project_8_starter

April 24, 2023

1 Project 8: Backtesting

In this project, you will build a fairly realistic backtester that uses the Barra data. The backtester will perform portfolio optimization that includes transaction costs, and you'll implement it with computational efficiency in mind, to allow for a reasonably fast backtest. You'll also use performance attribution to identify the major drivers of your portfolio's profit-and-loss (PnL). You will have the option to modify and customize the backtest as well.

1.1 Instructions

Each problem consists of a function to implement and instructions on how to implement the function. The parts of the function that need to be implemented are marked with a `# TODO` comment. Your code will be checked for the correct solution when you submit it to Udacity.

1.2 Packages

When you implement the functions, you'll only need to you use the packages you've used in the classroom, like Pandas and Numpy. These packages will be imported for you. We recommend you don't add any import statements, otherwise the grader might not be able to run your code.

1.2.1 Install Packages

```
In [1]: !pip install --upgrade setuptools
```

```
Collecting setuptools
  Downloading https://files.pythonhosted.org/packages/b0/3a/88b210db68e56854d0bcf4b38e165e03be37
    100% || 962kB 14.1MB/s ta 0:00:01
Installing collected packages: setuptools
  Found existing installation: setuptools 38.4.0
    Uninstalling setuptools-38.4.0:
      Successfully uninstalled setuptools-38.4.0
Successfully installed setuptools-59.6.0
```

```
In [2]: import sys
        !{sys.executable} -m pip install -r requirements.txt
```

```
Requirement already satisfied: matplotlib==2.1.0 in /opt/conda/lib/python3.6/site-packages (from
Collecting numpy==1.16.1 (from -r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/f5/bf/4981bcbee43934f0adb8f764a1e70ab0ee5a
    100% || 17.3MB 1.8MB/s eta 0:00:01
Collecting pandas==0.24.1 (from -r requirements.txt (line 3))
  Downloading https://files.pythonhosted.org/packages/e6/de/a0d3defd8f338eaf53ef716e40ef6d6c277c
    100% || 10.1MB 4.4MB/s eta 0:00:01
Collecting patsy==0.5.1 (from -r requirements.txt (line 4))
  Downloading https://files.pythonhosted.org/packages/ea/0c/5f61f1a3d4385d6bf83b83ea495068857ff8
    100% || 235kB 19.6MB/s ta 0:00:01
Collecting scipy==0.19.1 (from -r requirements.txt (line 5))
  Downloading https://files.pythonhosted.org/packages/0e/46/da8d7166102d29695330f7c0b912955498854
    100% || 48.2MB 646kB/s eta 0:00:01    67% |          | 32.4MB 45.2MB/s eta 0:00:01
Collecting statsmodels==0.9.0 (from -r requirements.txt (line 6))
  Downloading https://files.pythonhosted.org/packages/85/d1/69ee7e757f657e7f527cbf500ec2d295396e
    100% || 7.4MB 8.6MB/s eta 0:00:01
Collecting tqdm==4.19.5 (from -r requirements.txt (line 7))
  Downloading https://files.pythonhosted.org/packages/71/3c/341b4fa23cb3abc335207dba057c790f3bb3
    100% || 61kB 16.2MB/s ta 0:00:01
Requirement already satisfied: six>=1.10 in /opt/conda/lib/python3.6/site-packages (from matplot
Requirement already satisfied: python-dateutil>=2.0 in /opt/conda/lib/python3.6/site-packages (f
Requirement already satisfied: pytz in /opt/conda/lib/python3.6/site-packages (from matplotlib==
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.6/site-packages/cycler-0.1
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /opt/conda/lib/python
tensorflow 1.3.0 requires tensorflow-tensorboard<0.2.0,>=0.1.0, which is not installed.
moviepy 0.2.3.2 has requirement tqdm==4.11.2, but you'll have tqdm 4.19.5 which is incompatible.
Installing collected packages: numpy, pandas, patsy, scipy, statsmodels, tqdm
  Found existing installation: numpy 1.12.1
    Uninstalling numpy-1.12.1:
      Successfully uninstalled numpy-1.12.1
  Found existing installation: pandas 0.23.3
    Uninstalling pandas-0.23.3:
      Successfully uninstalled pandas-0.23.3
  Found existing installation: patsy 0.4.1
    Uninstalling patsy-0.4.1:
      Successfully uninstalled patsy-0.4.1
  Found existing installation: scipy 1.2.1
    Uninstalling scipy-1.2.1:
      Successfully uninstalled scipy-1.2.1
  Found existing installation: statsmodels 0.8.0
    Uninstalling statsmodels-0.8.0:
      Successfully uninstalled statsmodels-0.8.0
  Found existing installation: tqdm 4.11.2
    Uninstalling tqdm-4.11.2:
      Successfully uninstalled tqdm-4.11.2
Successfully installed numpy-1.16.1 pandas-0.24.1 patsy-0.5.1 scipy-0.19.1 statsmodels-0.9.0 tqd
```

