Developing a Generic Predictive Computational Model using Semantic data Pre-Processing with Machine Learning Techniques and its application for Stock Market Prediction Purposes

1st Natalia Yerashenia
School of Computer Science and Engineering
University of Westminster
London, United Kingdom
N.Yerashenia1@westminster.ac.uk

2nd Alexander Bolotov
School of Computer Science and Engineering
University of Westminster
London, United Kingdom
A.Bolotov@westminster.ac.uk

3d David Chan You Fee
School of Computer Science and Engineering
University of Westminster
London, United Kingdom
D.Chanyoufee2@westminster.ac.uk

***Abstract*—In this paper, we present a Generic Predictive Computational Model (GPCM) and apply it by building a Use Case for the FTSE 100 index forecasting. This involves the mining of heterogeneous data based on semantic methods (ontology), graph-based methods (knowledge graphs, graph databases) and advanced Machine Learning methods. The main focus of our research is data pre-processing aimed at a more efficient selection of input features. The GPCM model pipeline's cycles involve the propagation of the (initially raw) data to the Graph Database structured by an ontology and regular updates of the features' weights in the Graph Database by the feedback loop from the Machine Learning Engine. The Graph Database queries output the most valuable features that, in turn, serve as the input for the Machine Learning-based prediction. The end-product of this process is fed back to the Graph Database to update the weights.**

We report on practical experiments evaluating the effectiveness of the GPCM application in forecasting the FTSE 100 index. The underlying dataset contains multiple parameters related to predicting time-series data, where Long Short-Term Memory (LSTM) is known to be one of the most efficient machine learning methods. The most challenging task here has been to overcome the known restrictions of LSTM, which is capable of analysing one input parameter only. We solved this problem by combining several parallel LSTMs, a Concatenation unit, which merges the LSTMs' outputs (into a time-series matrix), and a Linear Regression Unit, which produces the final result.

***Index Terms*—semantic data analysis, graph database, ontology, stock analysis, computational model, neural network, linear regression, FTSE 100, LSTM, Protégé, Neo4j, Python**

I. Introduction

The paper introduces a Generic Predictive Computational Model (*GPCM*) and applies it building a Use Case for the FTSE 100 index forecasting. This involves the mining of heterogeneous data based on semantic methods, graph-based methods (ontology, knowledge graphs, graph databases) and advanced Machine Learning methods. The main focus of our research is data pre-processing aimed at a more efficient selection of input features.

GPCM has the following pipeline. Raw Data collected from an external source is fed to the model Graph Database which is additionally structured by the ontology and, after the first model cycle, is regularly updated by the feedback loop from the Machine Learning Engine. The latter is used to update the weights (values of importance) of the nodes (features) in the Graph Database. The Graph Database queries output the most valuable features that subsequently serve as the input for the Machine Learning-based prediction. The end-product of this process is fed back to the Graph Database to update the weights.

In the course of practical experiments, we built a Use Case and evaluated the effectiveness of the *GPCM* model application for forecasting the FTSE 100 index[1]. To adapt our model for this Use Case we developed a special *Market Index Prediction Ontology (MIPO)* and a comprehensive Machine Learning Engine utilising Long Short-Term Memory (LSTM). This Machine Learning Engine was based on the use of multiple parameters related to predicting time-series data: the market index itself and the set of macroeconomic indices, used as additional external impact factors. The most challenging task here was to overcome the known restrictions of LSTM, the primary and one of the most accurate methods of forecasting time-series market data – LSTM is intended to analyse one input parameter only (dealing with time-series

[1]https://markets.ft.com/data/indices/tearsheet/summary?s=ftse:fsi

2378-1971/22/$31.00 ©2022 IEEE
DOI 10.1109/CBI54897.2022.00013

vectors).

Therefore, to enable LSTM to tackle multiple input parameters, a unique hybrid Machine Learning Engine was developed. It consists of several parallel LSTMs, a Concatenation unit, which merges the LSTMs' outputs (into a time-series matrix), and a Linear Regression Unit, which produces the final result.

To develop an Ontology which structures the Graph Database we use Protégé[2]. As a graph database environment, Neo4j[3] was chosen. The integrated code for the *GPCM* model is written and executed in Python; it will be explained in the relevant sections and can also be found at our GitHub repository [1].

The structure of the paper is as follows. Section II overviews work related to the application of the LSTM for time-series data prediction as the stand-alone single-component models. In Section III the architecture of the Generic Predictive Computational Model is presented. This model is implemented in Section IV which presents a Use Case for the Market Index Prediction purposes (FTSE 100 index): Section IV-A describes input data, Section IV-B presents the development of the relevant Ontology, Section IV-C introduces the graph DB hosting multiple parameter input data, Section IV-D describes the hybrid ML engine. Section V evaluates the developed model. Finally, Section VI summarises the contribution of this paper and draws paths for future work.

II. Related Works

Recurrent Neural Networks. The most popular method for performing classification and other analysis of data sequences are *Recurrent Neural Networks* (*RNNs*) [2]. However, in problems of time-series analysis, a modification of such networks is especially distinguished – *Long Short-Term Memory* (*LSTM*) networks [3].

The idea behind a RNN is to use information consistently. In traditional neural networks, all inputs and outputs are assumed to be independent. But this is not suitable for many tasks [3]. For example, if it is necessary to predict the next value of the market index, it is best to consider the values that precede it. RNNs are called recurrent because they perform the same task for each element of the sequence, depending on previous computations [4]. Another interpretation of a RNN is a network with a "memory" that considers prior information.

Pure recurrent neural networks are not used very often in practice. The main reason for this is the *vanishing gradient problem* [2]. Ideally, for recurrent neural networks, a long chain of "memories" is needed as an input so that the network can connect data relationships over significant distances in time [5]. For example, such a network could make real progress in understanding how events in the stock market are related. However, the more time steps we have, the more chances that backpropagation gradients will either pile up and explode or disappear.

To reduce the vanishing gradient problem and therefore allow recurrent neural networks to perform well in practice, there must be a way to reduce the multiplication of fewer than zero gradients [4]. A modified version of the RNN – the LSTM, is a specially designed logical unit that will help reduce the vanishing gradient problem enough to make recurrent neural networks more practical for long-term tasks [6]. It does this by creating an internal memory state that is simply added to the processed input signal, which significantly reduces small gradients' multiplicative effect. In addition, the timing and impacts of previous inputs are controlled by a concept called *the forget gate*. Forget gate determines which states are remembered or forgotten.

