Pairs Trading Strategy

Fernando Montes

Pairs trading is a market-neutral trading strategy that matches a long position with a short position in a pair of highly correlated instruments. The strategy profits from the difference between the two instruments when the long position goes up more than the short, or the short position goes down more than the long. This report documents strategies developed in order to maximize profits using pair trading. In the case considered here, pricing of two securities, ABC and XYZ, was used.

The document is organized as follows: The data provided for this analysis is described in Sec. 1. The analysis included the development of 10 strategies, described in Sec. 2. The optimal capital allocation using a recommended strategy is described in Sec. 3, and scaling of the model is described in Sec. 4.

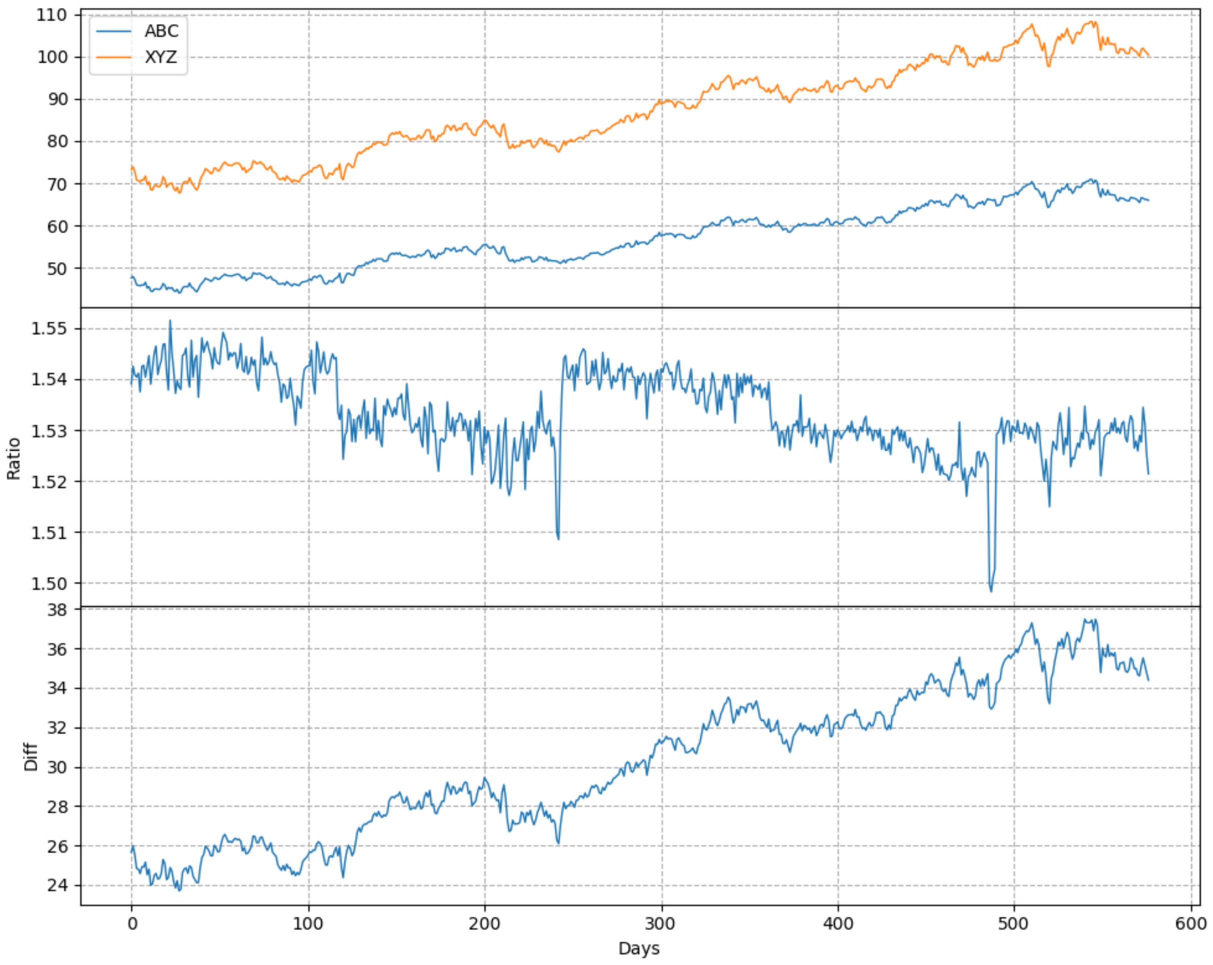
1 Data exploration

```
In [309]: from src.models.__init__ import *
```

ABC price, XYZ price, ratio of the prices and their difference in the time period from 2016-01-04 to 2018-05-02 were provided to develop the strategies:

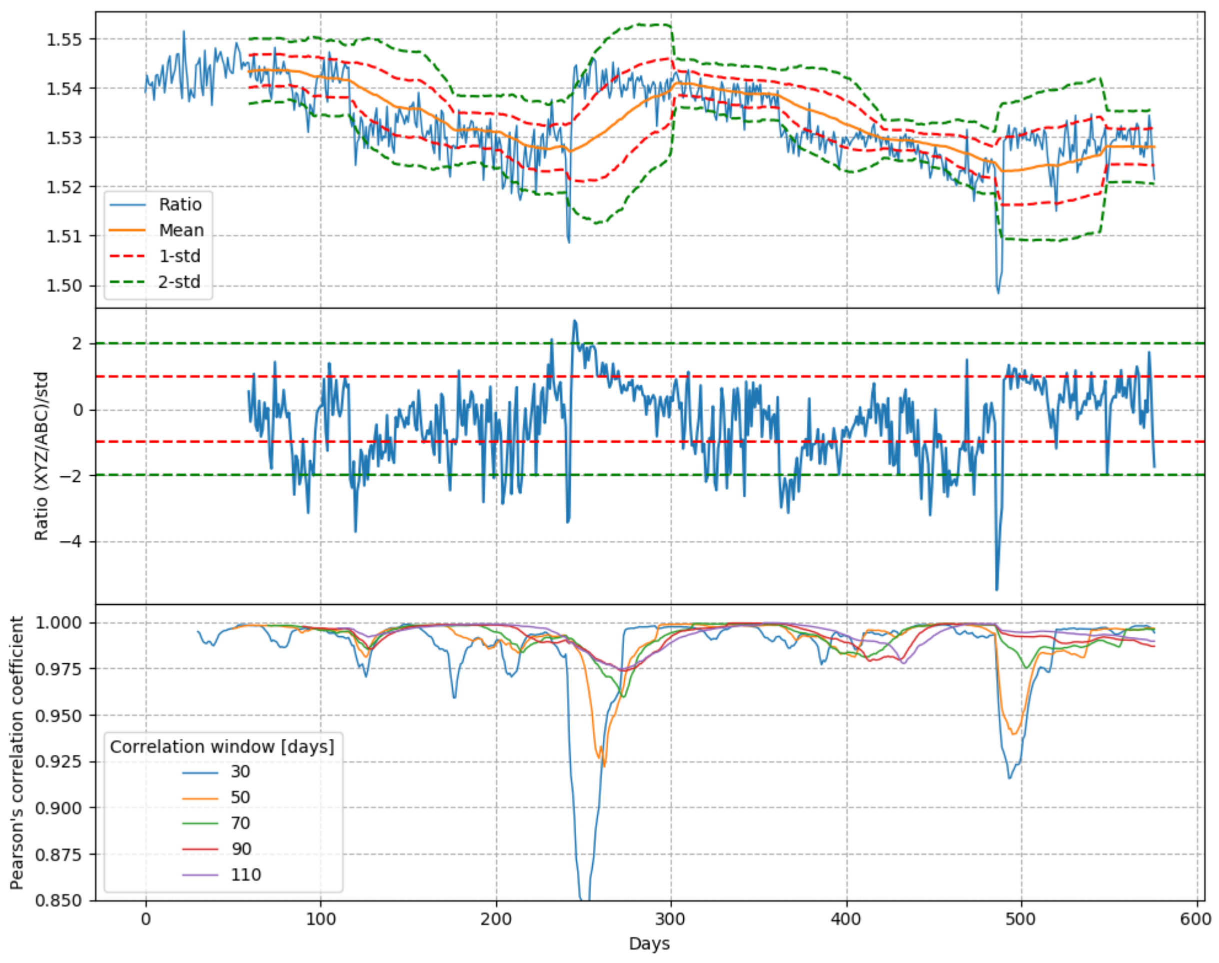
```
In [319]: seriesFull = pd.read_csv('../data/raw/pairs.csv', header=0)
          print(seriesFull.head())
          firstLook(seriesFull)
```

```
         DATE        ABC        XYZ     RATIO       DIFF
0  2016-01-04  47.595426  73.253267  1.539082  25.657841
1  2016-01-05  47.878171  73.851865  1.542496  25.973693
2  2016-01-06  47.202725  72.730107  1.540803  25.527382
3  2016-01-07  45.946080  70.774909  1.540391  24.828829
4  2016-01-08  45.820415  70.615571  1.541138  24.795155
```



Both the ratio XYZ/ABC and their difference are not stationary. Since in a traditional approach, stationarity is desired, a rolling estimate of the mean of the ratio is plotted below. A normalized ratio, defined as the ratio XYZ/ABC divided by the standard deviation of the rolling mean, is stationary and it is also plotted. The normalized ratio can signal when it might be a good time to enter or exit the market.

```
In [7]: rollingEstimates(seriesFull, 60) # Using a 60-day rolling window
```



It is observed that a rolling Pearson's coefficient is close to 1 over the whole range which indicates that the pair XYZ and ABC are truly correlated. In addition, the "recovery time" of the normalized ratio once it exceeds 1-std (or even 2-std) is of the order of days which may indicate that the market positions may need to be changed on a daily basis.

2 Modeling

The data from 2016-01-04 to 2017-08-21 was randomly separated between train (70%) and validation data (30%). The train data was used to optimize the parameters of a given algorithm. The validation data was used to optimize hyper-parameters. The annualized rate of return (AR) was used as the metric to decide how good a strategy was. The AR was calculated by

$$AR = 100 \times \left(\prod_{i=2}^{N} (1 + r_i)^{256/N} - 1 \right),$$

where r_i is the return between day i and $i - 1$ and N is the total number of days. The expected AR and risk for each strategy were quantified using a Monte Carlo (MC) simulation with 10000 iterations. In each MC iteration, an AR was calculated by randomly selecting (with replacement) the daily returns on the unseen data from 2017-08-22 to 2018-05-02 (test data) after applying a given strategy. The AR mean and standard deviation of the MC distribution correspond to the expected AR and associated risk of the strategy, respectively. All transactions (buy or sell XYZ and ABC) are assumed to have no cost.