1.2.2 Load Packages

```
In [3]: import scipy
        import patsy
        import pickle

        import numpy as np
        import pandas as pd

        import scipy.sparse
        import matplotlib.pyplot as plt

        from statistics import median
        from scipy.stats import gaussian_kde
        from statsmodels.formula.api import ols
        from tqdm import tqdm
```

1.3 Load Data

We'll be using the Barra dataset to get factors that can be used to predict risk. Loading and parsing the raw Barra data can be a very slow process that can significantly slow down your backtesting. For this reason, it's important to pre-process the data beforehand. For your convenience, the Barra data has already been pre-processed for you and saved into pickle files. You will load the Barra data from these pickle files.

In the code below, we start by loading 2004 factor data from the `pandas-frames.2004.pickle` file. We also load the 2003 and 2004 covariance data from the `covaraince.2003.pickle` and `covaraince.2004.pickle` files. You are encouraged to customize the data range for your backtest. For example, we recommend starting with two or three years of factor data. Remember that the covariance data should include all the years that you choose for the factor data, and also one year earlier. For example, in the code below we are using 2004 factor data, therefore, we must include 2004 in our covariance data, but also the previous year, 2003. If you don't remember why must include this previous year, feel free to review the lessons.

```
In [4]: barra_dir = '../../data/project_8_barra/'

        data = {}
        for year in [2004]:
            fil = barra_dir + "pandas-frames." + str(year) + ".pickle"
            data.update(pickle.load( open( fil, "rb" ) ))

        covariance = {}
        for year in [2003, 2004]:
            fil = barra_dir + "covariance." + str(year) + ".pickle"
            covariance.update(pickle.load( open(fil, "rb" ) ))

        daily_return = {}
        for year in [2004, 2005]:
            fil = barra_dir + "price." + str(year) + ".pickle"
            daily_return.update(pickle.load( open(fil, "rb" ) ))
```

1.4 Shift Daily Returns Data (TODO)

In the cell below, we want to incorporate a realistic time delay that exists in live trading, we'll use a two day delay for the `daily_return` data. That means the `daily_return` should be two days after the data in `data` and `cov_data`. Combine `daily_return` and `data` together in a dict called `frames`.

Since reporting of PnL is usually for the date of the returns, make sure to use the two day delay dates (dates that match the `daily_return`) when building `frames`. This means calling `frames['20040108']` will get you the prices from "20040108" and the data from `data` at "20040106".

