MTH 9899 Final Project White Paper

Yizhou Wang, Ming Fu, Shangwen Sun

March 23, 2022

Contents

1 Feature engineering

1.1 Creation of features

In this subsection, we will introduce the way we create each feature. In total, there are 30 features.

1.1.1 Features based on intraday return data

- 10/17res= $R_{residual,10:00} - R_{residual,17:30}$
- 10/17raw= $R_{raw,10:00} - R_{raw,17:30}$
- 16/17res= $R_{residual,16:00} - R_{residual,17:30}$
- 16/17raw= $R_{raw,16:00} - R_{raw,17:30}$

1.1.2 Features based on interday return data

- 17ma1raw= $MovingAverage(R_{raw,17:30}, 1)$
- 17ma3raw= $MovingAverage(R_{raw,17:30}, 3)$
- 17ma5raw= $MovingAverage(R_{raw,17:30}, 5)$
- 17ma20raw= $MovingAverage(R_{raw,17:30}, 20)$
- 17ma1-3raw= $17ma1raw - 17ma3raw$
- 17ma1-5raw= $17ma1raw - 17ma5raw$
- 17ma1-20raw= $17ma1raw - 17ma20raw$
- 17ma1res= $MovingAverage(R_{res,17:30}, 1)$
- 17ma3res= $MovingAverage(R_{res,17:30}, 3)$
- 17ma5res= $MovingAverage(R_{res,17:30}, 5)$
- 17ma20res= $MovingAverage(R_{res,17:30}, 20)$
- 17ma1-3res= $17ma1res - 17ma3res$
- 17ma1-5res= $17ma1res - 17ma5res$
- 17ma1-20res= $17ma1res - 17ma20res$

1.1.3 Market value

- market_value= $CleanMid_{17:30} \times SharesOutstanding_{17:30}$

1.1.4 Features based on intraday volume data

- 10/17vol= $\frac{CumVolume_{10:00}}{CumVolume_{17:30}}$
- 16/17vol= $\frac{CumVolume_{16:00} - CumVolume_{10:00}}{CumVolume_{17:30}}$

1.1.5 Features based on turnover of stocks

- turnover_ma1= $MovingAverage(CumVolume_{17:30}/SharesOutstanding_{17:30}, 1)$
- turnover_ma3= $MovingAverage(CumVolume_{17:30}/SharesOutstanding_{17:30}, 3)$
- turnover_ma5= $MovingAverage(CumVolume_{17:30}/SharesOutstanding_{17:30}, 5)$
- turnover_ma20= $MovingAverage(CumVolume_{17:30}/SharesOutstanding_{17:30}, 20)$

1.1.6 Features from the original data

- estVol= $estVol_{17:30}$
- cleanMid17= $CleanMid_{17:30}$
- cleanMid16= $CleanMid_{16:00}$
- cleanMid10= $CleanMid_{10:00}$
- vol17= $CumVolume_{17:30}$

The below figures 1-4 are scatter plots of all features we created.

1.2 Feature selection

Feature selection is one of the first and important steps while performing any machine learning task. A feature in case of a dataset simply means a column. When we get any dataset, not necessarily every feature is going to have an impact on the output variable. If we add these irrelevant features in the model, it will just make the model worst. This gives rise to the need of doing feature selection. Here, based on the given data and created features, we only need to discuss about numeric feature selection.

Generally speaking, there are three methods to conduct feature selection: the embedded method, the wrapper method and the filter method. However, since the number of features in our model is quite large and Lasso cannot deal well with co-linear variables, therefore, we do not use embedded method like Lasso to conduct feature selection. In the following part, we try feature selection methods of step forward selection, backward elimination and filter methods using different indicators like variance inflation factor(VIF) and Pearson Correlation respectively to compare the results.

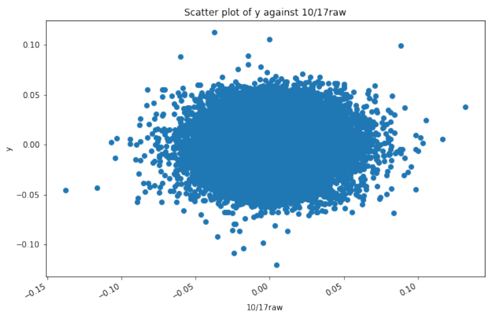
Note that though filter method is less accurate. It is quite useful for checking multi co-linearity. Though wrapper methods give more accurate results but as they are computationally expensive, these method are suited when you have less features.

1.2.1 Filter method

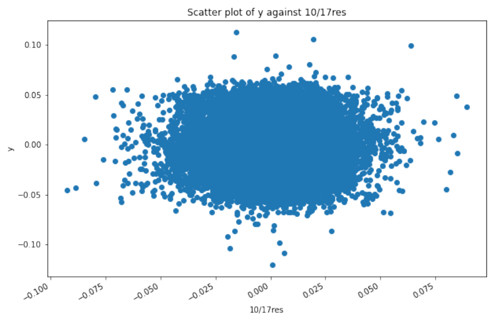
In this method, we filter and take only the subset of the relevant features. The filtering here is doing using correlation matrix and variance inflation factor.

1.2.1.1 Pearson correlation

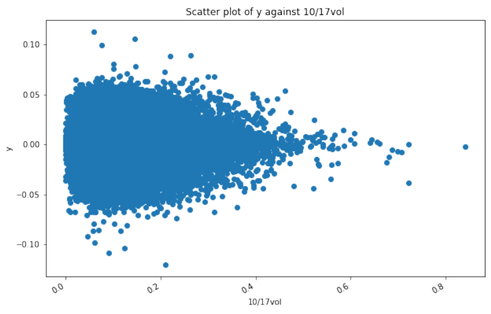
Here, as in figure 5, we first plot the Pearson correlation heatmap and see the correlation of independent variables with the output variable.



