
Production Systems and Information Engineering
Volume 10 (3), pp. 82–97
doi: 10.32968/psaie.2022.3.7

NO-CODE PLATFORM FOR TIME SERIES PREDICTION

Youcef Guichi
University of Miskolc, Hungary
Institute of Information Technology
youcef.guichi.contact@gmail.com

Erika Baksáné Varga
University of Miskolc, Hungary
Institute of Information Technology
vargaerika@iit.uni-miskolc.hu

Abstract. Machine learning platforms that provide no-code capabilities allow for users to use machine learning without advanced domain knowledge or training. These tools enable those having no formal expertise in software development to create machine learning applications. Our analysis of existing no-code ML platforms reveals that most of them offer core ML, but not deep learning network models and can not be accessed for free. Therefore we have created a web-based application that allows researchers to experiment with machine learning solutions concerning time series prediction without the need for coding. This paper presents a case study demonstrating the operation of the platform.

Keywords: no-code platform, time series prediction, RNN models

1. Introduction

There are thousands of different programming languages that exist. Although they may share some common features, they were originally created for different application areas or for different programming styles. Machine learning (ML) is a special application area yielding the creation of specific programming languages. These are definitely more appropriate for machine learning tasks than others. Still, it is a question to choose from them when solving a business problem. For instance, machine learning engineers often use Python for NLP problems, while they prefer R or Python for sentiment analysis tasks, and are likely to use Java for security and threat detection [1].

On the other hand, no-code development is a recent approach that allows non-technical users to solve problems through visual modeling tools and configuration. The use of no-code platforms require less formal programming

expertise, which helps close the IT gap that many companies face [2]. No-code ML platforms deploy ML models, so analysts have the power of quick data predictions. The most widely used no-code ML platforms that can be applied for business purposes are as follows.

- BigML: an open-source web-based platform that offers machine learning and application integration services to businesses. For time series analysis it automatically selects the best model that fits the data.
- CreateML: a no-code drag and drop platform available for iOS developers to create and train custom machine learning models, or to use pre-trained templates on Mac.
- Data Robot: a popular cloud-based end-to-end enterprise AI platform for the fast and easy deployment of accurate predictive models. It enables business analysts to build predictive analytics without knowledge of ML or programming. It aids in the preparation, development, deployment, monitoring, and maintenance of enterprise-scale AI applications.
- Google AutoML: a no-code cloud platform including different types of data and covering a broad range of use cases from computer vision and video intelligence to NLP and translation.
- Obviously AI: a platform that uses state-of-the-art NLP to perform complex tasks on user-defined CSV data. It can be used by marketers and business owners who want to forecast revenue flow, optimize business processes, build a more effective supply chain, and conduct personalized automated marketing campaigns.

Table 1 summarizes some properties of these no-code ML platforms. The studied features are the capability for time series analysis (TSA), the inclusion of deep learning models (DLM), the possibility of customizing the selected model, whether we should install the software on our computer and whether we should pay for it. As can be seen, most of them offer core ML, but not deep learning network models and can not be accessed for free.

Table 1. Features of existing no-code platforms

	TSA	**DLM**	**Customization**	**No-install**	**Free**
BigML	X	X		X	
CreateML			X	X	
Data Robot	X		X	X	
Google AutoML			X	X	
Obviously AI	X		X	X	X

Based on our study, we conclude that there is a need for a free no-code ML platform that integrates deep learning models for business purposes. Our aim

is to develop a web-based platform that allows researchers to deliver end-to-end machine learning solutions without the need for coding, allowing them to focus more on analysis, making the process faster and more efficient. As a first approach, we have created a platform that can serve as support for solving time series prediction tasks.

2. Time series

A time series can be considered as a collection of observations of well-defined data items that are gained from repeated measurements over time. For example, recording the value of exchange rates each week of the year would comprise a time series. This is because the collected data are well defined, and consistently measured at equally spaced intervals [3].

