Project Dow: Extending EclipseTrader

Emmanuel Sotelo

CS491A/B

Summer/Fall 2008

Abstract

The investor is always looking for a way to maximize his return on investment. In order to achieve this goal, he must analyze the stock he plans to purchase through at least one of the many stock analysis methods. His goal is to be profitable, therefore he must predict the best moment to buy and then sell the stock. In his quest to achieve his goal, the investor must make sure that he has an adequate toolset at his disposal. The toolset selected will be dependent on the stock analysis method chosen by the investor.

Technical analysis is a method for analyzing stocks. It studies a stock's price and volume history in an effort to forecast future price movements. The investor who makes usage of technical analysis for investing decisions, requires the ability find and analyze stock price patterns in his toolset.

For this project we extend the functionality of a stock analysis program known as EclipseTrader. It is a stock analysis tool primarily built with the technical analyst in mind. To extend its functionality, we add the ability to define custom technical indicators, the ability to back test algorithmic and technical indicator based trading strategies, better data export capabilities, and neural network based stock price forecasting capability. These new features will increase its value as a technical analysis tool.

Introduction

Technical analysis is a stock analysis method that tries to forecast future movements in stock price by analyzing the its past price and volume movements. It makes use of known price movement patterns in an attempt to forecast where the price of the stock is likely to go. By analyzing its price movement patterns, the investor can make predictions on where the price of the stock is likely to go.

Technical analysis makes use of mathematical transformations known as Technical Indicators. Technical indicators give the investor a method to analyze stock price patterns. They expose patterns which may not be immediately obvious in the stock price chart. Technical indicators can be simple or complex transformations. EclipseTrader comes

preloaded with some of the most widely used technical indicators. However, it is sometimes necessary to go beyond the usage of the commonly used indicators. Currently the only way to add new technical indicators to EclipseTrader requires tedious programming and detailed knowledge of the EclipseTrader architecture. Such a task is easily accomplished by the average user. In this project we extend the functionality of EclipseTrader by allowing the trader to easily define custom technical indicators.

The investor cannot simply rely on one single tool for his analysis. There are times when it is necessary for the investor to analyze the technical indicator data in other programs. However, EclipseTrader provides no way for technical indicator data to be exported. To solve this problem, we add the capability to export technical indicator data. The data is exported in the commonly accepted Comma Separated Values (CSV) format. Also, the exportable data includes data generated by custom technical indicators.

Having the stock price movement patterns exposed by the technical indicators, the investor can now begin to formulate an algorithmic strategy for the buying and selling of stocks. With an algorithmic trading strategy, stocks are only purchased or sold when a specific set of conditions is met. The use of algorithmic strategies to trade stock removes the emotion from the process, thus making it more objective. However, it should be noted that before being put into use with real money, the algorithmic strategy should be thoroughly tested. Also, they cannot account for unexpected events and should be revised periodically. With this in mind, we add to EclipseTrader, the capability to test out algorithmic trading strategies. With this capability, the investor is able to use a simple Graphical User Interface (GUI) to define a technical indicator driven trading strategy. In case this in not enough for the investor, we also add the capability for complex trading algorithms to be defined. The algorithms are defined in a Java compatible format without the necessity to compile any code.

In technical analysis, the ability to analyze or discover patterns in stock price momentum is essential. However, sometimes the pattern might not be easily observed or it is very complex to detect using traditional methods. Fortunately for the investor, computers are great at pattern recognition and analysis. Artificial Neural Networks (ANNs) are

excellent pattern recognizers, and as a result are very suitable for use in technical analysis. An ANN could be trained with the data of a stock's previous price movements. Once trained, the ANN can be used in an attempt to predict that stock's future price. We also extend EclipseTrader by providing it with the capability to train an ANN using a stock's price and volume history and then using that ANN to forecast the stock's next day's closing price.

Technological Background

EclipseTrader

This project is built on top of EclipseTrader. It is an open-source stock analysis program. Some of its features include automated stock data retrieval and storage, stock data charting, technical analysis facilities, virtual account management, financial news watching, and simple price pattern analysis with backtesting. However, it is necessary to note that this program is incomplete and some features are not fully implemented. Version 0.30 is the highest version available.

Figure 1: EclipseTrader displaying stock price chart.