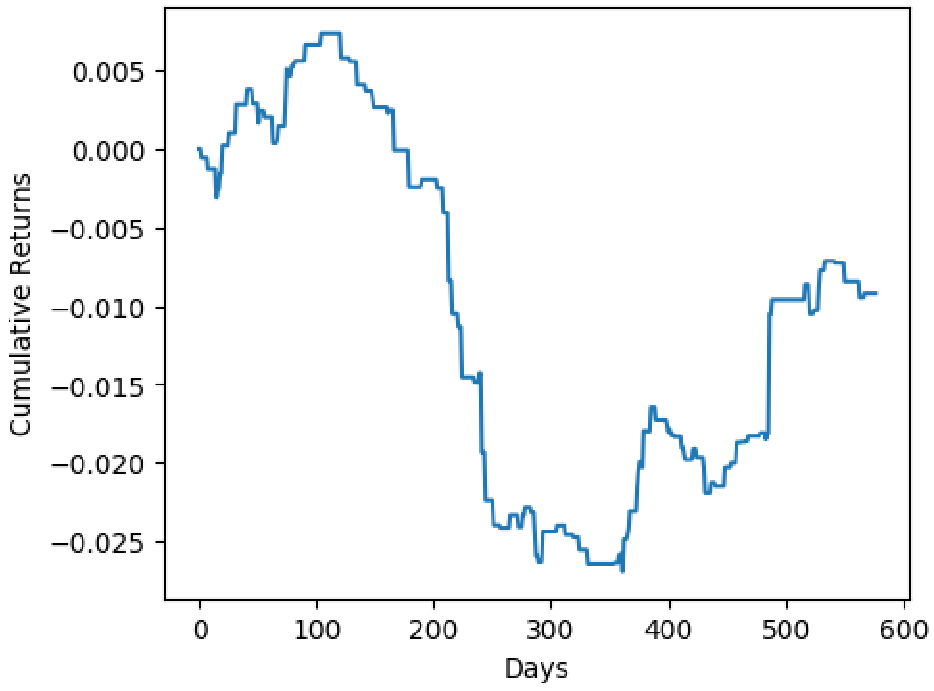
Comment about the code: The machine learning algorithms used in our analysis employ either native Scikit-Learn base estimator classes or have been written based on them. Two of the algorithms employed (RNN and LSTM) used the TensorFlow framework, and therefore two new classes, RNNClassifier and LSTMClassifier (Scikit-Learn compatible) were written. The strategies discussed here were coded using unique classes that rely heavily on parent inheritance. Preprocessing of the input data consisted of renormalizing the prices, ratio and price difference. In addition, one of the strategies uses the normalized ratio described earlier as an additional feature.

2.1 Random strategy:

A strategy that uses random positions (either long ABC, short XYZ or short ABC, long XYZ) was studied to understand the the downside of using pair trading. Even though it does not constitute a valid strategy, it provides insight for future strategies. An entry point is where a pair strategy position is made. An exit point corresponds to a position that returns to neutral. The positions are hedged only at the entry points (**not** continuously hedged if the position is maintained). This decision was made since it is likely than in a real scenario there would be a cost per transaction.

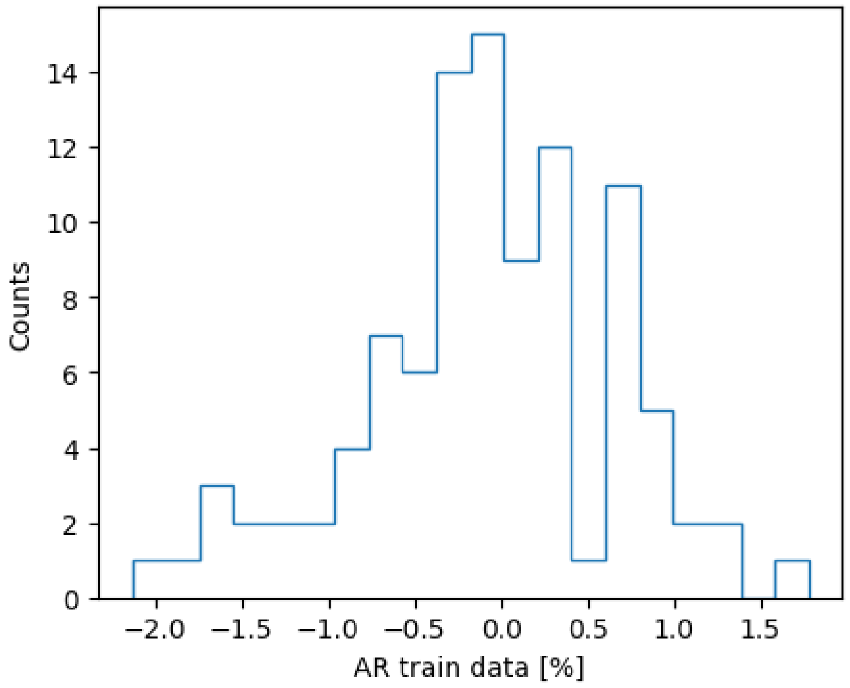
```
In [44]: random_str = randomStrategy()
         random_str.apply()
```

```
In [45]: random_str.plot()
```



The final return is close to zero as expected. Since it is a random strategy it is useful to obtain the results of larger sample:

```
In [121]: random_str.histogram()
```



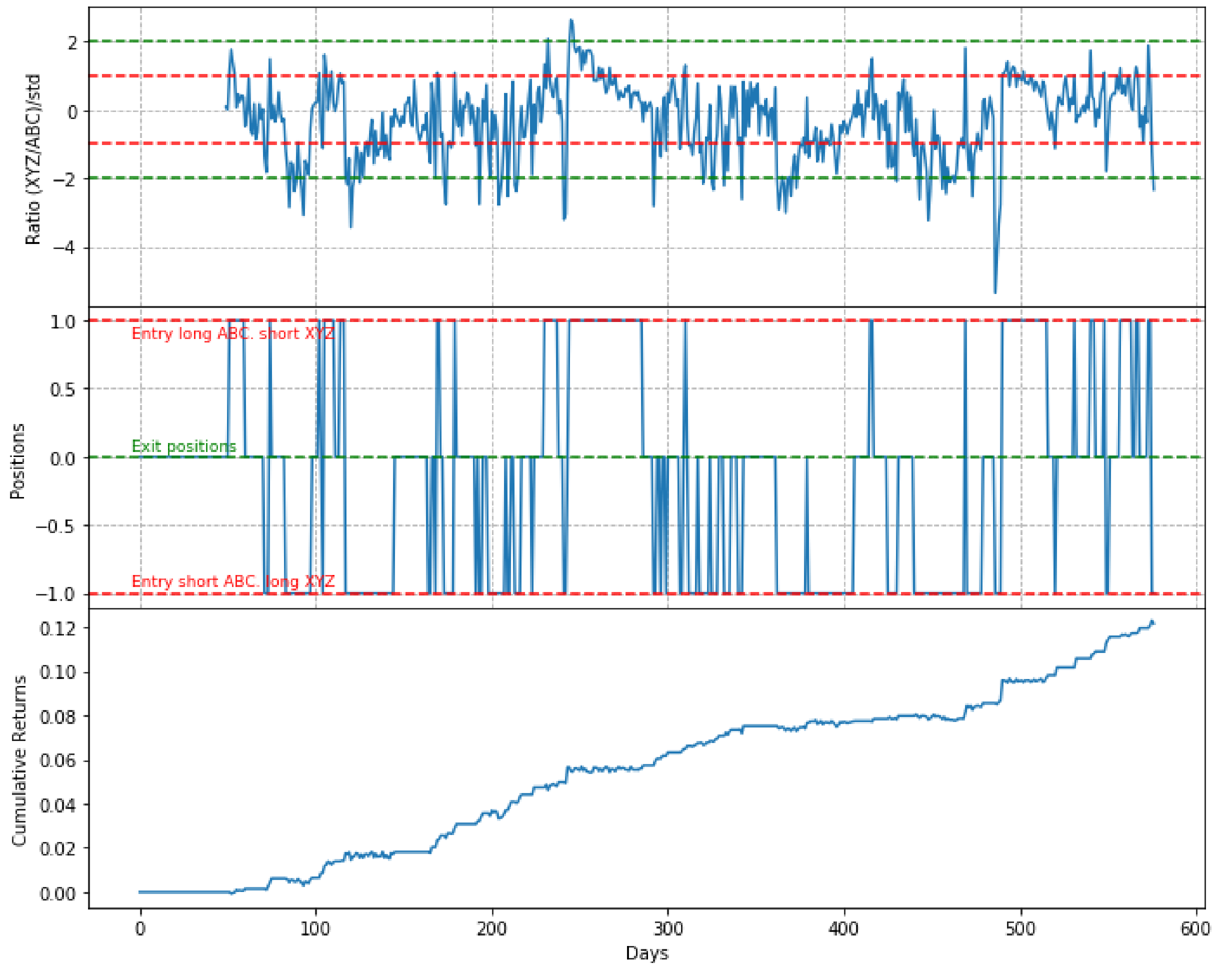
The result above indicates that unless a strategy is biased towards negative returns (not randomly wrong), using pair trading is not likely to generate a negative AR lower than ~ -1%. This conclusion assumes no change in the pricing correlation of the underlying assets.

2.2 Threshold entry and exit strategy:

This strategy defines entry and exit points based on how far the ratio XYZ/ABC is from its rolling mean. The use of the normalized ratio XYZ/ABC instead of the difference (XYZ-ABC) was used since the latter has a slope that would need to be subtracted to obtain stationarity. If the normalized ratio passes a given entry threshold it is defined as an entry point. If the normalized ratio goes below a given exit threshold, it is defined as an exit point. The entry and exit thresholds, along with the rolling window range are hyper-parameters. The positions are hedged only at the entry points (**not** continuously hedged).

```
In [11]: basic_str = basicStrategy()
         basic_str.apply(50, 1, 0) # 50-day rolling window, 1-std entry point,
                                   # back to the mean for exit point
```

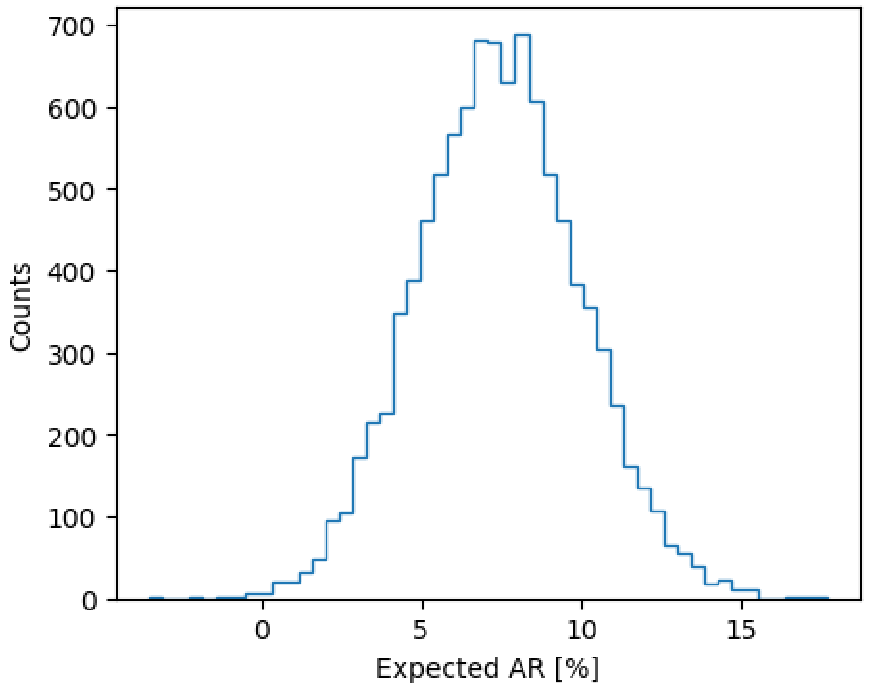
```
In [12]: basic_str.plot()
```



The expected AR and associated risk of this strategy are:

```
In [147]: basic_str.ARdistribution()
```

```
Expected AR is 7.42% +- 2.55% (1-sigma confidence)
```