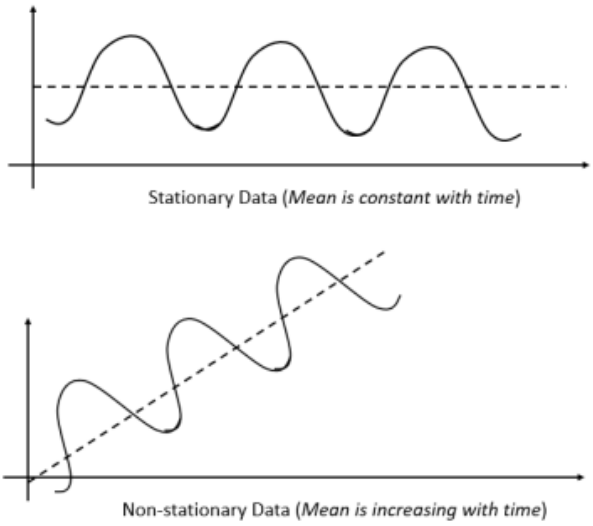

Figure 1. Stationary and non-stationary time series [5]

Time series analysis is a method for examining the changes of a variable or a combination of variables over a period of time. It helps organizations understand the underlying causes of trends or regular patterns over time. Data visualization techniques support business users to follow seasonal trends and dig deeper into why these trends occur. Time series forecasting can be used to predict the likelihood of future events by showing likely changes in the data [4].

Time series have mainly two types. In stationary time series the data have constant mean, variance and covariance. While non-stationary data are constantly fluctuating over time or are affected by time, that is these time series have volatile mean, variance and covariance as shown in Figure 1.

An observed time series can be decomposed into three components: the trend (long term direction), the seasonal (systematic, usually calendar related movements) and the irregular (unsystematic, short term fluctuations).

The trend is the long term movement in a time series. The seasonal component consists of effects that are reasonably stable with respect to timing, direction and magnitude. It arises from systematic, calendar related influences. Seasonality in a time series can be identified by regularly spaced peaks and troughs which have a consistent direction and approximately the same magnitude. There is a so called cyclic pattern, which can be confused with seasonal behaviour. A cycle occurs when the data exhibit rises and falls that are not of a fixed frequency. The duration of these fluctuations is usually at least 2 years.

The irregular component is what remains after the seasonal and trend components of a time series have been estimated and removed. It results from short term fluctuations in the series which are neither systematic nor predictable [3].

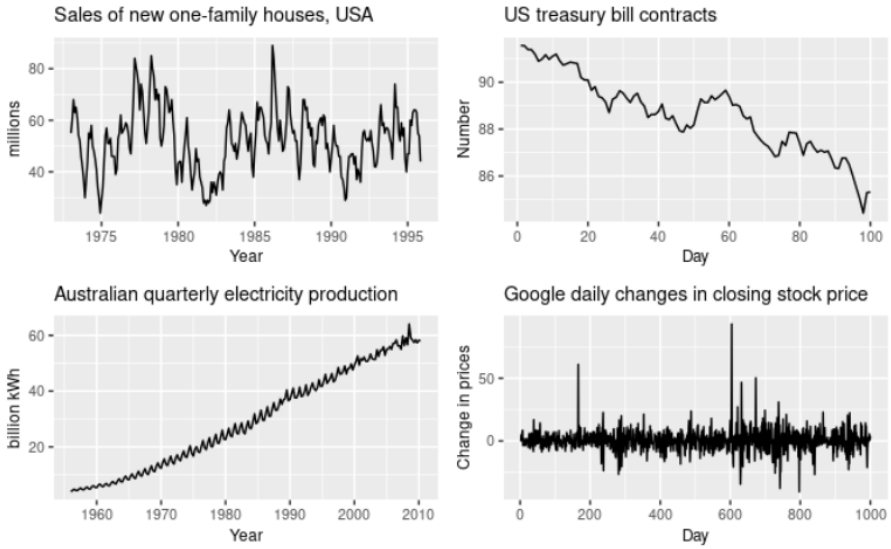

Figure 2. Time series examples [6]

In practice, however, many time series include a mixture of trend, cycles and seasonality. When choosing a forecasting method, the first task is to

identify the time series patterns in the data, and then choose a method that is able to capture the patterns properly. The examples from [6] show different combinations of the components in Figure 2. The first diagram of the monthly housing sales exposes strong seasonality with cyclic behaviour; the Australian quarterly electricity production has an increasing trend with seasonality; the US treasury bill contracts have a decreasing trend with no seasonality; while the daily change in the Google closing stock price shows neither trend, no seasonality or cyclic behaviour. In this case there are no patterns that would help us developing a forecasting model.

3. Methods of time series prediction

Time series analysis involves developing models to gain an understanding of the data. Then time series forecasting is the next step to fit the developed model to historical, time-stamped data in order to predict future values.

