Data Science Internship

Machine Learning Assignment

by Basel Mather

ProgressSoft
www.progressoft.com

ProgressSoft Assignment

Table of Contents

GitHub Repository for the Solutions: Click Here

Problem 1 – Face Blurring:

Approach

This project aims to blur faces in images while keeping the rest of the image unchanged. To achieve this, we utilized two approaches to detect faces in an image:

The solution consists of two models: A pre-trained model using **Haar Cascade Classifier**, and a trained model from scratch with **YOLO v8**.

The Solution includes multiple files:

- ***FaceBlurring.ipynb***: which is a Jupyter notebook that has the models testing and the evaluation.
- ***YOLO_Training.ipynb***: which is a Jupyter notebook for training the **YOLO** model on a custom dataset, the training process happened on **Google Colab**.
- ***FB_YOLO.py***: which is a Python script to blur the faces of an image and save the blurred image using the trained **YOLO** weights.
- ***FB_CascadeClassifier.py***: a Python script to blur the faces of an image and save the blurred image using the **Haar Cascade Classifier**.
- **Pretrained_models file**: a file that contains the **Haar Cascade** pre-trained model, and the weights for the trained **YOLO model**.
- **Roboflow_data file**: a file that contains custom data for face-to-train the **YOLO** model.
- **Face file**: a file that contains images for faces to test the models.

Haar Cascades (pre-trained model):

I used a pre-trained Haar Cascade model to detect faces in an image.
haarcascade_frontalface_default.xml classifier is available in the cv2 library for face detection. I implemented this approach in a Python script **FB_CascadeClassifier.py**.

Figure 1 - Before Blurring

Figure 2 - After Blurring (Cascade Classifier)

Figure 3 - Before Blurring

Figure 4 - After Blurring (Cascade Classifier)

As we can see, the **haarcascade_frontalface_defualt** is not accurate for the faces, as you can see when the face has some kind of rotation the model is not performing very well, that's why I build a model from scratch using **YOLOv8** as the second approach.

The link for the **Haar Cascade** pre-trained model is here.

In the **FB_CascadeClassifier.py** file, I have written a function called **blur_face.**
The function **blur_face()** takes an image path and applies Gaussian blur to the faces detected in the image using a pre-trained **Haar cascade classifier**.

YOLO v8 (train the model from scratch):

I build a **YOLO** model and train it from scratch on an annotated dataset from Roboflow the link for it from here. I made the training process on Colab because they have better GPUs. The link for the training notebook is here. (Please note that I connected my Google Drive account the Colab so I can give the model the path for the images directly without the need to upload all the files into Google Colab)

After training the model on Google Colab, I downloaded the trained weights and used them to make detection locally, I applied this on the ***FB_YOLO.py*** file. In this script, I made a function called ***yolo_face_blur***.

Figure 1 - Before Blurring

Figure 2 - After Blurring (YOLO)

Figure 2 - Before Blurring

Figure 1 - After Blurring (YOLO)

As we can see, the **YOLO** model performed better than the cascade classifier. This model is version **8** of the YOLO (You Only Look Once) model. It is a powerful algorithm that makes many data augmentation while training to make better performance, and in this task, we use the **nano** model which is the smallest model of the other YOLO v8 models.

Dataset:

The dataset used for training the **YOLO** model is the Face Detection dataset from roboflow. This dataset contains 3 folders for training. It has a train, validation, and test folders for the sets. The training folder contains **2,871** images, the validation folder contains **267** images, and the testing folder contains **145** images. All the images are labeled from the roboflow website. Which gives a total of **3,283** images.

Evaluation:

For the evaluation, I chose the **IoU** (Intersection over Union) metric with precision, and recall. Precision and Recall can be calculated from the IoU.

I used a function called ***calculate_iou*** which has an argument for **bounding box 1** and **bounding box 2**, then calculate the IoU between these two boxes, I tested the IoU between the two models to how similar they are two each other for a specific image, and I got I **0.67 IoU**, which means that the detections for the models are close to each other.

Then, I build an evaluation function to test the model on a test dataset, this function is called ***evaluate_model***, which takes the **ground_truth_path**, the **bb_model_func**, and the **iou_threshold**. This function calculates the IoU between the bounding box detection from the model and the ground truth bounding box for a test dataset, then calculate the precision and the recall from the IoU, and returns the precision, recall, and IoU.