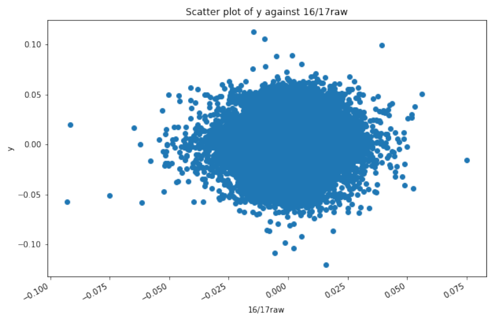
(a) 10_17raw



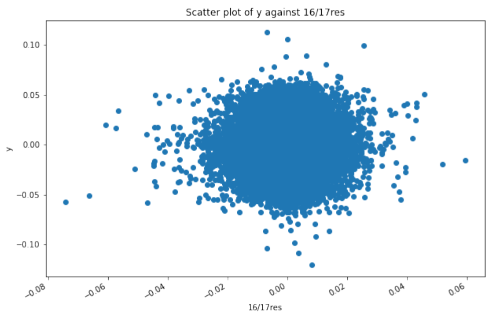
(b) 10_17res



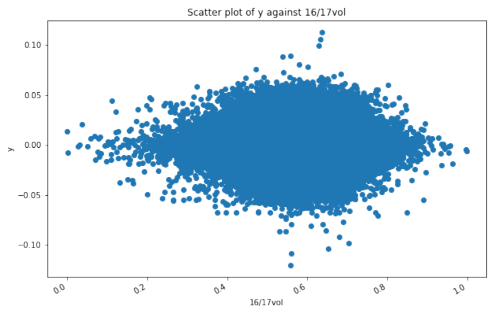
(c) 10_17vol



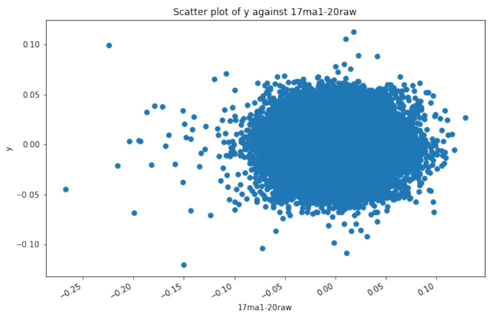
(d) 16_17raw



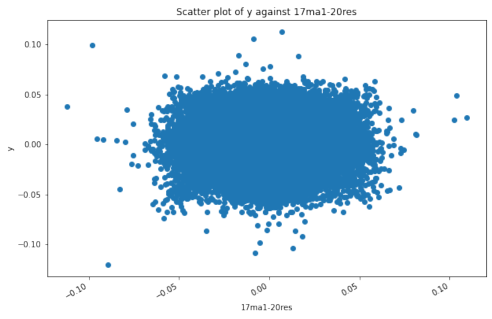
(e) 16_17res



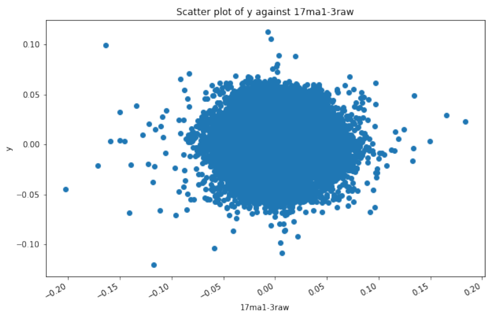
(f) 16_17vol



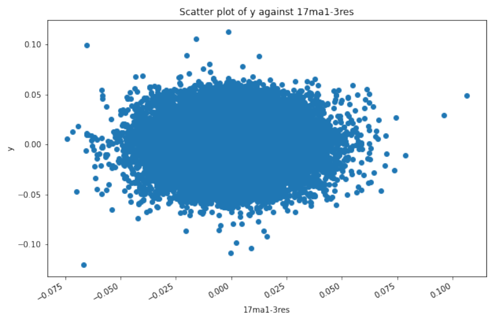
(g) 17ma1-20raw



(h) 17ma1-20res

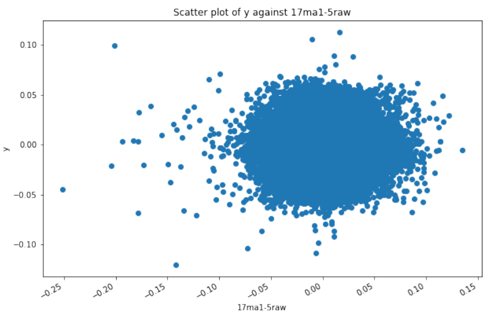


(i) 17ma1-3raw

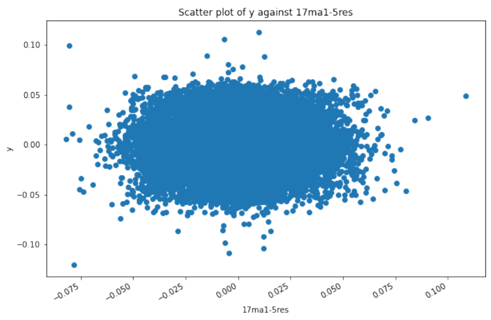


(j) 17ma1-3res

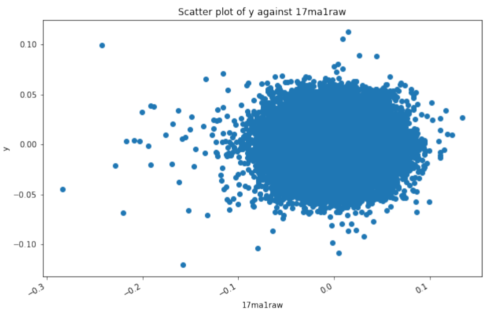
Figure 1: Scatter plots of features



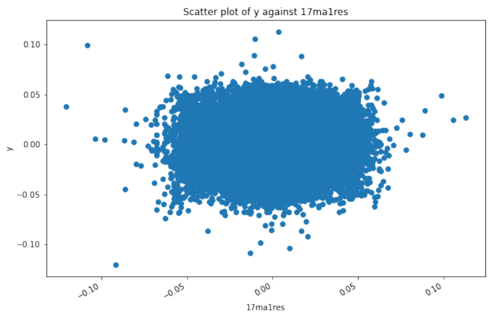
(a) 17ma1-5raw



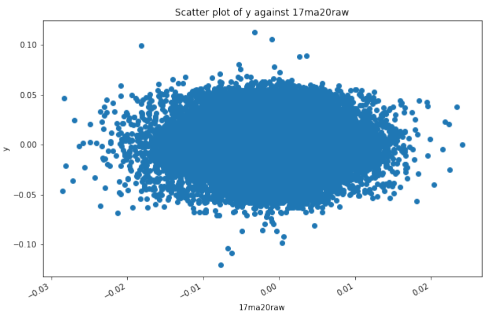
(b) 17ma1-5res



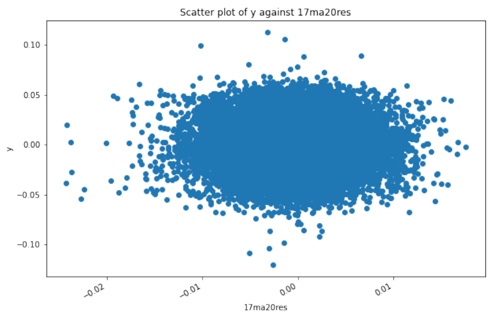
(c) 17ma1raw



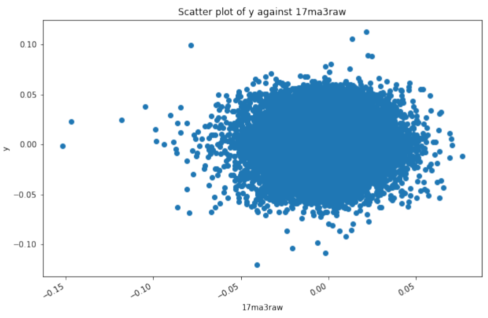
(d) 17ma1res



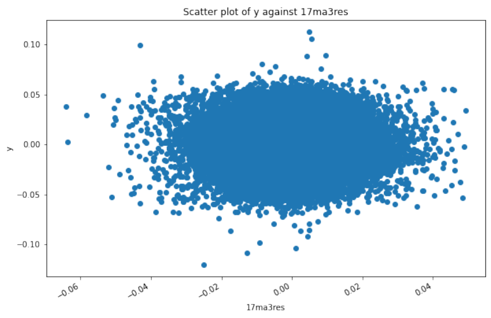
(e) 17ma20raw



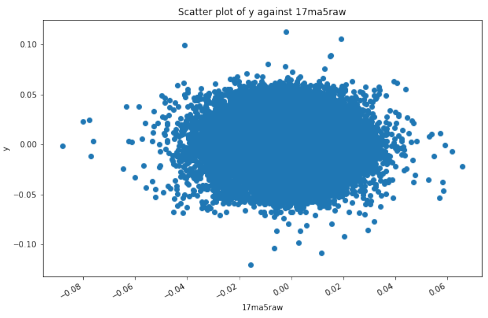
(f) 17ma20res



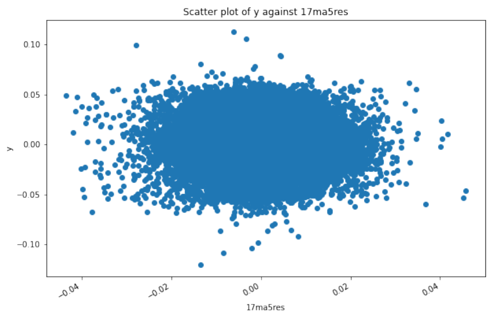
(g) 17ma3raw



(h) 17ma3res

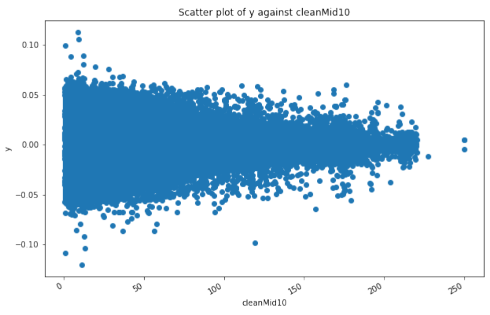


(i) 17ma5raw

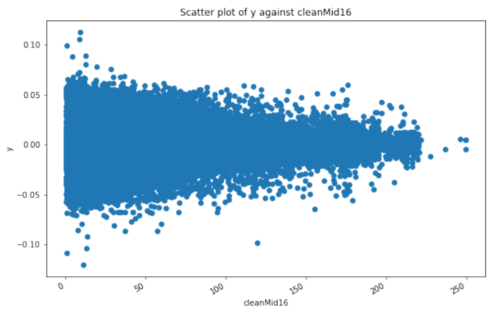


(j) 17ma5res

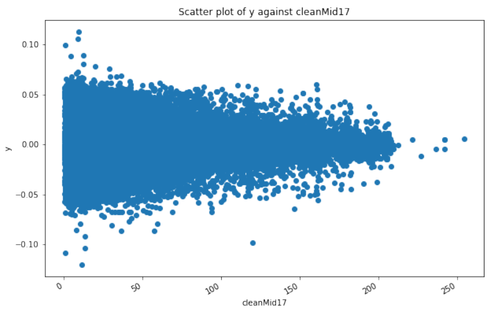
Figure 2: Scatter plots of features



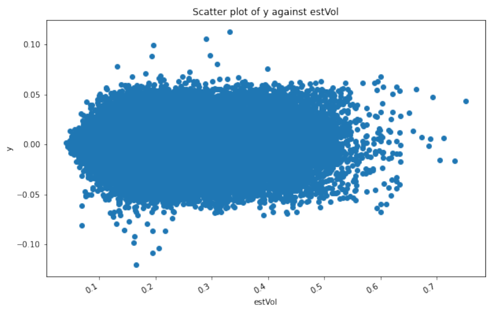
(a) cleanMid10



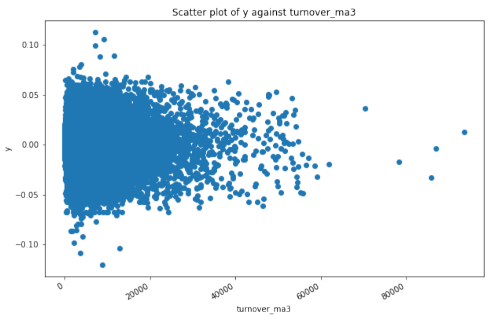
(b) cleanMid16



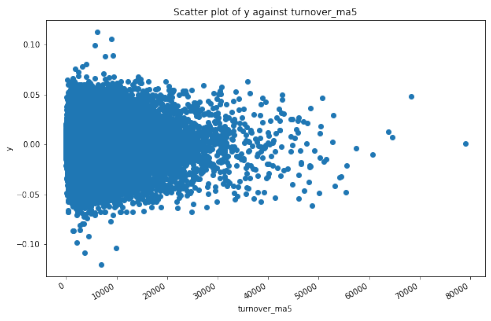
(c) cleanMid17



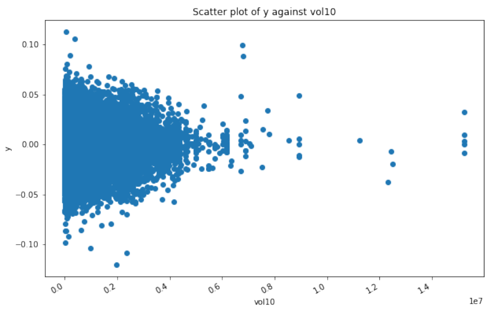
(d) estVol



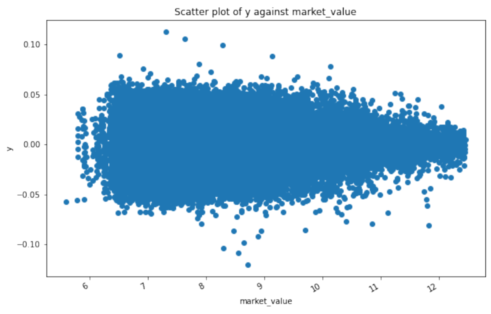
(e) turnover_ma3



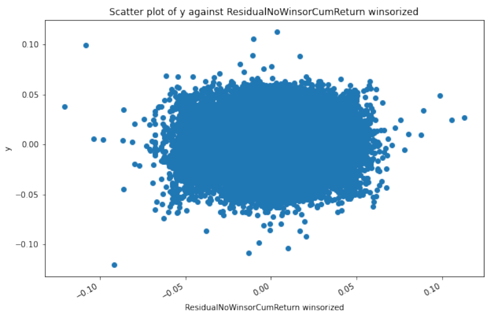
(f) turnover_ma5



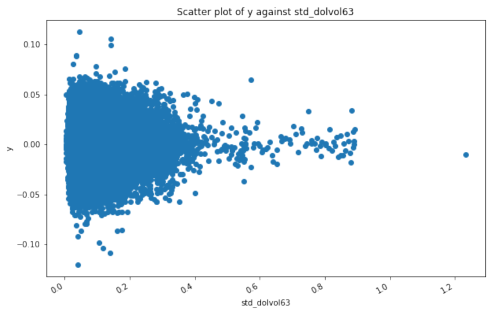
(g) vol10



(h) market_value

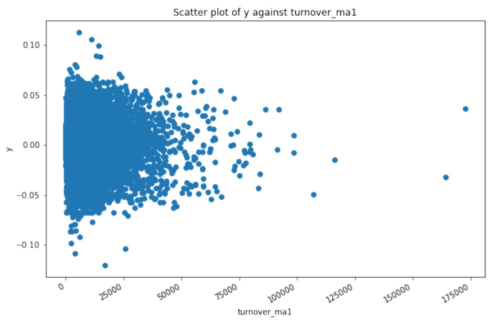


(i) ResidualCumReturn winsorized

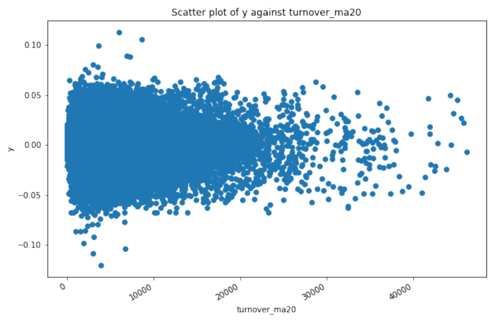


(j) std_dolvol63

Figure 3: Scatter plots of features



(a) turnover_ma1



(b) turnover_ma20

Figure 4: Scatter plots of features

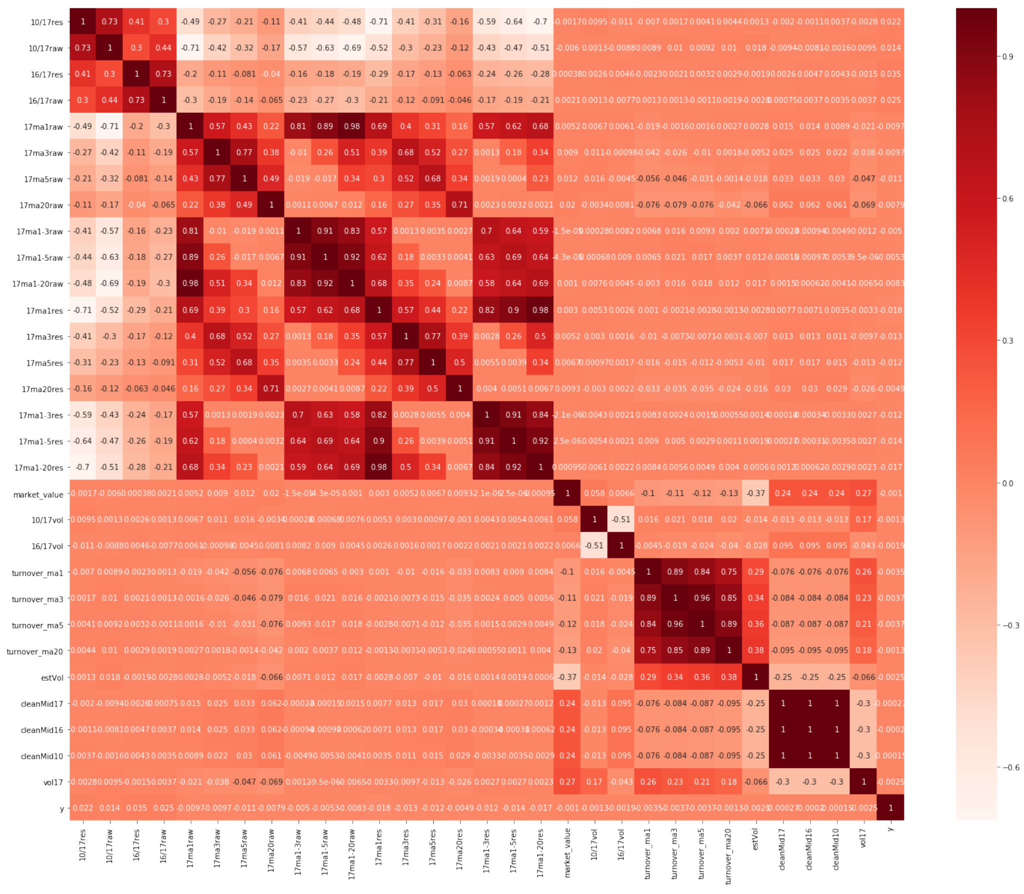


Figure 5: Pearson correlation heatmap of 30 features

1.2.1.2 Variance inflation factor

Then we use Variance Inflation Factor (VIF) to filter our features, which is not intended to improve the quality of the model, but to remove the autocorrelation of independent variables.

