00_data_prep

September 29, 2021

1 How to transform data into factors

Based on a conceptual understanding of key factor categories, their rationale and popular metrics, a key task is to identify new factors that may better capture the risks embodied by the return drivers laid out previously, or to find new ones.

In either case, it will be important to compare the performance of innovative factors to that of known factors to identify incremental signal gains.

We create the dataset here and store it in our data folder to facilitate reuse in later chapters.

1.1 Imports & Settings

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline

import numpy as np
import pandas as pd
import pandas_datareader.data as web

# from pyfinance.ols import PandasRollingOLS
# replaces pyfinance.ols.PandasRollingOLS (no longer maintained)
from statsmodels.regression.rolling import RollingOLS
import statsmodels.api as sm
from talib import RSI, BBANDS, MACD, NATR, ATR

from sklearn.feature_selection import mutual_info_classif,␣
↪mutual_info_regression

import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```
sns.set_style('whitegrid')
idx = pd.IndexSlice
```

1.2 Load US equity OHLCV data

The `assets.h5` store can be generated using the the notebook create_datasets in the data directory in the root directory of this repo for instruction to download the following dataset.

We load the Quandl stock price datasets covering the US equity markets 2000-18 using `pd.IndexSlice` to perform a slice operation on the `pd.MultiIndex`, select the adjusted close price and unpivot the column to convert the DataFrame to wide format with tickers in the columns and timestamps in the rows:

Set data store location:

[4]:
```
DATA_STORE = '../data/assets.h5'
```

[5]:
```
YEAR = 12
```

[6]:
```
START = 1995
END = 2017
```

[7]:
```
with pd.HDFStore(DATA_STORE) as store:
    prices = (store['quandl/wiki/prices']
              .loc[idx[str(START):str(END), :], :]
              .filter(like='adj_')
              .dropna()
              .swaplevel()
              .rename(columns=lambda x: x.replace('adj_', ''))
              .join(store['us_equities/stocks']
                    .loc[:, ['sector']])
              .dropna())
```

[8]:
```
prices.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 10241831 entries, ('AAN', Timestamp('1995-01-03 00:00:00')) to
('ZUMZ', Timestamp('2017-12-29 00:00:00'))
Data columns (total 6 columns):
 #   Column  Non-Null Count     Dtype
---  ------  --------------     -----
 0   open    10241831 non-null  float64
 1   high    10241831 non-null  float64
 2   low     10241831 non-null  float64
 3   close   10241831 non-null  float64
 4   volume  10241831 non-null  float64
 5   sector  10241831 non-null  object
dtypes: float64(5), object(1)
memory usage: 508.7+ MB
```

[9]:
```
len(prices.index.unique('ticker'))
```

[9]: 2369

1.3 Remove stocks with less than ten years of data

[10]:
```
min_obs = 10 * 252
nobs = prices.groupby(level='ticker').size()
to_drop = nobs[nobs < min_obs].index
prices = prices.drop(to_drop, level='ticker')
```

[11]:
```
prices.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 9532628 entries, ('AAN', Timestamp('1995-01-03 00:00:00')) to
('ZUMZ', Timestamp('2017-12-29 00:00:00'))
Data columns (total 6 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   open    9532628 non-null  float64
