KSCHOOL
Master in Data Science
edition 19

MACHINE LEARNING FOR ALGORITHMIC TRADING
An example with APPLE stock

Authored by
Isabel Puche Marín

Academic Year 2019-2020

Table of Contents

List of Abbreviations 4

1. Introduction 5

2. Selection of trading platform 6

3. Methodological Approach 7

3.1. Trading Strategies 7

3.2. Process Flow 7

3.3. ML Model development 8

3.3.1. Time Series 8

3.3.2. Multi-target Regression 8

3.3.3. The choice of model 9

3.4. Model backtesting 10

3.4.1. Trading Algorithm 10

3.4.2. Backtesting 11

Backtest's report 11

Pyfolio's tear-sheets 11

3.5. Visualisation 11

4. Technological issues 13

4.1. Setting up the workspace 13

4.2. Pitfalls 13

4.2.1. Zipline's version updates 13

4.2.2. Package versions compatibility 13

4.2.3. Common errors 13

Pyfolio and Pandas for backtesting 13

Zipline and Empyrical/Numpy interactions 14

5. Main Conclusions and Future Steps 15

5.1. Main Conclusions 15

5.2. Future Steps 16

6. Annex I: Front-end User Guide 17

List of Abbreviations

TA	Technical Analysis
ML	Machine Learning
DL	Deep Learning
IDE	Integrated Development Environment
SMA	Simple Moving Average

1. Introduction

Despite newly available Machine/Deep Learning techniques, financial time series are considered hard to predict, all the more so in the presence of more efficient traded exchanges. However still, trading strategies can create opportunities to monetise forecast through rule-based trading and gain returns over the risk incurred. This is called **algorithmic trading,** "a process for executing orders utilizing automated and pre-programmed trading instructions based on complex formulas, combined with mathematical models and human oversight in order to make decisions to buy or sell financial securities on an exchange" (investopedia).

In the last decades with increasingly available information and technology improvements, machine learning (and deep learning) for algorithmic trading has gained in relevance, mainly for high frequency trading, to such an extent that nowadays trading bots abound. In spite of this, advanced techniques and strategies still coexist with more traditional ones, not yet overridden, hinging on the investor's profile and objectives.

In a novel incursion in this topic, this project aims at assessing **whether ML based *buy-and-hold* trading strategies have the power to outperform similar strategies based on technical analysis.**

In such a purpose, **backtesting** will allow for a proper strategy's performance and viability evaluation thus conducting a simulation and analysis for the risk and profitability of trading strategy over a period of time.

2. Selection of trading platform

Both trading and backtesting can barely be programmed in Python with Numpy, Pandas and Scipy, but there exist already built-in libraries in Python with a somewhat large supporting community.

According to literature, **Zipline** is one of the most mature of all currently available choices. It has a rich functionality and scales well with large amounts of data. It has been developed and maintained by Quantopian, a Boston based investment firm, aiming at untapping talent among users through open competitions. The selected strategy is allocated a sum of money and the author paid based on performance. The Quantopian website uses the backtesting engine Zipline, but the library can be installed locally to reproduce a research environment. Trading in the website is in principle easier, but robust quantitative modelling and trading with non-US equities need a local environment.

Zipline comes in with the functionality of extracting free data source from Quandl, a financial provider. It provides US equities from 1990 until 2018, when the service was discontinued. It also integrates other useful libraries for financial statistics (**Empyrical)** and backtesting (**Pyfolio**), and is compatible with other traditional Python libraries, such as Pandas, Numpy, Scipy and Matplotlib. Zipline also offers a wide range of built-in factors for modelling.

However, local installation is non-trivial, as it requires a conda environment installation for 3.5 (or 2.7) version of Python and it has its downsides in its use, as we shall see.

3. Methodological Approach

3.1. Trading Strategies

The project is based on two momentum strategies, consisting of exploiting upward and downward trends in the belief they will continue its current direction: a **simple moving average crossover strategy** (TA Strategy) and a **machine learning based strategy** (ML Strategy).