The steps are as follows:

- Calculate the VIF factors.
- Inspect the factors for each predictor variable, if the VIF is larger than 10, multicollinearity is likely present and you should consider dropping the variable.

After dropping columns with VIF larger than 10, 15 features are left. They are '10/17res', '10/17raw', '16/17res', '16/17raw', '17ma3raw', '17ma3res', '17ma5res', 'market_value', '10/17vol', '16/17vol', 'turnover_ma1', 'turnover_ma20', 'estVol', 'cleanMid17', 'vol17'.

1.2.2 Step forward selection

Forward selection is an iterative method in which we start with having no feature in the model. In each iteration, we keep adding the feature which best improves our model till an addition of a new variable does not improve the performance of the model.

Here, we set the number of features as 22. Step forward selection selects the following features: '16/17res', '16/17raw', '17ma3raw', '17ma5raw', '17ma20raw', '17ma1-3raw', '17ma1-5raw', '17ma3res', '17ma20res', '17ma1-20res', 'market_value', '10/17vol', '16/17vol', 'turnover_ma3', 'turnover_ma5', 'turnover_ma20', 'std_dolvol63', 'estVol', 'cleanMid17', 'cleanMid16', 'cleanMid10', 'vol10'

1.2.3 Backward elimination

In this method, we feed all the possible features to the model at first. We check the performance of the model and then iteratively remove the worst performing features one by one till the overall performance of the model comes in acceptable range.

Here, we set the number of features as 21. Backward elimination leaves us with the following features: '16/17res', '16/17raw', '17ma1raw', '17ma3raw', '17ma5raw', '17ma1-3raw', '17ma1-5raw', '17ma1res', '17ma3res', '17ma20res', '17ma1-3res', 'market_value', '10/17vol', '16/17vol', 'turnover_ma3', 'std_dolvol63', 'cleanMid17', 'cleanMid10', 'vol10', 'ResidualCumReturn winsorized'.

In the model fitting process, we tried these three different subsets and also the whole feature set. And we will only record the best performance among them.

1.3 Feature Importance

Feature importance refers to techniques that assign a score to input features based on how useful they are at predicting a target variable.

Feature importance scores play an important role in a predictive modeling project, including providing insight into the data, insight into the model, and the basis for dimensionality reduction and feature selection that can improve the efficiency and effectiveness of a predictive model on the problem.

Specifically, permutation feature importance is a model inspection technique that can be used for any fitted estimator when the data is tabular. This is especially useful for non-linear or opaque estimators. The permutation feature importance is defined to be the decrease in a model score when a single feature value is randomly shuffled. This procedure breaks the relationship between the feature and the target, thus the drop in the model score is indicative of how much the model depends on the feature. This technique benefits from being model agnostic and can be calculated many times with different permutations of the feature. The permutation feature importance for each model can be found in Section 3.

Moreover, bin plots of target variable versus the features are presented in figure 6-9 below.

2 Data splitting

The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model. This step is necessary and important, without which serious data leakage might occur.

Sometimes, validation set is also necessary so that we could do hyperparameter tuning in cross validation, early stopping to avoid overfitting and so on.

Therefore, we introduce the procedure of data splitting with respect to 2 different scenarios (with or without validation set) in the following.

2.1 Train-validation-test split

See in figure 10 the idea of train-validation-test split.

2.1.1 Early stopping

When we use early stopping rounds in model fitting, a validation set is needed. Therefore, we use data from 2014-2015 as training set, 2016 as validation set, and 2017 as test set.

When new hold-out data from year 2018 is available, we use 2014-2016 as training set, 2017 as validation set, 2018 as test set.

2.1.2 Hyperparameter tuning

When we do hyperparameter tuning via cross validation, a validation set is also needed. In this case, we use data from 2014 to 2016 to do cross validation and data from 2017 as test set.

In cross validation, specifically, we used time series cross validation to fine tune each parameter. Because it's important to notice that the common cross validation technique will cause leakage for time series data. By shuffling past and future data, the learner learns the future that it is not supposed to know. As a result, the cross validation scores will be very good, but when the time comes to use the learner to try to predict the real future, the results will be very bad.

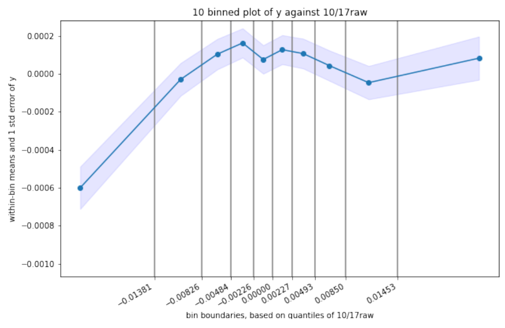
Therefore, to prevent leakage, It is better to set the old data as training set and the new data as validation set. We introduce the following two time series split methods which are used in cross validation.

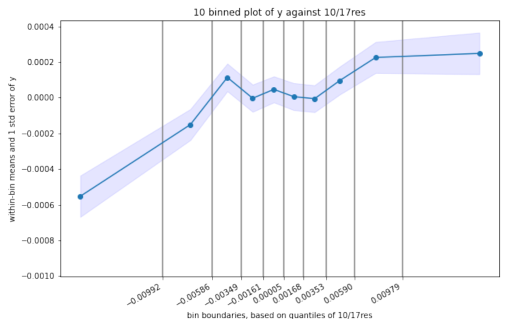
2.1.2.1 Time series split

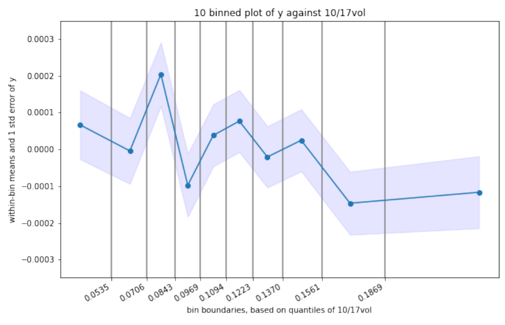
- Expanding TimeSeriesSplit (from sklearn.model_selection)

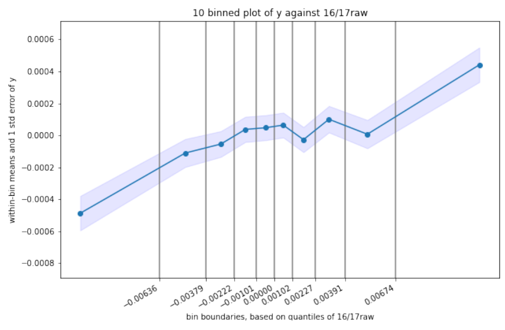
  From image 11a, we can see that as time goes by, the amount of training data is increasing.

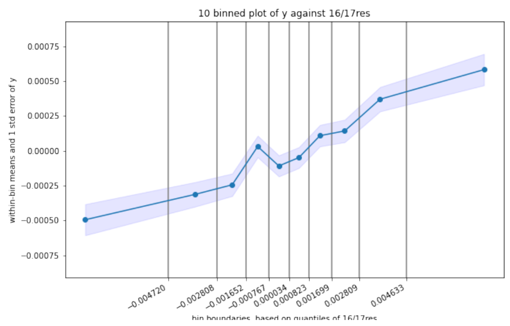
- Rolling TimeSeriesSplit

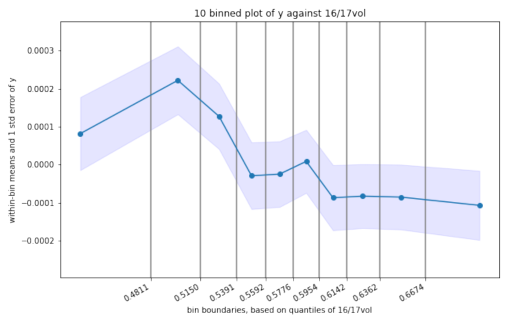

(a) 10_17raw

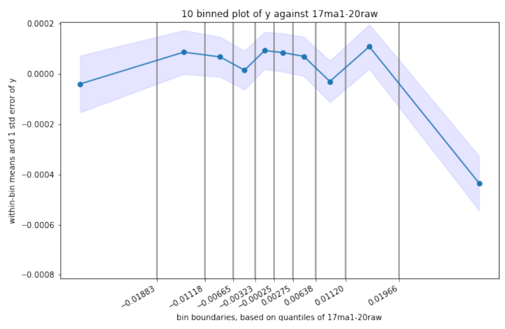

(b) 10_17res

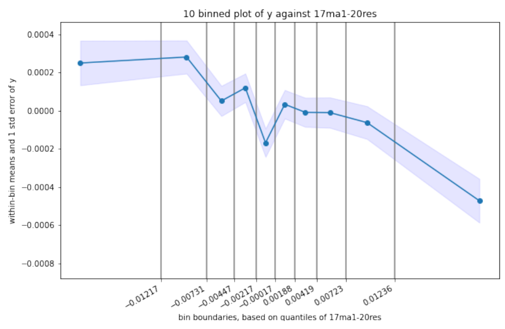

(c) 10_17vol

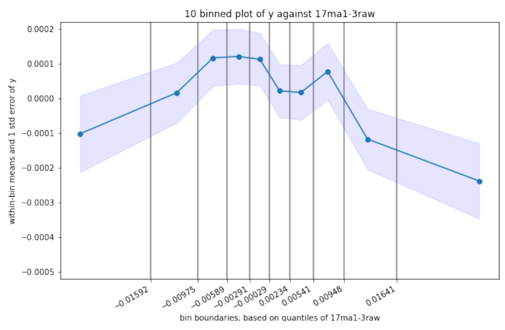

(d) 16_17raw


(e) 16_17res


(f) 16_17vol


(g) 17ma1-20raw


(h) 17ma1-20res


(i) 17ma1-3raw

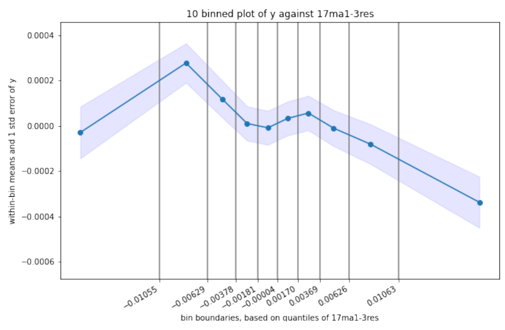
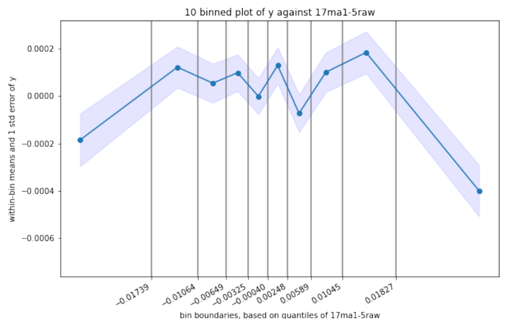
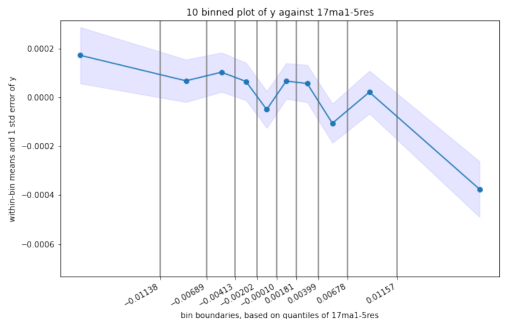

(j) 17ma1-3res

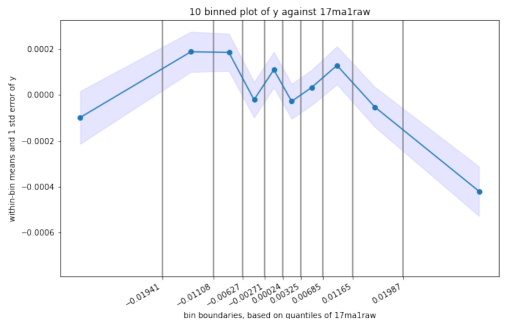
Figure 6: Bin plots of significant feature versus target variable



