Stock Selection Based on Kernel Principal Component Analysis

HUANG POYUAN

December 3, 2020

0.1 Introduction

This project investigate how Kernel Principal Component Analysis (KPCA) reduce variables in our predicting model so as to make further portfolio management. The motivation comes from the drawback of using PCA on large dataset where all SP500 or NASDAQ tickers are included in our universe. PCA deals with linear classification whereas KPCA deals with non-linear classification and have even more versatile decision boundaries based on the choice of kernel function. In this paper, I will introduce the method of KPCA and then the trading strategy using it. Also, I will present empirical studies and analysis over the SP500 top30 stocks daily dataset.

0.2 Kernel PCA

The idea of using principal component analysis on is that one wish to extract only the important factors that explained moments the most of our universe. Although one might lose some information or variance per se about the market, one does not need to include all factors in the predicting model that will cause statistically insignificant or collinearity of the parameters.
In the case of having T ¡¡ N dataset where T is the number of observation and N is the number of features, PCA on the stock covariance matrix may be tricky as it no longer becomes full rank. However, KPCA maps the original space to higher dimension to find a suitable hyper-plane for classification. The kernel function does the mapping for us. Once we get kernel matrix, we decompose the matrix as we did on covriance matrix.

- Centered data $\sum h(x) = 0$
- Compute kernel matrix K by kernel function $< h(x_i), h(x_j) >= K(x_i, x_j)$
- Decompose K to get eigen-values and eigen-vectors
- Set threshold of variance explained to determine the number of extracted factors
- Project data to eigen-vectors to get principal components.

0.2.1 Kernel functions

Kernel function is a similarity measure. The optimal kernel function selection can be done by cross-validation or prior knowledge of dataset. Here we only introduce radial basis function (RBF) and polynomial (POLY) function.

- POLY of degree d: $K(x_i, x_j) = (x_i^T x_j + 1)^d$
- RBF with tuning γ: $K(x_i, x_j) = exp(-\gamma||x_i - x_j||_2^2)$

In general, RBF can map to infinite dimensions and its decision boundaries are more flexible. POLY has better performance in non-linear classification and is less time consuming. As for the downside, RBF may be hard to keep track of data after transformation whereas POLY can easily over-fit as degree raises.

0.2.2 Comparison between RBF vs POLY

The dataset are top30 SP500 from 2006-12-29 to 2020-09-29 and is regrouped monthly. The training window is 24 (months). The statistical summary of predicted model are shown below.

	RBF (gamma)		**POLY (d)**	
Parameters	1	2	3	4
1st Explained Var	0.239	0.181	0.412	0.408
Accumulated Var	0.447	0.376	0.619	0.616

POLY is better than RBF in terms of explained variance. However, POLY shows a potential over-fit since the explained variance does not increase much as the degree raise. And BRF has balanced explained variance over four components and that accumulated variance increase as smaller gamma is used.

0.2.3 Cross-Validation tuning parameter

An optimal choice of kernel parameter will improve the classification precision for the first few components. Here we use GridSearchCV from "sklearn" and use MSE as our scoring function. The optimal kernel is RBF with gamma 0.03. Part of the code:

```
param_grid = [{
            "gamma": np.linspace(0.03, 4, 20),
            "kernel": ["rbf","poly" ]
            }]

kpca=KernelPCA(fit_inverse_transform=True, n_jobs=-1)  # n_jobs using how many parallel processor
grid_search = GridSearchCV(kpca, param_grid, cv=3, scoring=MSE) #cv: how many fold
```

Figure 1: CV

0.3 Methodology

Given window (wd), numbers of tickers (N), number of factors extracted (p), active weights of each stock (aw_i).

- Select data window Data[wd] and centered the data X
- Do KPCA to get $PC_i = X\frac{v_i}{\sqrt{\lambda_i}}$ up to p

- Construct multi-factor model $r_i = \alpha_i + \sum_{j=1}^{p} \beta_{ij} PC_j$
- Predict r_{it+1} based on the t th projected PC
- Select stocks that have higher predicted return than the benchmark
- Select more stocks that have less correlation to benchmark if the benchmark is negative.
- Use mean-variance optimization to get optimal aw_i
- Execute asset allocation based on the weights and roll the window

0.3.1 Mean-Variance Optimization

Based on long only and beta neutral policy, we can quadratic programming with restrictions. Let $x = [\Delta x_1, \Delta x_2, ..\Delta x_n]$ be the active weights for the selected stocks. λ is risk-aversion parameter and μ is mean return. $\beta = [\beta_1, \beta_2, ..\beta_n]$ is beta of each stock. We use scipy optimize package.

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & \frac{\lambda}{2} x^{'} \Sigma x - \mu^{'} x \\ \text{subject to} \quad & 1^{'} x = 0 \\ & \beta^{'} x = 0 \\ & 1^{'} x \succeq w_b \\ & \sqrt{x^{'} \Sigma x} \leq te_d \end{aligned}$$

```
arg_fun = (dt, mu)
arg_cos0 = (I, 0.0)  # aw = 0
arg_cos1 =(beta, b0)   #beta neutral
arg_cos2 = (S, track)  #tracking error
bnds = ((-1/n, 1.0001),) * len(dt) # long only
cons = (
       {'type': 'eq', 'fun': cons0, 'args': arg_cos0},
       {'type': 'eq', 'fun': cons1, 'args': arg_cos1},
       {'type': 'ineq', 'fun': cons2, 'args': arg_cos2}
       )

guess = (1/n,) * len(dt)
result = scipy.optimize.minimize(fun, x0 = guess, method = 'SLSQP',
                                 args = arg_fun, constraints = cons,
                                 bounds=bnds,
                                 options={'maxiter': 101})
```

Figure 2: Optimization

0.4 Empirical Study

0.4.1 Dataset

The data set is derived from yfinance. The universe is top 80 stocks and GSPC daily adjusted close price starting from 2009-12-31 to 2020-10-30.
Each time, we select window size of 24 days x 80 to construct our model. We start our initial CASH = 10000 and no additional cash are invested afterwards. RBF gamma 0.03 is used.

