Machine Learning Engineer Nanodegree
Capstone Project

Stock Predictor & Automatic Trading System

Miguel Tasende

October 22, 2017

1 Definition

Project Overview

Predicting the stock price trend by interpreting the seemly chaotic market data has always been an attractive topic to both investors and researchers. Among those popular methods that have been employed, Machine Learning techniques are very popular due to the capacity of identifying stock trend from massive amounts of data that capture the underlying stock price dynamics. According to market efficiency theory, US stock market is semi-strong efficient market, which means all public information is calculated into a stock's current share price, meaning that neither fundamental nor technical analysis can be used to achieve superior gains in a short-term (a day or a week) [1].

Much economic research has been conducted into the Efficient Markets Hypothesis theory, which posits that stock prices already reflect all available information and are therefore unpredictable. According to the EMH, stock prices will only respond to new information and so will follow a random walk. If they only respond to new information, they cannot be predicted. Most research with machine learning forecasting has focused on Artificial Neural Networks (ANN). Recent research in the field has used another technique known as Support Vector Machinesin addition to or as an alternative to ANNs [2].

In this Capstone Project, some other techniques will be explored, to try defy the Efficient Markets Hypothesis.

Problem Statement

Given a time series of (Open, High, Low, Close, Volume) values for the stocks that are presently (as of January 17th, 2017) included in the S&P500 index, a prediction of the "Close" value for any of them, at some "future" date, is to be made. By "future" it is meant that the date must be after the dates used as training data. The necessary time period of data needed for the prediction, as well as the prediction expected accuracy in future dates, are to be explored in the project, and the solution should take that into account.

As a secondary problem, an Automatic Trading System will be implemented, and in at least one version, will use information from the previously trained predictor. The goal of the Automatic Trading System is to try to maximize profit in a certain time horizon. Given today's values of (Open, High, Low, Close, Volume), decide to BUY or SELL some equity (and in which quantity), assuming it is possible to do so at the Close price (that is an approximation, of course). The recommender would have been trained with historical data, and will continue to learn from new data, as it arrives.

Note: This proposal is inspired in the Assignments of the "Machine Learning for Trading" course, by Tucker Balch (Udacity) and it's corresponding course in Georgia Tech [3]. The soultion is expected to be a slight improvement (or at least an interesting experiment), combining Supervised Learning and Reinforcement Learning on the same trading system.

Metrics

For the prediction part, one of the metrics used was the R^2 score of the predictions against the historical values for the periods defined (7 days, 14 days, 28 days, and 56 days after the last training sample). Initially a standarized version of the RMSE was proposed, but the R^2 is very similar, and was already implemented in the used libraries. To get a single number (there were about 300 stocks to test) the simple average was taken. The other used metric is the mean relative error on the final dates of those periods (day 7, day 14, day 28, day 56) among all the predicted stock values. In particular, a goal of the project was to achieve a mean relative error of less than 5% in day 7. The formulas for the metrics can be seen below (the index "j" refers to the stock that is being taken into account, and "i" is the "time" index).

$$\mu^j = \sum_i y_i^j$$

$$r2^j = 1 - \sqrt{\frac{\sum_i (ypred_i^j - y_i^j)^2}{\sum_i (y_i^j - \mu^j)^2}}$$

$$Avg_r 2 = \sum_j r2^j$$

Mean relative error:

$$MRE^i = \sum_j \left| \frac{ypred_i^j - y_i^j}{y_i^j} \right|$$

For the automatic trader, the metrics to use will be the total cumulative return in the periods, and the Sharpe Ratio in that period. Those metrics will be used to compare against the S&P500 index as a benchmark.

$$CumRet^j = \frac{value_i^j}{value_0} - 1$$

And the Sharpe Ratio in the period is the one below (the free interest rate will be considered zero). "mean" is the regular mean, and "std" is the standard deviation.

$$SR^j = \sqrt{SamplesPerYear} \times \frac{mean(dailyReturn^j)}{std(dailyReturn^j)}$$

The R^2 score is the typical metric used to assess the "goodness of fit" of an approximation to a series of real values. It gives a global, standarized metric of how close are the predicted values to the "real" ones. That's why it was chosen to evaluate the problem of predicting the real valued "Close" prices. The MRE gives a more intuitive idea of the error that the predictor is expected to produce.

Cumulative Return means the total earnings of the investment. It is the naÃ¯ve "goal" of the Automatic Trader, and so it is a natural metric to use. Despite that, volatility is an undesired feature of any investing strategy. That is for, at least, two reasons:

- Sometimes volatility is seen as a measure of "risk", and risky investments are undesired.
- A portfolio that has a lot of volatility is probably less liquid: if you want to withdraw the fruits of your investment it is very likely that, with a volatile strategy, you will find that at the moment you need the money the portfolio is going through a bad period, and a that withdrawal could cause heavy losses. If the strategy has less volatility, the returns are more consistent and the moment to retire the money matters less (the money is more available).

For that reasons the Sharpe Ratio is considered as the best metric for the Automatic Trader, because it considers the Cumulative Return, but divided by the Volatility.

2 Analysis

Data Exploration

The data consists of the (Open, High, Low, Close, Volume) daily values for the stocks that are presently included in the S&P500, from January 22^{nd}, 1993, until December 31^{st}, 2016. Initially, the idea was to download them from Yahoo Finance, but as the API was not functional at the time, Google Finance was used instead. Some functions were implemented to download the data and save it as a pickled dataframe. As the amount of data is relatively big (at least for a regular internet connection), the functions only update the symbols that are not present in the dataframe, and save to disk everytime a symbol is completely downloaded. That way, if anything fails, the progress is not lost. As the API limited the amount of data per call, it was retrieved in batches of 10 years, and then put together. An approximately 120MB pickled dataframe is able to contain all the data needed for the project.
There are three "axis" for the data: Date, Symbol and Feature. For that reason a multi-indexed dataframe was chosen to store it. The structure can be seen below.

		SPY	MMM	ABT	ABBV	ACN	ATVI	AYI	ADBE	AMD	AAP	...	XEL	XRX	XL
date	feature														
1993-02-04	Open	0.00	0.00	0.00	NaN	NaN	NaN	NaN	0.00	0.00	NaN	...	0.00	0.00	0.0
	High	45.09	26.47	6.97	NaN	NaN	NaN	NaN	2.78	20.88	NaN	...	22.81	14.17	2.7
	Low	44.88	25.88	6.78	NaN	NaN	NaN	NaN	2.70	20.12	NaN	...	22.50	14.09	2.6
	Close	45.00	26.06	6.84	NaN	NaN	NaN	NaN	2.73	20.12	NaN	...	22.81	14.15	2.6
	Volume	531500.00	4122400.00	5190800.00	NaN	NaN	NaN	NaN	6441600.00	1330200.00	NaN	...	162800.00	1675602.00	703
1993-02-05	Open	0.00	0.00	0.00	NaN	NaN	NaN	NaN	0.00	0.00	NaN	...	0.00	0.00	0.0
	High	45.06	27.19	6.88	NaN	NaN	NaN	NaN	2.77	20.50	NaN	...	22.69	14.38	2.7
	Low	44.72	26.41	6.69	NaN	NaN	NaN	NaN	2.59	19.62	NaN	...	22.31	14.11	2.4
	Close	44.97	27.19	6.88	NaN	NaN	NaN	NaN	2.60	19.62	NaN	...	22.56	14.38	2.5
	Volume	492100.00	4561600.00	4448400.00	NaN	NaN	NaN	NaN	9843200.00	1024600.00	NaN	...	73600.00	3104598.00	719

Figure 1: The raw initial dataset.

As can be readily seen there is some missing data. That is due to "real" missing data, and also because come of the companies that are presently in the index did not exist, or were not public, in the entire period.
One decision to make is how to deal with that missing data: Discard some symbols with a lot of missing data? Fill the missing data in way that keeps causality?
Also, to train the predictor, some samples will be taken, then another question arises, because it may not be the same to fill the missing data at the "full dataset" level than to do it at the "sample" level.
Those issues were addressed, and will be discussed below.

There are 503 symbols, with 5 features, in 6024 dates. The data used for the predictor is the "Close" data, and it looks like follows:

	SPY	MMM	ABT	ABBV	ACN	ATVI	AYI	ADBE	AMD	AAP	...	XEL	XRX	XLNX	XL	XYL	YHOO
date																	
1993-01-29	43.94	24.50	6.88	NaN	NaN	NaN	NaN	2.59	18.75	NaN	...	22.00	14.28	2.50	NaN	NaN	NaN
1993-02-01	44.25	24.69	6.88	NaN	NaN	NaN	NaN	2.72	19.12	NaN	...	22.19	14.09	2.62	NaN	NaN	NaN
1993-02-02	44.34	24.72	6.53	NaN	NaN	NaN	NaN	2.84	20.25	NaN	...	22.06	14.09	2.64	NaN	NaN	NaN
1993-02-03	44.81	25.19	6.91	NaN	NaN	NaN	NaN	2.70	20.50	NaN	...	22.38	14.03	2.68	NaN	NaN	NaN
1993-02-04	45.00	26.06	6.84	NaN	NaN	NaN	NaN	2.73	20.12	NaN	...	22.81	14.15	2.67	NaN	NaN	NaN

5 rows × 503 columns

Figure 2: Dataset used for the Predictor

The descriptive stats table for each feature are not too revealing, as there are 503 symbols. So the mean was plotted, and one clear outlier was found. After investigating it was concluded that the data for that symbol was wrong. The graph with the means for all the symbols, before discarding the outlier, can be seen below.

