06_ml_for_trading

September 29, 2021

1 ML for Trading: How to run an ML algorithm on Quantopian

The code in this notebook is written for the Quantopian Research Platform and uses the 'Algorithms' rather than the 'Research' option we used before.

To run it, you need to have a free Quantopian account, create a new algorithm and copy the content to the online development environment.

1.1 Imports & Settings

1.1.1 Quantopian Libraries

[ ]:
```
from quantopian.algorithm import attach_pipeline, pipeline_output,␣
 ↪order_optimal_portfolio
from quantopian.pipeline import Pipeline, factors, filters, classifiers
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.data import Fundamentals
from quantopian.pipeline.data.psychsignal import stocktwits
from quantopian.pipeline.factors import (Latest,
                                         CustomFactor,
                                         SimpleMovingAverage,
                                         AverageDollarVolume,
                                         Returns,
                                         RSI,
                                         SimpleBeta,
                                        ␣
 ↪MovingAverageConvergenceDivergenceSignal as MACD)
from quantopian.pipeline.filters import QTradableStocksUS
from quantopian.pipeline.experimental import risk_loading_pipeline, Size,␣
 ↪Momentum, Volatility, Value, ShortTermReversal

import quantopian.optimize as opt
from quantopian.optimize.experimental import RiskModelExposure
```

1.1.2 Other Python Libraries

[ ]:
```
from scipy.stats import spearmanr
import talib
import pandas as pd
import numpy as np
from time import time
from collections import OrderedDict

from scipy import stats
from sklearn import linear_model, preprocessing, metrics, cross_validation
from sklearn.pipeline import make_pipeline
```

1.1.3 Strategy Positions

[ ]:
```
# strategy parameters
N_POSITIONS = 100 # Will be split 50% long and 50% short
TRAINING_PERIOD = 126 # past periods for training
HOLDING_PERIOD = 5 # predict returns N days into the future

# How often to trade, for daily, alternative is date_rules.every_day()
TRADE_FREQ = date_rules.week_start()
```

1.1.4 Custom Universe

We define a custom universe to limit duration of training.

[ ]:
```
def Q250US():
    """Define custom universe"""
    return filters.make_us_equity_universe(
        target_size=250,
        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'),
    )
```

1.2 Create Alpha Factors

[ ]:
```
def make_alpha_factors():

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

    def PriceToEarningsTTM():
```

```
        """Closing price divided by earnings per share (EPS)"""
        return Fundamentals.pe_ratio.latest

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

    def Capex_To_Cashflows():
        return (Fundamentals.capital_expenditure.latest * 4.) / \
            (Fundamentals.free_cash_flow.latest * 4.)

    def EBITDA_Yield():
        return (Fundamentals.ebitda.latest * 4.) / \
            USEquityPricing.close.latest

    def EBIT_To_Assets():
        return (Fundamentals.ebit.latest * 4.) / \
            Fundamentals.total_assets.latest

    def Return_On_Total_Invest_Capital():
        return Fundamentals.roic.latest

    class Mean_Reversion_1M(CustomFactor):
        inputs = [Returns(window_length=21)]
        window_length = 252

        def compute(self, today, assets, out, monthly_rets):
            out[:] = (monthly_rets[-1] - np.nanmean(monthly_rets, axis=0)) / \
                np.nanstd(monthly_rets, axis=0)

    def MACD_Signal():
        return MACD(fast_period=12, slow_period=26, signal_period=9)

    def Net_Income_Margin():
        return Fundamentals.net_margin.latest

    def Operating_Cashflows_To_Assets():
        return (Fundamentals.operating_cash_flow.latest * 4.) / \
            Fundamentals.total_assets.latest

    def Price_Momentum_3M():
        return Returns(window_length=63)

    class Price_Oscillator(CustomFactor):
        inputs = [USEquityPricing.close]
        window_length = 252
```

```
        def compute(self, today, assets, out, close):
            four_week_period = close[-20:]
            out[:] = (np.nanmean(four_week_period, axis=0) /
                      np.nanmean(close, axis=0)) - 1.

    def Returns_39W():
        return Returns(window_length=215)

    class Vol_3M(CustomFactor):
        inputs = [Returns(window_length=2)]
        window_length = 63

        def compute(self, today, assets, out, rets):
            out[:] = np.nanstd(rets, axis=0)

    def Working_Capital_To_Assets():
        return Fundamentals.working_capital.latest / Fundamentals.total_assets.
↪latest

    def sentiment():
        return SimpleMovingAverage(inputs=[stocktwits.bull_minus_bear],
                                   window_length=5).rank(mask=universe)

    class AdvancedMomentum(CustomFactor):
        """ Momentum factor """
        inputs = [USEquityPricing.close,
                  Returns(window_length=126)]
        window_length = 252

        def compute(self, today, assets, out, prices, returns):
            out[:] = ((prices[-21] - prices[-252])/prices[-252] -
                      (prices[-1] - prices[-21])/prices[-21]) / np.
↪nanstd(returns, axis=0)

    def SPY_Beta():
        return SimpleBeta(target=sid(8554), regression_length=252)

    return {
        'Price to Sales': PriceToSalesTTM,
        'PE Ratio': PriceToEarningsTTM,
        'Dividend Yield': DividendYield,
        # 'Capex to Cashflows': Capex_To_Cashflows,
        # 'EBIT to Assets': EBIT_To_Assets,
        # 'EBITDA Yield': EBITDA_Yield,
        'MACD Signal Line': MACD_Signal,
        'Mean Reversion 1M': Mean_Reversion_1M,
        'Net Income Margin': Net_Income_Margin,
```

```
        # 'Operating Cashflows to Assets': Operating_Cashflows_To_Assets,
        'Price Momentum 3M': Price_Momentum_3M,
        'Price Oscillator': Price_Oscillator,
        # 'Return on Invested Capital': Return_On_Total_Invest_Capital,
        '39 Week Returns': Returns_39W,
        'Vol 3M': Vol_3M,
        'SPY_Beta': SPY_Beta,
        'Advanced Momentum': AdvancedMomentum,
        'Size': Size,
        'Volatitility': Volatility,
        'Value': Value,
        'Short-Term Reversal': ShortTermReversal,
        'Momentum': Momentum,
        # 'Materials': materials,
        # 'Consumer Discretionary': consumer_discretionary,
        # 'Financials': financials,
        # 'Real Estate': real_estate,
        # 'Consumer Staples': consumer_staples,
        # 'Healthcare': health_care,
        # 'Utilities': utilities,
        # 'Telecom ': telecom,
        # 'Energy': energy,
        # 'Industrials': industrials,
        # 'Technology': technology
    }
```

