Fundamentals.total_assets.latest
        return working_capital / assets

    @staticmethod
    def WorkingCapitalToSales(mask):
        """Current Assets less Current liabilities (Working Capital), divided␣
↪by Sales"""

        working_capital = Fundamentals.working_capital.latest
        sales = AnnualizedData([Fundamentals.total_revenue_asof_date,
                                Fundamentals.total_revenue], mask=mask)
        return working_capital / sales


    class MertonsDD(CustomFactor):
        """Merton's Distance to Default """

        inputs = [Fundamentals.total_assets,
                  Fundamentals.total_liabilities,
                  libor.value,
                  USEquityPricing.close]
        window_length = 252

        def compute(self, today, assets, out, tot_assets, tot_liabilities, r,␣
↪close):
            mertons = []

            for col_assets, col_liabilities, col_r, col_close in zip(tot_assets.
↪T, tot_liabilities.T,
                                                                     r.T, close.
↪T):
                vol_1y = np.nanstd(col_close)
                numerator = np.log(
                        col_assets[-1] / col_liabilities[-1]) + ((252 *␣
↪col_r[-1]) - ((vol_1y ** 2) / 2))
                mertons.append(numerator / vol_1y)

            out[:] = mertons
```

[25]:
```
QUALITY_FACTORS = {
    'AssetToEquityRatio'    : QualityFactors.AssetToEquityRatio,
    'AssetTurnover'         : QualityFactors.AssetTurnover,
    'CurrentRatio'          : QualityFactors.CurrentRatio,
    'DebtToAssetRatio'      : QualityFactors.DebtToAssetRatio,
    'DebtToEquityRatio'     : QualityFactors.DebtToEquityRatio,
    'InterestCoverage'      : QualityFactors.InterestCoverage,
    'MertonsDD'             : QualityFactors.MertonsDD,
    'WorkingCapitalToAssets': QualityFactors.WorkingCapitalToAssets,
    'WorkingCapitalToSales' : QualityFactors.WorkingCapitalToSales,
}
```

[26]:
```
quality_result, t = factor_pipeline(QUALITY_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
quality_result.info()
```

```
Pipeline run time 36.81 secs
<class 'pandas.core.frame.DataFrame'>
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 9 columns):
AssetToEquityRatio        45176 non-null float64
AssetTurnover             50314 non-null float64
CurrentRatio              45680 non-null float64
DebtToAssetRatio          50080 non-null float64
DebtToEquityRatio         48492 non-null float64
InterestCoverage          35250 non-null float64
MertonsDD                 50362 non-null float64
WorkingCapitalToAssets    45680 non-null float64
WorkingCapitalToSales     45669 non-null float64
dtypes: float64(9)
memory usage: 3.8+ MB
```

1.5.7 Payout

[27]:
```
class PayoutFactors:

    @staticmethod
    def DividendPayoutRatio(mask):
        """Dividends Per Share divided by Earnings Per Share"""

        dps = AnnualizedData(inputs = [Fundamentals.
 ↪dividend_per_share_earnings_reports_asof_date,
                                       Fundamentals.
 ↪dividend_per_share_earnings_reports], mask=mask)
```

```
        eps = AnnualizedData(inputs = [Fundamentals.
↪basic_eps_earnings_reports_asof_date,
                                       Fundamentals.
↪basic_eps_earnings_reports], mask=mask)
        return dps / eps

    @staticmethod
    def DividendGrowth(**kwargs):
        """Annualized percentage DPS change"""
        return Fundamentals.dps_growth.latest
```

[28]:
```
PAYOUT_FACTORS = {
    'Dividend Payout Ratio': PayoutFactors.DividendPayoutRatio,
    'Dividend Growth': PayoutFactors.DividendGrowth
}
```

[29]:
```
payout_result, t = factor_pipeline(PAYOUT_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
payout_result.info()
```

```
Pipeline run time 23.15 secs
<class 'pandas.core.frame.DataFrame'>
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 2 columns):
Dividend Growth          40517 non-null float64
Dividend Payout Ratio    39947 non-null float64
dtypes: float64(2)
memory usage: 1.2+ MB
```

1.5.8 Profitability

[30]:
```
class ProfitabilityFactors:

    @staticmethod
    def GrossProfitMargin(mask):
        """Gross Profit divided by Net Sales"""

        gross_profit = AnnualizedData([Fundamentals.gross_profit_asof_date,
                                       Fundamentals.gross_profit], mask=mask)
        sales = AnnualizedData([Fundamentals.total_revenue_asof_date,
                                Fundamentals.total_revenue], mask=mask)
        return gross_profit / sales