The Eclipse Rich Client Platform

EclipseTrader is an Eclipse Rich Client Platform (RCP) application. The Eclipse RCP is an open source application development framework. It provides developers with a rich set of components to build cross platform enterprise grade applications. By building their

application on the Eclipse RCP, developers can save time by focusing directly on the business logic of their application. Some of the facilities that the Eclipse RCP provides developers include, GUI components, file system manipulation, user help systems, online software update mechanisms, thread creation and scheduling, and more.

The Eclipse RCP is built on a component (Plug-in) based modular architecture. The plug-ins, connect to other plug-ins through extension points. An extension point is a programmer defined point in the plug-in where other plug-ins may connect to add additional functionality. It is up to the plug-in developer to decide if it will include an extension point. In order for that an application to be considered a RCP, it must contain a minimal set of plug-ins.

In the Eclipse RCP, all functionality lies in the plug-ins. In fact, almost everything in the Eclipse RCP is a plug-in. The only component that is not a plug-in is the Platform Runtime. The Platform Runtime acts as a basic kernel, its only function is to get the application started. Also, plug-ins in the Eclipse RCP are only loaded upon first use, this improves loading time and helps conserve Random Access Memory (RAM).

BeanShell

BeanShell is an open-source Java interpreter. Not only does it interpret Java code, it also allows the user create simplified scripts without having to worry about all the formalities of the Java language such pre-initializing variables and objects or having to specify their type. BeanShell runs inside the Java Virtual Machine (JVM) allowing the user to import and access packages and classes. It can be embedded inside of java applications to provide them with scripting abilities. As an embedded system, objects can be passed into BeanShell to be evaluated by the user defined script and then retrieved back by the host application. Additionally, BeanShell maintains a very light weight memory footprint. The complete system has a memory footprint of approximately 280K.

The Neural Network

This project makes use of an Artificial Neural Network (ANN) framework. The framework used is described and implemented in Introduction to Neural Networks for Java by Jeff Heaton. Specifically, we use the network structure described in Chapter 10: "Application to the Financial Markets". Using this framework saves us from the complexities of having to design and implement ANN algorithms. Instead, we only deal with the implementation of the ANN structure. The ANN is assembled in an object oriented manner through the instantiation of java objects and the invocation of methods. However, the ANN is not taken directly from the book. Changes are made to the network to make it adequate for our purposes. Also, the neural network code is released as open source. Thus, allowing us to freely use it without the necessity to ever have to worry about licensing fees.

System Overview

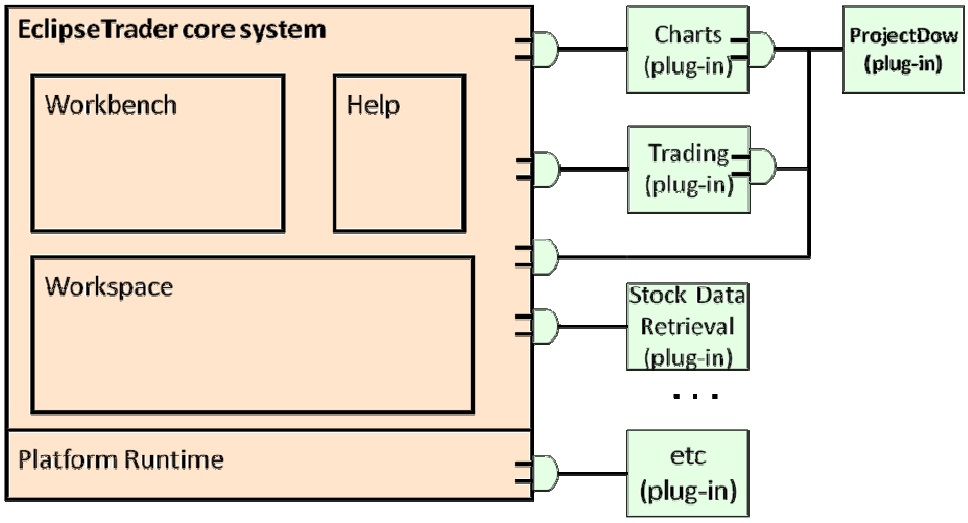

Figure 2: General overview of the Project Dow architecture

As an Eclipse RCP application, we build upon EclipseTrader through the use of its plug-in extension points. Thus, Project Dow is primarily an EclipseTrader plug-in. However, a bug found in the backtesting system was fixed. As consequence, a new slightly different version of EclipseTrader is forked in the process.