Traditional types of forecasting methods include decompositional, smooth-based, moving average, and exponential smoothing techniques. Decompositional methods split a time series into deterministic (i.e. predictable) and non-deterministic components. Data smoothing data removes random variation and outliers, and shows underlying trends and cyclic components. The moving-average model is a common approach for modeling univariate time series. It specifies that the output variable depends linearly on the current and various past values of a stochastic (i.e. imperfectly predictable) term, thereby yielding a stationary model. The main difference between the moving-average model and the exponential smoothing technique is that in the moving-average model the past observations are weighted equally, while exponential smoothing functions assign exponentially decreasing weights to the observations over time [7]. In practice, there are more sophisticated models, like ARIMA and TBATS, that combine some of the above approaches.

Regression models can also be applied in time series forecasting. In the simplest case, we assume that the time series of interest (y) has a linear relationship with another time series (x). In the example shown in Figure 3, the growth rates of personal consumption expenditure (y) and personal income (x) are displayed together with the fitted regression line which has a positive slope. In this case, the slope coefficient means that 1 percentage point increase in personal income results an average increase of 0.28 percentage points in personal consumption expenditure.

More advanced forecasting methods include artificial neural networks, which allow for complex non-linear relationships between the response variable and its predictors. A neural network can be considered as a network of neurons

which are organized in layers. The predictors (or inputs) form the bottom layer, and the forecasts (or outputs) form the top layer. In the simplest case, a neural network (NN) having only input and output layers corresponds to linear regression. For more complex and non-linear forecasting tasks, hidden layers must also be added to the NN architecture. With time series data, lagged values of the time series can be used as inputs to a neural network. This is called neural network autoregression (NNAR) model, which needs two input parameters: the number of lagged input values and the number of nodes in the hidden layer. When there is no hidden layer (i.e. the second parameter is 0), the model is equivalent with the ARIMA (AutoRegression Integrated Moving-Average) model [6].

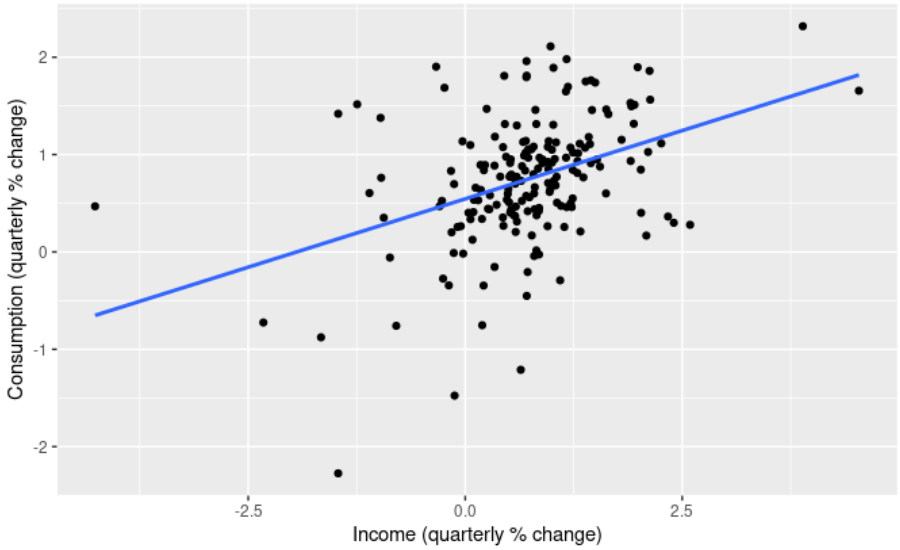

Figure 3. Forecasting consumption vs income with linear regression [6]

Although Convolutional Neural Networks (CNNs) are designed to efficiently handle image data, the ability of CNNs to learn and automatically extract features from raw input data can be applied to time series forecasting problems as well. A sequence of observations can be treated like a one-dimensional image that a CNN model can read and learn a representation that is most relevant for the prediction problem. It supports multivariate input, multivariate output and learning arbitrary but complex functional relationships [8].

Recurrent Neural Networks (RNNs) add the explicit handling of order between observations when learning a mapping function from inputs to outputs. Unlike a traditional NN, where the output is solely dependent on the input

values, the recurring aspect yields that the calculation at time t is based on the information provided at time $t-1$. See the architecture in Figure 4.

