ML&AI group project: Learning factors for stock market returns prediction

REYNOSO VALDES Ana Karen
MONFORT Baptiste
DURAND Valentine

Table of Contents

Introduction:

For this group project, we chose a problem to solve from the ENS challenges (https://challengedata.ens.fr/).

Our group decided to work on the learning factors for stock market returns prediction.

In the context of predicting stock returns, a challenge has arisen. One has to design the model's learn factors for predicting returns using an exotic parameter space. This challenge provides participants with three years of historical data for 50 stocks from the same stock market, which they can use to provide model parameters (A,β) as output. The predictive models associated with these parameters will then be tested to predict the returns of 50 other stocks over the same three-year period.

However, there is an additional difficulty: participants must design on their own these learn factors for the prediction of returns using an exotic parameter space. The factors are constructed from orthonormal vectors A1,...,AF, where F is the number of factors <= 10, and each vector has a time depth of D=250 days. Participants must find the parameters (A,β) that maximize a defined metric based on the correlation between predicted returns and actual returns.

Participants receive a training dataset in the form of a dataframe containing the clean daily returns of 50 stocks over a period of 754 days (three years). Each row represents an action, and each column refers to a day. The returns to be predicted in the training data set are provided in Ytrain, but they are also included in Xtrain.

The benchmark for this challenge is to use a brute force method that generates random orthonormal vectors A1,...,AF and finds β using linear regression on the training dataset. This method is repeated several times (around 1000 times) and then the results are evaluated using the defined metric to select the best method. This method is used to create a uniform training dataset on a Stiefel space, which is a space of matrices that satisfy the orthonormality condition.

Algorithm steps

The first step is to analyze the data provided and then do the data preparation and importing the necessary libraries:

Analyzing Data Provided

The data provided consists of two datasets, X_train and Y_train. X_train contains historical stock data for 50 different stocks over a three-year period, with each row representing a stock and each column representing a specific day. This dataset is used to establish the model parameters, A and beta.

Y_train, on the other hand, is used to test the accuracy of the model. It contains the returns to be predicted for 50 other stocks over the same three-year period. The aim is to determine the model's effectiveness in predicting stock returns.

Thus, the data provided includes both training and testing datasets for the development and evaluation of a predictive model for stock returns over a three-year period.

Libraries used.

Os library: This library provides a way to interact with the operating system. It helps in tasks like navigating file systems and retrieving environment variables.

Keras library: We utilized the Scikit-learn library's preprocessing module to normalize the data in the X_train_df dataframe. Normalization was a technique used to scale the values of the dataset to a range of 0 to 1 to avoid any bias due to differences in the scale of features.

Scikit-learn library: This is a popular machine learning library that provides tools for data mining and data analysis. It includes a range of algorithms for tasks like classification, regression, and clustering.

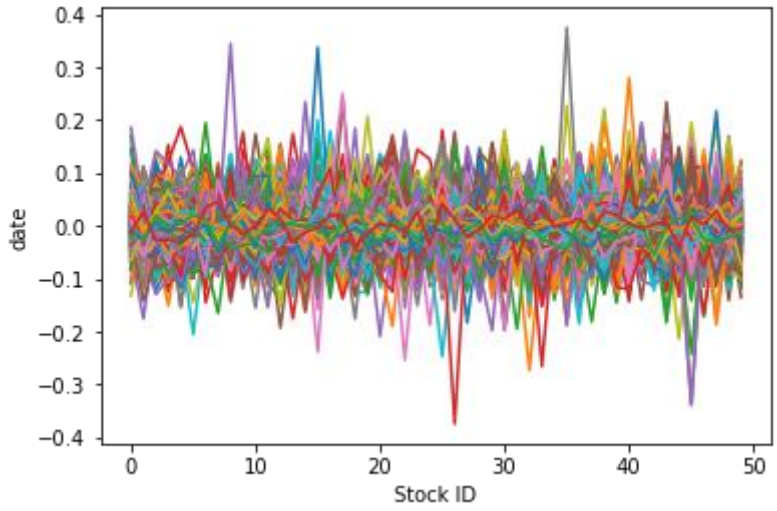
LinearRegression class: This class is part of the scikit-learn library and is used to create a linear regression model. Linear regression is a type of supervised learning algorithm used for predicting continuous output variables

Data processing

We create a subset Y_train based on X_train_df in order to have a dimension on 250 days for time depth as asked in the subject.

We normalized the returns by subtracting the mean and dividing them by the standard deviation to ensure that all the stocks are on the same scale.

