FXCM Java API

Testing Our strategy

By Itamar Eden

In the following tutorial I'm going to demonstrate and explain how to backtest a strategy in an efficient way while putting an emphasis on our main objective: generating maximum profits with minimum risk. We will go over each part of the backtesting process in details, step by step, in a straightforward way.

The prerequisites for this tutorial include a basic to moderate understanding of the Java language, including object oriented approach for developing programs, and also a basic understanding of the financial markets and trading concepts.

This tutorial is the last part of the FXCM Java API tutorials series and as so, it relies heavily on the first two tutorials: "FXCM Java API – Pulling Historical Data" and "FXCM Java API – Building a Strategy". In order for you to make the most of this tutorial, please go over the first two thoroughly.

At the end of this tutorial you can find the complete listing of all source code files. So stay tuned and see how you can take advantage of the FXCM Java API to build and test your own strategies!

Before we start, please take a moment to read the following ***disclaimer***:

Trading foreign exchange on margin carries a high level of risk, and may not be suitable for all investors. Past performance is not indicative of future results. The high degree of leverage can work against you as well as for you. Before deciding to invest in foreign exchange you should carefully consider your investment objectives, level of experience, and risk appetite. The possibility exists that you could sustain a loss of some or all of your initial investment and therefore you should not invest money that you cannot afford to lose. You should be aware of all the risks associated with foreign exchange trading, and seek advice from an independent financial advisor if you have any doubts.

Testing Our Strategy

In this section we are to going to wrap together all of the things we've seen in the previous tutorials in order for us to see the bottom line: whether our strategy can (hopefully) turn a profit. we will build a *main* method that would be responsible for pulling historical data from the API, run our strategy and finally display the results for us to analyze.

First things first: Before we build our main method, we should create a StrategyResult class that will summarize the results of each backtesting of a strategy.

Each strategy that we will run, will return an instance of `StrategyResult` class with the result of each backtesting (for more details please see the `runStrategy()` method in the `RsiSignals` class in "FXCM Java API – Building a Strategy"). Though strategies vary from one another, each backtesting of a strategy should provide the same form of statistics. Obviously, one of the most important characteristic of a strategy is whether it can turn a profit. Another very important characteristic is the maximum drawdown. The maximum drawdown is defined as the maximum loss from a peak to a trough of a portfolio, before a new peak is attained, and in essence it measures the downside risk of our strategy. A high maximum drawdown means that even if our strategy was able to turn a nice profit for us eventually, during the time it was executed there was a period of a big loss and that's definitely a factor we must take into account.

Let's examine the importance of the maximum drawdown with the following example: say we have two strategies, the first one generates 10% profit in a year with 2% maximum drawdown while the second one generates 20% profit with 20% maximum drawdown. Which one would you prefer?

Though at first glance it seems that the second strategy is better since it generates more profit, we should consider the fact that it entails much more risk. During the entire year it was executed, there was a period with 20% loss from a peak to a trough. That's a lot. The first strategy however, had suffered only a 2% loss from peak to a trough. The fact that the second strategy generated 20% in the past year does not necessarily means that it will generate the same results this year as ***past performance can never guarantee future results***, but it is safe to assume that the second strategy is much more volatile and risky than the first one. Another aspect to consider is that in a low interest rate environment (as is the case in all of the western world) the cost of leverage is very cheap and so, with the use of leverage with a ratio of 1:2 in the first strategy, we can generate almost the same return (20% minus leverage cost) with much less risk. To conclude: the first strategy is the better one. We can have the same returns as the second strategy but with much less risk. The key takeaway from this example is that as important the strategy's profit and maximum drawdown are on their own, it is the combination of them that matters the most.

Aside from the profit and maximum drawdown, for each backtesting we'd also want to see the ratio of winning trades and losing trades and the maximum profit the strategy generated during the backtesting process.

