04_preparing_the_model_data

September 29, 2021

1 Long-Short Strategy, Part 1: Preparing Alpha Factors and Features

In this section, we'll start designing, implementing, and evaluating a trading strategy for US equities driven by daily return forecasts produced by gradient boosting models.

As in the previous examples, we'll lay out a framework and build a specific example that you can adapt to run your own experiments. There are numerous aspects that you can vary, from the asset class and investment universe to more granular aspects like the features, holding period, or trading rules. See, for example, the **Alpha Factor Library** in the Appendix for numerous additional features.

We'll keep the trading strategy simple and only use a single ML signal; a real-life application will likely use multiple signals from different sources, such as complementary ML models trained on different datasets or with different lookahead or lookback periods. It would also use sophisticated risk management, from simple stop-loss to value-at-risk analysis.

Six notebooks cover our workflow sequence:

1. `preparing_the_model_data` (this noteboook): we'll engineer a few simple features from the Quandl Wiki data
2. trading_signals_with_lightgbm_and_catboost: we tune hyperparameters for LightGBM and CatBoost to select a model, using 2015/16 as our validation period.
3. evaluate_trading_signals: we compare the cross-validation performance using various metrics to select the best model.
4. model_interpretation: we take a closer look at the drivers behind the best model's predictions.
5. making_out_of_sample_predictions: we generate predictions for our out-of-sample test period 2017.
6. backtesting_with_zipline: evaluate the historical performance of a long-short strategy based on our predictive signals using Zipline.

1.1 Imports & Settings

[20]:
```
import warnings
warnings.filterwarnings('ignore')
```

[21]:
```
%matplotlib inline
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import talib
from talib import RSI, BBANDS, MACD, ATR
```

[22]:
```
MONTH = 21
YEAR = 12 * MONTH
```

[23]:
```
START = '2010-01-01'
END = '2017-12-31'
```

[24]:
```
sns.set_style('darkgrid')
idx = pd.IndexSlice
```

[25]:
```
percentiles = [.001, .01, .02, .03, .04, .05]
percentiles += [1-p for p in percentiles[::-1]]
```

[26]:
```
T = [1, 5, 10, 21, 42, 63]
```

1.2 Loading Quandl Wiki Stock Prices & Meta Data

[27]:
```
DATA_STORE = '../data/assets.h5'
ohlcv = ['adj_open', 'adj_close', 'adj_low', 'adj_high', 'adj_volume']
with pd.HDFStore(DATA_STORE) as store:
    prices = (store['quandl/wiki/prices']
              .loc[idx[START:END, :], ohlcv] # select OHLCV columns from 2010␣
↪until 2017
              .rename(columns=lambda x: x.replace('adj_', '')) # simplify␣
↪column names
              .swaplevel()
              .sort_index())
    metadata = (store['us_equities/stocks'].loc[:, ['marketcap', 'sector']])
```

[28]:
```
prices.volume /= 1e3 # make vol figures a bit smaller
prices.index.names = ['symbol', 'date']
metadata.index.name = 'symbol'
```

1.3 Remove stocks with insufficient observations

We require at least 7 years of data; we simplify and select using both in- and out-of-sample period; please be aware that it would be more accurate to use only the training period to remove data to avoid lookahead bias.

[29]:
```
min_obs = 7 * YEAR
nobs = prices.groupby(level='symbol').size()
keep = nobs[nobs > min_obs].index
prices = prices.loc[idx[keep, :], :]
```

1.3.1 Align price and meta data

[30]:
```
metadata = metadata[~metadata.index.duplicated() & metadata.sector.notnull()]
metadata.sector = metadata.sector.str.lower().str.replace(' ', '_')
```

[31]:
```
shared = (prices.index.get_level_values('symbol').unique()
          .intersection(metadata.index))
metadata = metadata.loc[shared, :]
prices = prices.loc[idx[shared, :], :]
```

1.3.2 Limit universe to 1,000 stocks with highest market cap

Again, we simplify and use the entire sample period, not just the training period, to select our universe.

[32]:
```
universe = metadata.marketcap.nlargest(1000).index
prices = prices.loc[idx[universe, :], :]
metadata = metadata.loc[universe]
```

[33]:
```
metadata.sector.value_counts()
```

[33]:
```
consumer_services        187
finance                  168
technology               116
health_care              103
capital_goods             94
basic_industries          67
public_utilities          66
consumer_non-durables     61
energy                    51
consumer_durables         36
miscellaneous             28
transportation            23
Name: sector, dtype: int64
```