 1   high    9532628 non-null  float64
 2   low     9532628 non-null  float64
 3   close   9532628 non-null  float64
 4   volume  9532628 non-null  float64
 5   sector  9532628 non-null  object
dtypes: float64(5), object(1)
memory usage: 473.5+ MB
```

[12]:
```
len(prices.index.unique('ticker'))
```

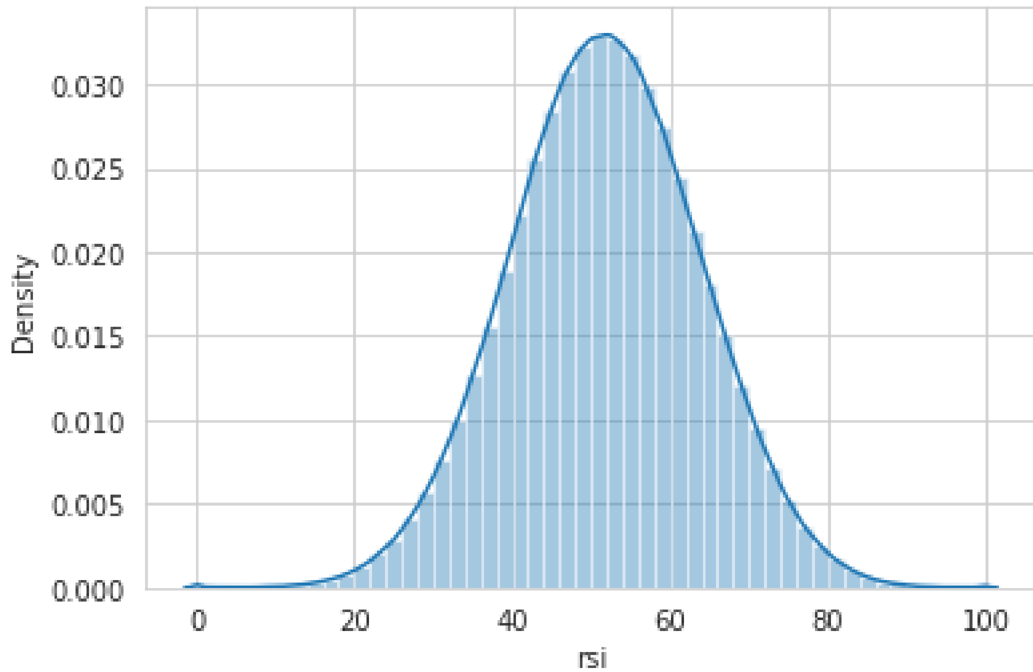
[12]: 1883

1.4 Add some Basic Factors

1.4.1 Compute the Relative Strength Index

[13]:
```
prices['rsi'] = prices.groupby(level='ticker').close.apply(RSI)
```

[14]:
```
sns.distplot(prices.rsi);
```



1.4.2 Compute Bollinger Bands

[15]:
```
def compute_bb(close):
    high, mid, low = BBANDS(np.log1p(close), timeperiod=20)
    return pd.DataFrame({'bb_high': high,
                         'bb_mid': mid,
                         'bb_low': low}, index=close.index)
```

[16]:
```
prices = (prices.join(prices
                      .groupby(level='ticker')
                      .close
                      .apply(compute_bb)))
```

[17]:
```
prices.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 9532628 entries, ('AAN', Timestamp('1995-01-03 00:00:00')) to
('ZUMZ', Timestamp('2017-12-29 00:00:00'))
Data columns (total 10 columns):
 #   Column   Non-Null Count    Dtype
---  ------   --------------    -----
 0   open     9532628 non-null  float64
 1   high     9532628 non-null  float64
 2   low      9532628 non-null  float64
 3   close    9532628 non-null  float64
```

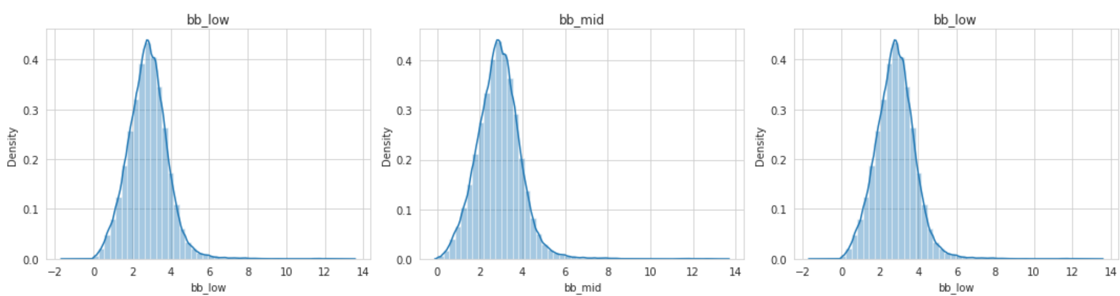
```
 4   volume   9532628 non-null  float64
 5   sector   9532628 non-null  object
 6   rsi      9506266 non-null  float64
 7   bb_high  9496851 non-null  float64
 8   bb_mid   9496851 non-null  float64
 9   bb_low   9496851 non-null  float64
dtypes: float64(9), object(1)
memory usage: 1022.4+ MB
```

[18]:
```python
prices.filter(like='bb_').describe()
```

[18]:

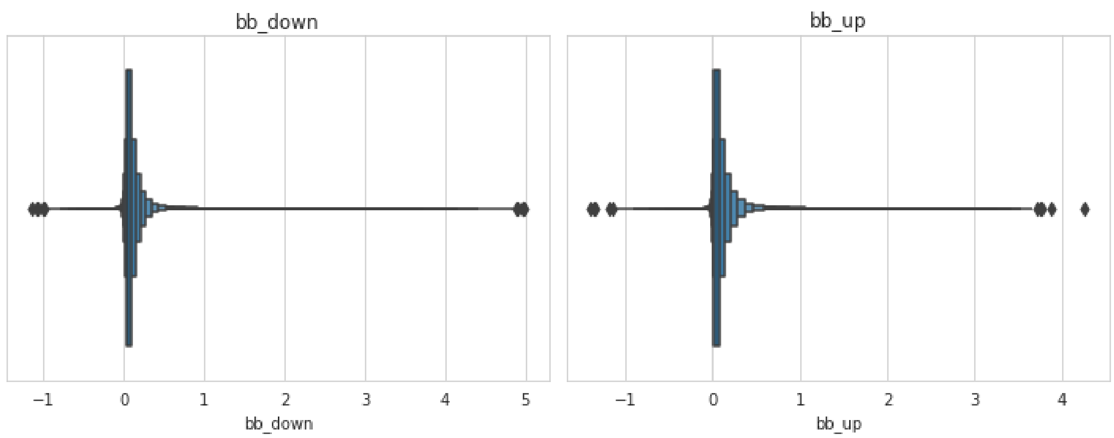
	bb_high	bb_mid	bb_low
count	9.496851e+06	9.496851e+06	9.496851e+06
mean	2.954140e+00	2.881157e+00	2.808174e+00
std	1.024536e+00	1.026901e+00	1.032999e+00
min	8.933146e-03	8.933146e-03	-1.568426e+00
25%	2.303724e+00	2.226078e+00	2.146471e+00
50%	2.940911e+00	2.868116e+00	2.796484e+00
75%	3.555602e+00	3.487039e+00	3.420498e+00
max	1.376991e+01	1.358056e+01	1.346225e+01

[19]:
```python
fig, axes = plt.subplots(ncols=3, figsize=(15,4))
for i, col in enumerate(['bb_low', 'bb_mid', 'bb_low']):
    sns.distplot(prices[col], ax=axes[i])
    axes[i].set_title(col);
fig.tight_layout();
```



[20]:
```python
prices['bb_up'] = prices.bb_high.sub(np.log1p(prices.close))
prices['bb_down'] = np.log1p(prices.close).sub(prices.bb_low)
```

[21]:
```python
fig, axes = plt.subplots(ncols=2, figsize=(10,4))
for i, col in enumerate(['bb_down', 'bb_up']):
    sns.boxenplot(prices[col], ax=axes[i])
    axes[i].set_title(col);
fig.tight_layout();
```



1.4.3 Compute Average True Range

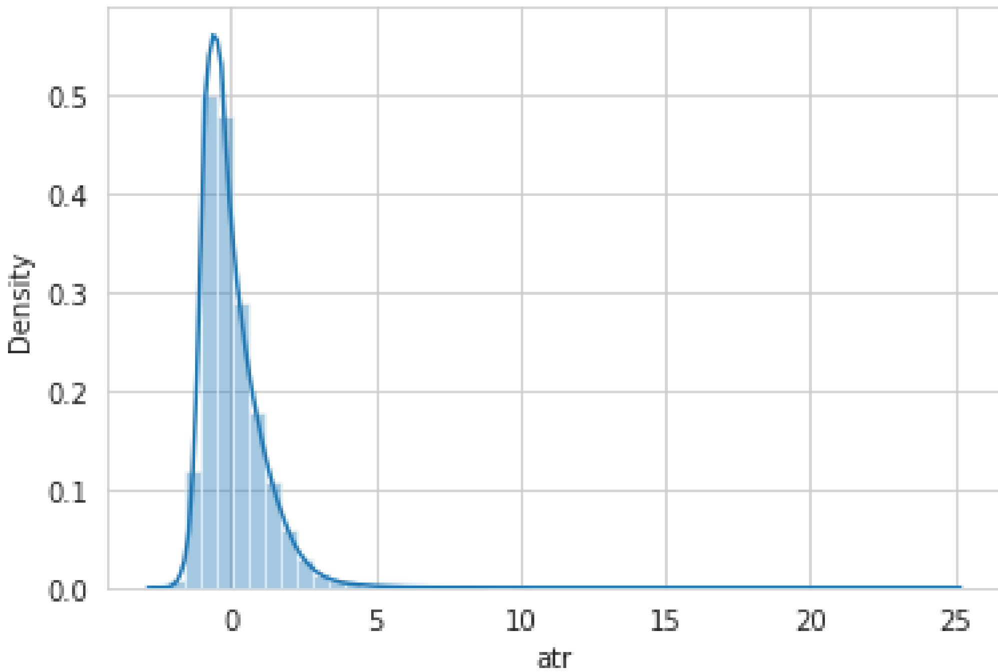
Helper for indicators with multiple inputs:

[22]:
```
by_ticker = prices.groupby('ticker', group_keys=False)
```

[23]:
```
def compute_atr(stock_data):
    atr = ATR(stock_data.high,
              stock_data.low,
              stock_data.close,
              timeperiod=14)
    return atr.sub(atr.mean()).div(atr.std())
```

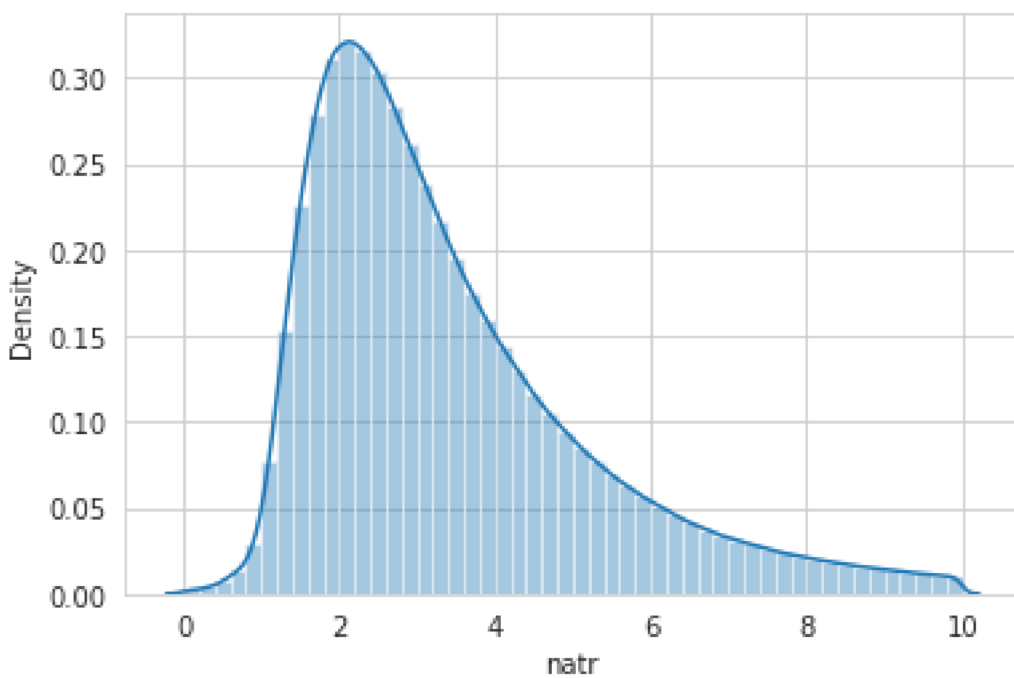
[24]:
```
prices['atr'] = by_ticker.apply(compute_atr)
```

[25]:
```
sns.distplot(prices.atr);
```




