Ανάπτυξη Λογισμικού Για Αλγοριθμικά Προβλήματα

Χειμερινό Εξάμηνο, Ακαδημαϊκό Έτος 2021-2022

Assignment 3

Μηνάς Διολέτης 1115201400272
Ιωάννης Ροβιθάκης 1115201800164

Github link: https://github.com/rvthak/AlgoProject3

Project Structure

- **forecast.py :** Forecasts future stock prices using LSTM NN
- **detect.py :** Detects price anomalies using LSTM Autoencoder
- **reduce.py :** Reduces the size of the given input curves using a Convolutional Autoencoder

QUERY A : STOCK PRICE PREDICTION

Preprocessing

- We split the dataset for training and evaluation, using 80% of the data to train our model on and 20% to check the resulting model's accuracy and generalization capabilities.
- We used a Robust Scaler to normalize our data in order to achieve better results. For each stock, we fit_transformed the train data and simply transformed the test data to scale the data correctly. After a prediction is made we use the same robust scaler (fitted with the train data of this specific stock) to inverse transform the predicted stock price values back to their original scale and receive the final results.
- We split the data into overlapping window-sized subsequences, creating sequences with 7 days worth of historical data. Our model uses 7 days worth of values to make a prediction about the next day.

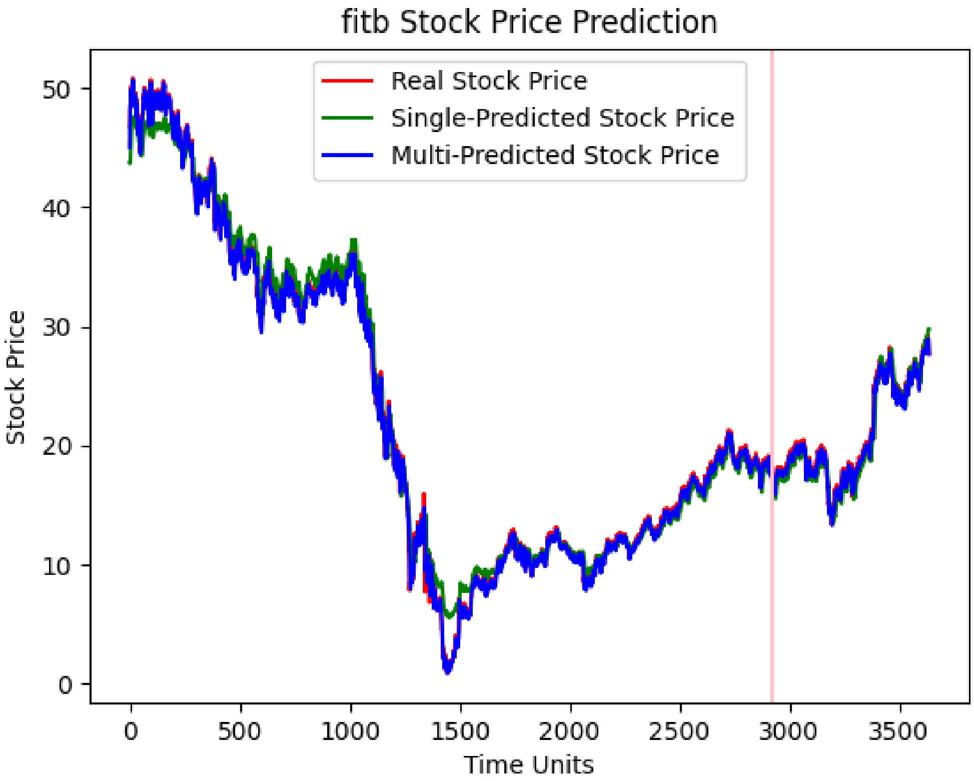
We trained a Single-model for each individual stock, trained by only this specific stock and a single more general, Multi-model, trained using data from all stocks

Our model is a simple LSTM neural network that consists of **5 LSTM** layers (each containing 20 nodes), separated by **Dropout layers** (10% Dropout). It uses the **adam** optimizer and **mean squared error** as a loss function.

