CQF Final Project Report

Deep Learning for Financial Time Series

This task is developing a machine-learning model to predict positive moves of stock price. The whole task can be split into following subtasks:

1. Download original historical data
2. Data preprocessing to get clean and high-informative features
3. Train and select the model
4. Use the selected model to generate trading signal
5. Develop trading strategies based on these predicted trading signal
6. Backtest these strategies to prove your trading idea

1. Data Job

1.1. Download OHLC data

The exam requires "then training and testing over up to 5 years should be sufficient.", which exclude some small or new companies from consideration.

As a result, I decide to choose daily S&P 500 index as my study target, because it has enough data and is relatively less noisy than individual company's stock.

I wrote a Python script to download S&P 500 index from Yahoo Finance, starting from 2000-01-03, ending at 2023-07-28. This data has 5930 rows, and 4 columns: each day's open/high/low/close price and volume. FYI, Yahoo Finance provides both Close price and Adjusted Close price, I use Adjusted Close price but rename it as Close in this project.

The raw data I am going to use is like the snapshot in Figure 1-1.

Date	Open	High	Low	Close	Volume
2000-01-03	1469.250000	1478.000000	1438.359985	1455.219971	931800000
2000-01-04	1455.219971	1455.219971	1397.430054	1399.420044	1009000000
2000-01-05	1399.420044	1413.270020	1377.680054	1402.109985	1085500000
2000-01-06	1402.109985	1411.900024	1392.099976	1403.449951	1092300000
2000-01-07	1403.449951	1441.469971	1400.729980	1441.469971	1225200000

Figure 1-1 snapshot of the raw data

Plot the timeseries of the close price and volume can be seen in Figure 1-2. I can tell from the plot, the close price has an upward trend, which violates Machine Learning's i.i.d assumption for training samples, and will make model training difficult.

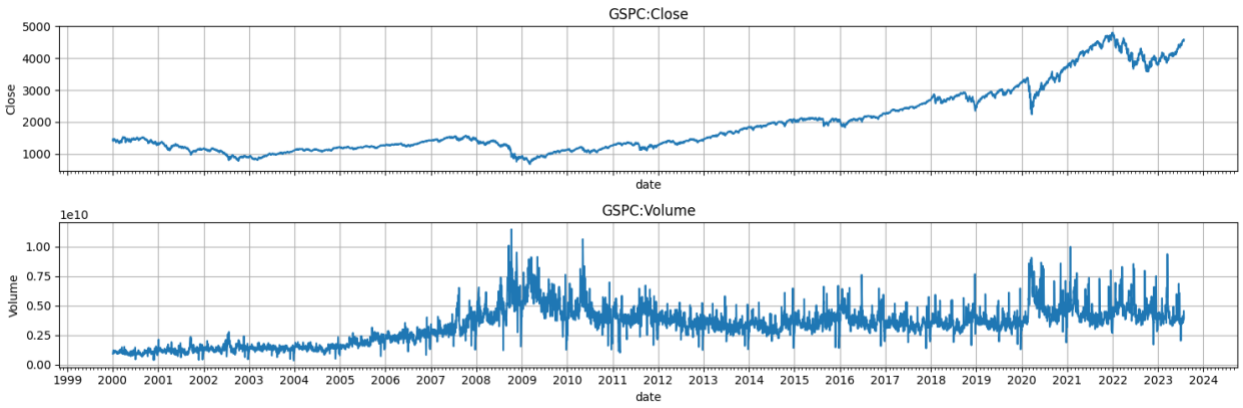

Figure 1-2 timeseries plot the close price and volume

1.2. Generate Features

1.2.1. Label

I model this prediction task as a binary classification task, and define label like this:

- If tomorrow's close price increase more than 0.25% comparing with today's close price, than I label today's example as positive
- Otherwise, today's example is labeled as negative example

```
def __add_label(self):
    # target to fit, tomorrow minus today
    labels = (self.df["Close"].shift(-1) > 1.0025 * self.df["Close"]).astype(int)
    # self.df["Close"].shift(-1) has its last value as Nan, then '>' return False
    # which is incorrect as last example's label
    # set its value to NAN, then this example will be dropped later
    labels.iloc[-1] = np.nan
    self.df["forward_move"] = labels

    self.df["forward_change"] = self.df["Close"].shift(-1) - self.df["Close"]
```

Besides the binary label "forward_move", I also create a column named "*forward_change*" to record the strength of the change from today's price to tomorrow's price.

This "*forward_change*" column can be used to give different weight to each sample. The basic idea is: **profit or loss mainly depends on whether I can accurately predict these significant changes, predicting small changes incorrectly will not have a serious impact.** I tried re-weighting samples differently according to this "forward_change" column, but didn't get any promising results. So, in this project, all examples are still equality weighted, but I believe a reweighting plan based on change amplitude is worth more researching effort.

1.2.2. Price-related features

Besides keeping original open/high/low/close prices in the feature set, I add the difference between open/close price, and difference between high/low price, into feature set. These two differences reflect the degree of disagreement between the long & short sides within a day.

```
def __add_price_range(self):
    self.df["o-c"] = self.df["Open"] - self.df["Close"]
    self.df["h-l"] = self.df["High"] - self.df["Low"]
```

I add the rolling statistics (moving average, exponential moving average, standard deviation) of close price as features.

```
def __add_rolling_price(self, windows):
    for window in windows:
        self.df[f"price_{window}d_sma"] = self.df["Close"].rolling(window).mean()
        self.df[f"price_{window}d_std"] = self.df["Close"].rolling(window).std()
        self.df[f"price_{window}d_ewma"] = self.df["Close"].ewm(span=window, min_periods=window).mean()
```

I add rolling statistics of the return (i.e., relative price change) as features. Notice the difference of this "return" feature and label:

- Label "forward move" is generated based on tomorrow's price minus today's price. In another word, it's a forward return.
- Feature "return" is generated based on today's price minus yesterday's price. In another word, it's a backward return, hence no future information is leaked.

```
def __add_rolling_return(self, windows):
    # today minus yesterday
    self.df["return_b1d"] = np.log(self.df["Close"]).diff()

    for window in windows:
        assert window != 1
        self.df[f"return_b{window}d"] = self.df["return_b1d"].rolling(window).sum()
        self.df[f"return_b{window}d_std"] = self.df["return_b1d"].rolling(window).std()
```

1.2.3. Volume-related features

Rolling statistics (moving average and standard deviation) about trading volume each day is added into feature set.

```
def __add_rolling_volume(self, windows):
    volumes = self.df["Volume"]
    for window in windows:
        assert window != 1
        self.df[f"vlm_{window}d_avg"] = volumes.rolling(window).mean()
        self.df[f"vlm_{window}d_std"] = volumes.rolling(window).std()
```

The Chaikin Oscillator examines both the strength of price moves and underlying buying and selling pressure. A Chaikin Oscillator reading above zero indicates net buying pressure, while one below zero registers net selling pressure. Divergence between price and the Chaikin Oscillator is the indicator's most frequent signal, and often flags a short-term reversal in price. I calculated this indicator by calling TA lib.

```
def __add_chaikin(self):
    """Chaikin A/D Line"""
    chaikin = talib.AD(
```

```
            high=self.df["High"],
            low=self.df["Low"],
            close=self.df["Close"],
            volume=self.df["Volume"],
        )
        self.df["chaikin"] = chaikin
```

On-Balance Volume (OBV) is a technical indicator that measures the flow of positive and negative volume. OBV analyzes the trading direction and reflects the buying or selling pressure of a stock. It adds volume on days when the price rises and subtracts it on days when the price declines. I calculate this indicator by calling TA lib directly.

```
def __add_obv(self):
    self.df["obv"] = talib.OBV(real=self.df["Close"], volume=self.df["Volume"])
```

1.2.4. Classical Technical Indicators

If short-term SMA (Simple Moving Average) crosses over Long-term SMA from below to above, it's called gold-cross. Otherwise, if short-SMA crosses over long-SMA from above to below, it's called dead-cross.

```
def __add_ma_price_cross(self, short_win, long_win):
    short_ma, long_ma = self.df[f"price_{short_win}d_sma"], self.df[f"price_{long_win}d_sma"]

    cross = short_ma > long_ma
    prev_cross = cross.shift(1).fillna(False)

    self.df["is_pricema_gold"] = ((~prev_cross) & cross).astype(int)
    self.df["is_pricema_dead"] = (prev_cross & (~cross)).astype(int)
```

The Relative Strength Index (RSI) is a technical analysis tool that measures the speed and magnitude of a security's recent price changes. The RSI is used to determine overbought or oversold conditions in an asset or market.

```
def __add_rsi(self, windows):
    for window in windows:
        rsi = talib.RSI(self.df["Close"], window)
        self.df[f"rsi_{window}d"] = rsi

        over_bought = rsi >= 80
        over_sold = rsi <= 20

        # yesterday is over_sold, today is not, buy signal
        self.df[f"is_rsi{window}_gold"] = (over_sold.shift(1) & (~over_sold)).astype(int)
        # yesterday is over_bought, today is not, sell signal
        self.df[f"is_rsi{window}_dead"] = (over_bought.shift(1) & (~over_bought)).astype(int)
```

MACD stands for Moving Average Convergence/Divergence, is one of the most popular technical indicators in trading. It can be used as a trend or momentum indicator and signal opportunities to enter and exit positions.
It's a gold cross when MACD line crosses over MACD signal line from below to above, and it's a dead cross when MACD line crosses over MACD signal line from above to below.

```
def __add_macd(self):
    # use default parameters, fastperiod=12, slowperiod=26, signalperiod=9
    macd, macdsignal, macdhist = talib.MACD(self.df["Close"])
    self.df["macd"] = macd
    self.df["macdsignal"] = macdsignal
    # macdhist = macd - macdsignal, which makes 'macd' redundant
    self.df["macdhist"] = macdhist

    # yesterday's hist become today's prev_hist
    prev_hist = self.df["macdhist"].shift(1)
    self.df["is_macd_gold"] = ((prev_hist <= 0) & (self.df["macdhist"] > 0)).astype(int
)
    self.df["is_macd_dead"] = ((prev_hist >= 0) & (self.df["macdhist"] < 0)).astype(int
)
```

Bollinger Bands are a statistical chart that shows the prices and volatility of a financial instrument or commodity over time. The bands are plotted at a standard deviation level above and below a simple moving average of the price. The bands automatically widen when volatility increases and contract when volatility decreases. It's a gold cross when stock price up-cross the upper band, and it's a dead cross when stock price down-cross the lower band.

```
def __add_bbands(self, windows):
    for window in windows:
        boll_up, boll_middle, boll_low = talib.BBANDS(self.df["Close"], timeperiod=wind
ow)

        up_cross = self.df["Close"] > boll_up
        prev_up_cross = up_cross.shift(1).fillna(False)

        low_cross = self.df["Close"] < boll_low
        prev_low_cross = low_cross.shift(1).fillna(False)

        self.df[f"is_boll{window}_gold"] = ((~prev_up_cross) & up_cross).astype(int)
        self.df[f"is_boll{window}_dead"] = ((~prev_low_cross) & low_cross).astype(int)
```

The Average True Range (ATR) is a technical indicator that measures the volatility of an asset's price. It shows how much price fluctuates, on average, during a given time frame. ATR is calculated as the average of the true ranges over the period.

```
def __add_atr(self, windows):
    for window in windows:
        atr = talib.ATR(
            high=self.df["High"],
            low=self.df["Low"],
            close=self.df["Close"],
            timeperiod=window,
        )
```

```
9.          self.df[f"atr_{window}d"] = atr
```

1.2.5. Feature summary

All the features generated are summarized as below. But notice, they will face feature selection in the following step, and only subset of them will be used in fitting the neural network.

Table 1-1 all generated features

	Feature	Meaning
Numerical	Open/High/Low/Close	Original price information
	c-o / h-l	Price fluctuation in a day
	price_{5/10/20}_{sma/ewma/std}	Rolling statistics of close price
	return_b1d	Return from yesterday to today
	return_b{5/10/20}{_std}	Rolling statistics of returns
	macd / macdsignal / macdhist	MACD indicator
	rsi_{5/10/20}d	RSI indicators of different windows
	atr_{5/10/20}d	ATR indicators of different windows
	Volume	Volume in a day
	vlm_{5/10/20}d_{avg/std}	Rolling statistics of trading volumes
	chaikin / obv	Volume-based indicator
Categorical	is_macd_{gold/dead}	Gold or dead cross from MACD indicator
	is_pricema_{gold/dead}	Gold or dead cross from Moving Average Price
	is_rsi{5/10/20}_{gold/dead}	Over-bought or over-sold signal from RSI
	is_boll{5/10/20}_{gold/dead}	Gold or dead cross from Bollinger Band indicator

1.3. Exploratory Data Analysis

To avoid any possible information leakage, Exploratory Data Analysis (EDA) is only performed on training dataset. Details about dataset split is presented in section 1.6.

1.3.1. EDA on categorical features

Categorical features are those technical indicators. These indicators only have two unique values, 1 represents a buying or selling point (i.e., gold or dead cross) is detected, 0 represents no signal is detected.

I calculate the mean value of each indicator feature, and plot them as in Figure 1-3. From the plot, we can see even the highest mean is only about 0.04, which indicates that all these indicator features are extremely sparse, their value remain 0 most of the time.
Even worse, for features like “is_rsi20_dead” & “is_boll5_gold” & “is_boll5_dead”, their values remain 0 all the time. Such no-variation features should be dropped before fitting model.

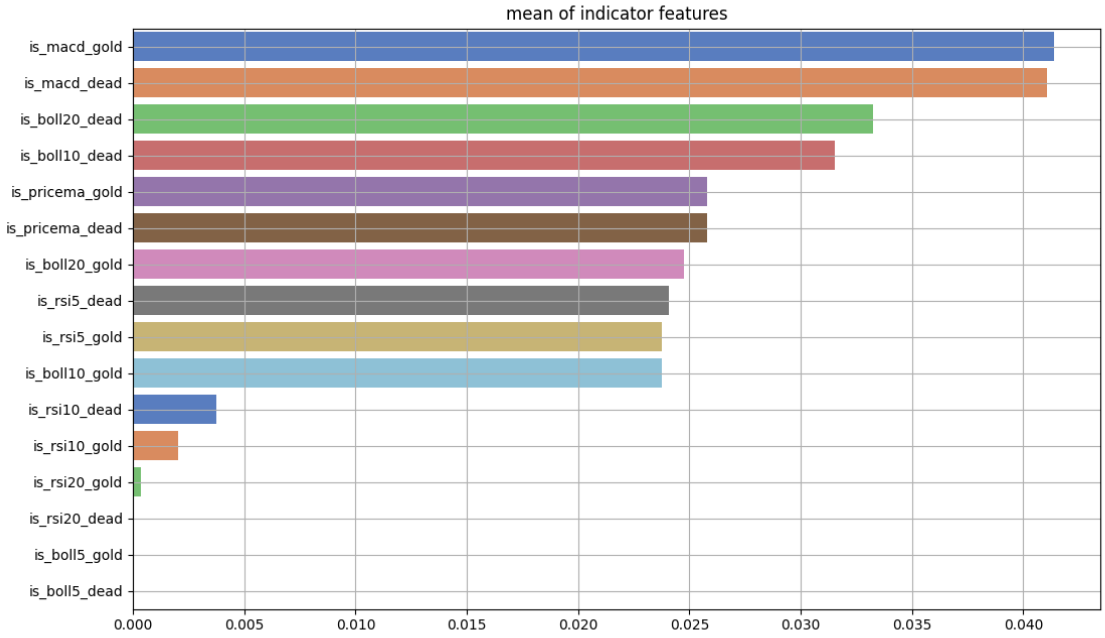

Figure 1-3 mean value of each indicator feature

The reason I add these indicators into feature set, is because I think they are good summary of human traders' experience. However, the highly sparseness make them very difficult to be trained.