```
[26]: prices['natr'] = by_ticker.apply(lambda x: NATR(high=x.high, low=x.low, close=x.
      ↪close))
```

```
[27]: sns.distplot(prices.natr[prices.natr<10]);
```

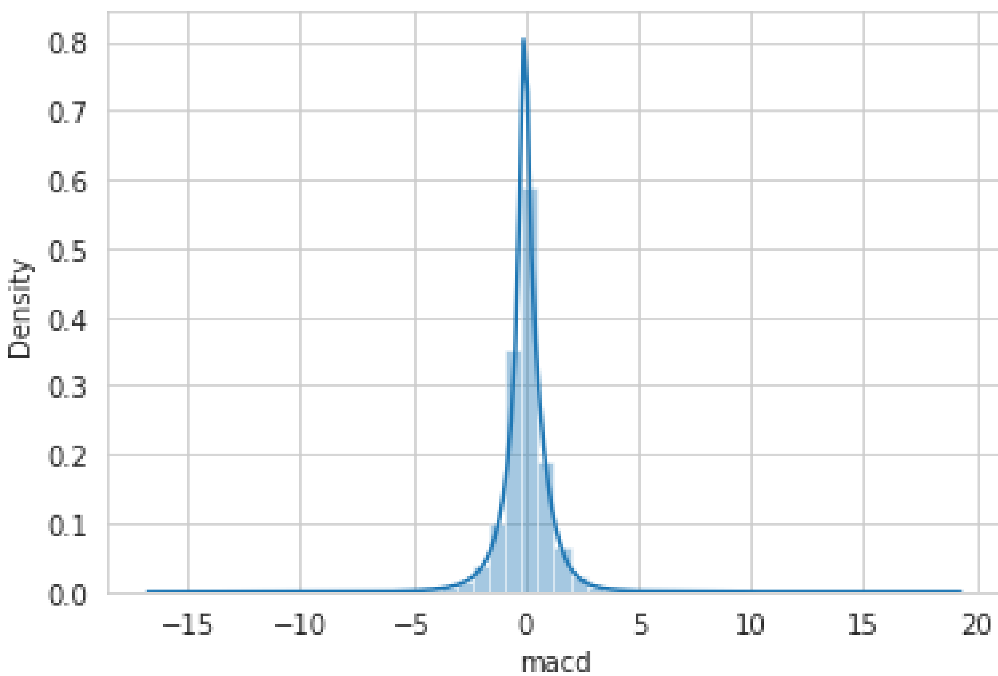


1.4.4 Compute Moving Average Convergence/Divergence

[28]:
```
def compute_macd(close):
    macd = MACD(close)[0]
    return macd.sub(macd.mean()).div(macd.std())