Our StrategyResult class should therefore look like this:

```
import java.util.Comparator;
import java.util.List;

public class StrategyResult implements Comparator<StrategyResult>{

	private double profit;
	private double maxDrawdown;
	private double maxProfit;
	private int numOfWinningTrades;
	private int numOfLosingTrades;
	private int numOfTrades;
	private double winsRatio;
	private double lossesRatio;
	private Strategy strategy;


	public StrategyResult() {

	}

	public StrategyResult(double profit, double maxProfit, double maxDrawdown, int numOfWinningTrades,
				int numOfLosingTrades, Strategy strategy) {

		this.profit = profit;
		this.maxProfit = maxProfit;
		this.maxDrawdown = maxDrawdown;
		this.numOfWinningTrades = numOfWinningTrades;
		this.numOfLosingTrades = numOfLosingTrades;
		this.numOfTrades = numOfWinningTrades+numOfLosingTrades;
		this.strategy = strategy;

		if(this.numOfTrades==0) {	// no trades were made => ratio is 0.
			this.winsRatio=0;
			this.lossesRatio=0;
		}else {
			this.winsRatio = ((double) this.numOfWinningTrades)/this.numOfTrades;
			this.lossesRatio = ((double) this.numOfLosingTrades)/this.numOfTrades;
		}
	}

		// getters and setters goes here...


	@Override
	public int compare(StrategyResult sr1, StrategyResult sr2) {
		if (sr1.getProfitInPerc() < sr2.getProfitInPerc()) return 1;
		if (sr1.getProfitInPerc() > sr2.getProfitInPerc()) return -1;
		return 0;
	}

	public static double calculateAvgStrategyProfit(List <StrategyResult> strategySummary){

	    double sumOfProfits=0;

	    for (StrategyResult strategyResult : strategySummary) {
			sumOfProfits+=strategyResult.getProfitInPerc();
	    }

	    double avgProfit=sumOfProfits/strategySummary.size();

	    return avgProfit;

	}

}
```

Key notes for the `StrategyResult` class:

The fields `profit`, `maxProfit`, `maxDrawdown` represents percentage values. It means that if a strategy generates a 2% return then the value of `profit` is 0.02. we use percentage since we want the strategy to be generic and to fit for any trade size.

The `strategy` field holds the data of all of the parameters that were used for that specific backtest

the class implements the Comparator interface and therefore has to override the compare method. We will use this method to compare the profit of each backtest we're going to run for a particular strategy. Since our goal is to maximize our profits, we need to backtest our strategy multiple times while fine tuning the strategy's parameters each time, in order to find the best combination of parameters that will generate the maximum return. for example: we can run the strategy 11 times with different stop-loss in each run. We can start with a 0.5% stop-loss for the first run and then increment it by 0.05% for each consecutive run all the way up to 1%. Here is where the compare method comes into play. After backtesting the strategy multiple times with different parameters, we will have a list of backtest results and we will use the compare method to sort that list based on the profit in each backtest.

The purpose of the static method `calculateAvgStrategyProfit` `is` to go through the list of backtest results and calculate the average backtest profit.

Ok so now that we have a nice framework for displaying and analyzing the results of backtesting our strategies, the only thing left to do is to build the main method for testing strategies. For this purpose, we'll create the following Tester class:

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collections;
import java.util.List;

import com.fxcm.fix.Instrument;
import com.fxcm.fix.UTCDate;
import com.fxcm.fix.UTCTimeOnly;

public class Tester{

public static void main(String[] args){

          List <Candlestick> candleSticksList;
          double stopLossPerc=0.003;
          double takeProfitPerc=0.003;
          int bottomRSI=10;
          List <StrategyResult> strategySummary = new ArrayList<StrategyResult>();
          UTCDate startDate;
          UTCTimeOnly startTime;
          Instrument asset = new Instrument("AUD/USD");

          // get the current time and roll back 1 year
          Calendar instance = Calendar.getInstance();
          instance.roll(Calendar.YEAR, -1);

          // set the starting date and time of the historical data
          startDate = new UTCDate(instance.getTime());
          startTime = new UTCTimeOnly(instance.getTime());
```

```
                                    // Tester class continued...


        try{

            // create an instance of the JavaFixHistoryMiner
            HistoryMiner miner = new HistoryMiner("your user name", "your password", "Demo", startDate,
                                                  startTime, asset);

            // login to the api
            miner.login(miner,miner);