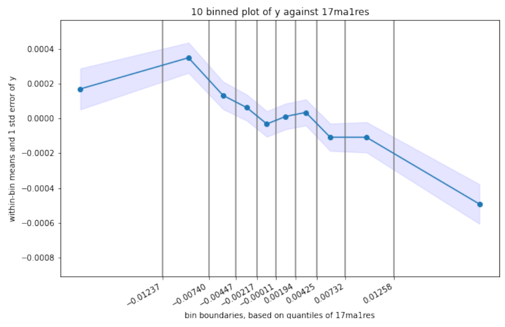
(a) 17ma1-5raw



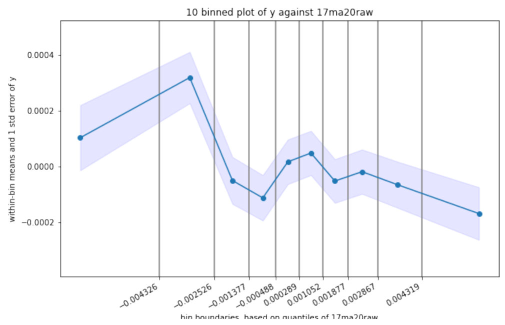
(b) 17ma1-5res



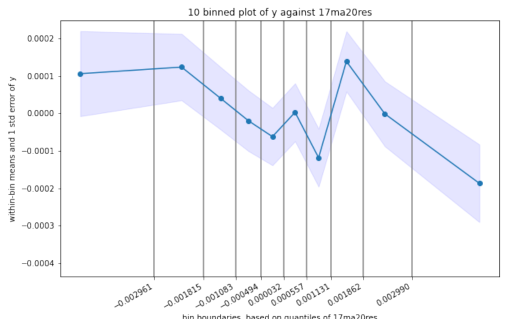
(c) 17ma1raw



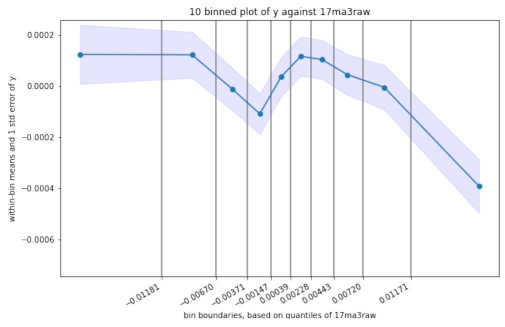
(d) 17ma1res



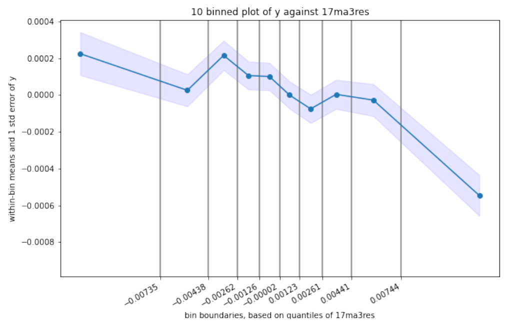
(e) 17ma20raw



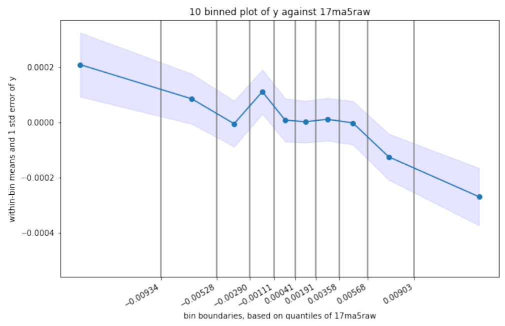
(f) 17ma20res



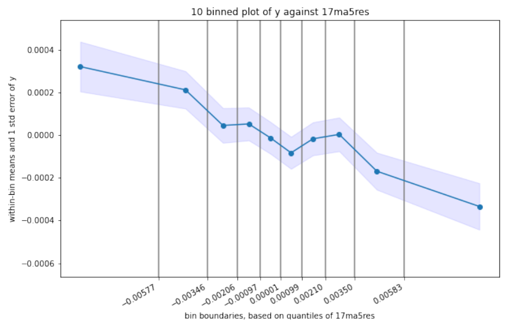
(g) 17ma3raw



(h) 17ma3res

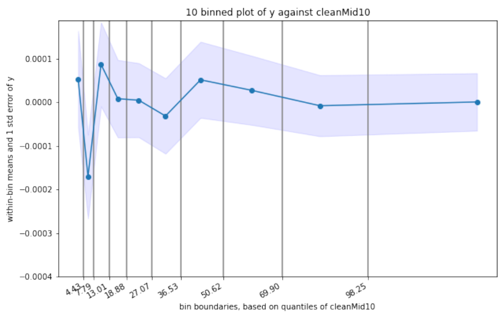


(i) 17ma5raw

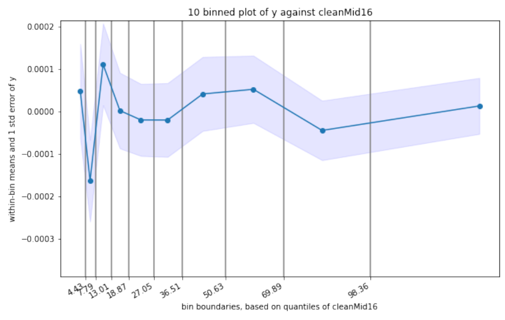


(j) 17ma5res

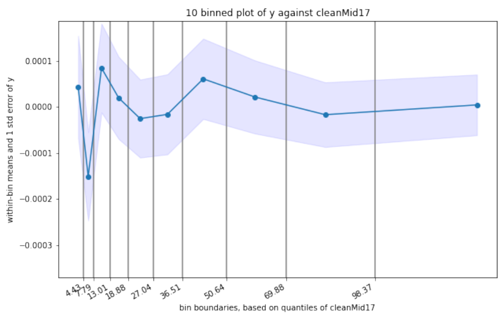
Figure 7: Bin plots of significant feature versus target variable



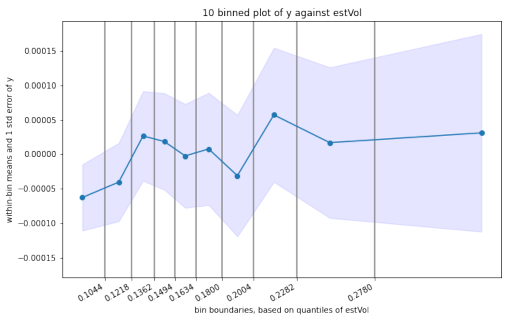
(a) cleanMid10



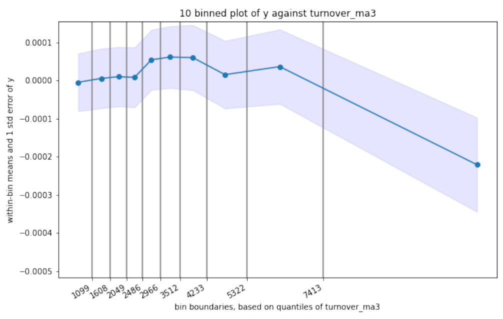
(b) cleanMid16



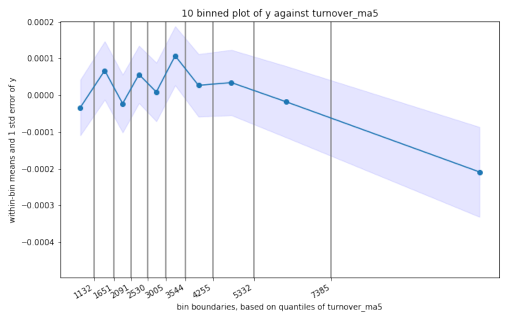
(c) cleanMid17



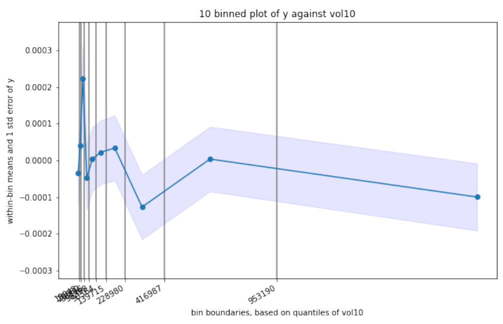
(d) estVol



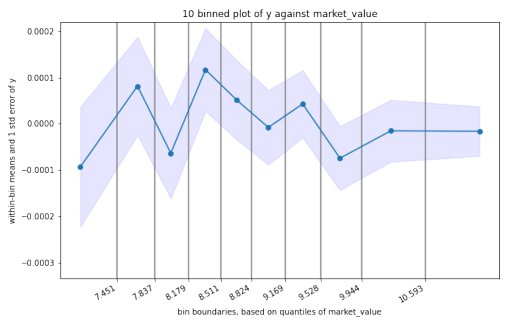
(e) turnover_ma3



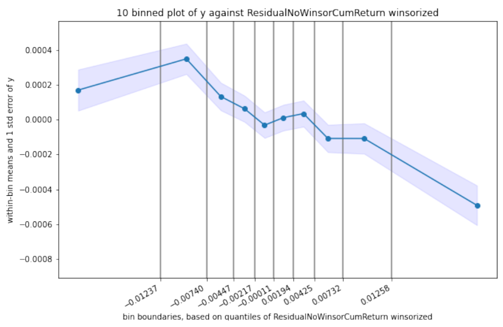
(f) turnover_ma5



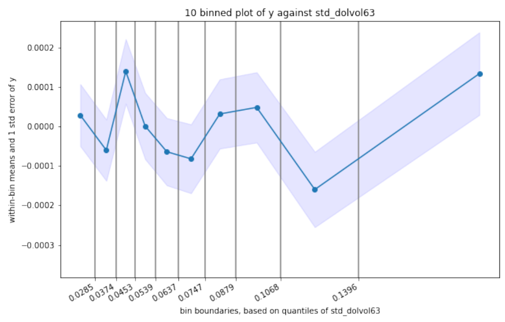
(g) vol10



(h) market_value

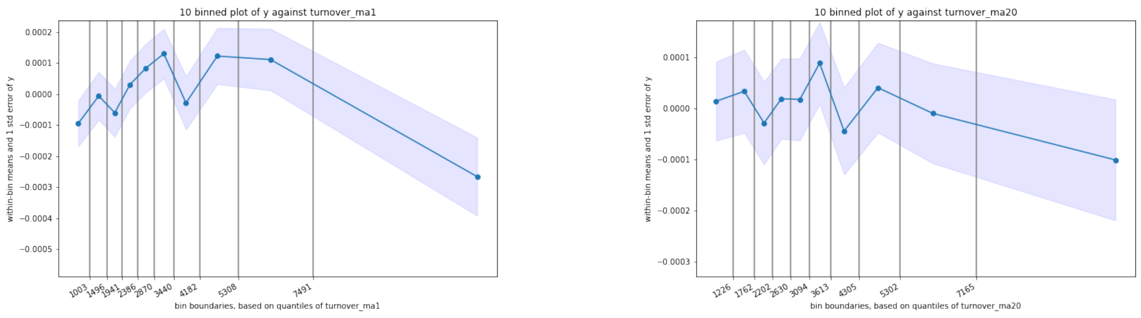


(i) ResidualCumReturn winsorized



(j) std_dolvol63

Figure 8: Bin plots of significant feature versus target variable



(a) turnover_ma1 (b) turnover_ma20

Figure 9: Bin plots of significant feature versus target variable

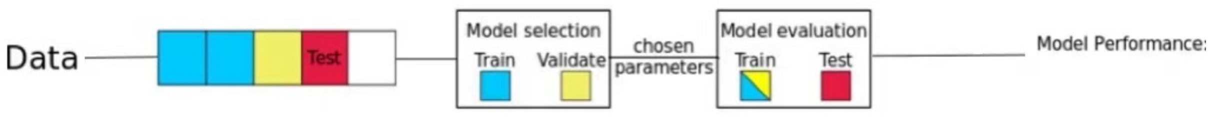


Figure 10: Train-validation-test data splits

We remove old data every time and equalize the amount of data between folds, as shown in the image 11b below.