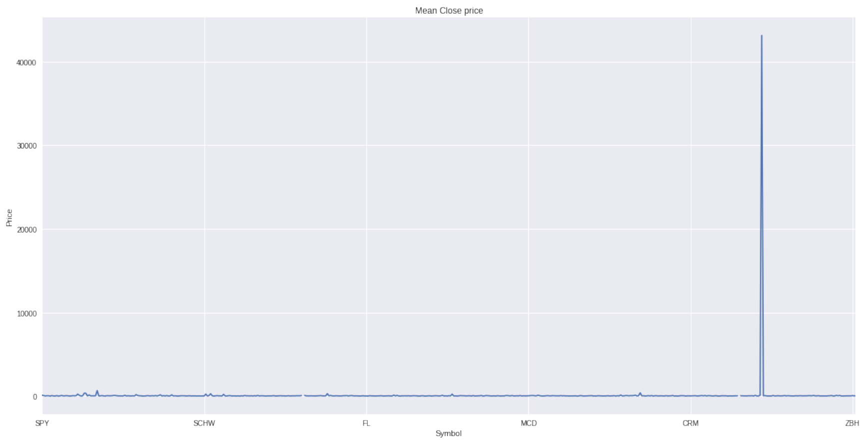


Figure 3: Mean of the Close price for all the symbols

And after filtering that symbol, some histograms for the "Mean Close price", "Standard Deviation of the Close Price" and "Maximum Close price" were created:

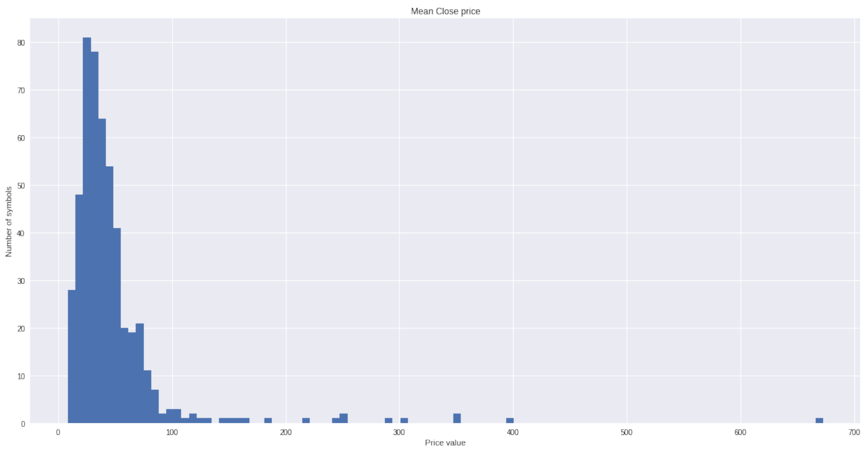


Figure 4: Histogram of the Mean of the Close price.

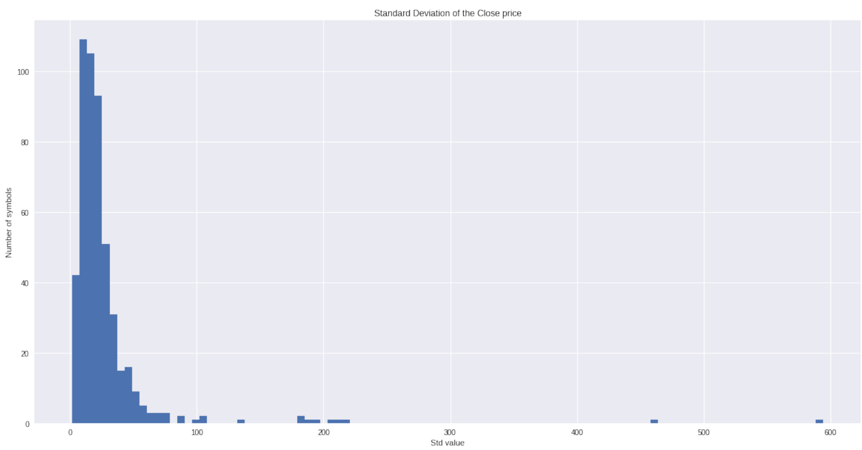


Figure 5: Histogram of the Standard Deviation of the Close Price.

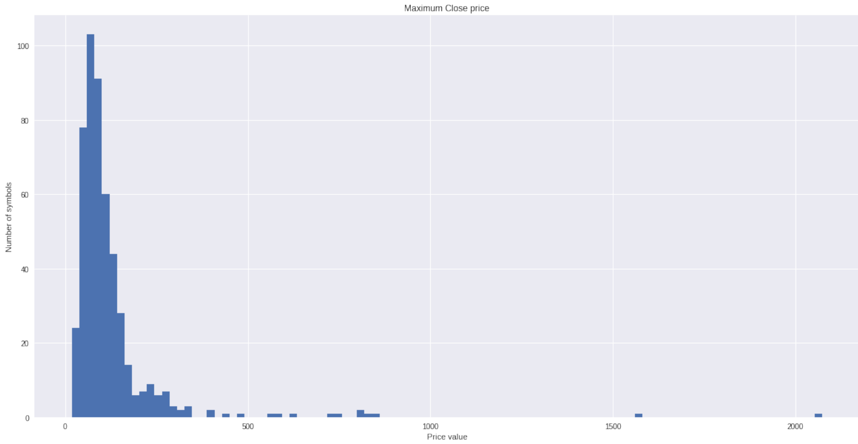


Figure 6: Histogram of the Maximum of the Close price.

The missing data distribution among the symbols can be seen below. The 1% threshold was drawn in red.

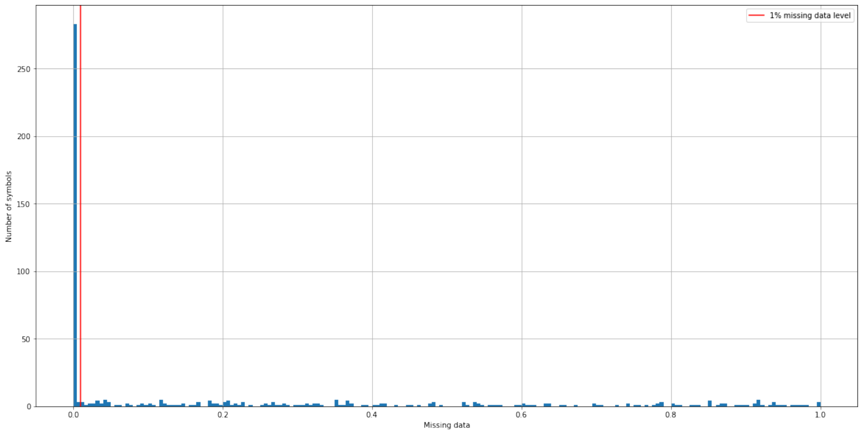


Figure 7: Missing Data

For the Autotrader, all the features for one symbol (SPY in this case) are used (in fact, only Close and Volume were used this time). The data for the Autotrader can be seen below.

feature	Close	High	Low	Open	Volume
date					
1993-01-29	43.94	43.97	43.75	0.0	1003200.0
1993-02-01	44.25	44.25	43.97	0.0	480500.0
1993-02-02	44.34	44.38	44.12	0.0	201300.0
1993-02-03	44.81	44.84	44.38	0.0	529400.0
1993-02-04	45.00	45.09	44.88	0.0	531500.0

Figure 8: Dataset used for the Atomatic Trader

The descriptive statistics for the Automatic Trader data are as follows:

feature	Close	High	Low	Open	Volume
count	6024.000000	6005.000000	6005.000000	6005.000000	6.024000e+03
mean	120.379515	121.140626	119.524448	97.583151	5.931430e+07
std	44.215695	44.437438	44.109630	69.694832	7.937370e+07
min	43.410000	43.530000	42.810000	0.000000	0.000000e+00
25%	92.847500	93.720000	91.600000	0.000000	5.380350e+06
50%	119.540000	120.290000	118.770000	115.300000	3.417315e+07
75%	141.412500	142.300000	140.570000	141.170000	7.948068e+07
max	227.760000	228.340000	227.000000	227.410000	8.141804e+08

Figure 9: Descriptive Statistics for the Automatic Trader data.

And below, a plot of the Close and Volume histograms and pairwise plots can be seen. There doesn't seem to be any interesting dependence between the features.

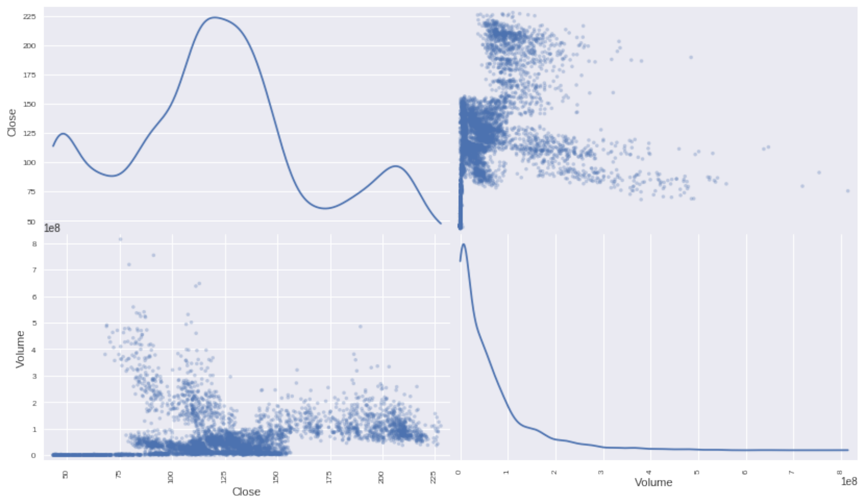


Figure 10: Histograms and pairwise plots of the Close and Volume features

Exploratory Visualization

The natural way to visualize the data is a standarized (first value = 1.0) plot of the equity price. Also, some functions were implemented to show the cases of a fixed allocation strategy (keep the amount of some equities to a fixed percentage of the total portfolio value), and the case of a dynamic strategy (custom buy or sell orders). Those visualizations also include some relevant metrics (e.g.: Sharpe Ratio, Cumulative Return, Daily Return), and a comparison with the `S&P500` index benchmark.
Some examples of the visualizations can be seen below:

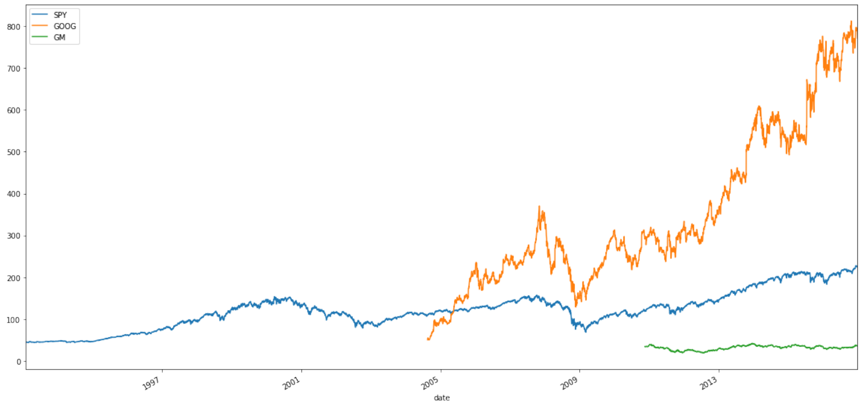


Figure 11: Graph of three symbols in the entire period.

```
sharpeRatio = 0.883609
cumRet = 0.234600
sdDReturn = 0.019109
averageDReturn = 0.001064
Final Value: 123460.000000

(Value    0.883609
 dtype: float64, Value    0.2346
 dtype: float64, Value    0.001064
 dtype: float64, Value    0.019109
 dtype: float64, Value    1.234600
 SPY      0.975587
 Name: 2011-12-20 00:00:00, dtype: float64)
```

Figure 12: Metrics and plot for the execution of dynamic orders.

Algorithms and Techniques

PREDICTOR

The algorithms considered were:

- Dummy implementation (calculates the mean of the base period)
- Linear Regressor: The most common Regressor, it tries to approximate a real-valued function with a linear combination of the features.
- KNN Regressor: It will compare the feature vector with all the other training samples, keep the ones that are closer according to a vectorial distance, and then interpolate using the target values of those close samples. It may interpolate in different ways: taking the mean of the target values, using a weighted mean (inverse distance as weight, for example), or even performing a linear regression on the k neighbors.

- Decision Tree Regressor: It defines the predicted values according to certain decision rules, that is, depending on different thresholds combinations on the features, the Decision Tree Regressor will classify a "base sequence" as belonging to one set, and predict a value based on that classification (the mean of the set, for example). To train the Decision Tree, splits of the total dataset are done, trying to minimize a cost function, in those splits.

- Random Forest Regressor: It trains many small Decision Tree Regressors, on random subsets of the dataset. On each split decision for each Decision Tree, the possible features on which to split are selected randomly. It then predicts a value equal to the mean of all the Decision Tree Regressors.

All the algorithms considered are Regressors, because the problem requires the estimation of a real-valued target. Regressors are the most commont kinds of algorithms to try to approximate a real-valued function.

Many different options with a lot of data are to be explored. Predictors will be trained for 1, 7, 14, 28 and 56 days ahead (5 different predictors). For each of those predictors different "base days" (7, 14, 28, 56, 112) and different "training days" (1 year, 2 years, 3 years) are to be used. That is 15 combinations per-predictor.
A rolling evaluation is to be implemented implemented in the entire train-val period (from 1993 until 2015), in which the predictors are to be trained for "training days" and then evaluated on the day that is "ahead days" (one day has about 300 points of data, because of the different symbols) from the last training day, and then the whole process repeated after adding "`step_eval`" days.
Above that, in the case of the Random Forest Algorithm, better hyperparameters will be searched for. For all that reasons, the training/validation process is slow, so parallelization will be implmeneted, using the multiprocessing library in Python.

AUTOMATIC TRADER

The algorithms chosen for the Automatic Trader were:

- Simple Q-learning: It is a model-free algorithm, which means that it doesn't try to predict the next states, or the next rewards by creating a model of the Environment. It tries to estimate the expected reward for every state/action combination (which it stores in a matrix, called "Q"), from the present "until infinity". With that estimation, on each state that is presented to the algorithm, it chooses the action that maximizes the expected reward (and updates the Q matrix when it receives new rewards).

- Dyna-Q: Noting that the Q matrix updates very slowly, or very costly, because it requires a new real-world datum for each entry update (and those matrices can be very big), the dyna improvement is created. This algorithm is no longer totally model-free, as it tries to estimate a model of the environment (predict next state and next reward), and it uses it to update the Q matrix many times between the "real-world data updates". When a new data arrives, the Q matrix is updated as in simple Q-learning, but then new simulated or "hallucinated" data is created and the Q matrix is updated with that data, as if it were real (without changing the learning parameters). The "hallucinated" data is created with a model that, in its most basic form consists of predicting for the next state the one that has been seen more frequently given the current one and the action taken, and an exponentially weighted mean of the historic values of reward for the current state and action.

- Custom Dyna-Q modification: On this modification, the way the "hallucinated" data is produced in the Dyna part of the algorithm is changed. Instead of the procedure described above, the Predictor from the first part is used to estimate the next state and reward. A virtual Environment is needed to provide those estimates in a way that the Agent can understand them (as state/reward pairs, instead of Close/Volume predictions).

All the algorithms are forms of Reinforcement Learning, which is fit for the problem of learning how to optimize a sequential behaviour in a seemingly random environment.

The dataset is divided, in the time variable, in a training set and a test set. The agents are trained on the training set many times (epochs), and then tested in the test set (with or without "online" learning).

Some technical indicators were implemented, and the states of the agents correspond to combinations of those indicators (previously quantized). Also, the possible actions of the agents correspond to different allocations of one equity as a fraction the total portfolio value (the allowed fractions are configurable).

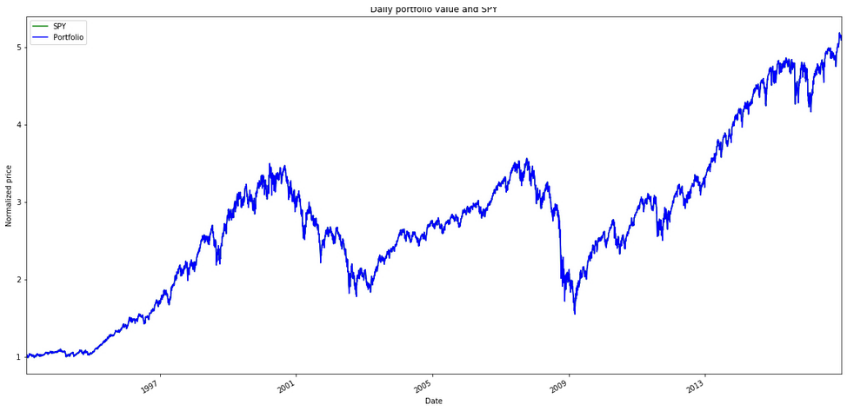
Benchmark

PREDICTOR

More than a benchmark, there was one objective for the predictor: The Mean Absolute (relative) Error should be less than 5% for the "7 days ahead" prediction.

AUTOMATIC TRADER

For the Automatic Trader the chosen benchmark was the $S\&P500$ index. Below there is a plot, generated with an implemented function (described in the "Exploratory Visualization" section), that shows the values of the benchmark in the entire period.




```
Start Date: 1993-01-22 00:00:00
End Date: 2016-12-31 00:00:00
Symbols: ['SPY']
Allocations [1.0]
Sharpe Ratio: 0.457087056488317
Volatility (stdev of daily returns): 0.011797115215508
Average Daily Return: 0.000339683517391
Cumulative Return: 4.087164314974967
```

Figure 13: The SPY benchmark.

3 Methodology

Data Preprocessing

PREDICTOR

The datasets were generated and saved (in pickle format) in the "`n00_datasets_generation`" notebook. The process is as follows:

- Drop the stock symbols that have more than 1% missing data in the full period.
- Generate the training/validation samples: samples of "base days" days and a "label" that is "ahead days" ahead, every "step" days (overlapping is allowed in the training set; the validation labels are "out of sample" so the trainer never sees the next ones).
- Drop the samples that have more than 10% missing data.
- Fill the remaining missing data, per-sample, with the last known value first ("forward" filling), and the remaining ones with "backward" filling. That is to try to keep causality as much as possible.
- Save all the datasets with the name `x_base{base_days}_ahead{ahead_days}.pkl`, `y_base{base_days}_ahead{ahead_days}.pkl`.

AUTOMATIC TRADER

The Automatic Trader works with only one symbol, and the symbol chosen was "SPY" because it tracks the benchmark that was defined. The preprocessing is done with partial chunks of the data, when the technical indicators are calculated. The process is as follows:

- When the indicators are created, a StandardScaler is fit with the data (this may violate causality, but it is only done with the training data, so it is not "breaking the rules"; a version that doesn't violate causality may perform better in the test set).

- When the (quantized) values of the indicators are asked for, the Indicator object fills the missing data (first "forward" then "backwards"), and scales the resulting value. The values that are passed to the Indicator objects are (Open, High, Low, Close, Volume), for one symbol, from the starting date to the current date.

Implementation

First, the function "`download_capstone_data`", in the `data_sources` package, takes care of downloading all the necessary data for both parts of the project.