            // keep mining for historical data before logging out
            while(miner.stillMining) {

                Thread.sleep(1000);

            }


            Thread.sleep(1000);


            // log out of the api
            miner.logout(miner,miner);

            // convert rates to candlesticks
            miner.convertHistoricalRatesToCandleSticks();

            // display the collected rates
            miner.displayHistory();

            candleSticksList=miner.candlesticksList;

            RsiSignals.calculateRsiForDataSet(candleSticksList, 14);

            for (double i = stopLossPerc; i < stopLossPerc+0.005 ; i+=0.0005) {   // adjust stop loss

                for (double j = takeProfitPerc; j < takeProfitPerc+0.005; j+=0.0005) { // adjust take profit

                        for (int k = bottomRSI; k <= bottomRSI+20; k+=5) {  // adjust bottom and top rsi

                             Strategy rsiSignals = new RsiSignals(i, j, k);

                             StrategyResult sr = ((RsiSignals) rsiSignals).runSrtategy(candleSticksList);

                             strategySummary.add(sr);
                        }

                }

            }


           Collections.sort(strategySummary,new StrategyResult());       // sort results list

           double avergeProfit=StrategyResult.calculateAvgStrategyProfit(strategySummary);

           System.out.println("Average profit=> "+(double)Math.round(10000*avergeProfit)/100+"%");
```

```
                                    // Tester class continued...


    System.out.println("10 best results ____________________________________________________________ ");

    for (int i = 0; i < 10; i++) {

        RsiSignals rs = (RsiSignals) strategySummary.get(i).getStrategy();

        System.out.println("profit:"+(double)Math.round(strategySummary.get(i).getProfit()*10000)/100 +"%" +
        " | max-drawdown:"+(double)Math.round(strategySummary.get(i).getMaxDrawdown()*10000)/100 +"%" +
        " | wins:"+(double)Math.round(10000*strategySummary.get(i).getWinsRatio())/100 + "%" +
        " | losses:"+(double)Math.round(10000*strategySummary.get(i).getLossesRatio())/100 + "%" +
        " | s-l:"+(double)Math.round(rs.getStopLoss()*10000)/100 +"%" +
        " | t-p:"+(double)Math.round(rs.getTakeProfit()*10000)/100 +"%" +
        " | bottom-rsi:"+rs.getFloorRSI() +
        " | top-rsi:"+rs.getCeilingRSI() + "\n");
    }


    System.out.println("10 worst results ____________________________________________________________ ");

    for (int i = strategySummary.size()-1; i > strategySummary.size()-11; i--) {

        RsiSignals rs = (RsiSignals) strategySummary.get(i).getStrategy();

        System.out.println("profit: "+(double)Math.round(strategySummary.get(i).getProfit()*10000)/100+"%" +
        " | max-drawdown:"+(double)Math.round(strategySummary.get(i).getMaxDrawdown()*10000)/100 +"%" +
        " | wins:"+(double)Math.round(10000*strategySummary.get(i).getWinsRatio())/100 + "%" +
        " | losses:"+(double)Math.round(10000*strategySummary.get(i).getLossesRatio())/100 + "%" +
        " | s-l:"+(double)Math.round(rs.getStopLoss()*10000)/100 +"%" +
        " | t-p:"+(double)Math.round(rs.getTakeProfit()*10000)/100 +"%" +
        " | bottom-rsi:"+rs.getFloorRSI() +
        " | top-rsi:"+rs.getCeilingRSI() + "\n");
    }