    @staticmethod
    def NetIncomeMargin(mask):
        """Net income divided by Net Sales"""
```

```
        net_income = AnnualizedData([Fundamentals.
↪net_income_income_statement_asof_date,
                                     Fundamentals.net_income_income_statement],␣
↪mask=mask)
        sales = AnnualizedData([Fundamentals.total_revenue_asof_date,
                                Fundamentals.total_revenue], mask=mask)
        return net_income / sales
```

[31]:
```
PROFITABIILTY_FACTORS = {
    'Gross Profit Margin': ProfitabilityFactors.GrossProfitMargin,
    'Net Income Margin': ProfitabilityFactors.NetIncomeMargin,
    'Return on Equity': Fundamentals.roe.latest,
    'Return on Assets': Fundamentals.roa.latest,
    'Return on Invested Capital': Fundamentals.roic.latest
}
```

[32]:
```
profitability_result, t = factor_pipeline(PAYOUT_FACTORS)
print('Pipeline run time {:.2f} secs'.format(t))
payout_result.info()
```

```
Pipeline run time 23.27 secs
<class 'pandas.core.frame.DataFrame'>
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 2 columns):
Dividend Growth          40517 non-null float64
Dividend Payout Ratio    39947 non-null float64
dtypes: float64(2)
memory usage: 1.2+ MB
```

[33]:
```
# profitability_pipeline().show_graph(format='png')
```

1.5.9 Build Dataset

Get Returns

[34]:
```
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()
```

```
<class 'pandas.core.frame.DataFrame'>
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
dtypes: float64(4)
memory usage: 1.9+ MB
```

[35]:
```
data = pd.concat([returns,
                  value_result,
                  momentum_result,
                  quality_result,
                  payout_result,
                  growth_result,
                  efficiency_result,
                  risk_result], axis=1).sortlevel()
data.index.names = ['date', 'asset']
```

[36]:
```
data['stock'] = data.index.get_level_values('asset').map(lambda x: x.asset_name)
```

Remove columns and rows with less than 80% of data availability

[37]:
```
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))
```

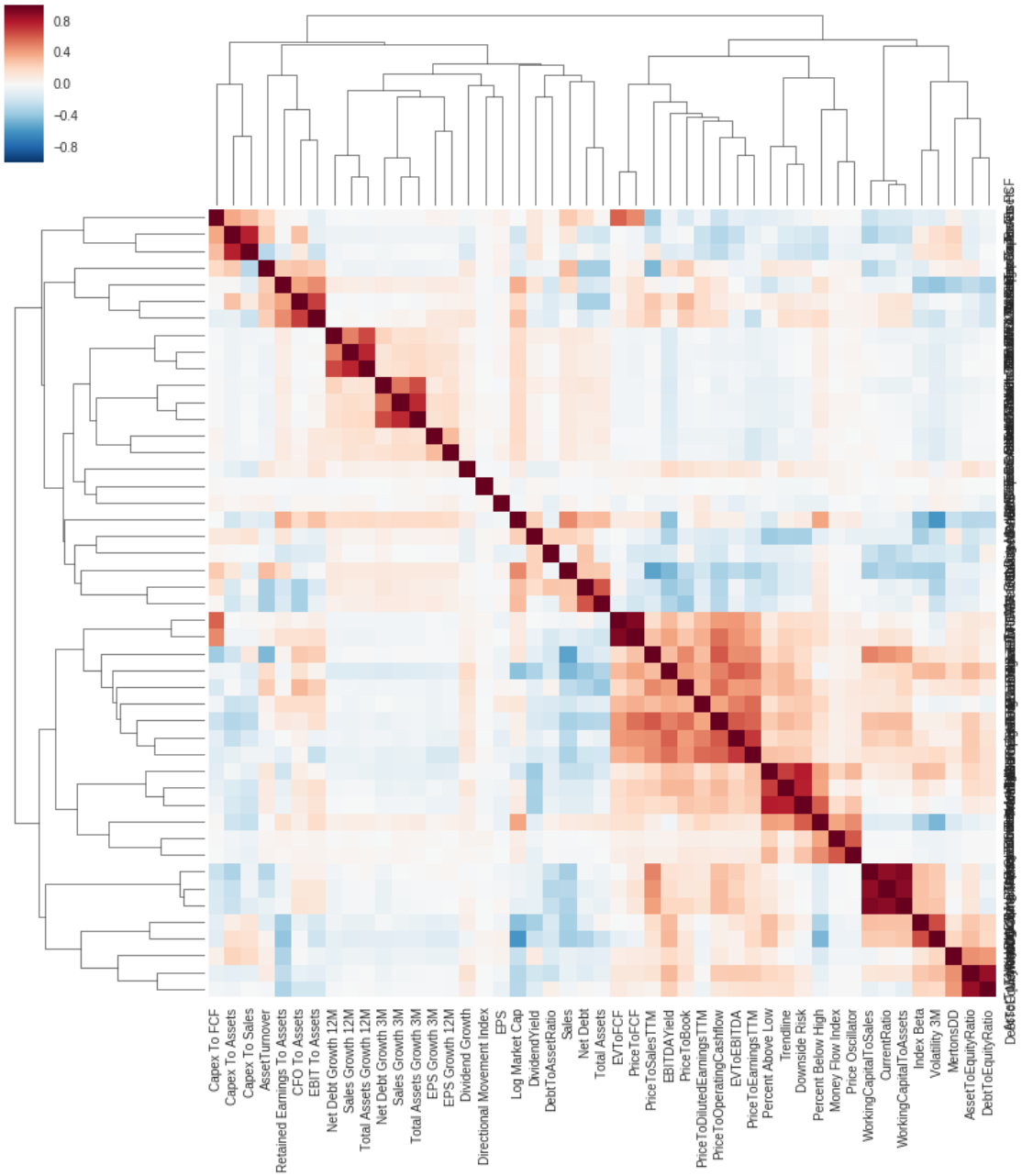
```
2,985 rows and 3 columns dropped
```

[38]:
```
data.sort_index(1).info()
```

