Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Project of an application based on machine learning for stock market prediction
Code

Vladislav Sorokin

student record book number 295462

Dzianis Harbatsenka

student record book number 295452

Nikita Zakharov

student record book number 295464

thesis supervisor
prof. PW dr hab. inż., Jerzy Balicki

Warsaw 2021

Version 1.0

Warsaw, 24.11.20

DECLARATION

We declare that this piece of work which is the basis for recognition of achieving learning outcomes in the Group Project course was completed on our own.

SVR

Function to train SVR

```
function [dates, y, fxpr, Bestmse] = train_svr(dtname,
maxItr, file_name,isAutoHyper,...

kernelFunctionName,MaxEvalNum,epsilon,loopMaxKernelScale,
loopMaxBoxConstraint)
% train_svr("PATH_TO_DATA\WIG20_TRAIN.csv", 10)

data = readtable(dtname,'Format','%{yyyy-MM-
dd}D%d%d%d%d%d');
N=height(data);
if isAutoHyper
    trainingPoints = N - 5;
else
    if N <= 35
       trainingPoints = N - 6;
       testingPoints = 4;
    elseif N <= 95
       trainingPoints = N - 15;
       testingPoints = 10;
    else
       trainingPoints = N - 15;
       testingPoints = 10;
    end
    xpr = double(data{trainingPoints + 1:trainingPoints +
testingPoints,2});
    ypr = double(data{trainingPoints + 1:trainingPoints +
testingPoints,5});
    datespr = data{trainingPoints + 1:trainingPoints +
testingPoints,1};

end

x = double(data{1:trainingPoints,2});
y = double(data{1:trainingPoints,5});
dates = data{1:trainingPoints,1};

% Algorithm
if (isAutoHyper == true) %Using matlab cross validation
    BestMdl =
fitrsvm(x,y,'KernelFunction',kernelFunctionName,'Optimize
Hyperparameters','all',...

'HyperparameterOptimizationOptions',struct('AcquisitionFu
nctionName',...
        'expected-improvement-plus',
'MaxObjectiveEvaluations', MaxEvalNum),...
```

```
        'IterationLimit',maxItr);
    fxpr = predict(BestMdl, x);
    Bestmse = norm(y-fxpr)^2/N;
else % iterate through and check best parameters
    %    Matlab          sklearn.svm.SVR
    %KernelScale   -        gamma
    %BoxConstraint -          C
    BestKernelScale = 1e-9; %for debug
    KernelScale = 1e-9;
    BoxConstraint = 1;
    BestBoxConstraint = 1;%for debug
    loopCountKernelScale = 1;
    loopCountBoxConstraint = 1;
    Mdl =
fitrsvm(x,y,'KernelFunction',kernelFunctionName,...
        'IterationLimit',maxItr,'Epsilon',
epsilon,'Standardize',true);
    BestMdl = Mdl;
    fxpr = predict(Mdl, xpr);
    Bestmse = norm(ypr-fxpr)^2/testingPoints; %mean sq
error
    while loopCountBoxConstraint <= loopMaxBoxConstraint
        while loopCountKernelScale <= loopMaxKernelScale
            Mdl =
fitrsvm(x,y,'KernelFunction',kernelFunctionName,...
                'IterationLimit',maxItr,'BoxConstraint',
BoxConstraint,...
                'KernelScale', KernelScale, 'Epsilon',
epsilon,'Standardize',true);
            fxpr = predict(Mdl, xpr);
            mse = norm(ypr-fxpr)^2/testingPoints;
            mapeError = mape(ypr,fxpr,testingPoints);
            if (Bestmse > mse)
                BestMapeError = mapeError;
                BestMdl = Mdl;
                BestKernelScale = KernelScale;
                BestBoxConstraint = BoxConstraint;
                Bestmse = mse;
            end
            loopCountKernelScale = loopCountKernelScale +
1;
            KernelScale = KernelScale * 10;
        end %while loopCountKernelScale
        KernelScale = 1e-9;
        loopCountKernelScale = 0;
        loopCountBoxConstraint = loopCountBoxConstraint +
1;
```

```
        BoxConstraint = BoxConstraint * 10;
    end %while loopCountBoxConstraint

end
if (isAutoHyper == true)
    fxpr = predict(BestMdl, x);
else
    fxpr = [predict(BestMdl, x);predict(BestMdl, xpr)];
    y = [y;ypr];
    dates = [dates;datespr];
end

Mdl = BestMdl;

%disp('finished')
save ("models/" + file_name,"Mdl");
end
```