Note: We're not shifting `covariance`, since we'll use the "DataDate" field in `frames` to lookup the covariance data. The "DataDate" field contains the date when the `data` in `frames` was recorded. For example, `frames['20040108']` will give you a value of "20040106" for the field "DataDate".

```
In [5]: frames ={}
        dlyreturn_n_days_delay = 2

        # TODO: Implement
        N = len(data)
        shifted = zip(sorted(data.keys()), sorted(daily_return.keys())[dlyreturn_n_days_delay:N
        for d, p in shifted: # date in data and date in price
            frames[p] = data[d].merge(daily_return[p], on = 'Barrid')
```

1.5 Add Daily Returns date column (Optional)

Name the column `DlyReturnDate`. **Hint**: create a list containing copies of the date, then create a pandas series.

```
In [ ]: # Optional
```

1.6 Winsorize

As we have done in other projects, we'll want to avoid extremely positive or negative values in our data. Will therefore create a function, `wins`, that will clip our values to a minimum and maximum range. This process is called **Winsorizing**. Remember that this helps us handle noise, which may otherwise cause unusually large positions.

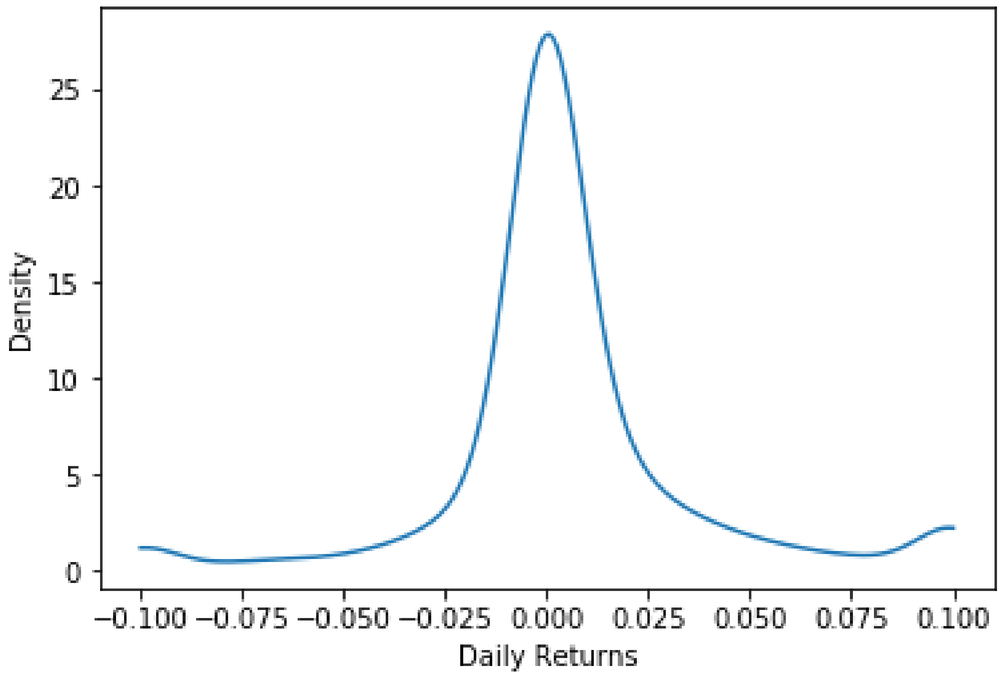
```
In [6]: def wins(x,a,b):
            return np.where(x <= a,a, np.where(x >= b, b, x))
```

1.7 Density Plot

Let's check our `wins` function by taking a look at the distribution of returns for a single day `20040102`. We will clip our data from `-0.1` to `0.1` and plot it using our `density_plot` function.

```
In [7]: def density_plot(data):
            density = gaussian_kde(data)
            xs = np.linspace(np.min(data),np.max(data),200)
            density.covariance_factor = lambda : .25
            density._compute_covariance()
```

```
        plt.plot(xs,density(xs))
        plt.xlabel('Daily Returns')
        plt.ylabel('Density')
        plt.show()

    test = frames['20040108']
    test['DlyReturn'] = wins(test['DlyReturn'],-0.1,0.1)
    density_plot(test['DlyReturn'])
```