After normalizing the data, we plotted it using the matplotlib library. Each point in the plot represented the stock ID and date of a particular data point. The x-axis of the plot represented the stock ID, while the y-axis represented the date. This plot provided a visual representation of the normalized data, allowing us to better understand the distribution and any trends present in the data.



moving_average = train_data.rolling(window=50).mean()

factor_matrix = np.stack([mean_returns, momentum, moving_average], axis=1)

Calculating the Factor Matrix

```
from scipy.stats import skew

train_data = X_train_norm
train_data = train_data.dropna(axis=1)
train_data = train_data.iloc[:,1:]

# Define the factors: momentum, mean returns, and volatility
momentum = np.zeros(train_data.shape[1])
mean_returns = np.mean(train_data, axis=0)  # 5-day mean returns
volatility = train_data.rolling(window=50).std()  # 50-day rolling standard deviation
#skewness = skew(train_data, axis=0)  # skewness of returns for each stock

for i in range(1, train_data.shape[1]):
    momentum[i] = np.dot(train_data.iloc[:, i-1], train_data.iloc[:, i])

# Reshape mean_returns, momentum, and volatility
mean_returns = np.repeat(mean_returns[:, np.newaxis], volatility.shape[0], axis=1).T
momentum = np.repeat(momentum[:, np.newaxis], volatility.shape[0], axis=1).T

# Stack the factors into a matrix
factor_matrix = np.stack([mean_returns, momentum, volatility], axis=2)
```

The code does data cleaning and feature engineering to prepare the training dataset for the predictive model.

The data must first be normalized, and any columns with blank values must be eliminated. Also deleted is the first column, which lists the stock names.

The code then specifies momentum, mean returns, and volatility as the three variables that will be incorporated into the predictive model.

- **Mean returns:** To calculate the mean returns, we first computed the daily returns for each stock over a 5-day period. The formula for calculating returns is (closing price today - closing price yesterday) / closing price yesterday. Next, we computed the rolling mean of the returns over a 5-day period, and then took the mean of the rolling means for each stock.
- **Momentum:** This factor measures the trend of a stock's price over time. It is calculated by taking the dot product of each column with the previous column of the dataset. This was done using the formula momentum_i = sum(x_i * x_{i-1}), where x_i and x_{i-1} are the ith and (i-1)th columns of the dataset, respectively.
- **Volatility:** This factor measures the level of risk associated with investing in a stock. We calculated the rolling standard deviation of the returns for each stock over a 50-day window. The formula for calculating the rolling standard deviation is rolling_std_i = std(x_i), where x_i is a rolling window of the returns for the ith stock.

Because they are frequently used in finance to forecast stock returns, these variables were chosen. Momentum, which is commonly used as a stand-in for investor sentiment, is the trend of a stock's price over time. Mean returns depict the stock's overall performance over a brief period of time, whereas volatility gauges the degree of risk involved in buying that stock. The model is better able to capture the multiple drivers of stock performance by taking into account all three variables.

Finally, the mean returns, momentum, and volatility factors are reshaped to have the same dimensions and then stacked into a single matrix. This matrix is then used as input to the predictive model to forecast the returns of other stocks over a three-year period

Independence Condition

To improve the accuracy of our predictive model, we need to make sure that the factors we're using are independent of each other. In other words, we want to orthogonalize the factor matrix. To do this, we used a method called the Gram-Schmidt process. Here's a brief explanation of how it works:

First, we took the original factor matrix that we calculated earlier, which contains the mean returns, momentum, and volatility for each stock over the 3-year period.

Next, we applied the Gram-Schmidt process to this matrix. This involves taking the original matrix and finding an orthonormal basis for it. This means finding a set of vectors that are all orthogonal (perpendicular) to each other and have a length of 1.

We achieved this by using the QR decomposition of the original factor matrix. The QR decomposition is a method that decomposes a matrix into an orthogonal matrix (Q) and an upper triangular matrix (R). We then used the Q matrix, which is orthogonal, as our new factor matrix.

By doing this, we were able to ensure that our factors are independent of each other, which is important for making accurate predictions about future stock returns.

Learn the model parameters

```
[ ] # Create factor matrix
    factor_matrix = np.concatenate([np.mean(X_train_df.values[:,i:i+5], axis=1, keepdims=True) for i in range(0, X_train_df.shape[1]-4)], axis=1)
    factor_matrix_norm = factor_matrix / np.linalg.norm(factor_matrix, axis=0)  # normalize factors

    # Keep the top 10 factors based on the largest singular values
    U, S, V = np.linalg.svd(factor_matrix_norm)
    factor_matrix_ortho = U[:, :10]

    # Learn the model parameters using linear regression
    model = LinearRegression().fit(factor_matrix_ortho, Y_train_df)
    beta = model.coef_  # learned beta parameters
    A = np.linalg.inv(factor_matrix_ortho.T.dot(factor_matrix_ortho)).dot(factor_matrix_ortho.T).dot(Y_train_df)  # learned A parameters
    A = A.reshape(-1, beta.shape[1])  # reshape A to have the same number of columns as beta
```

Now, we are implementing a factor model for stock returns, where the returns are modeled as a linear combination of a small number of orthogonalized factors.