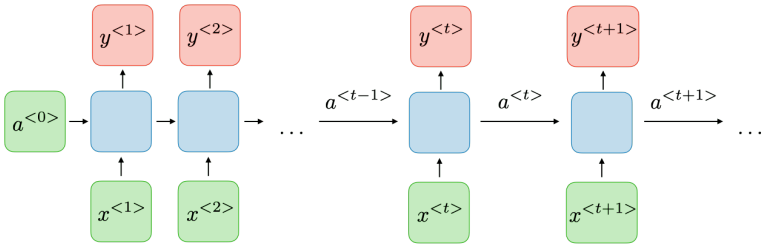

Figure 4. Recurrent Neural Network architecture [9]

Although RNNs are more convenient for processing sequential input than a traditional NN design, they are extremely difficult to train to handle long-term dependencies due to the multiplicative gradient that can exponentially increase or decrease with respect to the number of layers. In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs.

RNNs using LSTM units [10] solve the problem of vanishing gradient, but still suffer from the exploding gradient problem. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate [11]. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. LSTM networks are well-suited to making predictions based on time series data, since there can be lags of unknown duration between important events in a time series and this model is relatively insensitive to gap length.

Gated Recurrent Units (GRUs) [12] are a gating mechanism in RNNs, but with less parameters than LSTMs. They have several variants from fully gated units incorporating update and reset gates, to minimal gated units where the new gates are merged into a single forget gate. Since GRUs have been shown to yield better performance on smaller and less frequent datasets than LSTM [13], we decided to include them in our experimental no-code time series prediction platform.

4. The Tune-Quick platform

Tune-Quick is a web-based platform designed to help in the creation of end-to-end machine learning solutions. The application can be downloaded from *github.com/YoucefGuichi/tune_quick*. Figure 5 explains what is Tune-Quick at system level. Users can provide data and set model parameters through the GUI of the platform. The platform applies a queue structure to collect input data that are waiting for processing. In Python, the queue module

implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements a FIFO queue with all the required locking mechanisms.

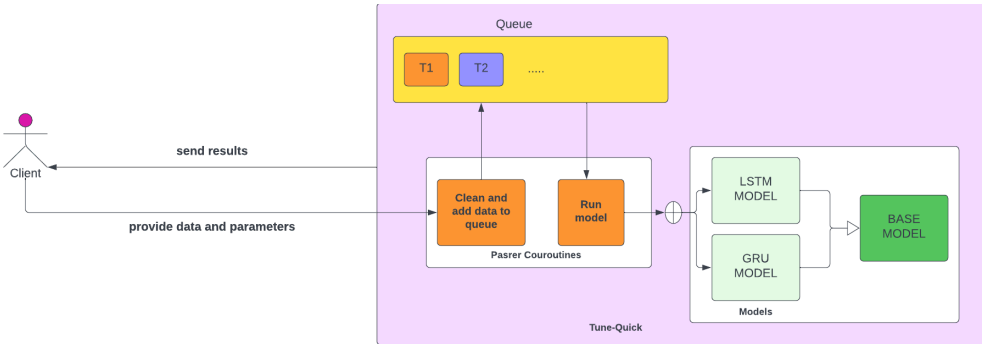

Figure 5. Tune-Quick system architecture

The input received is parsed by two coroutines of the Parser class. The reason for the use of the queue is to make the coroutines independent of each other. Whenever a user submits a dataset and a model, it will be sent to the queue. Many users can do this at the same time, so each one will wait for his turn. The main steps, that are executed, are listed as follows (see Figure 6).

1. Create and initialize the queue.
2. Create a task for the Add model coroutine.
3. Create a task for the Run model coroutine.
4. Gather the tasks that will run concurrently.

```
async def main(self):

    queue = asyncio.Queue()
    task1 = asyncio.create_task(self.add_model_with_dataset_to_queue(queue))
    task2 = asyncio.create_task(self.run_model(queue))
    await asyncio.gather(*[
        task1, task2]
    )
```

Figure 6. The Main coroutine of the application

The Add data to queue routine (listed in Figure 7) will run only if the request method is POST. It stores the data entered by the user in the dataset variable, and selects the suitable algorithm from the algorithms dictionary (which is LSTM or GRU in the present stage of the experiment). After reading the data

and choosing the correct model, the coroutine will send them to the queue as a task to be executed.

```
async def add_model_with_dataset_to_queue(self, queue):
    try:
        if self.request.method == "POST":
            dataset = pd.read_csv(
                io.StringIO(self.request.files["csv-file"].stream.read().decode("UTF8"), newline=None))
            seq = Sequential()
            model = self.algorithms[self.request.form["algorithm"]](dataset, seq)
            queue.put_nowait(model)
    except Exception as err:
        raise err
```

Figure 7. Add model to queue coroutine

The tasks waiting for processing in the queue will be picked up by another process, the so called Run model coroutine (see its code in Figure 8). This triggers the functions associated with each model in the following order.