2.2 Train-test split

Except the two cases mentioned above which need a validation set, we only need train and test set in usual model fitting and predictions. In this sense, we use data from 2014-2016 as the training set, 2017 as the test set.

When new hold-out data 2018 is available, we use 2014-2017 as the training set, 2018 as the test set.

2.3 Missing value manipulation

Missing value manipulation is important data pre-processing procedure, which will directly decides the data we input into each model.

First, we forward fill missing values with respect to each columns based on Time and Id, which means we fill the missing values using the previous first not NaN value of that stock at the same time in a day.

Then, we need to deal with cumulative features because they can't be smaller than its value earlier on the same day.

Therefore, we forward fill missing values cross-sectionally on the original dataset based on Date and Id, compare with the forward filling values we achieved before and choose the larger one.

Ultimately, we simply drop those rows still with NaN's.

2.4 Model implementation

Generally, the model implementation details are as follows:

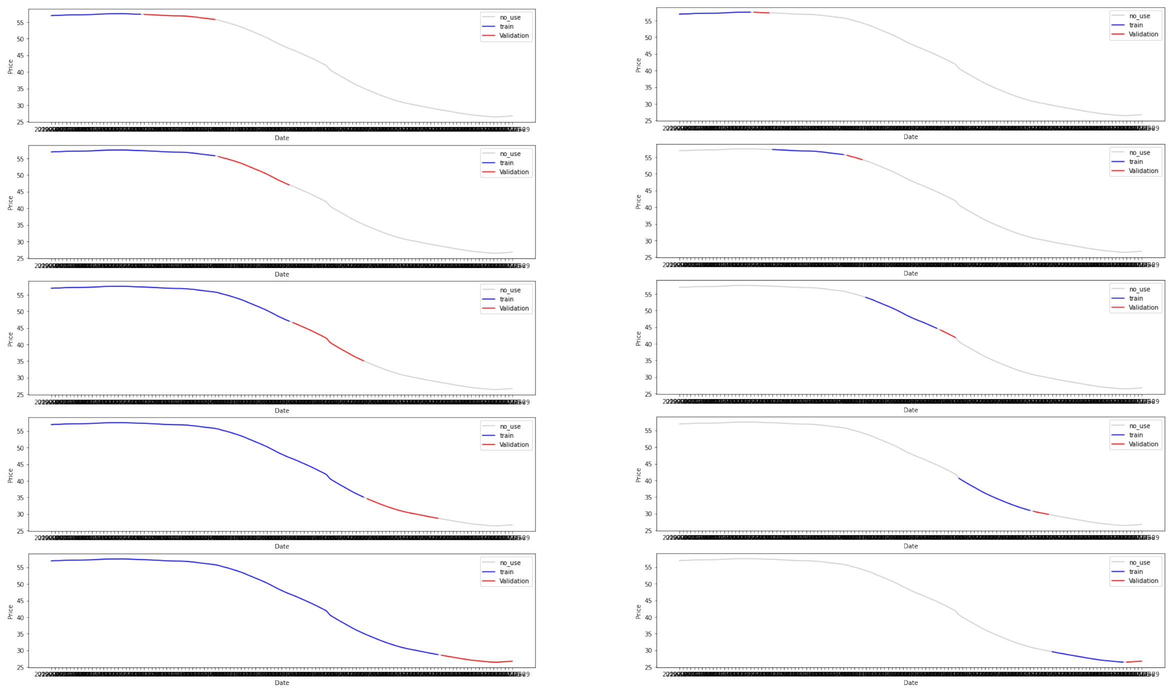

(a) TimeSeriesSplit (b) Rolling TimeSeriesSplit

Figure 11: Two time series split methods for cross validation

- First, we load raw data from zip file, merge csv files, deal with missing value and winsorize data cross-sectionally with 5 MAD.
- Then, we did feature creation, feature selection to get the model input dataframe with selected features and target variable.
- After that, time series cross validation is performed based on two time series split methods to select the hyperparameters in the model.
- Finally, model with best hyperparameters is fit and test, where predictions are made on each subset and weighted R^2 is computed.

3 Models

In the following sections, we introduce models we use regarding the algorithm, hyperparameter tuning, feature importance and performance.

3.1 Regression models

A regression model provides a function that describes the relationship between one or more independent variables and a response, dependent, or target variable.

3.1.1 Ordinary linear regression

Ordinary least squares (OLS) is a type of linear least squares method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by the principle of least squares: minimizing the sum of the squares of the

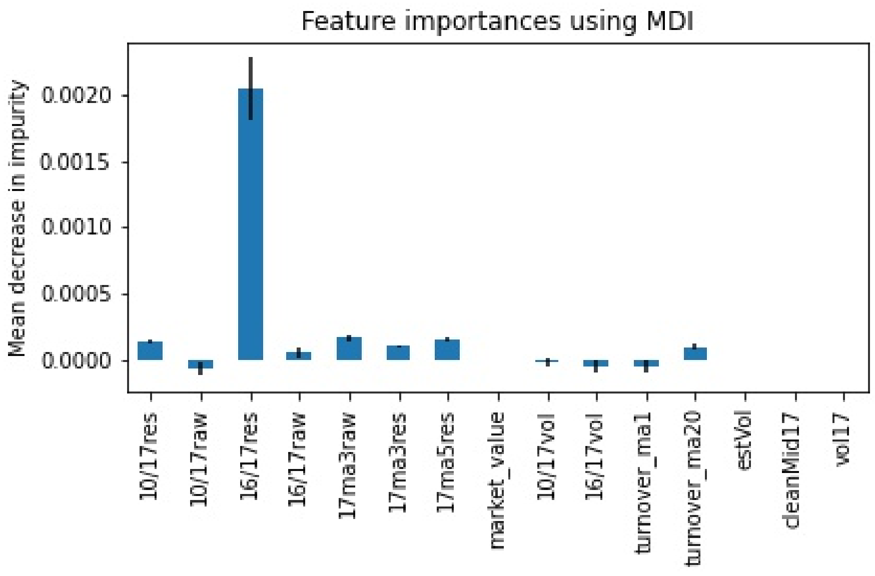


Figure 12: Permutation feature importance of ordinary linear regression

Table 1: Performance of OLS

Model name	18 R^2_{test} (bps)
OLS (15 features)	11.2

differences between the observed dependent variable in the given dataset and those predicted by the linear function of the independent variable.

By the Gauss–Markov theorem, OLS estimator is optimal in the class of linear unbiased estimators when the errors are homoscedastic and serially uncorrelated. Under these conditions, the method of OLS provides minimum-variance mean-unbiased estimation when the errors have finite variances. Under the additional assumption that the errors are normally distributed, OLS is the maximum likelihood estimator.

The loss function of OLS is

$$\mathcal{L}_{OLS} = ||Y - X^T\beta||^2$$

where closed form solution exists as

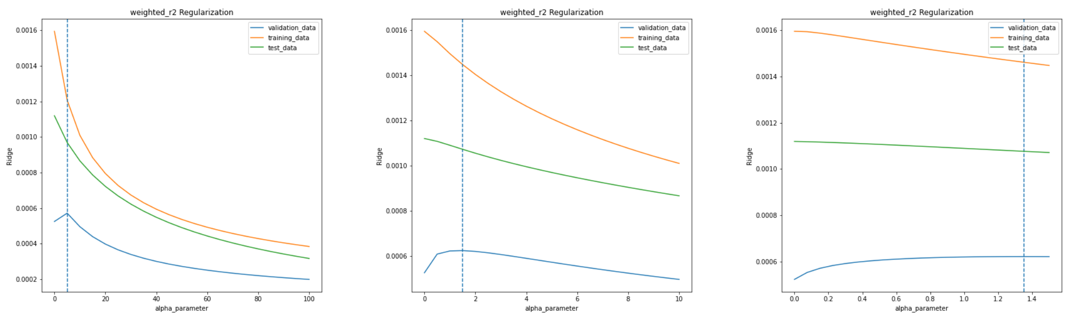
$$\hat{\beta} = (X^TX)^{-1}X^TY$$

3.1.1.1 Feature importance

Feature importance in ordinary linear regression is shown in figure 12. Gini Importance or Mean Decrease in Impurity (MDI) calculates each feature importance as the sum over the number of splits (across all tress) that include the feature, proportionally to the number of samples it splits.

3.1.1.2 Performance

The performance of OLS is given in table 1 below, where R^2 is weighted by 1/estVol, which uses 14-17 to train, the new hold-out sample 18 to test.



(a) α from 0 to 100 (b) α from 0 to 10 (c) α from 0 to 1.5

Figure 13: Cross validation results of Ridge regression

3.1.2 Ridge regression

Ridge regression was developed as a possible solution to the imprecision of least square estimators when linear regression models have some multicollinear independent variables—by creating a ridge regression estimator. This provides a more precise ridge parameters estimate, as its variance and mean square estimator are often smaller than the least square estimators previously derived. Specifically, Ridge regression adds a L_2 penalty to the loss function of OLS.

The loss function of Ridge Regression is

$$\mathcal{L}_{Ridge} = ||Y - X^T\beta||^2 + \alpha||\beta||^2$$

where closed form solution exists as

$$\hat{\beta} = (X^T X + \alpha I)^{-1} X^T Y$$

3.1.2.1 Hyperparameter tuning

Ridge regression has a hyperparameter α which needs to be fine tuned via cross validation.

Figure 13 includes the plots of Ridge loss function versus α. First, we do cross validation for α from 0 to 100, then granular search for α is implemented for α from 0 to 10 and 0 to 1.

The best α chosen from cross validation is 1.35.

3.1.2.2 Feature importance

Feature importance in Ridge regression is shown in figure 14. Gini Importance or Mean Decrease in Impurity (MDI) calculates each feature importance as the sum over the number of splits (across all tress) that include the feature, proportionally to the number of samples it splits.

3.1.2.3 Performance

The performance of Ridge regression is given in table 2 below, where R^2 is weighted by 1/estVol, which uses 14-17 to train, the new hold-out sample 18 to test.

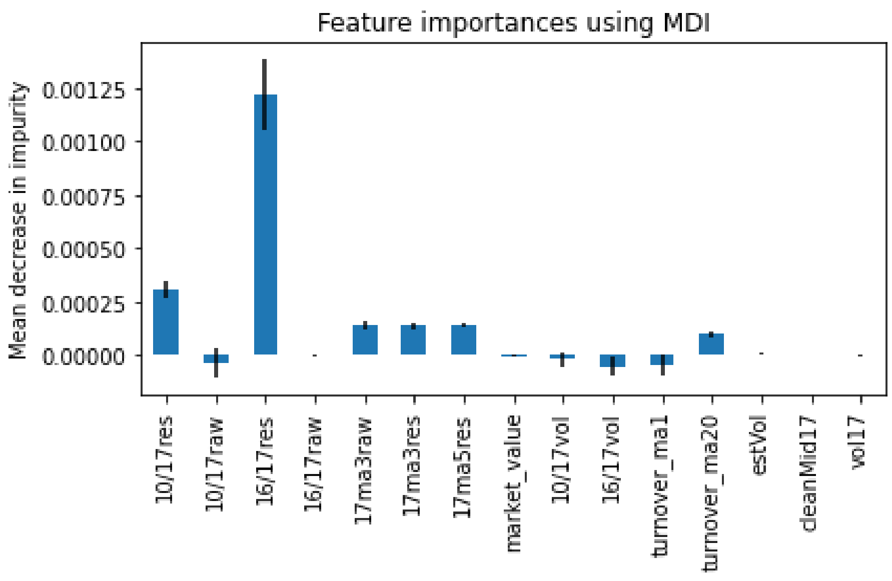


Figure 14: Permutation feature importance of Ridge regression

Table 2: Performance of Ridge regression

Model name	CV R^2 (bps)	18 R^2_{test} (bps)
Ridge (15 features)	6.2	10.8

3.2 Gradient boosting

Gradient boosting is a method that goes through cycles to iteratively add models into an ensemble. It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. Then, we start the cycle:

- First, we use the current ensemble to generate predictions for each observation in the dataset. To make a prediction, we add the predictions from all models in the ensemble.
- These predictions are used to calculate a loss function.
- Then, we use the loss function to fit a new model that will be added to the ensemble. Specifically, we determine model parameters so that adding this new model to the ensemble will reduce the loss. The "gradient" in "gradient boosting" refers to the fact that we'll use gradient descent on the loss function to determine the parameters in this new model.
- Finally, we add the new model to ensemble, and repeat!