PREDICTOR

The predictors in itself are very simple: they're just wrapers around the scikit-learn classes that preserve the Dataframe information (scikit-learn classes work with numpy arrays, generally). All the predictor implementations work on the datasets saved from the preprocessing stage. The most interesting part of the Predictor implementation is the training and evaluation infrastructure. There are two packages that take care of the training and evaluation (other than the notebooks):

- evaluation.py: Contains the actual training/evaluation functions.
  - `roll_evaluate`: This function takes a dataset of samples and labels, and roll-evaluates a given predictor in the full period, moving the training period by "step" days each time. The training time, step between train/evaluation periods, and the days ahead for the prediction are all configurable. It uses `run_single_val` to run each train/evaluation. It is to be noted that the predictions are matrices, because they are made in different time points for different symbols. In fact, this function already calculates the per-date mean and standard deviation for the metrics of the predictions in the training set (in the symbols dimension). The validation results are returned as raw matrices, to be processed later.
  - `run_single_val`: Runs a single training and validation of a given predictor.
  - `get_metrics`: Given the predictions and labels matrices, it returns the metrics for each symbol, calculated in the "time" dimension.
  - `get_metrics_in_time`: Given the predictions and labels matrices, it returns the metrics for each date, calculated in the "symbols" dimension.
- misc.py: Contains some functions created to parallelize the training/evaluation process using the multiprocessing library of Python. The process won't be explained here in detail, as it just a technical issue to make the computations go faster.

The training / validation process can be seen below. That process was repeated for different "base days", and different "training days", to find the best combinations, for each "ahead days".

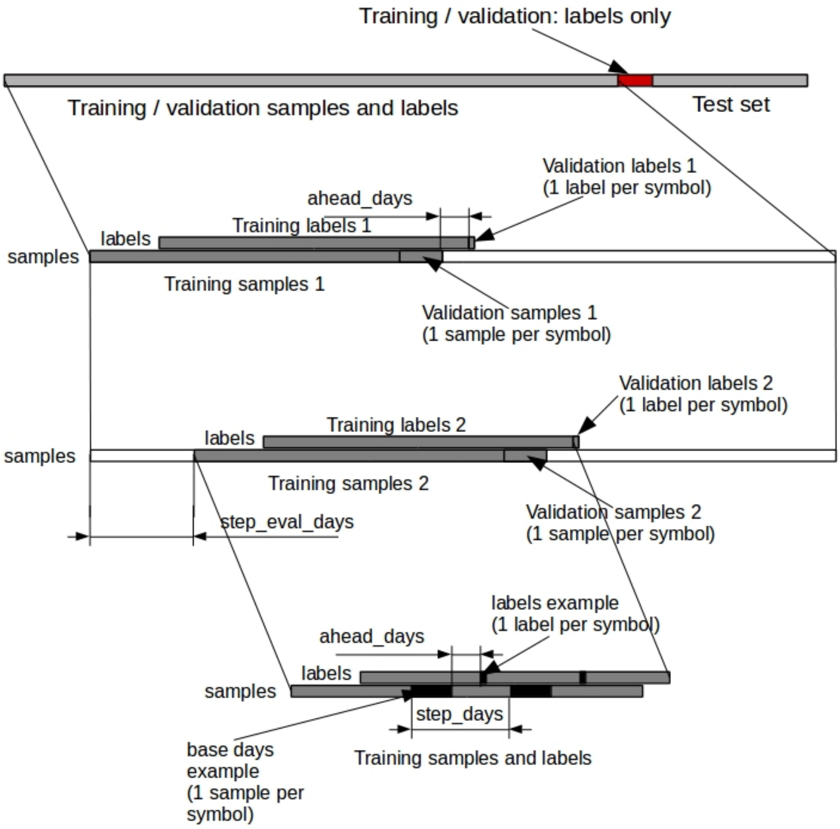


Figure 14: The training / validation process, for one predictor and one set of parameters.

AUTOMATIC TRADER

The Automatic Trader part of the project was designed as an object oriented project. The implemented objects were:

- Agent: An abstract Q-learner or dyna-Q learner. It only sees a number of states (codified as integers), and a number of possible actions. Then, it learns the best possible actions using reinforcement learning. In its "regular" versions it doesn't know anything else about the environment, that's why it was tested (initially) with a very different problem (from Tucker Balch's Machine Learning for Trading course): to solve a maze. The version that includes the Close and Volume predictor has more information about the actual environment, and is not so general (couldn't be used without changes in other different problems). Externally, the states correspond to combinations of the possible (discrete) values of some indicators, and the actions correspond to possible fractions of the total value of the portfolio, that the agent "recommends" to hold in a particular day.

- Environment: This class directly interacts with the Agent. It works as a translator between the "real world" and the "states/actions" of the Agent. It contains one Portfolio that keeps track of the current positions and its values, as well as the current date, and some Indicators (three, in the versions tested). It also keeps a version of the market data (another version is stored by the Portfolio). When the Agent asks for an action to be taken (an integer), the Environment understands that a target fraction of the entire portfolio value is "desired" to be held in one asset (the autotrader works only with one asset [SPY, in the experiments conducted], and only with positive positions; doesn't allow to sell short). It then calculates the closest amount of shares of that asset that it is possible to buy, considers the current positions held, and sends the necessary orders to the Portfolio object, to achieve that. After the Portfolio executes the orders and updates the current date, the Environment asks the Indicators to calculate their new values, transforms the Indicator's values vector to a single integer state, and returns that state to the Agent. The reward for the agent is the increase in the total value of the portfolio, on that day.

- Portfolio: A class that keeps track of the owned assets. It has a copy of the market data. It updates the current date, excecutes orders, and calculates the values of the positions held.

- Order: It's just a Pandas Series with some predefined structure. It has three entries: SYMBOL, ORDER, SHARES. SYMBOL corresponds to the ticker of the asset, ORDER is either BUY, SELL, or NOTHING, and SHARES is an amount of shares. The three types of orders where implemented when a simpler set of actions was assumed (fixed amount, variable order type). In the end, as the amounts to buy or sell are calculated from the "desired fraction of the total value", which is what the Agent directly decides, the BUY order was enough (zero and negative values are accepted).

- Indicator: This class, given some market data, calculates a particular indicator, then it scales it, and finally it quantizes it. The "indicator function" to apply, as well as the intervals for quantization have to be passed at the creation of the Indicator. Also the total market data (the training set only) has to be passed at creation, to fit the scaler (a scikit-learn StandardScaler). That fitting of the scaler is the main reason for the existence of the "`set_test_data`" function in the Environment class: that resets the data in the Environment's Indicators but doesn't fit the scaler again. The Indicator class has functions to transform a real value into a quantized value, a real value into the quantized interval it belongs to, and an interval index to a quantized value that is close (the mean of the interval's edges).

- Quantizer: This class is a smaller version of the Indicator (the design may have been better if an Indicator contained a Quantizer but because of the way the code was developed there are two separate classes). It is used to get the quantized possible fractions of the total value of the portfolio, that are the possible actions that the Agent can choose. When returning a quantized value, it returns one of the edges of the interval, instead of the mean, as the Indicator would. That was important to be coherent with the math in the Environment class (when calculating the actual amount of shares to buy/sell).

Finally, the functions in "simulator.py" take care of initializing everything, and "`simulate_period`" in particular, makes the Agent and the Environment interact through a full dataset of market information (one "Epoch").

One problem that was found while implementing the Indicators was that they were more easily, and efficiently, implemented in a vectorial, rolling way. One option was to pre-calculate all the values and then use them, but the whole Agent-Environment system was designed to work in a per-date basis. That is, the simulation was designed to be

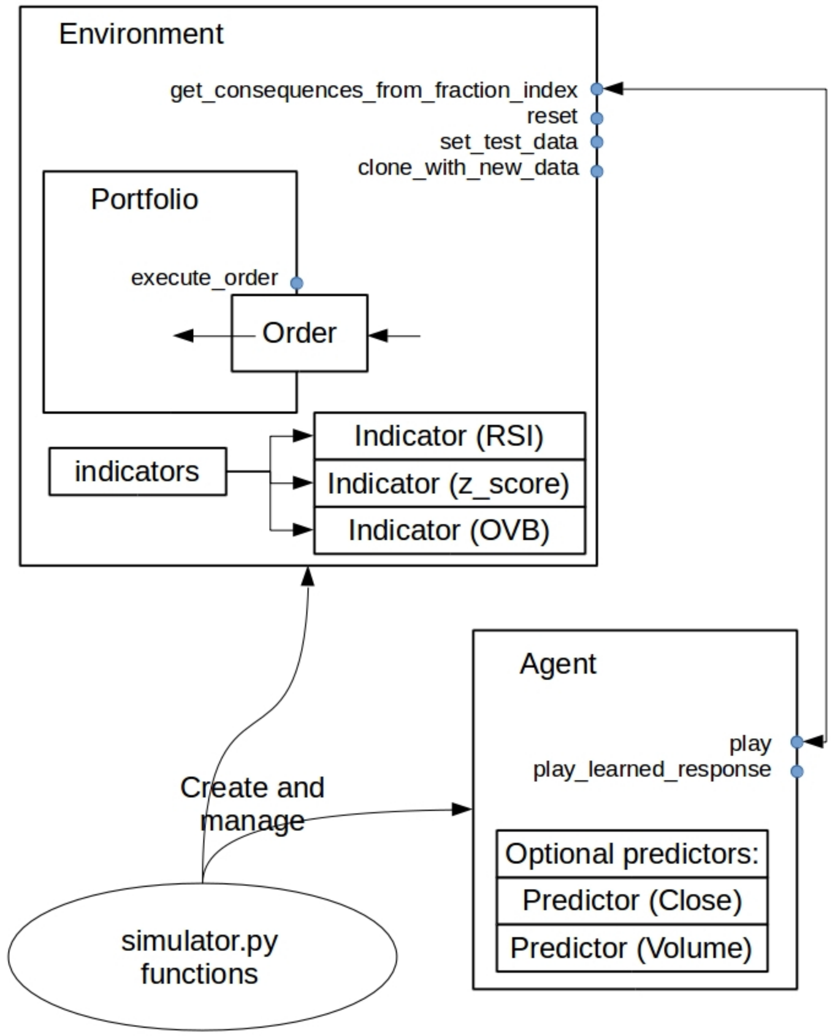


Figure 15: Design of the Automatic Trader

"realistic", sending one datum per date. That was a good way to prevent violations of causality. On the other hand performance suffers a lot from that design. The decision was to favor the "clean" design instead of the performance gains, for the moment. One option to consider, would be to ask the Indicators to internally pre-compute values, but not changing the current interface to the Environment. That may, in the future, provide better performance, and still confine the risks of violations of causality only to the indicator's calculations.