I applied this function for both models, but there is something wrong, I got a very bad IoU even though I saw the detections and they were very well. I went a little deeper and I found the problem. The problem is that the range for the detection bounding box values is different from the ground truth. The ground truth bounding box has values between 0 and 1, but the detection has values much higher than that. I tried to rescale both bounding boxes in many ways but I couldn't solve the problem.

The detections for both models are very accurate, and you can test them yourself, I will tell you how to run the scripts to see for yourself.

How to Run the Scripts:

1. Download the Face Detection dataset from roboflow and extract it.

2. Install the required packages by running ***pip install -r requirements.txt***.

3. To blur a specific image using the **Haar cascade classifier** write the following command
   python FB_CascadeClassifier.py < image path >
   The blurred image will be saved as ***cascade_output.jpg***

4. To blur a specific image using the **YOLO** model write the following command
   python FB_YOLO.py < image path >
   The blurred image will be saved as ***Yolo_output.jpg***

Note: I only tested it on Windows.

Future Enhancements:

1. We could explore other face detection models and techniques to improve the accuracy of the module. For example, we could use the MTCNN model, which is known to be robust and accurate for face detection.

2. We could use an adversarial approach to blur the faces, which would improve the privacy of the individuals in the image while preserving the quality of the image.

3. We could explore using deep learning-based methods to detect and blur other sensitive information in the image, such as license plates, personal identification numbers, and so on.

Problem 2 – Face Mask Classifier:

Approach:

This problem is designed to detect whether a person in an image is wearing a face mask or not. It is based on deep learning models trained on the Face Mask Classification dataset from Kaggle.

This solution consists of two models: Transfer Learning with the ResNet50 model and a custom model built using TensorFlow. Both models trained on the same dataset have achieved high accuracy in detecting whether a person is wearing a mask.

This solution includes four Python files:

- **resnet_train.py**: script for training the ResNet50 model.
- **tf_model_train.py**: script for building and training a custom model using TensorFlow.
- **evaluate.py**: script for evaluating the performance of the models on a test dataset.
- **test.py**: script for testing the models on a single image and showing it with the predicted class.

In addition, a Jupyter notebook file called **FaceMaskDetection.ipynb** includes the code for all four python files to showcase the result and have all the codes in one place.

The trained models are also included in the solution:

- **resnet.h5**: trained ResNet50 model (Transfer Learning)
- **tf_model.h5**: trained custom model using TensorFlow (Build Model from Scratch)

Figure 1 - ResNet50

Figure 2 - ResNet50

Figure 1 - TF model

Figure 2 - TF model

Dataset:

The dataset used for training the models is the Face Mask Detection dataset from Kaggle. The dataset consists of approximately 12,000 images, including 6,000 images with people wearing masks and 6,000 images with people not wearing masks.

The dataset contains 3 files, train, validation, and test file. The training file has 10K images (5K with mask, and 5K without mask), the validation file has 800 images (400 with mask, and 400 without mask), and the testing file has 800 images (400 with mask, and 400 without mask).

Training:

The ResNet50 and custom models were trained on the same dataset using the Keras API in TensorFlow. The ResNet50 model was fine-tuned using transfer learning from pre-trained ImageNet weights, while the custom model was built from scratch.

The resnet_train.py script can be run to train the ResNet50 model. The tf_model_train.py script can be run to build and train the custom model using TensorFlow. Both scripts take the path to the dataset as input.

ResNet50:

This script trains a machine learning model to detect whether a person is wearing a face mask or not using a pre-trained **ResNet50** model. The ResNet50 model is a powerful image classification model that has already been trained on a large dataset of images.

To fine-tune the pre-trained ResNet50 model for our specific use case, the script adds several new classification layers on top of the base model. It then compiles the new model using the **Adam** optimizer and **categorical cross-entropy** loss, with metrics for **accuracy**, **precision**, and **recall**.

The script uses the **ImageDataGenerator** from the **Keras** library to generate batches of training and validation data from directories containing images. These images are resized and normalized to have pixel values between 0 and 1.

The script then trains the new model on the training data and validates it on the validation data for a specified number of epochs. It saves the trained model to a specified file name.