EclipseTrader is composed of many plug-ins. Project Dow extends EclipseTrader by plugging into them. It does so in three ways, first it plugs into the EclipseTrader core system. The core system is composed of a basic Eclipse RCP application and the Core plug-in which defines the fundamental classes and interfaces. It will then plug in into the Charts plug-in. The Charts plug-in is primarily responsible for displaying the stock and indicator data to the user. The user can view the data in graph or table form. The Charts plug-in also provides the interface and extendable classes used for defining technical indicators. This project extends the Charts plug-in by providing it with the ability to allow the user define custom technical indicators. And finally, Project Dow will plug into the Trading plug-in. This plug-in provides the backtesting system that is used for testing the user defined algorithm and technical indicator trading strategies. It also provides us with the interfaces and classes that we use to extend it by creating the Algorithmic and Indicator trading system components of Project Dow.

Since it is dependent on technical indicator data, the Indicator trading system component interacts with the Charts plug-in by accessing the indicator interfaces so that it may compute the necessary technical indicator data for its operations. These same indicator interfaces are also accessed by our project's data export component to allow technical indicator data to be exported to CSV format. Access to the custom indicator component we create is included in all this.

The Stock Price Predictor component of our plug-in interfaces only with the EclipseTrader core system for its operations. It uses only stock price and transaction volume data which are provided by the Core plug-in. Unlike Custom Indicator component, this component does not interface with any of the other components in our plug-in.

Design and Implementation

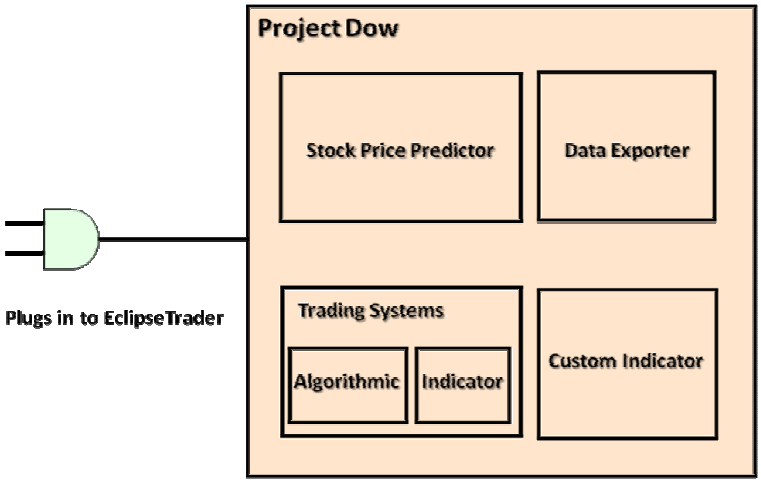

Figure 3: Overview of Project Dow Plug-in

Custom Technical Indicator

The custom technical indicator makes use of the BeanShell framework. In the input box for the technical indicator, the user defines an indicator. The custom indicator is defined using Java code. That code is stored in the indicator's preference page object where it is passed to the custom indicator class via a 'get' method call. In the custom indicator class, the custom code is then passed to the BeanShell interpreter for execution. In the his code, user must set the value of the variable *returnValue* to the result of the custom indicator. After the user code has been executed, the custom indicator class retrieves the value stored in the *returnValue* variable. Since this is the desired value, it is stored for later usage.

The user defined code will be executed several times. At each iteration, the variable that contains the stock's data will be updated with the data for the date is currently being analyzed. The result will then be stored with the others. The stock data variable contains the stock's open price, high price, low price, close price, and volume. When the program closes, the user code is not lost, it is automatically saved by EclipseTrader in its settings files ready to be retrieved when needed again in the future. However, the technical indicator data is not saved and must be recalculated once again. This occurs for all technical indicators.

```
returnValue = 2*(Math.sin(bars[0].getHigh()));
```

Example 1: A simple Custom Technical Indicator

This demonstration technical indicator retrieves the stock's high price for the current day, gets its Sine value, and multiplies it by 2. The result then stored in the variable *returnValue*.

