Quantitative Research Modeling Library

Operation Manual v1

Team
sdMay19-06

Team Members
Josiah Anderson -- Meeting Facilitator
Doh Yun Kim -- Scribe
Gabriel Klein -- Report Manager
Drake Mossman -- Communication Manager
Jacob Richards -- Quality Assurance Manager
Nathan Schaffer -- Overseer

Client
Joseph Byrum
(Principal Financial Group)

Advisor
Srikanta Tirthapura

Contact
sdmay19-06@iastate.edu
https://sdmay19-06.sd.ece.iastate.edu

Last Updated:
29 April 2019

Getting Started

To begin using the Dynamic Risk Premium 2.0 Pipeline (DRP) simply make sure the package is located somewhere on the python path and import it using the following line.

```
import DynamicRiskPremium as drp
```

Once imported, reference the other sections of this manual for information regarding the available functions and objects, or browse the well-commented code itself to understand the internal implementation.

Aggregation

Overview

The aggregation module of the pipeline is where you can create factor portfolios based on a percentile range of a factor across a universe of stocks in the database that include columns for input features and an output response.

Using the aggregation portion of the pipeline is as simple as creating an object describing the aggregation you want done, and then calling a function to do the aggregation. See the FactorPortfolioFactory section for information on how to describe the desired aggregation, and see the create_factor_portfolios section for information on how to start the aggregation process.

If you'd also like to preprocess the raw data or postprocess the resulting portfolio, see the sections for FactorPortfolioPreprocessor and FactorPortfolioPostprocessor.

FactorPortfolio

The FactorPortfolio object is the main output of the aggregation portion of the pipeline. It contains several attributes describing the kind of aggregation used to create it as well as the pandas DataFrame holding the aggregated data.

```
class FactorPortfolio:
    """
    This is the class representing a factor portfolio.

    Attributes:
        self:          the FactorPortfolio object
        start_date:    the start date of this portfolio as "yyyymmdd"
        end_date:      the end date of this portfolio as "yyyymmdd"
        data_period:   the period between raw stock level data points (weekly ('W') or monthly ('M'))
        prime_factor:  the factor on which we will apply the percentile division to choose stocks as a string
        pos_factor:    the expectation for the prime factor to be positively correlated with desirable buying (True or False)
        start_ptile:   the starting percentile of stocks in this portfolio based on this.prime_factor as an integer
        end_ptile:     the ending percentile of stocks in this portfolio based on this.prime_factor as an integer
        in_factors:    the factors to include in this portfolio as a list of strings
        in_predictors: the input predictors for aggregating this.in_factors as a list ("mean", "median", "skew", etc.)
        agg_type:      the method of aggregation for the portfolio ("Static", "Dynamic")
        ret_type:      the type of returns to be used ("Absolute", "Excess")
        ret_horizon:   the return horizon over which to aggregate ("1w", "1m", "3m", etc.)
        out_response:  the aggregation method for the returns in this portfolio ("mean", "var", "skew")
        dyn_rb_period: optional, for a dynamic portfolio, the time period for rebalancing ("1w", "1m", "3m", etc.)
        data:          a pandas Dataframe holding the factor portfolio data
        aggregated:    a boolean representing whether the portfolio is complete
        pred_data:     a pandas Dataframe holding the predicted data based off the model
        pred_id:       the prediction id that was used to run the predictions on this portfolio
    """
```

You can create a FactorPortfolio by simply providing values for each of the descriptive attributes, from start_date to out_response, and dyn_rb_period if necessary for a dynamic portfolio. The following picture shows a typical initialization.

```
date_range = (19900101, 19920101)
data_period = 'W'
prime_factor = 'bk_p'
pos_factor = True
in_factors = ["prev_12m_ret", "beta_1y", "fcf_p", "sales_g"]
ptile_range = (90, 100)
in_predictors = ["mean", "skew"]
ret_horizon = "1m"
ret_type = "Absolute"
agg_type = "Dynamic"
out_response = "mean"
dyn_rb_period = "1w"

fp = drp.FactorPortfolio(start_date=date_range[0], end_date=date_range[1], data_period=data_period,
                    prime_factor=prime_factor, pos_factor=pos_factor, start_ptile=ptile_range[0],
                    end_ptile=ptile_range[1], in_factors=in_factors, in_predictors=in_predictors,
                    agg_type=agg_type, ret_type=ret_type, ret_horizon=ret_horizon,
                    out_response=out_response, dyn_rb_period=dyn_rb_period)
```

Once you've described a factor portfolio using this object, you can then aggregate it using create_factor_portfolios. See the create_factor_portfolios section for more details. Usually you will want to aggregate more than one portfolio at a time, in which case a FactorPortfolioFactory may be more useful. See the FactorPortfolioFactory section for more details.

Once a FactorPortfolio has been aggregated (using create_factor_portfolios), its data attribute will contain a pandas DataFrame filled with aggregated data. The DataFrame will contain a row for every date between start_date and end_date either weekly or monthly depending on data_period and one column for each combination of in_factor and in_predictor formatted as (in_factor)_(in_predictor) along with one output response column formatted as

fut_(ret_horizon)[_excess]_ret[_dyn]_(out_response) where the strings in brackets are only included if the portfolio is over excess returns or uses dynamic aggregation respectively.