```
<class 'pandas.core.frame.DataFrame'>
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]))
Data columns (total 52 columns):
AssetToEquityRatio             47377 non-null float64
AssetTurnover                  47377 non-null float64
CFO To Assets                  47377 non-null float64
Capex To Assets                47377 non-null float64
Capex To FCF                   47377 non-null float64
Capex To Sales                 47377 non-null float64
```

```
CurrentRatio                    47377 non-null float64
DebtToAssetRatio                47377 non-null float64
DebtToEquityRatio               47377 non-null float64
Directional Movement Index      47377 non-null float64
Dividend Growth                 47377 non-null float64
DividendYield                   47377 non-null float64
Downside Risk                   47377 non-null float64
EBIT To Assets                  47377 non-null float64
EBITDAYield                     47377 non-null float64
EPS                             47377 non-null float64
EPS Growth 12M                  47377 non-null float64
EPS Growth 3M                   47377 non-null float64
EVToEBITDA                      47377 non-null float64
EVToFCF                         47377 non-null float64
Index Beta                      47377 non-null float64
Log Market Cap                  47377 non-null float64
MertonsDD                       47377 non-null float64
Money Flow Index                47377 non-null float64
Net Debt                        47377 non-null float64
Net Debt Growth 12M             47377 non-null float64
Net Debt Growth 3M              47377 non-null float64
Percent Above Low               47377 non-null float64
Percent Below High              47377 non-null float64
Price Oscillator                47377 non-null float64
PriceToBook                     47377 non-null float64
PriceToDilutedEarningsTTM       47377 non-null float64
PriceToEarningsTTM              47377 non-null float64
PriceToFCF                      47377 non-null float64
PriceToOperatingCashflow        47377 non-null float64
PriceToSalesTTM                 47377 non-null float64
Retained Earnings To Assets     47377 non-null float64
Returns10D                      47377 non-null float64
Returns1D                       47377 non-null float64
Returns20D                      47377 non-null float64
Returns5D                       47377 non-null float64
Sales                           47377 non-null float64
Sales Growth 12M                47377 non-null float64
Sales Growth 3M                 47377 non-null float64
Total Assets                    47377 non-null float64
Total Assets Growth 12M         47377 non-null float64
Total Assets Growth 3M          47377 non-null float64
Trendline                       47377 non-null float64
Volatility 3M                   47377 non-null float64
WorkingCapitalToAssets          47377 non-null float64
WorkingCapitalToSales           47377 non-null float64
stock                           47377 non-null object
dtypes: float64(51), object(1)
memory usage: 19.2+ MB
```

[39]:
```
g = sns.clustermap(data.drop(['stock'] + return_cols, axis=1).corr())
plt.gcf().set_size_inches((14,14));
```



Prepare Features

[40]:
```
X = pd.get_dummies(data.drop(return_cols, axis=1))
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
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: 182 entries, DividendYield to stock_YELP INC
dtypes: float64(182)
memory usage: 66.1+ MB
```

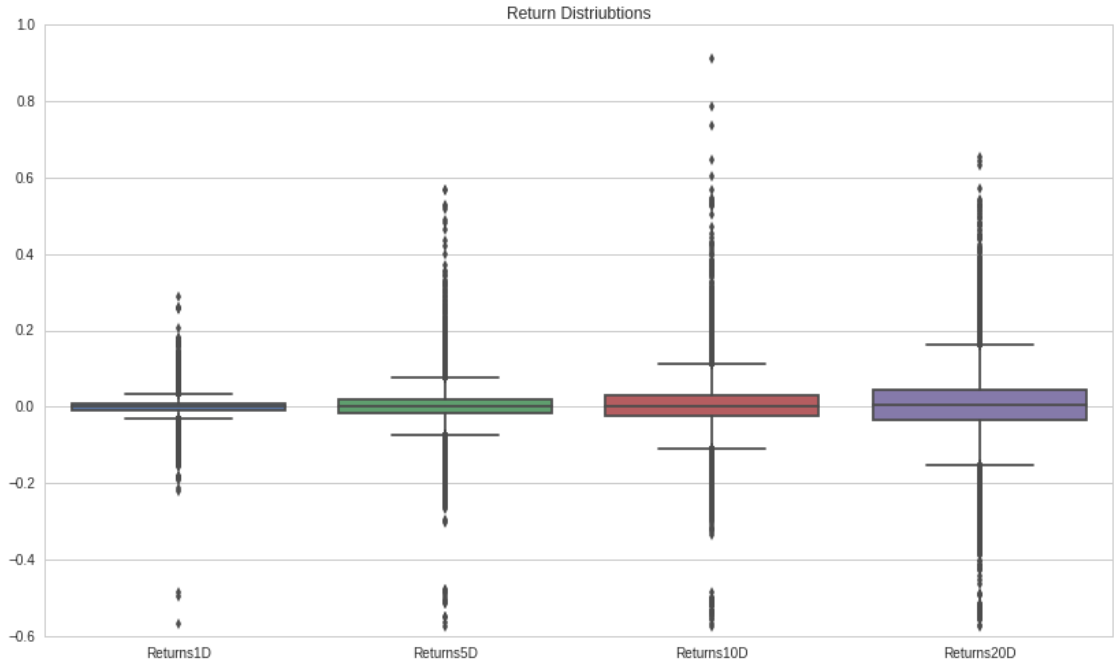
Shifted Returns

[41]:
```
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()
```