prices['macd'] = prices.groupby(level='ticker').close.apply(compute_macd)
```

[29]:
```
sns.distplot(prices.macd);
```



1.5 Compute dollar volume to determine universe

[30]:
```
prices['dollar_volume'] = (prices.loc[:, 'close']
                           .mul(prices.loc[:, 'volume'], axis=0))

prices.dollar_volume /= 1e6
```

[31]:
```
prices.to_hdf('data.h5', 'us/equities/prices')
```

[32]:
```
prices = pd.read_hdf('data.h5', 'us/equities/prices')
prices.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 9532628 entries, ('AAN', Timestamp('1995-01-03 00:00:00')) to
('ZUMZ', Timestamp('2017-12-29 00:00:00'))
Data columns (total 16 columns):
 #   Column         Non-Null Count    Dtype
---  ------         --------------    -----
 0   open           9532628 non-null  float64
 1   high           9532628 non-null  float64
 2   low            9532628 non-null  float64
 3   close          9532628 non-null  float64
 4   volume         9532628 non-null  float64
 5   sector         9532628 non-null  object
 6   rsi            9506266 non-null  float64
 7   bb_high        9496851 non-null  float64
 8   bb_mid         9496851 non-null  float64
 9   bb_low         9496851 non-null  float64
 10  bb_up          9496851 non-null  float64
 11  bb_down        9496851 non-null  float64
 12  atr            9506266 non-null  float64
 13  natr           9506266 non-null  float64
 14  macd           9470489 non-null  float64
 15  dollar_volume  9532628 non-null  float64
dtypes: float64(15), object(1)
memory usage: 1.2+ GB
```

1.6 Resample OHLCV prices to monthly frequency

To reduce training time and experiment with strategies for longer time horizons, we convert the business-daily data to month-end frequency using the available adjusted close price:

[33]:
```
last_cols = [c for c in prices.columns.unique(0) if c not in ['dollar_volume',␣
↪'volume',
                                                              'open', 'high',␣
↪'low']]
```

[34]:
```
prices = prices.unstack('ticker')
```

[35]:
```
data = (pd.concat([prices.dollar_volume.resample('M').mean().stack('ticker').
↪to_frame('dollar_volume'),
                   prices[last_cols].resample('M').last().stack('ticker')],
                  axis=1)
        .swaplevel()
        .dropna())
```

[36]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 452529 entries, ('AAN', Timestamp('1995-02-28 00:00:00')) to
```

```
('ZUMZ', Timestamp('2017-12-31 00:00:00'))
Data columns (total 12 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   dollar_volume  452529 non-null  float64
 1   atr            452529 non-null  float64
 2   bb_down        452529 non-null  float64
 3   bb_high        452529 non-null  float64
 4   bb_low         452529 non-null  float64
 5   bb_mid         452529 non-null  float64
 6   bb_up          452529 non-null  float64
 7   close          452529 non-null  float64
 8   macd           452529 non-null  float64
 9   natr           452529 non-null  float64
 10  rsi            452529 non-null  float64
 11  sector         452529 non-null  object
dtypes: float64(11), object(1)
memory usage: 43.2+ MB
```

1.7 Select 500 most-traded equities

Select the 500 most-traded stocks based on a 5-year rolling average of dollar volume.

[37]:
```
data['dollar_volume'] = (data.loc[:, 'dollar_volume']
                         .unstack('ticker')
                         .rolling(window=5*12, min_periods=12)
                         .mean()
                         .stack()
                         .swaplevel())
```

[38]:
```
data['dollar_vol_rank'] = (data
                           .groupby('date')
                           .dollar_volume
                           .rank(ascending=False))

data = data[data.dollar_vol_rank < 500].drop(['dollar_volume',␣
 ↪'dollar_vol_rank'], axis=1)
```

[39]:
```
len(data.index.unique('ticker'))
```

[39]:
```
905
```

1.8 Create monthly return series

To capture time series dynamics that reflect, for example, momentum patterns, we compute historical returns using the method `.pct_change(n_periods)`, that is, returns over various monthly periods as identified by lags.

We then convert the wide result back to long format with the `.stack()` method, use `.pipe()` to apply the `.clip()` method to the resulting `DataFrame`, and winsorize returns at the [1%, 99%] levels; that is, we cap outliers at these percentiles.

Finally, we normalize returns using the geometric average. After using `.swaplevel()` to change the order of the `MultiIndex` levels, we obtain compounded monthly returns for six periods ranging from 1 to 12 months:

[40]:
```
outlier_cutoff = 0.01
lags = [1, 3, 6, 12]
returns = []
```

[41]:
```
for lag in lags:
    returns.append(data
                   .close
                   .unstack('ticker')
                   .sort_index()
                   .pct_change(lag)
                   .stack('ticker')
                   .pipe(lambda x: x.clip(lower=x.quantile(outlier_cutoff),
                                          upper=x.quantile(1-outlier_cutoff)))
                   .add(1)
                   .pow(1/lag)
                   .sub(1)
                   .to_frame(f'return_{lag}m')
                   )

returns = pd.concat(returns, axis=1).swaplevel()
returns.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 202879 entries, ('AAPL', Timestamp('1996-02-29 00:00:00')) to
('ZIXI', Timestamp('2017-12-31 00:00:00'))
Data columns (total 4 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   return_1m   202879 non-null  float64
 1   return_3m   201069 non-null  float64
 2   return_6m   198362 non-null  float64
 3   return_12m  192972 non-null  float64
dtypes: float64(4)
memory usage: 7.0+ MB
```