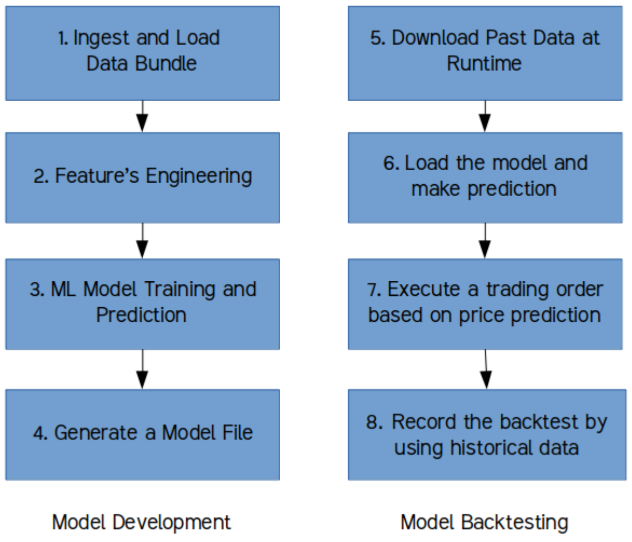
TA strategy's signals are triggered each time a rolling moving average cross the price. When the moving average crosses up the the price, a buying signal is recorded and vice-versa. ML strategy's signals are triggered at the end of each trading window (8 days) by comparing price prediction (with ML techniques) with (a subset of) historical data (32 days): if the prediction's maximum value is bigger than the mean of the historical sub-sample, a buying signal is activated and the other way round.

These are built on top of a *buy-and-hold strategy*, as defined by investopedia as "a passive investment strategy in which an investor buys stocks (or other types of securities such as ETFs) and holds them for a long period regardless of fluctuations in the market (…) and short-term price movements". Capital constraints and stop-losses have not been specifically addressed.

TA Strategy is considered the **baseline strategy.**

3.2. Process Flow

In ML strategy, there are two major processes —one on **model development** and another on **model backtesting**. In TA strategy, there is only a model backtesting process.


1. Ingest and Load Data Bundle
2. Feature's Engineering
3. ML Model Training and Prediction
4. Generate a Model File
Model Development
5. Download Past Data at Runtime
6. Load the model and make prediction
7. Execute a trading order based on price prediction
8. Record the backtest by using historical data
Model Backtesting

3.3. ML Model development

3.3.1. Time Series

As aforementioned, US equities from 1990 until 2018 are made available by Quandl in a data bundle. A **bundle** is an interface to import and read data into Zipline and store it in its own preferred format (OHLCV plus dividends and splits). The library is able to read data incrementally, and only holds a part of the data in its memory at any given time. **Ingesting** is then the process of reading data with the help of a bundle, and storing it in Zipline's format. For data processing, the bundle needs to be **loaded** beforehand.

For this project, **APPLE** daily time series has been used, for no particular reason other than being the first U.S. company that recently hit $2 trillion. The original idea was to employ IBEX35 bluechips, more concretely Repsol's data, but despite creating a custom data bundle, for which a specific process is required in Zipline (see 00 notebook), an assertion error occurred at algorithm runs that could not be handled.

Despite the benefits of using (simple or log) returns instead of price in terms of stationnarity, a low correlation between features and targets and low explanatory power of regressors were reported, probably due to time series' loss of memory. That is the reason for using **close price**.

3.3.2. Multi-target Regression

A subset of historical data (32 lagged variables) has been used to predict some values ahead (8 lead variables). For the sake of interpretability, other factors based on fundamental or alternative analysis (e.g. sentiment) have been disregarded since model's specifications were deemed to be good enough and avoided to add more noise. Furthermore, TA strategy is solely based on price.

As such, this is **multi-target regression** problem, which involves predicting more than one numerical values given an input example. Some algorithms do support multiouput regression inherently, such as linear regression and decision trees. However there are also special workaround models that can be used to wrap and use those algorithms that do not natively support predicting multiple outputs. This is where Scikit's function multi-output regressor comes at hand for it fits one regressor per target.

As a drawback, it is not feasible to build a model's pipeline for data scaling and model fitting, which can ultimately help to select the best model when some of them need to be passed a wrapper.