[34]:
```
prices.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 2004775 entries, ('AAPL', Timestamp('2010-01-04 00:00:00')) to
('NTCT', Timestamp('2017-12-29 00:00:00'))
Data columns (total 5 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   open    2004775 non-null  float64
 1   close   2004775 non-null  float64
 2   low     2004775 non-null  float64
 3   high    2004775 non-null  float64
 4   volume  2004775 non-null  float64
```

```
dtypes: float64(5)
memory usage: 84.9+ MB
```

[35]:
```python
metadata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1000 entries, AAPL to NTCT
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   marketcap  1000 non-null   float64
 1   sector     1000 non-null   object
dtypes: float64(1), object(1)
memory usage: 23.4+ KB
```

1.3.3 Rank assets by Rolling Average Dollar Volume

Compute dollar volume

[36]:
```python
prices['dollar_vol'] = prices[['close', 'volume']].prod(1).div(1e3)
```

21-day moving average

[40]:
```python
# compute dollar volume to determine universe
dollar_vol_ma = (prices
                 .dollar_vol
                 .unstack('symbol')
                 .rolling(window=21, min_periods=1) # 1 trading month
                 .mean())
```

Rank stocks by moving average

[41]:
```python
prices['dollar_vol_rank'] = (dollar_vol_ma
                            .rank(axis=1, ascending=False)
                            .stack('symbol')
                            .swaplevel())
```

[42]:
```python
prices.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 2004775 entries, ('AAPL', Timestamp('2010-01-04 00:00:00')) to
('NTCT', Timestamp('2017-12-29 00:00:00'))
Data columns (total 7 columns):
 #   Column           Non-Null Count    Dtype
---  ------           --------------    -----
 0   open             2004775 non-null  float64
 1   close            2004775 non-null  float64
 2   low              2004775 non-null  float64
 3   high             2004775 non-null  float64
```

```
 4   volume           2004775 non-null  float64
 5   dollar_vol       2004775 non-null  float64
 6   dollar_vol_rank  2004775 non-null  float64
dtypes: float64(7)
memory usage: 115.5+ MB
```

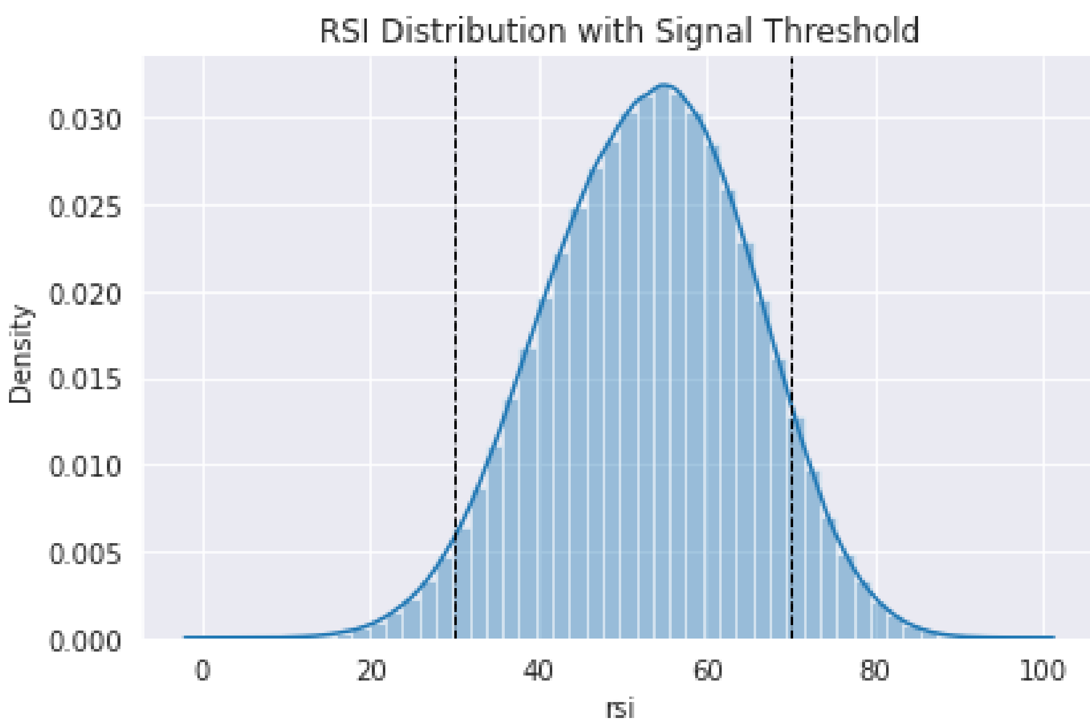
1.4 Add some Basic Factors

See appendix for details on the below indicators.