1.3.2. EDA on numerical features

I plot histogram of each numerical features in Figure 1-4, to get a feeling about the distribution of these numerical features.

From Figure 1-4, I have following observations:

- Some features, for example, volume-related features like Volume or vlm_xx, have highly skewed distribution.
- Some features, for example, price-related features like price_xxx, exhibit a multimodal distribution.

Above findings tell me that commonly-used Standardized Scaler (aka, Z-score scaling, minus mean then divide standard deviation), is **not suitable for these highly-skewed or multimodal features**, because simple mean and standard deviation will be affected by outliers, and cannot reflect the true middle point and spread of the distribution. "first log transformation, then standardization" may be a good idea to make such features distributed as normal as possible.

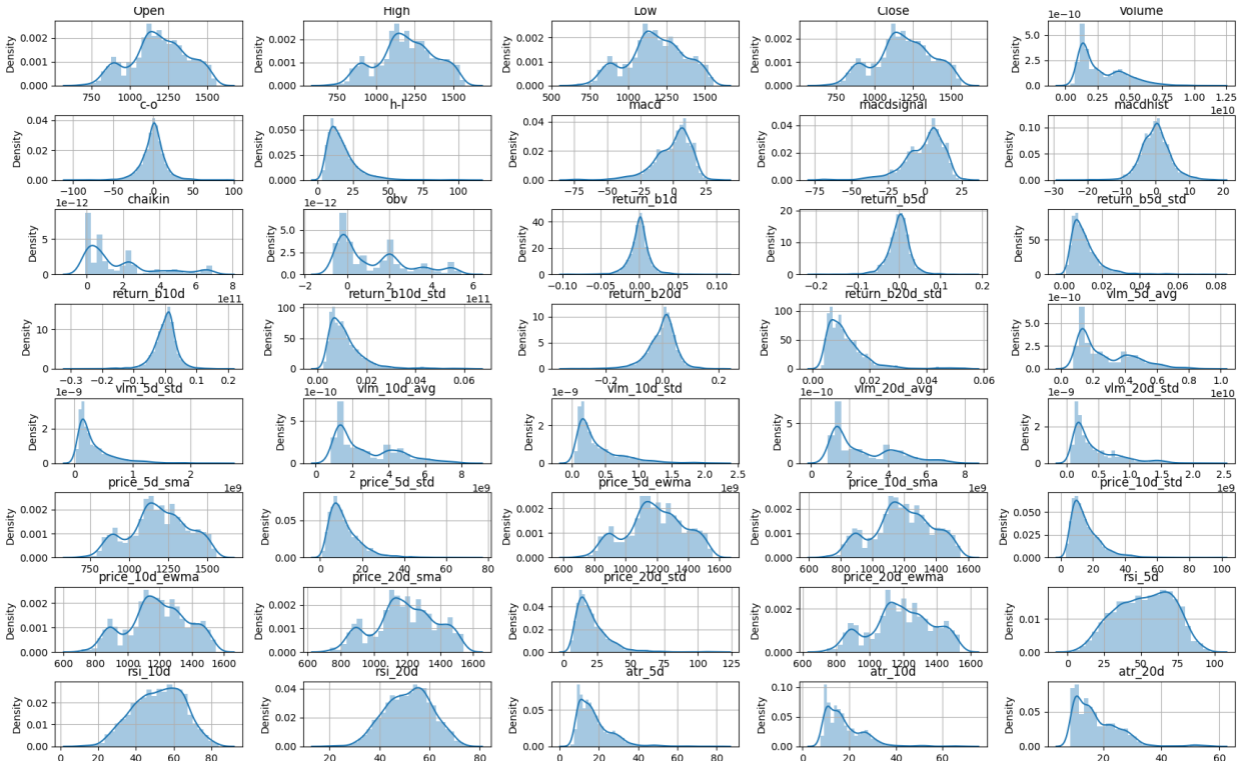

Figure 1-4 distribution of each numeric features

1.3.3. EDA on correlation

I calculate correlation coefficient between every two numerical features. The correlation coefficients can be visualized as in Figure 1-5, and top correlated feature pairs can be seen in Table 1-2 (each row is a feature, and CR1~CR4 are top features correlated with current feature and its correlation coefficient).

These correlation statistics will give me hint during feature selection. Obviously, for highly correlated feature pairs, I only need to include one of them in final feature set for model training, and drop the other one.

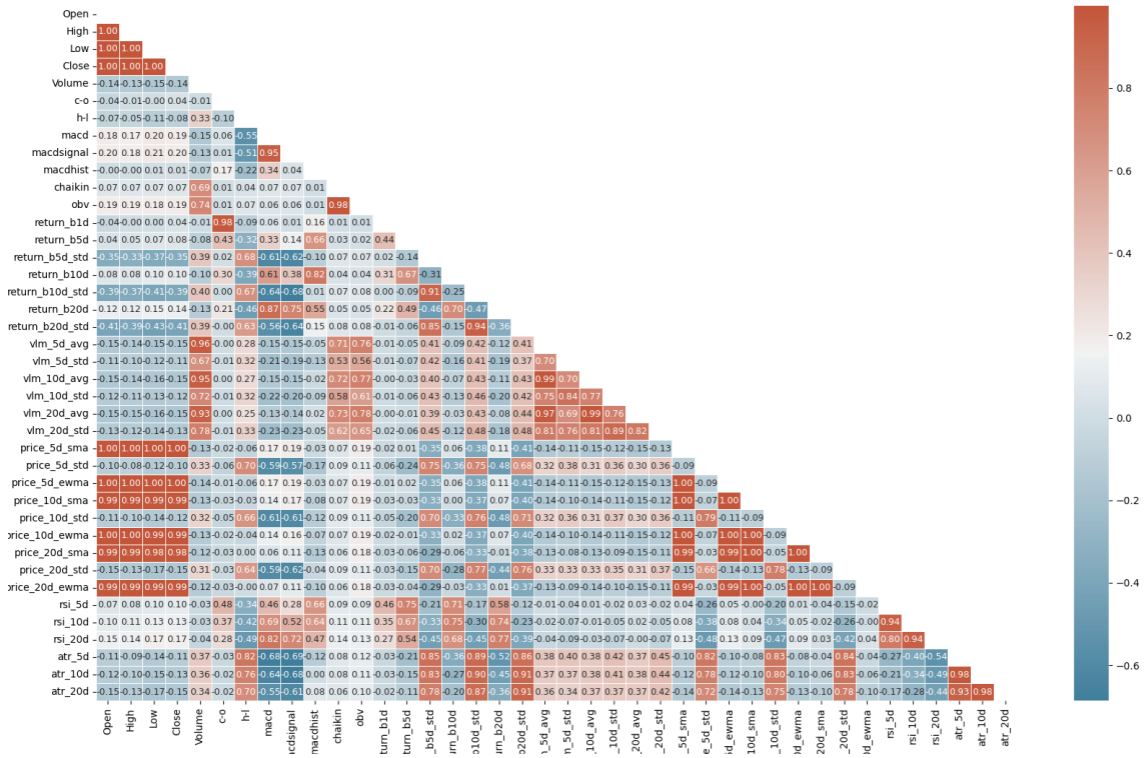

Figure 1-5 correlation among numeric features

Table 1-2 Top related feature pairs

feature	**CR1**	**CR2**	**CR3**	**CR4**
Open	High:1.00	Low:1.00	Close:1.00	price_5d_sma:1.00
High	Open:1.00	Low:1.00	Close:1.00	price_5d_sma:1.00
Low	Open:1.00	High:1.00	Close:1.00	price_5d_sma:1.00
Close	Open:1.00	High:1.00	Low:1.00	price_5d_sma:1.00
Volume	vlm_5d_avg:0.96	vlm_10d_avg:0.95	vlm_20d_avg:0.93	
c-o	return_b1d:0.98			
h-l				
macd	macdsignal:0.95			
macdsignal	macd:0.95			
macdhist				
chaikin	obv:0.98			
obv	chaikin:0.98			
return_b1d	c-o:0.98			
return_b5d				
return_b5d_std	return_b10d_std:0.91			
return_b10d				
return_b10d_std	return_b5d_std:0.91	return_b20d_std:0.94		

return_b20d				
return_b20d_std	return_b10d_std:0.94	atr_10d:0.91	atr_20d:0.91	
vlm_5d_avg	Volume:0.96	vlm_10d_avg:0.99	vlm_20d_avg:0.97	
vlm_5d_std				
vlm_10d_avg	Volume:0.95	vlm_5d_avg:0.99	vlm_20d_avg:0.99	
vlm_10d_std				
vlm_20d_avg	Volume:0.93	vlm_5d_avg:0.97	vlm_10d_avg:0.99	
vlm_20d_std				
price_5d_sma	Open:1.00	High:1.00	Low:1.00	Close:1.00
price_5d_std				
price_5d_ewma	Open:1.00	High:1.00	Low:1.00	Close:1.00
price_10d_sma	Open:0.99	High:0.99	Low:0.99	Close:0.99
price_10d_std				
price_10d_ewma	Open:1.00	High:1.00	Low:0.99	Close:0.99
price_20d_sma	Open:0.99	High:0.99	Low:0.98	Close:0.98
price_20d_std				
price_20d_ewma	Open:0.99	High:0.99	Low:0.99	Close:0.99
rsi_5d	rsi_10d:0.94			
rsi_10d	rsi_5d:0.94	rsi_20d:0.94		
rsi_20d	rsi_10d:0.94			
atr_5d	atr_10d:0.98	atr_20d:0.93		
atr_10d	return_b20d_std:0.91	atr_5d:0.98	atr_20d:0.98	
atr_20d	return_b20d_std:0.91	atr_5d:0.93	atr_10d:0.98	

1.4. Feature Selection

I use Tree Model to select important features for training the neural network. I have several reasons for my choice:

- Some Dimension Reduction algorithms like PCA / SVD can only model linear relationship between features, which is not enough for complex scenarios.
- Linear algorithms like logistic regression, can also give feature importance. However, such algorithm needs lots of preprocessing, and cannot model feature interaction. For example, Feature A and Feature B may not be that important themselves, but their co-appearance is very informative. Unfortunately, linear algorithm cannot model such interaction automatically.
- Tree Models overcome above limitations. They don't need a lot of feature preprocessing. They can handle both numerical & categorical features very well. They can model non-linear relationship between feature and target, and automatically model & evaluate feature interaction.

I fit a Xgboost model on train dataset (no peek at validation & test dataset), and use SHAP module to get feature importance.

```
1.  def explain_by_gbdt(self):
2.      feat_names = list(self._Xtrain.columns)
3.
4.      gbm = xgb.XGBClassifier(max_depth=10, n_estimators=50, early_stopping_rounds=5, eva
    l_metric="auc")
5.      gbm.fit(
6.          self._Xtrain,
7.          self._ytrain,
8.          eval_set=[(self._Xval, self._yval)],
9.      )
10.
11.     feat_importances = pd.Series(gbm.feature_importances_, index=feat_names)
12.     feat_importances.sort_values(ascending=False, inplace=True)
13.     self._save_feat_importances("gbdt", feat_importances)
14.
15.     explainer = shap.Explainer(gbm)
16.     shap_values = explainer(self._Xtrain)
17.     shap.plots.bar(shap_values,max_display=40)
18.     shap.plots.beeswarm(shap_values, max_display=40)
```

Bar plot of each feature's importance is in Figure 1-6. The longer the bar, the more important that feature is.

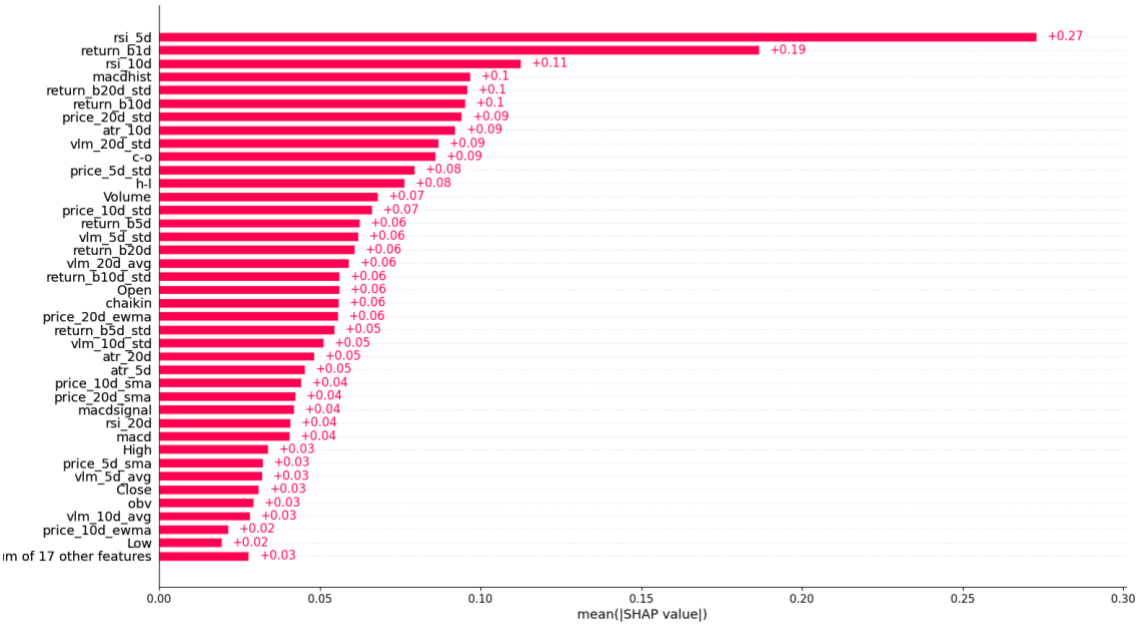

Figure 1-6 SHAP bar plot of each feature's importance

Beeswarm plot of feature importance is in Figure 1-7. Comparing with bar plot, beeswarm plot not only can provide strength of the feature, but also can provide direction, i.e., when the feature grows larger, the target is likely to become more positive or more negative. For

example, in Figure 1-7, I see that the larger 'rsi_5d' is, the more negative the target is. This is consistent with common sense, since large RSI indicates over-bought, the stock price is likely to drop.