This ML model has been inspired from the course Machine Learning for Algorithmic Trading Bots with Python, by Mustafa Qamar-ud-Din.

3.3.3. The choice of model

Several models have been fitted in order to allow for a robust comparison. Here is a list with each models specificities and results.

– **Linear regression** captures in principle general (linear) trend but tends to miss details and to output biased models. However, this has not been the case, being one of the most performing models in terms of goodness of fit and error term (though the latter is not specially surprising)

– **Decision trees** can also capture non-linear relations but in theory do not model well enough variance data (since they have limited depth). Contrary to our intuition, it works well with training data and does predict well enough test data.

– **Random forest** instead provide a good balance between bias-variance models. This model (i) is a collection or ensemble of decision trees, (ii) uses bootstrapping aggregating (bagging) - with replacement and, (iii) uses a sample of features at each split (thus helping reduce the model's variance).

– **Gradient Boosting** is similar to random forest models, but the difference is that trees are built successively. With each iteration, the next tree fits the residual errors from the previous tree in order to improve the fit.

Gradient Boosting and Random Forest have been the most performing models, however the error term in both are bigger than in linear regression. Despite that, they both have successively plugged into the trading strategy.

– **Adaboost** (subsequent weak learners are tweaked in favour of those instances misclassified by previous classifiers)**, Kneighbors** (measures euclidean distance to find similar points for predictions) and **Support Vector Machines** (maps data into separated categories) have also been fitted with poorer results than those achieved by Gradient Boosting and Random Forest.

In order to prevent big features outweigh small ones, which is the case with increasing time series like AAPLE value stock, it is preferable to **scale data**. This has been done for KNN's model and SVM.

Hyperparameters have been tuned with ParameterGrid function and GridSearchCV. **Cross-validation** has also been performed with TimeSeriesSplit thus preventing data to reshuffle by fitting the models on the oldest data and evaluating on the newest data.

Finally, the model was saved in a **Joblib** format.

3.4. Model backtesting

3.4.1. Trading Algorithm

Zipline Quickstart guide provides a thoroughful documentation on trading algorithm but the structure is briefly laid out herebelow (see DataCamp's tutorial for more information):

– initialize(): the first function is called when the program is started and performs one-time startup logic. It takes an object context, which is used to store the state during a backtest or live trading and can be referenced in different parts of the algorithm

– handle_data(): this function is called once per minute during simulation or live-trading to decide what orders, if any, should be placed each minute. The function requires context and data as input, the latter being another object that stores several API functions, such as current() to retrieve the most recent value of a given field for a given asset or history() to get trailing windows of historical pricing or volume data. The data object allows you to retrieve the price, which is the forward-filled, returning last known price, if there is one. If there is none, an NaN value will be returned. Another object is the portfolio, accessible through the context object. It allows to enter the portfolio has_positions object, including number of shares and price paid. Other portfolio's properties (not used here) are cash (current amount of cash in your portfolio) and amount (whole number of shares in a certain position). Finally, order() places an order by the amount number of shares.

– _test_args() define start and end date and capital base.

– analyze() settles the backtesting report

3.4.2. Backtesting

- **Backtest's report**

After running the algorithm a backtest's report is issued. Possible columns to check can be envisioned in run_algorithm script under *strategies*' folder. Most importantly, these reports return profit and loss measures, capital used, portfolio value, turnover and leverage ratios, etc (more in *reports* folder).

- **Pyfolio's tear-sheets**

Pyfolio provide backtest ratios and graphs most commonly used in financial analysis, such as returns (annual / monthly, cumulative and its distribution), measures of volatility including Sharpe and Sortino ratios, drawdowns, first to fourth moments of time series'distribution, among others.

3.5. Visualisation

Considering the impossibility of using Streamlit and Altair saver (interactive graphs), see paragraph 4.2.2, Flask has been used instead to create a web page with static files (images) to display backtest results. (User guide on Annex).

This webpage has three directories: (i) index, (ii) a form to be filled with the investor's name and (iii) a home directory with the two trading strategies' portfolio values and backtest results. The dossier for flask webpage is located in the *viz* subfolder of *images.*

This webpage has been built with the help of The Ultimate Flask Course, by Anthony Herbert. The Quickstart guide from official documentation is to be found here.