   }catch (Exception e) {
        e.printStackTrace();
   }

  }

}
```

In the main method we start with creating an instance of the Calendar class to take advantage of the powerful roll method. We then use this instance to set the `startDate` and `startTime` variables before passing them to the HistoryMiner constructor along with our account credentials and the instrument we want the data for. after constructing a HistoryMiner instance, we call the login method and assign the HistoryMiner instance itself as listeners.

For the strategy testing process to be successful, we need it to be synchronous. We first need to get all the historical data from the FXCM server and *only* then run the strategy and display the results. We can achieve this by using a *while* loop with the `stillMining` flag as the exit condition. As long as the program is still mining for historical data, it will stay in the loop and only after that process is complete, it would move forward to test the strategy.

After we obtain all the data that we need, we logout and convert our raw data into candlesticks. then we call `calculateRsiForDataSet` - the static method of the `RsiSignals` class, and pass it the list of candlesticks and the duration we want to use in the calculation. We can choose any positive duration we want for the calculation, but as mentioned before, 14 is considered the lucky number so we'll use that.

Since our goal is to maximize our profits, we will backtest our strategy multiple times while fine tuning the strategy's parameters each time, in order to find the best combination of parameters that will generate the maximum return. We are going to use 3 *for* loops, one inside the other, where each loop is responsible for adjusting only one parameter: stop-loss, take-profit or the top and bottom RSI. In the inner most loop we create an instance of `RsiSignals` with the proper arguments, run the strategy for each instance and add the result to our list of strategy results.

Bear in mind that we can choose any (logical) range we want for each parameter as long as it makes sense in terms of trading. In this example we initialize the stop-loss to 0.3% and increment it by 0.05% in each iteration until it reaches 0.75%. The same thing goes for the take-profit. We also initialize the bottom RSI to 10 (which means the top RSI is 90 as together they must add up to a 100) and increment it by 5 in each iteration until it reaches 30. All in all, we run the strategy 500 times with different parameters in each time: 10 different stop-loss X 10 different take-profit X 5 different bottom RSI.

In the last part of the Tester class we display the results (see below). s-l and t-p stands for stop-loss and take-profit, respectively.

The following results were derived from backtesting the strategy with 1-minute historical data for the AUD/USD pair, ranging from October 9, 2016 until October 9, 2017. Obviously results for different assets or different time periods can vary considerably.

```
Average profit=> 2.85%

10 best results _____________________________________________________________________________________

profit:25.65% | max-drawdown:-4.38% | wins:57.07% | losses:42.93% | s-l:0.45% | t-p:0.45% | bottom-rsi:30 | top-rsi:70

profit:23.75% | max-drawdown:-3.67% | wins:74.40% | losses:25.60% | s-l:0.65% | t-p:0.30% | bottom-rsi:30 | top-rsi:70

profit:20.40% | max-drawdown:-3.67% | wins:70.21% | losses:29.79% | s-l:0.70% | t-p:0.40% | bottom-rsi:25 | top-rsi:75

profit:19.55% | max-drawdown:-3.36% | wins:58.39% | losses:41.61% | s-l:0.55% | t-p:0.50% | bottom-rsi:30 | top-rsi:70

profit:19.40% | max-drawdown:-5.38% | wins:57.72% | losses:42.28% | s-l:0.35% | t-p:0.30% | bottom-rsi:30 | top-rsi:70

profit:19.40% | max-drawdown:-3.63% | wins:73.60% | losses:26.40% | s-l:0.65% | t-p:0.30% | bottom-rsi:25 | top-rsi:75

profit:18.55% | max-drawdown:-3.82% | wins:72.24% | losses:27.76% | s-l:0.70% | t-p:0.35% | bottom-rsi:25 | top-rsi:75

profit:18.50% | max-drawdown:-4.69% | wins:51.71% | losses:48.29% | s-l:0.40% | t-p:0.45% | bottom-rsi:30 | top-rsi:70

profit:18.30% | max-drawdown:-5.65% | wins:71.36% | losses:28.64% | s-l:0.60% | t-p:0.30% | bottom-rsi:25 | top-rsi:75

profit:18.00% | max-drawdown:-3.42% | wins:60.94% | losses:39.06% | s-l:0.60% | t-p:0.50% | bottom-rsi:30 | top-rsi:70
```