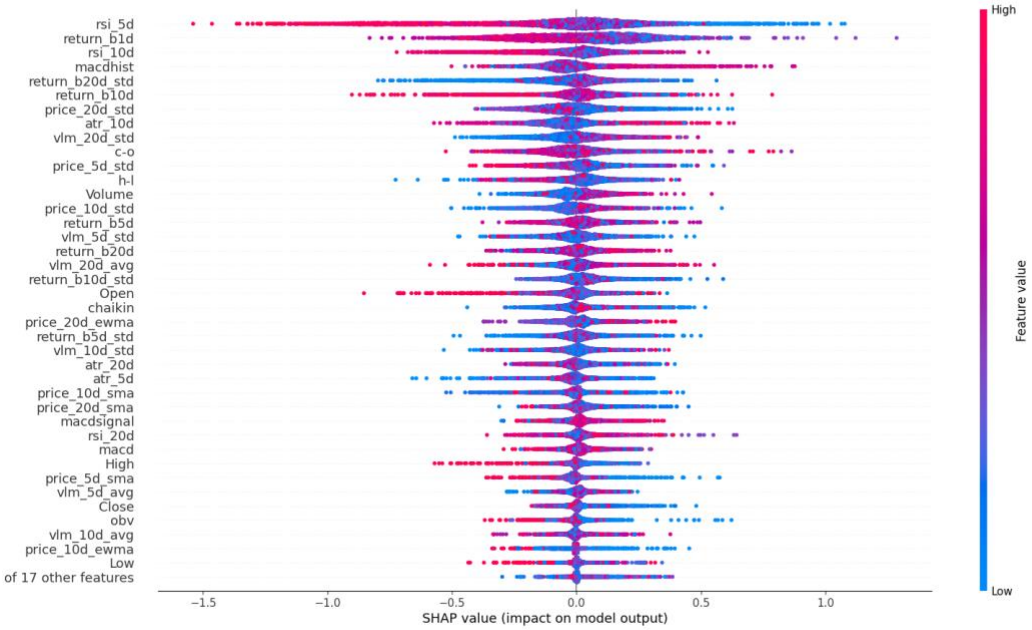

Figure 1-7 SHAP beeswarm plot of each feature's importance

From above plot, I can tell that features like rolling statistics are very important. Contrary to my initial thoughts, indicator features telling whether gold/dead cross happen are not that informative, maybe this is because such indicator features are too sparse.

1.5. Feature Preprocessing

Originally there are 56 raw features generated, after Exploratory Data Analysis and Feature Selection, 43 features are kept to train the model.

Based on how they are preprocessed, these 43 features can be divided into 3 groups, listed in Table 1-3:

- ***SCALING***: feature's distribution isn't that skewed, so it can be directly standardized, i.e., first subtract mean and then divide standard deviation.
- ***LOG-SCALING***: feature's distribution is highly skewed, direct standardization won't make it normal-like. So, a logarithmic transformation is first applied to compress feature's range, then standardization is applied to make feature's distribution as normal as possible.
- ***COPY***: mainly used on indicator features. These features are either 0 or 1, which can be directly copied from raw dataset to final dataset without any further preprocessing.

Table 1-3 selected feature and how they are preprocessed

Preprocessing	**Feature set**
SCALING	c-o, chaikin, macdhist, macdsignal, obv, return_b10d, return_b10d_std, return_b1d, return_b20d, return_b20d_std, return_b5d, return_b5d_std, rsi_10d, rsi_20d, rsi_5d
LOG-SCALING	Close, Volume, atr_10d, atr_20d, atr_5d, h-l, price_10d_sma, price_10d_std, price_20d_sma, price_20d_std, price_5d_sma, price_5d_std, vlm_10d_avg, vlm_10d_std, vlm_20d_avg, vlm_20d_std, vlm_5d_avg, vlm_5d_std
COPY	is_boll10_dead, is_boll10_gold, is_boll20_dead, is_boll20_gold, is_macd_dead, is_macd_gold, is_pricema_dead, is_pricema_gold, is_rsi5_dead, is_rsi5_gold

1.6. Split Dataset

Since we are dealing with a timeseries task, the most important rule is **never leaking any future information when fitting the model**. Also, the exam requires "training and testing over up to 5 years", so I split train/validation/test dataset chronologically according to "2:1:1" rule, shown in Table 1-4.

Table 1-4 dataset splits information

Dataset	**Time range**	**Number of samples**	**Positive ratio**
Train	2000-02-18 ~ 2011-11-04	2948	40.03%
Validation	2011-11-07 ~ 2017-09-15	1474	35.89%
Test	2017-09-18 ~ 2023-07-27	1474	40.50%

There are several controversies surrounding the data split.

1.6.1. Imbalanced problem

From Table 1-4, we can see that all three datasets have a positive ratio less than 0.5, hence all datasets are imbalanced. However, I decide to ignore this problem, and train these imbalanced datasets directly without any rebalance work. I have two reasons for my decision:

- Rebalancing involves re-weighting the samples, or over-sampling / down-sampling. All these methods change the true distribution of the dataset. Model trained from rebalanced dataset has great bias, and its predictions aren't consistent with real world. You have to correct the predictions before using them in making decision, which is troublesome and error-prone.
- **I am a recommender-system expert working for large internet company**. It's very common for me to deal with extremely-imbalanced dataset (e.g., positive: negative=1:100) during Click Through Rate prediction. From my experience, imbalance with a 30%~40% positive ratio isn't that serious, most of the machine learning algorithm can handle them well.

Although I choose not rebalancing the dataset before training, there are several points requiring special attention:

- Neural network to be trained won't predict positive or negative label directly, but give the probability for each example to be positive. I need to define a thresholds to translate probability to positive or negative label. However, **the choice of the thresholds is subjective to human bias**.
- Accuracy / Confusion Matrix aren't good metric for evaluating classification model trained on imbalanced dataset, because they are highly dependent on the choice of the threshold. I mainly use AUC to evaluate model's performance. AUC measure model's capability to rank positive examples before negative examples, which isn't affected by the choice of threshold.

1.6.2. Drifting problem

Since I am fitting a 23-year timeseries, data distribution has drifted seriously as time goes on. Such drift can be illustrated in Figure 1-8.

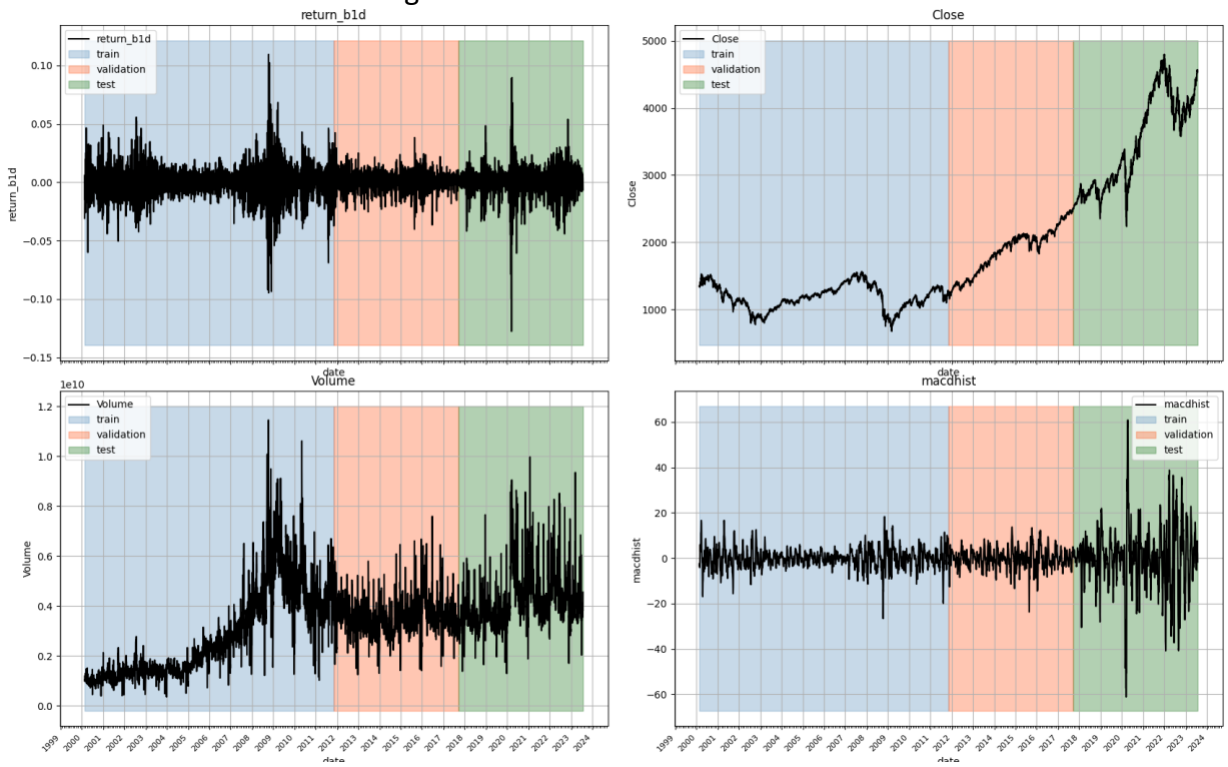

Figure 1-8 data distribution drift in three datasets

- In "Close" subplot, I can see that US stocks are in a bull market from a long-term perspective, the S&P 500 index is moving higher and higher.
- In "return_b1d" (i.e., today's return against yesterday) subplot, I can see vicious oscillation in train and test dataset. The first oscillation is due to 2008 subprime crisis, while the second one is caused by 2019 COVID-19 crisis. However, such oscillation is lacked in validation period, which **highly damage the reliability of any metrics generated from validation dataset.**

In summary, **distribution drifting is a serious problem which violates the "independent identically distributed" foundation of Machine Learning**. I know my split plan is far from perfection, **however that's the best I can do now**. To alleviate the problem caused by drifting distribution, I will pay special attention to **NOT overfitting** during model training. More research about timeseries drifting problem will be left to future work.

2. Model Job

Advanced declaration, the exam requires "*The best model should be presented only after performing the hyperparameter optimization*". I know there are some automatic frameworks such as Keras Tuner, however I still prefer manual hyperparameter optimization in this exam. The reason for my decision is listed below:

- Some hyperparameter tuning algorithm, such as GridSearch, is very inefficient. I prefer using my own knowledge and experience to guide the next tuning operation. For example, if you find model performs much better on train dataset than on validation / test datasets, then your next step should be increasing regularization, other than let GridSearch waste time in the opposite tuning direction.
- There is no guarantee that the hyperparameters found by these automatic frameworks are globally the best. Because these frameworks still rely on initial search range fed by human, which is still subjective to human bias.
- Because of the distribution drift problem mentioned in 1.6.2, best validation metric cannot guarantee best performance on test dataset. Automatic framework searches best validation metric, which is a kind of over-optimization and time-waste. It is enough to find a set of OK hyperparameters by manual hyperparameter search.

2.1. Simple RNN

Model building codes are listed below.

Code 2-1 Simple RNN

```
class SimpleRNN(BaseModel):
    def __init__(self, configs) -> None:
        super().__init__(configs)
        self._feat_names = configs["feature_group"]["feature"]["feat_names"]

    def build(self):
        l2 = self.configs.get("l2", 0)

        model = keras.Sequential()
        model.add(keras.Input(shape=(self.configs["seq_len"], len(self._feat_names))))

        for idx, units in enumerate(self.configs["rnn"]):
            return_sequences = idx != len(self.configs["rnn"]) - 1
            lstm = layers.LSTM(
                name=f"lstm_{idx}",
                units=units,
                return_sequences=return_sequences,
                kernel_regularizer=regularizers.l2(l2),
                recurrent_regularizer=regularizers.l2(l2),
                activity_regularizer=regularizers.l2(l2),
            )
```

```
            model.add(lstm)

        for idx, units in enumerate(self.configs["dense"]):
            hidden = layers.Dense(
                name=f"hidden_{idx}",
                units=units,
                activation="relu",
                kernel_regularizer=regularizers.l2(l2),
            )
            model.add(hidden)

        final_layer = layers.Dense(name="final", units=1, activation="sigmoid")
        model.add(final_layer)

        return model
```

Because of the high sparsity of the indicator features, I only include numerical features as input to LSTM. Maybe feeding indicators is helpful to LSTM, I have to leave it to future research due to time limitation.

The codes for building baseline Simple RNN are listed in Code 2-1, and structure of the built model is illustrated in Figure 2-1.

Code 2-2 build baseline Simple RNN

```
REALNUM_FEATS = ['Close', 'Volume', 'c-o', 'h-l', 'macdsignal', 'macdhist',
    'chaikin', 'obv', 'return_b1d', 'return_b5d', 'return_b5d_std',
    'return_b10d', 'return_b10d_std', 'return_b20d', 'return_b20d_std',
    'vlm_5d_avg', 'vlm_5d_std', 'vlm_10d_avg', 'vlm_10d_std', 'vlm_20d_avg',
    'vlm_20d_std', 'price_5d_sma', 'price_10d_sma', 'price_20d_sma', 'price_5d_std',
    'price_10d_std', 'price_20d_std', 'rsi_5d', 'rsi_10d', 'rsi_20d', 'atr_5d', 'atr_10d
', 'atr_20d']
feat_group = {"feature": {"mode": "window", "feat_names": REALNUM_FEATS}}
configs = {
    "seq_len": 20,
    "rnn": [64],
    "dense": [64],
    "epochs": 100,
    "earlystop_patience": 10,
    "l2": 0.01,
    "feature_group": feat_group,
}

model = SimpleRNN(configs).build()
```

Beside the baseline version, there are several variants with different configurations are trained and evaluated in this project, listed in Table 2-1.

Table 2-1 variants of Simple RNN

Variant	**Configuration**
baseline	Configuration used in Code 2-2
no_regularization	Change L2-regularization coefficient to 0

longer_window	Change the sequence fed into LSTM from 20 to 40, use longer history
smaller_layer_32	Shrink units in RNN & Dense Layer from 64 to 32
larger_layer_128	Increase units in RNN & Dense Layer from 64 to 128
double_layers_32	Original baseline model only has one LSTM layer and one Dense layer. This variant has two LSTM layers, both has 32 units, as well as two Dense layers, both has 32 units.