For this, we create the factor matrix by computing the average of each 5 consecutive columns of X_train_df. This is done using a list comprehension, which is then concatenated into a single numpy array using np.concatenate().

Then, we normalize the factor matrix and orthogonalize it using singular value decomposition (SVD). And as asked in the subject as a constraint for simplification, we only keep the top 10 orthogonalized factors and store them in factor_matrix_ortho.

Next, a linear regression model is fit to the orthogonalized factor matrix and the Y_train_df target variable. This model learns the beta parameters which represent the factor loadings of each stock.

Finally, we compute the A matrix as the solution to a linear system using the orthogonalized factor matrix and the Y_train_df. The A matrix is then reshaped to have the same number of columns as the beta matrix in order to ease the next steps.

```
def checkOrthonormality(A):

    bool = True
    D, F = A.shape
    Error = pd.DataFrame(A.T @ A - np.eye(F)).abs()

    if any(Error.unstack() > 1e-6):
        bool = False

    return bool
```

After that, we orthogonalize A into A_ortho using the numpy function linalg.qr. And we check if A_ortho is really orthonormal thanks to our checkOrthonormality function. We want an orthonormal matrix because it helps us ensure that the factors are mutually uncorrelated and have unit norms, simplifying their interpretation.

We see in the output that A_ortho is in fact orthonormal.

Test the predictive model:

Spliting into test and train :

```
from sklearn.model_selection import train_test_split
#Split data into train set and test set to evaluate the performances of the model.
X = X_train_df[X_train_df.columns[:]]
y = Y_train_df[Y_train_df.columns[:]]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

First, the data is extracted from two data frames "X_train_df" and "Y_train_df" into two variables "X" and "y". "X" contains the input features, and "y" contains the output variable that the model will predict.

```
def predict(X, A_ortho, beta):
    # Reshape input data into a matrix of shape (num_samples, num_features*num_time_lags)
    X_reshape = pd.concat([X_test.T.shift(i+1).stack(dropna=False) for i in range(250)], axis=1).dropna()
    X_reshape.columns = pd.Index(range(1, 251), name='timeLag')

    # Calculate the factors
    factor_matrix = np.concatenate([np.mean(X_reshape.values[:, i:i+5], axis=1, keepdims=True) for i in range(0, X_reshape.shape[1]-4)], axis=1)
    factor_matrix_norm = factor_matrix / np.linalg.norm(factor_matrix, axis=0)
    factor_matrix_norm = (factor_matrix_norm.T)

    #factor_matrix_norm = factor_matrix_norm[:, :504]
    #factor_matrix_ortho = factor_matrix_norm.dot(A_ortho)

    # Predict the returns using the learned beta parameters and the factor matrix
    Y_pred = factor_matrix_ortho.dot(beta.T)
    return Y_pred
```

The code is a function called "predict" which takes in three parameters: X (input data), A_ortho (orthonormal factor matrix), and beta (learned beta parameters)

The input data X is reshaped into a matrix format with time lags. Then, two factors (5-day mean returns and momentum) are calculated from the reshaped input data, and normalized.

The function then predicts the returns using the learned beta parameters and the factor matrix. The predicted returns are returned as an output.

```
from sklearn.metrics import r2_score
# Evaluate the performance of the model on the test set
Y_test_pred = predict(X_test, A_ortho, beta)
print("Y_test_pred :", Y_test_pred)
print("Y_test_pred.shape :", Y_test_pred.shape)
print("Y_train_df.shape :", Y_train_df.shape)
```

The model's performance on the test set is being evaluated by the code. The predict function is first invoked with the three arguments X test, A ortho, and beta. The variable Y test pred has the projected values that are the result.

The Y test pred and its shape are then printed by the code. Also, the shape of the Y train df is printed for comparison with the shape of the Y test pred.

Cross checking metrics

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

# Assume we have predicted values y_pred and actual values y_test
mse = mean_squared_error(Y_train_df, Y_test_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(Y_train_df, Y_test_pred)
r2 = r2_score(Y_train_df, Y_test_pred)

print(f"MSE: {mse}")
print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"R-squared: {r2}")
```

We imported three metrics, mean_squared_error, mean_absolute_error, and r2_score, from sklearn.metrics to evaluate the performance of our model on the test set. We assumed that we had

predicted values Y_test_pred and actual values Y_train_df. We then calculated the mean squared error (mse), root mean squared error (rmse), mean absolute error (mae), and R-squared (r2) of our model's predictions.