[42]:
```
returns.describe()
```

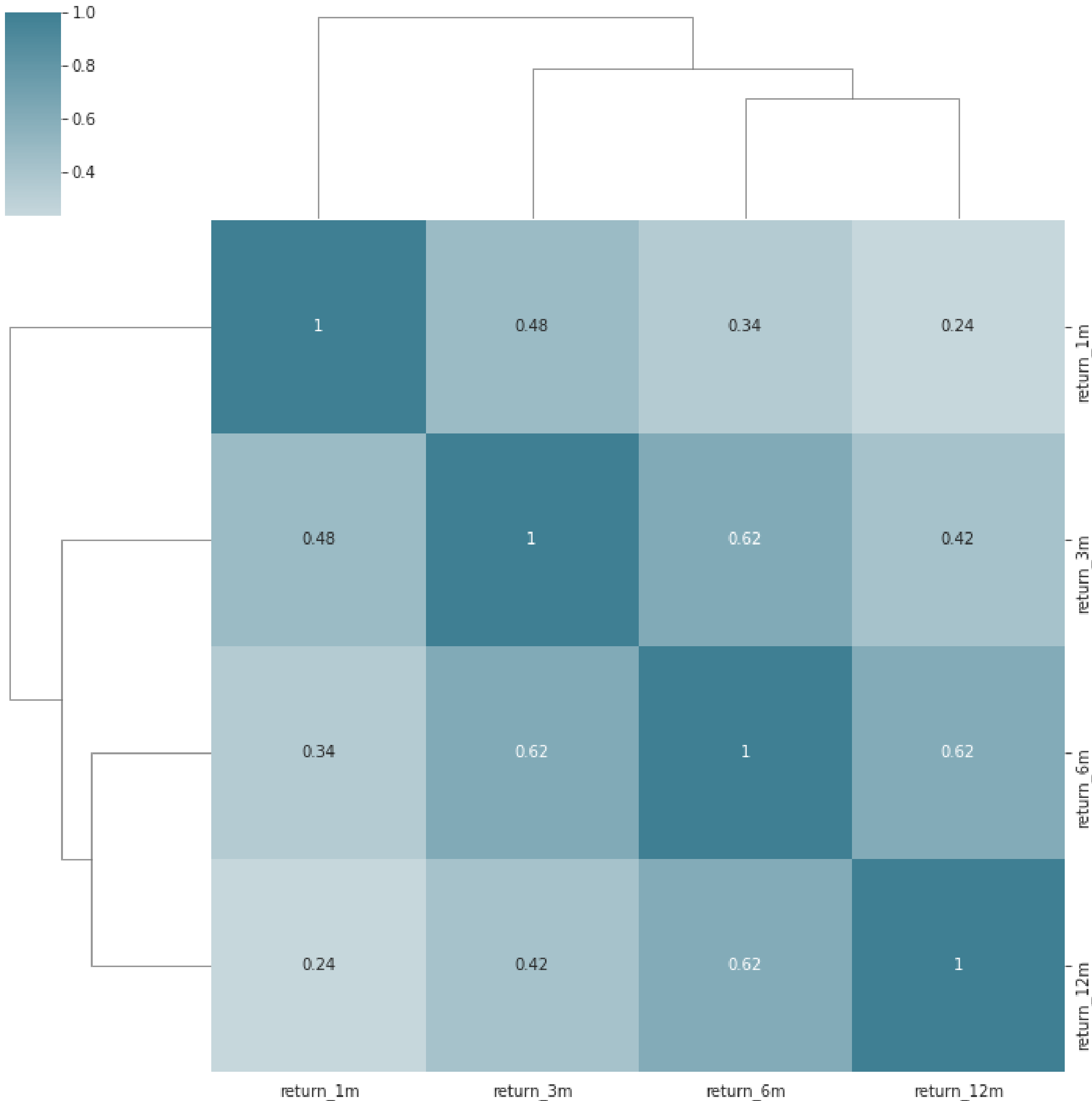
[42]:

	return_1m	return_3m	return_6m	return_12m
count	202879.000000	201069.000000	198362.000000	192972.000000
mean	0.007333	0.004950	0.004345	0.004167

```
std       0.085992    0.050986    0.037222    0.027246
min      -0.273767   -0.179499   -0.134177   -0.093876
25%      -0.011852   -0.003188   -0.000486    0.000000
50%       0.000000    0.000000    0.000000    0.000000
75%       0.032949    0.023727    0.018920    0.015461
max       0.331175    0.180733    0.131169    0.099726
```

[43]:

```python
cmap = sns.diverging_palette(10, 220, as_cmap=True)
sns.clustermap(returns.corr('spearman'), annot=True, center=0, cmap=cmap);
```



[44]:

```python
data = data.join(returns).drop('close', axis=1).dropna()
data.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 121589 entries, ('AAPL', Timestamp('1997-01-31 00:00:00')) to
('ZION', Timestamp('2017-12-31 00:00:00'))
Data columns (total 14 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   atr         121589 non-null  float64
 1   bb_down     121589 non-null  float64
 2   bb_high     121589 non-null  float64
 3   bb_low      121589 non-null  float64
 4   bb_mid      121589 non-null  float64
 5   bb_up       121589 non-null  float64
 6   macd        121589 non-null  float64
 7   natr        121589 non-null  float64
 8   rsi         121589 non-null  float64
 9   sector      121589 non-null  object
 10  return_1m   121589 non-null  float64
 11  return_3m   121589 non-null  float64
 12  return_6m   121589 non-null  float64
 13  return_12m  121589 non-null  float64
dtypes: float64(13), object(1)
memory usage: 13.5+ MB
```

[45]:
```
min_obs = 5*12
nobs = data.groupby(level='ticker').size()
to_drop = nobs[nobs < min_obs].index
data = data.drop(to_drop, level='ticker')
```

[46]:
```
len(data.index.unique('ticker'))
```

[46]: 613

We are left with 613 tickers.

