Make money with

HIREUS

Zelin Xu
Senyang Wu
Jiaguo Bei
Pinrui Pan

FACTORS

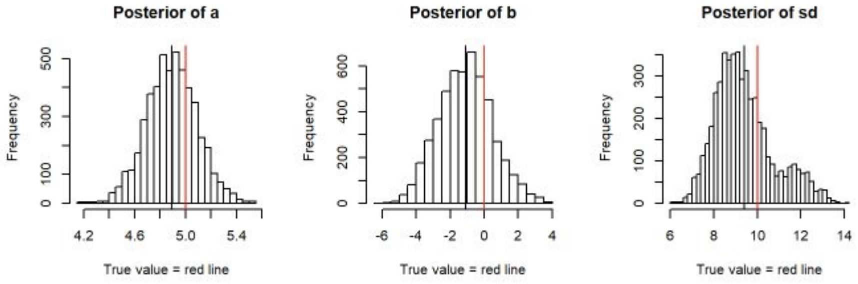
- Size Factor
- Price-Based Factors
  - Moving Average (Simple and Exponential)
  - Bollinger Band (BB)
  - Momentum
  - Beta
  - Relative Strength Index (RSI)
  - Stochastic Oscillator
  - Money Flow Index (MFI)
- Volume-Based Factor
  - Liquidity
  - Volume-ratio

FACTOR TEST

- Divide the universe into 5 groups every week based on the factor values and calculate the cumulative demeaned return for each group

How to get rank

PROBABILITIES



- Confidence interval (Bayesian approach: argmax β P(β|y))
- MC methods
- Ensemble learning (Random Forest)
- Multinomial Logistic Regression (Panel Data)

GET WEIGHTS OF THE PORTFOLIO

- Long assets that have maximum probability in rank 5 and short assets with maximum probability in rank 1

- Max $\quad$ Sharpe Ratio = (Rp − Rf) / σp

  st. $\quad$ Rp = Σ(wi · Ri)

  σp = sqrt(ΣΣwi*wj*σij)

  wi > 0 if i in long assets, wi<0 if i in short assets, else wi = 0

  abs(wi) <= 0.15

TIME SERIES MODELS

- Autoregressive models: AR, ARIMA, ARIMAX
- LSTM, ATTENTION models (softmax output)
- training/validation: 5 months/ 1 month, move forward weekly

return_i = f(x_i_t, x_i_t-1, x_i_t-2, ...x_i_t-T) (AR, ARIMA/ LSTM)

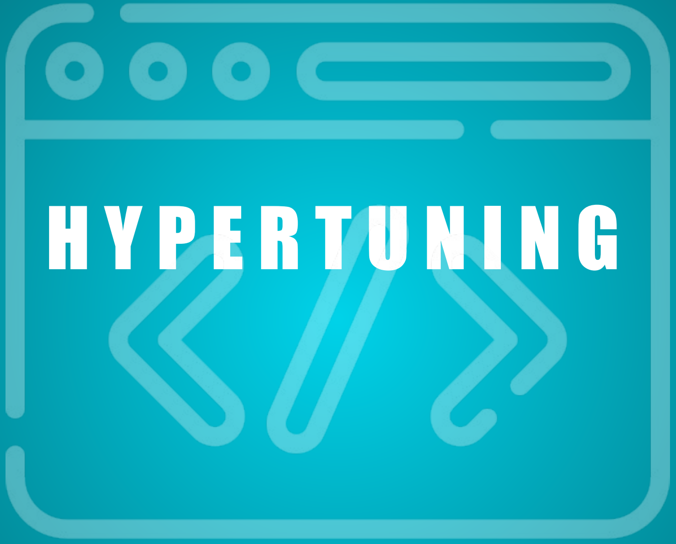
vs.

return_i = f(x_o_t, x_o_t-1..., x_i_t-T) (LSTM, ATTENTION, ARIMAX)