```
10 worst results ____________________________________________________________________________________

profit: -13.30% | max-drawdown:-15.35% | wins:40.64% | losses:59.36% | s-l:0.50% | t-p:0.60% | bottom-rsi:25 | top-rsi:75

profit: -12.95% | max-drawdown:-17.42% | wins:38.50% | losses:61.50% | s-l:0.50% | t-p:0.65% | bottom-rsi:25 | top-rsi:75

profit: -12.45% | max-drawdown:-15.42% | wins:52.46% | losses:47.54% | s-l:0.75% | t-p:0.55% | bottom-rsi:30 | top-rsi:70

profit: -12.10% | max-drawdown:-16.39% | wins:40.37% | losses:59.63% | s-l:0.60% | t-p:0.70% | bottom-rsi:20 | top-rsi:80

profit: -11.10% | max-drawdown:-11.50% | wins:37.20% | losses:62.80% | s-l:0.50% | t-p:0.70% | bottom-rsi:30 | top-rsi:70

profit: -10.75% | max-drawdown:-16.19% | wins:35.17% | losses:64.83% | s-l:0.45% | t-p:0.70% | bottom-rsi:30 | top-rsi:70

profit: -10.50% | max-drawdown:-12.95% | wins:30.22% | losses:69.78% | s-l:0.35% | t-p:0.70% | bottom-rsi:30 | top-rsi:70

profit: -9.75%  | max-drawdown:-11.74% | wins:50.24% | losses:49.76% | s-l:0.65% | t-p:0.55% | bottom-rsi:25 | top-rsi:75

profit: -9.75%  | max-drawdown:-14.18% | wins:36.14% | losses:63.86% | s-l:0.50% | t-p:0.75% | bottom-rsi:25 | top-rsi:75

profit: -9.60%  | max-drawdown:-12.08% | wins:25.88% | losses:74.12% | s-l:0.30% | t-p:0.75% | bottom-rsi:25 | top-rsi:75
```

Below is a full list of the classes we've used in this tutorial:

StrategyResult.java:

```
import java.util.Comparator;
import java.util.List;

public class StrategyResult implements Comparator<StrategyResult>{

	private double profit;
	private double maxDrawdown;
	private double maxProfit;
	private int numOfWinningTrades;
	private int numOfLosingTrades;
	private int numOfTrades;
	private double winsRatio;
	private double lossesRatio;
	private Strategy strategy;


	public StrategyResult() {

	}

	public StrategyResult(double profit, double maxProfit, double maxDrawdown, int numOfWinningTrades,
				int numOfLosingTrades, Strategy strategy) {

		this.profit = profit;
		this.maxProfit = maxProfit;
		this.maxDrawdown = maxDrawdown;
		this.numOfWinningTrades = numOfWinningTrades;
		this.numOfLosingTrades = numOfLosingTrades;
		this.numOfTrades = numOfWinningTrades+numOfLosingTrades;
		this.strategy = strategy;

		if(this.numOfTrades==0) {	// no trades were made => ratio is 0.
			this.winsRatio=0;
			this.lossesRatio=0;
		}else {
			this.winsRatio = ((double) this.numOfWinningTrades)/this.numOfTrades;
			this.lossesRatio = ((double) this.numOfLosingTrades)/this.numOfTrades;
		}
	}

		// getters and setters goes here...


	@Override
	public int compare(StrategyResult sr1, StrategyResult sr2) {
		if (sr1.getProfitInPerc() < sr2.getProfitInPerc()) return 1;
		if (sr1.getProfitInPerc() > sr2.getProfitInPerc()) return -1;
		return 0;
	}

	public static double calculateAvgStrategyProfit(List <StrategyResult> strategySummary){

	    double sumOfProfits=0;

	    for (StrategyResult strategyResult : strategySummary) {
			sumOfProfits+=strategyResult.getProfitInPerc();
	    }

	    double avgProfit=sumOfProfits/strategySummary.size();

	    return avgProfit;