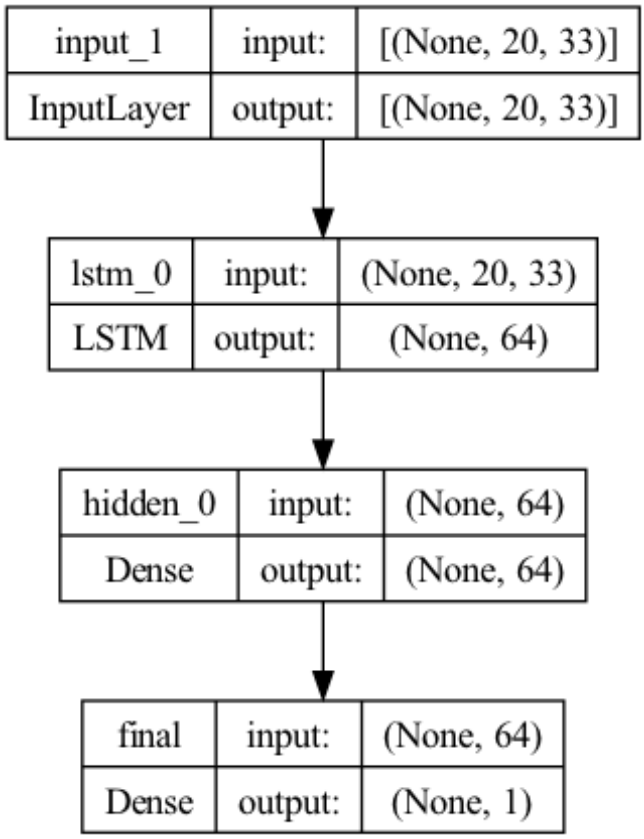

Figure 2-1 Simple RNN structure

2.2. Improve RNN with latest indicators

Although indicator features are very sparse, I want to see whether they can improve my model. I use technical indicators in my model like this:

1. Feed numerical features into LSTM, same as Simple RNN
2. After LSTM gets its final state, before feeding them into Dense Layers, concatenate them with latest (i.e., last time step) technical indicators.
3. Combined states (including both LSTM state and latest technical indicators) are passed through several Dense layers, the final output is the probability that stock price will move up the next day.

Above steps are implemented in Code 2-3.

Code 2-3 Improved RNN with latest indicators

```
class ImproveRnnWithIndicators(BaseModel):
    def __init__(self, configs) -> None:
        super().__init__(configs)
        self._realnum_features = configs["feature_group"]["realnum"]["feat_names"]
        self._indicator_features = configs["feature_group"]["indicator"]["feat_names"]


    def build(self):
        l2 = self.configs.get("l2", 0)

        # history: [batch_size, window_size, feature_dim]
        realnum_input = keras.Input(
         name="realnum", shape=(self.configs["seq_len"], len(self._realnum_features))
        )
        # current step: [batch_size, feature_dim]
        indicator_input = keras.Input(name="indicator", shape=(len(self._indicator_features),))

        X = realnum_input
        for idx, units in enumerate(self.configs["rnn"]):
            return_sequences = idx != len(self.configs["rnn"]) - 1
            lstm = layers.LSTM(
                name=f"lstm_{idx}",
                units=units,
                return_sequences=return_sequences,
                kernel_regularizer=regularizers.l2(l2),
                recurrent_regularizer=regularizers.l2(l2),
                activity_regularizer=regularizers.l2(l2),
            )
            # input: [batch_size, window_size, units]
            # output: [batch_size, window_size, units] or [batch_size, units]
            X = lstm(X)

        # [batch_size, ind_feature_dim + rnn_units]
        X = tf.concat([indicator_input, X], axis=1)

        for idx, units in enumerate(self.configs["dense"]):
            dense = layers.Dense(
                name=f"dense_{idx}",
                units=units,
                activation="relu",
                kernel_regularizer=regularizers.l2(l2),
            )
            X = dense(X)

        final_layer = layers.Dense(name="final", units=1, activation="sigmoid")
        logit = final_layer(X)

        model = keras.Model(inputs=[realnum_input, indicator_input], outputs=logit)
        model.summary()

        return model
```

Codes for building baseline Improved RNN is listed in Code 2-4, and the structure of built model is illustrated in Figure 2-2.

Code 2-4 build baseline improved RNN

```
REALNUM_FEATS = ['Close', 'Volume', 'c-o', 'h-l', 'macdsignal', 'macdhist',
 'chaikin', 'obv', 'return_b1d', 'return_b5d', 'return_b5d_std',
 'return_b10d', 'return_b10d_std', 'return_b20d', 'return_b20d_std',
 'vlm_5d_avg', 'vlm_5d_std', 'vlm_10d_avg', 'vlm_10d_std', 'vlm_20d_avg',
 'vlm_20d_std', 'price_5d_sma', 'price_10d_sma', 'price_20d_sma', 'price_5d_std',
 'price_10d_std', 'price_20d_std', 'rsi_5d', 'rsi_10d', 'rsi_20d', 'atr_5d', 'atr_10d',
 'atr_20d']
INDICATOR_FEATS = ["is_macd_gold","is_macd_dead","is_pricema_gold","is_pricema_dead",
                   "is_rsi5_gold", "is_rsi5_dead", "is_boll10_gold", "is_boll10_dead",
                   "is_boll20_gold", "is_boll20_dead",]
feat_group = {
    "realnum": {"mode": "window", "feat_names": REALNUM_FEATS},
    "indicator": {"mode": "last", "feat_names": INDICATOR_FEATS},
}
configs = {
    "seq_len": 20,
    "rnn": [64],
    "dense": [64],
    "epochs": 100,
    "earlystop_patience": 10,
    "l2": 0.01,
    "feature_group": feat_group,
}

model = ImproveRnnWithIndicators(configs).build()
```

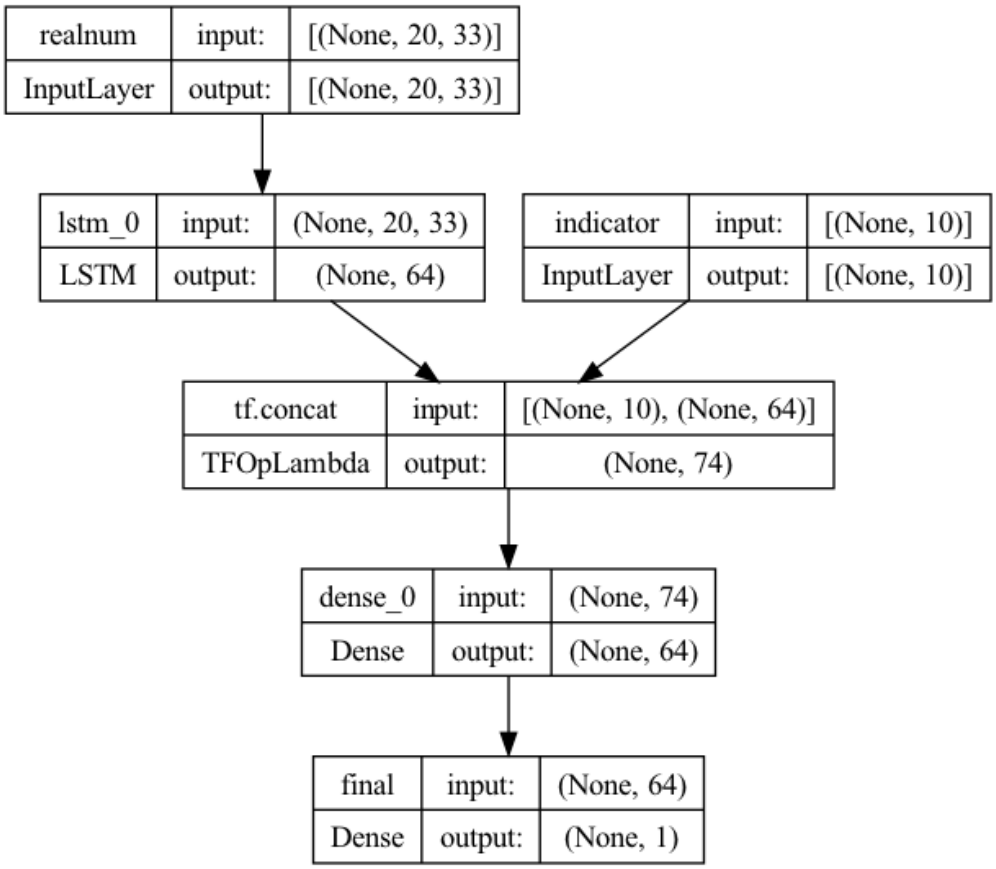

Figure 2-2 Improved RNN structure

Besides the baseline improved RNN, some variants with different configurations are also trained and evaluated in this project.

Table 2-2 variants of Improved RNN

Variant	**Configuration**
improved_rnn	Baseline Improved RNN, with configurations used in Code 2-4
impv_rnn_longer_win	Increase the sequence length from 20 to 40, using longer history

2.3. Model Comparison & Selection

Model evaluation results are summarized in Table 2-3.

- Each column is the configuration and evaluation metric of one model. Model name (i.e., column name) can be referenced in Table 2-1 and Table 2-2.
- I use two metrics to evaluate different models. One is AUC, the other one is "***score_stddev***", which is the standard deviation of predicted scores. If "score_stddev" is very small, it means the model generates nearly constant scores, which indicates very poor discernibility. **When two models' AUC are close, the model with larger score_stddev is better.**

Table 2-3 model evaluation results

config or metric	**baseline**	**no_regularization**	**longer_window**	**smaller_layer_32**	**larger_layer_128**	**double_layers_32**	**improved_rnn**	**impv_rnn_longer_win**
seq_len	20	20	40	20	20	20	20	40
rnn	64	64	64	32	128	32,32	64	64
dense	64	64	64	32	128	32,32	64	64
epochs	100	100	100	100	100	100	100	100
earlystop_patience	10	10	10	10	10	10	10	10
l2	0.01	0	0.01	0.01	0.01	0.01	0.01	0.01
rand_seed	4862	4862	4862	4862	4862	4862	4862	4862
model	SimpleRNN	SimpleRNN	SimpleRNN	SimpleRNN	SimpleRNN	SimpleRNN	ImproveRnn	ImproveRnn
AUC#train	0.589	0.623	0.586	0.591	0.588	0.509	0.592	0.598
AUC#val	0.613	0.584	0.611	0.612	0.613	0.532	0.614	0.613
AUC#test	0.589	0.572	0.584	0.585	0.588	0.522	0.583	0.576
score_stddev#train	0.048	0.077	0.046	0.052	0.055	0.000	0.066	0.052
score_stddev#val	0.036	0.052	0.035	0.046	0.044	0.000	0.053	0.039
score_stddev#test	0.045	0.041	0.042	0.060	0.060	0.000	0.069	0.050

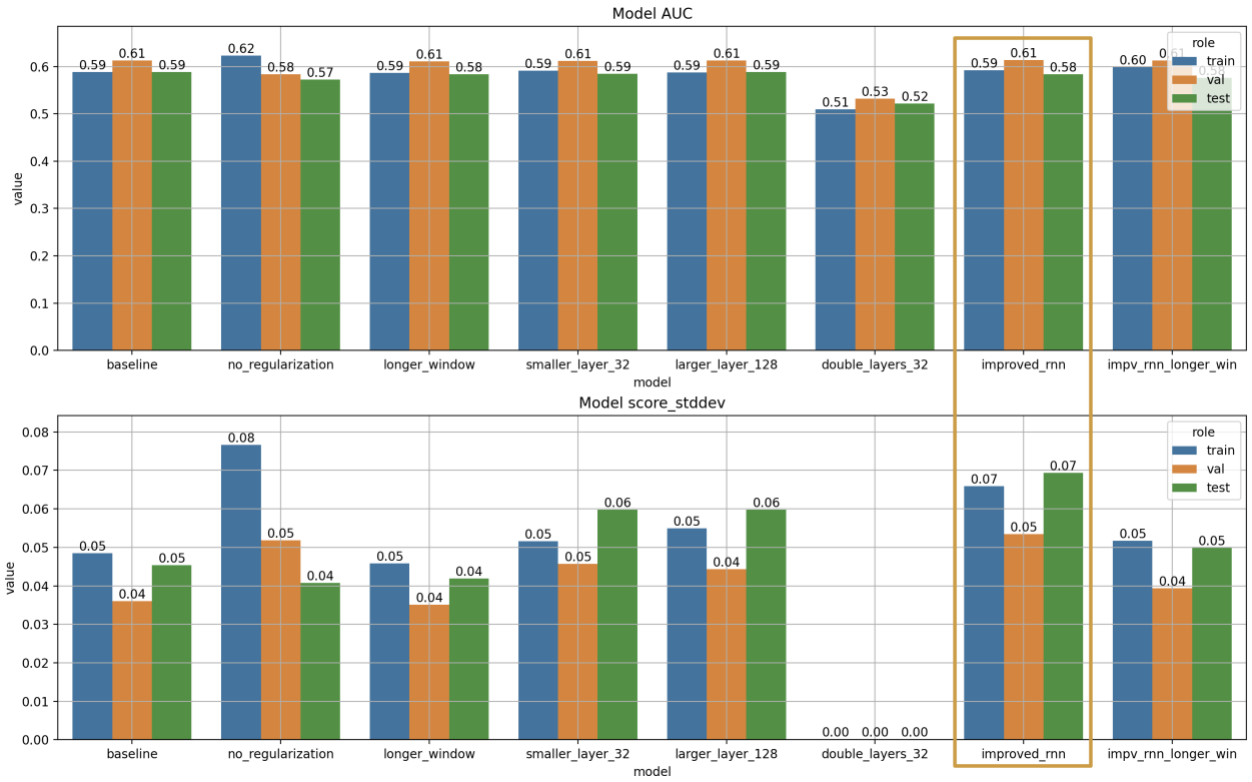

Figure 2-3 model evaluation comparison

From Figure 2-3, I can draw following conclusions:

- Model "no_regularization" has largest train AUC and train score_stddev, but it has worse metrics on validation / test datasets. This phenomenon reflects overfitting. Considering the distribution drift between train/valuation/test periods, model "no_regularization" isn't suitable.
- Model "double_layer_32" has the lowest AUC, what's even worse is that all its three score_standdev are nearly 0. This phenomenon reflects the model generates nearly constant predictions, which has very poor discernibility. A guess is that adding more layers increases model complexity, which results in serious overfitting.
- Other models have very similar AUC which are all near 0.6. **Considering the stochastic nature and heavy noise in stock price movement, I am satisfied with these "high" AUC**, which is much better than random guess (AUC=0.5) and reflects the model did learn some patterns from the data.
- When AUCs are close, model with larger score_stddev is selected, because large score_stddev reflects strong discernibility. In this case, model "improved_rnn" is the better one.