1.8 Factor Exposures and Factor Returns

Recall that:

$r_{i,t} = \sum_{j=1}^{k} (\beta_{i,j,t-2} \times f_{j,t})$
where $i = 1...N$ (N assets),
and $j = 1...k$ (k factors).

where $r_{i,t}$ is the return, $\beta_{i,j,t-2}$ is the factor exposure, and $f_{j,t}$ is the factor return. Since we get the factor exposures from the Barra data, and we know the returns, it is possible to estimate the factor returns. In this notebook, we will use the Ordinary Least Squares (OLS) method to estimate the factor exposures, $f_{j,t}$, by using $\beta_{i,j,t-2}$ as the independent variable, and $r_{i,t}$ as the dependent variable.

```
In [8]: def get_formula(factors, Y):
            L = ["0"]
            L.extend(factors)
            return Y + " ~ " + " + ".join(L)
```

```
        def factors_from_names(n):
            return list(filter(lambda x: "USFASTD_" in x, n))

        def estimate_factor_returns(df):
            ## build universe based on filters
            estu = df.loc[df.IssuerMarketCap > 1e9].copy(deep=True)

            ## winsorize returns for fitting
            estu['DlyReturn'] = wins(estu['DlyReturn'], -0.25, 0.25)

            all_factors = factors_from_names(list(df))
            form = get_formula(all_factors, "DlyReturn")
            model = ols(form, data=estu)
            results = model.fit()
            return results
```

```
In [10]: facret = {}

         for date in frames:
             facret[date] = estimate_factor_returns(frames[date]).params
```

```
In [11]: my_dates = sorted(list(map(lambda date: pd.to_datetime(date, format='%Y%m%d'), frames.k
```

1.9 Choose Alpha Factors

We will now choose our alpha factors. Barra's factors include some alpha factors that we have seen before, such as:

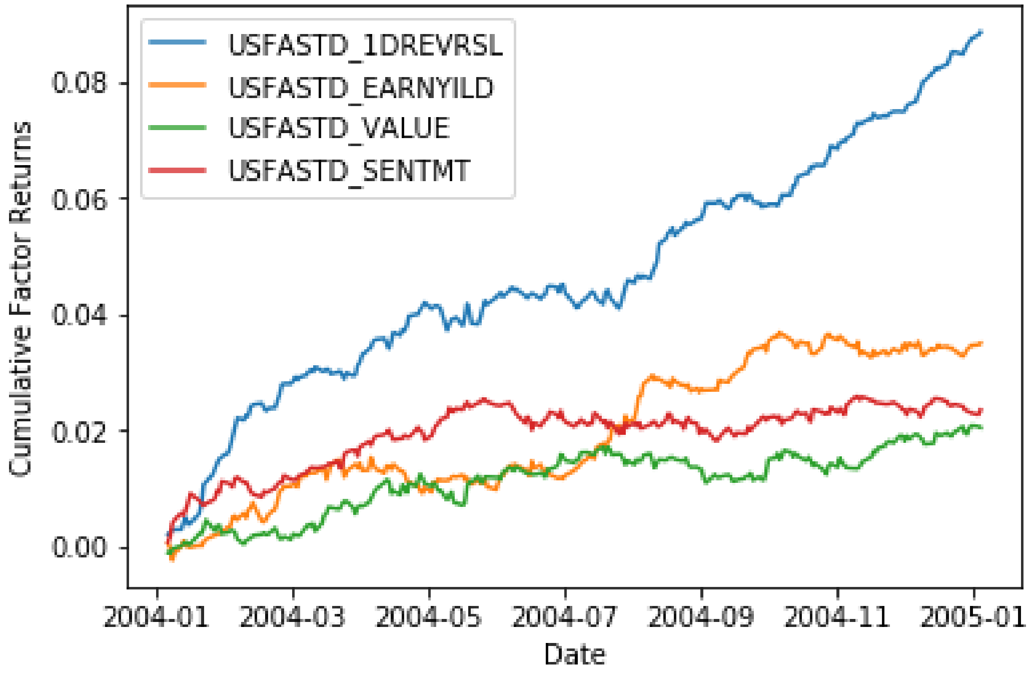
- **USFASTD_1DREVRSL** : Reversal
- **USFASTD_EARNYILD** : Earnings Yield
- **USFASTD_VALUE** : Value
- **USFASTD_SENTMT** : Sentiment