Function to predict price using SVR

```
function [datespr, ypr, rst,errors,mse,Mape] =
predict_svr(dtname,model,daysToPredict)
data = readtable(dtname,'Format','%{yyyy-MM-
dd}D%d%d%d%d%d');

N=height(data);
predictPoints = daysToPredict;
shift = N - 5;

xpr = double(data{shift + 1:shift + predictPoints,2});
ypr = double(data{shift + 1:shift + predictPoints,5});
datespr = data{shift + 1:shift + predictPoints,1};

tst = zeros(1,daysToPredict);
rst = zeros(1,daysToPredict);
tst(1) = xpr(1); %assume that we know 1st Open Price
i = 1;
while (i < daysToPredict + 1)
    rst(i) =  predict(model, tst(i));
    i = i + 1;
    if (i ~= daysToPredict + 1)
        tst(i) = rst(i-1); %previous day close = new day
open
    end
end

% Mean Square error (Gaussian Kernel)
```

```
mse = norm(ypr-rst)^2/daysToPredict;
i = 1;
errors = zeros(1, daysToPredict);
Mape = mape(ypr, rst, daysToPredict) * 100;
while i <= daysToPredict
    errors(i) = ((ypr(i)-rst(i))/(rst(i))) * 100;
    i = i + 1;
end
end
```

LSTM

parameters.py

```
import os
import time
from tensorflow.keras.layers import LSTM

# Window size or the sequence length
N_STEPS = 5
# Lookup step, 1 is the next day
LOOKUP_STEP = 5

TRAIN_SIZE = 0

# test ratio size, 0.2 is 20%
TEST_SIZE = 0.4
# features to use
FEATURE_COLUMNS = ["Open", "High", "Low", "Close", "Volume"]
# date now
date_now = time.strftime("%Y-%m-%d")

### model parameters
TRAIN_RAW = True
N_LAYERS = 4
# LSTM cell
CELL = LSTM
# 256 LSTM neurons
UNITS = 32
# 40% dropout
DROPOUT = 0.2

### training parameters

# mean squared error loss
LOSS = "mse"
OPTIMIZER = "rmsprop"
BATCH_SIZE = 64
EPOCHS = 300

# Apple stock market
ticker = "WIG20"
ticker_data_filename = os.path.join("data", f"WIG20_d.csv")
# model name to save
model_name = f"{date_now}_{ticker}-{LOSS}-{CELL.__name__}-seq-{N_STEPS}-step-
{LOOKUP_STEP}-layers-{N_LAYERS}-units-{UNITS}"

# Row data name
row_data = f"wig20_d.csv"
```

stock_prediction.py

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from yahoo_fin import stock_info as si
from collections import deque
from parameters import *
```

```python
import numpy as np
import pandas as pd
import random


def load_data(ticker, n_steps=50, scale=True, shuffle=True, lookup_step=1,
              test_size=0.2, feature_columns=['High', 'Low', 'Open', 'Close',
'Volume']):
    """
    Loads data from Yahoo Finance source, as well as scaling, shuffling, normalizing
and splitting.
    Params:
        ticker (str/pd.DataFrame): the ticker you want to load, examples include
AAPL, TESL, etc.
        n_steps (int): the historical sequence length (i.e window size) used to
predict, default is 50
        scale (bool): whether to scale prices from 0 to 1, default is True
        shuffle (bool): whether to shuffle the data, default is True
        lookup_step (int): the future lookup step to predict, default is 1 (e.g next
day)
        test_size (float): ratio for test data, default is 0.2 (20% testing data)
        feature_columns (list): the list of features to use to feed into the model,
default is everything grabbed from yahoo_fin
    """
    # see if ticker is already a loaded stock from yahoo finance
    if isinstance(ticker, str):
        # load it from yahoo_fin library
        if TRAIN_RAW == True:
            header_list = ["High", "Low", "Open", "Close", "Volume"]
            df = pd.read_csv(os.path.join("data", row_data), sep=';',
names=header_list, header=0, encoding='latin-1')
        else:
            df = si.get_data(ticker, start_date='06.07.2010', end_date='06.05.2020')