```
<class 'pandas.core.frame.DataFrame'>
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]))
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)
memory usage: 1.8+ MB
```

[42]:
```
ax = sns.boxplot(y[return_cols])
ax.set_title('Return Distriubtions');
```

```
/usr/local/lib/python2.7/dist-packages/seaborn/categorical.py:2171: UserWarning:
The boxplot API has been changed. Attempting to adjust your arguments for the
new API (which might not work). Please update your code. See the version 0.6
release notes for more info.
  warnings.warn(msg, UserWarning)
```



[44]:
```
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([outliers, 1-outliers]).values)]

model_data[target] = np.log1p(model_data[target])
features = model_data.drop(target, axis=1).columns
dates = model_data.index.unique()

print(model_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 45114 entries, 2014-01-02 to 2015-12-16
Columns: 183 entries, Returns10D to stock_YELP INC
dtypes: float64(183)
memory usage: 63.3 MB
None
```

[45]:
```
model_data[target].describe()
```

[45]:
```
count    45114.000000
mean         0.001159
std          0.045740
```

```
min         -0.157448
25%         -0.025013
50%          0.002817
75%          0.028880
max          0.146139
Name: Returns10D, dtype: float64
```

[46]:
```
idx = pd.IndexSlice
```

1.5.10 Logistic Regression: Classification

[49]:
```
def time_series_split(d, 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)
```

[47]:
```
target = 'Returns10D'
label = (y[target] > 0).astype(int).to_frame(target)
model_data = pd.concat([label, X], axis=1).dropna().reset_index('asset',␣
↪drop=True)

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

print(model_data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 47377 entries, 2014-01-02 to 2015-12-31
Columns: 183 entries, Returns10D to stock_YELP INC
dtypes: float64(182), int64(1)
memory usage: 66.5 MB
None
```

With this new categorical outcome variable, we can now train a logistic regression using the default L2 regularization. For logistic regression, the regularization is formulated inversely to linear regression: higher values for imply less regularization and vice versa. We evaluate 11 parameter values using cross validation as follows:

[62]:
```
nfolds = 250
Cs = np.logspace(-5, 5, 11)
```

```
scaler = StandardScaler()

logistic_results, logistic_coeffs = pd.DataFrame(), pd.DataFrame()
for C in Cs:
    print(C)
    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)
```