1. Clean and prepare the dataset for the model.
2. Split the data into train and test sets.
3. Start training. It takes the parameters, such as the number of epochs, the type of optimizer, and batch size, from the user input.
4. Predict the results.
5. Plot predictions and send the results to the front end of the platform.

```
async def run_model(self, queue):
    if self.request.method == "POST":
        # get model from the queue
        model = await queue.get()
        logger.info('cleaning ans preparing the data...')
        try:
            model.clean_and_prepare_dataset()
            model.split_data()
        except Exception as err:
            raise err

        logger.info('training started...')
        model.train(
            optimizer=self.request.form["optimizer"],
            loss=self.request.form["loss-function"],
            epochs=int(self.request.form["epochs"]),
            batch_size=int(self.request.form["batch-size"]))
        logger.info('training done')
        model.predict()
        self.chart = model.plot_predictions(self.request.form["chart-title"], self.request.form["x-axis"],
                                            self.request.form["y-axis"])
```

Figure 8. Run model coroutine

The Tune-Quick platform is designed to integrate multiple models. Technically, it has a Model base class which defines the common attributes and

functionalities for all NN models. The derived classes contain specific methods, namely the train function which differs from one model to another and takes parameters from the user input. At present, the platform is at a test phase, so there is only two models implemented in separate classes as shown in the UML class diagram in Figure 9.

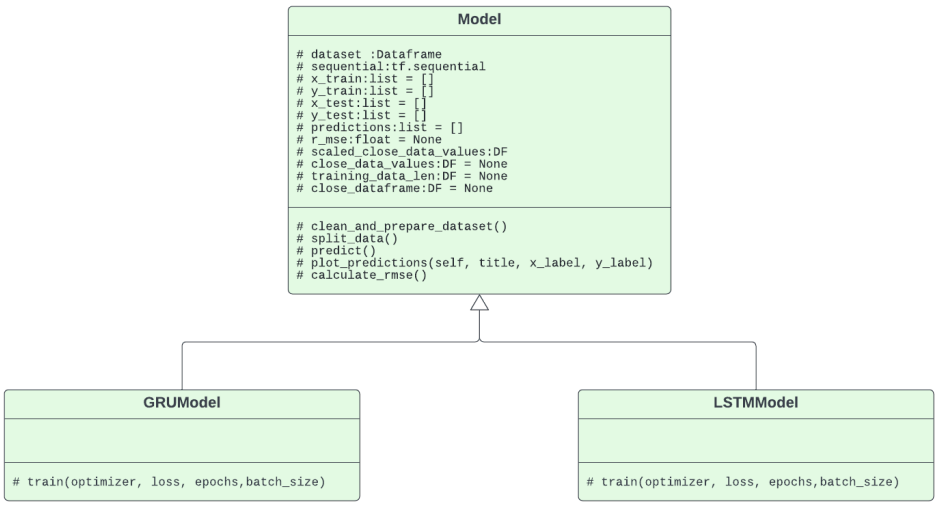

Figure 9. UML class diagram of the implemented models

The first method cleans and prepares the data so that they fit to the predictive models. This process consists of the following steps (see Figure 10).

1. Drop duplicates.
2. Drop nan values.
3. Transform the date column from text to date type.
4. Set the index to date because the data is a timeseries.

```
def clean_and_prepare_dataset(self):
    """transfer the data to the correct format"""
    self.dataset.drop_duplicates(inplace=True)
    self.dataset.dropna(inplace=True)
    self.dataset["Date"] = pd.to_datetime(self.dataset["Date"])
    self.dataset["Date"] = self.dataset["Date"].dt.date
    self.dataset.set_index("Date", inplace=True)
    logger.info("validation done")
```

Figure 10. Steps of data preparation

The second function splits the data into train and test sets. This consists of the next steps (see Figure 11).