We will choose these alpha factors for now, but you are encouraged to come back to this later and try other factors as well.

```
In [12]: alpha_factors = ["USFASTD_1DREVRSL", "USFASTD_EARNYILD", "USFASTD_VALUE", "USFASTD_SENT

         facret_df = pd.DataFrame(index = my_dates)

         for dt in my_dates:
             for alp in alpha_factors:
                 facret_df.at[dt, alp] = facret[dt.strftime('%Y%m%d')][alp]

         for column in facret_df.columns:
                 plt.plot(facret_df[column].cumsum(), label=column)
```

```
        plt.legend(loc='upper left')
        plt.xlabel('Date')
        plt.ylabel('Cumulative Factor Returns')
        plt.show()
```

```
/opt/conda/lib/python3.6/site-packages/pandas/plotting/_converter.py:129: FutureWarning: Using a

To register the converters:
        >>> from pandas.plotting import register_matplotlib_converters
        >>> register_matplotlib_converters()
  warnings.warn(msg, FutureWarning)
```



1.10 Merge Previous Portfolio Holdings

In order to optimize our portfolio we will use the previous day's holdings to estimate the trade size and transaction costs. In order to keep track of the holdings from the previous day we will include a column to hold the portfolio holdings of the previous day. These holdings of all our assets will be initialized to zero when the backtest first starts.

```
In [13]: def clean_nas(df):
             numeric_columns = df.select_dtypes(include=[np.number]).columns.tolist()

             for numeric_column in numeric_columns:
                 df[numeric_column] = np.nan_to_num(df[numeric_column])
```

```
            return df

In [14]: previous_holdings = pd.DataFrame(data = {"Barrid" : ["USA02P1"], "h.opt.previous" : np.
         df = frames[my_dates[0].strftime('%Y%m%d')]

         df = df.merge(previous_holdings, how = 'left', on = 'Barrid')
         df = clean_nas(df)
         df.loc[df['SpecRisk'] == 0]['SpecRisk'] = median(df['SpecRisk'])
```

1.11 Build Universe Based on Filters (TODO)

In the cell below, implement the function `get_universe` that creates a stock universe by selecting only those companies that have a market capitalization of at least 1 billion dollars **OR** that are in the previous day's holdings, even if on the current day, the company no longer meets the 1 billion dollar criteria.

When creating the universe, make sure you use the `.copy()` attribute to create a copy of the data. Also, it is very important to make sure that we are not looking at returns when forming the portfolio! to make this impossible, make sure to drop the column containing the daily return.

```
In [15]: def get_universe(df):
             """
             Create a stock universe based on filters

             Parameters
             ----------
             df : DataFrame
                 All stocks

             Returns
             -------
             universe : DataFrame
                 Selected stocks based on filters
             """

             # TODO: Implement
             floor = 1e9
             minimum_prev_holding = 0

             universe = df.drop(['DlyReturn'], axis = 1).copy()
             universe = df.loc[(universe['IssuerMarketCap'] >= floor) | abs((df['h.opt.previous'

             return universe

         universe = get_universe(df)

In [16]: date = str(int(universe['DataDate'][1]))
```

1.12 Factors

We will now extract both the risk factors and alpha factors. We begin by first getting all the factors using the `factors_from_names` function defined previously.

```
In [17]: all_factors = factors_from_names(list(universe))
```

We will now create the function `setdiff` to just select the factors that we have not defined as alpha factors

```
In [18]: def setdiff(temp1, temp2):
             s = set(temp2)
             temp3 = [x for x in temp1 if x not in s]
             return temp3
```

```
In [19]: risk_factors = setdiff(all_factors, alpha_factors)
```

We will also save the column that contains the previous holdings in a separate variable because we are going to use it later when we perform our portfolio optimization.

```
In [20]: h0 = universe['h.opt.previous']
```

1.13 Matrix of Risk Factor Exposures

Our dataframe contains several columns that we'll use as risk factors exposures. Extract these and put them into a matrix.