1.3 Custom Machine Learning Factor

Here we define a Machine Learning factor which trains a model and predicts forward returns

[ ]:
```
class ML(CustomFactor):
    init = False

    def compute(self, today, assets, out, returns, *inputs):
        """Train the model using
        - shifted returns as target, and
        - factors in a list of inputs as features;
            each factor contains a 2-D array of shape [time x stocks]
        """

        if (not self.init) or today.strftime('%A') == 'Monday':
            # train on first day then subsequent Mondays (memory)
            # get features
            features = pd.concat([pd.DataFrame(data, columns=assets).stack().
↪to_frame(i)
                                  for i, data in enumerate(inputs)], axis=1)
```

```
            # shift returns and align features
            target = (pd.DataFrame(returns, columns=assets)
                      .shift(-HOLDING_PERIOD)
                      .dropna(how='all')
                      .stack())
            target.index.rename(['date', 'asset'], inplace=True)
            features = features.reindex(target.index)

            # finalize features
            features = (pd.get_dummies(features
                                       .assign(asset=features
                                               .index.
↪get_level_values('asset')),
                                       columns=['asset'],
                                       sparse=True))


            # train the model
            self.model_pipe = make_pipeline(preprocessing.Imputer(),
                                            preprocessing.MinMaxScaler(),
                                            linear_model.LinearRegression())


            # run pipeline and train model
            self.model_pipe.fit(X=features, y=target)
            self.assets = assets # keep track of assets in model
            self.init = True

        # predict most recent factor values
        features = pd.DataFrame({i: d[-1] for i, d in enumerate(inputs)},␣
↪index=assets)
        features = features.reindex(index=self.assets).assign(asset=self.assets)
        features = pd.get_dummies(features, columns=['asset'])

        preds = self.model_pipe.predict(features)
        out[:] = pd.Series(preds, index=self.assets).reindex(index=assets)
```

1.4 Create Factor Pipeline

Create pipeline with predictive factors and target returns

[ ]:
```
def make_ml_pipeline(alpha_factors, universe, lookback=21, lookahead=5):
    """Create pipeline with predictive factors and target returns"""

    # set up pipeline
    pipe = OrderedDict()
```

```
    # Returns over lookahead days.
    pipe['Returns'] = Returns(inputs=[USEquityPricing.open],
                              mask=universe,
                              window_length=lookahead + 1)

    # Rank alpha factors:
    pipe.update({name: f().rank(mask=universe)
                 for name, f in alpha_factors.items()})

    # ML factor gets `lookback` datapoints on each factor
    pipe['ML'] = ML(inputs=pipe.values(),
                    window_length=lookback + 1,
                    mask=universe)

    return Pipeline(columns=pipe, screen=universe)
```