1. Get the data to be used in the training phase.
2. Take 95% from the actual data as training data and the rest for the test set.
3. Scale the data which is very important for the training, especially for avoiding local optima.
4. Split the train set into x_train and y_train.
5. Then turn both subsets to an array because the NN models expect arrays as input.
6. Reshape the arrays to suit RNNs.

```
def split_data(self):
    """split the data into x_train and y_train and scale it"""
    # Create a new dataframe with only the 'Close column
    self.close_dataframe = self.dataset.filter(['Close'])
    self.close_data_values = self.close_dataframe.values

    # Get the number of rows to train the model on
    self.training_data_len = int(np.ceil(len(self.close_data_values) * .95))
    self.scaled_close_data_values = scaler.fit_transform(self.close_data_values)
    train_data = self.scaled_close_data_values[0:int(self.training_data_len), :]

    # Split the data into x_train and y_train data sets
    for i in range(60, len(train_data)):
        self.x_train.append(train_data[i - 60:i, 0])
        self.y_train.append(train_data[i, 0])
    self.x_train, self.y_train = np.array(self.x_train), np.array(self.y_train)
    self.x_train = np.reshape(self.x_train, (self.x_train.shape[0], self.x_train.shape[1], 1))
```

Figure 11. Splitting the data into train and test sets

After training the model, the next phase is to calculate the predictions based on the test set (see Figure 12).

1. Take the rest (5%) of the data.
2. And repeat the same process as when splitting the training data.

```
def predict(self):
    """calculate the predictions"""
    test_data = self.scaled_close_data_values[self.training_data_len - 60:, :]
    self.y_test = self.close_data_values[self.training_data_len:, :]
    for i in range(60, len(test_data)):
        self.x_test.append(test_data[i - 60:i, 0])
    x_test = np.array(self.x_test)
    self.x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
    self.predictions = self.sequential.predict(x_test)
    self.predictions = scaler.inverse_transform(self.predictions)
```

Figure 12. Making predictions for the test values

The final step is to plot the prediction and convert it to HTML which can be displayed in the front end. This function involves the next stages (see the list in Figure 13).

1. Get the training data.
2. Get the validation data.
3. Plot the predicted values.
4. Convert the diagram to HTML.

```
def plot_predictions(self, title, x_label, y_label):
    """plot predictions and convert it to html elements"""
    train = self.close_dataframe[:self.training_data_len]
    valid = self.close_dataframe[self.training_data_len:]
    valid['Predictions'] = self.predictions
    fig, ax = plt.subplots(figsize=(16, 6))
    ax.set_title(title, fontsize=15)
    ax.set_xlabel(x_label, fontsize=15)
    ax.set_ylabel(y_label, fontsize=15)
    ax.plot(train['Close'])
    # ax.plot(valid[['Close']])
    ax.plot(valid['Predictions'], linestyle="dashed")
    ax.legend(['Train', 'Predictions'], loc='lower right')
    plt.show()
    chart = mpld3.fig_to_html(fig)
    return chart
```

Figure 13. Plotting the predicted values

In order to ensure that the results are valid, we also calculate the root mean square error (RMSE) using the function in Figure 14.

```
def calculate_rmse(self):
    """calculate root mean square error"""
    self.r_mse = np.sqrt(np.mean(((self.predictions - self.y_test) ** 2)))
```

Figure 14. Calculation of RMSE

In the training phase, the model specific methods are executed. When the GRU model is initiated, the following procedure runs (see Figure 15).

1. Define 3 layers. In each layer, except for the output layer, there are 50 GRU units. The last layer is a dense layer where the final output is produced.
2. Define a dropout of 0.2 which skips some nodes to help reduce overfitting.
3. Use mean squared error (MSE) as loss function and keep the default Adam optimizer. The role of the optimizer is to help in reaching the local optima of the problem very quickly.
4. Use 20 as the default value for the number of epochs.