The hyper-parameters of the strategy (window range for the rolling estimates, and entry and exits thresholds) were optimized using train and validation data since this strategy does not have parameters to be optimized:

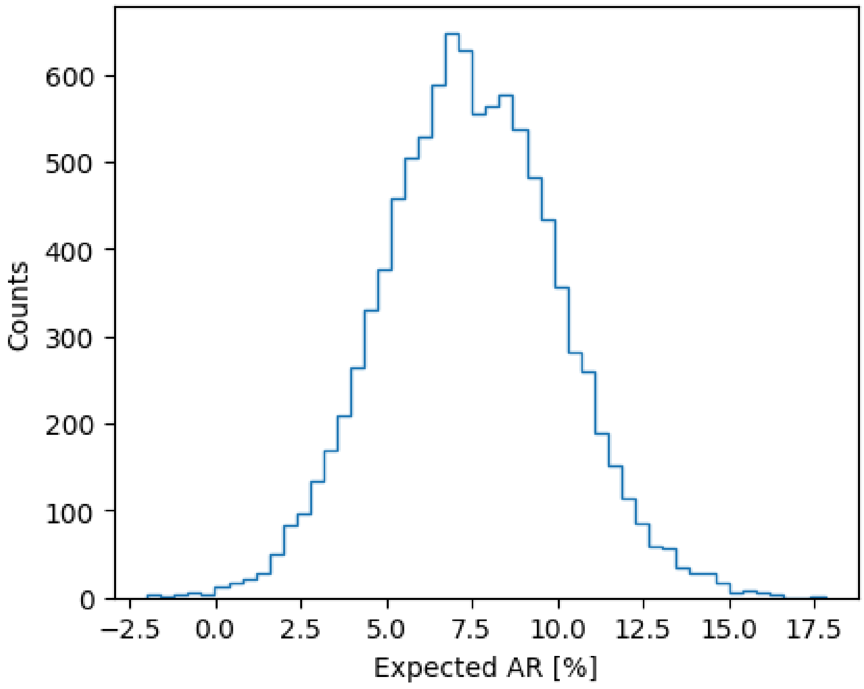
```
In [126]: window = [2, 5, 10, 20, 30]
          entry_threshold = [0.5, 0.75, 1, 1.5, 2]
          exit_threshold = [0.25, 0, -0.25]
          results = basic_str.optimization(window, entry_threshold, exit_threshold)
```

```
Best parameters: window range: 5 - entry: 0.5 - exit: 0 - AR: 12.37%
```

The best hyper-parameters are a 5-day rolling window, a 0.5 standard deviation entry threshold and an exit threshold of zero (when the normalized ratio goes back to the mean). The expected AR and risk of the strategy using the optimized hyper-parameters are:

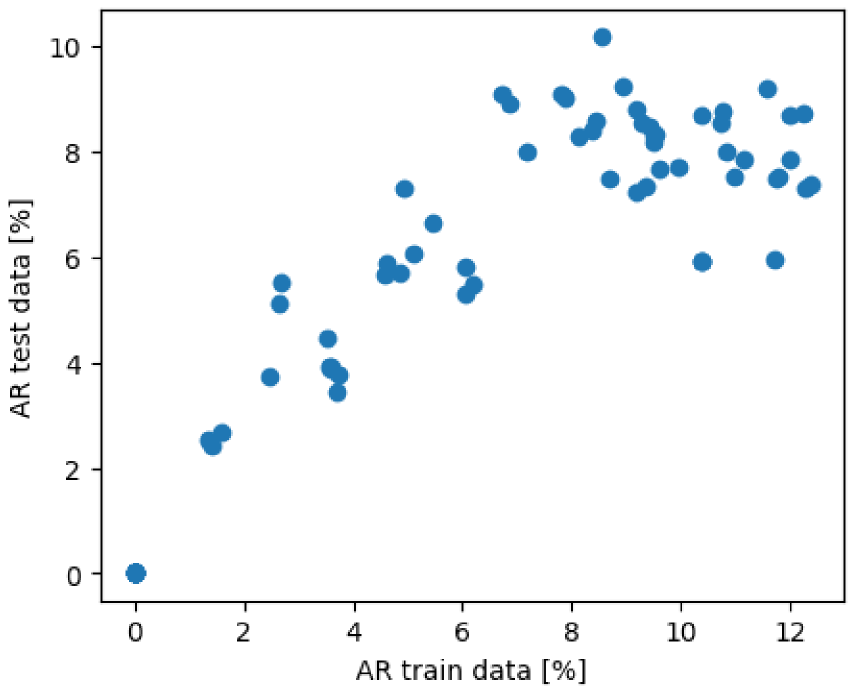
```
In [149]: basic_str.apply(5, 0.5, 0) # Best result based on the train data
          basic_str.ARdistribution()
```

```
Expected AR is 7.46% +- 2.58% (1-sigma confidence)
```



To better understand the predictive power of the strategy (since this strategy does not have parameters but hyper-parameters) it is instructive to plot the AR of the train and test data for all the hyper-parameters studied:

```
In [137]: plotAR(results)
```



There is a correlation between the train and test data which indicates the strategy results on the train data have predictive power over the strategy results on the test data.

2.3 Recurrent Neural Network (RNN) regression strategy:

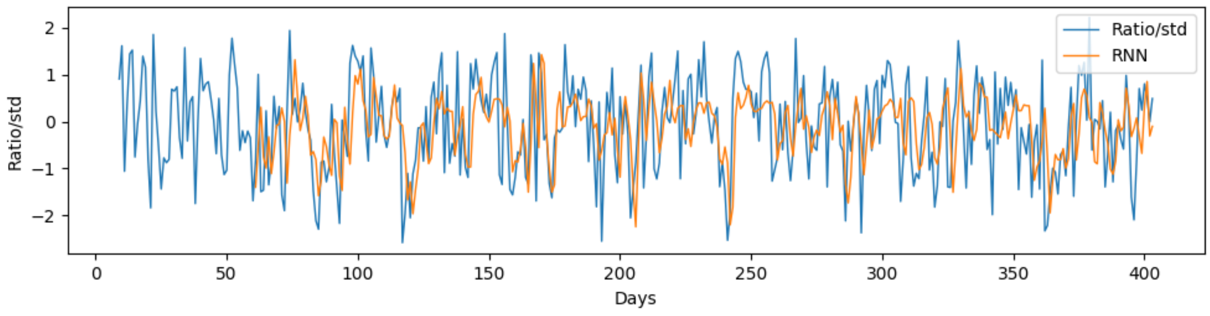
This strategy attempted to define entry and exit points based on a regression analysis on the movement of (XYZ/ABC)/std. A Recurrent Neural Network (RNN) was trained to predict how the normalized ratio XYZ/ABC evolves as a function of time. Since the dataset is not very large, the RNN consisted of only 1-layer. The number of neurons and the number of sequences were hyper-parameters to be optimized.

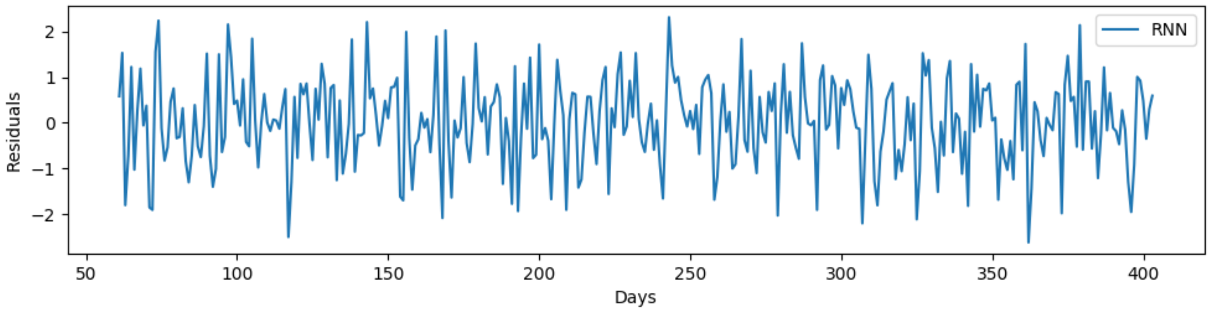
```
In [531]: train_data = seriesFull.loc[:len(seriesFull)*0.7]
          validation_data = seriesFull.loc[len(seriesFull)*0.7:]

          rnn = RNNRegression(n_neurons=10, learning_rate=0.001, fit_range=50,
                              rolling_window=10)
          rnn.fit(train_data)
```

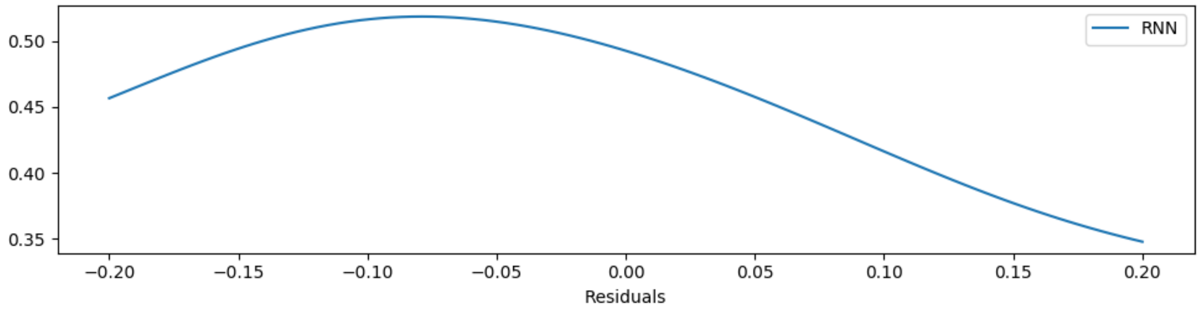
```
Iteration 0 - model RMSE:1.11244 - best RMSE:1.11244
Iteration 200 - model RMSE:0.95273 - best RMSE:0.95273
Iteration 400 - model RMSE:0.94645 - best RMSE:0.94645
Iteration 600 - model RMSE:0.92527 - best RMSE:0.92527
Iteration 800 - model RMSE:0.91341 - best RMSE:0.91341
Iteration 1000 - model RMSE:0.89867 - best RMSE:0.89867
Iteration 1200 - model RMSE:0.89090 - best RMSE:0.89090
Iteration 1400 - model RMSE:0.87839 - best RMSE:0.87839
Iteration 1600 - model RMSE:0.85764 - best RMSE:0.85764
Iteration 1800 - model RMSE:0.83656 - best RMSE:0.83656
Iteration 2000 - model RMSE:0.82123 - best RMSE:0.82123
Iteration 2200 - model RMSE:0.81498 - best RMSE:0.81498
Iteration 2400 - model RMSE:0.81322 - best RMSE:0.81322
Iteration 2600 - model RMSE:0.81216 - best RMSE:0.81216
Iteration 2800 - model RMSE:0.81330 - best RMSE:0.81216
Iteration 3000 - model RMSE:0.81279 - best RMSE:0.81216
Iteration 3200 - model RMSE:0.81317 - best RMSE:0.81216
Iteration 3400 - model RMSE:0.81390 - best RMSE:0.81216
Early stopping!
```

```
Out[531]: RNNRegression(activation=<function elu at 0x133f13ae8>, fit_range=50,
                 learning_rate=0.001, n_neurons=10,
                 optimizer_class=<class 'TensorFlow.python.training.adam.AdamOptimizer'>,
                 rolling_window=10)
```