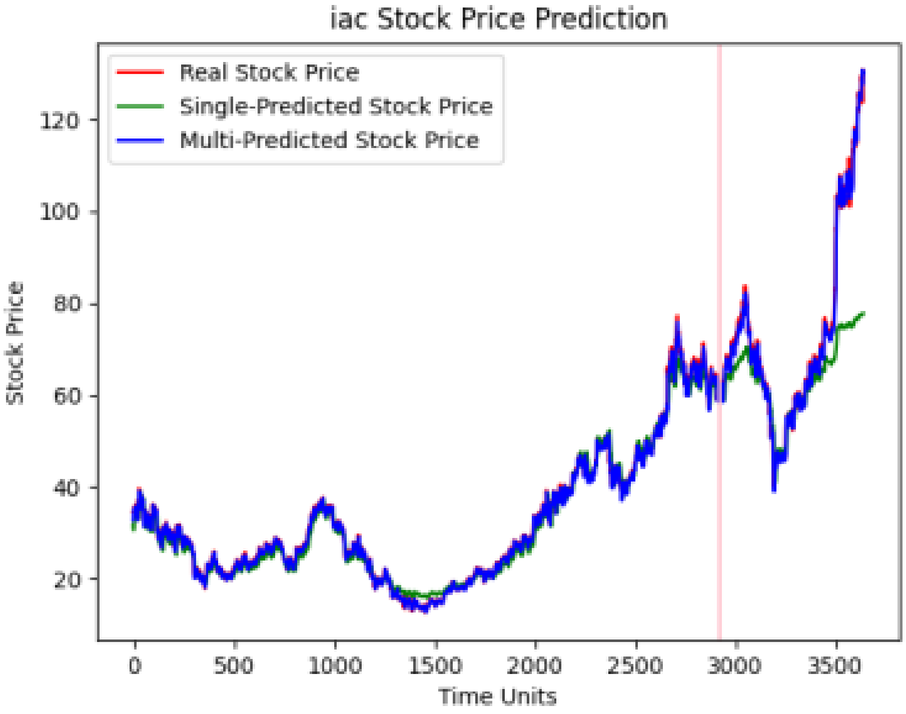
The model was trained using a **batch size of 100** (as a tradeoff for faster training) and **7 epochs** for the single models and 5 epochs for the multi model respectively. The model's hyperparameters were tweaked based on multiple experiments to provide the best results we could achieve. We used evaluate() and the pre-made test sets to calculate the dataset-average loss, and used this as a benchmark for our multi model's accuracy to perform and generalize in new/unseen data.

The resulting models performed well beyond our expectations. The dropout layers combined with the relatively small amount of epochs prevented our models from overfitting to the training data, while still retaining very low loss values.

Some interesting examples are:
(**Note:** The pink vertical line represents the train/test dataset split point)



In this example we see both the single and multi models perform excellently and follow the real prices correctly, even well inside the unknown test-set.



In this scenario, once again we see our models perform inline with the real values, even in very steep and highly volatile price movements inside the unknown test-set. We also see that the single model sometimes struggles to follow very big moves on some occasions while the multi model has no trouble doing so. This behavior is expected since the single model was trained using far less data (only this specific stock's historic prices) when the multi model was trained using data from all 360 stocks. So it is normal to see the single model perform less aggressively in periods of high volatility since its reduced training greatly affects its "vision" and behavior. Finally it is really astounding to witness how well the multi model performs in all the stocks it was tested on. Its generalization capabilities are truly exciting, despite the fact that our model consists of only 5 layers.
(We tried training a model with 15 layers using a very similar Network architecture but it surprisingly failed to outperform the 5-layer implementation)

QUERY B : ANOMALY DETECTION

Preprocessing

- We split the dataset for training and evaluation, using 80% of the data to train our model on.
- Next, we scale the data values to [0,1], applying the same transformation to train and test data.
- After that, we split the data to overlapping window-sized subsequences, creating sequences with 7 days worth of historical data.
- Finally, we concatenate all the training data