The data, such as industry category, are already one-hot encoded, but if this were not the case, then using `patsy.dmatrices` would help, as this function extracts categories and performs the one-hot encoding. We'll practice using this package, as you may find it useful with future data sets. You could also store the factors in a dataframe if you prefer.

How to use patsy.dmatrices `patsy.dmatrices` takes in a formula and the dataframe. The formula tells the function which columns to take. The formula will look something like this:
`SpecRisk ~ 0 + USFASTD_AERODEF + USFASTD_AIRLINES + ...`
where the variable to the left of the ~ is the "dependent variable" and the others to the right are the independent variables (as if we were preparing data to be fit to a model).

This just means that the `pasty.dmatrices` function will return two matrix variables, one that contains the single column for the dependent variable `outcome`, and the independent variable columns are stored in a matrix `predictors`.

The `predictors` matrix will contain the matrix of risk factors, which is what we want. We don't actually need the `outcome` matrix; it's just created because that's the way patsy.dmatrices works.

```
In [21]: formula = get_formula(risk_factors, "SpecRisk")
```

```
In [22]: def model_matrix(formula, data):
             outcome, predictors = patsy.dmatrices(formula, data)
             return predictors
```

```
In [23]: B = model_matrix(formula, universe)
         BT = B.transpose()
```

1.14 Calculate Specific Variance

Notice that the specific risk data is in percent:

```
In [24]: universe['SpecRisk'][0:2]
```

```
Out[24]: 0     9.014505
         1    11.726327
         Name: SpecRisk, dtype: float64
```

Therefore, in order to get the specific variance for each stock in the universe we first need to multiply these values by `0.01` and then square them:

```
In [25]: specVar = (0.01 * universe['SpecRisk']) ** 2
```

1.15 Factor covariance matrix (TODO)

Note that we already have factor covariances from Barra data, which is stored in the variable `covariance`. `covariance` is a dictionary, where the key is each day's date, and the value is a dataframe containing the factor covariances.

```
In [26]: covariance['20040102'].head()
```

```
Out[26]:             Factor1          Factor2  VarCovar  DataDate
         0  USFASTD_1DREVRSL  USFASTD_1DREVRSL  1.958869  20040102
         1  USFASTD_1DREVRSL      USFASTD_BETA  1.602458  20040102
         2  USFASTD_1DREVRSL    USFASTD_DIVYILD -0.012642  20040102
         3  USFASTD_1DREVRSL    USFASTD_DWNRISK -0.064387  20040102
         4  USFASTD_1DREVRSL  USFASTD_EARNQLTY  0.046573  20040102
```

In the code below, implement the function `diagonal_factor_cov` to create the factor covariance matrix. Note that the covariances are given in percentage units squared. Therefore you must re-scale them appropriately so that they're in decimals squared. Use the given `colnames` function to get the column names from `B`.

When creating factor covariance matrix, you can store the factor variances and covariances, or just store the factor variances. Try both, and see if you notice any differences.

```
In [27]: def colnames(B):
             if type(B) == patsy.design_info.DesignMatrix:
                 return B.design_info.column_names
             if type(B) == pandas.core.frame.DataFrame:
                 return B.columns.tolist()
             return None
```

```
In [28]: def diagonal_factor_cov(date, B):
             """
             Create the factor covariance matrix

             Parameters
             ----------
```