        index = df.index
        number_of_rows = len(index)
        if (number_of_rows < 40):
            TEST_SIZE = 5 / number_of_rows
            TRAIN_SIZE = number_of_rows - 5
        elif (number_of_rows < 90):
            TEST_SIZE = 5 / number_of_rows
            TRAIN_SIZE = number_of_rows - 5
        else:
            TEST_SIZE = 5 / number_of_rows
            TRAIN_SIZE = number_of_rows - 5

    elif isinstance(ticker, pd.DataFrame):
        # already loaded, use it directly
        df = ticker
    else:
        raise TypeError("ticker can be either a str or a `pd.DataFrame` instances")

    for col in feature_columns:
        df[col] = df[col].astype(float)
    # this will contain all the elements we want to return from this function
    result = {}
    # we will also return the original dataframe itself
    result['df'] = df.copy()

    # make sure that the passed feature_columns exist in the dataframe
```

```
    if scale:
        column_scaler = {}
        # scale the data (prices) from 0 to 1
        for column in feature_columns:
            scaler = preprocessing.MinMaxScaler()
            df[column] = scaler.fit_transform(np.expand_dims(df[column].values,
axis=1))
            column_scaler[column] = scaler

        # add the MinMaxScaler instances to the result returned
        result["column_scaler"] = column_scaler

    # add the target column (label) by shifting by `lookup_step`
    df['future'] = df['Close'].shift(-lookup_step)

    # last `lookup_step` columns contains NaN in future column
    # get them before droping NaNs
    last_sequence = np.array(df[feature_columns].tail(lookup_step))

    # drop NaNs
    df.dropna(inplace=True)

    sequence_data = []
    sequences = deque(maxlen=n_steps)
    training_close_values = []
    for entry, target in zip(df[feature_columns].values, df['future'].values):
        sequences.append(entry)
        if len(sequences) == n_steps:
            sequence_data.append([np.array(sequences), target])

    for entry in df["Close"].values:
        training_close_values.append(entry)

    result['training_close_values'] = training_close_values


    # get the last sequence by appending the last `n_step` sequence with
`lookup_step` sequence
    # for instance, if n_steps=50 and lookup_step=10, last_sequence should be of 59
(that is 50+10-1) length
    # this last_sequence will be used to predict in future dates that are not
available in the dataset
    last_sequence = list(sequences) + list(last_sequence)
    # shift the last sequence by -1
    last_sequence = np.array(pd.DataFrame(last_sequence).shift(-1).dropna())
    # add to result
    result['last_sequence'] = last_sequence

    # construct the X's and y's
    X, y = [], []
    for seq, target in sequence_data:
        X.append(seq)
        y.append(target)

    # convert to numpy arrays
    X = np.array(X)
    y = np.array(y)

    # reshape X to fit the neural network
    X = X.reshape((X.shape[0], X.shape[2], X.shape[1]))
```

```
    # split the dataset
    result["X_train"], result["X_test"], result["y_train"], result["y_test"] =
train_test_split(X, y,

test_size=TEST_SIZE,

shuffle=True)
    # return the result
    return result


def create_model(input_length, units=256, cell=LSTM, n_layers=2, dropout=0.3,
                 loss="mean_absolute_error", optimizer="rmsprop"):
    model = Sequential()
    for i in range(n_layers):
        if i == 0:
            # first layer
            model.add(cell(units, return_sequences=True, input_shape=(None,
input_length)))
        elif i == n_layers - 1:
            # last layer
            model.add(cell(units, return_sequences=False))
        else:
            # hidden layers
            model.add(cell(units, return_sequences=True))
        # add dropout after each layer
        model.add(Dropout(dropout))

    model.add(Dense(1, activation="linear"))
    model.compile(loss=loss, metrics=["mean_absolute_error"], optimizer=optimizer)
    model.summary()

    return model
```

start.py

```
import os
import parameters
import sys
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from stock_prediction import create_model, load_data
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard
import pandas as pd
import matplotlib.pyplot as plt
from parameters import *
from sklearn.metrics import accuracy_score
import numpy as np



class Window(QWidget):
    def __init__(self):
        super().__init__()
        self.title = "LSTM for Stock Price Prediction"
        self.left = 300
```