1.9 Rolling Factor Betas

We will introduce the Fama—French data to estimate the exposure of assets to common risk factors using linear regression in Chapter 8, Time Series Models.

The five Fama—French factors, namely market risk, size, value, operating profitability, and investment have been shown empirically to explain asset returns and are commonly used to assess the risk/return profile of portfolios. Hence, it is natural to include past factor exposures as financial features in models that aim to predict future returns.

We can access the historical factor returns using the `pandas-datareader` and estimate historical exposures using the `PandasRollingOLS` rolling linear regression functionality in the `pyfinance` library as follows:

Use Fama-French research factors to estimate the factor exposures of the stock in the dataset to the 5 factors market risk, size, value, operating profitability and investment.

[47]:
```python
factors = ['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA']
factor_data = web.DataReader('F-F_Research_Data_5_Factors_2x3',
                             'famafrench',
                             start=START)[0].drop('RF', axis=1)
factor_data.index = factor_data.index.to_timestamp()
factor_data = factor_data.resample('M').last().div(100)
factor_data.index.name = 'date'
factor_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 314 entries, 1995-01-31 to 2021-02-28
Freq: M
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Mkt-RF  314 non-null    float64
 1   SMB     314 non-null    float64
 2   HML     314 non-null    float64
 3   RMW     314 non-null    float64
 4   CMA     314 non-null    float64
dtypes: float64(5)
memory usage: 14.7 KB
```

[48]:
```python
factor_data = factor_data.join(data['return_1m']).dropna().sort_index()
factor_data['return_1m'] -= factor_data['Mkt-RF']
factor_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 115181 entries, ('A', Timestamp('2001-12-31 00:00:00', freq='M')) to
('ZION', Timestamp('2017-12-31 00:00:00', freq='M'))
Data columns (total 6 columns):
 #   Column     Non-Null Count   Dtype  
---  ------     --------------   -----  
 0   Mkt-RF     115181 non-null  float64
 1   SMB        115181 non-null  float64
 2   HML        115181 non-null  float64
 3   RMW        115181 non-null  float64
 4   CMA        115181 non-null  float64
 5   return_1m  115181 non-null  float64
dtypes: float64(6)
memory usage: 5.8+ MB
```

[49]:
```python
factor_data.describe()
```

[49]:

```
              Mkt-RF            SMB            HML            RMW  \
count  115181.000000  115181.000000  115181.000000  115181.000000
mean        0.006184       0.002261       0.001918       0.003343
```

```
std          0.044295       0.031484       0.031217       0.029219
min         -0.172300      -0.148900      -0.111200      -0.184800
25%         -0.019700      -0.016900      -0.015100      -0.011600
50%          0.011700       0.001100       0.000200       0.004200
75%          0.034900       0.022700       0.018000       0.014700
max          0.113500       0.180800       0.125800       0.133800

                CMA      return_1m
count  115181.00000  115181.000000
mean        0.00242       0.005456
std         0.02146       0.092551
min        -0.06860      -0.387267
25%        -0.01060      -0.043902
50%        -0.00020       0.002598
75%         0.01430       0.050718
max         0.09560       0.503475
```

[54]:

```python
T = 60
# betas = (factor_data
#          .groupby(level='ticker', group_keys=False)
#          .apply(lambda x: PandasRollingOLS(window=min(T, x.shape[0]-1),
#                                            y=x.return_1m,
#                                            x=x.drop('return_1m', axis=1)).
 ↪beta)
#         .rename(columns={'Mkt-RF': 'beta'}))
betas = (factor_data.groupby(level='ticker',
                             group_keys=False)
         .apply(lambda x: RollingOLS(endog=x.return_1m,
                                     exog=sm.add_constant(x.drop('return_1m',␣
 ↪axis=1)),
                                     window=min(T, x.shape[0]-1))
                .fit(params_only=True)
                .params
                .rename(columns={'Mkt-RF': 'beta'})
                .drop('const', axis=1)))
```

[55]:

```python
betas.describe().join(betas.sum(1).describe().to_frame('total'))
```

[55]:

```
               beta           SMB           HML           RMW           CMA  \
count  79014.000000  79014.000000  79014.000000  79014.000000  79014.000000
mean       0.067841      0.192202      0.116937     -0.012575      0.004220
std        0.477380      0.587193      0.821477      0.877605      0.965009
min       -1.825339     -1.960546     -4.056384     -5.224988     -5.204906
25%       -0.252872     -0.204076     -0.411242     -0.492922     -0.536780
50%        0.047046      0.151163      0.059990      0.052307      0.053130
75%        0.370298      0.546711      0.593241      0.543717      0.596435
max        2.688829      3.286469      4.716294      4.029144      5.129094
```