FactorPortfolios also have a handy display method for getting a quick look at their parameters and data.

```
fp.display()
```

```
Prime Factor:           bk_p
Period:                 weekly from 19900101 -> 19920101
Percentile Range:       90 -> 100
Return Statistic:       mean Absolute fut_1m_ret
Aggregation Info:       Dynamic aggregation with1w rebalancing
                fut_1m_ret_dyn_mean  prev_12m_ret_mean  beta_1y_mean  \
period_yyyymmdd
19900202                   4.872825         -19.827034      0.894551
19900209                   5.328519         -19.767323      0.898226
19900216                   5.468158         -19.739338      0.897317
19900223                   3.880360         -19.835472      0.860942
19900302                   0.407380         -18.724543      0.928537
19900309                  -2.886542         -18.650483      0.944520
19900316                  -1.244205         -18.379325      0.936103
19900323                  -2.532094         -17.419750      0.939337
19900330                  -5.792733         -17.682417      0.903265
19900406                  -2.328498         -20.677467      0.924637
19900413                  -1.185327         -19.083382      0.922669
19900420                   2.311571         -22.873274      1.077679
```

FactorPortfolioFactory

The FactorPortfolioFactory object is a tool used for aggregating many factor portfolios at once without having to individually specify each one. It contains essentially the same attributes that FactorPortfolios do, but as lists of options instead of one particular choice. When passed to create_factor_portfolios, the factory will generate a FactorPortfolio object for each combination of attributes it contains. Note that input factors and input predictors are shared across all generated factor portfolios, and do not allow for multiple options.

```
class FactorPortfolioFactory:
    """
    This is the class representing a factor portfolio factory. This class generates FactorPortfolios based
    on combinations of the attributes given to it.

    Attributes:
        self:           the FactorPortfolioFactory object
        date_ranges:    the date ranges for portfolios as a list of 2-tuples ("yyyymmdd", "yyyymmdd")
        data_periods:   the periods between raw stock level data points for portfolios
                        as a list of strings (weekly ('W') or monthly ('M'))
        prime_factors:  the factors on which we will apply the percentile divisions to choose stocks as a list of strings
        ptile_ranges:   the percentile ranges for which portfolios will be aggregated based on prime_factors
                        as a list of 2-tuples of integers
        in_factors:     the factors to include in portfolios as a list of strings
        in_predictors:  the input predictors for aggregating this.in_factors as a list ("mean", "median", "skew", etc.)
        agg_types:      the methods of aggregation for the portfolios as a list ("Static", "Dynamic")
        ret_types:      the types of returns to be used as a list ("Absolute", "Excess")
        ret_horizons:   the return horizons over which to aggregate as a list ("1w", "1m", "3m", etc.)
        out_responses:  the aggregation methods for the returns in the portfolios as a list ("mean", "var", "skew")
        dyn_rb_periods: optional, for dynamic portfolios, the time periods for rebalancing as a list ("1w", "1m", "3m", etc.)
        neg_factors:    optional, a subset of prime_factors which are expected to be negatively-correlated
                        with desirable buying as a list of strings
    """
```

Creating a FactorPortfolioFactory is very similar to creating a FactorPortfolio. See the following image as an example.

```
date_ranges = [(19900101, 19920101)]
data_periods = ['W']
prime_factors = ["bk_p", "sales_p"]
in_factors = ["prev_12m_ret", "beta_1y", "fcf_p", "sales_g"]
ptile_ranges = [(90, 100), (0, 10)]
in_predictors = ["mean", "skew"]
ret_horizons = ["1m", "3m", "6m"]
ret_types = ["Absolute", "Excess"]
agg_types = ["Static", "Dynamic"]
out_responses = ["mean", "skew"]
dyn_rb_periods = ["1w"]

factory = drp.FactorPortfolioFactory(date_ranges=date_ranges, data_periods=data_periods,
                    prime_factors=prime_factors, ptile_ranges=ptile_ranges,
                    in_factors=in_factors, in_predictors=in_predictors,
                    agg_types=agg_types, ret_types=ret_types, ret_horizons=ret_horizons,
                    out_responses=out_responses, dyn_rb_periods=dyn_rb_periods)
```

Note that the factory in this example is capable of creating 96 portfolios, as all possible combinations include (2 prime factors) * (2 percentile ranges) * (3 return horizons) * (2 return types) * (2 aggregation types) * (2 output responses) = 96 portfolios.

Finally, FactorPortfolioFactorys also include a method for generating all of their FactorPortfolios without using create_factor_portfolios. However, note that the portfolios generated will not be aggregated, so this is generally not a useful function. It is almost always better to pass the whole factory to create_factor_portfolios. See the following image for an example of generating the portfolios, unaggregated.

```
factory.generate_factor_portfolios()
```

FactorPortfolioPreprocessor

Often you will want to preprocess raw data before it gets aggregated into a portfolio. This can be done by implementing a subclass of the abstract base class (ABC) FactorPortfolioPreprocessor. ABCs are classes that define certain functions that subclasses are required to actually implement. The FactorPortfolioPreprocessor ABC has two such functions.

The first is the __str__ function. This function should be implemented to return a unique string representation for this particular preprocessor. In other words, it should always return a consistent string that no other preprocessor returns, so that it can be uniquely identified by the pipeline. The string can be anything, but a more descriptive one will be easier to trace back to a particular preprocessor.

The second function is called preprocess and has one parameter that allows for the raw data to be passed in. The data will be given as a pandas DataFrame with a row for every date/stock combination specified to be in the given portfolio, along with its input factor and output response information. This function is what allows you the chance to modify this DataFrame in place before it gets passed along to the actual aggregation part of the pipeline. You can do anything from remove outliers to removing certain undesirable sin stocks.

Once you've implemented your version of a preprocessor, simply pass an instance of it into create_factor_portfolios to use it. See the create_factor_portfolios section for more details.

FactorPortfolioPostprocessor

You may want to modify data after it's been aggregated into a portfolio. This can be done by implementing a subclass of the abstract base class (ABC) FactorPortfolioPostprocessor. ABCs are classes that define certain functions that subclasses are required to actually implement. The FactorPortfolioPostprocessor ABC has two such functions.

The first is the __str__ function. This function should be implemented to return a unique string representation for this particular postprocessor. In other words, it should always return a consistent string that no other postprocessor returns, so that it can be uniquely identified by the pipeline. The string can be anything, but a more descriptive one will be easier to trace back to a particular postprocessor.

The second function is called postprocess and has one parameter that allows for the aggregated portfolio to be passed in. The data will be given as a FactorPortfolio object, just as one would expect from the output of create_factor_portfolios. This function is what allows you the chance to modify this object in place before it gets passed along to the part of the pipeline that saves the results in the database.