- **Flask App and html files**

The app.py script contains the following:

– necessary libraries imported from flask,

– an instantiated flask object called name,

– configuration adjustments to allow for debugging and a secret session to prevent cookies' session from being modified,

– app routes to webpage directories: (i) index, (ii) home and (iii) the form. The last two directories return a render template (located in the *templates* folder) named after each function,

– a form function defining two methods ‘GET’ and ‘POST’. The ‘POST’ method allows to retrieve the user name’s information through the form with a request function. It also specifies a submit button (see form.html). After doing so, it redirects the information to home directory. The ‘GET’ method allows to display this information in the home directory (see home.html),

– a home function defining the session name and rendering template of the home html file. This one lays out the titles’ structure and embed two static files with the url_for function that is, two png images located in the *static* folder.

4. Technological issues

4.1. Setting up the workspace

Following documentation, a conda environment has been created in order to (i) run Zipline in a Python version 3.5 environment, (ii) isolate Zipline's dependencies and (iii) control for possible interactions with base environment. The following libraries have also been installed: more importantly, Pandas, Numpy, Matplotlib, Seaborn and Altair, Scikit Learn, Joblib, Pyfolio and Flask. For replication purposes, check environment.yml file.

The IDE used for this project is PyCharm as it enables to write high quality Python code. The Autoenv magic for automating the environment's activation and deactivation has also been implemented.

4.2. Pitfalls

4.2.1. Zipline's version updates

Quantopian released in July 2020 a new Zipline **version 1.4**. Two issues should be then considered: (i) set benchmark to false (as the library was experiencing benchmark download problems during algorithm runs), (ii) set time to timestamp.

4.2.2. Package versions compatibility

For Python v3.5 compatibility purposes, older versions of libraries have been installed, thus limiting full potential. As an example, Scikit Learn v0.20 does not include stacking function for ensemble models. In other cases, libraries were directly not supported in py3.5 such as Keras and Tensorflow for deep learning techniques and Streamlit and Altair-saver for visualisation tasks.

4.2.3. Common errors

- **Pyfolio and Pandas for backtesting**

Pyfolio's backtesting tear sheets need data on positions and transactions for tear sheets calculations. Therefore, the context method position or has_position should be indicated in trading's algorithm handle_data function.

- **Zipline and Empyrical/Numpy interactions**

Errors during trading algorithm implementation occurred pointing to (i) returns calculation (empyrical), (ii) use of numpy's log1p and (iii) other deprecated numpy's functions (np.argmin()). These errors were in some cases partially solved, in others, test and trial with algorithm examples was conducted.

5. Main Conclusions and Future Steps

5.1. Main Conclusions

Under same trading conditions, no capital use constraints and stop loss, the following can be concluded:

- **Both strategies differ in performance...**

TA strategy records cumulative returns of 109%, whereas ML strategy reports 6050%. Capital used at the end of the period amounts to 60,537 USD and 116,175 USD respectively, in line with non-existent capital constraints. Three factors explain this: (i) larger capital amounts used in ML strategy than in TA strategy, (ii) ML strategy goes long all the way whereas TA strategy goes short until the beginning of 2017 when price clearly uptrends and, (iii) an important drawdown occurs during the second half of 2016 (with no obvious explanations) and undermines returns.

- **...and also in behaviour**

TA strategy performs very satisfactorily in 2015 (despite volatility), nosedives in 2016 and recovers in 2017. Selling signals under a negative price trend scenario seem to be triggered with lags, which could explain good results in 2015 and first half 2016 despite price downward trend. Reasonably enough, positive price trends consistently drives up returns (2017 - 2018).

In turn, ML strategy performs poorly in 2015 – 2016 and skyrockets in 2017, after large capital amounts have poured in.

Volatility is very high in ML strategy in mid-2016, when there is a change in cumulated returns tendency. Sharpe ratio increases from mid-2017 onwards, consistent with exploding returns. In turn, volatility is subdued in TA Strategy.