```
        self.top = 300
        self.width = 500
        self.height = 600
        self.initUI()

    def initUI(self):

        label1 = QLabel('Arial font', self)
        label1.setGeometry(15, 15, 500, 32)
        label1.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
        label1.setText("LSTM for Stock Price Prediction")
        label1.move(65, 15)
        label1.setAlignment(Qt.AlignCenter)
        label1.setFont(QFont('Arial', 16))

        button1 = QPushButton("Train", self)
        button1.move(125, 80)
        button1.resize(150, 70)
        button1.clicked.connect(self.button1_clicked)

        button2 = QPushButton("Test", self)
        button2.move(350, 80)
        button2.resize(150, 70)
        button2.clicked.connect(self.button2_clicked)

        qbtn = QPushButton('Quit', self)
        qbtn.clicked.connect(QApplication.instance().quit)
        qbtn.resize(qbtn.sizeHint())
        qbtn.move(470, 240)

        self.textbox = QTextEdit(self)
        self.textbox.setGeometry(50, 50, 300, 40)
        self.textbox.setText("The model is ready to start training.\nPress Train.")
        #label3.setText("Here the flow of the program will be displayed.
\nTraining... Trained. Testing... Tested. \nThe prediction is: ")
        self.textbox.setFont(QFont('Arial', 7))
        self.textbox.move(150, 170)


    def plot_graph(self, model, data):
        X_test = data["X_test"]
        y_pred = model.predict(X_test)
        y_test = data["y_test"]
        y_test =
np.squeeze(data["column_scaler"]["Close"].inverse_transform(np.expand_dims(y_test,
axis=0)))
        y_pred = np.squeeze(data["column_scaler"]["Close"].inverse_transform(y_pred))



        tableau20 = [(31/255, 119/255, 180/255), (174/255, 199/255, 232/255),
(255/255, 127/255, 14/255), (255/255, 187/255, 120/255)]

        plt.title('Prediction')
        plt.plot(y_test[-5:], lw=2.5, color=tableau20[2])
        plt.plot(y_pred[-5:], lw=2.5, color=tableau20[3])
        plt.xlabel("Days")
        plt.ylabel("Price")
        plt.legend(["Actual Price", "Predicted Price"])
        self.textbox.setPlainText(self.textbox.toPlainText() + "Predicted results " +
str(y_pred[-5:]) + "\n")
        plt.show()
```

```
    def predict(self, model, data):
        # retrieve the last sequence from data
        last_sequence = data["last_sequence"]
        # retrieve the column scalers
        column_scaler = data["column_scaler"]
        # reshape the last sequence
        last_sequence = last_sequence.reshape((last_sequence.shape[1],
last_sequence.shape[0]))
        # expand dimension
        last_sequence = np.expand_dims(last_sequence, axis=0)
        # get the prediction (scaled from 0 to 1)
        prediction = model.predict(last_sequence)
        # get the price (by inverting the scaling)
        predicted_price = column_scaler["Close"].inverse_transform(prediction)[0][0]

        return predicted_price


    def button1_clicked(self):
        self.textbox.setPlainText('Training...')
        # create these folders if they does not exist
        if not os.path.isdir("results"):
            os.mkdir("results")

        if not os.path.isdir("logs"):
            os.mkdir("logs")

        if not os.path.isdir("data"):
            os.mkdir("data")

        # load the data
        data = load_data(ticker, N_STEPS, lookup_step=LOOKUP_STEP,
test_size=TEST_SIZE, feature_columns=FEATURE_COLUMNS)

        # construct the model
        model = create_model(N_STEPS, loss=LOSS, units=UNITS, cell=CELL,
n_layers=N_LAYERS,
                             dropout=DROPOUT, optimizer=OPTIMIZER)

        # some tensorflow callbacks
        checkpointer = ModelCheckpoint(os.path.join("results", model_name),
save_weights_only=True, save_best_only=True,
                                       verbose=1)
        tensorboard = TensorBoard(log_dir=os.path.join("logs", model_name))

        print('# Fit model on training data')
        history = model.fit(data["X_train"], data["y_train"],
                            batch_size=BATCH_SIZE,
                            epochs=EPOCHS,
                            validation_data=(data["X_test"], data["y_test"]),
                            callbacks=[checkpointer, tensorboard],
                            verbose=1)

        model.save(os.path.join("results", model_name) + ".h5")
        self.textbox.setPlainText('The model finished training. Proceed with
testing.')


    def button2_clicked(self):
        self.textbox.setText('Testing...')
```

```
        # load the data
        data = load_data(ticker, N_STEPS, lookup_step=LOOKUP_STEP,
test_size=TEST_SIZE,
                         feature_columns=FEATURE_COLUMNS, shuffle=False)