Once you've implemented your version of a postprocessor, simply pass an instance of it into create_factor_portfolios to use it. See the create_factor_portfolios section for more details.

create_factor_portfolios

Once you're ready to start aggregating, create_factor_portfolios is the only function you need to know how to use. It's fairly self-explanatory, but a description of its parameters will follow.

```
def create_factor_portfolios(db_conn, portfolio=None, factory=None, identifier='ticker',
                             preprocessor=None, postprocessor=None, ignore_save=False, ignore_load=False):
    """
    Create a portfolio for each FactorPortfolio and/or for each
    list generated by a FactorPortfolioFactory

    Parameters:
        db_conn: the database connection to retrieve data from
        portfolio: a FactorPortfolio object describing a factor portfolio to create
        factory: a FactorPortfolioFactory object describing factor portfolios to create
        identifier: the stock identifier to use ("ticker", "sedol", "cusip")
        preprocessor: a FactorPortfolioPreprocessor to process stock level data before aggregation
        postprocessor: a FactorPortfolioPostprocessor to process factor portfolios after aggregation
        ignore_save: if true, will skip saving portfolios in the database
        ignore_load: if true, will skip loading portfolios from the database
    """
```

The database connection must be a psycopg2 connection able to read and write to the database connected to the pipeline. Either the portfolio parameter or the factory parameter or both can be used to specify what portfolios to aggregate using FactorPortfolio and FactorPortfolioFactory objects. The identifier parameter will only matter for preprocessors, since the stock identifiers will be aggregated out by the end of the function. The preprocessor and postprocessor arguments give you a chance to provide a FactorPortfolioPreprocessor or FactorPortfolioPostprocessor for any additional processing of the data needed. Finally, ignore_save and ignore_load allow you to skip saving or loading from the database if desired. See the corresponding sections for explanations of the other objects mentioned here.

Once finished, create_factor_portfolios will return a list of aggregated portfolios. A typical call to his function may look like as follows.

```
portfolios = drp.create_factor_portfolios(db_conn, factory=fp_factory,
                                          identifier='ticker', preprocessor=outlier_preprocessor)
```

Predictions

Overview

The Predictions is the step where the factor portfolios taken from Aggregation, and run the factor portfolios through the models created in the Predictions step. The models are created using the abstract based class Model.py

Model

The Model class is an abstract based class which is used to integrate the model one wishes to use into the Model class format.

```
class Model(ABC):
    """
    This is an abstract base class of the models
    Attributes:
        self:                the Model object
        factor_portfolio:    the factor portfolio this model will run predictions on
        training_params:     the parameters the model will run with in a dictionary
    """
```

These are the attributes of the abstract model class.

When defining the model one wants to use, the appropriate abstract functions should be extended. The two abstract functions that the user needs to be aware of is the training function and the __str__ function.

```
    @abstractmethod
    def training(self, factor_portfolio, training_params):
        pass
     """
     This is the function that will run the training on the model to get predictions

     Parameters:
         self:                    the Model object
         factor_portfolio:        the factor portfolio to run the training on
         training_params:         the training parameters used to run this model with
     """
```

The training function is where on the factor portfolio that is given, the model is run on with the training parameters to get the prediction data. In the extended class, this training function will need to be implemented by the user.

The __str__ function is simply the function to give the model a name to be kept track of.

When initializing a new model, the parameter needed is the factor portfolio on which the model is to be run on.

```
def set_parameter(self, param_name, param_value):
 """
 This is the function to set a parameter for the model.

 Parameters:
     self:                the Model object
     param_name:          the name of the parameter
     param_value:          the value of the parameter
 """
```

```
def set_parameters(self, new_training_params):
 """
 This is the function to set a group of parameters for the model.

 Parameters:
     self:                          the Model object
     new_training_params:           the group of parameters given in key: value dictionary
 """
```

There are two methods that can be used to set parameters to the model class. The set_parameter function is used to set one parameter at a time. The set_parameters is used to provide multiple parameters at once, in a key:value dictionary format.

The final important function is the do_training function. This is the function that will do the training, store the data into the database. When running the do_training, no other parameters are needed. The above mentioned functions are used with the initially stored variables to run the do_training.

Two additional functions are included to help visualize the different parameter in the factor portfolio. They are the display_model_ret, and display_metric().

So to run the entire Prediction step

1. First the appropriate model object is created with the factor portfolio wanted to run the predictions on.
2. Next, the set_parameters function is called on the object with the parameters in a list for with key:value
3. Call do_training() to get the predictions for that factor portfolio with the said model.

Stock Scoring

Overview

Stock Scoring is the final stage of the the pipeline, after predictions. In this stage, the predicted returns for factor portfolios are used to score all stocks in the buyable universe for desirability of expected returns. This is done in two stages. First, the predicted values are used to give a score of predictive power to each factor. These time-dependent scores of each factor are stored in a table called a factor policies table. Finally, these factor scores are used to score each individual stock at each time point to find a relative ranking of desirability of expected returns for each stock.

The DynamicRiskPremium package defines three abstract classes for this stage of the process: AbstractFactorPolicyGenerator, AbstractStockScorer, and AbstractStockScoringSystem. Each of these classes have opportunities for extending functionality of the pipeline, and a concrete implementation of all three is necessary for functioning scoring logic. The primary logic of the scoring stage is implemented by a single function within each of the first two classes, while the last class, AbstractStockScoringSystem, serves to wrap the creation of the the others and give a standard way to pass data from a group of aggregated factor portfolios with predicted returns to the factor policy generator and then to the stock scorer.