LSTM Autoencoder - Keras

- Our autoencoder takes a data subsequence, splitted as mentioned in preprocessing and outputs a sequence of the same shape.
- We pass the training data to the autoencoder 5 times, as the EPOCHS parameter indicates.
- We use a batch of size 100, which defines the number of samples to work through before updating the internal model parameters, where samples are the data subsequences mentioned above.
- All the LSTM layers have 20 nodes, and a 10% dropout rate. We use the dropout technique to prevent the model from overfitting, by randomly “cutting” nodes from each of these layers.
- Adding return_sequences=True in LSTM layer makes it return the previous sequence as an input on the next one.
- The RepeatVector layer simply repeats the input *n* times, while the TimeDistributed layer creates a vector with a length of the number of outputs from the previous layer

Finding Anomalies

- We detect the anomalies by calculating the mean absolute error
- When the error is larger than the given threshold, an anomaly is detected
- After building a dataframe for the loss and the anomalies, we represent the results graphically.

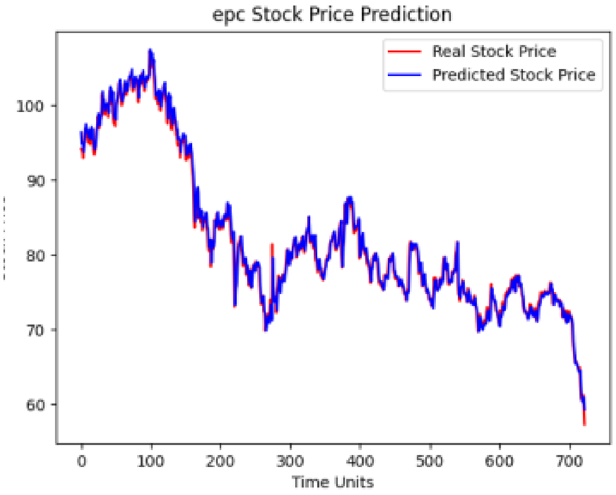
Recommended Executions

Depending on the time-series being evaluated, there are differences between the values of the thresholds needed, so as to have the expected anomalies shown in the graphs. Considering this, here are some time series that we have examined and the corresponding thresholds suggested:

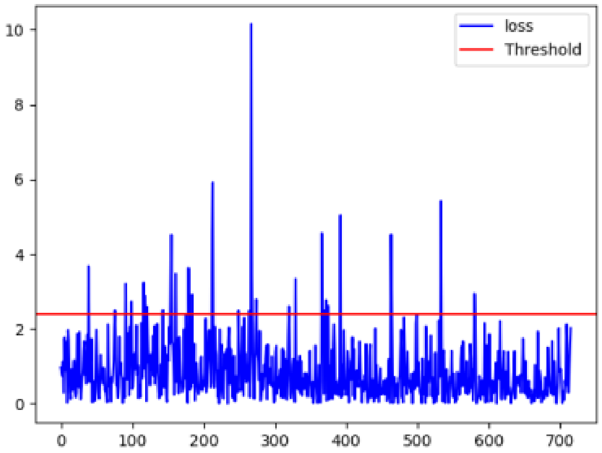
NAME (“-q”)	THRESHOLD (“-mae”)
zbh	2.8
see	1.4
mar	2.5
luv	2.4
epc	2.4
bac	0.7

You can run the examples above using:
python detect.py -d nasdaq2007_17.csv -mae ‘suggested threshold’ **-q** ‘one of the time series name from above’