Another problem that was identified was that the Indicators require scaling for the training to work well. To do so, the Indicators are fit with the full training dataset when the Environment is created. That is a (hopefully minor) violation of causality. A "causality respectful" scaling could be implemented, but would decrease performance and its benefits may not be great. In any case, the Scaler shall never be fit with the test data. To be able to use the training-fitted scalers with the test set, a function that changes the underlying data of the Environment, without fitting or doing other initialization tasks, was implemented in the Environment class (the first option, that was creating a new Environment for the Test set was not possible). Also, the starting date of the simulation cannot be the first day of data, because some indicators, and the predictors have functions that are not well defined if some days of previous data is not fed to them. That's why the function to set the test set, as well as others, allow to define the starting day of the simulation.

Finally, "inserting" the Predictor into the Agent, also had it's problems. The main issue is that the Agent, in its original version, works only with abstract states and actions, that are just integers, and rewards that are real numbers. It doesn't know anything about the real problem it is trying to solve in itself. The Environment class takes care of the translation between "real world" concepts and "abstract" states/actions/rewards from the Agent. As a solution, in the dyna "hallucination" process, a temporary Environment class is created, to hold only the necessary data for the "hallucinations". A function called `clone_with_new_data` was implemented in the Environment class, to return an Environment with the same Indicators and initialization than the real one, but containing only the data that was seen, until now, by the Agent. One extra problem was that only a "Close" Predictor was implemented in the first part of the project, but some of the used indicators require the "Volume" information, as well. To solve that a predictor with the same parameters as the "Close" predictor was trained with the "Volume" data. It is clearly a much worse predictor than the "Close" one, but in any case, the Volume is only used to calculate the new states (the rewards are calculated with the "Close"value), so even if the Agent moves to states that are not correct, so long as the rewards are well approximated, it should learn correctly, even if slower (it wouldn't focus so much on the really "important" states).

Refinement

PREDICTOR

Due to the high computational cost of the training / validation model, only two kinds of predictors were evaluated for all the "ahead days": Linear Regression and Random Forest.

For the Random Forest predictor some hyperparameter tuning was made. Before the tuning, the Linear Regressor was performing better in all the "ahead days" cases, as can be seen below (table with the best models before hyperparameter tuning):

	model	r2	mre	train_days	base_days
ahead_days					
1.0	linear	0.986599	0.015856	504.0	112.0
7.0	linear	0.923348	0.042367	756.0	112.0
14.0	linear	0.865259	0.060167	756.0	112.0
28.0	linear	0.758046	0.091966	756.0	112.0
56.0	linear	0.590426	0.127913	756.0	112.0

Figure 16: Best models before hyperparameter search

And the best Random Forest predictors, before hyperparameter search:
The parameters that were searched for, were:

- `n_estimators`: 50, 100
- `max_depth`: 5, 10

	r2	mre
ahead_days		
1.0	0.984864	0.018002
7.0	0.915048	0.044267
14.0	0.829452	0.063327
28.0	0.715802	0.096087
56.0	0.512861	0.136095

Figure 17: Initial results for the Random Forest predictor, before hyperparameter search

To do so, it was necessary to use a performance optimized instance from AWS, and even that way, the hyperparameter search took some hours. That's why no more parameters were searched for (other parts of the project were prioritized at this point). After the hyperparameter search, the Random Forest predictor was better than the Linear Regressor for most of the "ahead days" cases. The further the predicted day, the greater the improvement. Even so, as the Linear Regressor was faster and simpler, and the differences in the R^2 score were minimal (except in the case of 56 days ahead), the Linear Regressor was chosen as the best predictor. The results of the evaluation, after the hyperparameters search, can be seen below.

	linear	random_forest	diff	best
ahead_days				
1.0	0.986599	0.986126	-0.000474	linear
7.0	0.923348	0.929017	0.005669	random_forest
14.0	0.865259	0.868927	0.003668	random_forest
28.0	0.758046	0.765112	0.007066	random_forest
56.0	0.590426	0.615412	0.024986	random_forest

Figure 18: R^2 metrics and best models after hyperparameters search

AUTOMATIC TRADER

After implementing simple Q-learning, and dyna-Q, the Predictor from the previous section was added to the dyna process. Before explaining that, another improvement over simple Q-learning was added in all the versions. That was an idea that came to me when training the Q-learner with the labrinth scenario. To help the Agent learn faster a QExplore matrix is updated at each "move" that records the amount of times that a state/action combination was performed. That way, when the Agent performs a "random" action, it doesn't do it using a uniform distribution, but weights the values in the QExplore matrix, so that the unexplored actions in the current state are more likely to be taken. Resuming: Agent's random actions try to find new state/action combinations to learn faster.

Now, the Predictor 'improvement': the idea is that in the dyna iterations, the Q matrix is updated according to simulated iterations. In the basic dyna-Q Agent, the simulations were based on a simple model to predict the probability of new states, and rewards. With the improvement, the 'Close' and 'Volume' values for the next days are predicted with the Predictor, and that is fed to a “virtual” Environment that returns the new simulated states and rewards. As some indicators used the 'Volume' data, it was necessary to train a 'Volume' predictor for this part. That predictor is far more inaccurate than the 'Close' predictor. To train it, without going through the whole tuning process again, the 'base days' and 'training days' were taken the same as in the 'Close' predictor. All the predictors are trained for 'ahead days' = 1. Then, the predictions for further days are taken using the previously predicted day, in a recursive fashion. As a drawback, the addition of the Predictor to the Agent makes the Agent more “environment aware” (now it works with some specific knowledge of the environment, to be able to make the predictions).

4 Results

PREDICTOR

The final Predictor results can be seen below. The main metric for model selection was the R^2 score. There was a target of at least 5% of mean relative error in the “7 days ahead” prediction, that was accomplished: The MRE in the “7 days ahead” prediction is 3.5% in the test set.

	train_r2	test_r2	train_mre	test_mre
ahead_days				
1.0	0.983486	0.976241	0.008762	0.013906
7.0	0.906177	0.874892	0.026232	0.034764
14.0	0.826779	0.758697	0.037349	0.051755
28.0	0.696077	0.515802	0.052396	0.078545
56.0	0.494079	0.152134	0.073589	0.108190

Figure 19: Final results for the Predictor.

AUTOMATIC TRADER

Many different Agents were trained and tested. Below, a list of all of them with their parameter sets can be seen. The parameters are:

- dyna: The number of dyna-Q “halluciantion” iterations.
- states: The number of states. Directly related to the quatization of the technical indicators (more states = more fine grained quantization).

- actions: The number of possible actions for the Agent to take. Two actions means it can hold 0% or 100% of the equity, more actions means more possible fractions between 0% and 100%.
- `training_days`: The number of real market days used for training.
- epochs: The number of times that the training set is seen by the Agent.
- predictor: 0 if no predictor was used with the dyna iterations, 1 if the predictor was used.
- `random_decrease`: The base for the exponential decrease in the random exploration.

nb_name	dyna	states	actions	training_days	epochs	predictor	random_decrease
simple_q_learner	0	125	2	512	15	0	0.9999
simple_q_learner_1000_states	0	1000	2	512	15	0	0.9999
simple_q_learner_1000_states_4_actions_full_training	0	1000	4	5268	7	0	0.9999
simple_q_learner_1000_states_full_training	0	1000	2	5268	7	0	0.9999
simple_q_learner_100_epochs	0	125	2	512	100	0	0.9999
simple_q_learner_11_actions	0	125	11	512	10	0	0.9999
simple_q_learner_fast_learner	0	125	2	512	4	0	0.999
simple_q_learner_fast_learner_1000_states	0	1000	2	512	4	0	0.999
simple_q_learner_fast_learner_11_actions	0	125	11	512	4	0	0.999
simple_q_learner_fast_learner_3_actions	0	125	3	512	4	0	0.999
simple_q_learner_fast_learner_full_training	0	125	2	5268	4	0	0.999
simple_q_learner_full_training	0	125	2	5268	15	0	0.9999
dyna_q_1000_states_full_training	20	1000	2	5268	7	0	0.9999
dyna_q_learner	20	125	2	512	4	0	0.9999
dyna_q_with_predictor	20	125	2	512	4	1	0.9999
dyna_q_with_predictor_full_training	20	125	2	5268	4	1	0.9999
dyna_q_with_predictor_full_training_dyna1	1	125	2	5268	4	1	0.9999

Figure 20: Sets of parameters for all the Agents that were tested.