Hochreiter and Schmidhuber [7] introduced LSTM in 1997, and then optimised and popularised them in many subsequent works. Such networks do an excellent job of solving many problems and are widely used at this time and this method is considered to be one of the most powerful for market data analysis [6], [8], [3].

All the papers mentioned above consider LSTMs as the stand-alone method for time-series financial data prediction. However, we use the LSTM as the component of a comprehensive computational model, which aims to improve the efficiency of the prediction.

III. Generic Predictive Computational Model Architecture

The component-based architecture of the *Generic Predictive Computational Model* (*GPCM*) is shown in Fig. 1. It consists of the following components: *Raw Data Interpreter*; *Ontology*; *Graph Database*; *Feature Selection Component*; and *Machine Learning Engine*.

This system significantly improves the previous construction of a predictive model [9], [10]: the Graph Database component now has a three-layered structure and the model is enriched with the *Feedback Loop*. However, the use case presented in the above papers can still be considered as a use case for the enriched model.

Raw Data Interpreter. The role of this module is to collect raw data from the sources' standard databases. The data gained are subsequently stored in a convenient form to support its efficient management. The interpreter allows feeding the data in an appropriate commonly used format (e.g., CSV) to the Graph Database module (see below).

We aim at the construction of a *generic model* the realisation of which initially depends on the type of the input data. This will enable its application for data mining of all kinds of quantitative datasets; however, we believe it is more suitable for highly interconnected data. For our use case, we have determined to use financial/market data.

Semantic Graph Database. Graph Database (*GDB*) stores the raw data collected from the companies' standard databases (for our use cases we use the Neo4j environment for *GDB* construction) structuring data based on the Ontology. Additionally, *GDB* is dependent/dynamically updated by the feedback from the Machine Learning Engine. It allows to include the outcomes and analytical metadata (e.g. features' weights) gained from the model's previous iterations.

[2]https://protege.stanford.edu/products.php
[3]https://neo4j.com/product/neo4j-graph-database/

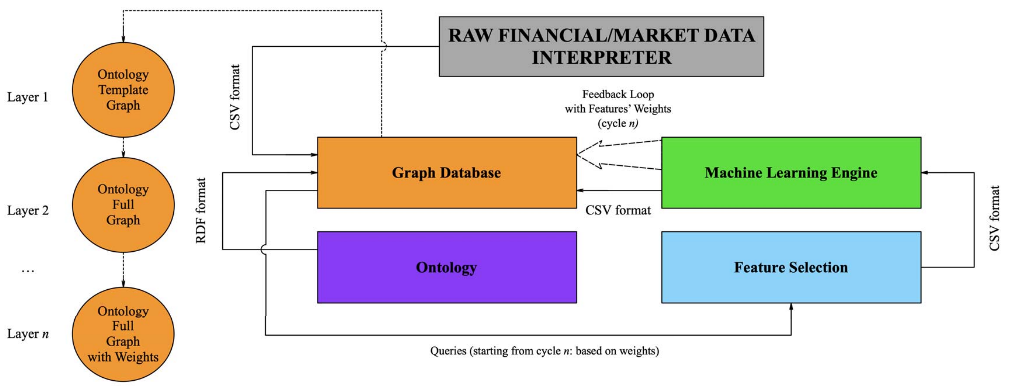


Fig. 1. GPCM Components and Dataflow

we use a *GDB* instead of a traditional database mainly because it copes well with highly interconnected data (e.g., financial data). It effectively implements the property graph model, right down to the storage layer. This means that the data is stored exactly as one represents it. Only a database that initially contains all possible connections can efficiently store, process, and query relations. While traditional databases compute relationships at query time through resource-intensive operations, a *GDB* stores connections and the model's data. Time traversal constants on large graphs facilitate the efficient representation of nodes and relationships. Therefore, regardless of the overall size of the dataset, graph databases do an excellent job of managing data with complex relationships and complex queries [11]. Moreover, the flexible nature of the graph allows it to adapt over time, subsequently adding new concepts and relationships to quickly access and speed up data processing as the analysis needs change.

The Ontology included into the system formalises the theoretical concepts needed for the data analysis and creates a framework according to which the entire system will operate. An ontology defines a standard concept library for users and developers who need to share information. It includes computer-interpretable formulations of the basic concepts of the subject area and the relationship between them, making them more accessible for complex reasoning. Both tools, ontology and graph database, are compatible with popular programming languages, including Java, JavaScript, .NET, Python, which helps automate and speed up the system management, construction process and integrate independent components into a single system.

In *Ontology template graph* (Definition IV.1 in [10]) nodes are labelled by "abstract empty containers" while in Ontology full graph (Definition IV.2 in [10]) these "abstract containers" are filled with the values gained from the concrete data (i.e., the specific use case metadata).

Now, we introduce an extra layer of the Ontology graph – *Ontology full graph with weights* (Definition III.1), where the Machine Learning Engine feedback loop adds the weights of the features after every model's iteration.

Definition III.1 (Ontology full graph with weights)**.** An *Ontology full graph with weights* is a labelled graph $G_w = < V, E, L_w >$, where V is the set of vertices, E is the set of edges (features), and L is the set of labels. Labels in $L = \quad value : 0, \quad value : i, \quad weight : 0, \quad weight : k$, where $value : 0$ is a constant meaning "the value is not yet identified" and i ranges over the real values taken from a company's dataset; $weight : 0$ is a constant meaning "the feature is irrelevant regarding this particular model's cycle" and k ranges over the weights which were added as the result of analysis executed during the previous model's cycle, besides k ranges in (0,1], where 1 means there is a direct dependence between the final outcome of the model and the feature's value.

When the template graph containing the core Ontology information is created, it should be filled with the data - in this way we transform a template graph into the full graph. After the first iteration of the model, the full graph is supplemented by weights of the input features.

Feature Selection. The attributes used to train the model have a significant impact on the quality of the results. Uninformative or poorly informative features may reduce the effectiveness of the model. Therefore, the process of selecting features that have the closest relationship with the target variables is performed. During each iteration, features are being corrected, taking into account the model's previous iterations.

Feature Selection (as well as Data Structuring, Filtering and Visualisation) is optimised by *GDB* queries. This enables the formulation of the queries of any complexity handled by a dedicated query language (e.g., Cypher as part of Neo4j in our case). The efficiency of queries though is supposed to be improved by an expert. The selected set of features feeds the Machine Learning Engine.