Here, in this project, we implemented three gradient boosting models, i.e. XGBoost, LightGBM and Extra trees. Their implementation and performance are presented in Section 3.2.1, 3.2.2, 3.2.3, respectively.

3.2.1 XGBoost

XGBoost stands for extreme gradient boosting, which is an implementation of gradient boosting with several additional features focused on performance and speed. It is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

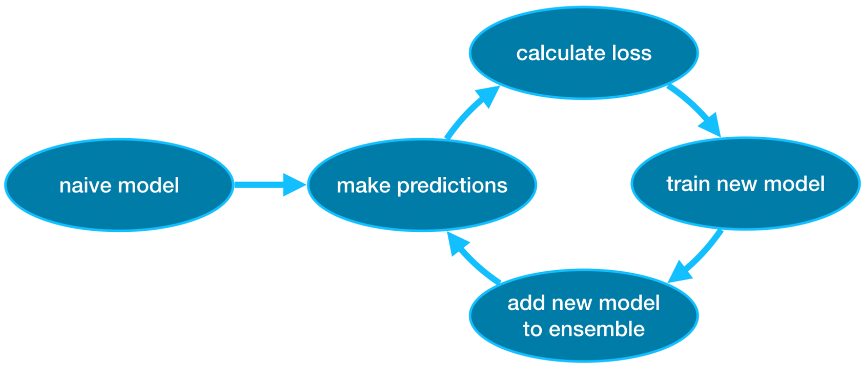


Figure 15: Gradient boosting chart

3.2.1.1 Hyperparameter tuning

XGBoost has a few parameters that can dramatically affect accuracy and training speed. Important features we looked at includes max_depth, min_child_weight, n_estimators, gamma, learning_rate, early_stopping_rounds.

The interpretation of each parameter and its candidate values for cross validation are listed below.

- max_depth [4,5,6]

  The maximum depth of a tree, which is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.

- min_child_weight [4,5]

  Defines the minimum sum of weights of all observations required in a child, which is used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree, while too high values can lead to under-fitting.

- n_estimators [100, 150]

  specifies how many times to go through the modeling cycle described above. It is equal to the number of models that we include in the ensemble.

  - Too low a value causes underfitting, which leads to inaccurate predictions on both training data and test data.
  - Too high a value causes overfitting, which causes accurate predictions on training data, but inaccurate predictions on test data (which is what we care about).

- gamma [0, 0.1]

  A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split, which makes the algorithm conservative.

- learning_rate [0.05, 0.1]

  Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number, known as the learning rate, before adding them in.

  In general, a small learning rate and large number of estimators will yield more accurate XGBoost models, though it will also take the model longer to train since it does more iterations through the cycle.

- early_stopping_rounds [0, 5, 20, 50]

  early_stopping_rounds offers a way to automatically find the ideal value for n_estimators. Early stopping causes the model to stop iterating when the validation score stops improving, even if we

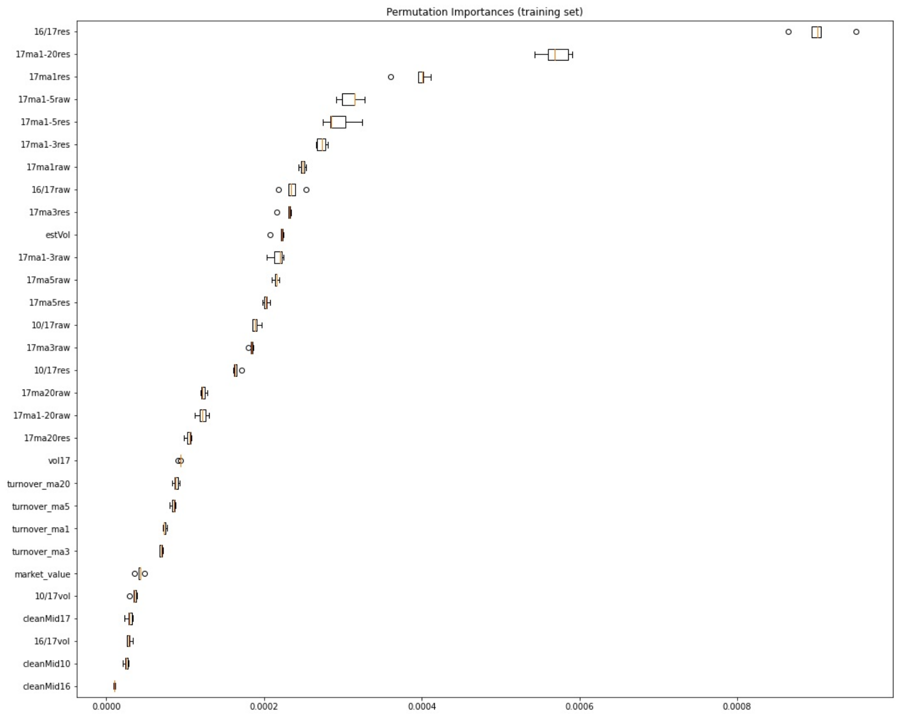


Figure 16: Permutation feature importance of XGBoost

> aren't at the hard stop for n_estimators. A good way to implement this is to set a high value for n_estimators and then use early_stopping_rounds to find the optimal time to stop iterating.
>
> Since random chance sometimes causes a single round where validation scores don't improve, you need to specify a number for how many rounds of straight deterioration to allow before stopping.
>
> When using early_stopping_rounds, you also need to set aside some data for calculating the validation scores - this is done by setting the eval_set parameter.

Table 3 shows the 1/ estVol weighted R^2 on different subsets with different hyperparameters listed above.

From it, we can see that the best parameters are best_paras: ('max_depth': 4, 'min_child_weight': 5, 'n_estimators': 100, 'gamma': 0, 'learning_rate': 0.1).

3.2.1.2 Feature importance

Feature importance in XGBoost is shown in figure 16.

3.2.1.3 Performance

The performance of XGBoost is given in table 4 below, where R^2 is weighted by 1/estVol, which uses 14-16 to train, 17 to validate, the new hold-out sample 18 to test. Here, validation set is needed because early stopping is used in model fitting

3.2.2 LightGBM

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

Table 3: Cross validation results of XGBoost

	max_depth	min_child_weight	n_estimators	gamma	learning_rate	R^2_{train}(bps)	R^2_{valid}(bps)
1	4	4	100	0	0.05	-613.9	-811.6
2	4	4	100	0	0.1	253.4	-0.8
3	4	4	100	0.1	0.05	-739.3	-814.7
4	4	4	100	0.1	0.1	-0.02	-0.8
5	4	4	150	0	0.05	188.5	0.9
6	4	4	150	0	0.1	362.9	-13.7
7	4	4	150	0.1	0.05	-4.2	-5.5
8	4	4	150	0.1	0.1	-0.02	-0.8
9	4	5	100	0	0.05	-615.1	-812.1
10	4	5	100	0	0.1	248.3	**1.6**
11	4	5	100	0.1	0.05	-739.3	-814.7
12	4	5	100	0.1	0.1	-0.02	-0.8
13	4	5	150	0	0.05	184.2	1.2
14	4	5	150	0	0.1	353.5	-10.3
15	4	5	150	0.1	0.05	-4.2	-5.5
16	4	5	150	0.1	0.1	-0.02	-0.8
17	5	4	100	0	0.05	-560.3	-813.9
18	5	4	100	0	0.1	382.6	-8.7
19	5	4	100	0.1	0.05	-739.3	-814.7
20	5	4	100	0.1	0.1	-0.02	-0.8
21	5	4	150	0	0.05	278.3	-0.2
22	5	4	150	0	0.1	559.0	-226.7
23	5	4	150	0.1	0.05	-4.2	-5.5
24	5	4	150	0.1	0.1	-0.02	-0.8
25	5	5	100	0	0.05	-561.8	-815.0
26	5	5	100	0	0.1	374.4	-4.4
27	5	5	100	0.1	0.05	-739.3	-814.7
28	5	5	100	0.1	0.1	-0.02	-0.8
29	5	5	150	0	0.05	276.4	-3.6
30	5	5	150	0	0.1	543.8	-20.9
31	5	5	150	0.1	0.05	-4.2	-5.5
32	5	5	150	0.1	0.1	-0.02	-0.8
33	6	4	100	0	0.05	-487.2	-814.5
34	6	4	100	0	0.1	560.2	-22.3
35	6	4	100	0.1	0.05	-739.3	-814.7
36	6	4	100	0.1	0.1	-0.02	-0.8
37	6	4	150	0	0.05	408.6	-3.6
38	6	4	150	0	0.1	812.6	-48.9
39	6	4	150	0.1	0.05	-4.2	-5.5
40	6	4	150	0.1	0.1	-0.02	-0.8
41	6	5	100	0	0.05	-497.2	-819.7
42	6	5	100	0	0.1	546.1	-16.4
43	6	5	100	0.1	0.05	-739.3	-814.7
44	6	5	100	0.1	0.1	-0.02	-0.8
45	6	5	150	0	0.05	397.9	-9.4
46	6	5	150	0	0.1	793.3	-40.9
47	6	5	150	0.1	0.05	-4.2	-5.5
48	6	5	150	0.1	0.1	-0.02	-0.8

Table 4: Performance of XGBoost

Model name	CV R^2 (bps)	14-16 R^2_{train} (bps)	17 R^2_{valid} (bps)	18 R^2_{test} (bps)
XGBoost (30 features)	1.6	164.5	1.4	2.3

- Faster training speed and higher efficiency
- Lower memory usage
- Better accuracy
- Support of parallel, distributed, and GPU learning
- Capable of handling large-scale data.

In XGBoost, trees grow depth-wise while in LightGBM, trees grow leaf-wise which is the fundamental difference between the two frameworks. Meanwhile, the implementations of these two models are similar to each other. Therefore, in the following sections, we only listed the important and different steps in LightGBM compared to XGBoost while omitted the rest of them.

3.2.2.1 Hyperparameter tuning

LightGBM has a few parameters that can dramatically affect accuracy and training speed. Important features we looked at includes num_leaves, max_depth, learning_rate, feature_fraction, bagging_fraction, early_stopping_rounds.

The interpretation of each parameter and its candidate values for cross validation are listed below.

- num_leaves [24, 80]

  This is the main parameter to control the complexity of the tree model. Theoretically, we can set num_leaves = 2ˆ(max_depth) to obtain the same number of leaves as depth-wise tree. However, this simple conversion is not good in practice. The reason is that a leaf-wise tree is typically much deeper than a depth-wise tree for a fixed number of leaves. Unconstrained depth can induce over-fitting. Thus, when trying to tune the num_leaves, we should let it be smaller than 2ˆ(max_depth).

- max_depth [5, 30]

  The maximum depth of a tree, which is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.

- learning_rate [0.05, 0.1]

  Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number, known as the learning rate, before adding them in.

- feature_fraction [0.1, 0.9]

  By default, LightGBM considers all features in a dataset during the training process. This behavior can be changed by setting feature_fraction to a value $\in [0, 1]$. Setting feature_fraction to 0.5, for example, tells LightGBM to randomly select 50% of features at the beginning of constructing each tree. This reduces the total number of splits that have to be evaluated to add each tree node. Decreasing feature_fraction also helps to reduce training time.

- bagging_fraction [0.8, 1]

  By default, LightGBM uses all observations in the training data for each iteration. It is possible to instead tell LightGBM to randomly sample the training data. This process of training over multiple random samples without replacement is called "bagging". Set bagging_fraction to a value $\in [0, 1]$ to control the size of the sample. Decreasing bagging$\in [0, 1]$ fraction also helps to reduce training time.

- early_stopping_rounds [5, 50]

  early_stopping_rounds offers a way to automatically find the ideal value for n_estimators, which avoids overfitting.