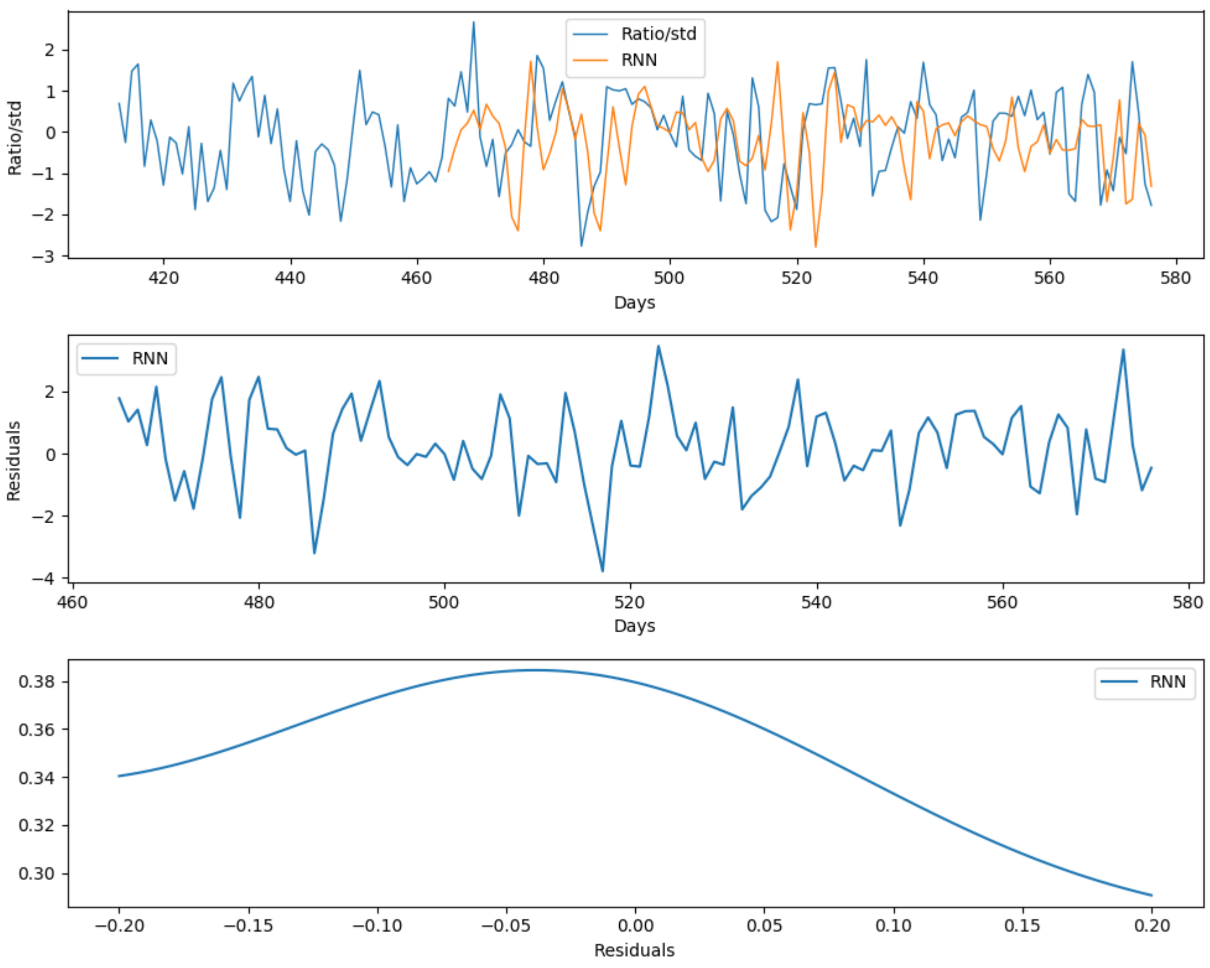
```
In [532]: train_data = rnn.rolling_estimate(train_data)
          plot_ratioFitting(results=train_data)
```







```
In [533]: test_data = rnn.rolling_estimate(test_data)
          plot_ratioFitting(results=test_data)
```



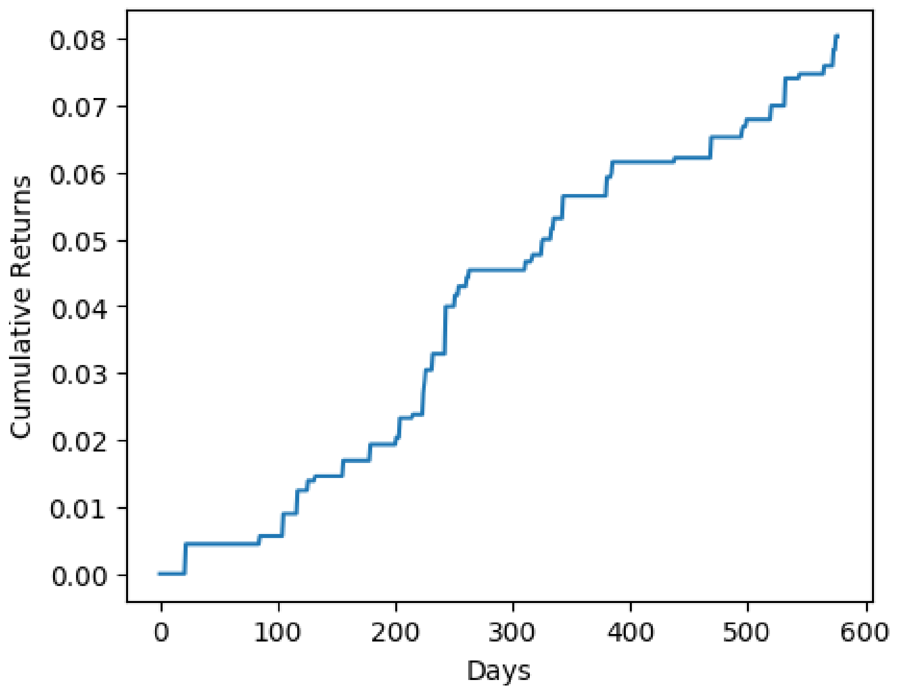
As shown in the graph above, the RNN regression algorithm does not predict the movement of the normalized ratio with an acceptable certainty. Even the overfitted train data is not an acceptable fit since it is "shifted" by one unit. Therefore, this is not a viable strategy and it was abandoned.

2.4 Label identification:

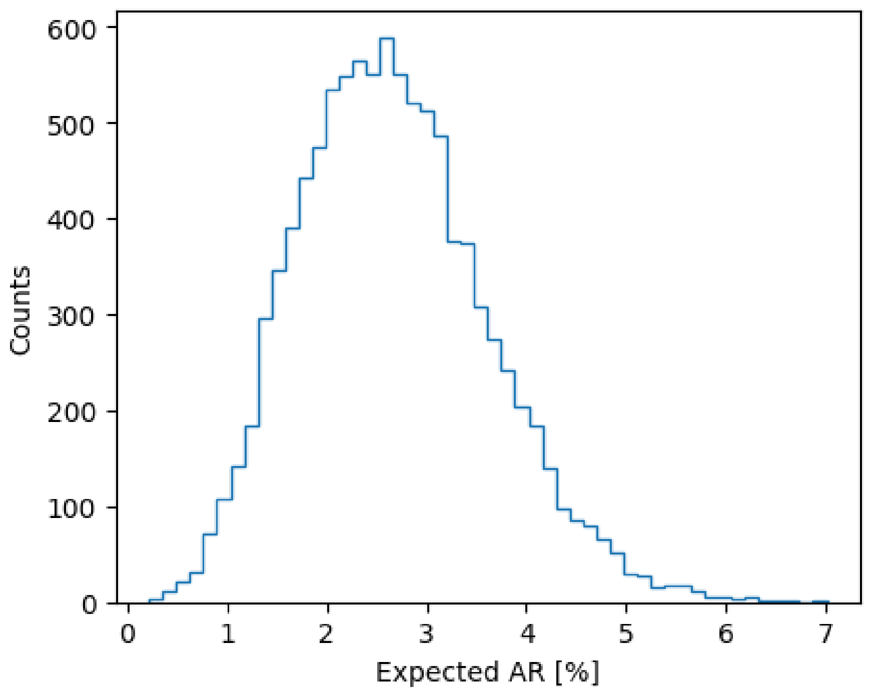
Previous strategies used the normalized ratio as a signal to enter or exit a given position. An alternative approach is to let the algorithm use all the available data to select the positions that will net the highest profit the following day. In order to train those algorithms, the ideal positions (or target labels) need to be identified first. In this section, strategies that use those identified best positions (looking at the future) were employed in order to gain insight into the best positions and to obtain the highest possible profit using pair trading with the data provided for this exercise.

Naively, one may think that only entry points for which ABC goes up and XYZ goes down or vice-versa yield the highest profits. Exit points are all other days that are not entry points. Using positions selected in this described manner, a strategy was developed that resulted in:

```
In [150]: bestPairs_str = bestPairsStrategy()
          bestPairs_str.apply()
          bestPairs_str.plot()
          bestPairs_str.ARdistribution()
```



```
Expected AR is 2.64% +- 0.94% (1-sigma confidence)
```



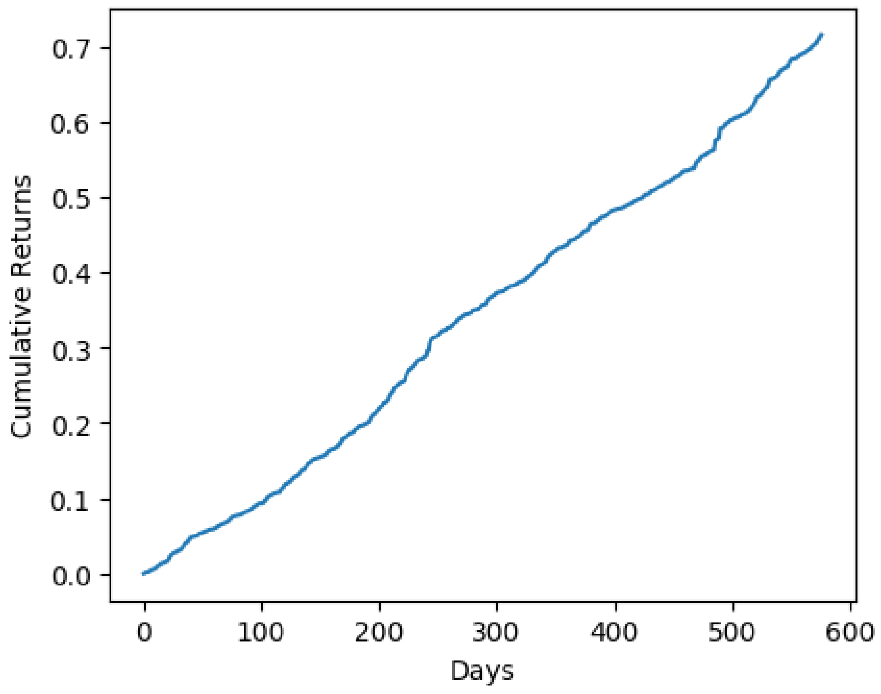
The expected AR of this strategy is actually worse than the optimized threshold entry and exit strategy. Although, this result may seem counter intuitive there are two reasons for this:

- There are some situations in which the hedging does not fully work (since the ratio XYZ/ABC has a downward tendency) and therefore being long on ABC and short on XYZ (instead of neutral) is actually better in the long term.
- More importantly, if the rolling mean of the ratio XYZ/ABC jumps up, it does not necessarily mean that XYZ will move down and ABC will move up. It could also be that only one of