Machine Learning Engine. Finally, financial analysis using selected key features can be carried out through Machine Learning Engine (*MLE*).

The *MLE* type choice depends on the particular input dataset characteristics and the specific nature of the analytical objectives (forecasting or classification). For example, for one of the variations of *GPCM – Bankruptcy Prediction Computational Model (BPCM)* (see [10]), where input features are a company's financial ratios, we use Classical Neural Network to identify which category a company relates to (bankrupt or non-bankrupt).

For *Market Index Prediction Computational Model* (see Section IV) the input parameters consisted of the Market Index (FTSE 100) and Macroeconomic Indicators historical values, so we applied a hybrid more complex engine, combining a bunch of Long Short-Term Memory networks plus the *LR*.

Feedback Loop. Besides returning the outcome of the current iteration, *MLE* allows for the analysis of the weights of the input features and for the transferring of this information to the *GDB* component via a feedback loop, enhancing the system performance for future iterations. In Fig. 2 the Feedback Loop is shown separately from the computational model.

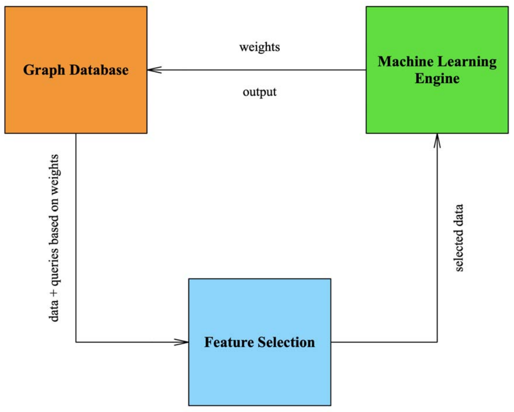


Fig. 2. GPCM Feedback Loop Architecture

Thus, the system represents a dynamic recurrent process of *n* iterations (cycles) that continually researches two central questions: how the provided features impact the target variables and what analytical conclusion the system produced for the particular iteration.

IV. Use Case: Implementation of the Predictive Computational Model (FTSE 100)

This Section will describe how to use *GPCM* for the purposes of FTSE 100 market index prediction.

The architecture of the Predictive Computational Model for FTSE 100 – *FPCM*, is the special case of *GPCM*.

A. Input Data for Market Index Predictive Computational Model

Macroeconomic statistics have a significant impact on the behaviour of investors in the financial market. Despite the rapid development of algorithmic trading, many financial market participants continue to build their trading strategies based on new data on the current macroeconomic situation. For example, the publication of data on oil stocks can affect the value of quotes for oil futures and other derivatives. Or statistics on the number and volume of new construction can give an idea of the phase of the economic cycle and the state of the economy as a whole, forcing investors to withdraw money from riskier assets and invest in safer companies.

Recently, more research has been concentrated on the analysis of the impact of macroeconomic indicators on financial markets [12], [13], [14], etc. Standard models for predicting the prices of market indices, such as the NASDAQ, S&P500, FTSE 100, use the time-series of the previous values of a given index solely [15]. We propose to supplement the input data of *FPCM* by the monthly values of some macroeconomic indicators.

FTSE 100 (abbreviated from Financial Times Stock Exchange, also known as Footsie) is one of the most popular (including for trading) stock indexes. The FTSE combines data from companies that make up approximately 80% of the London Stock Exchange market capitalisation, and, unlike all other indices, is not wholly-owned by the exchange: its shares are owned by the Financial Times[4].

As the input dataset for *FPCM*, we consider the average monthly index data from January 1, 1985 to October 1, 2020 -that is 430 time-slots (i.e. 1 month for 1 time-slot)- and the last 60 slots will be used as test slots for verifying the accuracy of the model prediction.

The composition of the FTSE 100 index is determined quarterly; it is unstable and difficult to predict, taking into account the data of companies as of the end of the previous business day. Every quarter, some companies that do not meet specific requirements are dropped from the list, and other companies are put in their place. In this regard, it was decided not to base our research on the performance of the companies that make up the index.

At the same time, most of the companies included in the index are British, political and economic news from the EU also have some impact on the FTSE. However, the main factors still belong to the UK, and news such as decisions on interest rates, GDP, production, and inflation influence the index fluctuations.

After an extensive research on the most popular macroeconomics indicators which are correlated with financial market fluctuations [13], [12], we have selected 16 of them that will help improve the predictions of the index. They are presented in Table I.

B. Developing Ontology of Market Index Prediction

Although there are no UK or International standards which define the relationships between macroeconomic indicators and the market indexes, the *MIPO* Ontology is based on experts opinion.

A formal physical representation of this Ontology is created in the Protégé environment. It is far more complex than *OBP* Ontology (developed for the Use Case of companies' bankruptcy prediction in [10]).

The *MIPO* Ontology contains the following four classes:

[4]https://markets.ft.com/data/indices/tearsheet/summary?s=ftse:fsi

TABLE I
MACROECONOMIC INDICATORS USED AS INPUT DATA FOR MARKET INDEX PREDICTIVE COMPUTATIONAL MODEL

Code	Category Name	Indicator Name
GDP	3*Economic	Gross Domestic Domestic
GNIph	Output	Gross National Disposable Income
EG		Economic Growth
IR	2*Prices	Retail Price Inflation Index
CPR		Consumer Price Inflation Index
UR	4*Labour	Unemployment Rate
AWE		Average Weekly Earnings
LC		Labour Cost Index
BDIR	4*Banking &	Bank Deposit Interest Rate
BCIR	Investments	Bank Credit Interest Rate
I		Net Investment by UK financial institutions
FDI		Foreign Direct Investments
RS	4*Business	Retail Sales Index
IP	Cycle	Industrial Production Rate
HP		House Price Index
PMI		Purchasing Managers Index (composite)

- *Prices Indexes*. Here we include the FTSE 100 index as a subclass. But the Ontology can be supplemented with other market indexes if the user needs it.
- *Macroeconomic Indicators*. At the moment, this class contains 16 indicators that affect market indexes.
- *National Statistics Indicators*. This is the upper class of the indicators as compared to class Macroeconomic Indicators. It consists of indicators provided and from these macroeconomic indicators are calculated – for example, Population figures which is the basis for Unemployment Rate calculation.
- *Changes in Government Regulations* is a special class that affect both Macroeconomic Indicators and National Statistics Indicators classes. It includes the UK Fiscal Policy, Monetary Policy, Labour Policy changes, etc.