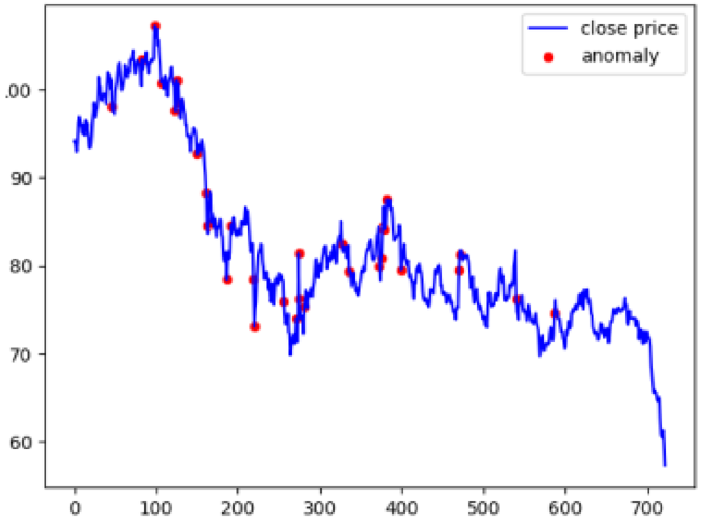
An interesting example



In this example, we can clearly see that our model performed really well, even when the stock has unexpected jumps or drops. Still, small deviations can be detected, which are more clear in the plot below.



With the **threshold set to 2.4**, every loss value that exceeds that threshold is considered as an anomaly. The anomalies detected are represented in the graph below.

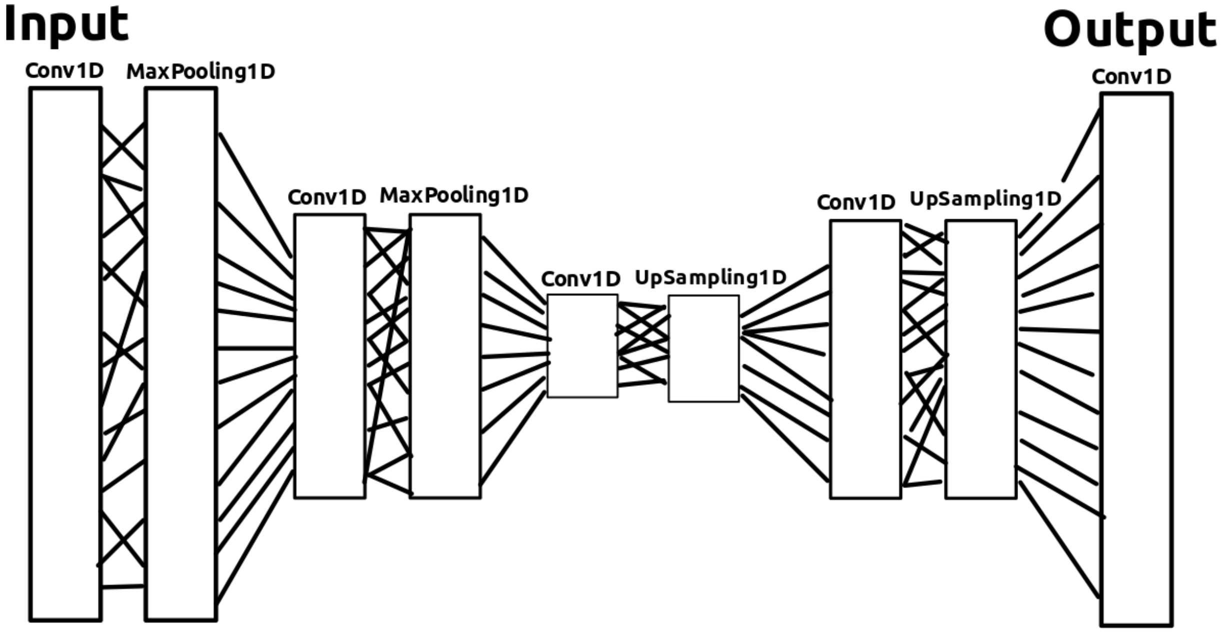


QUERY C : STOCK SIZE REDUCTION

Preprocessing

- We split the dataset for training and evaluation, using 80% of the data to train our model on and 20% to test the resulting model.
- We used a MinMaxScaler to normalize our data in [0,1] in order to achieve better results. For each stock, we fit_transformed the train data and simply transformed the test data to scale the data correctly.
- For the training and testing sets, we split the data into overlapping window-sized subsequences, creating sequences with 10 days worth of historical data.