The LSTM architecture is very similar to the GRU. The steps of model creation are listed in Figure 16.

```python
class GRUModel(Model):
    def __init__(self, dataset, sequential):
        super().__init__(dataset, sequential)

    def train(self, optimizer: str, loss='mean_squared_error', epochs=20,
              batch_size=32):
        self.sequential.add(GRU(units=50, return_sequences=True, input_shape=(self.x_train.shape[1], 1),
                                activation='tanh'))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(GRU(units=50, return_sequences=True, input_shape=(self.x_train.shape[1], 1),
                                activation='tanh'))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(GRU(units=50, return_sequences=True, input_shape=(self.x_train.shape[1], 1),
                                activation='tanh'))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(GRU(units=50, activation='tanh'))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(Dense(units=1))
        self.sequential.compile(optimizer=optimizer,
                                loss=loss)

        # fitting the model
        self.sequential.fit(self.x_train, self.y_train, epochs=epochs, batch_size=batch_size)
```

Figure 15. Creating a GRU model

```python
class LSTMModel(Model):

    def __init__(self, dataset, sequential):
        super().__init__(dataset, sequential)

    def train(self, optimizer: str, loss='mean_squared_error', epochs=1, batch_size=32):
        self.sequential.add(LSTM(units=50, return_sequences=True, input_shape=(self.x_train.shape[1], 1)))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(LSTM(units=50, return_sequences=True))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(LSTM(units=50, return_sequences=True))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(LSTM(units=50))
        self.sequential.add(Dropout(0.2))
        self.sequential.add(Dense(units=1))
        self.sequential.compile(optimizer=optimizer, loss=loss)
        self.sequential.fit(self.x_train, self.y_train, epochs=epochs, batch_size=batch_size)
```

Figure 16. Creating a LSTM model

5. Results

As a case study, we have uploaded stock market related time series data (downloaded from *finance.yahoo.com/quote/IBM/history* for the time period

01/01/2006 - 01/01/2018). Figure 17 shows a few sample records from the dataset.

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Volume	Name
2	1/3/2006	211.47	218.05	209.32	217.83	13137450	GOOGL
3	1/4/2006	222.17	224.7	220.09	222.84	15292353	GOOGL
4	1/5/2006	223.22	226	220.97	225.85	10815661	GOOGL
5	1/6/2006	228.66	235.49	226.85	233.06	17759521	GOOGL
6	1/9/2006	233.44	236.94	230.7	233.68	12795837	GOOGL

Figure 17. Sample data

Tune-Quick takes the Close column, which refers to the last traded price, as the input for the prediction. After the following parameters are set (see Figure 18), we get the result presented in Figure 19.

- CSV data file
- NN predictive model: LSTM
- Loss function: mean_squared_error
- Optimizer: Adam
- Number of epochs: 10
- Batch size: 16
- Title for the resulting chart: IBM Stock Prices Prediction
- Caption for x axis: Dates
- Caption for y axis: Price (USD)

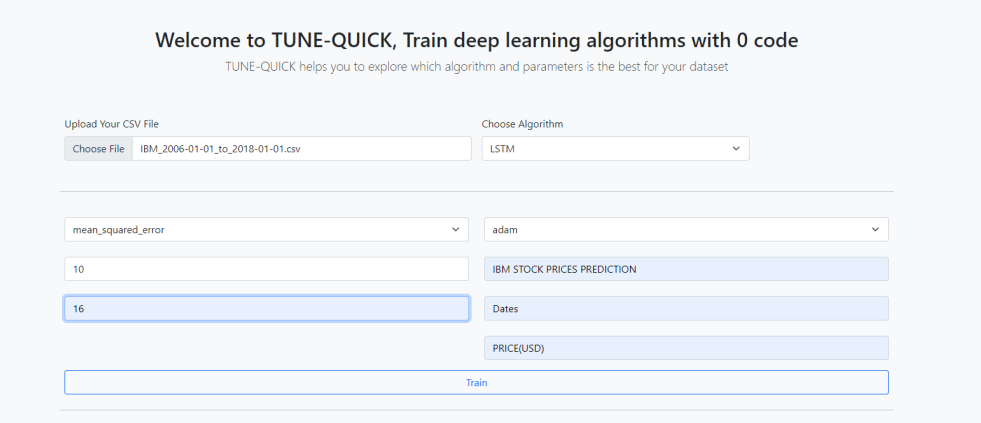

Figure 18. Tune-Quick front end with sample data

The result (see Figure 19) is displayed for the user as a chart that contains the training data in blue, as well as the predicted values in red.