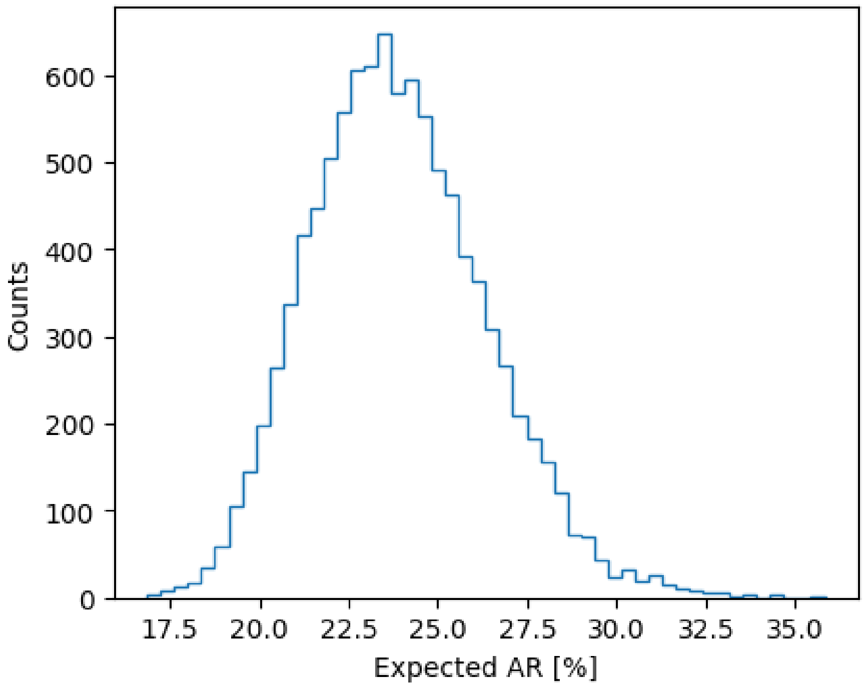
them moves in the “right” direction while the other one stays close to neutral. In that case keeping the previous position will still net a positive daily return. Since the threshold entry and exit strategy stays in the previous position until an exit signal is triggered, the threshold basic strategy benefits from this.

Based on the above observations, we can simply find the best possible position every day (again looking into the future) in a pair trading strategy regardless of how the individual components (ABC or XYZ) move. The position that nets a positive return the next day is selected for that day. This strategy will result in the best possible return with the given test data:

```
In [151]: bestPositions_str = bestPositionsStrategy()
          bestPositions_str.apply()
          bestPositions_str.plot()
          bestPositions_str.ARdistribution()
```



```
Expected AR is 23.90% +- 2.47% (1-sigma confidence)
```



The best possible AR on our test data using a pair strategy is 23.9%. The positions that result in this ceiling AR are defined as the ideal positions and will be used as the labels to train all remaining algorithms.

2.5 RNN classification strategy:

This strategy defines entry and exit points based on a RNN classification algorithm trained to predict the position that results in the best profits. The RNN algorithm consisted of a single layer. The input of the model consisted of the XYZ, ABC, DIFF and RATIO columns in the dataset. The sequence length (number of days considered) is a hyper-parameter. The algorithm was trained using train data only:

```
In [1124]: train_data = seriesFull.loc[:len(seriesFull)*0.7]
           train_data = bestPositions(train_data)
           rnnCl = RNNClassification(n_neurons=20, learning_rate=0.001, fit_range=20)
           rnnCl.fit(train_data, max_iterations=10000)
```

```
Iter 0 - model acc_train:0.5224 - model acc_validation:0.4522 - best acc_validation:0.4522
Iter 0 - model acc_train:0.5224 - model acc_validation:0.4522 - best acc_validation:0.4522
Iter 22 - model acc_train:0.5224 - model acc_validation:0.4609 - best acc_validation:0.4609
Iter 24 - model acc_train:0.4776 - model acc_validation:0.4870 - best acc_validation:0.4870
Iter 25 - model acc_train:0.4776 - model acc_validation:0.5130 - best acc_validation:0.5130
Iter 26 - model acc_train:0.4776 - model acc_validation:0.5478 - best acc_validation:0.5478
Iter 500 - model acc_train:0.5597 - model acc_validation:0.5217 - best acc_validation:0.5478
Iter 727 - model acc_train:0.5709 - model acc_validation:0.5565 - best acc_validation:0.5565
Iter 730 - model acc_train:0.5224 - model acc_validation:0.5652 - best acc_validation:0.5652
Iter 739 - model acc_train:0.5634 - model acc_validation:0.6000 - best acc_validation:0.6000
Iter 740 - model acc_train:0.5448 - model acc_validation:0.6261 - best acc_validation:0.6261
Iter 1000 - model acc_train:0.5896 - model acc_validation:0.5130 - best acc_validation:0.6261
Iter 1014 - model acc_train:0.5821 - model acc_validation:0.6348 - best acc_validation:0.6348
Iter 1189 - model acc_train:0.6045 - model acc_validation:0.6435 - best acc_validation:0.6435
Iter 1500 - model acc_train:0.5373 - model acc_validation:0.5304 - best acc_validation:0.6435
Iter 2000 - model acc_train:0.6157 - model acc_validation:0.5826 - best acc_validation:0.6435
Iter 2500 - model acc_train:0.6493 - model acc_validation:0.5913 - best acc_validation:0.6435
Iter 2736 - model acc_train:0.6493 - model acc_validation:0.6609 - best acc_validation:0.6609
Iter 3000 - model acc_train:0.6716 - model acc_validation:0.5652 - best acc_validation:0.6609
Iter 3500 - model acc_train:0.7276 - model acc_validation:0.5652 - best acc_validation:0.6609
Iter 4000 - model acc_train:0.7463 - model acc_validation:0.6000 - best acc_validation:0.6609
Iter 4500 - model acc_train:0.7575 - model acc_validation:0.6087 - best acc_validation:0.6609
Iter 5000 - model acc_train:0.7425 - model acc_validation:0.6435 - best acc_validation:0.6609
Iter 5500 - model acc_train:0.7351 - model acc_validation:0.6261 - best acc_validation:0.6609
Iter 5800 - model acc_train:0.7239 - model acc_validation:0.6696 - best acc_validation:0.6696
Iter 6000 - model acc_train:0.7164 - model acc_validation:0.6348 - best acc_validation:0.6696
Iter 6500 - model acc_train:0.7313 - model acc_validation:0.6087 - best acc_validation:0.6696
Iter 7000 - model acc_train:0.7276 - model acc_validation:0.6174 - best acc_validation:0.6696
Iter 7500 - model acc_train:0.7164 - model acc_validation:0.6174 - best acc_validation:0.6696
Iter 8000 - model acc_train:0.7239 - model acc_validation:0.5913 - best acc_validation:0.6696
Iter 8500 - model acc_train:0.7276 - model acc_validation:0.6435 - best acc_validation:0.6696
Iter 9000 - model acc_train:0.7201 - model acc_validation:0.6087 - best acc_validation:0.6696
Iter 9500 - model acc_train:0.7276 - model acc_validation:0.6174 - best acc_validation:0.6696
```

```
Out[1124]: RNNClassification(fit_range=20, learning_rate=0.001, n_neurons=20,
                     optimizer_class=<class 'TensorFlow.python.training.adam.AdamOptimizer'>)
```

```
In [1125]: #rnnCl.save("../models/RNNClassification-best")
```

For hyper-parameter optimization we took advantage of the compatibility in our created *RN-Nclassification* class and the native Scikit-Learn RandomizedSearchCV class:

```
In [1077]: from sklearn.model_selection import RandomizedSearchCV
           param_distribs = {
                "n_neurons": [10, 20, 30, 50],
                "fit_range": [5, 10, 20],
                "learning_rate": [0.001, 0.01]
           }
           rnd_search = RandomizedSearchCV(RNNClassification(), param_distribs, n_iter=24,
                                           random_state=42, verbose=2, cv=3, n_jobs=-1,
                                           iid=False)
           rnd_search.fit(train_data, max_iterations=10000)
```

```
Fitting 3 folds for each of 24 candidates, totalling 72 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:  6.9min
[Parallel(n_jobs=-1)]: Done  72 out of  72 | elapsed: 24.0min finished
```

```
Iter 0 - model acc_train:0.5018 - model acc_validation:0.4661 - best acc_validation:0.4661
Iter 0 - model acc_train:0.5018 - model acc_validation:0.4661 - best acc_validation:0.4661
Iter 9 - model acc_train:0.5018 - model acc_validation:0.4831 - best acc_validation:0.4831
Iter 10 - model acc_train:0.5091 - model acc_validation:0.5000 - best acc_validation:0.5000
Iter 11 - model acc_train:0.5055 - model acc_validation:0.5339 - best acc_validation:0.5339
Iter 33 - model acc_train:0.5091 - model acc_validation:0.5508 - best acc_validation:0.5508
Iter 61 - model acc_train:0.5091 - model acc_validation:0.5593 - best acc_validation:0.5593
Iter 118 - model acc_train:0.5673 - model acc_validation:0.5678 - best acc_validation:0.5678
Iter 500 - model acc_train:0.5927 - model acc_validation:0.4831 - best acc_validation:0.5678
Iter 1000 - model acc_train:0.6218 - model acc_validation:0.4831 - best acc_validation:0.5678
Iter 1340 - model acc_train:0.6145 - model acc_validation:0.6695 - best acc_validation:0.6695
Iter 1500 - model acc_train:0.6327 - model acc_validation:0.5085 - best acc_validation:0.6695
Iter 2000 - model acc_train:0.6218 - model acc_validation:0.5085 - best acc_validation:0.6695
Iter 2500 - model acc_train:0.6545 - model acc_validation:0.5424 - best acc_validation:0.6695
Iter 3000 - model acc_train:0.6691 - model acc_validation:0.5847 - best acc_validation:0.6695
Iter 3500 - model acc_train:0.7127 - model acc_validation:0.5424 - best acc_validation:0.6695
Iter 4000 - model acc_train:0.7200 - model acc_validation:0.4746 - best acc_validation:0.6695
Iter 4500 - model acc_train:0.7455 - model acc_validation:0.4915 - best acc_validation:0.6695
Iter 5000 - model acc_train:0.7636 - model acc_validation:0.4915 - best acc_validation:0.6695
Iter 5500 - model acc_train:0.7636 - model acc_validation:0.4915 - best acc_validation:0.6695
Iter 6000 - model acc_train:0.7745 - model acc_validation:0.5000 - best acc_validation:0.6695
Iter 6500 - model acc_train:0.6982 - model acc_validation:0.4915 - best acc_validation:0.6695
Iter 7000 - model acc_train:0.7564 - model acc_validation:0.5169 - best acc_validation:0.6695
Iter 7298 - model acc_train:0.5600 - model acc_validation:0.6864 - best acc_validation:0.6864
Iter 7304 - model acc_train:0.5709 - model acc_validation:0.7034 - best acc_validation:0.7034
Iter 7500 - model acc_train:0.6145 - model acc_validation:0.6271 - best acc_validation:0.7034
Iter 8000 - model acc_train:0.6582 - model acc_validation:0.5678 - best acc_validation:0.7034
Iter 8500 - model acc_train:0.6800 - model acc_validation:0.6186 - best acc_validation:0.7034
Iter 9000 - model acc_train:0.6909 - model acc_validation:0.6271 - best acc_validation:0.7034
Iter 9500 - model acc_train:0.7273 - model acc_validation:0.6525 - best acc_validation:0.7034
Iter 9523 - model acc_train:0.5818 - model acc_validation:0.7119 - best acc_validation:0.7119
```

```
Out[1077]: RandomizedSearchCV(cv=3, error_score='raise-deprecating',
                     estimator=RNNClassification(fit_range=50, learning_rate=0.01, n_neurons=10,
                    optimizer_class=<class 'TensorFlow.python.training.adam.AdamOptimizer'>),
                     fit_params=None, iid=False, n_iter=24, n_jobs=-1,
                     param_distributions={'n_neurons': [10, 20, 30, 50], 'fit_range': [5, 10, 20],
                     pre_dispatch='2*n_jobs', random_state=42, refit=True,
                     return_train_score='warn', scoring=None, verbose=2)
```