For all those Agents, the training results and test results were recorded. In the case of the test results, experiments with and without the Q-learning activated were run. The experiment without learning was run first (so that the Q matrix wasn't changed), and then the other. The results vary, being better one or the other in different Agents. The benchmark metrics are reported, as well as the increase between the Agent's metrics and the benchmark's. The detailed results can be seen on appendix A. The increase of the Sharpe Ratio in training, test without learning, and test with learning activated, are resumed in the table below:

	sharpe_i_train	sharpe_i_test_no_learn	sharpe_i_test_learn
nb_name			
dyna_q_1000_states_full_training	1.839774	-0.015916	0.045924
dyna_q_learner	2.104744	0.076456	-0.372623
dyna_q_with_predictor	1.686795	0.450512	0.402365
dyna_q_with_predictor_full_training	0.547037	-0.628075	-1.146213
dyna_q_with_predictor_full_training_dyna1	0.025605	-0.174313	-0.533559
simple_q_full_training	0.802568	-0.699006	-0.128732
simple_q_learner	0.384157	-0.076295	0.530880
simple_q_learner_1000_states	1.845339	-0.456463	-0.529460
simple_q_learner_1000_states_4_actions_full_training	1.786332	0.497334	0.666898
simple_q_learner_1000_states_full_training	1.834462	0.871204	0.016090
simple_q_learner_100_epochs	2.491662	0.199287	-0.349969
simple_q_learner_11_actions	-0.057651	-0.286551	0.026430
simple_q_learner_fast_learner	1.277035	0.521636	0.159303
simple_q_learner_fast_learner_1000_states	0.429755	0.380086	-0.266534
simple_q_learner_fast_learner_11_actions	1.639747	0.381111	0.518118
simple_q_learner_fast_learner_3_actions	1.343115	0.190571	-0.117275
simple_q_learner_fast_learner_full_training	0.587776	0.817865	0.817865

Figure 21: Increases of Sharpe Ratio over Benchmark's Sharpe Ratio for all the Agents.

It can be seen that some of the Agents outperform the benchmark (the ones that have an increase greater than 0, in the test set). Finally, the Agent with the best Sharpe Ratio increase in the test set, without learning, was selected as the "best" Agent, and its detailed characteristics are below.

	dyna	states	actions	training_days	epochs	predictor	random_decrease
simple_q_learner_1000_states_full_training	0.0	1000.0	2.0	5268.0	7.0	0.0	0.9999

Figure 22: Parameters for the selected Agent.

	cum_ret	epoch_time	sharpe	cum_ret_bench	sharpe_bench	sharpe_quotient	cum_ret_quotient
training	79.618002	159.316511	2.366638	3.304503	0.456677	5.182302	24.093793
test_no_learn	0.221239	9.844955	1.482707	0.107023	0.442715	3.349119	2.067218
test_learn	0.206955	9.233611	1.517675	0.107023	0.442715	3.428106	1.933751

Figure 23: Results for the selected Agent.

The resulting standarized (to the first-day value) returns of the Agent and the SPY benchmark, in the test set (without learning), can be seen below.

Figure 24: Test results for the best Agent.

The model seems to be doing well in the test set, but to use it as a real trading system many more things should be considered. Among them, the trading fees and slippage, but it should also be tested with more data. Also, some sensitivity analysis was done, to try to determine how consistent were the results. Four sensitivity tests were performed:

Training sensitivity test: The Agent was trained and tested 20 times (for the SPY symbol), with the exact same training and testing data. The results were recorded and analyzed. In particular, the Sharpe Ratio increases, with respect to the benchmark's, had the following statistical results.

	sharpe_delta
count	20.000000
mean	0.192517
std	0.478996
min	-0.563718
25%	-0.118045
50%	0.180377
75%	0.406261
max	1.464891

Figure 25: Descriptive statistics for the Sharpe Ratio increase.

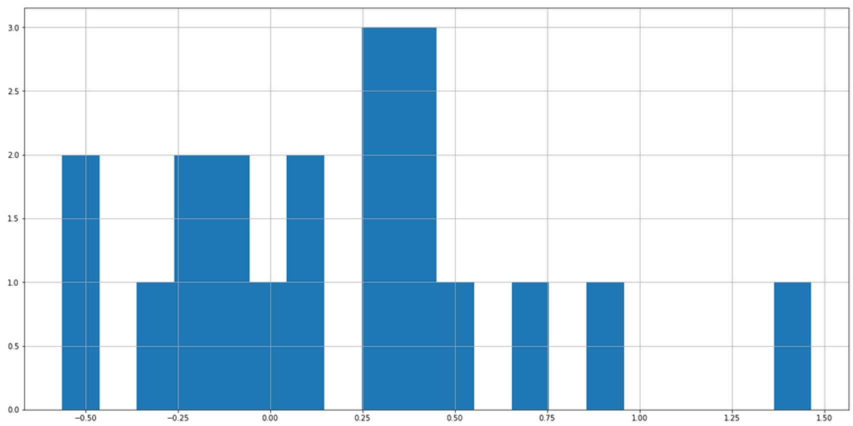


Figure 26: Histogram for the Sharpe Ratio increase.

It can be seen that there is a lot of variability with the random effects of training (it's not just initialization; the Agent takes random actions). That can be due to not having totally converged (not so likely, as other tests with more epochs showed variability also [2 or 3 were conducted]), or due to "locally optimal solutions". The mean result is still positive (0.19), but with a standard deviation of 0.48, it could be considered "zero".

Time datasets sensitivity test: The Agent was trained and tested 20 times (for the SPY symbol). The training set was increased from about 4000 days to about 5500. The test set was always taken as the immediately next 504 days of the training set. That was used to measure the sensitivity to changes in the testing peridod. Additionally, the amount of data in the training period was changed a bit. The Sharpe Ratio increases, with respect to the benchmark's, had the following statistical results.

	sharpe_delta
count	20.000000
mean	-0.403402
std	0.460441
min	-1.146399
25%	-0.742050
50%	-0.357618
75%	-0.012568
max	0.215397

Figure 27: Descriptive statistics for the Sharpe Ratio increase.

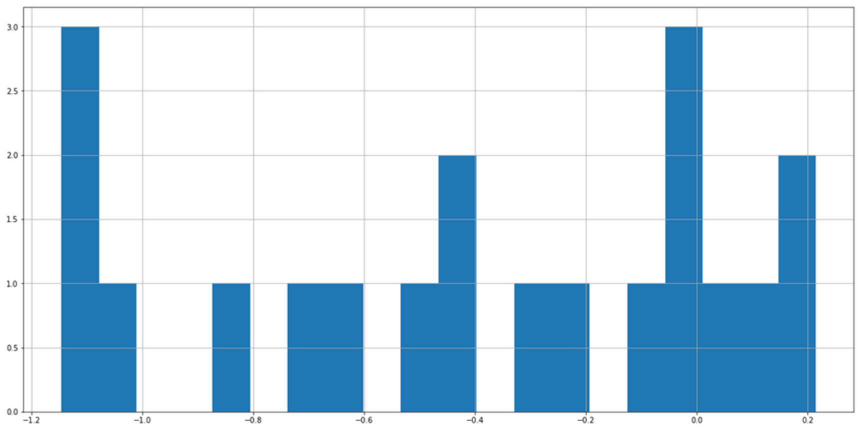


Figure 28: Histogram for the Sharpe Ratio increase.

Again, the results show great variability, this time with a negative mean of 0.40, but again “zero” within the error range ($std = 0.46$).

Other symbols without retraining sensitivity test: The Agent was tested without retraining for 100 different symbols, in the same test period, as originally used. The Sharpe Ratio increases, with respect to the benchmark's, had the following statistical results.

	sharpe_delta
count	100.000000
mean	-0.118791
std	0.759289
min	-1.797214
25%	-0.666129
50%	-0.074547
75%	0.400861
max	1.688194

Figure 29: Descriptive statistics for the Sharpe Ratio increase.

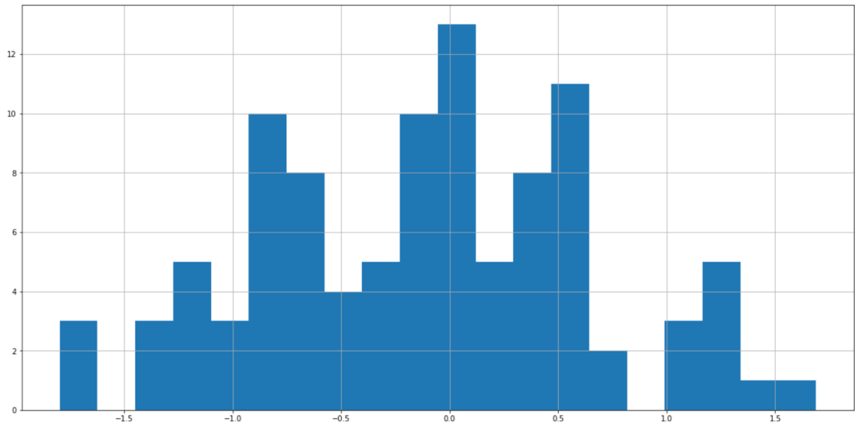


Figure 30: Histogram for the Sharpe Ratio increase.

Again, a lot of variability, and a "zero" mean increase in the Sharpe Ratio, to within the error range (taken as one standard deviation).

Other symbols with retraining sensitivity test: An Agent with the same hyperparameters was tested for 20 different symbols [1], in the same test period, as originally used. This time, the Agent was trained for the specific symbol before testing it. The idea of this test is to see how general the hyperparameters choice is, independent of the trained parameters. The Sharpe Ratio increases, with respect to the benchmark's, had the following statistical results.

	sharpe_delta
count	20.000000
mean	-0.049477
std	0.519187
min	-0.902369
25%	-0.422890
50%	-0.076248
75%	0.251227
max	1.163407

Figure 31: Descriptive statistics for the Sharpe Ratio increase.