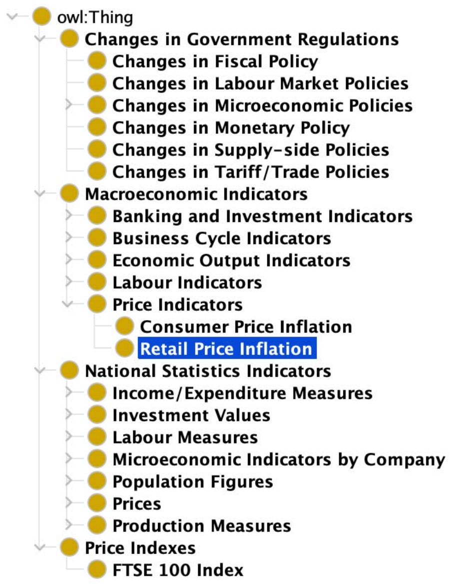
The *MIPO* Ontology is a prototype version, and it can be updated in future with the other factors affecting the changes in market index price.

The MIPO Ontology hierarchy developed in the Protégé environment is shown in Fig.3. The more detailed *MIPO* Ontology diagram is given in our footnote repository.

Besides the SubClassOf structure, the Ontology includes three types of non-hierarchical connections: *hasEffectOn* (e.g., sets the relationships between Macroeconomic Indicators and Market Indexes), *directlyRelatedTo* and *indirectlyRelatedTo* (e.g., establishes the relationships between Macroeconomic Indicators and National Statistics Indicators).

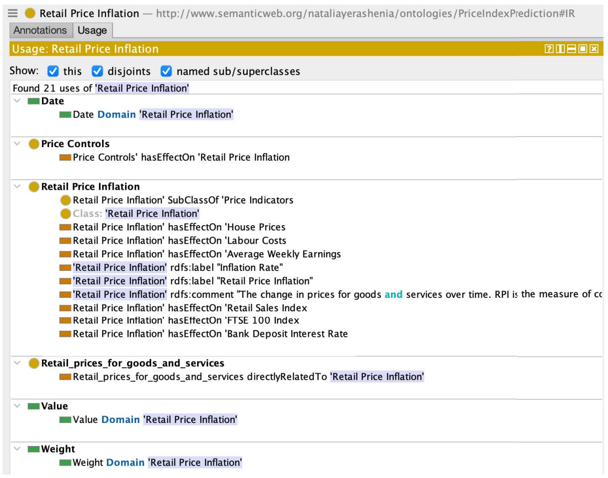
Figure 4 represents the ”Usage” tab of Retail Price Inflation (RPI) indicator in Protégé. It shows that an entity contains three data attributes: *Weight* (which is the indicator importance coefficient), *Value* (the actual value of the Ratio) and *Date* (time-slot). The rest of the indicators and the market index contain the same data attributes. Also, this tab shows the relations of the indicator with other entities in the Ontology. Moreover, every entity includes information about synonyms and term definitions.

The full version of *MIPO* Ontology in *OWL* format – *Price_Index_Prediction_v3.2.owl* is used to create the skeleton of the *FPCM* model *GDB* in Neo4j.



- owl:Thing
  - Changes in Government Regulations
    - Changes in Fiscal Policy
    - Changes in Labour Market Policies
    - Changes in Microeconomic Policies
    - Changes in Monetary Policy
    - Changes in Supply-side Policies
    - Changes in Tariff/Trade Policies
  - Macroeconomic Indicators
    - Banking and Investment Indicators
    - Business Cycle Indicators
    - Economic Output Indicators
    - Labour Indicators
    - Price Indicators
      - Consumer Price Inflation
      - Retail Price Inflation
  - National Statistics Indicators
    - Income/Expenditure Measures
    - Investment Values
    - Labour Measures
    - Microeconomic Indicators by Company
    - Population Figures
    - Prices
    - Production Measures
  - Price Indexes
    - FTSE 100 Index

Fig. 3. Protégé Environment: Ontology of Market Index Prediction (MIPO)



```
Retail Price Inflation — http://www.semanticweb.org/nataliayerashenia/ontologies/PriceIndexPrediction#IR
Annotations  Usage
Usage: Retail Price Inflation
Show: ☑ this  ☑ disjoints  ☑ named sub/superclasses
Found 21 uses of 'Retail Price Inflation'
Date
    Date Domain 'Retail Price Inflation'
Price Controls
    Price Controls' hasEffectOn 'Retail Price Inflation
Retail Price Inflation
    Retail Price Inflation' SubClassOf 'Price Indicators
    Class: 'Retail Price Inflation'
    Retail Price Inflation' hasEffectOn 'House Prices
    Retail Price Inflation' hasEffectOn 'Labour Costs
    Retail Price Inflation' hasEffectOn 'Average Weekly Earnings
    'Retail Price Inflation' rdfs:label "Inflation Rate"
    'Retail Price Inflation' rdfs:label "Retail Price Inflation"
    'Retail Price Inflation' rdfs:comment "The change in prices for goods and services over time. RPI is the measure of cc
    Retail Price Inflation' hasEffectOn 'Retail Sales Index
    Retail Price Inflation' hasEffectOn 'FTSE 100 Index
    Retail Price Inflation' hasEffectOn 'Bank Deposit Interest Rate
Retail_prices_for_goods_and_services
    Retail_prices_for_goods_and_services directlyRelatedTo 'Retail Price Inflation'
Value
    Value Domain 'Retail Price Inflation'
Weight
    Weight Domain 'Retail Price Inflation'
```

Fig. 4. Protégé Environment: Retail Price Inflation Usages in MIPO Ontology

C. Developing Semantic Graph Database

The first step in creating a Neo4j *GDB* for Market Index Prediction is to transfer the *MIPO* Ontology *LSTM* made in Protégé into Neo4j.

First, we create and open an empty graph *’FTSE100’* in the Neo4j app window. Then, we define the LAN connection parameters to connect to the *GDB* in Python.

Now we need to import the *LSTM* itself (the Owlready2 library can help us with this) and then proceed directly to extracting information from the *MIPO* Ontology using it as a base for building a graph.

Below we describe the initial two stages of this process:

Stage 1. Establish the connection to the Neo4j database.

Stage 2. Load the Ontology into memory (assigning the Ontology to the "onto" variable).