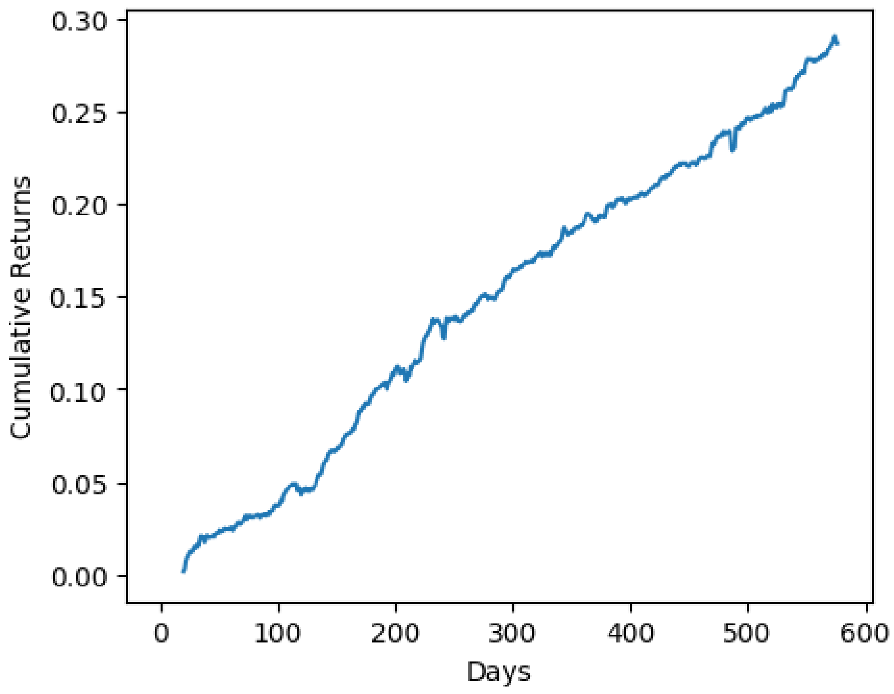
```
In [1078]: rnd_search.best_params_
```

```
Out[1078]: {'n_neurons': 50, 'learning_rate': 0.001, 'fit_range': 10}
```

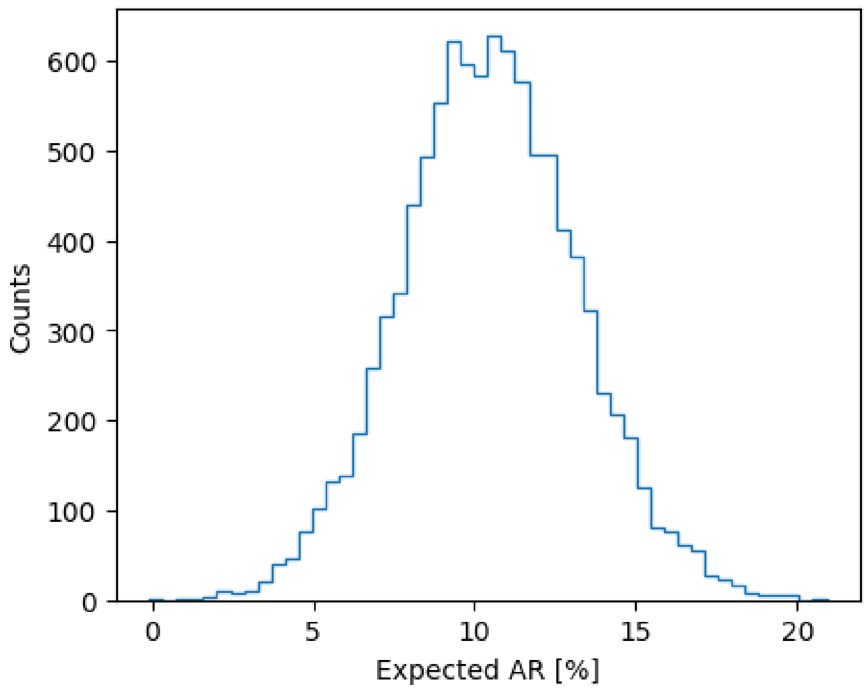
An strategy following the positions calculated by the trained RNN algorithm results in:

```
In [152]: RNNCl_str = RNNClStrategy()
          RNNCl_str.apply(threshold = 0.5)
          RNNCl_str.plot()
          RNNCl_str.ARdistribution()
```

```
INFO:TensorFlow:Restoring parameters from ../models/RNNClassification-best
```



```
Expected AR is 10.43% +- 2.75% (1-sigma confidence)
```



This method can certainly benefit from more data. The number of neurons in the single layer is low because the algorithm overfits the training data otherwise. The best result of the grid search was not used because of the problem of overfitting. It is worth to point out that it is better to have a position than to be neutral in a given day. Since the RNN classifier uses a softmax classifier, the assignment probability to a given position can be used to define either an entry point or to remain neutral (by using a threshold). Raising the threshold above a simple majority decreases the AR.

2.6 Position classification Long short-term memory (LSTM) strategy:

This strategy defines entry and exit points based on a LSTM classifier algorithm trained to predict the position that results in the best profits. The LSTM is trained using a single layer. The input data consists of XYZ, ABC, DIFF and RATIO. The sequence length is a hyper-parameter. The algorithm was trained using train data only:

```
In [1135]: train_data = seriesFull.loc[:len(seriesFull)*0.7]
           train_data = bestPositions(train_data)
           lstmCl = LSTMClassification(n_neurons=5, learning_rate=0.001, fit_range=20)
           lstmCl.fit(train_data, max_iterations=5000)
```

```
Iter 0 - model acc_train:0.4776 - model acc_validation:0.5478 - best acc_validation:0.5478
Iter 0 - model acc_train:0.4776 - model acc_validation:0.5478 - best acc_validation:0.5478
Iter 500 - model acc_train:0.4776 - model acc_validation:0.5478 - best acc_validation:0.5478
Iter 1000 - model acc_train:0.5224 - model acc_validation:0.4522 - best acc_validation:0.5478
Iter 1500 - model acc_train:0.5224 - model acc_validation:0.4522 - best acc_validation:0.5478
Iter 1926 - model acc_train:0.5522 - model acc_validation:0.5565 - best acc_validation:0.5565
Iter 1930 - model acc_train:0.5522 - model acc_validation:0.5739 - best acc_validation:0.5739
Iter 2000 - model acc_train:0.5560 - model acc_validation:0.5391 - best acc_validation:0.5739
Iter 2167 - model acc_train:0.5784 - model acc_validation:0.5913 - best acc_validation:0.5913
Iter 2242 - model acc_train:0.5896 - model acc_validation:0.6087 - best acc_validation:0.6087
Iter 2262 - model acc_train:0.5933 - model acc_validation:0.6174 - best acc_validation:0.6174
Iter 2265 - model acc_train:0.5933 - model acc_validation:0.6261 - best acc_validation:0.6261
Iter 2268 - model acc_train:0.5933 - model acc_validation:0.6348 - best acc_validation:0.6348
Iter 2408 - model acc_train:0.6082 - model acc_validation:0.6522 - best acc_validation:0.6522
Iter 2499 - model acc_train:0.6604 - model acc_validation:0.6696 - best acc_validation:0.6696
Iter 2500 - model acc_train:0.6418 - model acc_validation:0.6261 - best acc_validation:0.6696
Iter 2512 - model acc_train:0.6530 - model acc_validation:0.6783 - best acc_validation:0.6783
Iter 2622 - model acc_train:0.6604 - model acc_validation:0.6870 - best acc_validation:0.6870
Iter 2657 - model acc_train:0.6791 - model acc_validation:0.6957 - best acc_validation:0.6957
Iter 2802 - model acc_train:0.6903 - model acc_validation:0.7043 - best acc_validation:0.7043
Iter 3000 - model acc_train:0.6828 - model acc_validation:0.6957 - best acc_validation:0.7043
Iter 3394 - model acc_train:0.6866 - model acc_validation:0.7130 - best acc_validation:0.7130
Iter 3396 - model acc_train:0.6903 - model acc_validation:0.7217 - best acc_validation:0.7217
Iter 3500 - model acc_train:0.6903 - model acc_validation:0.6783 - best acc_validation:0.7217
Iter 4000 - model acc_train:0.7090 - model acc_validation:0.6870 - best acc_validation:0.7217
Iter 4500 - model acc_train:0.7090 - model acc_validation:0.6696 - best acc_validation:0.7217
```

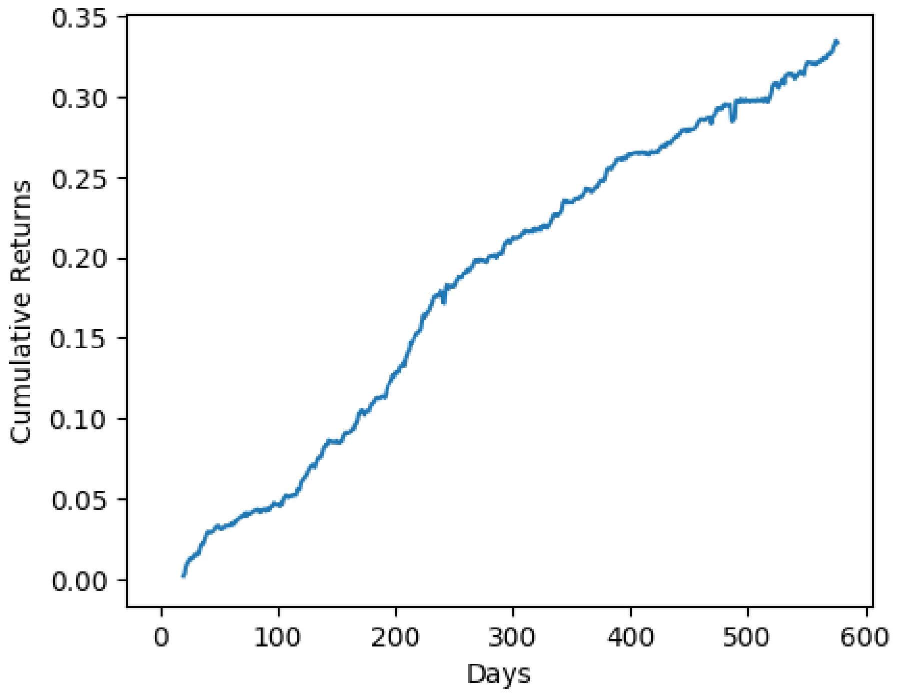
```
Out[1135]: LSTMClassification(fit_range=20, learning_rate=0.001, n_layers=None,
                     n_neurons=5,
                    optimizer_class=<class 'TensorFlow.python.training.adam.AdamOptimizer'>)
```

```
In [1136]: #lstmCl.save("../models/LSTMClassification-best")
```