        #   construct the model
        model = create_model(N_STEPS, loss=LOSS, units=UNITS, cell=CELL,
n_layers=N_LAYERS,
                             dropout=DROPOUT, optimizer=OPTIMIZER)

        model_path = os.path.join("results", model_name) + ".h5"
        model.load_weights(model_path)

        # evaluate the model
        results = model.evaluate(data["X_test"], data["y_test"])
        self.textbox.setPlainText('test loss, test acc:' + str(results) + '\n')
        print('test loss, test acc:', results)

        self.plot_graph(model, data)


def start():
    app = QApplication(sys.argv)
    win = Window()
    win.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    start()
```

CNN

Initial setup

The following solution is implemented in R language with usage of external packages. Before the execution, all the packages has to be installed. These can be done via running the **package-installer.R** script:

```r
install.packages("shiny")
install.packages("shinyjs")
install.packages("shinydashboard")
install.packages("tensorflow")
install.packages("keras")
install.packages("ggplot2")
install.packages("stringr")
```

Execution

After the successful installation of all the packages, solution may be executed via running **runme.R** script, which will referee to the app.R in cnn-app folder:

```r
library("shiny")
library("shinyjs")
library("shinydashboard")
library("tensorflow")
library("keras")
library("ggplot2")
library("stringr")

runApp("cnn-app", launch.browser = TRUE)
```

Setting `launch.browser = TRUE` indicates the execution of the application straight in the browser. After the start of the script, GUI will appear shortly and will be fully available when all the packages will be included.

Application components

There are 3 main files, which form the application:

- **app.R** gets ui and server components and launches the server:

```r
source("ui.R")
source("server.R")

shinyApp(ui = ui, server = server)
```

- **ui.R** responsible for main visual components of the application:

```r
width <- 400
button_width <- "100px"
```

```r
ui <- dashboardPage(
  dashboardHeader(title = "Stock prediction CNN", titleWidth = width),
  dashboardSidebar(
    width = width,

    useShinyjs(),

    h3("Data section", style = "margin-left: 12px"),

    selectInput("data",
      width = "100%",
      label = "Choose data",
      choices = NULL
    ),

    hr(),

    h3("Model section", style = "margin-left: 12px"),

    radioButtons("model_radio",
      label = "Choose model",
      choices = list("New" = TRUE, "Pretrained" = FALSE),
      selected = FALSE
    ),

    uiOutput(outputId = "dynamicModels"),

    hr(),

    uiOutput(outputId = "dynamicSection"),

    fluidRow(
      column(
        4,
        actionButton("prepare",
          label = "Prepare data"
        )
      ),
      column(
        6,
        uiOutput("prep_state")
      )
    ),

    hr(),

    h3("Actions section", style = "margin-left: 12px"),

    fluidRow(
      # column(4,
      #        actionButton("train",
      #                     label = "Train"),),
      # column(4,
```

```
      #        actionButton("test",
      #                     label = "Test"),),
      # column(4, actionButton("prediction",
      #                        label = "Prediction")),
      column(
        12,
        div(
          style = "display: flex",
          div(
            style = "display: flex, flex: 1",
            actionButton("train",
              width = "100%",
              label = "Train"
            )
          ),
          div(
            style = "display: inline-block, flex: 1",
            actionButton("test",
              width = "100%",
              label = "Test"
            )
          ),
          div(
            style = "display: inline-block, flex: 1",
            actionButton("prediction",
              width = "100%",
              label = "Prediction"
            )
          ),
        )
      )
    )
  ),

  dashboardBody(
    tabsetPanel(
      id = "plotTabs", type = "tabs",
      tabPanel("Raw Data", value = "dataPlot", plotOutput(outputId = "dynamicPlot")),
      tabPanel("Train", value = "trainPlot", plotOutput(outputId = "dynamicTrain")),
      tabPanel("Test", value = "testPlot", plotOutput(outputId = "dynamicTest")),
      tabPanel("Prediction", value = "predictionPlot", plotOutput(outputId = "dynamicPrediction"))
    ),

    verbatimTextOutput("console")
  )
)
```

- **server.R** handles the main logic of the GUI:

```
server <- function(input, output, session) {

  # Usage of helper functions

  source("helper_functions/functions.R")
```

```r
source("helper_functions/model.R")
source("helper_functions/execute.R")
source("helper_functions/plots.R")