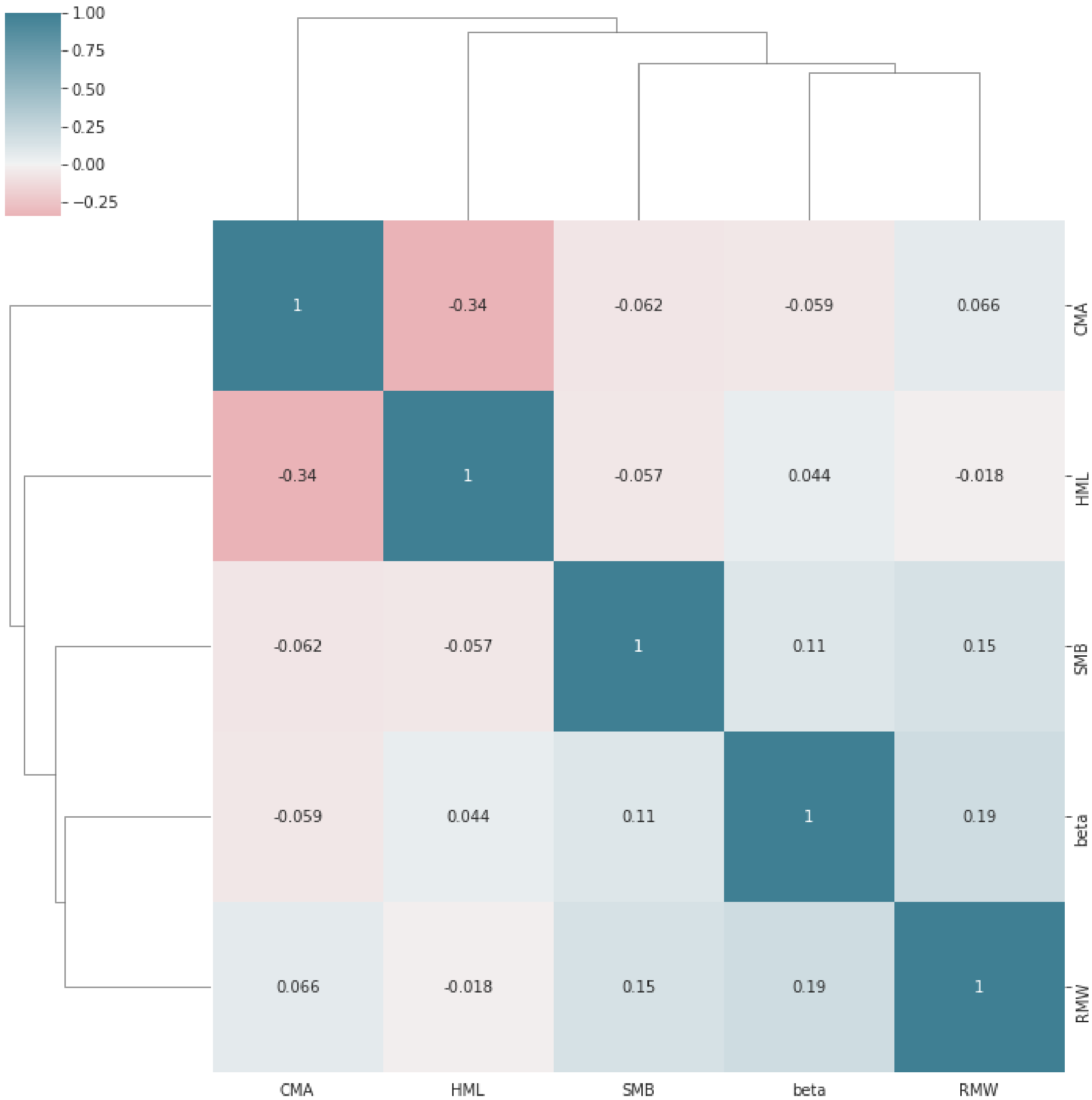
```
                total
count  115181.000000
mean        0.252877
std         1.377355
min       -10.257558
25%        -0.135471
50%         0.000000
75%         0.842115
max        10.580404
```

[57]:
```python
betas.describe().join(betas.sum(1).describe().to_frame('total'))
```

[57]:
```
               beta           SMB           HML           RMW           CMA  \
count  79014.000000  79014.000000  79014.000000  79014.000000  79014.000000
mean       0.067841      0.192202      0.116937     -0.012575      0.004220
std        0.477380      0.587193      0.821477      0.877605      0.965009
min       -1.825339     -1.960546     -4.056384     -5.224988     -5.204906
25%       -0.252872     -0.204076     -0.411242     -0.492922     -0.536780
50%        0.047046      0.151163      0.059990      0.052307      0.053130
75%        0.370298      0.546711      0.593241      0.543717      0.596435
max        2.688829      3.286469      4.716294      4.029144      5.129094

                total
count  115181.000000
mean        0.252877
std         1.377355
min       -10.257558
25%        -0.135471
50%         0.000000
75%         0.842115
max        10.580404
```

[58]:
```python
cmap = sns.diverging_palette(10, 220, as_cmap=True)
sns.clustermap(betas.corr(), annot=True, cmap=cmap, center=0);
```



[59]:
```
data = (data
        .join(betas
              .groupby(level='ticker')
              .shift())
        .dropna()
        .sort_index())
```

[60]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 78401 entries, ('A', Timestamp('2006-12-31 00:00:00')) to ('ZION',
Timestamp('2017-12-31 00:00:00'))
Data columns (total 19 columns):
```