To run this script, you need to provide four arguments in the command line. The first two arguments should be the **paths to the training and validation** data directories, respectively. The third argument should be the **number of epochs** to train the model, and the fourth argument should be the **file name** to save the trained model.

Custom Trained Model:

The first step is to define the architecture of the model. In this case, the model is created using the Keras Sequential API. The model is a series of convolutional and max pooling layers, followed by two fully connected layers, with the final layer outputting a probability distribution over the two classes using the **softmax** activation function.

The next step is to compile the model. Here, the model is compiled with **binary cross-entropy** loss and the **Adam** optimizer. The program then creates an instance of the **ImageDataGenerator** class, which is used for data augmentation. The program then loads the **training and validation** data sets using the **flow_from_directory** method of the **ImageDataGenerator** class.

Finally, the program trains the model using the **fit** method of the model object. With the **tf_model_train.py** file, the **training data** and **validation data** are passed in as arguments, along with the **number of epochs** to train for. Once training is complete, the model is saved to a file specified by the user.

Evaluation:

The performance of the models was evaluated on a test dataset using **accuracy**, **precision**, **recall**, and **F1-score** metrics. The **evaluate.py** script can be run to calculate the performance metrics for either the ResNet50 or the custom model. The script takes the **path to the test dataset** and the **path to the pre-trained model** as inputs.

After applying the evaluation step to our models, I got the following values for the metrics.

Models \ Metrics	Accuracy	Precision	Recall	F1-score
ResNet50 model	0.91	0.93	0.87	0.9
Custom model	0.98	0.97	0.98	0.98

As we can see, the custom model performed better than the ResNet50 for transfer learning, maybe it is because the custom model has 19,264,322 parameters, or maybe because the testing data is small, and when we move this solution to production, the ResNet50 maybe will better result that the custom model.

How to Run the Scripts

1. Download the Face Mask Detection dataset from Kaggle and extract it to a folder.

2. Install the required packages by running pip install -r requirements.txt

3. To train and save the ResNet50 model, run the command ***python resnet_train.py <path to the training data> <path to the validation data> <number of epochs> <saved model name>***

4. To train and save the custom TensorFlow model, run the command ***python tf_model_train.py <path to the training data> <path to the validation data> <number of epochs> <saved model name>***

5. To evaluate the model performance on test data, run the command ***python evaluate.py <path to the test dataset folder> <path to the saved model file>***

6. To test the model on a single image, run the command ***python test.py <path to the test image> <path to the saved model file>***

Future Enhancement:

In the future, we can further improve the performance of the model by using data augmentation and hyperparameter tuning. We can also explore other pre-trained models like InceptionV3 and EfficientNet, which might give better performance. Furthermore, we can use object detection models to detect the presence of masks and other objects in the image.

Problem 3- Wrapper Class Library

Approach:

For this problem, I created a .**net 6** class and **Python** class (I will talk about it later) that loads a trained model and uses it to make predictions on images to classify if the person wearing a mask or not.

The C# implementation:

I used the **ML.NET** framework to create a prediction engine that loads the **ONNX** model file, processes the input image, and returns the predicted result. The input image is first resized to the required size and then converted to a pixel array. Finally, the processed image is passed to the prediction engine, which returns the predicted result.

The solution has two main files:

- ***FaceMaskClassifier.cs***: this file has the code for the class and what it does.
- ***Program.cs***: this file is the main file that has the code for making objects from the class giving it the image path as input for it and printing the prediction.

That was what I planned HAHA, but unfortunately, there is a bug with the code and I can't fix it, the class is built in a good way, and the main program also have to bugs, but when I run the code, an error pops up that related to loading the model, and it says that there is an error for initializing python when loading the model. I tried many things to solve this issue but unfortunately, I couldn't. There is a lack of bugs solved in stack overflow for the **C#** programming language. That is why I solved this problem using Python.

The Python Implementation:

I created a wrapper class that loads the model and uses it to make predictions on images. The input image is first load the model then the image using **Keras preprocessing** library and then resized to the required size (the size that the model trained on), then the image is normalized and passed to the loaded model, which returns the predicted class.

This solution has two main files:

- ***Face_mask_classifier_py.py***: which has the wrapper class that does everything mentioned above.