Vanilla LSTM

```
[16] class LSTMModel(tf.keras.Model):
        def __init__(self, lstm_units=50, lstm_depth=10, universe_size=110, num_ranks=5, dropout_rate=0.2):
            super(LSTMModel, self).__init__(name='rank_lstm_model')
            self.num_ranks = num_ranks
            # Initialize LSTM layers with specified depth, units, and return sequences
            self.lstm_layers = [layers.LSTM(lstm_units, return_sequences=True) for _ in range(lstm_depth - 1)]
            self.lstm_layers.append(layers.LSTM(lstm_units, return_sequences=False))

            # Initialize Dropout layers with specified dropout rate
            self.dropout_layers = [layers.Dropout(dropout_rate) for _ in range(lstm_depth)]

            # Initialize the final Dense layer with universe_size * num_ranks units
            self.final_Dense = layers.Dense(universe_size * num_ranks)

        def call(self, input_tensor, training=False):
            x = input_tensor
            for lstm_layer, dropout_layer in zip(self.lstm_layers, self.dropout_layers):
                x = lstm_layer(x)
                x = dropout_layer(x, training=training)
            x = self.final_Dense(x)

            # Reshape the output to (batch_size, universe_size, self.num_ranks) and apply softmax along the last axis
            x = tf.reshape(x, (-1, self.final_Dense.units // self.num_ranks, self.num_ranks))
            x = tf.nn.softmax(x, axis=-1)
            return x
```

```
Epoch 20/500
5/5 [==============================] - 0s 46ms/step - loss: 1.5128 - categorical_accuracy: 0.3094 - val_loss: 1.5484 - val_categorical_accuracy: 0.2929
Epoch 21/500
5/5 [==============================] - 0s 49ms/step - loss: 1.5148 - categorical_accuracy: 0.3072 - val_loss: 1.5493 - val_categorical_accuracy: 0.2995
Epoch 22/500
5/5 [==============================] - 0s 47ms/step - loss: 1.5170 - categorical_accuracy: 0.3054 - val_loss: 1.5486 - val_categorical_accuracy: 0.2975
Epoch 23/500
5/5 [==============================] - 0s 47ms/step - loss: 1.5133 - categorical_accuracy: 0.3080 - val_loss: 1.5479 - val_categorical_accuracy: 0.2944
Epoch 24/500
5/5 [==============================] - 0s 47ms/step - loss: 1.5146 - categorical_accuracy: 0.3070 - val_loss: 1.5480 - val_categorical_accuracy: 0.3020
Epoch 25/500
5/5 [==============================] - 0s 48ms/step - loss: 1.5141 - categorical_accuracy: 0.3043 - val_loss: 1.5487 - val_categorical_accuracy: 0.2879
Epoch 26/500
5/5 [==============================] - 0s 53ms/step - loss: 1.5153 - categorical_accuracy: 0.3026 - val_loss: 1.5484 - val_categorical_accuracy: 0.2843
```



```python
def get_positional_encoding(num_features, seq_length):
    position = np.arange(num_features)[:, np.newaxis]
    div_term = np.exp(np.arange(0, seq_length, 2) * -(np.log(10000.0) / seq_length))
    positional_encoding = np.zeros((num_features, seq_length))
    positional_encoding[:, 0::2] = np.sin(position * div_term)
    positional_encoding[:, 1::2] = np.cos(position * div_term)

    positional_encoding = positional_encoding[np.newaxis, ...]
    return tf.cast(positional_encoding, dtype=tf.float32)

class AttentionLSTMModel(tf.keras.Model):
    def __init__(self, lstm_units=50, lstm_depth=10, universe_size=110, num_outputs=5, dropout_rate=0.2, num_heads=8, attention_units=64):
        super(AttentionLSTMModel, self).__init__(name='stock_attention_model')
        self.num_features = universe_size
        self.num_outputs = num_outputs
        self.lstm_units = lstm_units

        # Initialize LSTM layers
        self.lstm_layers = [layers.LSTM(lstm_units, return_sequences=True) for _ in range(lstm_depth)]

        # Initialize Dropout layers
        self.dropout_layers = [layers.Dropout(dropout_rate) for _ in range(lstm_depth)]

        # Multi-Head Attention Layer
        self.multi_head_attention = layers.MultiHeadAttention(num_heads=num_heads, key_dim=attention_units)

        # Final Dense layer to output (None, 110, 5)
        self.final_Dense = layers.Dense(num_outputs)

        # Positional Encoding
        self.positional_encoding = get_positional_encoding(universe_size, lstm_units)

    def call(self, input_tensor, training=False):
        x = input_tensor

        # Add positional encoding
        x += self.positional_encoding[:, :x.shape[1], :x.shape[2]]

        for lstm_layer, dropout_layer in zip(self.lstm_layers, self.dropout_layers):
            x = lstm_layer(x)
            x = dropout_layer(x, training=training)

        # Apply multi-head self-attention
        attention_output = self.multi_head_attention(x, x)
        x = attention_output

        # Apply the final Dense layer
        x = self.final_Dense(x)

        return x
```