############################################################
# Loacl variable initialization
############################################################

data_name <- reactiveVal()
data_frame <- NULL
name <- NULL
model <- NULL
X_train <- reactiveVal()
Y_train <- reactiveVal()
X_test <- reactiveVal()
test <- reactiveVal()
test_length <- NULL
test_result <- reactiveVal()
prediction <- reactiveVal()
prediction_result <- reactiveVal()
ts_length <- NULL
p_num <- NULL

prerared_flag <- reactiveVal(FALSE)
models_names <- reactiveVal()
data_names <- get_data_names()
history_change <- reactiveVal()


############################################################
# Event Handlers
############################################################

# Radio buttons choice changed handler
observeEvent(input$model_radio, {
  output$dynamicModels <- renderUI({
    if (input$model_radio == FALSE) {
      div(
        selectInput(
          "model",
          width = "100%",
          label = "Models",
          choices = models_names()
        ),
        actionButton("summary", label = "Summary"),
      )
    } else {
      div(
        textInput("modelName", label = "New Model", width = "100%"),
        fluidRow(
          column(
            4,
            numericInput("ts", "Trimeseries length", value = 10)
```

```
          ),
          column(
            4,
            numericInput("tl", "Testing length", value = 10)
          ),
          column(
            4,
            numericInput("pn", "Prediction length", value = 5)
          )
        ),
        div(
          actionButton("add", label = "Add model"),
        ),
        disable("prepare")
      )
    }
  })
  prerared_flag(FALSE)
})

# Model changed handler
observeEvent(input$model, {
  if (!is.null(input$model) & input$model != "") {
    from_name <- strsplit(input$model, " ")[[1]]

    # set variables
    name <<- from_name[[2]]
    ts_length <<- as.integer(from_name[[3]])
    test_length <<- as.integer(from_name[[4]])
    p_num <<- as.integer(from_name[[5]])

    model <<- model_load(input$model)
    enable("prepare")
    prerared_flag(FALSE)
  } else {
    disable("prepare")
  }
})

# Name input handler
observe({
  name <<- input$modelName
})

# Prepared_flag change handler
observe({
  if (prerared_flag()) {
    enable("train")
    enable("test")
    enable("prediction")
    output$prep_state <- renderUI({
      div(
        h4(style = "color: green", "OK")
```

```
      )
    })
  } else {
    disable("train")
    disable("test")
    disable("prediction")
    output$prep_state <- renderUI({
      h4(style = "color: red", "NOT READY")
    })
  }
})

# Display of data
observe({
  updateSelectInput(session, "data", choices = data_names)
})

# Data change handler
observeEvent(input$data, {
  data_name(input$data)

  # update models

  models_names(get_models_names(data_name()))

  # set corresponding tab section
  updateTabsetPanel(session, "plotTabs", selected = "dataPlot")
  prerared_flag(FALSE)
})

# Timeseries length input handler
observe({
  ts_length <<- input$ts
})

# Test length input handler
observe({
  test_length <<- input$tl
})

# Prediction number input handler
observe({
  p_num <<- input$pn
})

# Add model click handler
onclick("add", {
  if (!is.null(name)) {
    if (name != "" & !is.null(ts_length) & !is.null(p_num) & !is.null(test_length)) {
      model <- model_initialization(ts_length)
      model_name <- glue::glue("{data_name()} {name} {ts_length} {test_length} {p_num}")
      model_save(model, model_name)
```

```
      # update models
      models_names(get_models_names(data_name()))

      # update gui components
      updateRadioButtons(session, "model_radio", selected = FALSE)
    }
  }
})

# Prepare data click handler
onclick("prepare", {
  data <- prepare_data(data_frame, t_num = test_length, timeseires_length = ts_length, p_num = p_num)
  X_train(data$X_train)
  Y_train(data$Y_train)
  X_test(data$X_test)
  test(data$Test)
  prediction(data$Predict)

  prerared_flag(TRUE)
})