Table 5: Cross validation results of LightGBM

	learning_rate	num_leaves	feature_fraction	bagging_fraction	max_depth	R^2_{train}(bps)	R^2_{valid}(bps)
1	0.05	24	0.1	0.8	5	226.1	**-2.9**
2	0.05	24	0.1	0.8	30	262.1	-4.3
3	0.05	24	0.1	1	5	226.1	**-2.9**
4	0.05	24	0.1	1	30	262.1	-4.3
5	0.05	24	0.9	0.8	5	346.3	-22.1
6	0.05	24	0.9	0.8	30	395.9	-19.8
7	0.05	24	0.9	1	5	346.3	-22.1
8	0.05	24	0.9	1	30	395.9	-19.8
9	0.05	80	0.1	0.8	5	236.8	-3.4
10	0.05	80	0.1	0.8	30	698.0	-14.0
11	0.05	80	0.1	1	5	236.8	-3.4
12	0.05	80	0.1	1	30	698.0	-14.0
13	0.05	80	0.9	0.8	5	353.9	-20.4
14	0.05	80	0.9	0.8	30	1084.2	-62.0
15	0.05	80	0.9	1	5	353.9	-20.4
16	0.05	80	0.9	1	30	1084.2	-62.0
17	0.1	24	0.1	0.8	5	389.2	-33.6
18	0.1	24	0.1	0.8	30	454.0	-31.5
19	0.1	24	0.1	1	5	389.2	-33.6
20	0.1	24	0.1	1	30	454.0	-31.5
21	0.1	24	0.9	0.8	5	592.3	-57.0
22	0.1	24	0.9	0.8	30	690.7	-54.4
23	0.1	24	0.9	1	5	592.3	-57.0
24	0.1	24	0.9	1	30	690.7	-54.4
25	0.1	80	0.1	0.8	5	409.0	-39.1
26	0.1	80	0.1	0.8	30	1204.7	-90.8
27	0.1	80	0.1	1	5	409.0	-39.1
28	0.1	80	0.1	1	30	1204.7	-90.8
29	0.1	80	0.9	0.8	5	617.2	-59.0
30	0.1	80	0.9	0.8	30	1829.6	-132.3
31	0.1	80	0.9	1	5	617.2	-59.0
32	0.1	80	0.9	1	30	1829.6	-132.3

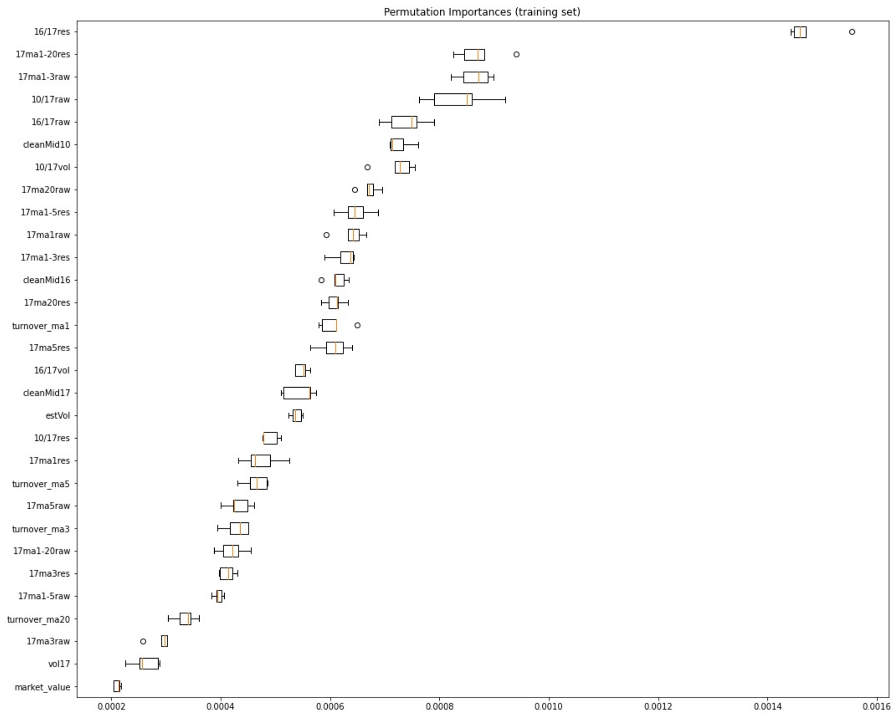


Figure 17: Permutation feature importance of LightGBM

Table 6: Performance of LightGBM

Model name	CV R^2 (bps)	14-16 R^2_{train} (bps)	17 R^2_{valid} (bps)	18 R^2_{test} (bps)
LightGBM (30 features)	-2.9	66.3	37.4	12.3

Table 5 shows the 1/ estVol weighted R^2 on different subsets with different hyperparameters listed above.

From it, we can see the best parameters are: ('learning_rate': 0.05, 'num_leaves': 24, 'feature_fraction': 0.1, 'bagging_fraction': [0.8, 1], 'max_depth': 5, eval_metric: [logloss, mae, rmsle, mphe], early_stopping_rounds: 0).

3.2.2.2 Feature importance

Feature importance in LightGBM is shown in figure 3.2.2.2.

3.2.2.3 Performance

The performance of LightGBM is given in table 6 below, where R^2 is weighted by 1/estVol, where we use 14-16 to train, 17 to validate, the new hold-out sample 18 to test. Here, validation set is needed because early stopping is used in model fitting.

3.2.3 Extra trees

Extra Trees is an ensemble machine learning algorithm that combines the predictions from many decision trees. It is related to the widely used random forest algorithm. It can often achieve as-good or better performance than the random forest algorithm, although it uses a simpler algorithm to construct the decision trees used as members of the ensemble. It is also easy to use given that it has few key hyperparameters and sensible heuristics for configuring these hyperparameters.

3.2.3.1 Hyperparameter tuning

Below are the interpretation and candidate values for important hyperparameters in Extra Trees regressor, which includes n_estimators, min_samples_leaf, min_samples_split, criterion, max_depth.

The interpretation of each parameter and its candidate values for cross validation are listed below.

- n_estimators [50, 100, 150]

  specifies how many times to go through the modeling cycle described above. It is equal to the number of models that we include in the ensemble.

  - Too low a value causes underfitting, which leads to inaccurate predictions on both training data and test data.
  - Too high a value causes overfitting, which causes accurate predictions on training data, but inaccurate predictions on test data (which is what we care about).

- min_samples_leaf [3,5]

  The minimum number of samples required to split an internal node.

- min_samples_split [2,4]

  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- criterion ['mse', 'mae']

  Different loss function used in model fitting.

- max_depth [8,32]

  The maximum depth of a tree, which is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.

Table 7 shows the 1/ estVol weighted R^2 on different subsets with different hyperparameters listed above.

From it, we can see that there are two best parameter groups, which are best_paras: ('n_estimators': 150, 'min_samples_leaf': 5, 'min_samples_split': 4, 'max_depth': 8).

3.2.3.2 Feature importance

Feature importance in Extra Trees is shown in figure 18.

3.2.3.3 Performance

The performance of Extra trees is given in table 8 below, where R^2 is weighted by 1/estVol, which uses 14-17 to train and validate, 18 to test.

4 Summary

Overall, we used five models, which includes two regression models, ordinary linear regression, Ridge regression; and three gradient boosting models, XGBoost, LightGBM and Extra Trees.

While regression models have simpler implementations and fewer hyperparameters to fine tune, gradient boosting methods yield more complicated relationships between features and target variables, thus more likely a greater performance.

Table 7: Cross validation results of Extra Trees

	n_estimators	min_samples_leaf	min_samples_split	max_depth	R^2_{train}(bps)	R^2_{valid}(bps)
1	50	3	2	8	91.7	11.8
2	50	3	2	32	4750.9	-76.6
3	50	3	4	8	93.4	12.9
4	50	3	4	32	4723.3	-75.6
5	50	5	2	8	87.3	14.3
6	50	5	2	32	36.6	-52.7
7	50	5	4	8	86.4	13.4
8	50	5	4	32	36.7	-55.5
9	100	3	2	8	93.1	13.7
10	100	3	2	32	4731.8	-47.5
11	100	3	4	8	92.0	13.5
12	100	3	4	32	4923.7	-48.1
13	100	5	2	8	88.2	14.5
14	100	5	2	32	37.0	-29.7
15	100	5	4	8	89.3	14.0
16	100	5	4	32	37.3	-31.0
17	150	3	2	8	91.6	13.5
18	150	3	2	32	4700.4	-36.7
19	150	3	4	8	92.4	13.5
20	150	3	4	32	47.2	-38.1
21	150	5	2	8	88.5	14.3
22	150	5	2	32	37.1	-24.2
23	150	5	4	8	89.3	**15.0**
24	150	5	4	32	3727.9	-23.8

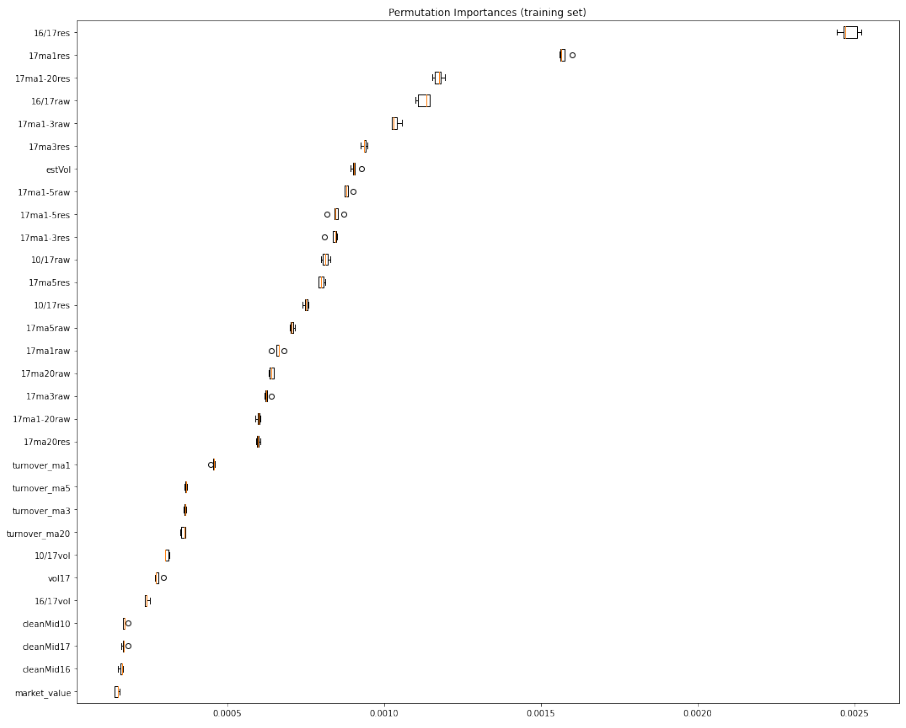


Figure 18: Permutation feature importance of Extra Trees

Table 8: Performance of Extra trees

Model name	CV R^2 (bps)	14-17 R^2_{train} (bps)	18 R^2_{test} (bps)
Extra trees (30 features)	15	54.8	11.3

Table 9: Model information

Model name	Parameters
Ridge	$\alpha = 1.35$
XGBoost	max_depth: 5,min_child_weight:5, n_estimators: 100, gamma: 0,learning_rate: 0.1
LightGBM	learning_rate: 0.05,num_leaves: 24,feature_fraction: 0.1, bagging_fraction: [0.8, 1], max_depth: 5, early_stopping_rounds: 0
Extra Trees	n_estimators: 150, min_samples_leaf: 5, min_samples_split: 4,max_depth: 8

Table 10: Performance of different models (measured in bps)

Model name	CV R^2	18 R^2_{test}
OLS (15 features)	NaN	11.2
Ridge (15 features)	6.2	10.8
	CV R^2	18 R^2_{test}
XGBoost (30 features)	1.6	2.3
LightGBM (30 features)	-2.9	12.3
Extra Trees (30 features)	15	11.3

The best parameters and weighted R^2 are presented below.

4.1 Parameters

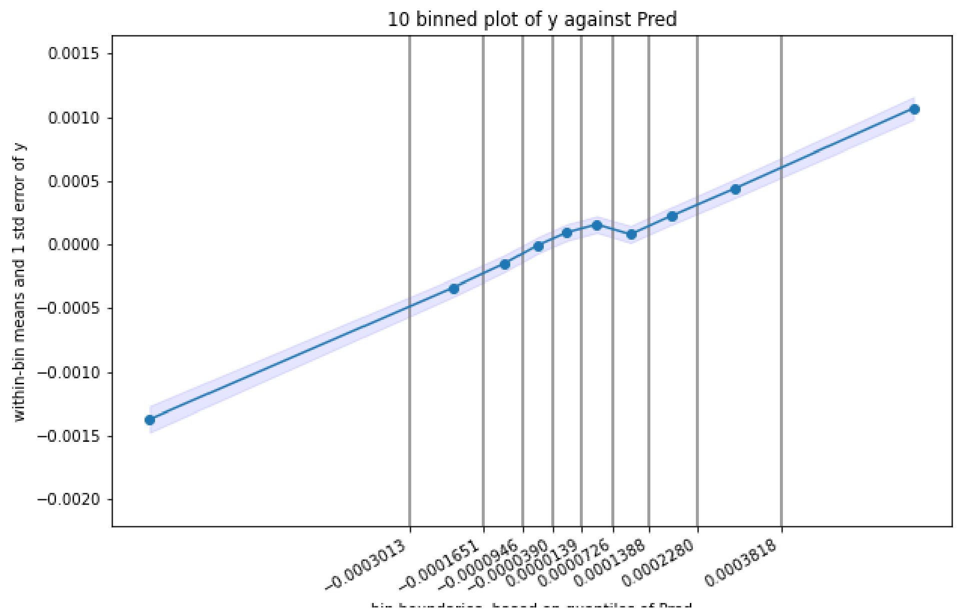
The best hyperparameters validated via cross validation for each model are listed below in table 9.