AbstractStockScoringSystem

```
class AbstractStockScoringSystem(ABC):

    """
    This is an abstract class made to allow implementation of a complete stock scoring system. It will make use of the abstract
    classes AbstractStockScorer and AbstractFactorPolicyGenerator.
    """
```

This class wraps the creation of a factor policy generator and a stock scorer while providing standard logic for passing data through the entire scoring system to get an output scores table. It has three functions: create_stock_scorer, create_factor_policy_generator, and do_scoring. This first two are abstract and must be implemented by extending classes. These implementations can and will often be trivial, calling constructors for the concrete implementations of the corresponding classes.

create_factor_policy_generator (self, factor_portfolio_list, data_desc):

```
@abstractmethod
def create_factor_policy_generator(self, factor_portfolio_list, data_desc):

    """
    This method must be overridden by subclasses of this class to create a factor policy object
    that can be used to score stocks with.

    Most of the time it will simply call the constructor of a class extending AbstractFactorPolicyGenerator
  Nathan-W-Schaffer, 7 days ago • Intial commit of stock scoring documentation
    Parameters:
        self :                  The AbstractStockScorer
        factor_portfolio_list :  List containing aggregated FactorPortfolio objects
        data_desc:  Dictionary with descriptors common to all aggregated factor portfolio statistics:
            period_start:   an integer in format YYYYMMDD for the starting period of all FactorPortfolios in fp_list
            period_end:     an integer in format YYYYMMDD for the ending period for all FactorPortfolios in fp_list
            period_type:    a string representing he period type ('W' or 'M') for all FactorPortfolios in fp_list
    """
```

This is an abstract function that needs to be overwritten in a subclass. The return for create_factor_policy_generator should be an instance of a class that extends AbstractFactorPolicyGenerator. If some data in the factor portfolios or the data_desc dictionary is needed in factor policy generation, it can be passed in here and saved to the factor policy generator in the constructor for use in generate_factor_policies.

create_stock_scorer (self, factor_portfolio_list, data_desc):

```
@abstractmethod
def create_stock_scorer(self, factor_portfolio_list, data_desc):

    """
    This method must be overridden by subclasses of this class to create a stock scorer object that
    can be used to score stocks with.

    Most of the time it will simply call the constructor of a class extending AbstractStockScorer

    Parameters:
        self :                  The AbstractStockScorer
        factor_portfolio_list :  List containing aggregated FactorPortfolio objects
        data_desc:  Dictionary with descriptors common to all aggregated factor portfolio statistics:
            period_start:   an integer in format YYYYMMDD for the starting period of all FactorPortfolios in fp_list
            period_end:     an integer in format YYYYMMDD for the ending period for all FactorPortfolios in fp_list
            period_type:    a string representing he period type ('W' or 'M') for all FactorPortfolios in fp_list
    """
```

This is an abstract function that needs to be overwritten in a subclass. The return for create_stock_scorer should be an instance of a class that extends AbstractStockScorer. If some data in the data_desc dictionary is needed for stock scoring, it can be passed in here and saved to the stock scorer in the constructor for use in score_stocks.

do_scoring (self, conn, factor_portfolio_list, data_desc):

```
def do_scoring(self, conn, factor_portfolio_list, data_desc):
    """
    This meathod meant to score stocks using the AbstractStockScorer and the AbstractFactorPolicyGenerator. This is
    an example of the complete process of the stock scoring steps.

    Parameters:
        self :                  The AbstractStockScorer
        conn :                  A connection to the database containing raw stock data
        factor_portfolio_list : List containing aggregated FactorPortfolio objects
        data_desc:              Dictionary with descriptors common to all aggregated factor portfolio statistics:
            period_start:   an integer in format YYYYMMDD for the starting period of all FactorPortfolios in fp_list
            period_end:     an integer in format YYYYMMDD for the ending period for all FactorPortfolios in fp_list
            period_type:    a string representing he period type ('W' or 'M') for all FactorPortfolios in fp_list
    """
```

This function has logic already implemented and will take as inputs a connection to the stock database (conn), a list of aggregated FactorPortfolio objects with predicted future returns (factor_portfolio_list), and a dictionary to describe some common attributes of the FactorPorfolios passed to it (data_desc). The FactorPortfolio objects must all have the same start date, end date, an period type, and those three pieces of information should be passed in data_desc as described in the in-code documentation. This function takes care of compiling predicted returns from all FactorPortfolio objects into a factor portfolio statistics table, passing that to the factor policy generator, passing the factor policies table to the stock scorer, and storing both the factor policies table and stock scores back to the database

This is an example of implementation of the entire stock scoring system:

```
class ExampleStockScoringSystem(drp.AbstractStockScoringSystem):

    def create_stock_scorer(self, factor_portfolio_list, data_desc):
        # Using concrete stock scorer implementation from DRP package
        return drp.WeightedAverageStockScorer()

    def create_factor_policy_generator(self, factor_portfolio_list, data_desc):
        return ExampleFPGenerator(factor_portfolio_list, data_desc)
```

AbstractFactorPolicyGenerator

```
class AbstractFactorPolicyGenerator(ABC):
    """
    This class is dedicated to the creation of a factor policy.
    """
```

This abstract class is dedicated to generating factor policies. These are tables which essentially score the predictive power of each factor at each time point. There is just one abstract method, generate_factor_policies, to be implemented in a subclass.

generate_factor_policies (self, fpst):

```
@abstractmethod
def generate_factor_policies(self, fpst):

    """
    This method must be overridden by a subclass of this class to generate factor policies. It takes data passed from
    the modeling stage as a factor portfolio statistics table in the form a pandas dataframe. The ouput of this method
    should provide a factor polciy table or weight table to a AbstractStockScorer class.

    Parameters:
        self : The AbstractStockScorer
        fpst : Factor Portfolio Statistics Table
    """

    pass
```

This method takes in a single pandas dataframe, referred to as a factor portfolio statistics table, or FPST for short. It must return a dataframe for the factor policies table, containing weights for every factor described by the given factor portfolios. It is quite possible the implementer of this class will want to use helper methods in their concrete implementation to help with, for example, the creation of a covariance table, which they are free to do. See the Factor Portfolio Statistics Table section for more details about the characteristics of this dataframe.

Factor Portfolio Statistics Table

This dataframe will be aggregated at the beginning of do_scoring from the pred_data dataframes in each of the FactorPortfolio objects passed to it. The index column should look identical to that of the data and pred_data dataframes in the factor portfolio objects. It will have an index labelled ‘date’ with integer values representing dates from the start date to the end date listed in the data_desc dictionary passed to do_scoring in the stock scoring system. The columns will be taken directly from each pred_data dataframe. Their column names can be expected to start with the name of a prime factor for which the predictions describe, but can contain more information afterwards, separated by a space. Though there is likely to be multiple statistics about each factor, the uniqueness of these column names is enforced in AbstractStockScoringSystem.do_scoring to aid with effective factor policy generation.