```
1e-05
0.0001
0.001
0.01
0.1
1.0
10.0
100.0
1000.0
10000.0
100000.0
```

1.5.11 Evaluate Results using AUC Score

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

[84]:
```
auc_by_C = logistic_results.groupby('C').apply(lambda x: roc_auc_score(y_true=x.
 ↪actuals.astype(int),
                                                        y_score=x.predicted))
```

[85]:
```
auc_by_C
```

[85]:
```
C
0.00001          0.532385
0.00010          0.539814
0.00100          0.549100
0.01000          0.554121
0.10000          0.561894
1.00000          0.569875
10.00000         0.574998
100.00000        0.575875
1000.00000       0.575214
10000.00000      0.574488
100000.00000     0.574317
dtype: float64
```

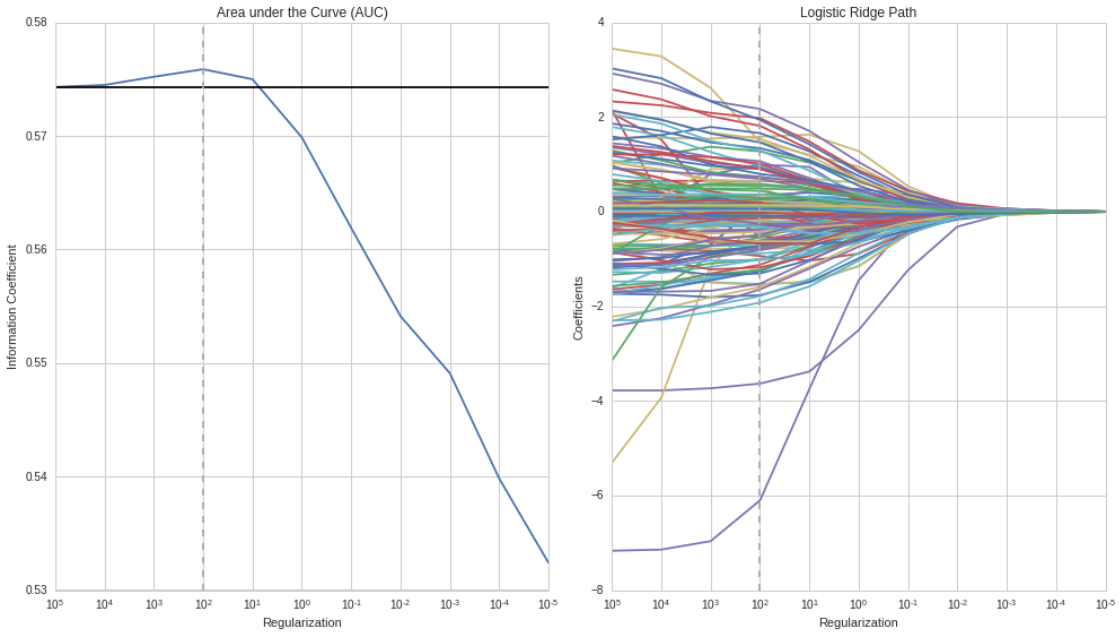
We can again plot the AUC result for the range of hyperparameter values alongside the coefficient path that shows the improvements in predictive accuracy as the coefficients are a bit shrunk at the optimal regularization value 102:

[86]:
```
base_auc = auc.iloc[-1]
best_auc = auc.max()
best_C = auc.idxmax()
```

[88]:
```
fig, axes = plt.subplots(ncols=2, sharex=True)

auc_by_C.sort_index(ascending=False).plot(logx=True, title='Area under the␣
 ↪Curve (AUC)', ax=axes[0])
axes[0].axhline(y=base_auc, c='black')
axes[0].axvline(x=best_C, c='darkgrey', ls='--')
axes[0].set_xlabel('Regularization')
axes[0].set_ylabel('Information Coefficient')

logistic_coeffs.T.sort_index(ascending=False).plot(legend=False, logx=True,␣
 ↪title='Logistic Ridge Path', ax=axes[1])
axes[1].set_xlabel('Regularization')
axes[1].set_ylabel('Coefficients')
axes[1].axvline(x=best_C, c='darkgrey', ls='--')
fig.tight_layout();
```



1.5.12 Ordinal Logit

[ ]:
```
target = 'Returns10D'
label = (y[target] > 0).astype(int).to_frame(target)
model_data = pd.concat([label, X], axis=1).dropna().reset_index('asset',␣
 ↪drop=True)

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

print(model_data.info())
```

[ ]:
```

```