More complicated

LSTM MODELS

Adding attention did not improve accuracy

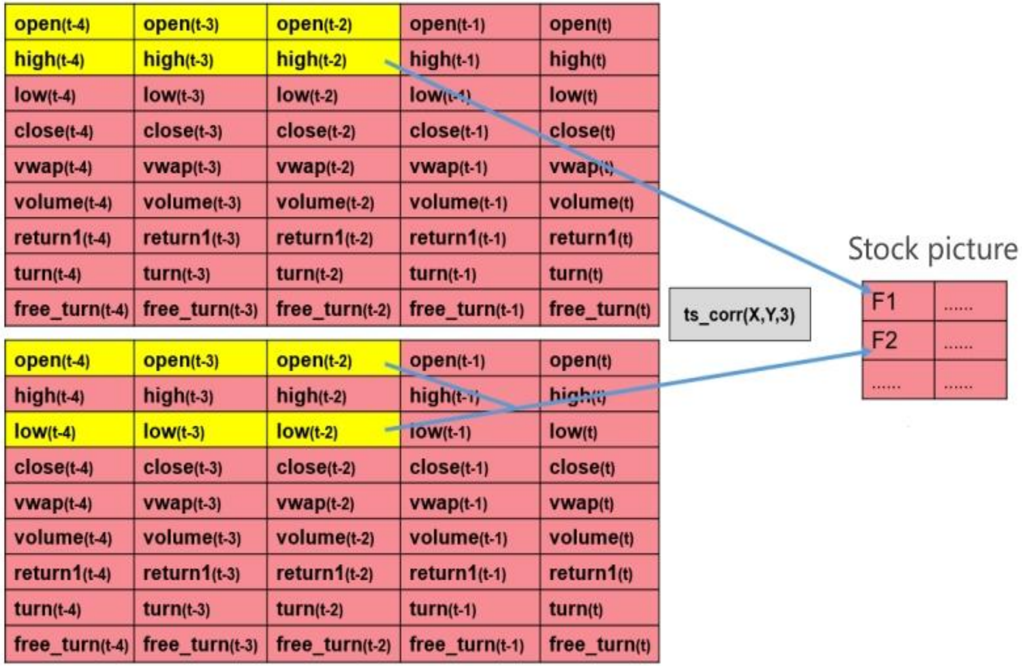
Adding extra factors into LSTM did not improve and was time consuming;