Nodes and Relationships are created from Python using Cypher queries. All elements of the Ontology are transferred as graph nodes. To make the graph node labels more readable, we decide to remove the name-space prefixes in the entity names, which is how they are naturally given in *RDF* format. Then hierarchical relationships between entities are added, as well as the relationships that are hidden in the annotations of the Ontology (non-hierarchical). Thus, an *Ontology template graph* (an empty graph without values and weights) is formed (Definition IV.2 in [10]).

To convert an *Ontology template graph* to an *Ontology full graph* (Definitions IV.1 in [10]), a graph that is filled with values, we store input data (from some external source) as *Processed_Input_Data_FTSE100_1985_21.csv*. In this conversion the first step is to export this data file to the Python environment and then match the names of the created nodes with the names of the columns.

The complexity of this Use Case is due to the presentation of the input data as a time-series (we have 430 monthly time-slots). This means that *GDB* should contain 430 variations of the Ontology full graph (these will be differentiated by the "Date" data property).

Our software solution reflects the limitations of the Neo4j environment. Neo4j environment doesn't provide any built-in tools to present the time-series data in a more convenient way. Official Neo4j website suggests the KeyLines plug-in[5], but unfortunately it is not available for academic purposes. Each FTSE100 and Macroeconomic Indicators nodes in Neo4j should contain: "Value", taken from the external source, and "Date" (for example, 01/10/2020 - the dataset's last time-slot). The dataset (*Processed_Input_Data_FTSE100_1985_21.csv*) is formed using the information from *Office of National Statistics* official website[6] and market indexes database on *Yahoo.Finance* website[7].

The code below describes nodes and edges for each time-slot being added to Neo4j. Here, the Pandas library [8] allows us to extract the CSV file's contents and store in the variable "processed_input_csv_dates" the first column of the CSV, containing the dates.

```
processed_input_csv_dates =
pd.read_csv
("Processed_Input_Data_FTSE100_
1985_21.csv").values[:, 0]
```

The variables for storing strings of the Cypher queries to add and match nodes and add edges are initialised. It should be noted that the "match_nodes_queries" variable is not actually used in this code snippet:

```
add_nodes_query = ""
match_nodes_queries = ""
create_edges_queries = ""
```

Then, two other variables are set up: $i = 0$, where the i variable will keep track of how many "add node" queries we have accumulated so far in the "add_nodes_query" variable during the for loop. Then: $row = 0$. The row variable keeps track of which row in our CSV data file we are on throughout the for loop.

For every date in the CSV: for date in "processed_input_csv_dates":

Append a new "add nodes" query to the "add_nodes_query" variable for the row and date we are currently on in the for loop.

```
add_nodes_query =
greeter.add_nodes_in_neo4j(onto, date, row,
add_nodes_query)
```

Increment the i and row for every loop iteration:

```
row += 1
i += 1
```

Once ten "add node" queries (i.e. add node queries for 10 time-slots) are stored in the "add_nodes_query" variable (i.e. this is signified when this condition is met: $(i/10 == 0)$, which means if $i/10$ results in a remainder of 0) then these 10 queries are executed on the database. The "add_nodes_query" variable is then reset to a blank string (i.e. ""). The 'for loop' then keeps going until the last date of the CSV file is reached.

Then the i and row variables are reset to 0:

```
i = 0
row = 0
```

Then we apply a process, similar to above, for the "add edge" queries. The edges are not added in batches of 10 time-slots like the nodes, as the add edge queries are more intensive than the add node queries: for date in processed_input_csv_dates:

```
greeter.add_annotation_edges_in_
neo4j(onto, date, row)
greeter.add_subclass_edges_in_
neo4j(onto, date, row)
row += 1
i += 1
```

Once all the nodes and edges for all time-slots have been added to the database, the connection to the database is closed.

To pass data from the *GDB* to other model components a separate Python file, *input_data_import_from_neo4j.py*, is used: it extracts the data from Neo4j to a new CSV file *Input_Data_from_Neo4j.csv*.

After the first iteration of the model, the system generates a CSV file, *weights_importance_uc2.csv*, with the weights of the model input indicators (see below how this file is created). After exporting and processing this file we form the next "layer" of the *GDB - Ontology full graph with weights* (see Definition III.1).

Without going into too much detail, we state that the code in the for loop assigns CSV values to the "Value" properties

[5]https://neo4j.com/blog/graphs-in-time-and-space/

[6]https://www.ons.gov.uk/

[7]https://finance.yahoo.com/quote/%5EFTSE?p=%5EFTSE

[8]https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html#user-guide

and the weight values to the ”Weight” properties (if the node’s name is present in the weight_columns list) of each node in the Neo4j database. We construct and execute a Cypher query that: finds the nodes in the database we want to update, then sets the ”Value” and ”Weight” properties of those nodes. We refer readers interested to see the code and the construction of this query to the GitHub repository [1].

The full Ontology of Market Index Prediction graph with weights, which built in Neo4j environment is given in our GitHub repository [1].

Next, based on the weights obtained, we can proceed with the *Feature Selection*. The Python file *cypher_queries_uc2.py* is responsible for it. Then, by querying indicators of the same category, we identify the one with the maximum weight. At the same time, Neo4j should display a graph containing only the indicators remaining after the selection. The data of this graph is then passed to the next component of the system.

According to Table I the Macroeconomic Indicators in *FPCM* Model are divided into five categories. For example, the feature selection Cypher query for Economic Output Indicators category should be present as:

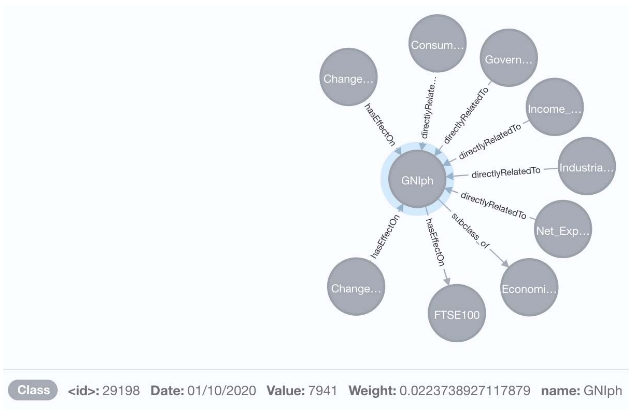
```
 economic_output_indicators_level_
 2_mw_node = session.run("""MATCH
 (n1:Class
 {name: "Economic_Output_Indicators_Level
 _2", Date: "01/10/2020"})<-
 [r1:subclass_of]-(scn1:Class)
    WITH max(scn1.Weight)
    AS maximum
    MATCH (n2:Class {name: "Economic_Output_
    Indicators_Level_2",
    Date: "01/10/2020"})<-[r2:subclass_of]-
    (scn2:Class)
    WHERE scn2.Weight = maximum
    RETURN scn2""").data()
...
eoi_l2_mw_node_name = economic_output_
indicators_level_2_mw_
node[0]["scn2"]
["name"]
...
max_weighted_features = [
    ...,
    ...,
    ...,
    eoi_l2_mw_node_name,
    ...]
```

The result of this query is shown in Fig.5.