Autoencoder Model: The model takes 10 prices, encodes them into 3 prices (latent dimension = 3) and then attempts to decode the 3 prices to reconstruct the original 10 prices as closely as possible. The autoencoder model consists of one **encoder** model and one **decoder** model that are almost symmetrical, with the decoder working in reverse.



(The image above is not in scale and the actual layer dimensions are not pictured)

The Conv1D layers apply a convolution operation to filter the data, while the MaxPooling1D and UpSampling1D layers scale the data down to encode and compress the information, or up to decode and uncompress the information accordingly.

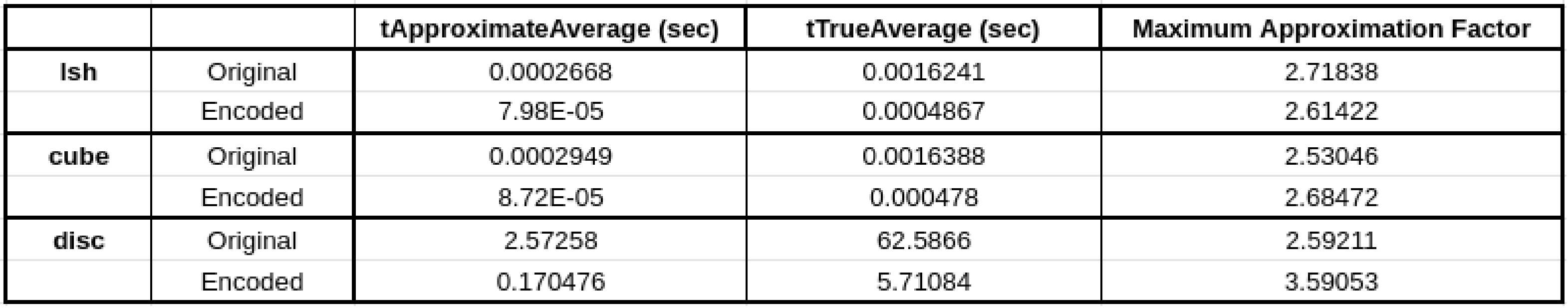
After the autoencoder model is trained, we isolate the encoder model and use it separately to encode any given input data. The encoder's output (**bottleneck**) is supposed to retain as much of the original information as possible in the smaller-compressed size.

We experimented extensively with the autoencoder's structure and hyperparameters and we could not improve the model's performance any further. We tried a variety of scalers and loss functions as well as a couple of different latent dimensions.

It is important to note that the model was trained using **overlapping window** sized data, but when the time comes for the encoder to encode any given input curves, **non-overlapping window** sized data is used. This allows us to train the model using all the available data and then reduce the output's size as needed. Since our goal is to reduce the input's size without losing too much information, overlapping windows on the encoder would not actually reduce the input size at all.

QUERY D : CURVE CLUSTERING ON ENCODED DATASET

Nearest Neighbor:
We ran experiments and compared the results between the original sized input and the reduced size - encoded input:



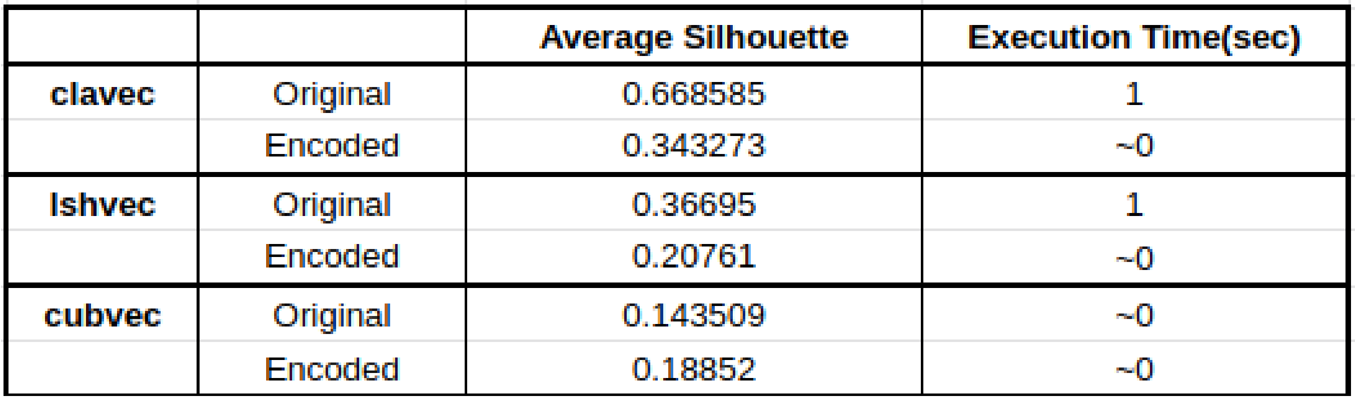
		tApproximateAverage (sec)	tTrueAverage (sec)	Maximum Approximation Factor
lsh	Original	0.0002668	0.0016241	2.71838
	Encoded	7.98E-05	0.0004867	2.61422
cube	Original	0.0002949	0.0016388	2.53046
	Encoded	8.72E-05	0.000478	2.68472
disc	Original	2.57258	62.5866	2.59211
	Encoded	0.170476	5.71084	3.59053

As expected, the encoded input runs much faster (~3x) because of its reduced size (~3 times smaller size). The truly astounding part is that the encoded dataset performs almost as good in clustering as the original sized dataset, despite the encoding! Especially when it comes to using Discrete Frechet Distance, the time benefits of using the encoded dataset far outweigh the minor loss in accuracy.

Unfortunately, due to the large size of the curves, we could not complete the measurements for Continuous Frechet. (Execution time would take more than 30minutes)

(All the above measurements were taken using the **same parameters**)
(Results based on: https://github.com/rvthak/AlgoProject2)

Clustering:
We also ran experiments and compared the results regarding the clustering methods shown below.



		Average Silhouette	Execution Time(sec)
clavec	Original	0.668585	1
	Encoded	0.343273	~0
lshvec	Original	0.36695	1
	Encoded	0.20761	~0
cubvec	Original	0.143509	~0
	Encoded	0.18852	~0

Either using the original sized input or the reduced size/encoded input, the execution time is almost instant. So, what remains for us to compare is the average silhouette produced by each method.

Most of the time, the original-sized input produces better silhouettes in every method. Still, we achieve good results by only using almost one third of the original data, which in larger scale input instances can be very helpful.

Unfortunately, execution time using the Frechet clustering methods (LSH and Lloyd's) is very time-consuming to produce results that can be presented, especially for the original-sized inputs.

(All the above measurements were taken using the **same parameters**)
(Results based on: https://github.com/minasd1/project_assignment2)

Execution

General:

Q.A:

```
python forecast.py -d ‘path/dataset.csv‘ -n ‘number of time series’
```

Q.B:

```
python detect.py -d ‘path/dataset.csv‘ -n ‘number of time series’
-mae ‘desired threshold’
```

Q.C:

```
python reduce.py -d ‘path/input.csv‘ -q ‘path/query.csv‘
-od ‘path/enc_input.csv‘ -oq ‘path/enc_query.csv‘
```

Examples:

Q.A:

```
python forecast.py -d nasdaq2007_17.csv -n 6
```

Q.B:

```
python detect.py -d nasdaq2007_17.csv -n 6 -mae 3
```

Q.C:

```
python reduce.py -d nasd_input.csv -q nasd_query.csv
-od enc_input.csv -oq enc_query.csv
```

For any further questions/suggestions,
feel free to contact us at any time at:
sdi1800164@di.uoa.gr or sdi1400272@di.uoa.gr