Example of an FPST from two FactorPortfolios with prime factors of mcap and sales_g:

```
Factor Portfolio Statistics Table:
          mcap pred_Mean_1m_Abs_ret  sales_g pred_Mean_1m_Abs_ret
date
20150821                   0.066887                     2.509556
20150828                  -1.762554                    -7.881269
20150904                   2.770887                    -6.432201
20150911                   2.989621                    -2.657597
20150918                   4.558844                    -6.908095
20150925                   8.674122                    -0.847489
20151002                   7.881352                    -0.392585
...                             ...                          ...
20151113                   0.162449                    -8.072539
20151120                  -2.720093                    -4.369131
20151127                  -0.066490                     1.307963
20151204                  -1.494904                    -0.982530
20151211                  -3.796095                    -6.359533
20151218                  -5.951692                   -15.641436
20151225                  -6.107696                   -16.905325
```

Factor Policies Table

This table is the output of the factor policy generator and one input to the stock scorer. Its index column will be like that of the fpst, with a label of ‘date’ and integer values representing dates from the start date to the end date listed in the data_desc dictionary passed to do_scoring in the stock scoring system. Column names will match the prime factors of the factor portfolios from which this table was generated. The values in these columns will be floating point numbers summing to 1.0 for any given row.

Example of a factor policies table for the factors mcap and sales_g:

```
Factor Policies Table:
              mcap   sales_g
date
20150821  0.076867  0.923133
20150828  0.008329  0.991671
20150904  0.321518  0.678482
20150911  0.319538  0.680462
20150918  0.317480  0.682520
20150925  0.648041  0.351959
20151002  0.667714  0.332286
...            ...       ...
20151113  0.113472  0.886528
20151120  0.155723  0.844277
20151127  0.318131  0.681869
20151204  0.342298  0.657702
20151211  0.248479  0.751521
20151218  0.144216  0.855784
20151225  0.091027  0.908973
```

AbstractStockScorer

```
class AbstractStockScorer(ABC):
    """
    This is the abstract base class capable of scoring stocks based on assigned weights from a given factor weights table.
    """
```

This class uses factor-level scores defined by a factor policies table, along with stock data queried from the database to generate stock-level scores for relative desirability of expected returns. This class has two methods, query_stock_data, and score_stocks. This first has already been implemented and creates a dataframe with percentile ranks of each stock for each factor in the factor portfolio list. The second is abstract and must be implemented in a subclass. The final result of this class will be a dataframe containing scores for every stock in the stock universe at every time point requested. This will be the final result of the entire DRP pipeline.

query_stock_data (conn, fp_list, data_desc):

```
def query_stock_data(self, conn, fp_list, data_desc):

    """
    This meathod is for querying the entire universe of stocks baised for the given factor portfolios and data_desc parameters.
    This will provide data in the form of a pandas dataframe to the score_stocks abstract method in order to produce scores.

    Parameters:
        self :      The AbstractStockScorer
        conn :      A connection to the database containing raw stock data
        fp_list:    List containing aggregated FactorPortfolio objects
        data_desc:  Dictionary with descriptors common to all aggregated factor portfolio statistics:
            period_start:   an integer in format YYYYMMDD for the starting period of all FactorPortfolios in fp_list
            period_end:     an integer in format YYYYMMDD for the ending period for all FactorPortfolios in fp_list
            period_type:    a string representing he period type ('W' or 'M') for all FactorPortfolios in fp_list
    """
```

This method is already implemented and will take as inputs a connection to the stock database (conn), a list of aggregated FactorPortfolio objects (fp_list), and a dictionary to describe some common attributes of the FactorPorfolios passed to it (data_desc). It will query the entire universe of stock level data and produce a dataframe containing a column of percentile ranks for each of the prime factors in the factor portfolios within fp_list. See the Stock Level Factor Data section for more details about the characteristics of this dataframe.

Stock Level Factor Data

This dataframe is one of two dataframes that are passed to score_stocks in the stock scorer. It has a double-level index, with labels of ‘date’ and ‘ticker’. For every row in the factor policies table and every stock in the database at that time point, there should be a row in the factor data table. The other columns will correspond to one of the prime factors in the factor portfolio list passed to query_stock_data and will be named with that factor name followed by ‘_rank’. The values in these columns will be floating point numbers from 0.0 to 1.0, inclusive, describing the percentile ranking of the given company for the given factor at that given time point (1.0 always meaning the highest value in that factor)

Below is an example of the output from this method with fp_list containing factor portfolios for mcap and sales_g and weekly data from August 21, 2015 to December 25, 2015.

```
Stock Level Factor Data:
                mcap_rank  sales_g_rank
date     ticker
20150821 A       0.654070      0.446705
         AA      0.636628      0.626938
         AABA    0.861434      0.536822
         AAL     0.830426      0.843023
         AAN     0.088178      0.912791
         AAP     0.674419      0.916667
         AAPL    1.000000      0.890504
...                   ...           ...
20151225 ZAYO    0.445305      0.869313
         ZBH     0.781220      0.788964
         ZBRA    0.218780      0.995160
         ZG      0.025169      0.983543
         ZION    0.383349      0.324298
         ZNGA    0.066796      0.825750
         ZTS     0.809293      0.408519
```

score_stocks(fp_list, fwt, data):

```
@abstractmethod
def score_stocks(self, fp_list, fwt, data):

    """
    This method must be overridden by a subclass of this class to score stocks. It takes data from two pandas dataframes
    fp_list and fwt, then should append and compute a score coloumb to the fp_list.

    Parameters:
        self :      The AbstractStockScorer
        fp_list :   List containing aggregated FactorPortfolio objects
        fwt :       Factor weights table containing weights for each time period and factor (a.k.a. Factor Policies Table)
        data_desc:  Dictionary with descriptors common to all aggregated factor portfolio statistics:
            period_start:   an integer in format YYYYMMDD for the starting period of all FactorPortfolios in fp_list
            period_end:     an integer in format YYYYMMDD for the ending period for all FactorPortfolios in fp_list
        period_type:    a string representing he period type ('W' or 'M') for all FactorPortfolios in fp_list
    """

    pass
```

This method must be overwritten by a subclass of AbstractStockScorer. It takes as inputs the list of factor portfolios (fp_list), a factor policies dataframe (fwt), which is also called a factor weights table here, and a stock-level factor data dataframe (data). It must generate a dataframe with stock scores as described in the Stock Scores Table section. The output score value for each ticker and time period should be between 0.0 and 1.0, inclusive, with higher values being indicative of more desirable returns. It is important to consider in the implementation of this function that some factors are ‘positive’ or good factors while others are ‘negative’ or bad. This information can be gotten from the FactorPortfolio objects in fp_list by accessing the ‘positive’ attribute and should be taken into account when scoring stocks. For more information about the characteristics of the factor policies dataframe, see the Factor Policies Table section of the FactorPolicyGenerator description.