[1] when training is involved the amount is very limited by the computational power I had available, and the lack of parallellization in the implementation

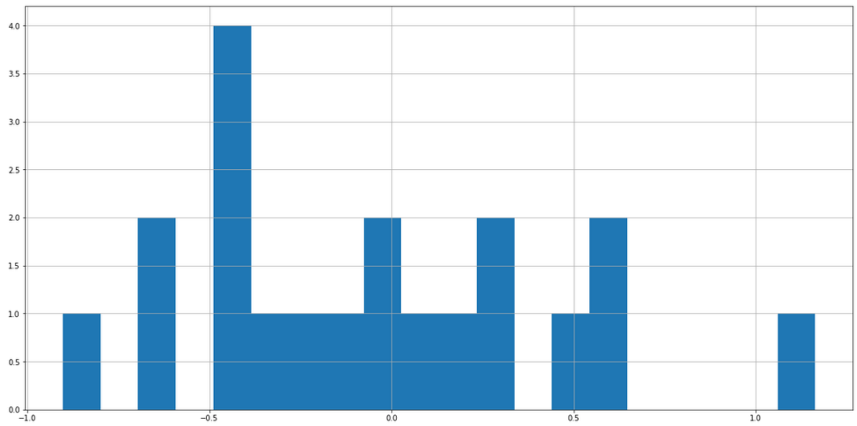


Figure 32: Histogram for the Sharpe Ratio increase.

Again, a lot of variability, and a "zero" mean increase in the Sharpe Ratio, to within the error range (taken as one standard deviation).

Resuming, the sensitivity analysis shows that there doesn't seem to be a consistent increase in the Sharpe Ratio with respect to the benchmark. The Efficient Markets hypothesis seems to be holding (for now).

5 Conclusion

Free-Form Visualization

PREDICTOR

In the graph of figure 33 a gap can be seen between the training and test sets. That is because no predictions for the test set are made with data from the training set (the gap is "`base_days`" wide, that is 112 market days, in this case).

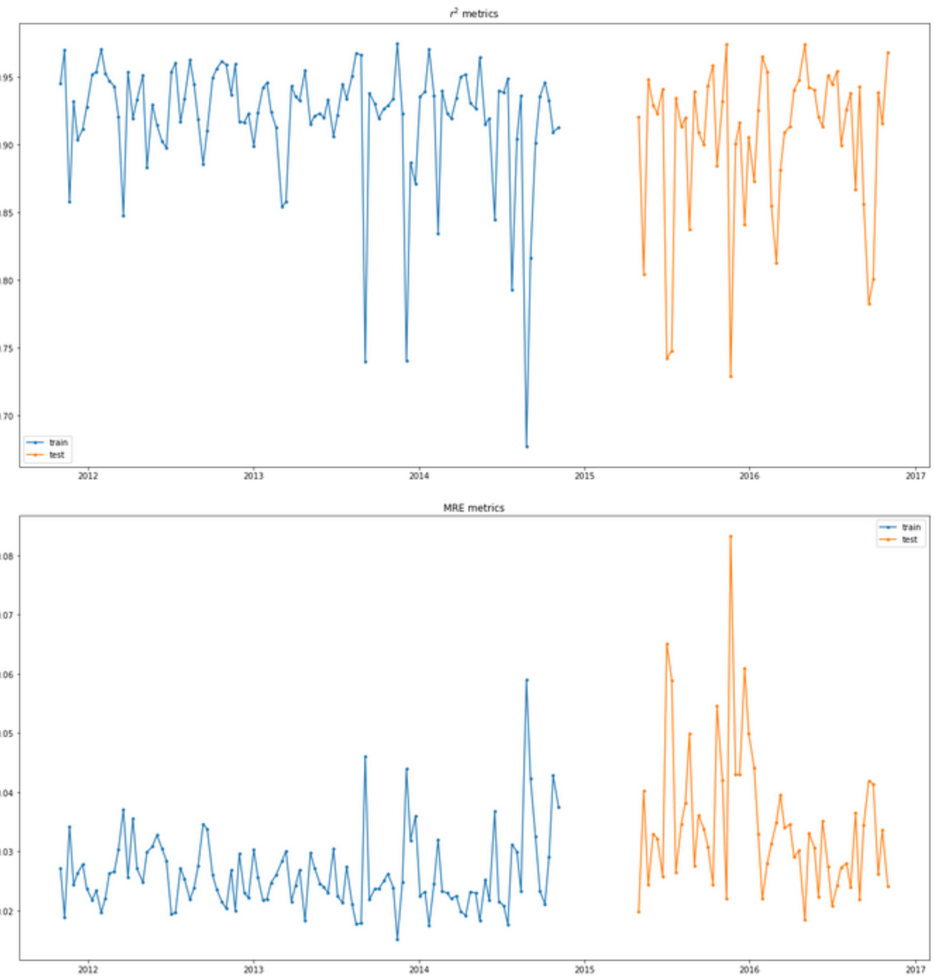


Figure 33: Final results for the Predictor for 7 days ahead, in time.

Initially, the predictors were trained for the entire period, but that was reduced, because it took too long. In one of the initial experiments, the training mean relative error from image 34 was seen (that behaviour was repeated in all the cases tested with the full period).

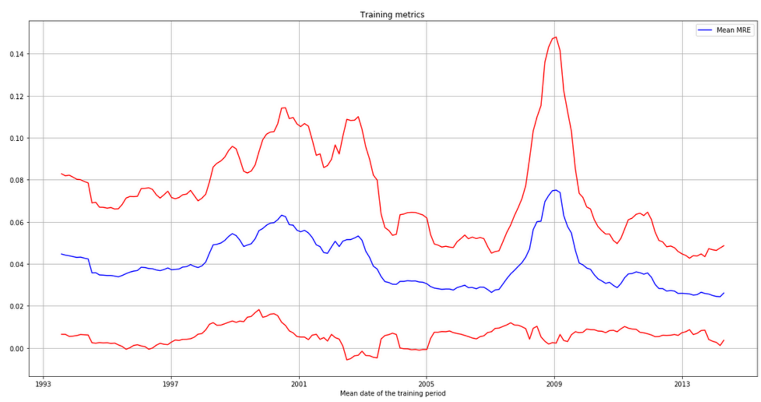


Figure 34: MRE metrics for the training in a 7 days ahead prediction, in time. The red line corresponds to the 2σ deviation of the MRE.

It can be seen that the 2008-2009 crisis has a high increase in the MRE of the predictor.

AUTOMATIC TRADER

The training results for the Automatic Trader had some interesting features. As expected, the trader performs much better than the benchmark (but it has "future" information, as it is in the training set), but interestingly it seems to be making most of its gains after the 2008-2009 crisis, as if it had found a pattern that is only valid in that period (see figure 35).

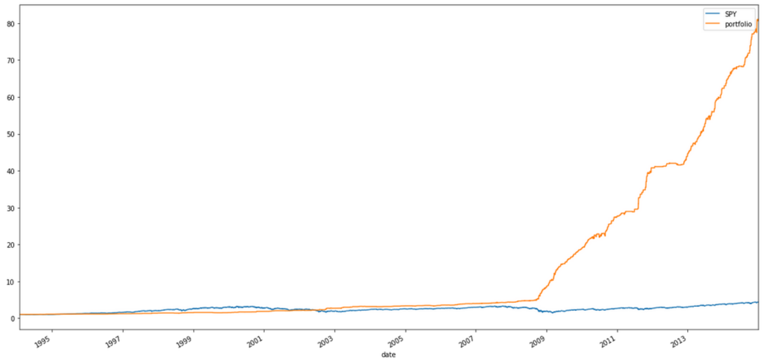


Figure 35: The training standarized cumulative return for the autotrader.

Reflection

A Closing Price predictor and an Automatic Trader were implemented.

The predictor was trained to predict 1, 7, 14, 28, an 56 days ahead. For the training, samples from all the "filtered" stocks from the $S\&P500$ list were used, as if they were from the same source. The resulting metrics are all coherent (more days ahead imply worse R^2 values, and greater MRE). The MRE objective of an MRE of less than 5% in the 7 days ahead prediction was achieved. Despite that, the metric is just the mean for all the symbols, and the full training period. It may have better results for some symbols or periods, and worse for others. The hypothesis of taking all the stock symbols as the same source of data may be too broad: training on a subset after some clustering, on one symbol alone, or on correlated symbols, may improve local results. The two most difficult things to note from this part were the need of carefully managing the time periods so that causality is preserved as much as possible, and the need of a lot of computational power to test many parameter sets.

The Automatic Trader was trained to buy and sell one symbol (one equity) to try to maximize the profit. The reward function was just the added total value of the portfolio, each day. Nevertheless, the Sharpe Ratio was increased with respect to the benchmark for some Agents, but the sensitivity analysis revealed that, in mean, that was no the case. The predictor from the previous part was added, but the results were worse than with the simple Q-learning. That may be due to the fact that the Volume predictor was really very badly tuned (as it was only added for that part there was no time to properly search for parameters). The time to train, with the predictor, was a lot more. One problem with this part of the project was that it was not possible to paralelize the execution of the Q-learner (it would require sharing the Q matrix between threads or processes, and the implementation would take too much development time).

Improvement

PREDICTOR

- Training to predict only one symbol could yield better results. Data from other symbols could be used differently.
- Using ARMA, ARIMA or LSTM, could improve the results.
- It would be interesting to train for other technical indicators, rather than just "Close" value. That would be more compatible with the Q-learner from the other part of the project. It can be achieved by changing the "Blob" function, in the samples creation (that was intentionally left configurable by changing that function).