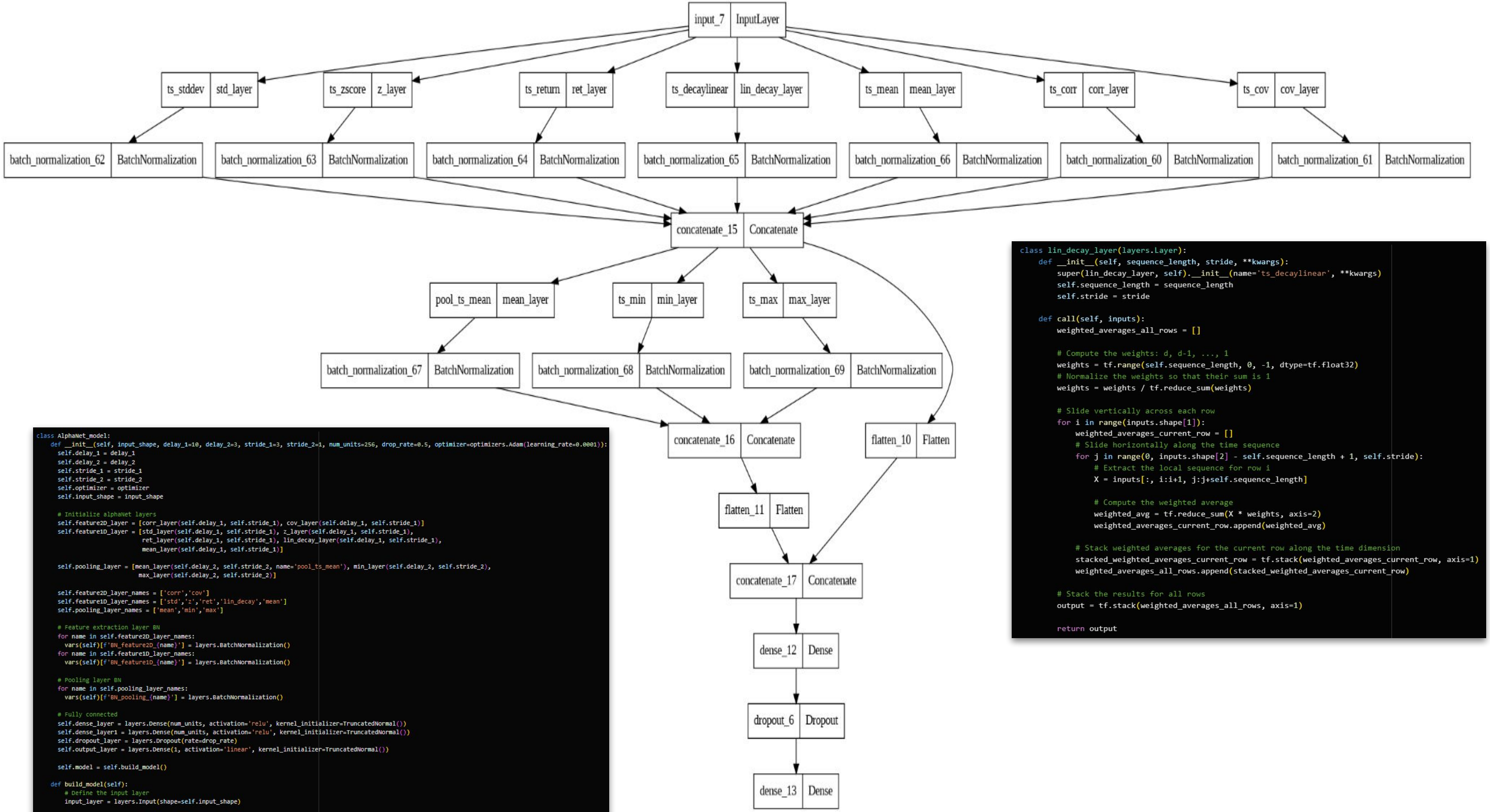
Having to tune 110 models.

ALPHANET

6 months training/6 months validation

- A feature extraction neural network from research published by Huatai Securities
- Customized layers inspired by convolution, treating stock values as pictures.
- It aims to find predictive factors by wrapping statistics of price data in a convolution-like manner.