Finally, the structured and selected Features (Ratios) are transferred to Python environment in the format *feature_selection_from_neo4j.csv*, which is served as input data for *MLE*.

D. Developing Machine Learning Engine: Hybrid Engine

As part of a computational model for FTSE 100 prediction, we developed a comprehensive Machine Learning Engine (*MLE*). It consists of 17 parallel *LSTM*s for the market index and 16 Macroeconomic Indicators, a Concatenation unit and the *LR* unit which analyses the *LSTM*s merged results



Class <id>: 29198 Date: 01/10/2020 Value: 7941 Weight: 0.0223738927117879 name: GNIph

Fig. 5. Neo4j Environment: Returning the Macroeconomic Indicator with the maximum weight in Economic Output Indicators category

(interested reader is referred to our GitHub repository [1] for detailed description of this Hybrid *MLE* for *FPCM*).

As the basis for the Python code, we use TensorFlow and Keras libraries, which allow to creating deep learning models quickly and easily.

a. Collecting the Data. For training the *LSTM*s, it is necessary to prepare a set of training data: examples of input data and their corresponding outputs.

As the training data, we use a monthly closing price and indexes values starting from January 1985. So, each *LSTM* and Macroeconomic Indicator input dataset has a vector form. The output of each LTSM is the particular feature’s values for the recent 60 months (the last date of the dataset is October 2020), also presented as a vector. Now we can compare the result with the real data to check the accuracy of *LSTM*s’ predictions.

We used *Office of National Statistics* official website and market indexes database on *Yahoo.Finance* to collect the data needed and save it in *Processed_Input_Data_FTSE100_1985_21.csv*; it contains 430 time-slots from January 1985 to October 2020 (see our GitHub repository [1]).

b. Design, Training and Quality Assessment of a Hybrid Machine Learning Engine

Developing the LSTMs. The first stage of creating a comprehensive *MLE* is developing a separate *LSTM* for each of the 17 features, plus one for the market index – FTSE 100. As mentioned before, one of the features of LSTM is that this approach can’t be used for processing multiple time-series input. So we have to build an LSTM for each of the features separately and then concatenate the results.

All 17 LSTMs have a similar code, and all of them, including the FTSE 100 LSTM, can be found in our GitHub repository [1].

The general algorithm we applied for LSTM construction is the following:

- Use the Pandas library to read the input CSV data file and extract a dataset of the values from all its rows of the first column.
- Set aside a small sample of the last values of the dataset for testing, add the remaining values to ”X_train” and ”y_train” arrays which will be used for training the LSTM.
- Convert the ”X_train” and ”y_train” into NumPy arrays.
- Reshape the ”X_train” array so it is suitable for use with the LSTM.
- Initialise the Input layer of the LSTM to match the dimensions of the ”X_train” array, so ”X_train” can be fed into the LSTM.
- Use TensorFlow library functions to add three layers to the LSTM, with 50 neurons per layer.
- Create an output layer with one neuron.
- Initialise the LSTM with the input and output layers specified in the earlier steps.
- Compile and run the LSTM.

Besides, we used SKLearn *MinMaxScaler*, and NumPy *reshape* functions to normalise and then denormalise the input data.

The model includes a stack of several fully connected network layers, each containing 50 units; the main *LSTM* layer carries an activation function - *LR*. Then, the model is compiled. We used the ’Mean Squared Error’ function as a loss function and ’ADAM’ (A Method for Stochastic Optimisation) [16] as a replacement optimisation algorithm. Finally, the model is trained using the .fit function (for FTSE 100 – with 50 epochs and batch size 20).

The *LSTM*s are saved as separate Python files: *lstm_FTSE_100.py*, *lstm_GDP.py*, etc.

The result of FTSE 100 *LSTM* prediction for 60 last months (5 years) is visualised in Fig. 6. The output graphs of the rest *LSTM*s can be found in our GitHub repository [1].

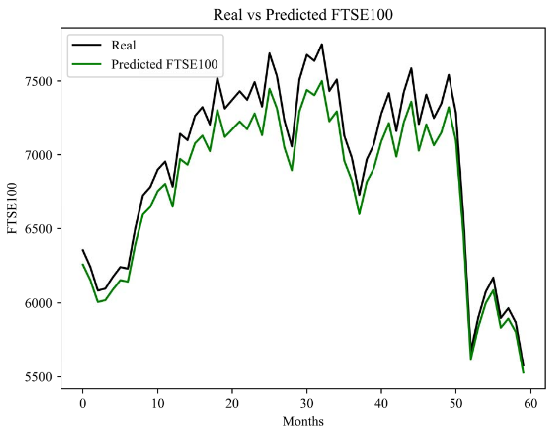


Fig. 6. FTSE 100 LSTM prediction

As we can see the LSTM-based predictions are consistently lower than the actual figures. The next step in the process, Linear Regression, aims to address this issue.

Developing Concatenated Output. The next unit of the *MLE* is Concatenation. Using this NumPy function, we present the output vectors of 17 *LSTM*s as one joint matrix – combined output. The Python code runs from a separate file *main.py*.

```
lstm_predicted_stock_prices = [
    lstm_FTSE_100.predicted_stock_price,
    ...
    lstm_HP.predicted_stock_price,
    lstm_PMI.predicted_stock_price]

number_of_lstms = len(lstm_predicted_
stock_prices)
combined_output = np.concatenate(lstm_
predicted_stock_prices, axis=1)
dataset_test = pd.read_csv('Processed_
Input_Data_FTSE100_1985_21.csv', header=0,
index_col=0)
real_stock_price = dataset_
test.iloc[0:, 0:1].values
real_stock_price = real_stock_
price[len(real_stock_price) -
testing_set_size:, 0:1]
combined_output = np.concatenate(
[combined_
output, real_stock_price], axis=1)
```

The combined output is saved as *combined_output.csv*, which will be used as an input data for the *LR* unit of *MLE*. This file contains

- 18 columns – *LSTM* Predicted FTSE100 index, 16 *LSTM* Predicted *MI*, Real FTSE100 index
- 60 rows (time-slots).