Finally, I choose model "improved_rnn" (i.e., LSTM improved by latest technical indicators), and use its prediction to develop trading strategies.

3. Strategy Job

Model prediction only answer the question "how probable the stock price will move up tomorrow?", and the model I selected in previous chapter did a good job on this question (i.e., AUC=0.6 > 0.5 from random guess).

But that's far from enough, I need the answer for following two questions:

- Whether I should buy or sell tomorrow?
- How much I should buy or sell tomorrow?

This chapter will focus on these two questions, and translate model predictions into real trading strategies. Framework "backtesting.py" is used to do backtesting and statistics in this chapter.

3.1. Select thresholds

Binary classification model won't give moving-up or down signal directly, but give the probability that tomorrow's stock price will move up, it's human's decision and responsibility to decide two thresholds:

- when today's predicted score is lower than down-threshold, I sell tomorrow;
- when today's predicted score is higher than up-threshold, I buy tomorrow.

The threshold can be selected by inspecting the distribution of the predicted score. For example, if I choose the median of all predicted score as threshold, that means I will have half of my time holding long position and the other half holding short position.

Percentiles of train/validation/test predicted scores are summarized in Table 3-1, histogram are plotted in Figure 3-1. The vertical dash lines in the plot represent median score of 3 datasets.

Table 3-1 percentiles of prediction scores in train/validation/test datasets

dataset	**10p**	**20p**	**30p**	**40p**	**50p**	**60p**	**70p**	**80p**	**90p**
train	0.316	0.336	0.357	***0.377***	***0.394***	***0.412***	0.434	0.461	0.501
val	0.305	0.322	0.334	***0.346***	***0.360***	***0.373***	0.387	0.410	0.444
test	0.275	0.292	0.305	***0.321***	***0.339***	***0.360***	0.389	0.414	0.460

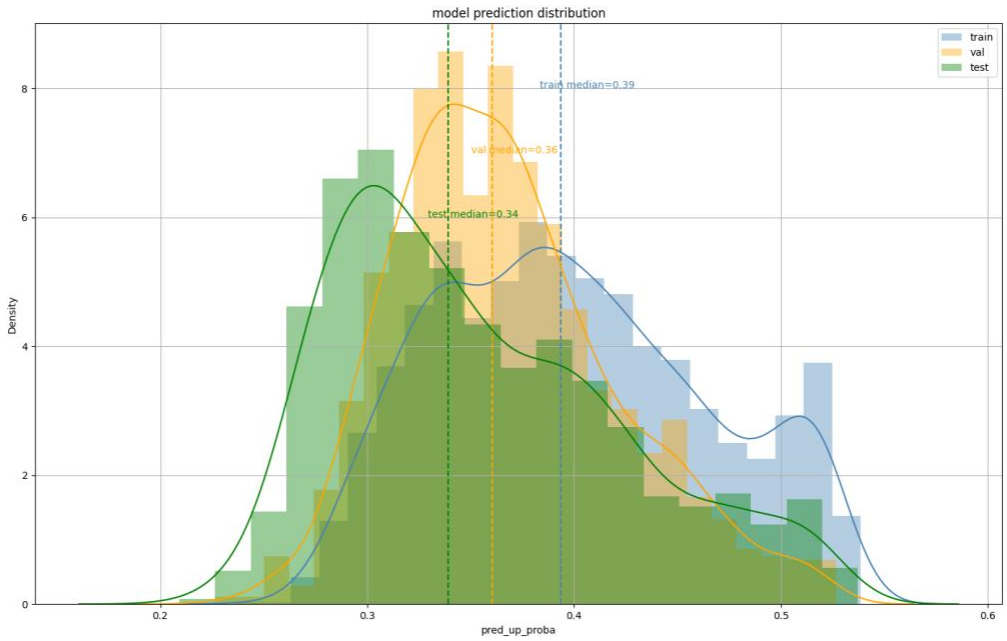

Figure 3-1 distribution of predicted scores

It's not surprising to find that the drift problem also exists in predictions. The median validation score (0.36) drifts left from median train score (0.39). Although I should not peek the test dataset, however the median test score (0.34) also drifts left from median validation score (0.36).

In following backtests, I choose thresholds like following:

- For validation dataset, the thresholds are [***0.33, 0.37***]. That is, when today's predicted score <=0.33, I short it tomorrow. When today's predicted score >=0.37, I long it tomorrow.
- **Here comes the most controversial part**. For test dataset, the thresholds are [***0.32,0.35***]. That is, when today's predicted score <=0.32, I short it tomorrow. When today's predicted score>=0.35, I long it tomorrow.

I admit that my threshold choices are **subjective, biased and even has the suspect peeking the future**. For my defense:

- Even without peeking dataset, I have already been quite aware the drift problem by using just the train & validation dataset. Since I already found the prediction distribution has drifted towards 0.3 left (from train median 0.39 to validation median 0.36), **it's a valid and reasonable guess that the test score distribution will also drift 0.1~0.3 left from validation scores. As a result, I made a valid and reasonable guess that the test thresholds should move left from validation thresholds**, in my case, that is [*0.33,0.37*] as validation thresholds to [*0.32,0.35*] as test thresholds.

- No matter how hard we try, overfitting and peeking cannot be fully eliminated. Because we always want to find the strategy that works well or even best in test dataset, and overfitting and data leakage happen unconsciously during this "optimization" process. No one can guarantee that best strategy on test dataset can repeat its success when deployed into real market. **For us developers, what we can do is aware of the source of the overfitting and peeking, and become more cautious when deploying such strategy into real market, for example more careful monitor and strict risk management**.

3.2. Signal Generator

The whole trading strategy contains two parts:

- ***Signal Generator***: this part tells me tomorrow's trading direction, i.e., buy or sell.
- ***Position Manager***: this part tells me tomorrow's trading size, i.e., how much to buy or sell.

There are several signal generators implemented in this project.

3.2.1. DummySignal

No real signal is generated in this class. The purpose of this class is to work with position manager "SellBuyEveryday" and simulate a Buy & Hold strategy. The Buy & Hold strategy will be used as a benchmark in following evaluation.

```
class DummySignal(StrategyWrapper):
    def next(self):
        """No real signal, work with SellBuyEveryday"""
        return 0
```

3.2.2. SmaSignal

This class implements the classical Simple Moving Average (SMA) strategy, that is,

- When short-term price SMA cross over long-term price SMA upwards, long it
- When short-term price SMA cross over long-term price SMA downwards, short it.

This SMA strategy will be used as a benchmark in following evaluation.

```
class SmaSignal(StrategyWrapper):
    def __init__(self, strategy: Strategy) -> None:
        super().__init__(strategy)
        close_prices = pd.Series(self._data.Close)

        short_win = self._config.get("short_win", 10)
        self.add_indicator(
            name="short_sma", series=close_prices.rolling(short_win).mean(), plot=True, overlay=True
        )

        long_win = self._config.get("long_win", 20)
        self.add_indicator(
            name="long_sma", series=close_prices.rolling(long_win).mean(), plot=True, overlay=True
        )
```

```
15.
16.     def next(self):
17.         short_sma = self.get_indicator("short_sma")
18.         long_sma = self.get_indicator("long_sma")
19.
20.         if crossover(short_sma, long_sma):
21.             return 1
22.
23.         elif crossover(long_sma, short_sma):
24.             return -1
25.
26.         return 0
```

3.2.3. ModelSignal

This is the main character in this project. It reads the prediction scores from model selected in 2.3, and generates 3 signals (i.e., term "indicator" used by backtesting.py).

Signal	**Meaning**
pred_proba	Original prediction score from RNN model
next_posititon	Compare pred_proba with thresholds specified in 3.1. • if pred_proba<=down_threshold, short it tomorrow; • if pred_proba>=up_threshold, long it tomorrow.
signal_strength	When pred_proba is far from thresholds, it has a large signal_strength; otherwise, its signal_strength is very small. This signal_strength will be used by position manager to adjust trading size. • Larger signal_strength means more confidence in trading direction, so I should increase the trading size. • Smaller signal_strength means lacking confidence in trading direction, so I should decrease the trading size.

```
1.  class ModelSignal(StrategyWrapper):
2.      def __init__(self, strategy: Strategy) -> None:
3.          super().__init__(strategy)
4.
5.          next_positions = []
6.          signal_strengths = []
7.          for proba in self._data["pred_up_proba"]:
8.              position = 0
9.              strength = 0
10.
11.             if proba >= self._config["up_threshold"]:
12.                 position = 1
13.                 strength = proba / self._config["up_threshold"]
14.             elif proba <= self._config["down_threshold"]:
15.                 position = -1
16.                 strength = self._config["down_threshold"] / proba
17.
18.             next_positions.append(position)
19.             signal_strengths.append(strength)
20.
```

```
        # when predicted probability is just at threshold, the strength should be 'signal_strength_scale'
        signal_strengths = np.asarray(signal_strengths) * self._config.get("signal_strength_scale", 1)

        self.add_indicator(name="pred_proba", series=self._data["pred_up_proba"], plot=True)
        self.add_indicator(name="next_position", series=next_positions, plot=True)
        self.add_indicator(name="signal_strength", series=signal_strengths, plot=True)


    def next(self):
        return self.get_indicator("next_position")[-1]
```

3.3. Position Manager

There are several position managers implemented in this project.

3.3.1. SellBuyEveryday

This class ignore any signal and just clear its previous position and buy it back every day. When commission is set to zero, this class works with DummySignal to simulate a Buy & Hold strategy.

```
class SellBuyEveryday(StrategyWrapper):
    def next(self, signal):
        self.position.close()
        self.buy()
```

3.3.2. SimplePosition

This class buy or sell based on signal generated from Signal Generator. When received a signal, this class just close its previous opposite trades and buy or sell with all equity, i.e., no position management.

```
class SimplePosition(StrategyWrapper):
    def next(self, signal):
        if signal > 0 and self.position.size <= 0:
            self.position.close()
            self.buy()

        elif signal < 0 and self.position.size >= 0:
            self.position.close()
            self.sell()
```

3.3.3. AtrPosition

This is the main character in position management. It implements the idea of **Turtle** Strategy, that is:

$$TradingSize = \frac{InitCash \times RiskFactor}{N \times ATR}$$ *Equation 3-1*

- Numerator reflects the worst loss I can tolerate in one trade. Normally RiskFactor is 1%~3%.
- Denominator reflects my expectation of price volatility, which is multiple (i.e., N) of Average True Range (ATR)

This position management strategy is implemented in Code 3-1.

- After market close today, if I get a “buy” signal, tomorrow I will close all previous short position and buy more shares.
- After market close today, if I get a “sell” signal, tomorrow I will close all previous long position and sell more shares.
- Trading size is calculated as *Equation 3-1*

Code 3-1 basic position management with ATR

```
class AtrPosition(StrategyWrapper):
    def __init__(self, strategy: Strategy) -> None:
        super().__init__(strategy)

        self._atrs = self.calculate_atrs(period=self._config.get("atr_period", 20))
        self._init_cash = self._strategy.equity  # 0-th bar has no trade yet

    def calculate_atrs(self, period):
        """
        Set the lookback period for computing ATR. The default value
        of 100 ensures a _stable_ ATR.
        """
        hi, lo, c_prev = (
            self._data.High,
            self._data.Low,
            pd.Series(self._data.Close).shift(1),
        )
        tr = np.max([hi - lo, (c_prev - hi).abs(), (c_prev - lo).abs()], axis=0)
        return pd.Series(tr).rolling(period).mean().bfill().values

    def close_prev_trades(self, close):
        left_size = 0
        for trade in self.trades:
            if close == "long" and trade.size > 0:
                trade.close()
            elif close == "short" and trade.size < 0:
                trade.close()
            else:
                left_size += trade.size
        return left_size

    def next(self, signal):
        atr = self._atrs[len(self._data) - 1]

        trade_size = int(self._init_cash * self._config["risk_ratio"] / (self._config["
n_atr"] * atr))
        if trade_size == 0:  # less than one share
            return

        if signal > 0:
            left_size = self.close_prev_trades("short")
            assert left_size >= 0
```

```
42.
43.             if left_size == 0 or (left_size > 0 and self._config["allow_scale_in"]):
44.                 self.buy(size=trade_size)
45.
46.         elif signal < 0:
47.             left_size = self.close_prev_trades("long")
48.             assert left_size <= 0
49.
50.             if left_size == 0 or (left_size < 0 and self._config["allow_scale_in"]):
51.                 self.sell(size=trade_size)
```

3.3.4. AtrPosition Variant: size by signal strength

This class is a variant of basic AtrPosition, which use predicted probability to adjust the trading size each time, like in *Equation 3-2*.

$$TradingSize = \frac{SignalStrength \times InitCash \times RiskFactor}{N \times ATR} \qquad \textit{Equation 3-2}$$

Compare with *Equation 3-1*, the only difference is adding Signal Strength into consideration. Signal Strength is generated by ModelSignal, described in 3.2.3. The basic idea is:

- When predicted up probability is much higher than up-threshold, or much smaller than down-threshold, this prediction has a large signal strength, which reflects model has strong confidence about tomorrow's trading direction, so I should increase tomorrow's trading size.
- Otherwise, if predicted up probability is close to up-threshold or down-threshold, this prediction has a small signal strength, which reflects model has weak confidence about tomorrow's trading direction, so I should decrease tomorrow's trading size.

The implementation can be seen in Code 3-2. Most of the codes are same as Code 3-1, the only difference is multiplying signal strength with cash.

Code 3-2 ATR position management adjusted by signal strength

```
01. def next(self, signal):
02.     atr = self._atrs[len(self._data) - 1]
03.
04.     if self._config.get("adjust_size_by_proba", False):
05.         signal_strength = abs(self.get_indicator("signal_strength")[-1])
06.     else:
07.         signal_strength = 1
08.
09.     trade_size = int(
10.         signal_strength * self._init_cash * self._config["risk_ratio"] / (self._config["n_atr"] * atr)
11.     )
12.     if trade_size == 0:  # less than one share
13.         return
14.
15.     if signal > 0:
16.         left_size = self.close_prev_trades("short")
17.         assert left_size >= 0
18.
19.         if left_size == 0 or (left_size > 0 and self._config["allow_scale_in"]):
20.             self.buy(size=trade_size)
21.
22.     elif signal < 0:
23.         left_size = self.close_prev_trades("long")
24.         assert left_size <= 0
25.
26.         if left_size == 0 or (left_size < 0 and self._config["allow_scale_in"]):
27.             self.sell(size=trade_size)
```

3.3.5. AtrPosition Variant: stop loss & take profit

Basic ATR position exits the position when an opposite signal is encountered. In case model prediction isn't as accurate as I expect, I need stop-loss or take-profit strategy to save my money.