1.5 Define Algorithm

[ ]:
```
def initialize(context):
    """
    Called once at the start of the algorithm.
    """
    set_slippage(slippage.FixedSlippage(spread=0.00))
    set_commission(commission.PerShare(cost=0, min_trade_cost=0))

    schedule_function(rebalance_portfolio,
                      TRADE_FREQ,
                      time_rules.market_open(minutes=1))

    # Record tracking variables at the end of each day.
    schedule_function(log_metrics,
                      date_rules.every_day(),
                      time_rules.market_close())

    # Set up universe
    # base_universe = AverageDollarVolume(window_length=63,␣
↪mask=QTradableStocksUS()).percentile_between(80, 100)
    universe = AverageDollarVolume(window_length=63, mask=QTradableStocksUS()).
↪percentile_between(40, 60)

    # create alpha factors and machine learning pipline
    ml_pipeline = make_ml_pipeline(alpha_factors=make_alpha_factors(),
                                   universe=universe,
                                   lookback=TRAINING_PERIOD,
                                   lookahead=HOLDING_PERIOD)
    attach_pipeline(ml_pipeline, 'alpha_model')
```

```
    attach_pipeline(risk_loading_pipeline(), 'risk_loading_pipeline')

    context.past_predictions = {}
    context.realized_rmse = 0
    context.realized_ic = 0
    context.long_short_spread = 0
```

1.6 Evaluate Model

Evaluate model performance using past predictions on hold-out data

[ ]:
```
def evaluate_past_predictions(context):
    """Evaluate model performance using past predictions on hold-out data"""
    # A day has passed, shift days and drop old ones
    context.past_predictions = {k-1: v for k, v in context.past_predictions.
↪items() if k-1 >= 0}

    if 0 in context.past_predictions:
        # Past predictions for the current day exist, so we can use todays'␣
↪n-back returns to evaluate them
        returns = pipeline_output('alpha_model')['Returns'].to_frame('returns')

        df = (context
              .past_predictions[0]
              .to_frame('predictions')
              .join(returns, how='inner')
              .dropna())

        # Compute performance metrics
        context.realized_rmse = metrics.
↪mean_squared_error(y_true=df['returns'], y_pred=df.predictions)
        context.realized_ic, _ = spearmanr(df['returns'], df.predictions)
        log.info('rmse {:.2%} | ic {:.2%}'.format(context.realized_rmse,␣
↪context.realized_ic))

        long_rets = df.loc[df.predictions >= df.predictions.median(),␣
↪'returns'].mean()
        short_rets = df.loc[df.predictions < df.predictions.median(),␣
↪'returns'].mean()
        context.long_short_spread = (long_rets - short_rets) * 100

    # Store current predictions
    context.past_predictions[HOLDING_PERIOD] = context.predictions
```

1.7 Algo Execution

1.7.1 Prepare Trades

```
[ ]: def before_trading_start(context, data):
         """
         Called every day before market open.
         """
         context.predictions = pipeline_output('alpha_model')['ML']
         context.predictions.index.rename(['date', 'equity'], inplace=True)
         context.risk_loading_pipeline = pipeline_output('risk_loading_pipeline')
         evaluate_past_predictions(context)
```

1.7.2 Rebalance

```
[ ]: def rebalance_portfolio(context, data):
         """
         Execute orders according to our schedule_function() timing.
         """

         predictions = context.predictions
         predictions = predictions.loc[data.can_trade(predictions.index)]

         # Select long/short positions
         n_positions = int(min(N_POSITIONS, len(predictions)) / 2)
         to_trade = (predictions[predictions>0]
                     .nlargest(n_positions)
                     .append(predictions[predictions < 0]
                             .nsmallest(n_positions)))

         # Model may produce duplicate predictions
         to_trade = to_trade[~to_trade.index.duplicated()]

         # Setup Optimization Objective
         objective = opt.MaximizeAlpha(to_trade)

         # Setup Optimization Constraints
         constrain_gross_leverage = opt.MaxGrossExposure(1.0)
         constrain_pos_size = opt.PositionConcentration.with_equal_bounds(-.02, .02)
         market_neutral = opt.DollarNeutral()
         constrain_risk = RiskModelExposure(
             risk_model_loadings=context.risk_loading_pipeline,
             version=opt.Newest)

         # Optimizer calculates portfolio weights and
         # moves portfolio toward the target.
         order_optimal_portfolio(
             objective=objective,
```

```
        constraints=[
            constrain_gross_leverage,
            constrain_pos_size,
            market_neutral,
            constrain_risk
        ],
    )
```

1.7.3 Track Performance

[ ]:
```
def log_metrics(context, data):
    """
    Plot variables at the end of each day.
    """
    record(leverage=context.account.leverage,
           #num_positions=len(context.portfolio.positions),
           realized_rmse=context.realized_rmse,
           realized_ic=context.realized_ic,
           long_short_spread=context.long_short_spread,
    )
```