However, at this stage, the features are not yet reliant on each other.

Linear Regression. To calculate the joint result, the final most accurate prediction of FTSE 100, considering the other 16 features, we use Linear Regression (*LR*) and realise it in Python by means of SKLearn library. First, the combined output should be split into two parts. The first (the oldest) time-slots of 17 input features plus the actual values of FTSE 100 for the same time-slots are used as the training data for the *LR*. Accordingly, the LR analyses the dependency of the features from the actual FTSE 100 price. The last (the newest) time-slots, e.g., last 12 months, are the *LR* input data (for testing).

```
# we define dataset
combined_output = pd.read_csv('combined_
output.csv', header=0)
print(combined_output)
X, y = combined_output.iloc[0:testing_
set_size-12,
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16]].values,
combined_output.iloc[0:testing_
set_size-12,
-1].values
print(X)
# we define the model
```

```
model = LinearRegression()
# we fit the model
model.fit(X, y)
# we test prediction with real data
test_X = combined_output.iloc[testing_
set_size-12:,
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16]].values
prediction = model.predict(test_X)
print(f'Prediction: {prediction}')
```

The model is tested using Mean Absolute Percentage Error (*MAPE*) formula to compare the accuracy of the FTSE 100 index predicted by *LSTM* only and by *LR* using multiple features. Once *LR* passes the testing, the *MLE* is ready to use.

Calculating the Weights for the Feedback Loop. After *LR* code for *FPCM* Model (*linear_regression.py*) is executed the *feature importance analysis* of the input data should be made. For these purposes the *Random Forest Feature* method is used. Finally, the result is saved as (*weights_importance_uc1.csv*), which is further transferred to the *GDB* (as a part of the feedback loop).

V. Application: Market Index Prediction Computational Model for FTSE 100 index

Once the dataset needed for the analysis is collected, it is exported to the template Neo4j *GDB* as the ”Values” of the particular nodes. There is no need to create the *GDB* and *OBP* Ontology from scratch in every new case, as they represent the general framework for market index data pre-processing. Further, the data file extracted from Neo4j containing 430 time-slots of FTSE 100 values, plus 16 *MI* values (*Input_Data_from_Neo4j.csv*) should be imported to the Python environment (also see our GitHub repository [1]). After exporting these datasets to the *MLE* the following files should be run one-by-one:

- *main.py* – runs 17 *LSTM*s and generates a combined output
- *linear_regression.py* – runs LR using an *LSTM*s' combined output as input.

This model was tested using the *MAPE* formula to compare the accuracy of FTSE 100 index predicted by *LSTM* only and by *LR* using multiple features. The result of this testing (last 12 months) is presented in Fig.7 and 8.

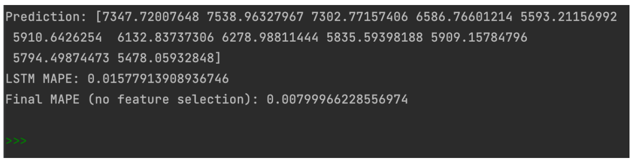
We can see that the complex hybrid *MLE* prediction (using 17 *LSTM*s and LR) is more accurate rather than the prediction results after *LSTM*.

After the *NN* result is received we can use *feature_importance_uc2.py* to evaluate the weights of the features. Its output file *weights_importance_uc1.csv* can be now added to Neo4j Graph by means of feedback loop.

The results of the Random Forest Feature Importance Analysis for *FPCM* are presented in Fig.9.

Then by means of Neo4j queries (*cypher_queries_uc2.py*), the weights are used to identify the 5 most valuable *MI* (one for each category of indicators). As a result, the model peaked the following ratios: CPI, I, IP, GNIph and UR. The usage of only five inputs will prevent the *NN* from overfitting in future, providing more accurate outcomes. To prove this, we run *LR* code using as an input the five selected *MI* only. The testing result (last 12 months) is presented in Fig.10 and 11.

After comparing the Final *MAPE* with feature selection (Fig.10) with the previous testing Final *MAPE* (Fig.7), we can state that the feature selection increased the efficiency of the *FPCM* Model.



```
Prediction: [7347.72007648 7538.96327967 7302.77157406 6586.76601214 5593.21156992
 5910.6426254  6132.83737306 6278.98811444 5835.59398188 5909.15784796
 5794.49874473 5478.05932848]
LSTM MAPE: 0.01577913908936746
Final MAPE (no feature selection): 0.00799966228556974


>>>
```

Fig. 7. Linear Regression (without Feature Selection) Results: Mean Absolute Percentage Error (Python Console)

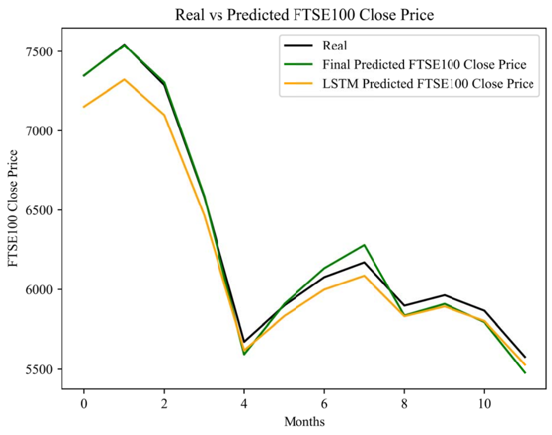


Fig. 8. Linear Regression without Feature Selection Results: the Plot

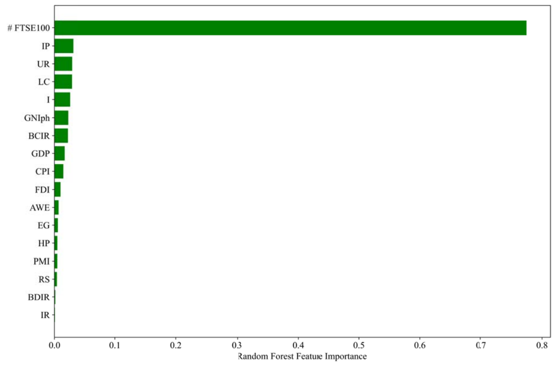
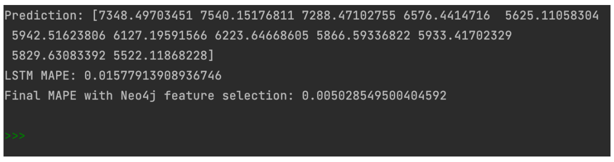