	}

}
```

Tester.java:

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collections;
import java.util.List;
import com.fxcm.fix.Instrument;
import com.fxcm.fix.UTCDate;
import com.fxcm.fix.UTCTimeOnly;

public class Tester{

   public static void main(String[] args){

          List <Candlestick> candleSticksList;
          double stopLossPerc=0.003;
          double takeProfitPerc=0.003;
          int bottomRSI=10;
          List <StrategyResult> strategySummary = new ArrayList<StrategyResult>();
          UTCDate startDate;
          UTCTimeOnly startTime;
          Instrument asset = new Instrument("AUD/USD");

          // get the current time and roll back 1 year
          Calendar instance = Calendar.getInstance();
          instance.roll(Calendar.YEAR, -1);

          // set the starting date and time of the historical data
          startDate = new UTCDate(instance.getTime());
          startTime = new UTCTimeOnly(instance.getTime());

          try{

            // create an instance of the JavaFixHistoryMiner
            HistoryMiner miner = new HistoryMiner("your user name", "your password", "Demo", startDate,
                                                  startTime, asset);

            // login to the api
            miner.login(miner,miner);

            // keep mining for historical data before logging out
            while(miner.stillMining) {
                Thread.sleep(1000);
            }

            Thread.sleep(1000);

            // log out of the api
            miner.logout(miner,miner);

            // convert rates to candlesticks
            miner.convertHistoricalRatesToCandleSticks();

            // display the collected rates
            miner.displayHistory();

            candleSticksList=miner.candlesticksList;

            RsiSignals.calculateRsiForDataSet(candleSticksList, 14);
```

```
                                    // Tester class continued...


    for (double i = stopLossPerc; i < stopLossPerc+0.005 ; i+=0.0005) {   // adjust stop loss

        for (double j = takeProfitPerc; j < takeProfitPerc+0.005; j+=0.0005) {    // adjust take profit

                for (int k = bottomRSI; k <= bottomRSI+20; k+=5) { // adjust bottom and top rsi

                        Strategy rsiSignals = new RsiSignals(i, j, k);
                        StrategyResult sr = ((RsiSignals) rsiSignals).runSrtategy(candleSticksList);
                        strategySummary.add(sr);
                }

        }

    }

    Collections.sort(strategySummary,new StrategyResult());       // sort results list

    double avergeProfit=StrategyResult.calculateAvgStrategyProfit(strategySummary);

    System.out.println("Average profit=> "+(double)Math.round(10000*avergeProfit)/100+"%");

    System.out.println("10 best results _____________________________________________________________ ");
    for (int i = 0; i < 10; i++) {

        RsiSignals rs = (RsiSignals) strategySummary.get(i).getStrategy();

        System.out.println("profit:"+(double)Math.round(strategySummary.get(i).getProfit()*10000)/100 +"%" +
        " | max-drawdown:"+(double)Math.round(strategySummary.get(i).getMaxDrawdown()*10000)/100 +"%" +
        " | wins:"+(double)Math.round(10000*strategySummary.get(i).getWinsRatio())/100 + "%" +
        " | losses:"+(double)Math.round(10000*strategySummary.get(i).getLossesRatio())/100 + "%" +
        " | s-l:"+(double)Math.round(rs.getStopLoss()*10000)/100 +"%" +
        " | t-p:"+(double)Math.round(rs.getTakeProfit()*10000)/100 +"%" +
        " | bottom-rsi:"+rs.getFloorRSI() +
        " | top-rsi:"+rs.getCeilingRSI() + "\n");
    }


    System.out.println("10 worst results _____________________________________________________________ ");
    for (int i = strategySummary.size()-1; i > strategySummary.size()-11; i--) {

        RsiSignals rs = (RsiSignals) strategySummary.get(i).getStrategy();

        System.out.println("profit: "+(double)Math.round(strategySummary.get(i).getProfit()*10000)/100+"%" +
        " | max-drawdown:"+(double)Math.round(strategySummary.get(i).getMaxDrawdown()*10000)/100 +"%" +
        " | wins:"+(double)Math.round(10000*strategySummary.get(i).getWinsRatio())/100 + "%" +
        " | losses:"+(double)Math.round(10000*strategySummary.get(i).getLossesRatio())/100 + "%" +
        " | s-l:"+(double)Math.round(rs.getStopLoss()*10000)/100 +"%" +
        " | t-p:"+(double)Math.round(rs.getTakeProfit()*10000)/100 +"%" +
        " | bottom-rsi:"+rs.getFloorRSI() +
        " | top-rsi:"+rs.getCeilingRSI() + "\n");
    }

   }catch (Exception e) {
        e.printStackTrace();
   }

 }

}
```