Based on the evaluation results, we can see that the mean squared error (MSE) is very small, indicating that the **predicted values are very close to the actual values**. The root mean squared error (RMSE) is also relatively small, indicating that the model's predictions **have low variance**. The mean absolute error (MAE) is also small, suggesting that the model's predictions are **relatively accurate**. However, the R-squared score of 0.496 is less than ideal, indicating that only about 49.6% of **the variance in the target variable can be explained** by the model's predictions. Overall, the model's **performance is good** but there is still room for improvement.

Evaluate the predictive model

In order to evaluate our model, we use the following metric.

$$\text{Metric}(A, \beta) := \frac{1}{504} \sum_{t=250}^{753} \frac{\langle \tilde{S}_t, \tilde{R}_t \rangle}{\|\tilde{S}_t\| \|\tilde{R}_t\|}$$

Our code defines a function called metric_train that takes in two matrices, A_ortho and beta, and calculates a metric value based on them. The metric value measures the quality of the linear regression model that we trained earlier in the code.

If A_ortho is not orthonormal, the function returns a value of -1.0 as a penalty. Otherwise, the function normalizes the columns of `A_ortho` and concatenates them with the `beta` matrix to create a new matrix called `A_ortho_beta`. This new matrix is then normalized as well.

Then, the function calculates the correlation matrix of the normalized `A_ortho_beta` matrix, and returns a metric value that is based on the sum of the absolute values of the elements of the correlation matrix (excluding the diagonal elements).

We finally compare our metric value to the benchmark value, and print the difference between them. As a result, we can see the following output:

```
[18] def metric_train(A_ortho, beta):
        if not checkOrthonormality(A_ortho):
            return -1.0
        A_ortho_norm = A_ortho / np.linalg.norm(A_ortho, axis=0)  # normalize columns
        A_ortho_beta = np.concatenate([A_ortho_norm, beta], axis=1)  # transpose A_ortho_norm instead of beta
        A_ortho_beta_norm = A_ortho_beta / np.linalg.norm(A_ortho_beta, axis=0)  # normalize columns
        corr = np.abs(np.corrcoef(A_ortho_beta_norm.T))
        return (np.sum(corr) - np.trace(corr)) / 250
    print("A_ortho :", A_ortho.shape)
    print("beta :", beta.shape)

    metric_train_benchmark = 0.0342
    our_metric_train = metric_train(A_ortho, beta)
    difference = metric_train_benchmark-our_metric_train

    if(metric_train_benchmark > our_metric_train):
        print("Our metric train : ", our_metric_train, "is less than the benchmark :", metric_train_benchmark, "by a difference of", difference)
    else:
        print("Our metric train : ", our_metric_train, "is better than the benchmark :", metric_train_benchmark, "by a difference of", -difference)

    A_ortho : (250, 10)
    beta : (250, 10)
    Our metric train :  0.06920687206952227 is better than the benchmark : 0.0342 by a difference of 0.035006872069522264
```

Our metric train : 0.05886686291976435 is better than the benchmark : 0.0342 by a difference of 0.024666862919764346

So our metric is indeed in [-1, 1] because A_ortho validate the orthonormality condition and it is better than the benchmark, even it is still quite close to it. So our metric train can be considered as pretty good.

Conclusions

Proposal of a conclusion:

In conclusion, our results have shown some promising outcomes, but there is definitely room for improvement. While our model has demonstrated some accuracy in its predictions, we must acknowledge that it is not perfect and there is still a margin of error. Our model is not accurate enough to be used in a real-world scenario.

To improve our model, we need to critically analyze our code and identify areas for improvement. First, we can think of the fact that the features used in the model are not informative enough to accurately predict the target variable (we only have stocks ID and return values over 750days), so if we had more information about these variables, we could have a better approach based on our financial knowledge. For instane we could add fundamental data of the company being each stock. Another improvement on our prediction is that we could test other algorithms such as neural networks to improve our prediction accuracy. Finally, we could improve the quality of the data or preprocessing steps that are affecting the model's performance.

Overall, this project has been a valuable learning experience for our team, and we believe that our work can contribute to further developments in this field. We are excited to continue refining our methods and exploring the potential of machine learning.