Drawdowns are huge (-417%) in ML strategy and occur at the beginning of the trading period (until beginning 2017), when portfolio value does not compensate for losses. TA strategy's drawdowns occur during the whole trading window and are non negligible (peaking at -70% at the beginning of 2017).

Other measures accounting for risk differences among strategies are risk exposure to trading positions and profit and losses. As appreciated, unit measures vary significantly across strategies.

- **So that, strategies might not be comparable in the end.**

Though it is true that ML is more powerful than TA for forecast analysis, we cannot, in light of the previous, conclude that both trading strategies are comparable, all the more so since basic risk management is lacking in these examples. That is also the reason, why profiling in financial domain is so important. There is no *one-size-fits-all* strategy for all investors but risk-adapted strategies.

5.2. Future Steps

This project has been conceived as a *research in progress* and as such, it has the vocation to further delve into stock prediction and algorithmic trading. Hence, the following could be further addressed:

– introduce **capital constraints** over traded shares and **stop-losses**, among others

– contain **volatility** with VaR (Value-at-Risk) and CVaR (Conditional-Value-at-Risk) metrics, that can also be predicted with (i) parametric estimations, such as Monte Carlo estimations and/or (ii) non-parametric estimations with ML, such as SVR and KDE

– explore other **trading strategies**, including mean-variance ones for portfolio diversification

– explore other machine learning and deep learning **models** for stock prediction

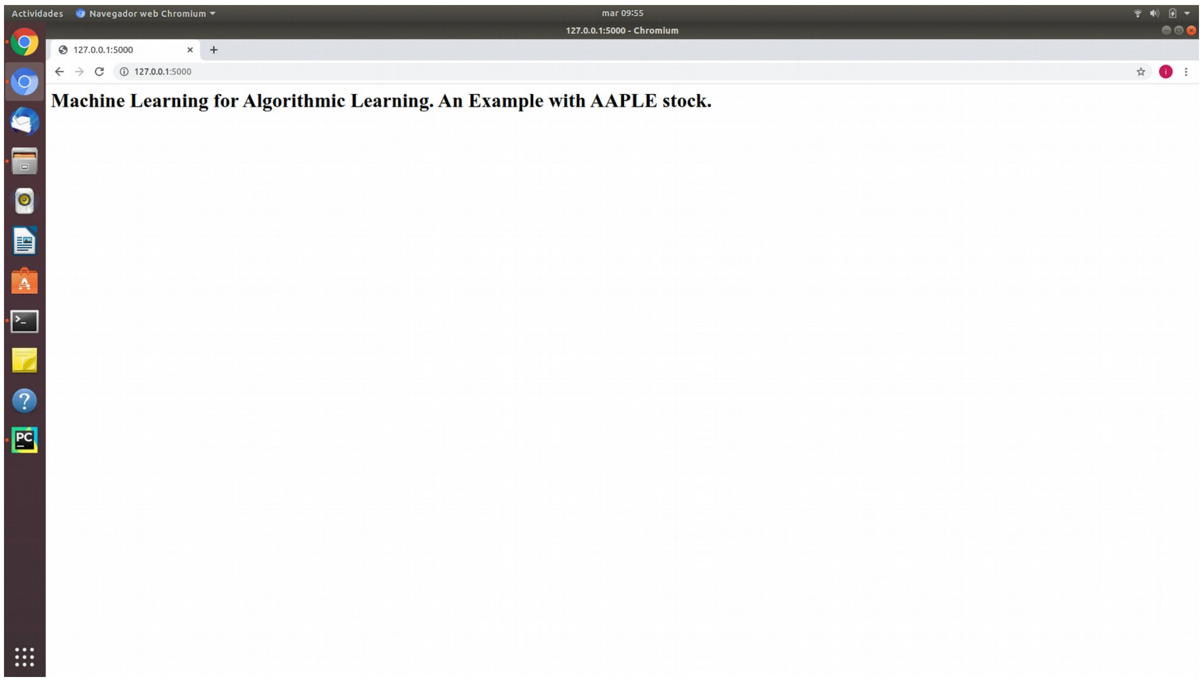
– expand features' engineering with readily available **built-in factors** at Quantopian's and,