# Summary click handler
onclick("summary", {
  output$console <- renderPrint({
    summary(model)
  })
})

# Train click handler
onclick("train", {
  updateTabsetPanel(session, "plotTabs", selected = "trainPlot")

  # custom callback
  cb <- callback_lambda(on_epoch_end = function(epoch, logs) {
    html("console", {
      glue::glue("Epoch: {epoch+1}/100, loss: {logs$loss}, val_loss: {logs$val_loss}")
    })
  })
  history <- model_training(model, X_train(), Y_train(), cb)

  history_change(history)
  print(glue::glue("{data_name()} {name} {ts_length} {test_length} {p_num}"))
  model_save(model, glue::glue("{data_name()} {name} {ts_length} {test_length} {p_num}"))
})

# Test click handler
onclick("test", {
  updateTabsetPanel(session, "plotTabs", selected = "testPlot")
  if (!is.null(model)) {
    test_result(model_prediction(model, X_test()))
  }
})
```

```
  # Prediction click handler
  onclick("prediction", {
    updateTabsetPanel(session, "plotTabs", selected = "predictionPlot")
    if (!is.null(model)) {
      last_ts <- X_test()[dim(X_test())[1], , ]
      known_open <- prediction()[1]

      last_ts <- append_timeseries(last_ts, known_open)
      prediction_result(real_prediction(model, last_ts, pnum = p_num))
    }
  })


  ############################################################
  # Plots
  ############################################################

  # Dynamic prediction plot
  output$dynamicPrediction <- renderPlot({
    if (!is.null(prediction_result())) {
      plot_prediction(prediction(), prediction_result())
    }
  })

  # Dynamic raw data plot
  output$dynamicPlot <- renderPlot({
    if (!is.null(data_name()) & data_name() != "") {
      data_frame <<- choose_data(data_name())
      plot_raw_data(data_frame)
    }
  })

  # Dynamic test data plot
  output$dynamicTest <- renderPlot({
    if (!is.null(test_result())) {
      plot_test(test(), test_result(), ts_length = ts_length)
    }
  })

  # Console text change
  output$console <- renderPrint({
    print("TRAIN CONSOLE")
  })


  # Dynamic train result plot
  output$dynamicTrain <- renderPlot({
    if (!is.null(history_change())) {
      plot(history_change())
    }
  })
}
```

Helper functions

Several custom function were implemented and used throughout the application. All the function are stored in helper_function folder. There are 4 of them:

- **functions.R** has useful functions for data transformation as well as extraction of features from inputs:

```
  form_timeseriese <- function(open, close = NULL, steps) {
  X <- matrix(ncol = steps)
  Y <- matrix(ncol = 1)

  for (i in 1:length(open)) {
    index <- i + steps - 1
    if (index > length(open)) {
      break
    }
    seq_x <- open[i:index]
    seq_y <- close[index]

    X <- rbind(X, seq_x)
    Y <- rbind(Y, seq_y)
  }

  X <- X[-1, ]
  Y <- Y[-1, ]

  dim(X) <- c(dim(X)[1], dim(X)[2], 1) # features is set to 1 as we have 1D data

  return(list(X = X, Y = Y))
}

append_timeseries <- function(ts, value) {
  ts <- c(ts, value) # append the previously predicted value
  ts <- ts[-1] # shift the 1st value
}

divide_data <- function(data, tnum, ts_length, pnum = 5) {
  open <- data$Open
  close <- data$Close
  size <- length(open)
  train_size <- length(open) - tnum - pnum
  test_size <- length(open) - pnum

  print(size)
  print(train_size)
  print(test_size)
  train <- open[1:train_size]
  test <- open[(train_size - ts_length + 2):test_size]
  predict <- open[(test_size + 1):size]
  actual <- close[1:train_size]
  return(list(Train = train, Test = test, Predict = predict, Actual = actual))
}

choose_data <- function(name) {
  data_name <- paste("train_data/", name, sep = "")
```

```
  data_frame <- read.csv(data_name)
  return(data_frame)
}

get_bounds <- function(data1, data2) {
  if (min(data1) < min(data2)) {
    v_min <- min(data1)
  } else {
    v_min <- min(data2)
  }

  if (max(data1) > max(data2)) {
    v_max <- max(data1)
  } else {
    v_max <- max(data2)
  }

  return(list(Min = v_min, Max = v_max))
}

get_models_names <- function(name) {
  files <- list.files(glue::glue("models/"))
  for (file in files) {
    if (!grepl(name, file)) {
      files <- files[files != file]
    }
  }
  return(files)
}

get_data_names <- function() {
  return(list.files("train_data/"))
}
```

- **execute.R** has 2 functions dedicated to divide the input data into train, test and prediction data sets and to perform real prediction with a model:

```
source("helper_functions/functions.R")

prepare_data <- function(data_frame, timeseires_length, t_num, p_num = 5){

  list <- divide_data(data_frame, tnum = t_num, ts_length = timeseires_length)
  train <- list$Train
  predict <- list$Predict
  actual <- list$Actual
  test <- list$Test

  train_list <- form_timeseriese(open = train, close = actual, steps = timeseires_length)

  X_train <- train_list$X
  Y_train <- train_list$Y

  test_list <- form_timeseriese(open = test, steps = timeseires_length)
```

```r
  X_test <- test_list$X

  return(list(X_train = X_train, Y_train = Y_train, X_test = X_test, Test = test, Predict = predict))
}

real_prediction <- function(model,last_ts,pnum = 5){

  temp_ts <- last_ts
  predictions <- vector()
  for (i in 1:pnum){
      dim(temp_ts) <- c(1,length(temp_ts),1)# change ts dimensionality to fit the model
      last_predicted <- model_prediction(model,temp_ts)
      dim(temp_ts) <- c(dim(temp_ts)[2])# change it's dimensionality back to normal
      temp_ts <- append_timeseries(temp_ts, last_predicted)
      predictions <- c(predictions, last_predicted) # append to already predicted values
  }

  return(predictions)
}
```

- **plots.R** implements ggplot's for every scenario:

```r
plot_raw_data <- function(data_frame) {
  size <- length(data_frame$Open)
  dd <- data.frame(1:size, data_frame$Open, data_frame$Close)
  colnames(dd) <- c("number", "open", "close")
  ggplot(dd) +
    geom_line(aes(x = number, y = open, color = "pse")) +
    geom_line(aes(x = number, y = close, color = "unemploy")) +
    scale_color_discrete(name = "Legend", labels = c("open", "close")) +
    ggtitle("DATA: close and open prices") +
    xlab("Number") +
    ylab("Stock Prices") +
    theme(plot.title = element_text(size = 20))
}

plot_test <- function(actual, test, ts_length) {
  dd <- data.frame(1:(length(actual) - ts_length + 1), actual[ts_length:length(actual)], test)
  colnames(dd) <- c("number", "actual", "predicted")
  ggplot(dd) +
    geom_line(aes(x = number, y = actual, color = "pse")) +
    geom_line(aes(x = number, y = predicted, color = "unemploy")) +
    scale_color_discrete(name = "Legend", labels = c("real", "predicted")) +
    ggtitle("MODEL: test result") +
    xlab("Number") +
    ylab("Stock Prices") +
    theme(plot.title = element_text(size = 20))
}

plot_prediction <- function(actual, test) {
  dd <- data.frame(1:length(actual), actual, test)
  colnames(dd) <- c("number", "actual", "predicted")
  ggplot(dd) +
    geom_line(aes(x = number, y = actual, color = "pse")) +
```

```
    geom_line(aes(x = number, y = predicted, color = "unemploy")) +
    scale_color_discrete(name = "Legend", labels = c("real", "predicted")) +
    ggtitle("MODEL: prediction result") +
    xlab("Number") +
    ylab("Stock Prices") +
    theme(plot.title = element_text(size = 20))
}
```

- **model.R** stores main functions for CNN model initialization, training and execution:

```
model_initialization <- function(timeseires_length){

  model <- keras_model_sequential() %>%
    layer_conv_1d(filters = 128, kernel_size = 2, activation = "relu",input_shape = c(timeseires_length
    layer_max_pooling_1d(pool_size = 2)

  model %>%
    layer_dropout(0.4) %>%
    layer_flatten() %>%
    layer_dense(units = 50, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
    optimizer = "adam",
    loss = "mse",
    #metric = "mse"
  )

  summary(model)

  return(model)
}

model_training <- function(model, X, Y, cb){
  history <- model %>% fit(
    x = X, y = Y,
    epochs = 1000,
    use_multiprocessing = TRUE,
    validation_split=0.2,
    verbose = 0,
    callbacks = list(cb),
  )
}

model_prediction <- function(model, X){
  prediction <- model %>% predict(
    X,
    verbose = 0
  )
  return(prediction)
}

model_save <- function(model, name){
  model %>% save_model_tf(paste("models/", name, sep = ""))
```

```r
}

model_load <- function(name){
  model <- load_model_tf(paste("models/", name, sep = ""))
  return(model)
}
```