The stop-loss and take-profit strategies are implemented in Code 3-3. The idea is simple, if current price has moved far away from entry price, then it's time to stop loss or take profit.

Code 3-3 implement stop-loss and take-profit strategy

```
def reset_sl_tp(self):
    bar_index = len(self._data) - 1
    atr = self._atrs[bar_index]

    for trade in self.trades:
        lb = trade.entry_price - atr * self._config["n_atr"]
        ub = trade.entry_price + atr * self._config["n_atr"]

        if trade.is_long:
            if self._config["stop_loss"]:
                sl = lb
                # stop loss for long trade, need to set price's lower boundary, the higher the tighter, use max
                trade.sl = max(trade.sl or -np.inf, sl)

            if self._config["take_profit"]:
                tp = ub
                # take profit for long trade, need to set price's upper boundary, the lower the tighter, use min
                trade.tp = min(trade.tp or np.inf, tp)
        else:
            if self._config["stop_loss"]:
                sl = ub
                # stop loss for short trade, need to set price's upper boundary, the lower the tighter, use min
                trade.sl = min(trade.sl or np.inf, sl)

            if self._config["take_profit"]:
                tp = lb
                # take profit for short trade, need to set price's lower boundary, the higher the tigher, use max
                trade.tp = max(trade.tp or -np.inf, tp)
```

The whole strategy code is in Code 3-4. Most of the codes are same as Code 3-1, except updating existing trades' stop-loss & take-profit price in last line.

Code 3-4 ATR position management with stop-loss & take-profit

```python
def next(self, signal):
    atr = self._atrs[len(self._data) - 1]

    if self._config.get("adjust_size_by_proba", False):
        signal_strength = abs(self.get_indicator("signal_strength")[-1])
    else:
        signal_strength = 1

    trade_size = int(
        signal_strength * self._init_cash * self._config["risk_ratio"] / (self._config["n_atr"] * atr)
    )
    if trade_size == 0:  # less than one share
        return

    if signal > 0:
        left_size = self.close_prev_trades("short")
        assert left_size >= 0

        if left_size == 0 or (left_size > 0 and self._config["allow_scale_in"]):
            self.buy(size=trade_size)

    elif signal < 0:
        left_size = self.close_prev_trades("long")
        assert left_size <= 0

        if left_size == 0 or (left_size < 0 and self._config["allow_scale_in"]):
            self.sell(size=trade_size)

    if self._config["stop_loss"] or self._config["take_profit"]:
        self.reset_sl_tp()
```

3.4. Strategy Comparison

Following strategies in Table 3-2 are backtested on both validation and test dataset. Notice ModelSignal's up-threshold & down-threshold are from section 3.1.

3.4.1. Backtests summary

Table 3-2 all strategies backtested

Strategy	**Signal generator**	**Position management**	**Description**
buy & hold	DummySignal	SellBuyEveryday with commission set to 0	Simulate Buy & Hold strategy as a benchmark
SMA+SimplePosition	SmaSignal	SimplePosition	Simulate classical Moving Average strategy as a benchmark
Model+SimplePos	ModelSignal	SimplePosition	Backtest signal from model prediction with no position management
Model+AtrPos	ModelSignal	AtrPosition	Backtest signal from model prediction, and manage position size using ATR like Turtle strategy

Model+AtrPos+AdjustSize	ModelSignal	AtrPosition with adjust-size enabled	Backest model prediction and ATR position management, also adjust position size with signal strength
Model+AtrPos+SLTP	ModelSignal	AtrPosition with stop-loss & take profit enabled	Backest model prediction and ATR position management, also stop-loss or take-profit when current price is far away from entry price

Backtest configurations and metrics on validation dataset are summarized in Table 3-3.

Table 3-3 backtest configuration and metric on validation set

config or metric	**buy & hold**	**SMA+SimplePosition**	**Model+SimplePos**	**Model+AtrPos**	**Model+AtrPos+AdjustSize**	**Model+AtrPos+SLTP**
00_bt_task	16171237	16171240	16171242	16171244	16171245	16171247
01_adjust_size_by_proba			TRUE	TRUE	TRUE	TRUE
01_allow_scale_in			TRUE	TRUE	TRUE	TRUE
01_commission	0	0.002	0.002	0.002	0.002	0.002
01_down_threshold			0.33	0.33	0.33	0.33
01_init_cash	1000000	1000000	1000000	1000000	1000000	1000000
01_n_atr			3	3	3	3
01_pos_class	SellBuyEveryday	SimplePosition	AtrPosition	AtrPosition	AtrPosition	AtrPosition
01_risk_ratio			0.03	0.03	0.03	0.03
01_role	test	test	val	val	val	val
01_sig_class	DummySignal	SmaSignal	ModelSignal	ModelSignal	ModelSignal	ModelSignal
01_signal_strength_scale			0.9	0.9	0.9	0.9
01_stop_loss			TRUE	TRUE	TRUE	TRUE
01_take_profit			TRUE	TRUE	TRUE	TRUE
01_up_threshold			0.37	0.37	0.37	0.37
11_Start	2011/12/5 00:00	2011/12/5 00:00	2011/12/5 00:00	2011/12/5 00:00	2011/12/5 00:00	2011/12/5 00:00
12_End	2017/9/15 00:00	2017/9/15 00:00	2017/9/15 00:00	2017/9/15 00:00	2017/9/15 00:00	2017/9/15 00:00
13_Duration	2111 days 00:00:00	2111 days 00:00:00	2111 days 00:00:00	2111 days 00:00:00	2111 days 00:00:00	2111 days 00:00:00
14_Exposure Time [%]	99.86254296	94.02061856	99.86254296	89.14089347	91.61512027	82.61168385
15_Equity Final [$]	1986219.433	569449.2172	2241577.174	2013203.897	1829404.212	1589936.603
16_Equity Peak [$]	1986219.433	1099523.198	2492364.366	2172186.747	1982095.337	1722519.151
17_Return [%]	98.62194329	-43.05507828	124.1577174	101.3203897	82.9404212	58.99366031

18_Buy & Hold Return [%]	98.89188181	98.89188181	98.89188181	98.89188181	98.89188181	98.89188181
19_Return (Ann.) [%]	12.62040549	-9.291953405	15.00439946	12.8839265	11.02760048	8.362284494
20_Volatility (Ann.) [%]	13.99716671	10.73781874	14.27134895	11.37514852	11.32353945	11.12615824
21_Sharpe Ratio	0.901640007	0	1.051365187	1.132638091	0.973865153	0.751587773
22_Sortino Ratio	1.470840691	0	1.845870691	1.982185777	1.637963843	1.192123044
23_Calmar Ratio	0.89177051	0	1.153425689	1.099377837	0.880462818	0.545571475
24_Max. Drawdown [%]	-14.15207764	-50.67514432	-13.00855322	-11.71928892	-12.52477703	-15.32756912
25_Avg. Drawdown [%]	-1.384479514	-5.310928635	-1.681510531	-1.223996508	-1.281245046	-1.494460582
26_Max. Drawdown Duration	417 days 00:00:00	1578 days 00:00:00	288 days 00:00:00	288 days 00:00:00	288 days 00:00:00	352 days 00:00:00
27_Avg. Drawdown Duration	18 days 00:00:00	141 days 00:00:00	20 days 00:00:00	18 days 00:00:00	20 days 00:00:00	29 days 00:00:00
28_# Trades	1453	74	118	103	106	159
29_Win Rate [%]	55.7467309	36.48648649	68.6440678	68.93203883	67.9245283	60.37735849
30_Best Trade [%]	3.738887707	6.977819486	6.636515596	9.107005483	6.636515596	5.12068859
31_Worst Trade [%]	-3.439479107	-6.989309775	-5.452592418	-5.452592418	-5.452592418	-4.862834086
32_Avg. Trade [%]	0.047149636	-0.758591617	0.686811581	0.962393306	0.75902957	0.387671919
33_Max. Trade Duration	5 days 00:00:00	97 days 00:00:00	141 days 00:00:00	128 days 00:00:00	141 days 00:00:00	57 days 00:00:00
34_Avg. Trade Duration	2 days 00:00:00	27 days 00:00:00	18 days 00:00:00	21 days 00:00:00	20 days 00:00:00	12 days 00:00:00
35_Profit Factor	1.206171499	0.510780168	3.249580374	4.029986037	3.107714962	1.525609919
36_Expectancy [%]	0.049863146	-0.723207532	0.701243279	0.985623309	0.778283786	0.416509812
37_SQN	2.405798211	-2.33327435	3.710244382	4.361696613	3.884867048	2.138266415

Backtest configurations and metrics on test dataset are summarized in Table 3-4.

Table 3-4 backtest configurations and metrics on test dataset

config or metric	**buy & hold**	**SMA+SimplePosition**	**Model+SimplePos**	**Model+AtrPos**	**Model+AtrPos+AdjustSize**	**Model+AtrPos+SLTP**
00_bt_task	16171238	16171241	16171248	16171249	16171251	16171252
01_adjust_size_by_proba			TRUE	TRUE	TRUE	TRUE
01_allow_scale_in			TRUE	TRUE	TRUE	TRUE
01_commission	0	0.002	0.002	0.002	0.002	0.002
01_down_threshold			0.32	0.32	0.32	0.32
01_init_cash	1000000	1000000	1000000	1000000	1000000	1000000
01_n_atr			3	3	3	3
01_pos_class	SellBuyEveryday	SimplePosition	AtrPosition	AtrPosition	AtrPosition	AtrPosition
01_risk_ratio			0.03	0.03	0.03	0.03

01_role	test	test	test	test	test	test
01_sig_class	DummySignal	SmaSignal	ModelSignal	ModelSignal	ModelSignal	ModelSignal
01_signal_strength_scale			0.9	0.9	0.9	0.9
01_stop_loss			TRUE	TRUE	TRUE	TRUE
01_take_profit			TRUE	TRUE	TRUE	TRUE
01_up_threshold			0.35	0.35	0.35	0.35
11_Start	2017/10/13 00:00	2017/10/13 00:00	2017/10/13 00:00	2017/10/13 00:00	2017/10/13 00:00	2017/10/13 00:00
12_End	2023/7/27 00:00	2023/7/27 00:00	2023/7/27 00:00	2023/7/27 00:00	2023/7/27 00:00	2023/7/27 00:00
13_Duration	2113 days 00:00:00	2113 days 00:00:00	2113 days 00:00:00	2113 days 00:00:00	2113 days 00:00:00	2113 days 00:00:00
14_Exposure Time [%]	99.86254296	94.50171821	99.86254296	79.72508591	83.78006873	67.90378007
15_Equity Final [$]	1774273.932	866064.8531	1573548.992	1772729.31	1794993.721	1028814.988
16_Equity Peak [$]	1875601.543	1036792.716	1720514.962	1880542.201	1897960.059	1075612.875
17_Return [%]	77.42739316	-13.39351469	57.35489925	77.27293103	79.49937214	2.881498788
18_Buy & Hold Return [%]	77.71673234	77.71673234	77.71673234	77.71673234	77.71673234	77.71673234
19_Return (Ann.) [%]	10.44075158	-2.459721947	8.168012695	10.42409351	10.66305427	0.493221007
20_Volatility (Ann.) [%]	23.23816654	19.01310776	22.64127901	19.58531325	18.80712261	16.4482146
21_Sharpe Ratio	0.449293259	0	0.360757565	0.532240326	0.566968935	0.029986295
22_Sortino Ratio	0.695391218	0	0.563034914	0.850919156	0.919103984	0.040777415
23_Calmar Ratio	0.307794242	0	0.261485066	0.370242099	0.403898311	0.01555608
24_Max. Drawdown [%]	-33.92120504	-34.64626769	-31.23701414	-28.1548034	-26.4003438	-31.705995
25_Avg. Drawdown [%]	-2.102551998	-34.64626769	-3.894958328	-2.402871949	-2.029578096	-8.290363883
26_Max. Drawdown Duration	570 days 00:00:00	1995 days 00:00:00	695 days 00:00:00	292 days 00:00:00	292 days 00:00:00	582 days 00:00:00
27_Avg. Drawdown Duration	23 days 00:00:00	1995 days 00:00:00	47 days 00:00:00	26 days 00:00:00	24 days 00:00:00	169 days 00:00:00
28_# Trades	1453	60	155	187	191	178
29_Win Rate [%]	55.88437715	31.66666667	65.80645161	66.31016043	65.96858639	61.23595506
30_Best Trade [%]	6.22003317	13.49327757	4.215255281	11.51032253	24.26246282	20.98638166
31_Worst Trade [%]	-6.891986959	-13.63068545	-7.405970226	-7.330119352	-7.330119352	-14.46270628
32_Avg. Trade [%]	0.040391966	-0.240135165	0.293350944	0.559488184	0.617036536	0.142430098
33_Max. Trade Duration	4 days 00:00:00	115 days 00:00:00	111 days 00:00:00	104 days 00:00:00	104 days 00:00:00	49 days 00:00:00
34_Avg. Trade Duration	2 days 00:00:00	34 days 00:00:00	14 days 00:00:00	14 days 00:00:00	14 days 00:00:00	9 days 00:00:00
35_Profit Factor	1.127635047	0.934411081	1.530120998	2.04975883	2.149750419	1.167600038
36_Expectancy [%]	0.046488458	-0.122780161	0.3121922	0.584202843	0.652932023	0.238295987
37_SQN	1.409653754	-0.430318492	1.943742798	3.772514318	3.669413081	0.097681733

3.4.2. Sharpe & Sortino Ratio comparison

Sharpe Ratio is the most important performance metric for me to evaluate strategies.