Trading System: Algorithmic

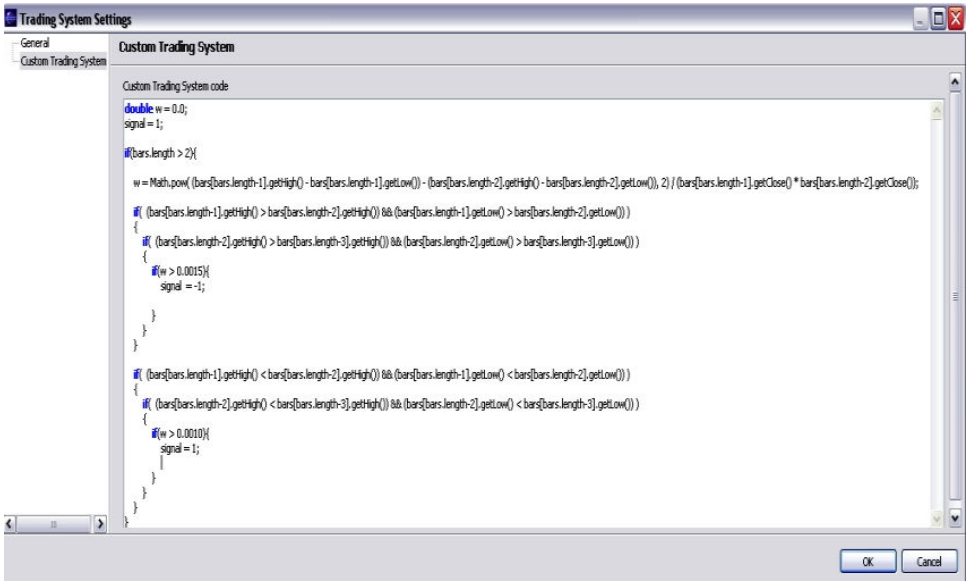

Figure 4: The Custom Trading System algorithm definition dialog.

The Algorithmic (Custom Trading System) trading system component also uses the BeanShell framework. As a result, it functions in a very similar manner to the Custom Technical Indicator component. First, the user code is saved in a preference page object where it is passed to the Custom Trading System class via a 'get' method call. In the custom trading system class, the custom code is then passed to the BeanShell interpreter for execution. However, instead of computing values for later storage, any data generated is for immediate usage. In his code, the user specifies conditions that will trigger a stock purchase or a stock sale. If the user wants to trigger a stock purchase, he sets the value of the variable *signal* equal to one (1), or negative one (-1) to trigger a stock sale. If the value is otherwise, a neutral (no action) signal will be triggered. The backtesting system will take the value of the variable *signal* and execute the appropriate action.

```
int last = bars.length-1;
If(bars[last].getClose() <= 90  ) signal = 1;

If(bars[last].getClose() >= 100  ) signal = -1;
```

Example 2: A simple Algorithmic Trading Strategy

This demonstration trading strategy will buy a stock when its price is at $90 or below and it will sell it when its price is at $100 or above.

It was during the implementation of this component that we found a bug in the backtesting system. The backtesting system made use of the *subList()* method from the *java.util.List* interface class in Java. The *subList()* method returns a sub list of items in between two indexes of a collection. However, this method excludes the item specified by the last index. This created a problem for us because it caused the backtesting system to exclude dates from its simulation. We solved this problem by using a 'for loop' based approach to create sub lists when necessary.

Trading System: Indicator

The Indicator Trading System allows the user to create and test trading strategies based on technical indicator data. The indicator trading system has access to the technical indicator data of a selected stock chart. Unlike the Custom Trading System where the user has to write an algorithm in Java code, the indicator trading system is completely GUI driven. The user selects from a drop-down list the indicator he would like to use and then specifies the buy or sell conditions in the designated input boxes. It is worthwhile to note that if the user has defined a Custom Technical Indicator for the selected chart, the user will be able use that indicator within this trading system.

Unfortunately, there exists a minor, but computationally inefficient issue when other plug-ins or components need to access technical indicator data. The indicator data that is computed when viewing a stock chart is computed inside of the class that draws the chart. We are not provide with a method to access this data. To solve this problem, we created a class to independently compute the indicator data of a stock chart whenever

necessary. As a result, the indicator trading system needs to compute the indicator data once again.

Indicator Data Exporter

The indicator Data Exporter allows the user to export the complete data set for a selected chart. This includes technical indicator data. While EclipseTrader already provides a data export feature, it only exports basic stock price and transaction volume data. However, this is data that is easily and freely obtainable. It is the technical indicator data that is of value to us. The data is exported CSV format. This format was chosen because it is a widely accepted format by data analysis applications.

Since this component also makes use of technical indicator data, it also suffers from that same issue regarding the access to indicator data. Like the indicator trading system, this component must also recompute indicator data for the selected chart. This component also makes use of that same class that we created to recompute indicator data.