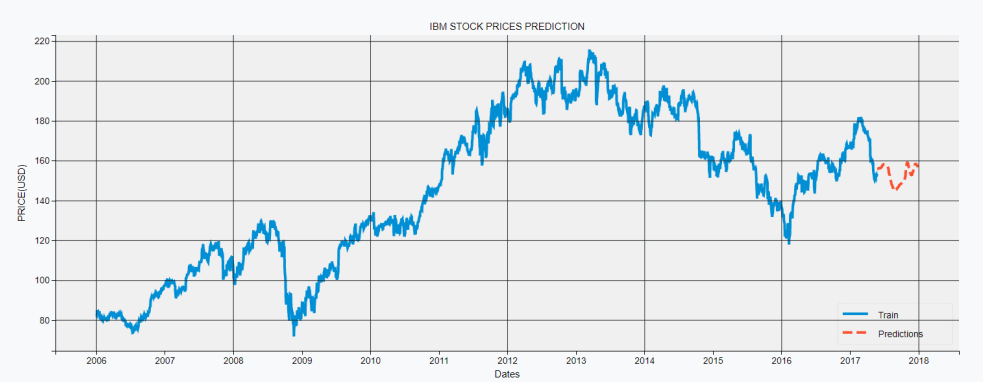

Figure 19. Case study result: stock price prediction

6. Summary

Machine learning platforms that provide no-code capabilities enable users to use machine learning without advanced domain knowledge or training. These tools allow for those with no formal expertise in software development to utilize machine learning without writing any code. In other words, no-code ML development is a shorter process than traditional ML development, because after defining the task and collecting the data, experts can directly upload the data and set the parameters for training the model and then they can immediately evaluate the results. So the benefits of no-code platforms are straightforward for data analysts: they can focus on the analysis, which in turn will be a faster and more efficient procedure for them.

We have studied the most widely used no-code ML platforms that can be applied for business purposes, and concluded that there is a need for a free no-code ML platform that integrates deep learning models. Therefore we have created a web-based application that allows researchers to experiment with machine learning solutions without the need for coding. Section 4 introduces the capabilities of the Tune-Quick platform. In its present experimental stage, two RNN predictive models (LSTM and GRU) are available for time series prediction tasks. We have shown the operation of the platform by means of a case study with IBM historical stock prices in Section 5. The results prove that the developed application can be applied for uploading the data and setting the parameters without any effort. The selected model is generated and the presented chart can be easily interpreted by experts, so it can be useful tool for data analysts.

References

[1] Gupta, S.: *What is the best language for Machine Learning?* Springboard: October 6, 2021. Available at: www.springboard.com/blog/data-science/best-language-for-machine-learning (Accessed: 14 September 2022)

[2] Creatio: *Why low-code & no-code development is a must for business growth?* Creatio: 2020. Available at: www.creatio.com/page/no-code (Accessed: 14 September 2022)

[3] Australian Bureau of Statistics: *Time Series Analysis: The Basics.* Available at: www.abs.gov.au/websitedbs/d3310114.nsf/home/\time+series+analysis:+the+basics (Accessed: 14 September 2022)

[4] Tableau: *Time Series Analysis: Definition, Types, Techniques, and When It's Used.* Available at: www.tableau.com/learn/articles/time-series-analysis (Accessed: 14 September 2022)

[5] Palachy, S.: *Stationarity in time series analysis* Towards Data Science: Apr 8, 2019. Available at: towardsdatascience.com/stationarity-in-time-series-analysis-90c94f27322 (Accessed: 14 September 2022)

[6] Hyndman, R. J., Athanasopoulos G.: *Forecasting: Principles and Practice.* OTexts: 2018.

[7] Dix, P.: *Why Time Series Matters for Metrics, Real-Time and Sensor Data* An InfluxData Case Study. InfluxData: 2020. Available at: www.influxdata.com/time-series-forecasting-methods (Accessed: 27 September 2022)

[8] Brownlee, J.: *Deep Learning for Time Series Forecasting - Predict the Future with MLPs, CNNs and LSTMs in Python.* Machine Learning Mastery: 2022.

[9] Amidi, A., Amidi S.: *Recurrent Neural Networks.* Available at: stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks (Accessed: 27 September 2022)

[10] Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Computation*, 9(8):1735-1780, 1997. doi:https://doi.org/10.1162/neco.1997.9.8.1735

[11] Gers, F.A., Schmidhuber, J., Cummins F.: Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451-2471, 2000. doi:https://doi.org/10.1162/089976600300015015

[12] Cho, K., van Merrienboer, B., Bahdanau, D., Bengio, Y.: *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches.* arXiv:1409.1259, 1994.

[13] Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* arXiv:1412.3555, 1994.