Stock Scores Table

This dataframe contains scores for every stock in the buyable stock universe at every time point requested and will be the final result of the entire DRP pipeline. The index of this table will be identical to that of the stock-level factor data table, with a level for dates and a level for tickers. There will be one non-index column called ‘score’ with floating point values from 0.0 to 1.0, inclusive. These represent how a given stock ranks against other stocks at that time point for desirable expected returns, where higher values are better.

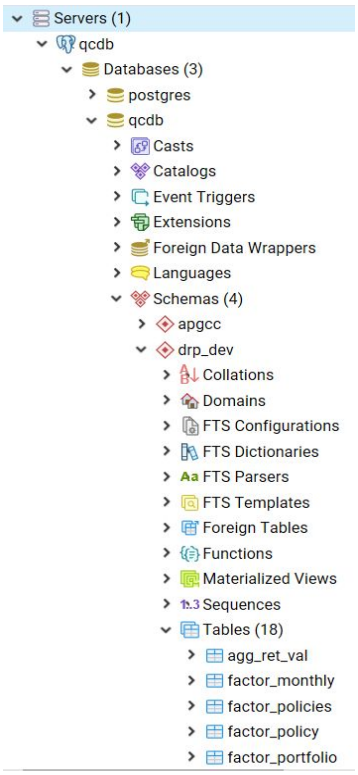
Here is an example of a stock scores dataframe:

```
Scores:
                    score
date     ticker
20150821 A       0.438959
         AA      0.606679
         AABA    0.506209
         AAL     0.791258
         AAN     0.912716
         AAP     0.871232
         AAPL    0.822054
...                   ...
20151225 ZAYO    0.840674
         ZBH     0.737062
         ZBRA    0.975685
         ZG      0.982750
         ZION    0.350910
         ZNGA    0.835531
         ZTS     0.388692
```

WeightedAverageStockScorer

This class is a concrete implementation of AbstractStockScorer which is fully implemented and included with the DRP package. The implemented class will multiply each factor value for each ticker and time period by the matching factor weight table for that time period. For factors that are ‘negative’ it multiplies the weight in the factor policy for that time point by 1.0 minus the rank for that factor to get a score for each stock. By doing this, stocks with lower values of ‘negative’ factors will receive higher scores.

Database



The database we are currently using is a postgreSQL database located on an EC2 instance of an AWS (Amazon Web Services) server. The architecture of the database is visible to the right. All of the tables, views and other tools used for the purposes of the Dynamic Risk Premium 2.0 project are located within the drp_dev schema.

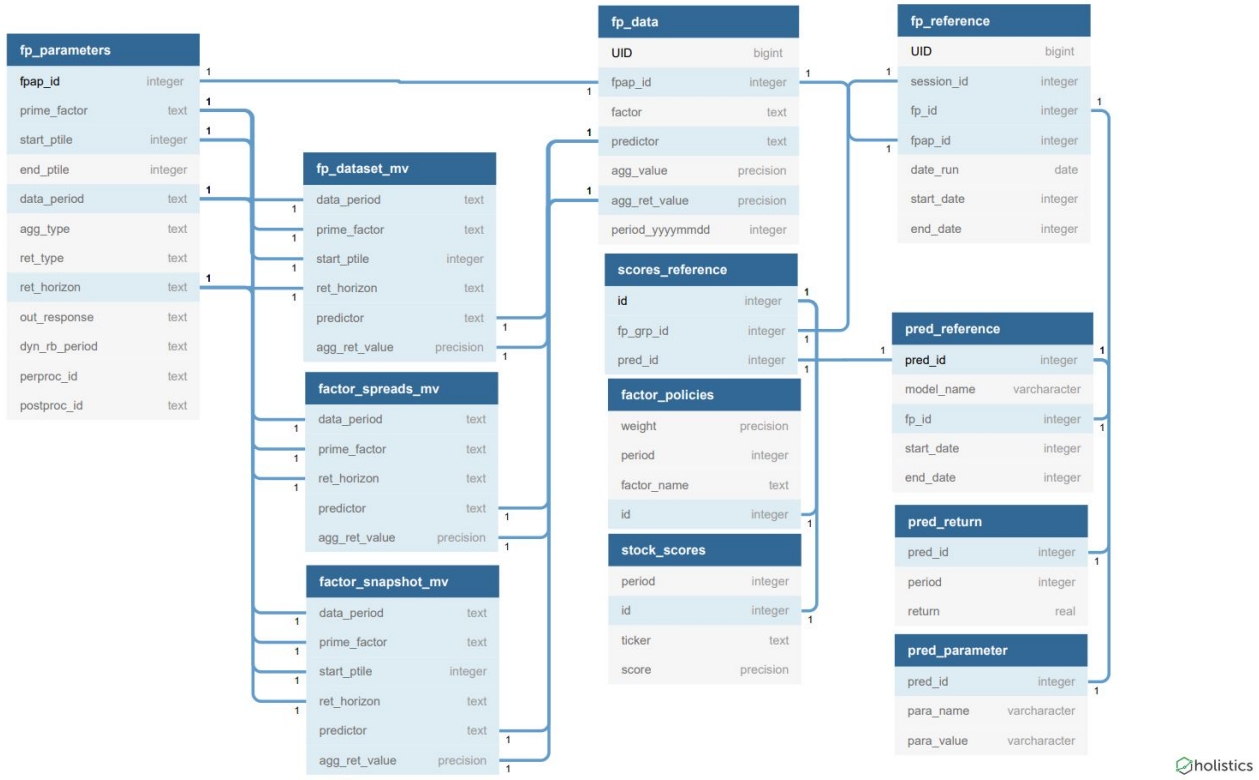
There are two main ways to access the database. The first method is to use pgAdmin4. This is a free software with an intuitive user interface that can be used to edit schemas, add/drop tables, and view diagnostic information. This software is highly recommended for administrator use. The second method of accessing the database is directly through a python script running on the server. This method allows easy read and write access to the data within the database for use within a script. The simplest way to do this is to make sure psycopg is installed on the server and then import it into the script. Below is an example of accessing the database from a script as well as the host address and port number needed to access the database. Your username and password may vary, as they are administered by the Principal team.

```
import psycopg2

if __name__ == "__main__":

    conn = psycopg2.connect(host="qc-aurora-cluster.cluster-coh4objazhte.us-east-1.rds.amazonaws.com",
                            database='qcdb', user='Capstone_User', password='5ZpMmX!jn0h3', port="5432")
```