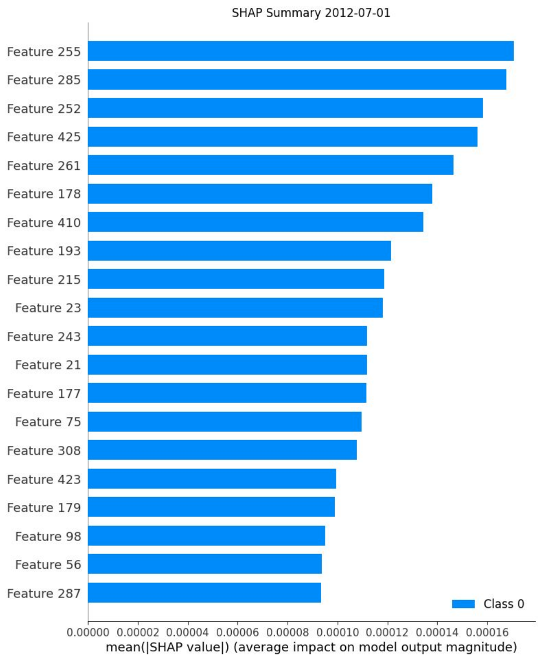


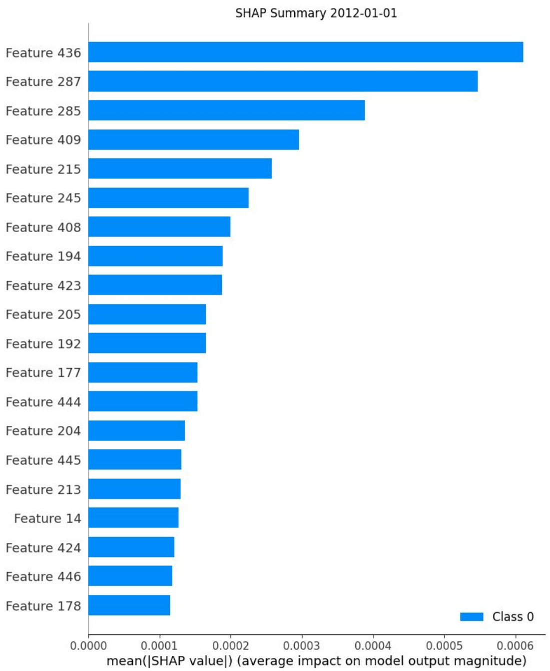

```python
class lin_decay_layer(layers.Layer):
    def __init__(self, sequence_length, stride, **kwargs):
        super(lin_decay_layer, self).__init__(name='ts_decaylinear', **kwargs)
        self.sequence_length = sequence_length
        self.stride = stride

    def call(self, inputs):
        weighted_averages_all_rows = []

        # Compute the weights: d, d-1, ..., 1
        weights = tf.range(self.sequence_length, 0, -1, dtype=tf.float32)
        # Normalize the weights so that their sum is 1
        weights = weights / tf.reduce_sum(weights)

        # Slide vertically across each row
        for i in range(inputs.shape[1]):
            weighted_averages_current_row = []
            # Slide horizontally along the time sequence
            for j in range(0, inputs.shape[2] - self.sequence_length + 1, self.stride):
                # Extract the local sequence for row i
                X = inputs[:, i:i+1, j:j+self.sequence_length]

                # Compute the weighted average
                weighted_avg = tf.reduce_sum(X * weights, axis=2)
                weighted_averages_current_row.append(weighted_avg)

            # Stack weighted averages for the current row along the time dimension
            stacked_weighted_averages_current_row = tf.stack(weighted_averages_current_row, axis=1)
            weighted_averages_all_rows.append(stacked_weighted_averages_current_row)

        # Stack the results for all rows
        output = tf.stack(weighted_averages_all_rows, axis=1)

        return output
```