AUTOMATIC TRADER

- Using some Data Augmentation may help in the training stage, and also measure the sensitivity of the solution to changes (a Monte Carlo simulation, for example). On the other hand, there may not be any "safe" and simple way to simulate new data (it may be very different from real data).
- Fees and slippage could be taken into account by the model, to make it more realistic.
- More indicators could be added and tested. The importance of the indicators in the selling and buying decisions could be assessed.
- Other Reinforcement Learning techniques could be tested (among others, Deep Reinforcement Learning, maybe).

References

[1] Yuning Zhang Yuqing Dai. Machine learning in stock price trend forecasting. `http://cs229.stanford.edu/proj2013/DaiZhang-MachineLearningInStockPriceTrendForecasting.pdf`.

[2] Saahil Madge. Predicting stock price direction using support vector machines. `https://www.cs.princeton.edu/sites/default/files/uploads/saahil_madge.pdf`.

[3] Tucker Balch. Machine learning for trading assignments. `http://quantsoftware.gatech.edu/CS7646_Fall_2016#Assignments`.

A Detailed results for the Automatic Trader Agents

nb_name	cum_ret	epoch_time	sharpe	cum_ret_bench	sharpe_bench	sharpe_increase	cum_ret_increase
dyna_q_1000_states_full_training	94.756962	242.8824055195	2.2964510955	3.3045026178	0.4566770028	1.8397740927	91.4524593822
dyna_q_learner	0.493825	18.8718209267	3.7064355887	0.4244923418	1.6016915494	2.1047440393	0.0693326582
dyna_q_with_predictor	0.539799	458.8401937485	3.288486721	0.4244923418	1.6016915494	1.6867951716	0.1153066582
dyna_q_with_predictor_full_training	2.565082	7850.3910565376	1.0037137588	3.3045026178	0.4566770028	0.547036756	-0.7394206178
dyna_q_with_predictor_full_training_dyna1	0.173743	730.5918335915	0.4822818742	3.3045026178	0.4566770028	0.0256048714	-3.1307596178
simple_q_full_training	1.739145	115.7019879818	1.2592450659	3.3045026178	0.4566770028	0.8025680631	-1.5653576178
simple_q_learner	0.383597	18.3308916092	1.9858481612	0.4244923418	1.6016915494	0.3841566118	-0.0408953418
simple_q_learner_1000_states	0.729261	18.2818813324	3.4470302926	0.4244923418	1.6016915494	1.8453387431	0.3047686582
simple_q_learner_1000_states_4_actions_full_training	30.149362	157.6974132061	2.2430093689	3.3045026178	0.4566770028	1.7863323661	26.8448593822
simple_q_learner_1000_states_full_training	87.257333	125.3501260281	2.2911390004	3.3045026178	0.4566770028	1.8344619976	83.9528303822
simple_q_learner_100_epochs	0.662728	9.0048823357	4.0933536291	0.4244923418	1.6016915494	2.4916620797	0.2382356582
simple_q_learner_11_actions	0.24127	11.0890343189	1.5440407783	0.4244923418	1.6016915494	-0.0576507712	-0.1832223418
simple_q_learner_fast_learner	0.546827	18.9312882423	2.8787265519	0.4244923418	1.6016915494	1.2770350025	0.1223346582
simple_q_learner_fast_learner_1000_states	0.397123	19.006957531	2.031446602	0.4244923418	1.6016915494	0.4297550525	-0.0273093418
simple_q_learner_fast_learner_11_actions	0.541966	18.9135041237	3.2414383161	0.4244923418	1.6016915494	1.6397467667	0.1174736582
simple_q_learner_fast_learner_3_actions	0.487369	18.467414856	2.9448069674	0.4244923418	1.6016915494	1.343115418	0.0628766582
simple_q_learner_fast_learner_full_training	0.784477	143.5039553642	1.0444534903	3.3045026178	0.4566770028	0.5877764875	-2.5200256178

Figure 36: Results for all the Agents in the Training set.

nb_name	cum_ret	epoch_time	sharpe	cum_ret_bench	sharpe_bench	sharpe_increase	cum_ret_increase
dyna_q_1000_states_full_training	0.068461	18.8205928802	0.4886396985	0.1070225832	0.4427154266	0.0459242719	-0.0385615832
dyna_q_learner	0.004359	18.0854635239	0.0700928916	0.1070225832	0.4427154266	-0.372622535	-0.1026635832
dyna_q_with_predictor	0.091546	338.3656888008	0.6954014538	0.0500215198	0.2930367523	0.4023647015	0.0415244802
dyna_q_with_predictor_full_training	-0.077089	375.830899477	-0.8531759696	0.0500215198	0.2930367523	-1.1462127219	-0.1271105198
dyna_q_with_predictor_full_training_dyna1	-0.006746	38.2427103519	-0.1563573518	0.0728803022	0.3772011735	-0.5335585253	-0.0796263022
simple_q_full_training	0.024973	7.9588027	0.3139835606	0.1070225832	0.4427154266	-0.128731866	-0.0820495832
simple_q_learner	0.195362	18.0976970196	0.9735950444	0.1070225832	0.4427154266	0.5308796178	0.0883394168
simple_q_learner_1000_states	-0.027372	17.7626729012	-0.0867440897	0.1070225832	0.4427154266	-0.5294595163	-0.1343945832
simple_q_learner_1000_states_4_actions_full_training	0.12868	9.899595499	1.1096135235	0.1070225832	0.4427154266	0.6668980969	0.0216574168
simple_q_learner_1000_states_full_training	0.059543	9.0827894211	0.4588056181	0.1070225832	0.4427154266	0.0160901915	-0.0474795832
simple_q_learner_100_epochs	0.008058	8.6537640095	0.0927462721	0.1070225832	0.4427154266	-0.3499691545	-0.0989645832
simple_q_learner_11_actions	0.071247	10.8271145821	0.46914566	0.1070225832	0.4427154266	0.0264302334	-0.0357755832
simple_q_learner_fast_learner	0.092493	17.8824291229	0.6020182965	0.1070225832	0.4427154266	0.1593028699	-0.0145295832
simple_q_learner_fast_learner_1000_states	0.025453	15.7245926857	0.1761813928	0.1070225832	0.4427154266	-0.2665340338	-0.0815695832
simple_q_learner_fast_learner_11_actions	0.140688	17.6730556488	0.9608337022	0.1070225832	0.4427154266	0.5181182756	0.0336654168
simple_q_learner_fast_learner_3_actions	0.040867	18.1006371975	0.3254406128	0.1070225832	0.4427154266	-0.1172748138	-0.0661555832
simple_q_learner_fast_learner_full_training	0.056606	12.2147328854	1.2605807834	0.1070225832	0.4427154266	0.8178653568	-0.0504165832

Figure 37: Results for all the Agents in the Test set, with "online" learning.

nb_name	cum_ret	epoch_time	sharpe	cum_ret_bench	sharpe_bench	sharpe_increase	cum_ret_increase
dyna_q_1000_states_full_training	0.065282	14.2249646187	0.4267994866	0.1070225832	0.4427154266	-0.01591594	-0.0417405832
dyna_q_learner	0.073073	16.4319849014	0.5191712068	0.1070225832	0.4427154266	0.0764557802	-0.0339495832
dyna_q_with_predictor	0.104034	6.6928989887	0.7435489844	0.0500215198	0.2930367523	0.4505122321	0.0540124802
dyna_q_with_predictor_full_training	-0.029741	8.5153381824	-0.3350379716	0.0500215198	0.2930367523	-0.6280747239	-0.0797625198
dyna_q_with_predictor_full_training_dyna1	0.00838	10.2367663383	0.2028884166	0.0728803022	0.3772011735	-0.1743127569	-0.0645003022
simple_q_full_training	-0.027946	8.0099005699	-0.2562905901	0.1070225832	0.4427154266	-0.6990060167	-0.1349685832
simple_q_learner	0.063725	17.7528762817	0.3664203166	0.1070225832	0.4427154266	-0.07629511	-0.0432975832
simple_q_learner_1000_states	-0.013047	17.6617598534	-0.0137477682	0.1070225832	0.4427154266	-0.4564631948	-0.1200695832
simple_q_learner_1000_states_4_actions_full_training	0.107919	13.8394801617	0.9400492988	0.1070225832	0.4427154266	0.4973338722	0.0008964168
simple_q_learner_1000_states_full_training	0.209947	8.8748004436	1.313919535	0.1070225832	0.4427154266	0.8712041084	0.1029244168
simple_q_learner_100_epochs	0.100324	9.1162464619	0.6420028403	0.1070225832	0.4427154266	0.1992874137	-0.0066985832
simple_q_learner_11_actions	0.019991	10.1873445511	0.1561645032	0.1070225832	0.4427154266	-0.2865509234	-0.0870315832
simple_q_learner_fast_learner	0.187941	18.1391232014	0.964351068	0.1070225832	0.4427154266	0.5216356414	0.0809184168
simple_q_learner_fast_learner_1000_states	0.161627	19.4526543617	0.8228017709	0.1070225832	0.4427154266	0.3800863443	0.0546044168
simple_q_learner_fast_learner_11_actions	0.12766	18.9010019302	0.8238261817	0.1070225832	0.4427154266	0.3811107551	0.0206374168
simple_q_learner_fast_learner_3_actions	0.080364	19.2215330601	0.633286256	0.1070225832	0.4427154266	0.1905708294	-0.0266585832
simple_q_learner_fast_learner_full_training	0.056606	11.4128265381	1.2605807834	0.1070225832	0.4427154266	0.8178653568	-0.0504165832

Figure 38: Results for all the Agents in the Test set, without "online" learning.