- ***Main.py***: which is the main file called the class that makes predictions and prints the result.

With the ***main.py*** file, I called the class with "***from face_mask_classifier_py import FaceMaskClassifier***" which is the name of the class. This file has a code for the **argparse** library just to run the script in an easy way. When initializing the class instance, we pass the image path and the model path as the parameters to the class, and after that call the predict function that returns the prediction, then print the results.

Test Results:

The program can be tested by providing different images as input and verifying that the prediction is accurate.

How to Run the Program:

- For **.net 6 (C#)**: As it should be, run the following command in cmd: ***dotnet run <img path> <model path>*** but the code will not run and it didn't handle the argument passes through the CMD because that the program will not run because of the bug I mentioned before.

- For **Python**: run the following command in cmd after running ***pip install -r requirements.txt*** : ***python main.py --img_path path/to/img --model_path path/to/model***

  Note: I only tested it on Windows

Future Enhancement:

Possible future enhancements for this problem could be to solve the problem with the .net 6 program, also include adding the ability to predict multiple images at once.

Problem 4 – Image Statistics

Approach:

The approach taken in this program is to calculate the statistics of a given image in terms of mean, standard deviation, minimum and maximum gray values, image height and width, and the number of pixels. To achieve this, the program first loads the image from the given path using the **Bitmap** class in the ***System.Drawing*** namespace. Then, it calculates the gray value of each pixel in the image by converting the RGB color of the pixel to its corresponding grayscale value. After that, it calculates the sum, squared sum, minimum, and maximum gray values of all the pixels to derive the mean, variance, and standard deviation. Finally, the program displays the calculated statistics on the console.

The solution has two files:

- ***Image_Stats.cs***: this program has the ***ImageStats*** class which has a function called ***CalculateStats*** that calculates the image statistics for a given image.

- ***Program.cs:*** this program is the main program that has the code for making an object of this class then call the ***ImageStats*** function and gives it the image path as an input for this function, then shows the Statistics for the image.

Justification for Approach:

The conversion of the RGB color of a pixel to its corresponding grayscale value through the mathematical equation from MathWorks. The equation is **0.2989 * R + 0.5870 * G + 0.1140 * B** (The R, G, and B correspond to the RGB channels in the image). The mean, variance, and standard deviation are commonly used statistical measures that provide insight into the distribution of gray values in an image. The minimum and maximum gray values provide information on the range of gray values in the image. The image height and width and the number of pixels provide information about the size of the image.

Evaluation & Test Results:

The program can be tested by providing different images as input and verifying that the calculated statistics are accurate. **NOTE**: I only tested it on Windows.

How to Run the Program:

To run the scripts run the following command on the CMD: ***dotnet run <path for the image>***

Then the program will output the statistics for the image, and it will show the:

- **Mean**
- **Standard Deviation**
- **Minimum Gray Value**
- **Maximum Gray Value**
- **Image Height**
- **Image Width**
- **Number of Pixels**

Example:

Mean: 142.36
Standard deviation: 37.15

Minimum gray value: 0
Maximum gray value: 255

Image Height: 2975
Image Width: 2082

Number of pixels: 6193950

Future Enhancement:

Possible future enhancements for this program include adding the ability to process multiple images at once and to display the image along with its statistics. Additionally, the program could be extended to perform more complex image processing tasks such as edge detection, image filtering, and object recognition.

Problem 5 – Time Series Prediction

Approach:

To predict the stock's most recent 20% prices, I used two approaches, one using Machine Learning and the other using Deep Learning.

The solution includes four files:

- ***TimeSeries_with_ML.ipynb***: Jupyter notebook to solve the problem with Machine Learning
- ***TimeSeries_with_DL.ipynb***: Jupyter notebook to solve the problem with Deep Learning
- ***prices.txt***: the data file which has the dates and the prices for the stock in txt format.
- ***prices.csv***: the data in CSV, the way of converting happened in the ***TimeSeries_with_ML.ipynb*** file.