– create a financial **dashboard**

6. Annex I: Front-end User Guide

- **Running Flask Apps**

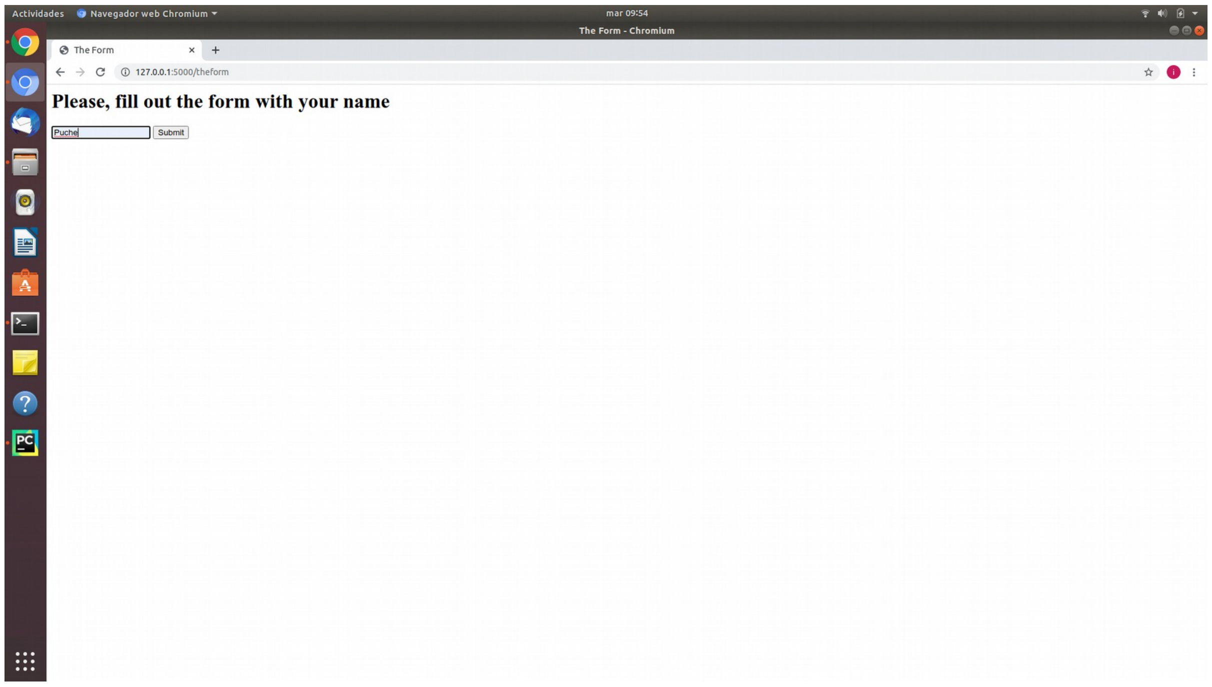
In order to run the flask app, type on the command line: (i) set_FLASK=app.py and (ii) python app.py. Then, click on the web route specified to open the browser. Here, you will be automatically logged in the *index* directory.



Machine Learning for Algorithmic Learning. An Example with AAPLE stock.

Index directory

After that, type in the url address the termination */the form* in order to access the form page where your name should be filled in. Press submit, and you will be redirected to */home* directory.