Stock Price Predictor

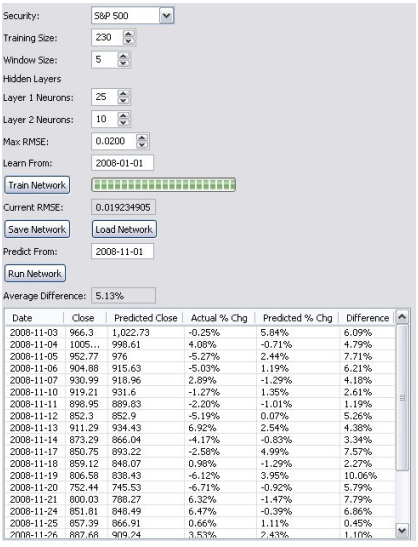

Figure 5: The Stock Price Predictor.

The Stock Price Predictor attempts to predict the following day's closing price for a selected stock. For its prediction, it uses the closing price of a stock and transaction volume data of that selected stock. The stock price predictor uses an Artificial Neural Network to

make its prediction. Specifically, it uses a FeedForward Neural Network. The network is trained using Backpropagation and Simulated Annealing. Since the process of training a network can take a prolonged amount of time, we create a separate java thread to train the network. This allows us to do other things while the network is being trained. However, for training the network, we just don't simply create a regular java thread. Instead, we take advantage of the Eclipse RCP Jobs API. By using an Eclipse Job thread, we can send the task to the background and are able to easily terminate thread from EclipseTrader if needed. Additionally, the user is able to save the network for future use. This is done using object Serialization.

The user has control over several of the network's parameters. It is up to the user to decide how many days the network should use for its training. The user can also decide on the size of the temporal window by adjusting the 'Window Size' field. This parameter decides how many days in the recent past, the network will take into account when making its prediction. To make the next prediction, the window will then slide on day forward. Next, the user can decide whether he wants one or two hidden layers. If the user wants only one hidden layer, then he sets the neurons parameter for layer two to zero, otherwise the network will have two hidden layers. By adjusting the 'Neurons' parameter field for each hidden layer, the user controls how many neurons he wants his hidden layers to have. The next adjustable parameter is the Max Root Mean Squared Error (RMSE). The Max RMSE parameter controls how large of an RMSE the user is willing to tolerate. Networks with a smaller RMSE are generally more precise in their predictions. And finally, in the 'Learn From' parameter field, the user controls from what date the network should start learning from.

System Evaluation

The Custom Indicator performed as expected with regards to the results produced. Also, the Custom Indicator operates seamlessly with other components just like the indicators that are hardwired into the system. However, since the user defined code is executed by an interpreter, the performance of the custom indicator is somewhat slower than that of the technical indicators which are hardwired into the system. This can be seen

when adding technical indicators to a chart. The Custom Technical Indicators will usually render at slightly slower speed than the hardwired ones

The Custom (Algorithmic) Trading System component also performed as expected with regards to the results produced. Like Custom Indictor, this component also executes its code on an interpreter. However, unlike the Custom Indicator, this issue is remedied by the fact that the backtesting component executes on a separate java thread.

The Indicator Trading System performs as expected. Since it does not execute any interpreted code, it runs at noticeably faster rate than the Custom Trading System.

The Indicator Data Export component works as expected and no issues were found.

Since making an exact prediction for a stock's closing price is near impossible. The best we can hope for is the closest prediction to the actual value. Thus, the Stock Price Predictor performed as expected. However, there was an insufficient amount of time in order to perform an adequate benchmark against real world results.

Conclusion

The primary goals of this project were accomplished. Thus, we consider this project to be a success. As a stock analysis tool, EclipseTrader is now more useful and valuable than before. This project has also served to showcase the benefits of the Eclipse RCP plug-in architecture.

However, there are still many improvements that can be made. Especially to the Stock Price Predictor component. In the future, we like to explore using technical indicator data to train the neural network and create a trading system that executes trades based on the neural network results. We would also like to explore Genetic Algorithm (GA) methods for network training. For example, we would like to simultaneously train a population of networks for several generations and then select the best performing network.

References

1. Heaton, Jeff. Introduction to Neural Networks for Java. 2nd ed. Chesterfield, MO: Heaton Research, 2008. 247-72.

2. Eclipse Trader: http://eclipsetrader.sourceforge.net

3. Eclipse Rich Client Platform: http://wiki.eclipse.org/index.php/Rich_Client_Platform

4. Technical Analysis: http://en.wikipedia.org/wiki/Technical_analysis

5. BeanShell: http://www.beanshell.org

6. Technical Indicators: http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators:introduction_to_tech