The Dataset:

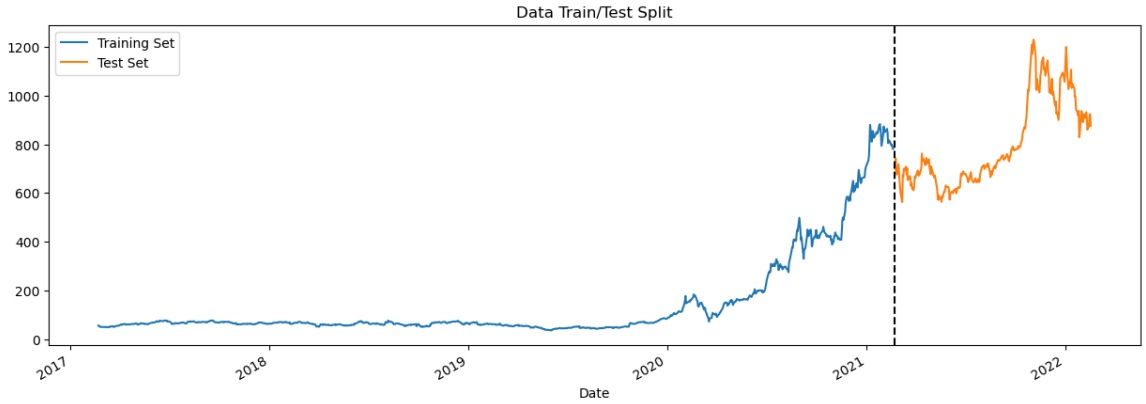
The Data come in .txt format which has values in a comma-separated way. It has 2 features, the **date**, and the **stock prices**. It has **1259** rows (instances) of data. The data has values for each stock for each day, and it starts from **2017-02-21** to **2022-02-17**, which is approximately **5 years**.

What I did is read this file in python, extract the values, and make them as Pandas data frame, make the data type for the date feature as datetime through pandas (for manipulating the data as a date data type), make the date as the index for this data frame, and save this data frame a CSV file using pandas ***.to_csv*** method.

Machine Learning Approach:

I first read the data from the prices.txt file, then extract the values needed and make a data frame from it, then save it as a CSV file. Before everything, I changed the date datatype to datetime, for working with it as a date, not as an object (string).

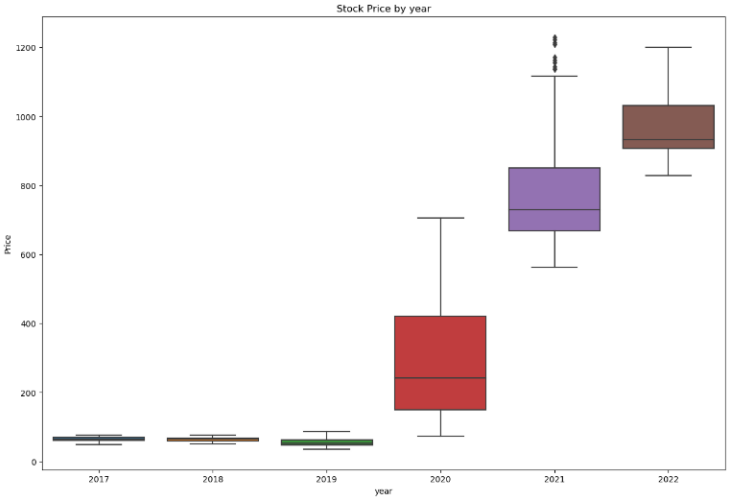
After that, I prepare the data for splitting. The splitting happened as the first **80%** of the data was the training dataset and the final **20%** was the test dataset. This give that we have **1,007** instances for the training data, and **252** instances for the testing data. This means that the data is **1259** instances (stock prices).



