Financial Time Series Forecasting Challenge
Group 8

Abstract

The analysis of time series data is the inquiry of datasets coming from observations that vary over time.
Financial time series data are used to analyze economic information, e.g., stock prices or companies' sales over time. In particular, we are asked to investigate the returns for 50 time steps in order to forecast the 51st one.

The Data

We are provided with a train and a test set.
The typical procedure consists of three main steps:

1. ***Frame the problem**:*
   First, we visualize our objective in forecasting the 51st one step.
   We deal with supervised and offline learning, using regression (for this reason, we will then compute both MAE and MSE to evaluate the error).
2. ***Look at the data**:*
   Our training dataset is composed of 50 columns and 7326 rows, whereas our test dataset is made of about the half of the rows (3141). There are no missing values.
   We dropped the 1st, "w", since a redundant vector of ones, and the last column, "y", after having recorded its values, since it is the output, our goal.
   We also computed both the Mean and the Standard Deviation vectors with values from every column.
   To investigate the significance of each column, we also calculated the Correlation Matrix, and we noticed that the overall data presented a good correlation coefficient
   (≥ 0.58). What is more, we observed that the last three columns, referring to the last three periods, are the most correlated. This is economically consistent, since, in making predictions, the data belonging to the most recent past are given a larger weight.
   Furthermore, we note that the correlation matrix is symmetric.
3. ***Train/Test split**:*
   The training dataset provided has been additionally split into *train_set* of 80% of the original and *test_set* of the remaining. This is done with a *train_test_split* function from *sklearn.model_selection.*

The toughest part was obviously choosing the right approach for the machine learning problem.

Regression

Clearly, we started from the models we covered in the lectures, the simplest ones.
For all the regression models we performed a preprocessing of the data: we applied the standard scaler model from *sklean.preprocessing* to standardize features by removing the mean and scaling to unit variance. The standard score of a sample x is calculated as $z = \frac{x-\mu}{\sigma}$.

The first model we present is the **Linear Regression Model**: it fits a linear model with coefficients to minimize the residual sum of squares between the observed targets in the datasets, and the targets predicted by the linear approximation (our is a Multiple Regression).
Fitting the model to our data, we then calculate the Mean Square Error (MSE) between our *train_set_target* ("y") and the predictions coming from the linear regression. Its value is approximately 0.0275274.

After this preliminary approach we attempt to compare different types of regressions to choose the most efficient in term of minimization of MSE. In particular, we apply: **Support Vector Machine Regressor** (SVR), **Gradient Regressor Bosting, XGBoost Regressor** (XGB)**, Random Forest Regressor, Stochastic Gradient Descent,** and **K-Nearest Neighbours Regressor** (KNN). The for-loop pointed that the outperforming model was the **SVR**, with a competing MSE of about 0.0218167, also beating the Linear Regression Model.

SVR

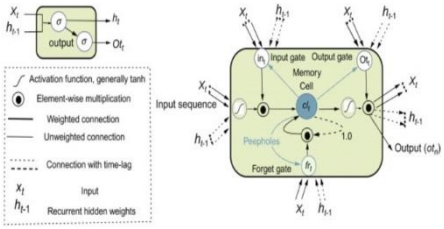
SVR gives us the flexibility to define how much error is acceptable in our model and will find an appropriate hyperplane to fit the data.
We proceed by applying the default parameters.
From this starting point, we apply a *GridSearchCV* to optimize the model according to its hyperparameters, and we focus on *C* and *epsilon*. *C* stands for the regularization parameter, which strength is inversely proportional to *C,* and *C* must be strictly positive. *Epsilon* defines a margin of tolerance within which no penalty is given to errors.
Hence, we performed different combinations of the two (*C*: [1,10,100], *epsilon*: [1,0.1,0.01]) to find the optimal bundle into the values 1 and 0.1 respectively.
Fitting the model to our data, we then calculate the Mean Square Error (MSE) between our *train_set_target* ("y") and the predictions coming from the SVR. Its value is around 0.02.

LSTM

After having performed the regression (to get about 0.02 of MSE), we decide to approach a completely different method, a Neural Networks, which potentially lead to more accurate predictions in the time series predictions framework we are dealing with.

Humans don't start their thinking from scratch every second. Thoughts have persistence.
Traditional neural networks can't do this, and it seems like a major shortcoming. **Recurrent Neural Networks** (RNN), instead, address this issue. They are networks with loops in them, allowing information to endure.

Long Short-Term Memory Networks (LSTM) are a special kind of recurrent neural networks that are capable of selectively remember patterns for long duration of time. They work extremely well on a large variety of problems, included Financial Time Series Analysis, and are now widely used.
It is a network of one input layer, one hidden layer and one output layer. LSTMs have a chain like structure of many cells, the long-term memory. Due to the recursive nature of the cells, previous information is stored within it.
The (fully) self-connected hidden layer contains memory cells and corresponding gate units as shown below.



After this short theory introduction, we proceed preprocessing the data: we use *MinMaxScaler*, since the scaling of the data generally improves the performance and stability of RNNs.
We transform the data to the correct shape of the algorithm, (7326, 50, 1), and we import the *Keras* library.

Subsequential to this phase, we define the characteristics of our LSTM model.

- **Early stopping** is a way to avoid the neural network from running too many times: it defines a *min_delta*, the minimum variation required to be considered an improvement in the loss function, and *patience*, the number of epochs with no improvement after which training will be stopped.
- **Units** are neurons, that define in number the dimension of the output (*units=50*).
- **Input_shape** is used to insert the dimension of our input dataset we previously reshaped.

Then, we create a *GridSearchCV* to optimize the MSE according to its hyperparameters, and we focus on:

- *batch_size* defines the number of samples that will be propagated through the network.
- *epochs* controls the number of times the algorithm will work through the entire training dataset.
- *optimizer* is a method to change the attributes of NN such as weights and learning rate to reduce the losses.

We obtained as best hyperparameters 25, 70 and "adam" respectively (the loss resulting from grid search might appear to have a wrong scale, but the results that are reached are generally correct).
Eventually, we build the model: it consists of three layers, the first two returning the sequences to the next ones. Dropouts with a 20% rate are added to prevent overfitting.
We chose the number of layers by progressively adding more and checking the performance of the model. Fitting the model to our data, we then calculate the Mean Square Error (MSE) between our *train_set_target* ("y") and the predictions coming from the LSTM. The best validation loss we managed to obtain was around 0.015, that was achieved after progressively fitting the model, which is generally shown to improve the performance of the model itself.

NB: to achieve a better performance, we exploited the power of calculation of Google Colab environment.