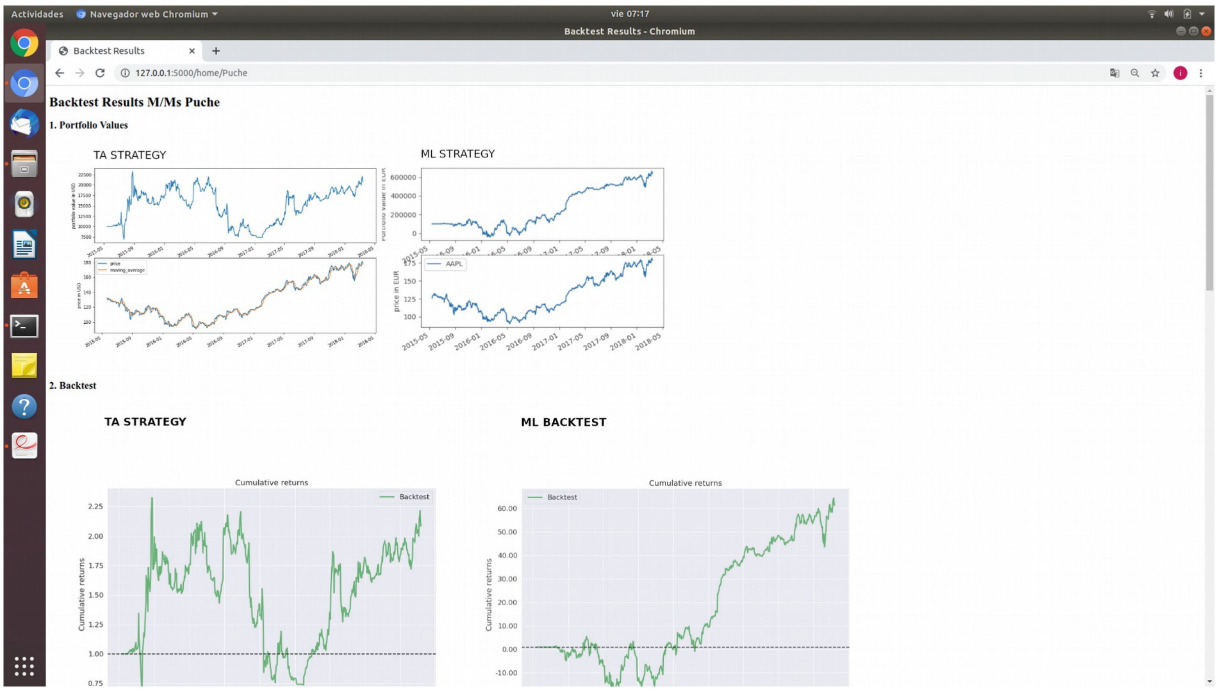
Please, fill out the form with your name

Puche Submit

The Form directory

- **Webpage visualisation**

Home directory contains the images saved in *static* folder:

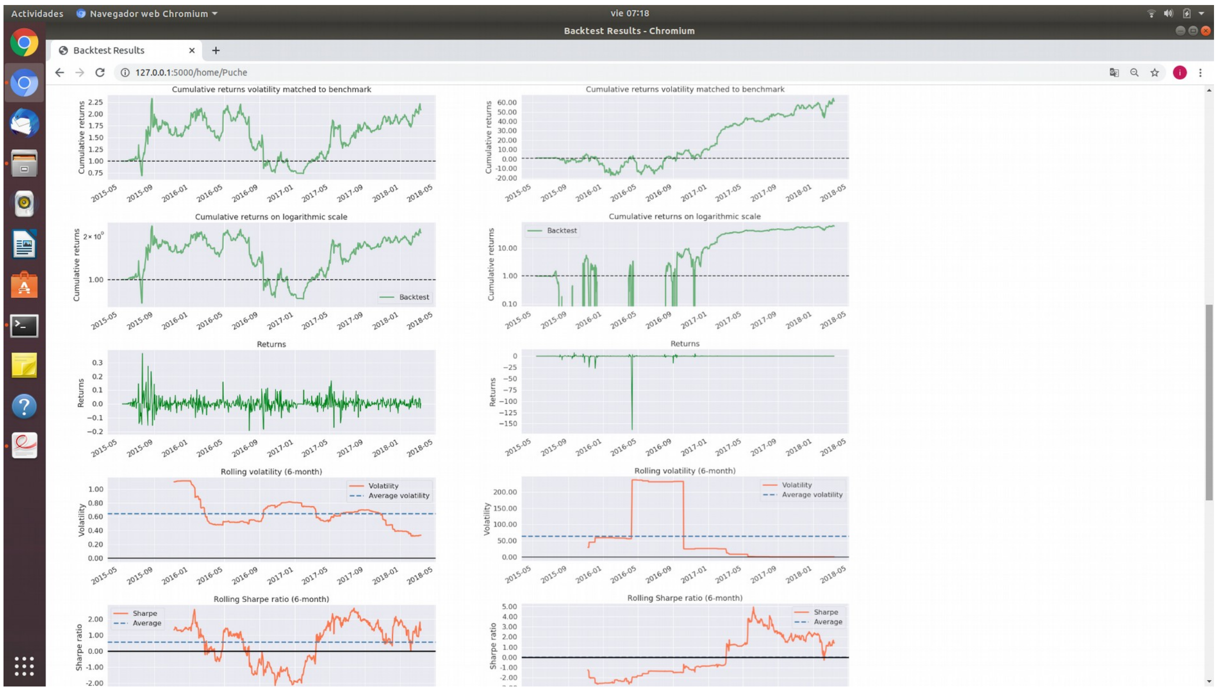


Backtest Results M/Ms Puche

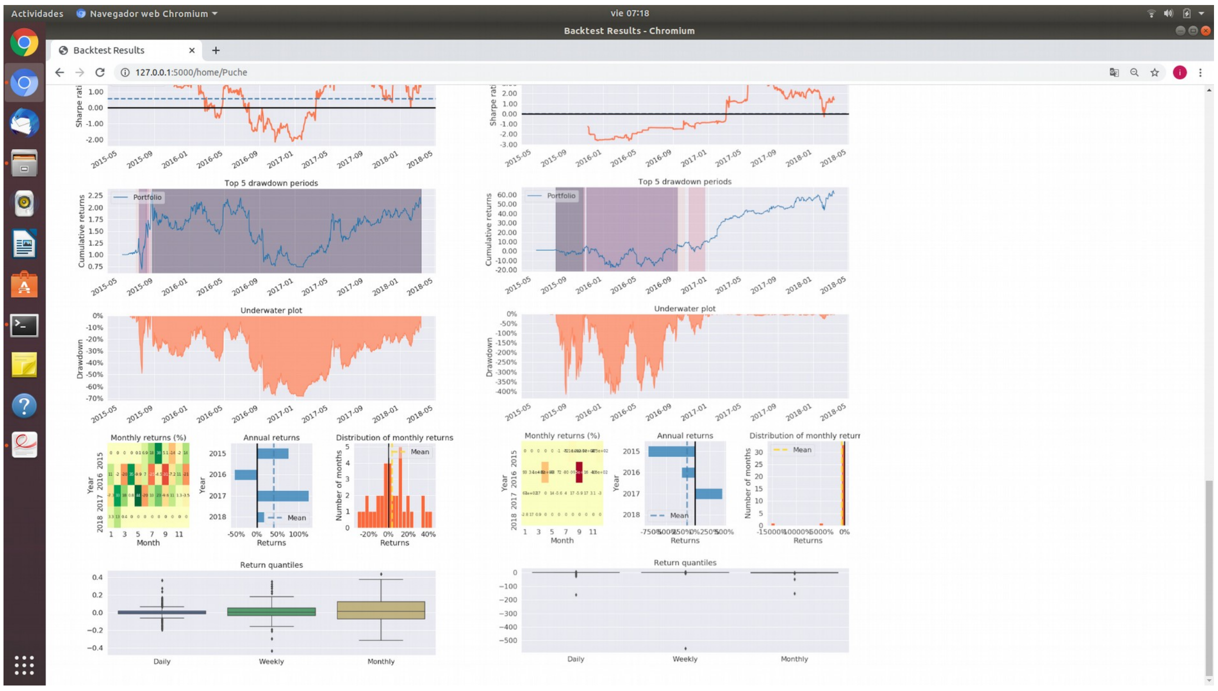
1. Portfolio Values

2. Backtest

Home directory (I)


Actividades
Navegador web Chromium
vie 07:18
Backtest Results - Chromium
Backtest Results
127.0.0.1:5000/home/Puche
Cumulative returns volatility matched to benchmark
Cumulative returns on logarithmic scale
Backtest
Returns
Rolling volatility (6-month)
Volatility
Average volatility
Rolling Sharpe ratio (6-month)
Sharpe
Average

Home directory (II)



Home directory (III)