Train / Test Split for the Data

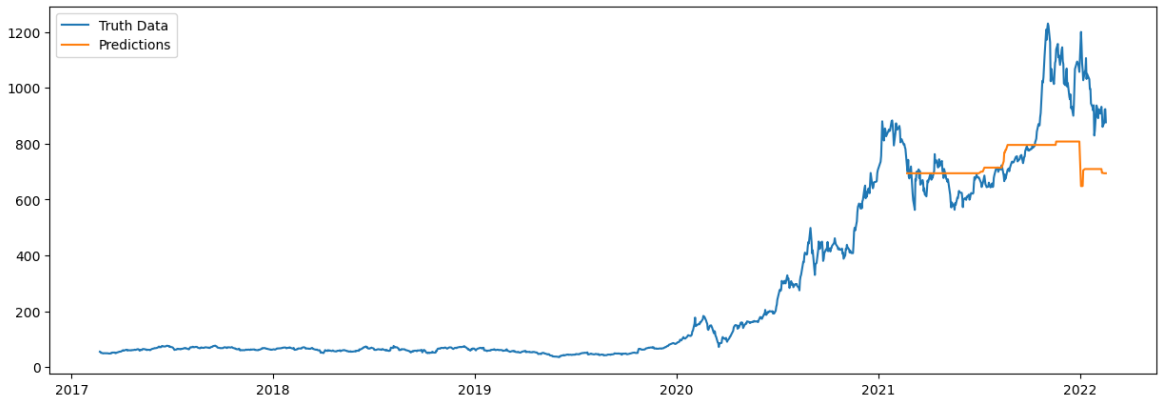
Then, I created a function called ***create_features*** and take data frame as input. This function creates features for the data frame, the features are related to the data and extract information from it, such as the day of the week, quarter, month, year, etc.

Then, do EDA (Exploratory Data Analysis) through some plots of the data. After this step, I split the data into features, and the target (X, y) just like it is a supervised learning approach.



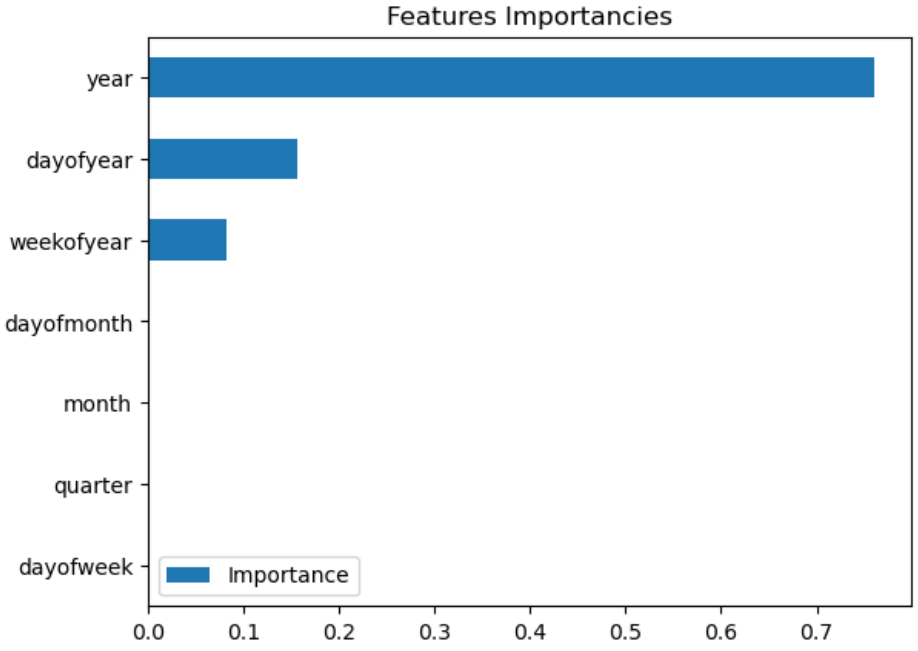
Stock Prices by Year

After that, build the model and train it on the data. The model I used is an ***XGBoost Regressor*** with a 1,000 estimator, 0.01 learning rate, max depth of 2, and evaluation metric as 'MAE' (Mean Absolute Error). Then, evaluate the model with the MAE (Mean Absolute Error), and RMSE (Root Mean Squared Error).



Model Performance on the Test Data

Finally, plot the feature importances for the model and horizontal bar chart, and we conclude from it that the ***year*** feature is the most important feature for the model.