1.4.1 Compute the Relative Strength Index

[43]:
```
prices['rsi'] = prices.groupby(level='symbol').close.apply(RSI)
```

[44]:
```
ax = sns.distplot(prices.rsi.dropna())
ax.axvline(30, ls='--', lw=1, c='k')
ax.axvline(70, ls='--', lw=1, c='k')
ax.set_title('RSI Distribution with Signal Threshold')
sns.despine()
plt.tight_layout();
```



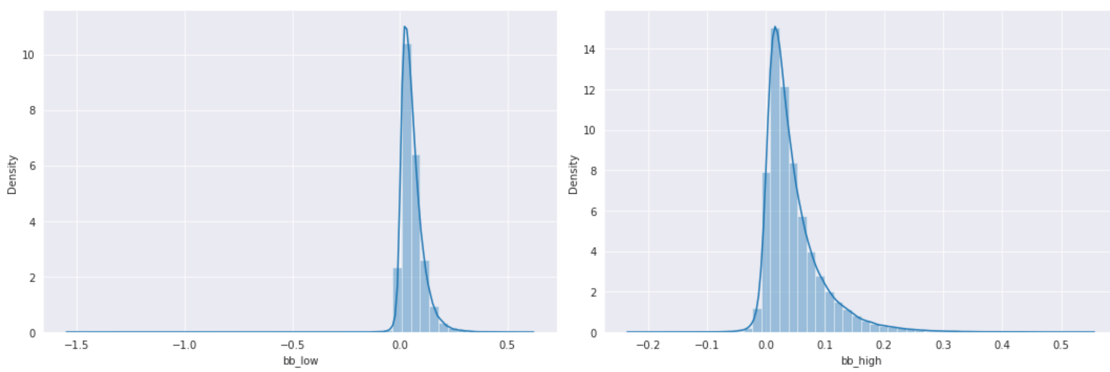
1.4.2 Compute Bollinger Bands

[45]:
```
def compute_bb(close):
    high, mid, low = BBANDS(close, timeperiod=20)
    return pd.DataFrame({'bb_high': high, 'bb_low': low}, index=close.index)
```

[46]:
```
prices = (prices.join(prices
                      .groupby(level='symbol')
                      .close
                      .apply(compute_bb)))
```

[47]:
```
prices['bb_high'] = prices.bb_high.sub(prices.close).div(prices.bb_high).
 ↪apply(np.log1p)
prices['bb_low'] = prices.close.sub(prices.bb_low).div(prices.close).apply(np.
 ↪log1p)
```

[48]:
```
fig, axes = plt.subplots(ncols=2, figsize=(15, 5))
sns.distplot(prices.loc[prices.dollar_vol_rank<100, 'bb_low'].dropna(),␣
 ↪ax=axes[0])
sns.distplot(prices.loc[prices.dollar_vol_rank<100, 'bb_high'].dropna(),␣
 ↪ax=axes[1])
sns.despine()
plt.tight_layout();
```



1.4.3 Compute Average True Range

[49]:
```
prices['NATR'] = prices.groupby(level='symbol',
                                group_keys=False).apply(lambda x:
                                                        talib.NATR(x.high, x.
 ↪low, x.close))
```

[50]:
```
def compute_atr(stock_data):
    df = ATR(stock_data.high, stock_data.low,
```

```
            stock_data.close, timeperiod=14)
    return df.sub(df.mean()).div(df.std())
```

[51]:
```
prices['ATR'] = (prices.groupby('symbol', group_keys=False)
                 .apply(compute_atr))
```

1.4.4 Compute Moving Average Convergence/Divergence

[52]:
```
prices['PPO'] = prices.groupby(level='symbol').close.apply(talib.PPO)
```

[53]:
```
def compute_macd(close):
    macd = MACD(close)[0]
    return (macd - np.mean(macd))/np.std(macd)
```

[54]:
```
prices['MACD'] = (prices
                  .groupby('symbol', group_keys=False)
                  .close
                  .apply(compute_macd))
```

1.4.5 Combine Price and Meta Data

[55]:
```
metadata.sector = pd.factorize(metadata.sector)[0].astype(int)
prices = prices.join(metadata[['sector']])
```

1.5 Compute Returns

1.5.1 Historical Returns

[56]:
```
by_sym = prices.groupby(level='symbol').close
for t in T:
    prices[f'r{t:02}'] = by_sym.pct_change(t)
```

1.5.2 Daily historical return deciles

[57]:
```
for t in T:
    prices[f'r{t:02}dec'] = (prices[f'r{t:02}']
                             .groupby(level='date')
                             .apply(lambda x: pd.qcut(x,
                                                      q=10,
                                                      labels=False,
                                                      duplicates='drop')))
```

1.5.3 Daily sector return deciles

[58]:
```
for t in T:
    prices[f'r{t:02}q_sector'] = (prices
                                  .groupby(['date', 'sector'])[f'r{t:02}']
                                  .transform(lambda x: pd.qcut(x,
                                                               q=5,
                                                               labels=False,
                                                              ␣
 ↪duplicates='drop')))
```