0.4.2 Result

	Portfolio	EqualWeight	Bench
End of Return(%)	635	0	207
Av. Annual TE	0.07		
Worst year of TE	2020		
Best year of TE	2013		
Av. Annual IR	1.29		
Worst year of IR	2016		
Best year of IR	2013		

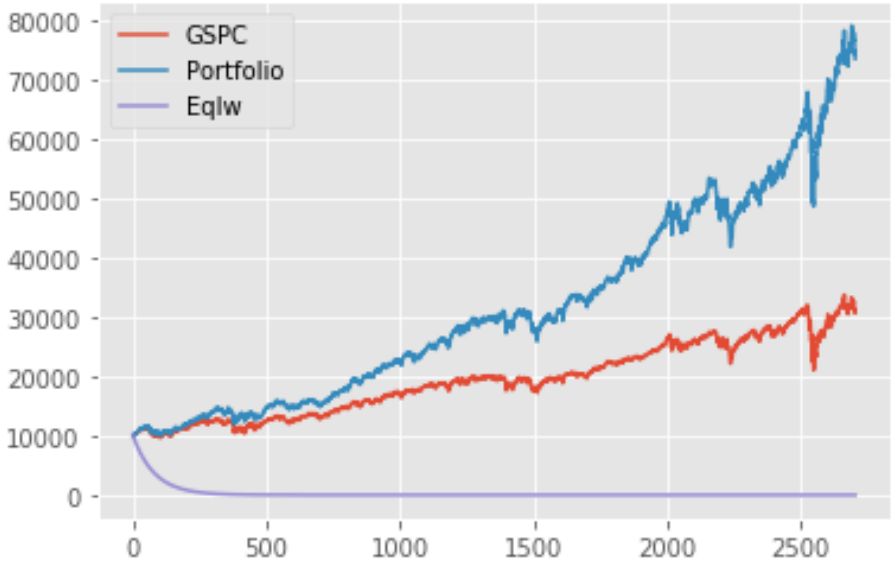

Figure 3: RBF

Note that 2020 is calculated up to October and the date of IR/TE is shifted since I put them on the start of next year.

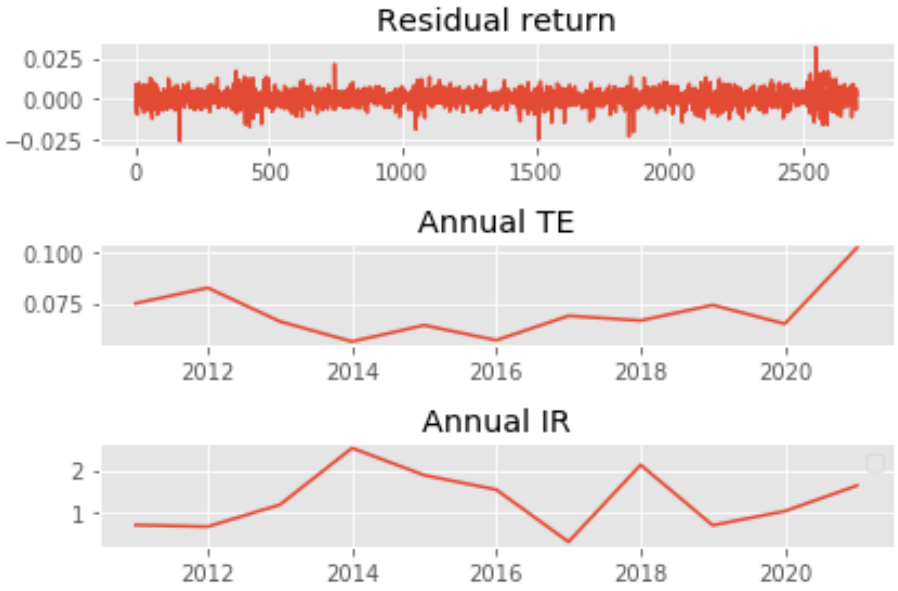

Figure 4: TE/IR

0.4.3 Analysis

The average TE of the portfolio is 7 % which is higher than our 3 % goal.
A possible reason is the bias from the choice of our universe. The data set of "top80" stocks somehow uses future information while I start my portfolio construction in 2010. This is why my portfolio grow abnormally in the last few years. Another possible reason is that during back-testing, making withdraw or deposit are not allowed. Only present (value) cash is used for next asset allocation.

As for KPCA technique itself, although in retrospect reading the paper, technical factors, evaluation factors and macroeconomic factor...etc are also included to have a higher confidence picking to-grow stocks, the effect of KPCA is not that apparent if the goal is to track the benchmark. One advantage using KPCA here is that in average, 37 stocks are being concerned instead of the whole universe of 30 stocks. This helps me to reduce computation when doing daily optimization since less stocks data set are included.

0.5 Code file

There are four .py files: kPCA-algo, Backtest, FINAL-PROJECT, cool. kPCA-algo, cool and Backtest are supported files and the main script is in FINAL-

PROJECT.

0.6 Reference

https://www.scirp.org/journal/paperinformation.aspx?paperid=98781
https://faculty.washington.edu/ezivot/econ589/estimatingstatisticalfactormodels.pdf
https://web.stanford.edu/ boyd/cvxbook/bv$_c$*vxbook.pdf*
https://www.sciencedirect.com/science/article/pii/S1877050918301236