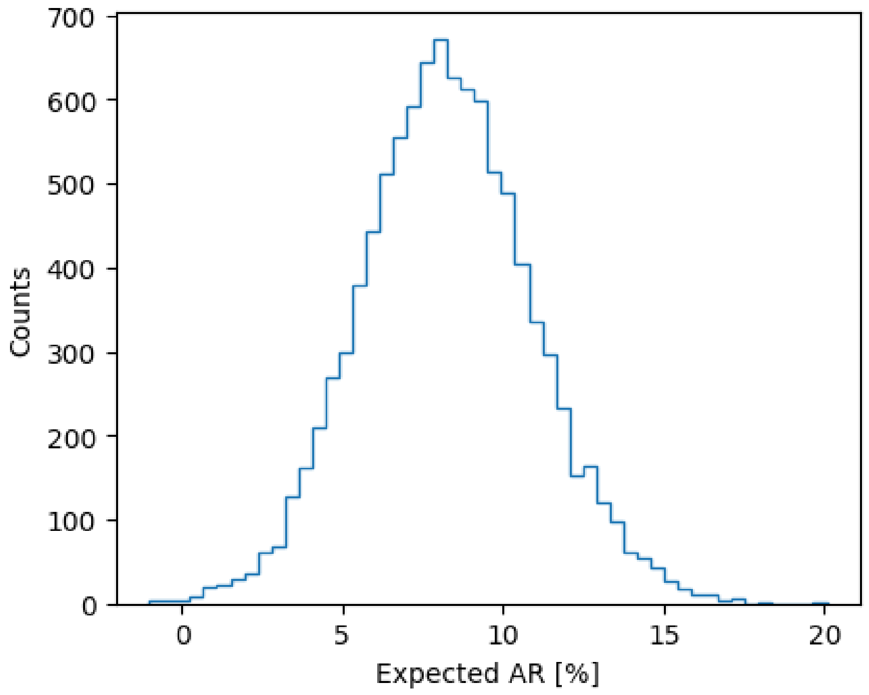
The strategy that uses the positions predicted by the trained LSTM algorithm results in

```
In [198]: LSTMCl_str = LSTMClStrategy()
          LSTMCl_str.apply(threshold = 0.5)
          LSTMCl_str.plot()
          LSTMCl_str.ARdistribution()
```

```
INFO:TensorFlow:Restoring parameters from ../models/LSTMClassification-best
```



```
Expected AR is 8.21% +- 2.74% (1-sigma confidence)
```



This method will certainly benefit from more data as well. The number of neurons in the single layer is already low (5 neurons) because the algorithm overfits the training data otherwise. As with the RNN strategy, it is better to have a position than be neutral. Raising the threshold above simple majority in the softmax probabilities decreases the test data AR.

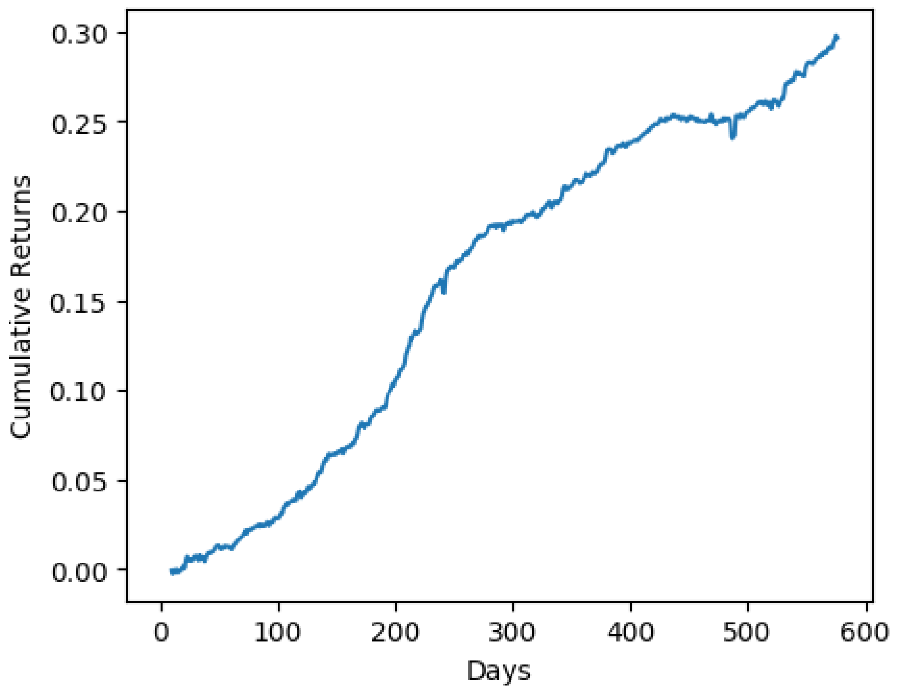
2.7 Position classification by a combination of logistic regression, random forest and K-neighbors strategy:

This strategy defines entry and exit points based on a hard voting classifier ensemble of random forest, logistic regression and k-neighbor classifier algorithms trained to predict the position that results in the best profits. The input data for a prediction is XYZ, ABC, DIFF and RATIO over several days. The number of days is a hyper-parameter that has been optimized. Hyper-parameters for all classifiers were also optimized.

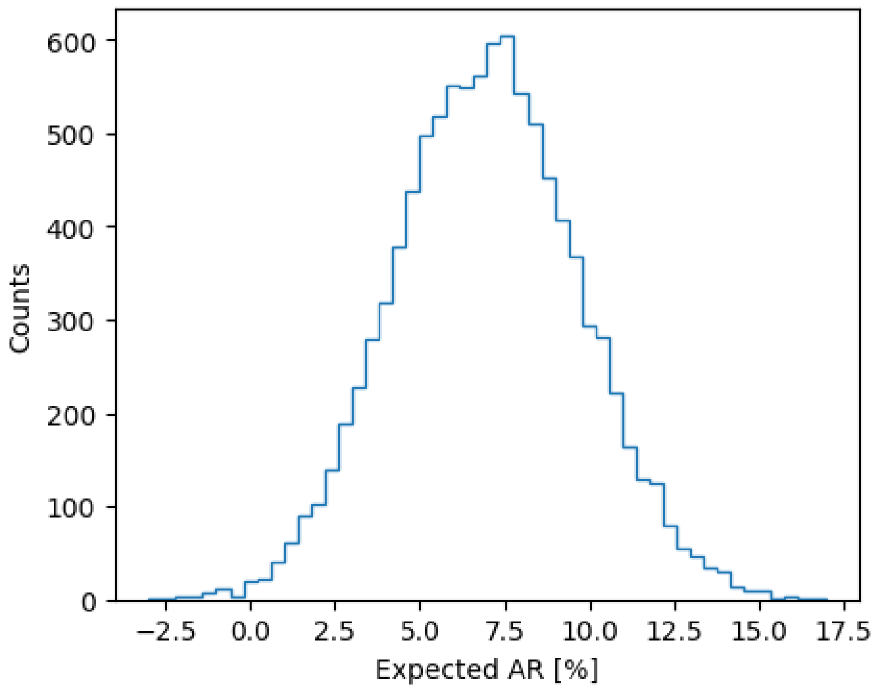
```
In [159]: train_data = seriesFull.loc[:len(seriesFull)*0.7]
          train_data = bestPositions(train_data)
          fit_data = 10 # Number of days
          classif = trainClassification(train_data, fit_data)
```

```
Model LogisticRegression - acc. train set: 0.6983 - acc. validation set: 0.6465
Model RandomForestClassifier - acc. train set: 0.7864 - acc. validation set: 0.6061
Model KNeighborsClassifier - acc. train set: 0.6271 - acc. validation set: 0.4848
Model VotingClassifier - acc. train set: 0.7729 - acc. validation set: 0.6364
```

```
In [160]: Classif_str = classifStrategy()
          Classif_str.apply(classif)
          Classif_str.plot()
          Classif_str.ARdistribution()
```



```
Expected AR is 6.95% +- 2.69% (1-sigma confidence)
```



This method will certainly benefit from more data as well. Some of the hyper-parameters in the random forest method are pretty low (max_depth = 3, min_samples_leaf = 15) already. The K-neighbor hyper-parameters were optimized but did not make much difference in terms of accuracy improvement. Algorithms SVC and Gaussian Processes classifier were also explored but their results were significantly worse than the algorithms selected, and were not included in the ensemble.

2.8 Ensemble strategy:

A strategy that combines the strategies discussed earlier into a hard voting ensemble was also implemented. Different combinations of the optimized threshold entry and exit, random forest, logistic regression, RNN and LSTM strategies into the ensemble were studied. If there was a tie in the predicted position, a position long on ABC and short on XYZ was chosen since it is better than to be neutral.

```
In [161]: all_data = bestPositions(seriesFull)
          train_data = all_data.loc[:len(all_data)*0.7]
          test_data = all_data.loc[len(all_data)*0.7:]
```

All the methods in the ensemble have already been trained. In case they need to be trained again the following command can be used:

```
In [162]: trainModelsEnsemble(train_data, fit_RNN = False, fit_LSTM = False)
```

The RNN and LSTM algorithms take longer to train and therefore they are optional (the saving option is also disabled for now). The accuracy between the ideal pair strategy positions and predicted positions for the different algorithms in the unseen data is the following:

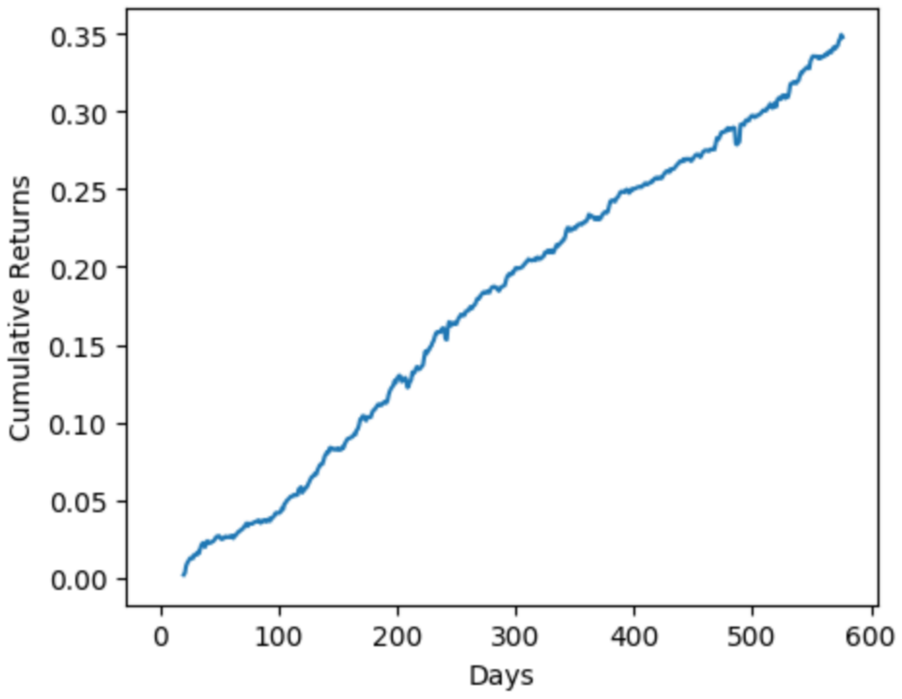
```
In [163]: checkAccuracy(test_data, ensemblePositions(test_data) )
```

```
INFO:TensorFlow:Restoring parameters from ../models/RNNClassification-best
INFO:TensorFlow:Restoring parameters from ../models/LSTMClassification-best
LogisticRegression accuracy: 0.5882352941176471
RandomForest accuracy: 0.6143790849673203
RNN accuracy: 0.6405228758169934
LSTM accuracy: 0.6470588235294118
BasicStrategy accuracy: 0.5098039215686274
Ensemble accuracy: 0.5816993464052288
IdealPosition accuracy: 1.0
```