Host Address: qc-aurora-cluster.cluster-coh4objazhte.us-east-1.rds.amazonaws.com

Port: 5432

Schema



Tables

The tables used for the Quant Modeling Library are all located within the drp_dev schema. Within this schema there are six tables; fp_parameters, fp_data, and fp_reference are used to store factor portfolio data, model_output table is used to store the output of the modeling step, and stock scores, factor_policies, and scores_reference are used to store the data created by the stock scoring step. The table shown above displays the relationship between these tables as well as the materialized views that will be discussed later. Descriptions of each table are located below.

fp_parameters

The factor parameters table is used to store the parameters used on a particular run of the DRP 2.0 pipeline. This allows the same set of parameters to be used multiple times, as well as to identify what parameters were used to create the given data in a particular run. It has a self incrementing primary key called ‘fpap_id’ as an identifier. The rest of the columns in this table

simple refer to the parameters of the factor portfolio. These are columns all have descriptive names to help with ease of understanding.

Connections:

fp_parameters.fpap_id	→	fp_data.fpap_id
fp_parameters.data_period, fp_parameters.prime_factor, fp_parameters.ret_horizon,	→	fp_dataset_mv, factor_spreads_mv, factor_snapshot_mv
fp_parameters.start_ptile	→	fp_dataset_mv, Factor_snapshot_mv

	fpap_id [PK] integer	prime_factor text	start_ptile integer	end_ptile integer	data_period text	agg_type text	ret_type text	ret_horizon text	out_response text	dyn_rb_period text	preproc_id text	postproc_id text
1	1	bk_p	78	100	M	static	absolute	1m	mean	1w	testpre	testpst
2	2	bk_p	80	100	W	static	absolute	1w	Mean	1w	testpre	testpost
3	4	happiness	98	100	W	Static	Absolute	1m	mean	1w	test	test
4	12	prev_1m_ret	98	100	W	Static	Absolute	1m	Mean	[null]	[null]	[null]
5	14	bk_p	98	100	W	Static	Absolute	1m	Mean	[null]	[null]	[null]

Image: fp_parameters example

fp_reference

The factor portfolio reference table is used to keep track of all of the moving parts throughout the DRP 2.0 pipeline. It allows for easier tracking of the data created in the pipeline by implementing a session ID to track the factor portfolios created within the same session. It also includes a factor portfolio ID and factor parameters ID which are then used to reference the corresponding rows in the fp_data and fp_parameters columns respectively. The start and end date columns refer to the dates over which the factor portfolio is being considered.

Connections:

fp_reference.session_id	→	scores_reference.fp_grp_id
fp_reference.fpap_id	→	fp_data.fpap_id
fp _ reference.fp_id	→	model_id.fp_id

	UID [PK] bigint	session_id integer	fp_id integer	fpap_id integer	date_run date	start_date integer	end_date integer
1	1	[null]	[null]	[null]	[null]	[null]	[null]
2	2	1	23	12	2019-03-12	19980101	20000101
3	3	1	23	12	2019-03-12	19980101	20000101
4	4	1	23	12	2019-03-12	19980101	20000101
5	9	5	28	18	[null]	19980101	20000101

Image: fp_reference example

Note: Currently the start_date and end_date columns are stored as integers in the yyyymmdd format. Ideally this will be converted to the date format used in the date_run column.

fp_data

The factor portfolio data table is used to handle all of the aggregated data created by a given factor portfolio and parameters combination. The aggregated value as well as the aggregated return values are given as double precision values.

Connections:

Fp_data.fpap_id	→	fp_paremeters.fpap_id, fp_reference.fpap_id
Fp_data.predictor, Fp_data.agg_ret_value	→	fp_dataset_mv, factor_spreads_mv, factor_snapshot_mv

	fpap_id integer	factor text	predictor text	agg_value double precision	agg_ret_value double precision	period_yyyymmdd integer	UID [PK] bigint
1	4	happin...	success	123.45	[null]	[null]	1
2	18	prev_1...	mean	-3.34841485	1.1167399	19980102	2
3	18	prev_1...	skew	0.76433365818431	1.1167399	19980102	3
4	18	prev_3...	mean	-10.0040557	1.1167399	19980102	4
5	18	prev_3...	skew	0.160069919453434	1.1167399	19980102	5

Image: fp_data example

Note: The period column is stored as an integer, but ideally would converted to a date type column.

pred_reference

The pred_reference table is the table used to store the reference data created by the prediction stage of the pipeline. This table allows the user to see which models were used on which factor portfolios.

Connections:

pred_reference.pred_id → scores_reference.pred_id
pred_return.pred_id

pred_reference.fp_id → fp_reference.fp_id

	pred_id [PK] integer	model_name character varying	fp_id integer	start_date integer	end_date integer
38	38	Empty_Model	382	20150821	20151231
39	39	Empty_Model	384	20150821	20151231
40	40	Empty_Model	383	20150821	20151231
41	41	Empty_Model	385	20150821	20151231
42	42	Empty_Model	382	20150821	20151231

Image: pred_reference example

pred_return

The pred return table shows the predictions returned for each factor portfolio for each of the date time period.