Fig. 9. Random Forest Feature Importance Analysis Result for FPCM features



```
Prediction: [7348.49703451 7540.15176811 7288.47102755 6576.4414716  5625.11058304
 5942.51623806 6127.19591566 6223.64668605 5866.59336822 5933.41702329
 5829.63083392 5522.11868228]
LSTM MAPE: 0.01577913908936746
Final MAPE with Neo4j feature selection: 0.005028549500404592


>>>
```

Fig. 10. Linear Regression with Neo4j Feature Selection Results: Mean Absolute Percentage Error (Python Console)

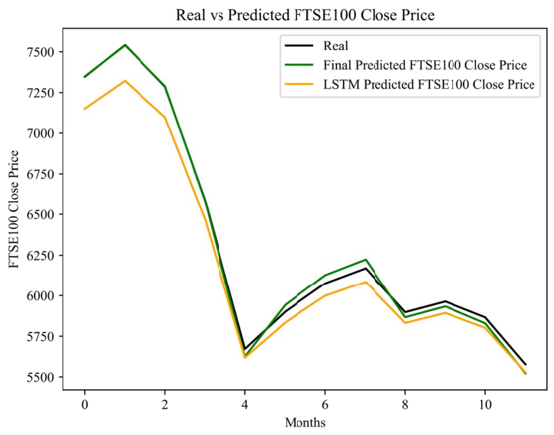


Fig. 11. Linear Regression with Feature Selection Results: the Plot

VI. Conclusions and Future Work

In this paper we refined a concept of the Generic Predictive Computational Model initially developed in [9] and [10] and applied it by building a Use Case for the FTSE 100 index forecasting. To adapt our model for this Use Case we have developed a special Market Index Prediction Ontology and a Hybrid Machine Learning Engine which is based on the use of multiple parameters related to predicting time-series data: the market index itself and the set of macroeconomic indices, used as additional external impact factors. Utilising LSTM, we presented a solution to overcome its known restriction which reduces LSTM's application to single input parameters only. In our solution we developed a novel Machine Learning Engine comprising several parallel LSTMs, a Concatenation unit, which merges the LSTMs' outputs (into a time-series DataFrame), and a Linear Regression Unit, which produces the final result.

Thus, the main scientific contributions of the paper are the following refinements of the concepts that formed the pipeline of the Generic Predictive Computational Model initially introduced in [9] and [10]: (1) the concept of GDB component is now defined with a third layer (weights); (2) the model is improved by the Feedback Loop; (3) the MLE component which utilises the combination of LSTM, able to work with multiple parameters, and LR which improves the accuracy of the model. Our results also demonstrate that being implemented in this way, an LSTM is more efficient than when it is applied as a stand-alone method (which is typically the case).

In combination with the results of [9] and [10] this paper introduces a novel generic framework for data analysis and prediction, presenting 'building blocks' for the construction of the predictive model. In future work we will develop a new use case tackling a different, not related to finance, dataset. The other direction of future research is the comparison of the obtained results with other tools used for FTSE 100 predictive calculation.

VII. Acknowledgement

We are grateful to *Dr Gabriele Pierantoni* and anonymous reviewers for their useful comments and suggestions.

References

[1] N. Yerashenia, A. Bolotov, and D. Chan, "Generic predictive computational model GPCM, GitHub repository." https://github.com/Yerashenia/Generic-Predictive-Computational-Model-GPCM, 2021.

[2] E. Tölö, "Predicting systemic financial crises with recurrent neural networks," *Journal of Financial Stability*, vol. 49, 2020.

[3] Y. Jang, I. Jeong, and Y. K. Cho, "Business failure prediction of construction contractors using a LSTM RNN with accounting, construction market, and macroeconomic variables," *Journal of management in engineering*, vol. 36, no. 2, 2020.

[4] C. L. Cocianu and M. Avramescu, "Financial data forecasting using recurrent neural networks," *Proceedings of the 18th International Conference on Informatics in Economy*, 2019.

[5] Y. Hu, A. Huber, J. Anumula, and S.-C. Liu, "Overcoming the vanishing gradient problem in plain recurrent networks," *arXiv*, 2018.

[6] M. Vochozka, J. Vrbka, and P. Suler, "Bankruptcy or success? the effective prediction of a company's financial development using LSTM.," *Sustainability*, vol. 12, no. 18, 2020.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[8] S. Edet, "Recurrent neural networks in forecasting S&P 500 index," *Available at SSRN 3001046*, 2017.

[9] N. Yerashenia and A. Bolotov, "Computational modelling for bankruptcy prediction: Semantic data analysis integrating graph database and financial ontology," in *2019 IEEE 21st Conference on Business Informatics (CBI)*, vol. 1, pp. 84–93, IEEE, 2019.

[10] N. Yerashenia, A. Bolotov, G. Pierantoni, and D. Chan, "Semantic data pre-processing for machine learning based bankruptcy prediction computational model," in *2020 IEEE 22nd Conference on Business Informatics (CBI)*, IEEE, 2020.

[11] J. Pokornỳ, "Graph databases: their power and limitations," in *IFIP International Conference on Computer Information Systems and Industrial Management*, pp. 58–69, Springer, 2015.

[12] A. Afify and H. E. Roman, "Estimating market index valuation from macroeconomic trends," *Quantitative Finance and Economics*, vol. 5, no. 2, pp. 287–310, 2021.

[13] D. Pilinkus, "Macroeconomic indicators and their impact on stock market performance in the short and long run: the case of the baltic states," *Technological and Economic Development of Economy*, no. 2, pp. 291–304, 2010.

[14] B. Weng, W. Martinez, Y.-T. Tsai, C. Li, L. Lu, J. R. Barth, and F. M. Megahed, "Macroeconomic indicators alone can predict the monthly closing price of major us indices: Insights from artificial intelligence, time-series analysis and hybrid models," *Applied Soft Computing*, vol. 71, pp. 685–697, 2018.

[15] D. Shah, H. Isah, and F. Zulkernine, "Stock market analysis: A review and taxonomy of prediction techniques," *International Journal of Financial Studies*, vol. 7, no. 2, p. 26, 2019.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv*, 2014.