1.5.4 Compute Forward Returns

[59]:
```
for t in [1, 5, 21]:
    prices[f'r{t:02}_fwd'] = prices.groupby(level='symbol')[f'r{t:02}'].
 ↪shift(-t)
```

1.6 Remove outliers

[60]:
```
prices[[f'r{t:02}' for t in T]].describe()
```

[60]:
```
                r01           r05           r10           r21           r42  \
count  2.003775e+06  1.999775e+06  1.994775e+06  1.983775e+06  1.962775e+06
mean   7.519751e-04  3.726962e-03  7.353932e-03  1.555927e-02  3.113691e-02
std    2.166262e-02  4.791746e-02  6.579895e-02  9.467552e-02  1.325751e-01
min   -8.757416e-01 -8.768476e-01 -8.778415e-01 -8.802285e-01 -8.867366e-01
25%   -8.088407e-03 -1.721664e-02 -2.291896e-02 -3.045918e-02 -3.531712e-02
50%    6.561680e-04  3.702235e-03  7.173181e-03  1.503253e-02  2.899023e-02
75%    9.509191e-03  2.440601e-02  3.707177e-02  5.927618e-02  9.305628e-02
max    1.216425e+01  1.252657e+01  1.252657e+01  1.252657e+01  1.181643e+01

                r63
count  1.941775e+06
mean   4.619119e-02
std    1.618423e-01
min   -8.863481e-01
25%   -3.696833e-02
50%    4.217809e-02
75%    1.219666e-01
max    1.166968e+01
```

We remove daily returns above 100 percent as these are more likely to represent data errors; we are using the 100 percent cutoff here in a somewhat ad-hoc fashion; you would want to apply more careful exploratory and historical analysis to decide which assets are truly not representative of the sample period.

[61]:
```
outliers = prices[prices.r01 > 1].index.get_level_values('symbol').unique()
```

[62]:
```python
prices = prices.drop(outliers, level='symbol')
```

1.7 Create time and sector dummy variables

[63]:
```python
prices['year'] = prices.index.get_level_values('date').year
prices['month'] = prices.index.get_level_values('date').month
prices['weekday'] = prices.index.get_level_values('date').weekday
```

1.8 Store Model Data

[64]:
```python
prices.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 1994931 entries, ('AAPL', Timestamp('2010-01-04 00:00:00')) to
('NTCT', Timestamp('2017-12-29 00:00:00'))
Data columns (total 39 columns):
 #   Column           Non-Null Count    Dtype
---  ------           --------------    -----
 0   open             1994931 non-null  float64
 1   close            1994931 non-null  float64
 2   low              1994931 non-null  float64
 3   high             1994931 non-null  float64
 4   volume           1994931 non-null  float64
 5   dollar_vol       1994931 non-null  float64
 6   dollar_vol_rank  1994931 non-null  float64
 7   rsi              1981001 non-null  float64
 8   bb_high          1976026 non-null  float64
 9   bb_low           1976022 non-null  float64
 10  NATR             1981001 non-null  float64
 11  ATR              1981001 non-null  float64
 12  PPO              1970056 non-null  float64
 13  MACD             1962096 non-null  float64
 14  sector           1994931 non-null  int64
 15  r01              1993936 non-null  float64
 16  r05              1989956 non-null  float64
 17  r10              1984981 non-null  float64
 18  r21              1974036 non-null  float64
 19  r42              1953141 non-null  float64
 20  r63              1932246 non-null  float64
 21  r01dec           1993933 non-null  float64
 22  r05dec           1989956 non-null  float64
 23  r10dec           1984981 non-null  float64
 24  r21dec           1974036 non-null  float64
 25  r42dec           1953141 non-null  float64
 26  r63dec           1932246 non-null  float64
 27  r01q_sector      1993933 non-null  float64
 28  r05q_sector      1989956 non-null  float64
```

```
 29  r10q_sector      1984981 non-null  float64
 30  r21q_sector      1974036 non-null  float64
 31  r42q_sector      1953141 non-null  float64
 32  r63q_sector      1932246 non-null  float64
 33  r01_fwd          1993936 non-null  float64
 34  r05_fwd          1989956 non-null  float64
 35  r21_fwd          1974036 non-null  float64
 36  year             1994931 non-null  int64
 37  month            1994931 non-null  int64
 38  weekday          1994931 non-null  int64
dtypes: float64(35), int64(4)
memory usage: 602.0+ MB
```

[65]:
```
prices.drop(['open', 'close', 'low', 'high', 'volume'], axis=1).to_hdf('data.
↪h5', 'model_data')
```