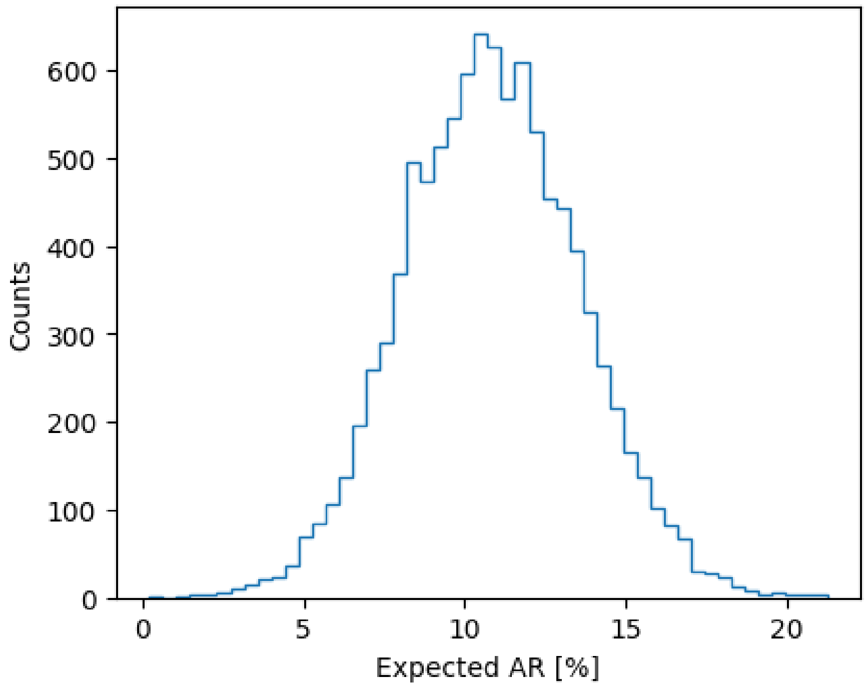
Based on the positions prediction accuracy, the best algorithms are RNN and LSTM. The following commands define and apply the strategy. The apply command has the option of selecting the algorithm of choice between *BasicStrategy* (threshold entry and exit), *RandomForest*, *LogisticRegression*, *RNN*, *LSTM* and *Ensemble*. An ensemble of Random Forest, RNN and LSTM results in:

```
In [181]: ensemble_str = ensembleStrategy(seriesFull)
          ensemble_str.apply(model = 'Ensemble', includeRF = True)
          ensemble_str.plot()
          ensemble_str.ARdistribution()
```

```
INFO:TensorFlow:Restoring parameters from ../models/RNNClassification-best
INFO:TensorFlow:Restoring parameters from ../models/LSTMClassification-best
```



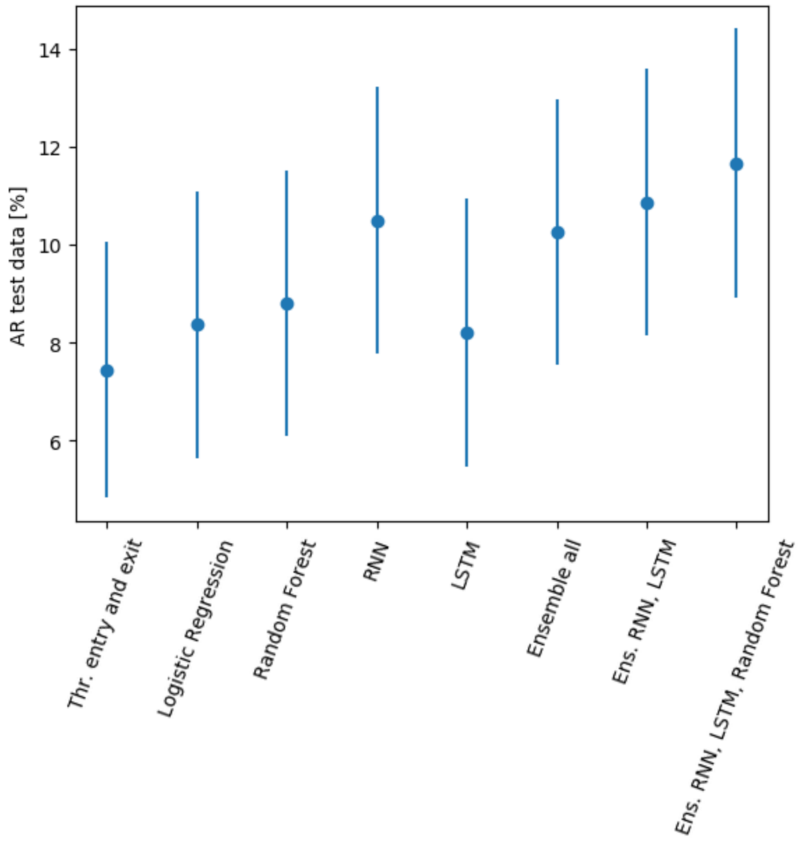
```
Expected AR is 11.70% +- 2.73% (1-sigma confidence)
```



2.9 Summary:

The expected AR and risk (defined as the AR 1-sigma confidence) for the strategies that yielded the best results are shown below:

```
In [204]: summary(ensemble_str)
```



A hard voting ensemble using RNN, LSTM and Random Forest predictions is slightly better than the other strategies (within their 1-std uncertainties) and it is recommended. The uncertainty in the predicted AR (or risk) is due to the limited number of days in the test data set and it only represents the *statistical* uncertainty. This risk does not include the *systematic* uncertainty due to changes in the behavior of the ABC and XYZ prices. The unaccounted risk may be significant if the algorithms are not trained frequently enough to capture such a change. It is recommended that if the correlated behavior of ABC and XYZ is expected to change over a time period of N number of days, the algorithms be trained in a shorter time scale. Based on the changes in the ABC and XYZ prices during the period of time considered in this analysis, the training of the algorithms could be done every few weeks.

Comment about the code: The following commands return a new prediction using the recommended trained model:

```
seriesFull = pd.read_csv('../data/raw/pairs.csv', header=0)
ensemble_str = ensembleStrategy(seriesFull)
ensemble_str.apply(model = 'Ensemble', includeRF = True)
ensemble_str.print()
```

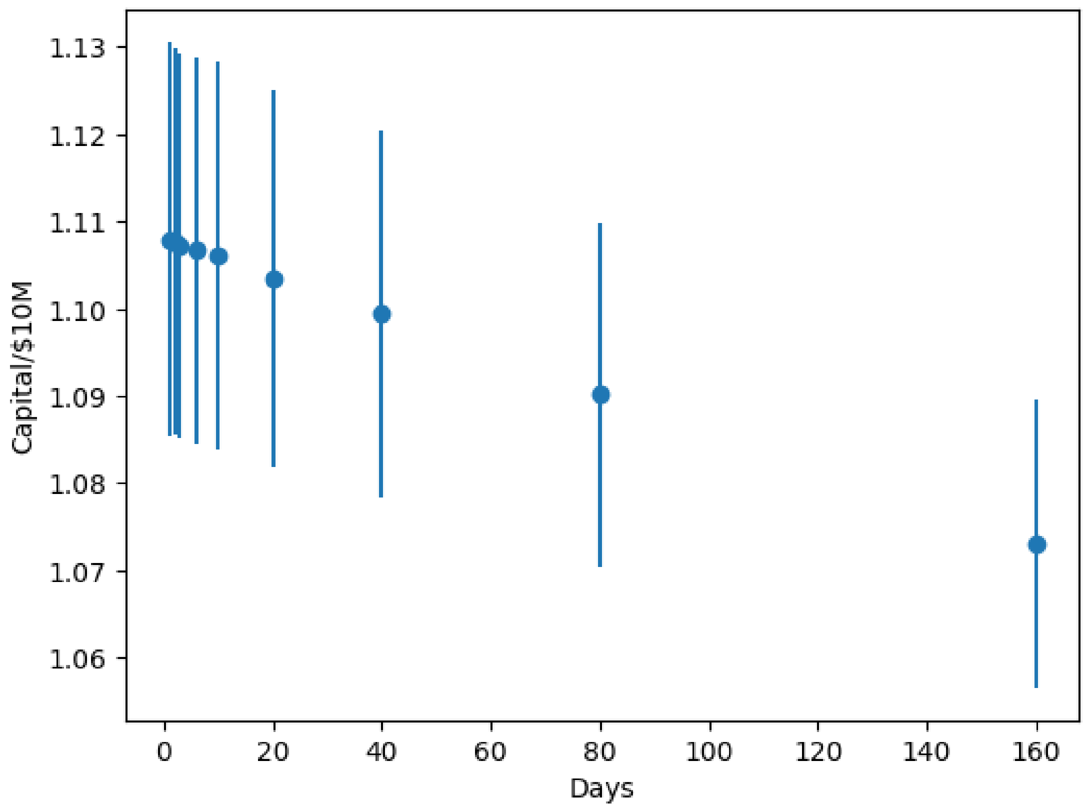
The last row of the printed table will have the recommended position (1 for long ABC, short XYZ and -1 for short ABC, long XYZ). The command to train the models is

```
trainModelsEnsemble(train_data, fit_RNN = True, fit_LSTM = True)
            # train_data has the same format as pairs.csv
```

3 Capital allocation

As discussed earlier, the best investment strategy uses the fact that it is better to pick a position than to be neutral in the long term. As such, one expects that the best capital allocation strategy will also require to invest all initial capital at the earliest opportunity. This capital allocation strategy was verified by a Monte Carlo simulation were the $10 million were allocated daily in equal amounts until all the money had been invested. The value of the investment portfolio 1 year after initial allocation as a function of the number of days to allocate the $10 millions is shown bellow:

```
In [322]: constantAllocationPlot(seriesFull)
```



4 Model scaling

The code returning the recommended position is not ready for production. In order to prepare it for production, version control and parallelization either by Multithreading or Multiprocessing (currently only implemented by using the Scikit-Learn RandomizedSearchCV parallelization option or by native sklearn methods) should be pursued. The code could also be optimized by creating vectorized functions, by using the python multiprocessing library (for example when searching the best models hyper-parameters), and by distributing TensorFlow computations among several CPUs or GPUs using the distributed TensorFlow framework. If the same model needs to be applied in sequence (for example if the same strategy is to be used several times per second) TensorFlow Queues may be an option.

Assuming access to a large cloud provider like Google Cloud Platform, AWS, Azure Machine Learning Studio or Kubeflow, it may possible to use distributed analytic frameworks such as Hadoop MapReduce. In this particular pair strategy scenario, a combination of TensorFlow Serving and Kubernetes can be used to deploy a web server to communicate with an API. In the case of several thousands of strategies being deployed simultaneously it would be possible to parallelize a pipeline of importing data from an SQL database, running the trained model and exporting the results back to the database. Offline learning is recommended. How often the models need to be trained will require testing. I would recommend to initially train the models once per week or if possible once per day and adjust as necessary. Model version control can be done by storing the trained model in a cloud server (like Amazon S3) and all the codes in Git. An alternative may be to use a DVC system.