```
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   atr          78401 non-null  float64
 1   bb_down      78401 non-null  float64
 2   bb_high      78401 non-null  float64
 3   bb_low       78401 non-null  float64
 4   bb_mid       78401 non-null  float64
 5   bb_up        78401 non-null  float64
 6   macd         78401 non-null  float64
 7   natr         78401 non-null  float64
 8   rsi          78401 non-null  float64
 9   sector       78401 non-null  object
 10  return_1m    78401 non-null  float64
 11  return_3m    78401 non-null  float64
 12  return_6m    78401 non-null  float64
 13  return_12m   78401 non-null  float64
 14  beta         78401 non-null  float64
 15  SMB          78401 non-null  float64
 16  HML          78401 non-null  float64
 17  RMW          78401 non-null  float64
 18  CMA          78401 non-null  float64
dtypes: float64(18), object(1)
memory usage: 11.8+ MB
```

1.10 Momentum factors

We can use these results to compute momentum factors based on the difference between returns over longer periods and the most recent monthly return, as well as for the difference between 3 and 12 month returns as follows:

[61]:
```
for lag in [3, 6, 12]:
    data[f'momentum_{lag}'] = data[f'return_{lag}m'].sub(data.return_1m)
    if lag > 3:
        data[f'momentum_3_{lag}'] = data[f'return_{lag}m'].sub(data.return_3m)
```

1.11 Date Indicators

[62]:
```
dates = data.index.get_level_values('date')
data['year'] = dates.year
data['month'] = dates.month
```

1.12 Target: Holding Period Returns

To compute returns for our one-month target holding period, we use the returns computed previously and shift them back to align them with the current financial features.

[63]:
```
data['target'] = data.groupby(level='ticker')[f'return_1m'].shift(-1)
```

[64]:
```
data = data.dropna()
```

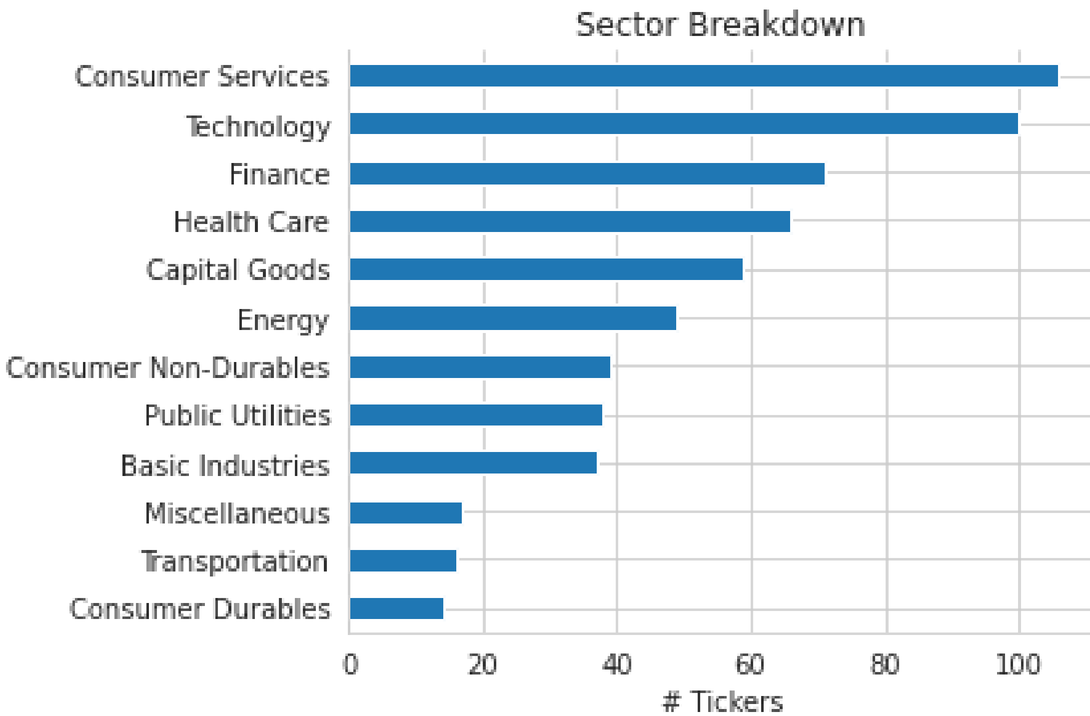
[65]:
```
data.sort_index().info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 77788 entries, ('A', Timestamp('2006-12-31 00:00:00')) to ('ZION',
Timestamp('2017-11-30 00:00:00'))
Data columns (total 27 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   atr            77788 non-null  float64
 1   bb_down        77788 non-null  float64
 2   bb_high        77788 non-null  float64
 3   bb_low         77788 non-null  float64
 4   bb_mid         77788 non-null  float64
 5   bb_up          77788 non-null  float64
 6   macd           77788 non-null  float64
 7   natr           77788 non-null  float64
 8   rsi            77788 non-null  float64
 9   sector         77788 non-null  object
 10  return_1m      77788 non-null  float64
 11  return_3m      77788 non-null  float64
 12  return_6m      77788 non-null  float64
 13  return_12m     77788 non-null  float64
 14  beta           77788 non-null  float64
 15  SMB            77788 non-null  float64
 16  HML            77788 non-null  float64
 17  RMW            77788 non-null  float64
 18  CMA            77788 non-null  float64
 19  momentum_3     77788 non-null  float64
 20  momentum_6     77788 non-null  float64
 21  momentum_3_6   77788 non-null  float64
 22  momentum_12    77788 non-null  float64
 23  momentum_3_12  77788 non-null  float64
 24  year           77788 non-null  int64
 25  month          77788 non-null  int64
 26  target         77788 non-null  float64
dtypes: float64(24), int64(2), object(1)
memory usage: 16.4+ MB
```

1.13 Sector Breakdown

[66]:
```
ax = data.reset_index().groupby('sector').ticker.nunique().sort_values().plot.
 ↪barh(title='Sector Breakdown')
ax.set_ylabel('')
ax.set_xlabel('# Tickers')
sns.despine()
```

```
plt.tight_layout();
```



1.14 Store data

[67]:
```
with pd.HDFStore('data.h5') as store:
    store.put('us/equities/monthly', data)
```

1.15 Evaluate mutual information

[68]:
```
X = data.drop('target', axis=1)
X.sector = pd.factorize(X.sector)[0]
```

[69]:
```
mi = mutual_info_regression(X=X, y=data.target)
```

[70]:
```
mi_reg = pd.Series(mi, index=X.columns)
mi_reg.nlargest(10)
```

[70]:
```
natr             0.111798
return_12m       0.060056
return_6m        0.054047
year             0.049061
return_3m        0.047197
momentum_3_12    0.039969
```

```
momentum_3_6     0.037950
bb_up            0.035975
momentum_12      0.035818
return_1m        0.034792
dtype: float64
```

[71]:
```python
mi = mutual_info_classif(X=X, y=(data.target>0).astype(int))
```

[72]:
```python
mi_class = pd.Series(mi, index=X.columns)
mi_class.nlargest(10)
```

[72]:
```
year          0.011498
month         0.006240
atr           0.005068
return_6m     0.004963
rsi           0.004232
return_12m    0.003727
sector        0.003231
RMW           0.002064
natr          0.001931
return_1m     0.001754
dtype: float64
```

[73]:
```python
mi = mi_reg.to_frame('Regression').join(mi_class.to_frame('Classification'))
```

[74]:
```python
mi.index = [' '.join(c.upper().split('_')) for c in mi.index]
```

[75]:
```python
fig, axes = plt.subplots(ncols=2, figsize=(12, 4))
for i, t in enumerate(['Regression', 'Classification']):
    mi[t].nlargest(20).sort_values().plot.barh(title=t, ax=axes[i])
    axes[i].set_xlabel('Mutual Information')
fig.suptitle('Mutual Information', fontsize=14)
sns.despine()
fig.tight_layout()
fig.subplots_adjust(top=.9)
```