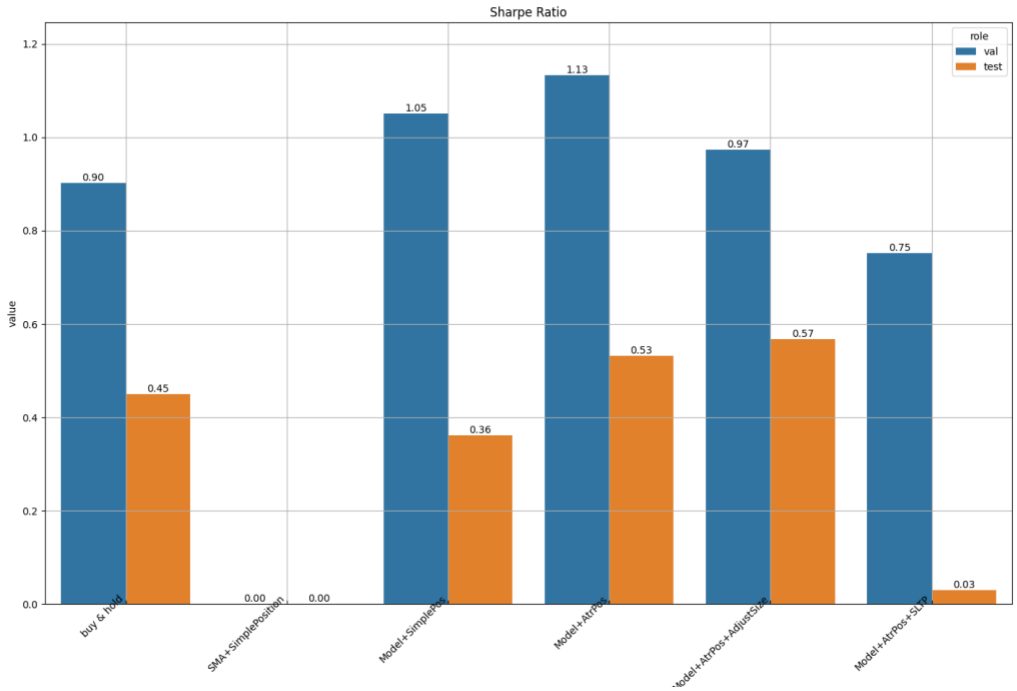

Figure 3-2 compare Sharpe Ratio of each strategy

From above picture, I have following observations:

- Classical SMA strategy suffer great loss on S&P 500.
- Model+SimplePos, i.e., using signal from model prediction with no position management, is better than Buy & Hold benchmark in validation and a little worse in test.
- Model+AtrPos, i.e., using signal from model prediction with ATR-based position management, is better than Buy & Hold benchmark in both validation and test.
- Model+AtrPos+AdjustSize, i.e., using signal from model prediction and adjusting position size based on signal strength, achieves best Sharpe Ratio on test dataset. **This may prove that using model prediction is not only useful in generating trading signal, but also helpful in adjusting the trading size**.
- Model+AtrPos+SLTP, i.e., using signal from model prediction stop-loss & take-profit when price moves too far away from entry price, is worse than benchmark on test. **This indicates that model prediction is not only useful in entering the position, but also helpful in existing the position**, compared with some empirical rules.

Sortino Ratio is similar to Sharpe Ratio, but replace the denominator with standard deviation of negative returns. Sortino Ratio is plotted in Figure 3-3, from which we can tell that, model-prediction based strategies normally perform better than benchmarks, and

Model+AtrPos+AdjustSize achieves best Sortino Ratio on test dataset. **This also proves model predictions' advantages in both generating trading signal and adjusting trading size**.

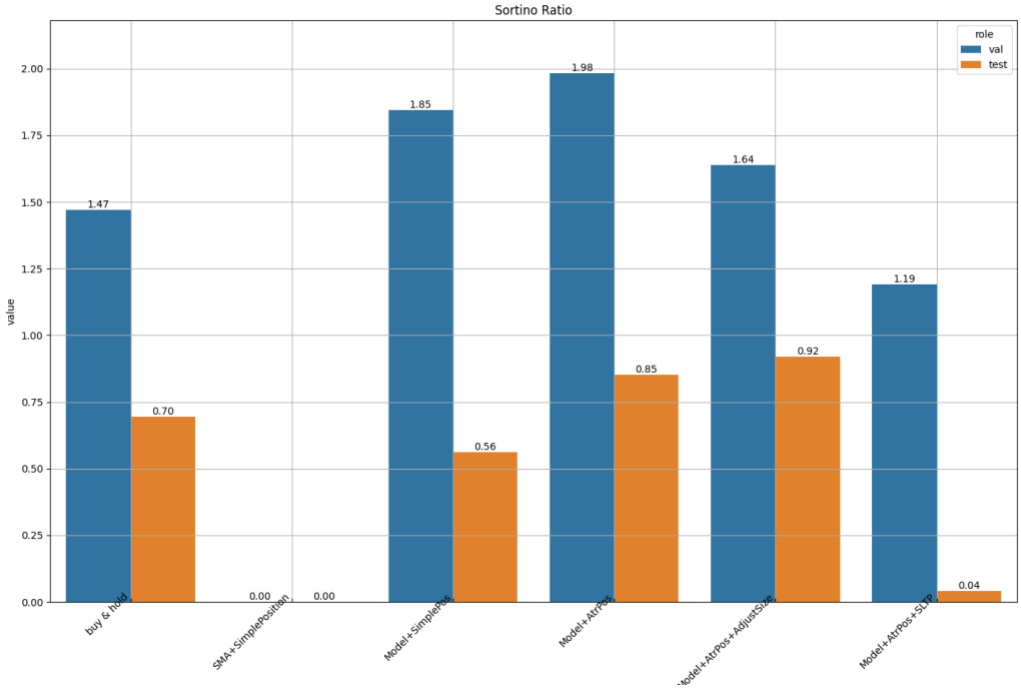

Figure 3-3 Compare Sortino Ratio of each strategy

In Figure 3-2 and Figure 3-3, I can also observe drift problem:

- All strategies have much higher Sharpe / Sortino Ratio in validation dataset than in test dataset.
- Model+AtrPos has largest Sharpe Ratio (1.13) and Sortino Ratio (1.98) in validation set. However, there is no strategy whose Sharpe Ratio is greater than 1 in test dataset.
- **Although disappointing, poor Sharpe Ratio in test dataset is totally understandable after all, consider the unprecedented COVID-19 crisis starting at 2019**.

3.4.3. Max-drawdown comparison

Max-drawdown ratio is plotted in Figure 3-4.

- Classical SMA strategy and "Model+AtrPos+SLTP" doesn't perform well.
- Model+SimplePos / Model+AtrPos / Model+AtrPos+AdjustSize are all better than Buy & Hold benchmark.
- Model+AtrPos+AdjustSize achieves the lowest Max-drawdown.

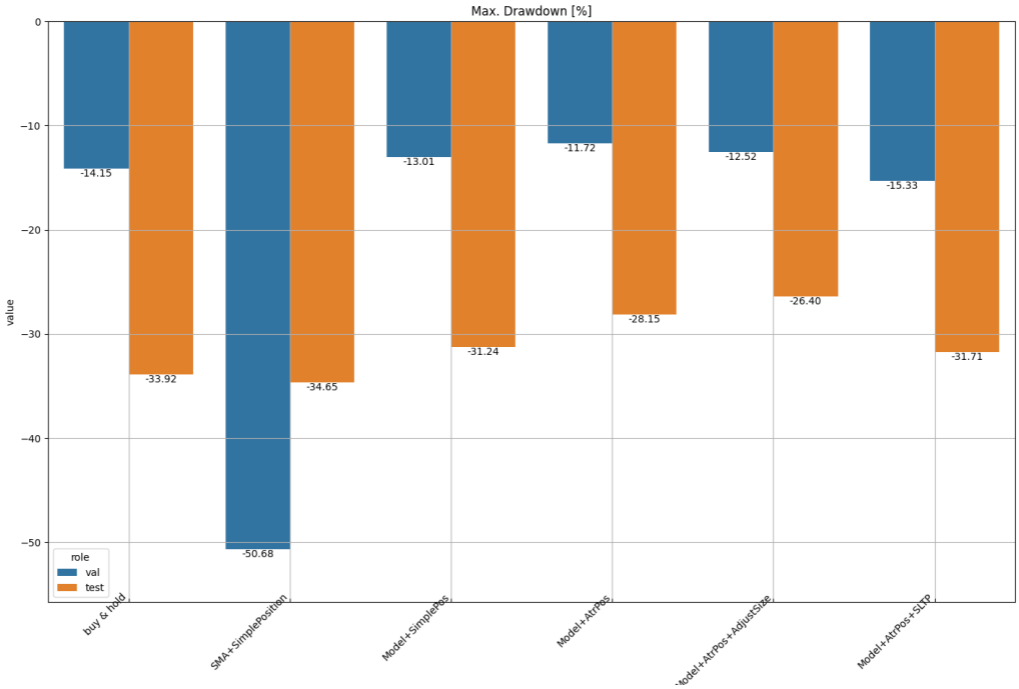

Figure 3-4 compare Max-drawdown of each strategy

Max-drawdown duration is plotted in Figure 3-5.

- Classical SMA strategy and “Model+AtrPos+SLTP” have very long max-drawdown duration, making them not suitable to be deployed into real market.
- Model+SimplePos has shorter max-drawdown duration in validation dataset, but has longer duration in test dataset.
- After applying position management, Model+AtrPos & Model+AtrPos+AdjustSize have reduced their max-drawdown duration in both validation and test dataset.

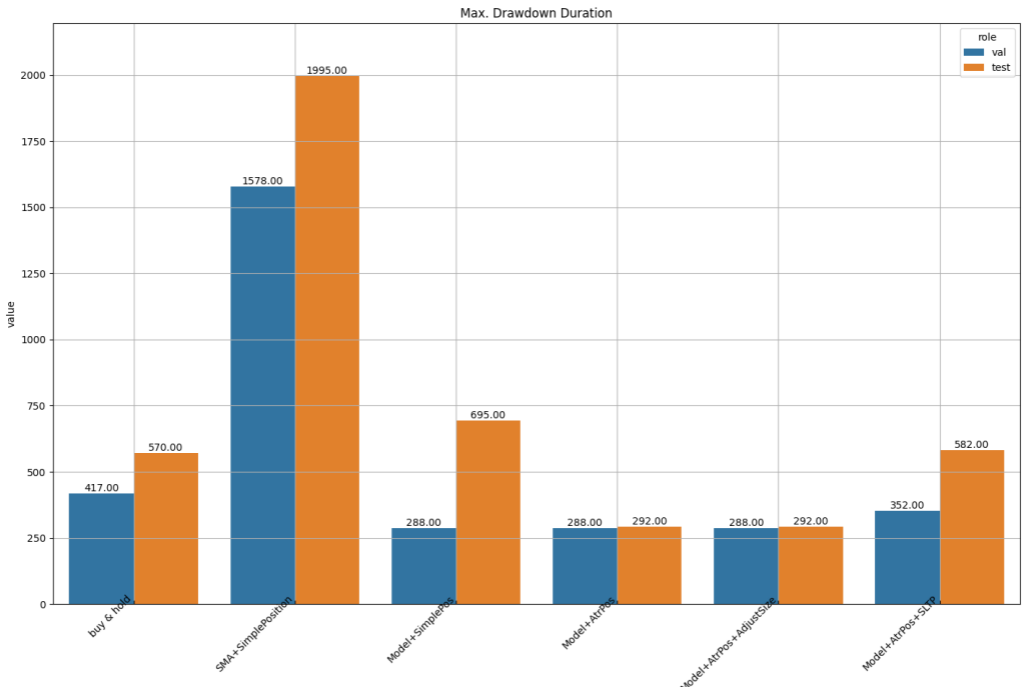

Figure 3-5 Compare Max-drawdown Duration of each strategy

3.4.4. Calmar Ratio comparison

Calmar Ratio in Figure 3-6 is a risk-adjusted measure. It's calculated by dividing average annual rate of return by its maximum drawdown. The larger the Calmar ratio is, the better your trading system is.

- Model+SimplePos / Model+AtrPos / Model+AtrPos+AdjustSize have larger Calmar ratio, and Model+AtrPos+AdjustSize achieves best Calmar ratio on test dataset.
- Model+SimplePos / Model+AtrPos have Calmar ratio larger than 1 which is considered good. However, it’s disappointing to find all strategies’ Calmar ratios are smaller than 1. This is understandable considering the COVID-19 crisis happened during the test period.

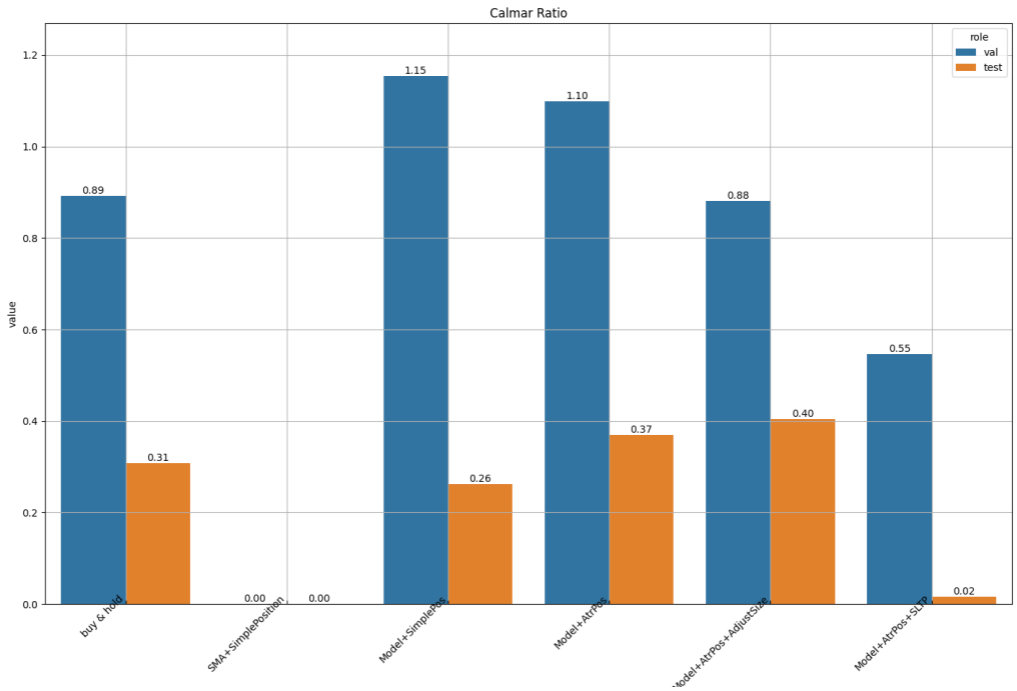

Figure 3-6 compare Calmar Ratio of each strategy

3.4.5. Win Rate comparison

All Model-based strategies have nearly 70% win-rate, which are much higher than Buy & Hold and SMA benchmarks. The high win-rate indicates the model did learn some patterns to distinguish next day's trading direction. (But still need careful selection of thresholds.)