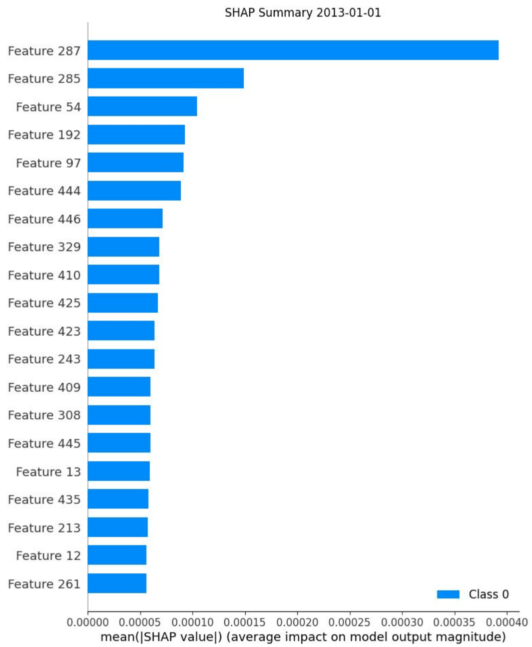
```python
class AlphaNet_model:
    def __init__(self, input_shape, delay_1=10, delay_2=3, stride_1=3, stride_2=1, num_units=256, drop_rate=0.5, optimizer=optimizers.Adam(learning_rate=0.0001)):
      self.delay_1 = delay_1
      self.delay_2 = delay_2
      self.stride_1 = stride_1
      self.stride_2 = stride_2
      self.optimizer = optimizer
      self.input_shape = input_shape

      # Initialize alphaNet layers
      self.feature2D_layer = [corr_layer(self.delay_1, self.stride_1), cov_layer(self.delay_1, self.stride_1)]
      self.feature1D_layer = [std_layer(self.delay_1, self.stride_1), z_layer(self.delay_1, self.stride_1),
                              ret_layer(self.delay_1, self.stride_1), lin_decay_layer(self.delay_1, self.stride_1),
                              mean_layer(self.delay_1, self.stride_1)]

      self.pooling_layer = [mean_layer(self.delay_2, self.stride_2, name='pool_ts_mean'), min_layer(self.delay_2, self.stride_2),
                            max_layer(self.delay_2, self.stride_2)]

      self.feature2D_layer_names = ['corr','cov']
      self.feature1D_layer_names = ['std','z','ret','lin_decay','mean']
      self.pooling_layer_names = ['mean','min','max']

      # Feature extraction layer BN
      for name in self.feature2D_layer_names:
        vars(self)[f'BN_feature2D_{name}'] = layers.BatchNormalization()
      for name in self.feature1D_layer_names:
        vars(self)[f'BN_feature1D_{name}'] = layers.BatchNormalization()

      # Pooling layer BN
      for name in self.pooling_layer_names:
        vars(self)[f'BN_pooling_{name}'] = layers.BatchNormalization()

      # Fully connected
      self.dense_layer = layers.Dense(num_units, activation='relu', kernel_initializer=TruncatedNormal())
      self.dense_layer1 = layers.Dense(num_units, activation='relu', kernel_initializer=TruncatedNormal())
      self.dropout_layer = layers.Dropout(rate=drop_rate)
      self.output_layer = layers.Dense(1, activation='linear', kernel_initializer=TruncatedNormal())

      self.model = self.build_model()

    def build_model(self):
        # Define the input layer
        input_layer = layers.Input(shape=self.input_shape)

        # Feature extraction
        extracted_features1D = [vars(self)[f'BN_feature1D_{name}'](self.feature1D_layer[i](input_layer)) for i, name in enumerate(self.feature1D_layer_names)]
        #for i, feature in enumerate(extracted_features1D):
```

SHAP:
Shapley additive explanations

Calculates how much each factor contributes to the outcome of the model.
Factors look like BN(ts_mean(BN(ts_corr(low,return, delay=10)))) etc

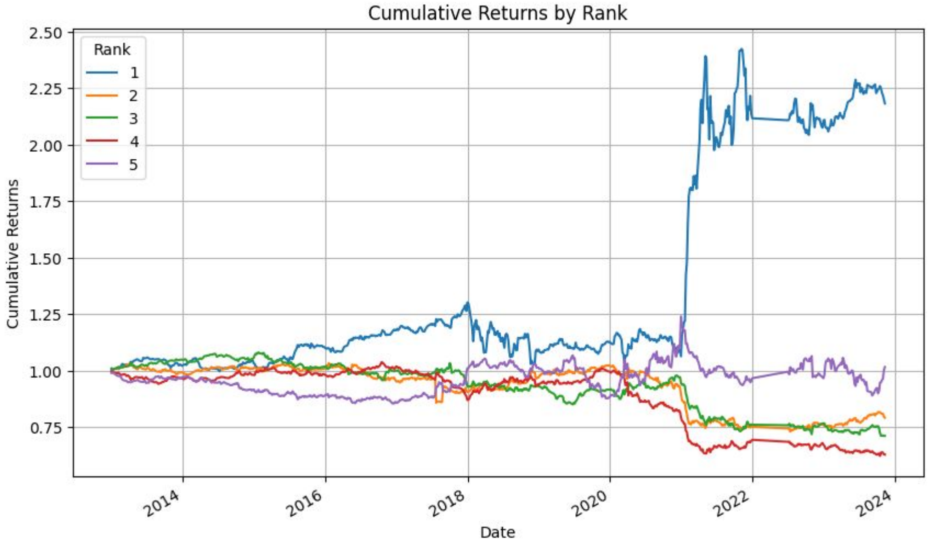






ALPHANET FACTOR

- Train the model using a rolling window of one year length, with a training set to validation set ratio of 1:1.
- Predict the returns for the latter half of the validation set using the trained model as the Alphanet factor.



OVERALL PERFORMANCE

Annual return: **0.2897**
Annual std: **0.0731**
Sharpe ratio: **3.9622**

Thank you