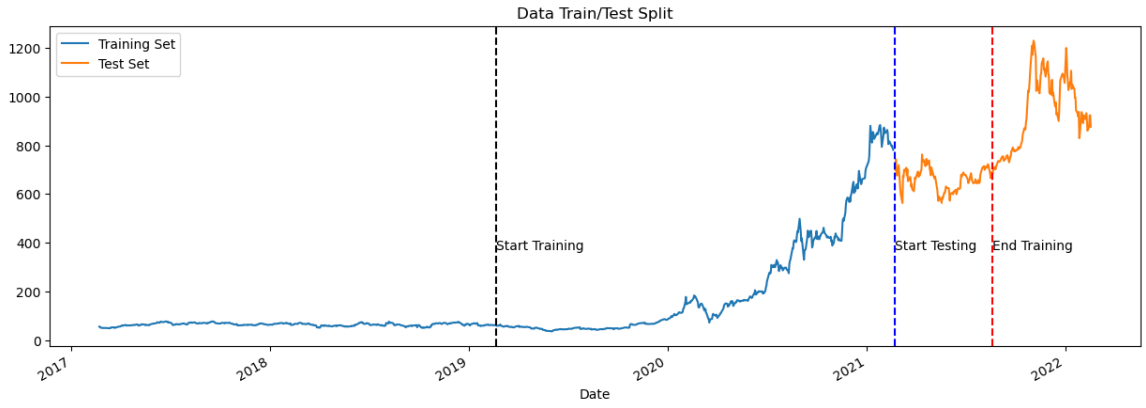
Feature Importances for the Model

The Deep Learning Approach:

I used a window method to preprocess the data, which involved creating windows of fixed size and predicting the next value in each window. I then defined 3 main functions I will use in the notebook, ***plot_series*** which visualizes time series data, ***windowed_dataset*** which generated dataset windows, and ***model_forecast*** which used an input model to generate predictions on data windows.

I then split the data into training and testing sets. That training data is in the range of 40% to 90% of the data, and the testing set has the last 20% of the data. I did this for the training set because the first 40% of the data as almost similar to each other and the range of the price for them is 20 – 100, and recent years have prices more than 500, so there is no benefit to train the model on a data its values kind of disappeared. Note that there will be 10% will be overlapping with the training and testing sets, which will not give us a real evaluation number in the evaluation phase.

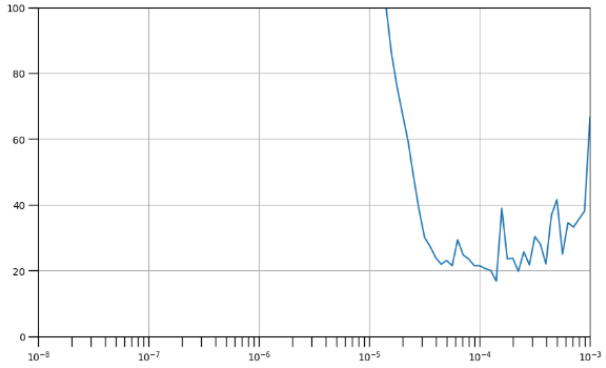
I used the window method because it is a common technique to preprocess time series data for deep learning models.



Train / Test Split for the Data on the DL Model

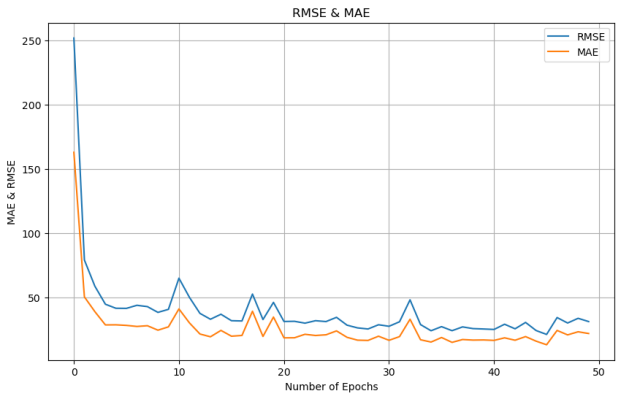
After that, I used a deep learning model with two LSTM layers and multiple Dense layers to train the model on the training set and then tested it on the testing set. I used an LSTM-based model because LSTMs are known to perform well on time series data and can capture the temporal dependencies in the data.

After building the model I trained it on the training data just to pick the best value for the learning rate for this model on this data. This happened with the ***callbacks.LearningRateScheduler***, and then pick the best value of the learning rate and train the model with this value.