4.2 Performance

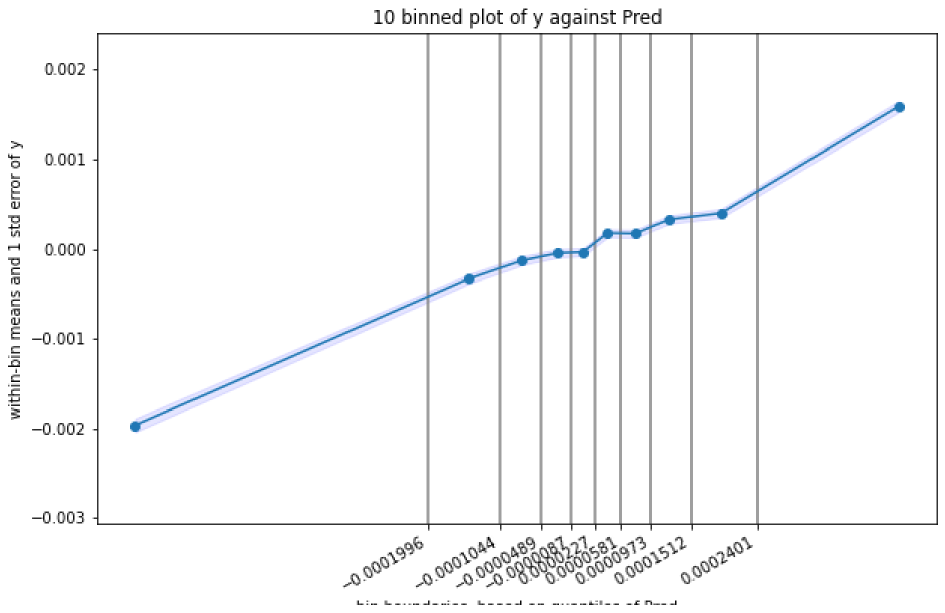
The performance of these models with the best hyperparameters are summarized below in table 10.

4.3 Plots

Finally, we present the bin plot, equal-weighted and weighted means drift plots and the 30-day moving average of correlation of the predictions versus the target variable of the two best performed model, i.e. LightGBM and Extra Trees.

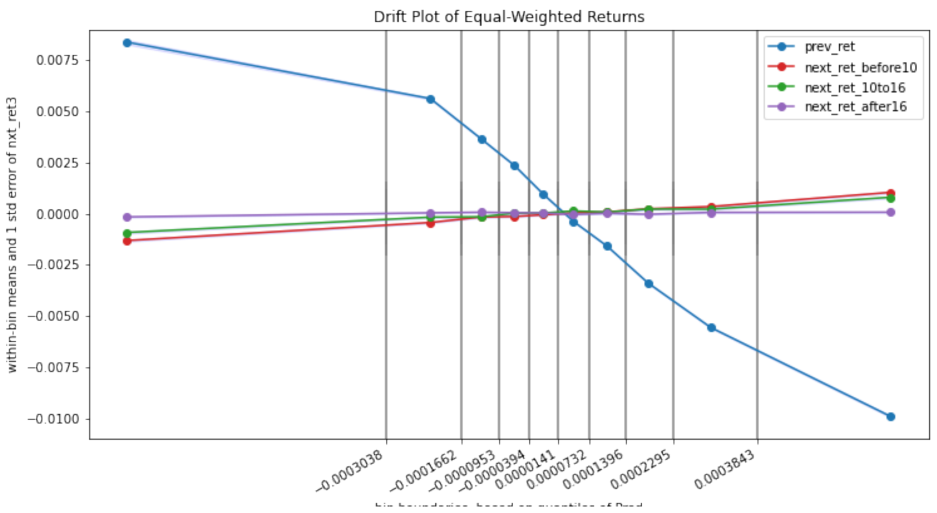


(a) LightGBM

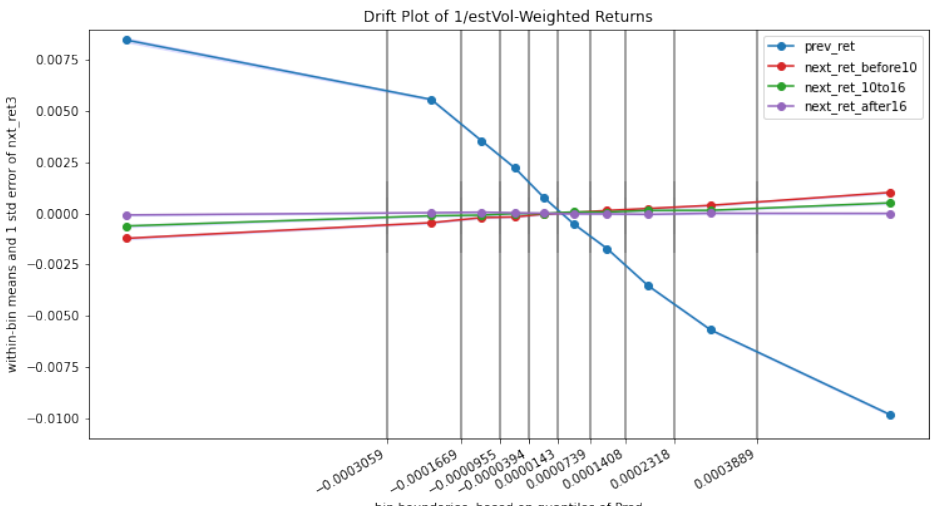


(b) Extra trees

Figure 19: Binplot

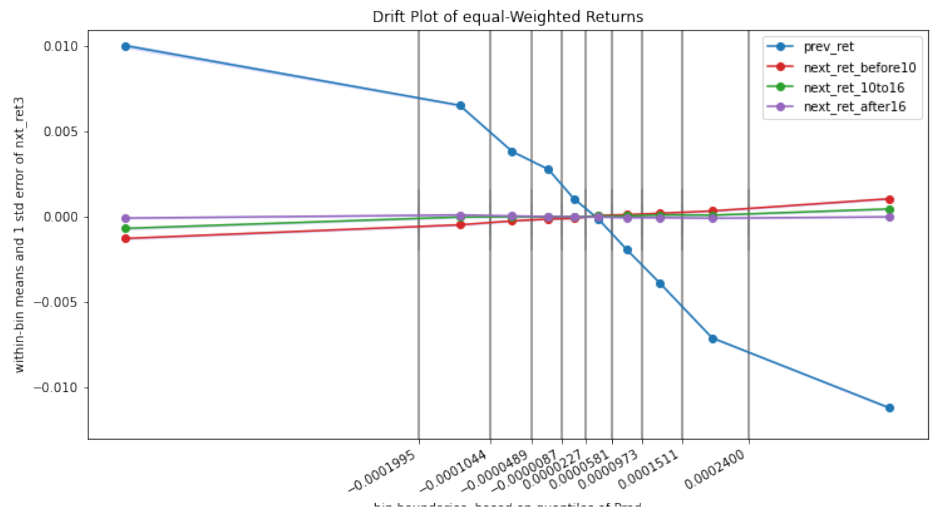


(a) Equal-weighted means

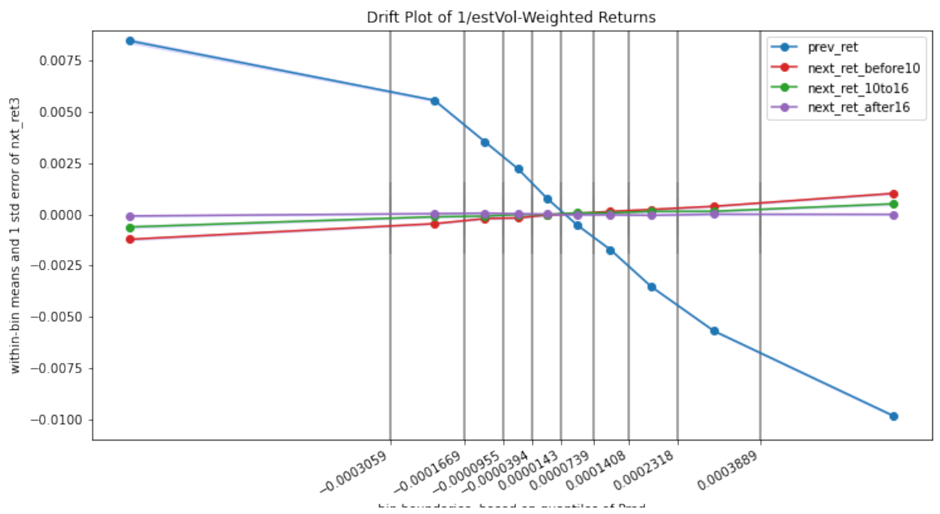


(b) Weighted means

Figure 20: Drift plots of LightGBM (Error bar is included but too small to be noticed)

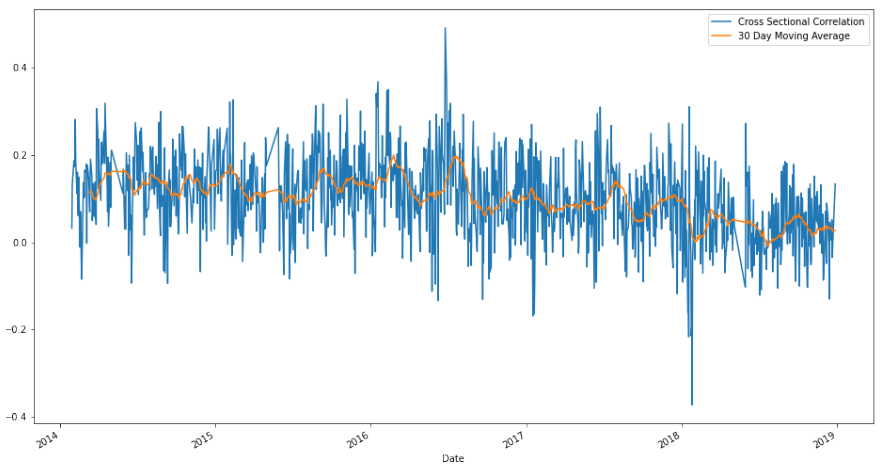


(a) Equal-weighted means

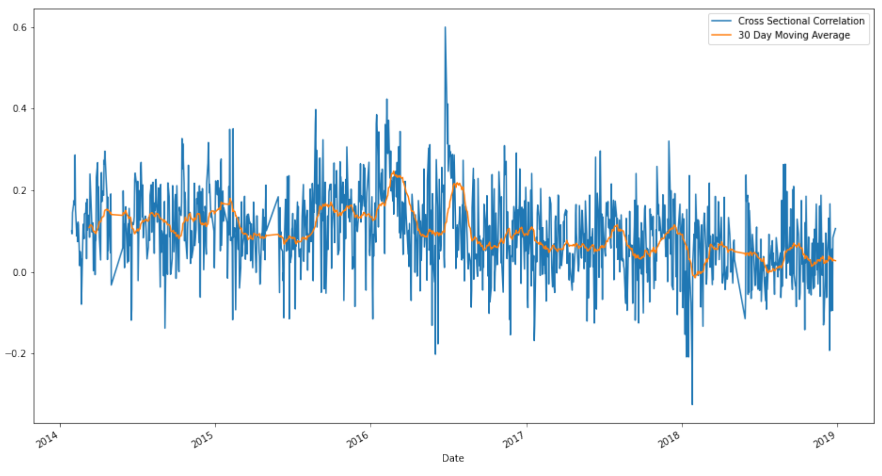


(b) Weighted means

Figure 21: Drift plots of Extra trees (Error bar is included but too small to be noticed)



(a) LightGBM



(b) Extra trees

Figure 22: 30-day moving average of correlation