Connections:

pred_return.pred_id → pred_reference.pred_id

	pred_id integer	period integer	return real
78	30	20150828	-1.76255
79	30	20150904	2.77089
80	30	20150911	2.98962
81	30	20150918	4.55884
82	30	20150925	8.67412
83	30	20151002	7.88135

Image: pred_return example

pred_parameter

The pred_parameter table shows for each prediction, what parameters were used to get the predictions from model. The pred_id is from which predictions these parameters were given too.

	pred_id integer	para_name character varying	para_value character varying
1	22	window	rolling
2	22	train_len	5
3	22	buffer_len	5
4	22	test_len	5

Image: pred_parameter example

Connections:

pred_parameter.pred_id → pred_reference.pred_id

scores_reference

The scores_reference table easily shows which score id is related to which predictions and belongs to which session the factor portfolios were aggregated from.

Connections:

scores_reference.id → factor_policies.id, stock_scores.id

scores_reference.fp_grp_id → fp_reference.session_id

scores_reference.pred_id → model_output.model_id

	id [PK] integer	fp_grp_id integer	pred_id integer
1	4	475	42
2	3	474	38
3	2	473	34

Image: score_reference example

stock_scores

The stock scores table stores some of the data found during the scoring stage of the pipeline. It holds the scoring value associated with the id and its period. The score is given as a double precision. The id is from the stock_scores.id, telling from which score id, these score come from

Connections:
stock_scores.id → stock_references.id

	period integer	id integer	ticker text	score double precision
1	20150821	3	A	0.438959187237681
2	20150821	3	AA	0.606678589877443
3	20150821	3	AABA	0.506209164292982
4	20150821	3	AAL	0.791257523240485
5	20150821	3	AAN	0.912716214605968
6	20150821	3	AAP	0.871231994911536
7	20150821	3	AAPL	0.822053936062491
8	20150821	3	ABBV	0.638900866372998

Image: stock_scores example

factor_policies

The factor policies table stores the weight of the factor, its associated reference id, and its period. For each period with the same id, the weights will add up to 1.

Connections:
factor_policies.id → stock_references.id

	weight double precision	period integer	factor_name text	id integer
16	0.692862515112597	20151009	mcap	3
17	0.307137484887403	20151009	sales_g	3
18	0.38130888295639	20151016	mcap	3
19	0.61869111704361	20151016	sales_g	3
20	0.334109739141673	20151023	mcap	3
21	0.665890260858327	20151023	sales_g	3
22	0.252765948890463	20151030	mcap	3

Materialized Views

The three materialized views are denoted with “mv” at the end of the table name. Fp_dataset_mv and factor_snapshot_mv get the same information from fp_data and fp_parameters but under different conditions. When fp_data.factor is fut_1m_ret, fut_3m_ret, fut_6m_ret, fut_9m_ret, or fut_12m_ret, fp_dataset_mv retrieves the information needed.

	data_period text	prime_factor text	start_ptile integer	ret_horizon text	predictor text	agg_ret_value double precision
1	M	bk_p	78	1m	mean	10

Image: fp_dataset_mv data example

Note: The materialized view data examples were made from manual inputs into the database. They are not meant to be 100% accurate and true information, they are only meant to show how the data appears within the views.

When fp_data.factor is prev_1m_ret, prev_3m_ret, prev_6m_ret, prev_9m_ret, or prev_12m_ret, factor_snapshot_mv and factor_spreads_mv retrieve the information they need.

	data_period text	prime_factor text	start_ptile integer	ret_horizon text	predictor text	agg_ret_value double precision
1	W	bk_p	98	1m	mean	1.1167399
2	W	bk_p	98	1m	skew	1.1167399
3	W	bk_p	98	1m	mean	1.1167399
4	W	bk_p	98	1m	skew	1.1167399
5	W	bk_p	98	1m	mean	1.1167399
6	W	bk_p	98	1m	skew	1.1167399
7	W	bk_p	98	1m	mean	7.9674761
8	W	bk_p	98	1m	skew	7.9674761
9	W	bk_p	98	1m	mean	7.9674761
10	W	bk_p	98	1m	skew	7.9674761
11	W	bk_p	98	1m	mean	7.9674761
12	W	bk_p	98	1m	skew	7.9674761
13	W	bk_p	98	1m	mean	5.6569117
14	W	bk_p	98	1m	skew	5.6569117
15	W	bk_p	98	1m	mean	5.6569117

Image: factor_snapshot_mv data example

Rows 1 - 2 are being repeated in rows 3 - 4 and 5 - 6 because it is getting the information for different fp_data.factor values. In this data example, rows 1 - 2 are grabbed when fp_data.factor = prev_1m_ret, 3 - 4 are grabbed when fp_data.factor = prev_3m_ret, and 5 - 6 are grabbed when fp_data.factor = prev_6m_ret.

	data_period text	prime_factor text	ret_horizon text	predictor text	agg_ret_value double precision
1	W	bk_p	1m	mean	1.1167399
2	W	bk_p	1m	skew	1.1167399
3	W	bk_p	1m	mean	1.1167399
4	W	bk_p	1m	skew	1.1167399
5	W	bk_p	1m	mean	1.1167399
6	W	bk_p	1m	skew	1.1167399
7	W	bk_p	1m	mean	7.9674761
8	W	bk_p	1m	skew	7.9674761
9	W	bk_p	1m	mean	7.9674761
10	W	bk_p	1m	skew	7.9674761
11	W	bk_p	1m	mean	7.9674761
12	W	bk_p	1m	skew	7.9674761
13	W	bk_p	1m	mean	5.6569117
14	W	bk_p	1m	skew	5.6569117
15	W	bk_p	1m	mean	5.6569117

Image: factor_spreads_mv data example

Note: Factor_spreads_mv currently contains all of the information of factor_snapshot_mv (with the exception of start_ptile) because of time constraints. Factor_spreads_mv was unable to be fully implemented, this image is a display of its current state.

The materialized views do not have a trigger function to automatically refresh the data, so in order to view the newest data, they will have to be manual refreshed.