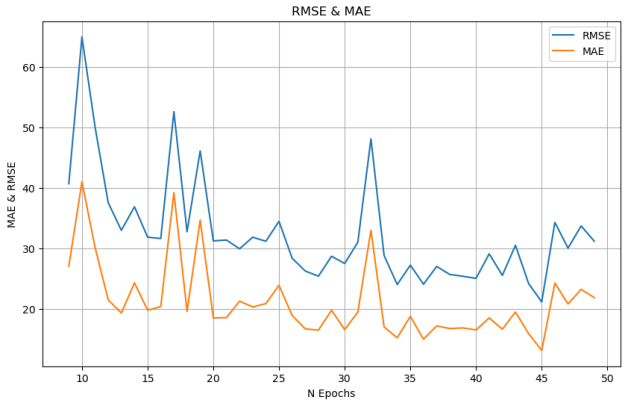


Learning Rate with the Loss

I chose MAE as the loss function, and the RMSE as the metrics. I choose them because they are popular metrics in time-series problems, which can show the model performance in a good way.

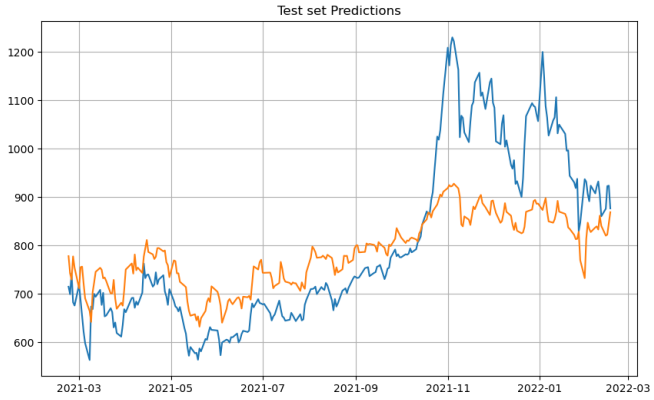


MAE & RMSE while Training

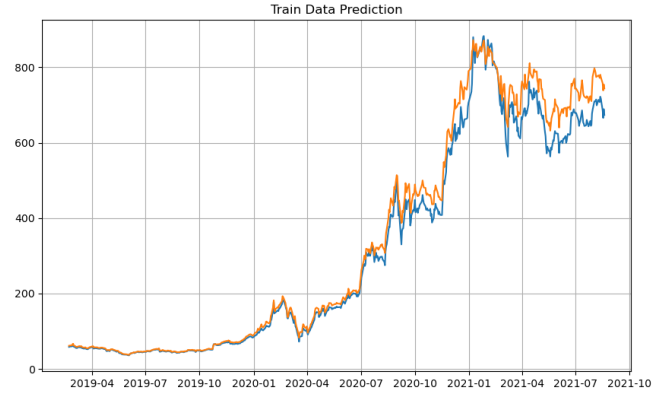


Zoom in to the MAE & RMSE while Training

Then, I made predictions on the train, and the test data, and plot the results.



Model Performance on the Test set



Model Performance on the Training set

Clearly, we can see that the model is overfitted on the training data, I tried the change the model architectures multiple times, but I couldn't prevent the overfitting, I tried to change some values for parameters like window size, batch size, and number of units (neurons) inside the layer, and many other parameters.

Evaluation:

The performance of the models was evaluated on the test dataset using MAE (Mean Absolute Error), and RMSE (Root Mean Square Error) metrics. After applying the evaluation step to our models, I got the following values for the metrics.

Model \ Metrics	MAE	RMSE
Machine Learning	118.24	166.59
Deep Learning	93.81	115.28

As we can see that the deep learning model performed better than the machine learning model, maybe that is because of the complexity of this model, which has multiple LSTMs and Dense layers. The DL model has **150,751 parameters.**

How to Run the Scripts:

For both the ML and DL notebooks, *write **pip install -r requirements.txt*** in **CMD**, then open them and execute the cells from above to bottom, and if you want to explore different values for the parameters of the models, notice that the prices.txt file should be in the same directory of the notebooks.

NOTE: I only tested it on Windows.

Future Enhancement:

For the machine learning approach, future enhancements could include trying out more regression models and hyperparameter tuning to improve the model's performance. For the deep learning approach, enhancements could include trying out different architectures and hyperparameter tuning to improve the model's performance. Additionally, feature engineering could also be explored to add more relevant features to the dataset.

General Information:

Python Version:

The script was developed using Python ***3.9.13***

platform-agnostic:

I only tested the solutions on Windows.