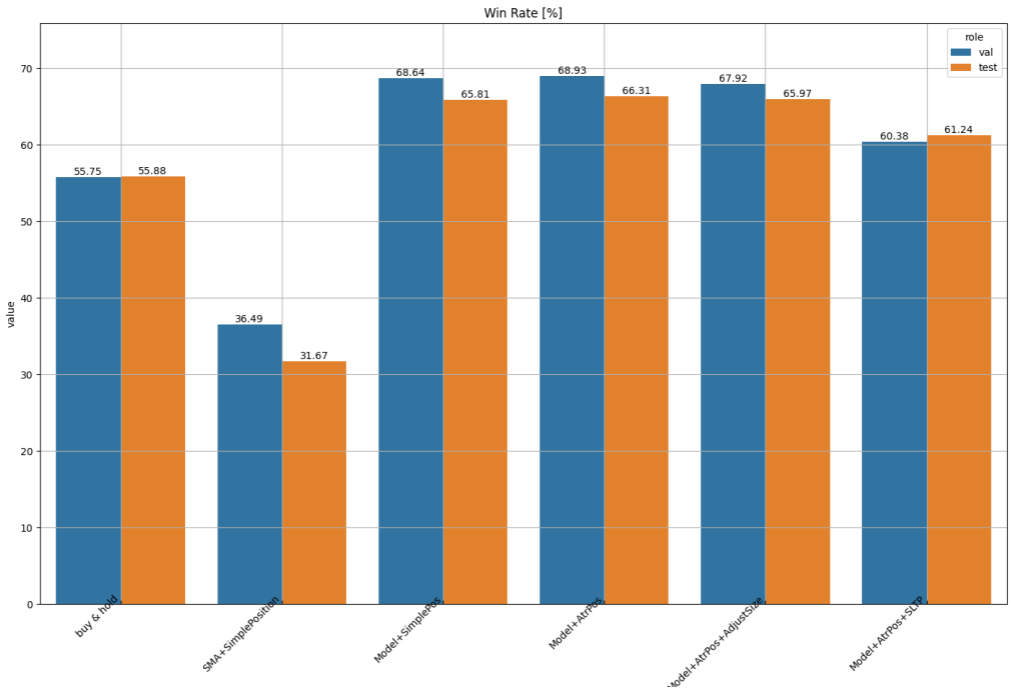

Figure 3-7 Compare Win-Rate of each strategy

3.4.6. Profit comparison

Profit Factor defined by backtesting.py as sum of positive returns divided by sum of negative returns. From Figure 3-8, I can see that Model+SimplePos / Model+AtrPos / Model+AtrPos+AdjustSize all have much higher Profit Factor than benchmarks, and Model+AtrPos+AdjustSize achieves best Profit Factor on test dataset.

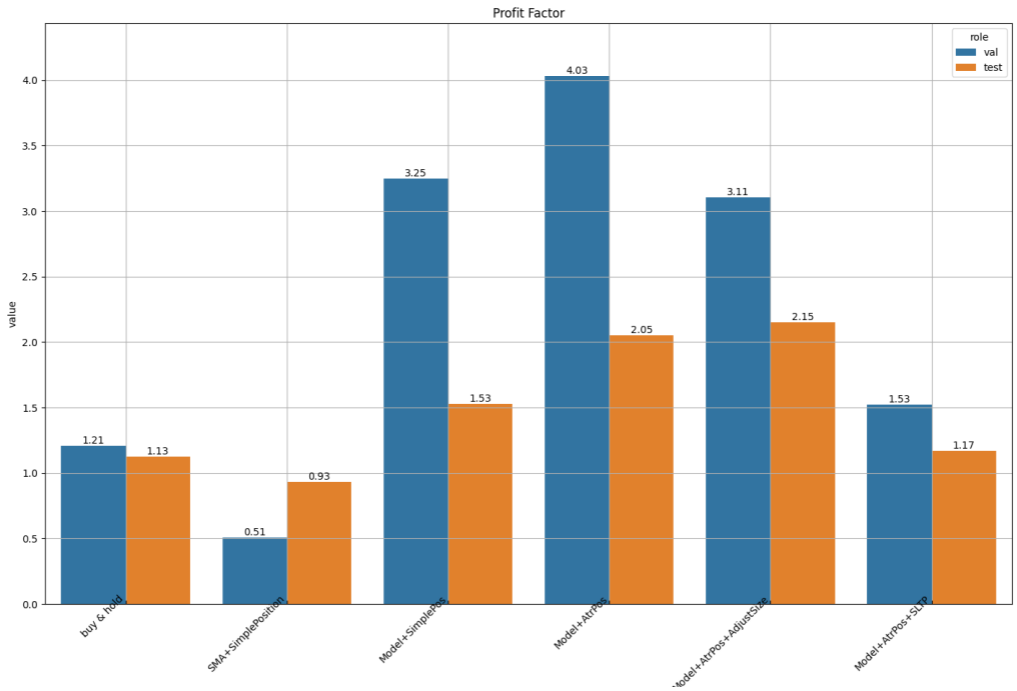

Figure 3-8 Compare Profit Factor of each strategy

3.4.7. System Quality Number comparison

System Quality Number (SQN) is defined as Equation *3-3*.

$$SQN = \sqrt{\#Trades} \times \frac{Mean(PnL)}{StdDev(PnL)} \qquad \textit{Equation 3-3}$$

- It is the ratio between PnL's mean and its standard deviation, multiplied by the square root of the number of trades.
- PnL's mean measures your system's profitability, while the standard deviation measures consistency.
- A large backtest sample size improves the statistical significance of your metrics
- The larger SQN is, the better your trading system is.

Figure 3-9 tells that Model+SimplePos & Model+AtrPos & Model+AtrPos+AdjustSize have much larger SQN than benchmarks. After applying position management, Model+AtrPos & Model+AtrPos+AdjustSize improves SQN even more.

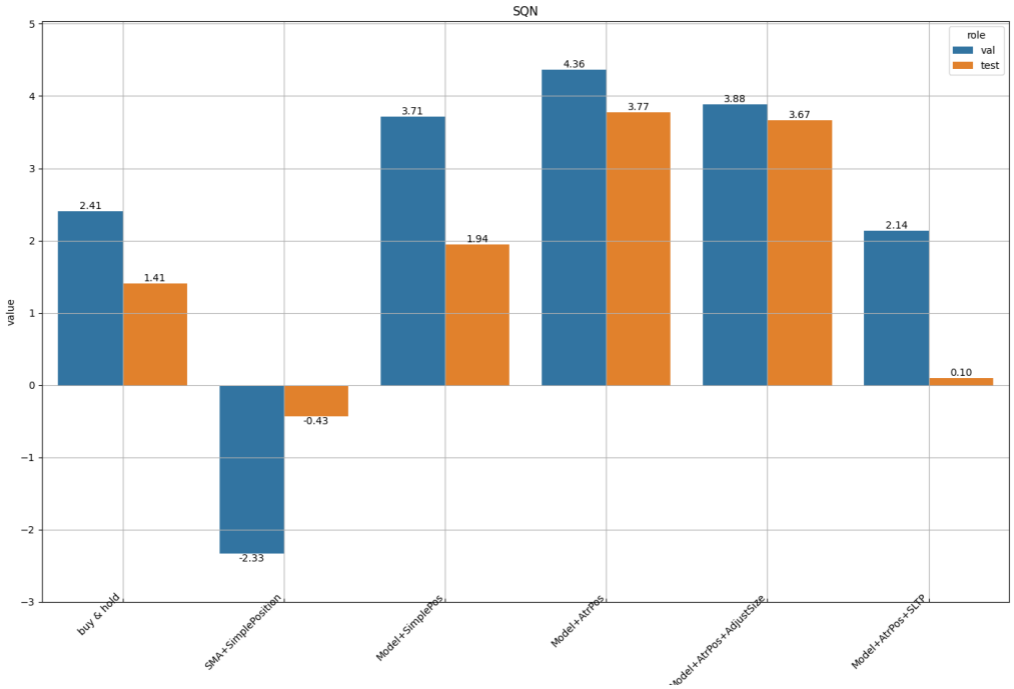

Figure 3-9 Compare SQN of each strategy

4. Summary & Conclusion

The first part of this project is training a good binary classification model that can distinguish moving-up cases from moving-down. Finally, I choose the "***improved_rnn***" model, which uses LSTM to model timeseries and improved by using latest technical indicators. This model can have AUCs near 0.6, which is good enough considering the stochastic nature and heavy noise in stock price movement. When model prediction is backtested, all model-based strategies have win-rate as high as near 70%, which also reflects the model did learnt some patterns and can distinguish moving-up from moving-down the next day.

The second part of this project is translating model prediction into real trading strategy. Finally, I choose "***Model+AtrPos+AdjustSize***" strategy, which has following 3 key points:

- Noticing the distribution drift problem, I carefully select the thresholds for validation datasets and make a reasonable guess about the thresholds for test dataset. By using these thresholds, model prediction probabilities are translated into buy or sell signal for next day.
- I implement the position management method by following Turtle Strategy. This method calculates trading size by considering worst loss I can tolerate and my expectation about price volatility.
- Besides used in generating trading signal, model prediction is also used in adjusting the trading size. Model prediction scores can reflect confidence about next day's trading direction. After trading size is calculated by Turtle method, I increase it if model prediction is strong and decrease it when model prediction is weak.

Detailed configurations and metrics of this strategy is summarized in Table 4-1.

Table 4-1 configurations and metrics of Strategy "Model+AtrPos+AdjustSize"

config or metric	**Model+AtrPos+AdjustSize On Validation**	**Model+AtrPos+AdjustSize On Test**
00_bt_task	16171245	16171251
01_adjust_size_by_proba	TRUE	TRUE
01_allow_scale_in	TRUE	TRUE
01_commission	0.002	0.002
01_down_threshold	0.33	0.32
01_init_cash	1000000	1000000
01_n_atr	3	3
01_pos_class	AtrPosition	AtrPosition
01_risk_ratio	0.03	0.03
01_role	val	test
01_sig_class	ModelSignal	ModelSignal
01_signal_strength_scale	0.9	0.9
01_stop_loss	TRUE	TRUE
01_take_profit	TRUE	TRUE
01_up_threshold	0.37	0.35
11_Start	2011/12/5 00:00	2017/10/13 00:00
12_End	2017/9/15 00:00	2023/7/27 00:00
13_Duration	2111 days 00:00:00	2113 days 00:00:00
14_Exposure Time [%]	91.61512027	83.78006873
15_Equity Final [$]	1829404.212	1794993.721
16_Equity Peak [$]	1982095.337	1897960.059
17_Return [%]	82.9404212	79.49937214
18_Buy & Hold Return [%]	98.89188181	77.71673234
19_Return (Ann.) [%]	11.02760048	10.66305427
20_Volatility (Ann.) [%]	11.32353945	18.80712261
21_Sharpe Ratio	0.973865153	0.566968935
22_Sortino Ratio	1.637963843	0.919103984
23_Calmar Ratio	0.880462818	0.403898311
24_Max. Drawdown [%]	-12.52477703	-26.4003438
25_Avg. Drawdown [%]	-1.281245046	-2.029578096
26_Max. Drawdown Duration	288 days 00:00:00	292 days 00:00:00
27_Avg. Drawdown Duration	20 days 00:00:00	24 days 00:00:00
28_# Trades	106	191
29_Win Rate [%]	67.9245283	65.96858639
30_Best Trade [%]	6.636515596	24.26246282

31_Worst Trade [%]	-5.452592418	-7.330119352
32_Avg. Trade [%]	0.75902957	0.617036536
33_Max. Trade Duration	141 days 00:00:00	104 days 00:00:00
34_Avg. Trade Duration	20 days 00:00:00	14 days 00:00:00
35_Profit Factor	3.107714962	2.149750419
36_Expectancy [%]	0.778283786	0.652932023
37_SQN	3.884867048	3.669413081

Backtest results proves the effectiveness of my strategies. For validation dataset, model-based strategies have Sharpe Ratio larger than 1 and Sortino Ratio approaching 2. Although on test dataset, model-based strategies don't perform as well as in validation dataset, after all, this is understandable considering the 2019 COVID-19 crisis. And *Model+AtrPos+AdjustSize* achieves best Sharp & Sortino Ratio on test dataset, which proves model predictions are effective in both generating the trading signal and adjusting the trading size.

However, there is still controversy left, that is the selection of decision thresholds. "not bad" AUCs reflect the model can give moving-up cases high score and rank them before moving-down cases. However, we still need thresholds to translate model predicted probability into concrete buy or sell signal. But, due to the distribution drifting, thresholds derived from history may not be suitable for future, causing losing trading opportunities. **In this project, I notice the US market is bullish from long-term perspective, and make a reasonable guess that the decision thresholds should become lower as time goes by. This guess is subjective and biased, and needs more backtesting and forward testing before deployed into real market**.

5. Future Work

Further work for improving the prediction mode includes common approaches like:

- *Add more features*. For example, more technical indicators can be derived from current timeseries. Also, information from other timeseries, such as VIX index, may be helpful.
- *More careful feature engineering*. For example, in this project I use log & standardization to scale skewed features, and I can try sklearn's RobustScaler next time, making this process more robust to outliers.
- *More complex neural network structure*. For example, I can stack RNN on top of CNN, or use Transform architecture to derive patterns from timeseries. Also, I can import Dropout and Batch Normalization to reduce overfitting.

Besides above common approaches, I tried following two directions. Although no promising results are achieved yet, I firmly believe both directions are worth paying more attention and investing more efforts.

- In this project, the last day used in training is 2011-11-04, that means the model hasn't been updated for more than 10 years. Continuous training is a common practice in internet industry, which always feed the model with latest information and keep the model fresh and updated. However, I tried such continuous-training plan once in this

project, and didn't get any improvement. Maybe the plain-sailing validation period make the model forget how to deal with turbulence like in 2008 subprime crisis, and suffer great loss during 2019 COVID-19 crisis.

- In this project, all examples increasing more than 0.25% are labeled as positive, and are treated by model equally. However, in reality, major price changes impact our PnL far more significantly than minor price change, which deserves more attention from model. Hence, a carefully-designed sample-reweight plan may be helpful.

However, I am a little pessimistic about the efforts paid on model improvement. Due to inevitable distribution drifting problem, adding more feature and making model complex can only lead to overfitting. **What we really need is counter measure for distribution drifting as time goes by, which help us make a more accurate guess about future decision thresholds**.

Future work for improving strategies includes:

- The first priority is finding a method to assist me guessing future decision thresholds more wisely. But I know this is very much difficult.
- Due to inevitable distribution drifting, my strategy isn't guaranteed to repeat its success in the future. Although "*Model+AtrPos+SLTP*" performs badly in this project, carefully-designed stop-loss & take-profit plans are still necessary in case model predictions fail in the future due to drifting.
- Adjusting trading size based on model prediction yields promising improvement. I will try more methods exploiting model prediction in position management, for example, Kelly criterion.