Accademic Year 2021/2022

POLITECNICO
MILANO 1863

Politecnico di Milano

Numerical Analysis for Machine Learning

Stock Prediction using Support Vector Machines

Filippo Manzardo
10864201

Prof. Edie Miglio
Version 1.0
January 27, 2022

Contents

1 Introduction

This document describes the implementation and various results of the Numerical Analysis for Machine Learning assigned project.

The main topic is *Stock Predictions*, which can be a wide argument and has possibly limitless approaches and outcomes.
In this paper, we explore the potential of Support Vector Machines [3] which are supervised learning models with associated learning algorithms that analyse data for classification and regression analysis. An SVM maps training examples to points in space to maximise the width of the gap between the two categories. New samples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.
In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

1.1 Dataset

The dataset used in this research consists of two-years-data of Bitcoin stock price. The data was crawled from Binance API and was saved in the main Github repository of this document. It consists of a multiple comma-separated value dataset that differs in time fragments, ranging from 1 day to 1 minute. As a result, we can choose the amount of data to feed our model saving time and resources.

The Data is subdivided into 5 columns, Open-High-Low-Close-Volume (also known as OHLCV) and the timestamp in microsecond as the index.

Timestamp	Open	High	Low	Close	Volume
1639526400000	42991.66	43845.27	41384.67	43253.95	2609.79577
1639612800000	43263.51	43800.00	41957.95	42051.07	1663.85951
1639699200000	42061.31	42379.99	40282.16	41049.14	2166.91148
1639785600000	41050.42	42144.42	40509.00	41702.15	1323.32180
1639872000000	41707.86	42972.89	41355.10	41517.04	1501.84984

Figure 1: Dataset example

OHLCV is a type of chart typically used to illustrate movements in the price of a financial instrument over time.

Open is the stock price at the beginning of the timestamp

High is the highest stock price at the beginning of the timestamp

Low is the lowest stock price at the beginning of the timestamp

Close is the stock price at the end of the timestamp

Volume is the number of shares traded in the time period

This data representation is very common in the financial market, known as Candlestick Chart. A candlestick chart is a style of financial chart used to describe price movements of a security, derivative, or currency.
It is similar to a bar chart in that each candlestick represents all four important pieces of information for that day: open and close in the thick body; high and low in the "candle wick". Being densely packed with information, it tends to represent trading patterns over short periods of time, often a few days or a few trading sessions.

Figure 2: Example of a Candlestick graph

2 Data Preprocessing

Data preprocessing is probably the most important phase of the entire process. Getting the preprocessing wrong may make any subsequent operation useless. Usually, it consists in choosing the target value, enriching data, normalizing and shaping the entire dataset.

In this implementation, the preprocessing of data is handled by python module: `preprocess_data.py`.

2.1 Prediction Method

SVMs are born as binary classifiers, which means that they discriminate in two classes, namely True and False, 0 and 1, etc. It exists an implementation, called Support Vector Regression Machines (SVR) [4] that uses regression instead of classification, but in Stock Prices Forecasting it's good practice not to work *directly on prices*.
Stock returns are a viable way to go, but to apply SVM to this problem, we can assume a time window of n-shift days which if after them our target value increases we assign a 1, otherwise 0.
For example, if we choose the 'Close' value to predict, if in 10-shift days our close has a higher value we buy now (1), otherwise we wait (0) since it would not be profitable.

In this way, we translated a regression problem into a binary classification problem, which is efficiently handled by SVM.

```
def preprocess_data(data, k=shift_days):

    column = 'Close'
    n = len(data)
    buy = np.zeros(n, dtype=np.int8)
    for i in range(k,n):
        if(data.iloc[i-k][column] < data.iloc[i][column]):
            buy[i-k] = np.int8(1)

    data['Buy'] = buy
```

2.2 Time Series Data Pre-Processing

Time series is a sequence of evenly spaced and ordered data collected at regular intervals. To predict a future value we might choose two approaches:

- The naive way: given an entry of features, we use it to predict our target value
- The windowing way: given n-samples of time-consecutive features, we use them to predict our value.

Intuitively, using multiple time information may result in better overall accuracy, since we might be able to capture past information and trends.

```
def preprocess_data(data, ...):

    window=k
    window_data = []
    labels = []
    prev_days_features= deque(maxlen=window)

    for i in train_x:  # iterate over the values
        # store all but the target 'buy'
        prev_days_features.append([n for n in i[:-1]])
        if len(prev_days_features) == window:
            window_data.append(np.array(prev_days_features))
            labels.append(np.int8(i[-1]))
```

2.3 Data Enrichment

Data enrichment is a process used to enhance, refine or otherwise improve raw data. Stock prices prediction solely based on OHCLV values might not yield good results. To enrich our data, we use famous technical indicators like BollingerBands [1], Moving Averages [8][6] and many more. This indicators are calculated using a Python Library called *TA-Lib* (ref), precisely its implementation exploiting the pandas module, pandas-ta.

In this process, we have to be careful about indicators that might have a 'look-ahead' feature, which would be considered cheating. These indicators, gently reported in the basic documentation, are *dpo's* and *ichimoku's* which have a property 'look-ahead' to be set False.

```
def add_indicators(data):

    # Add returns
    data_noLH=data.copy()
    data.ta.log_return(cumulative=True, append=True)
    data.ta.percent_return(cumulative=True, append=True)

    # Manage LookAhead Indicators
    column_LH = ['ISA_9', 'ISB_26', 'ITS_9', 'IKS_26', 'ICS_26', 'DPO_20']
    base_columns = ['Open', 'High', 'Low', 'Close', 'Volume']

    # Add all strategies, drop if too many NaN
    data.ta.strategy(ta.AllStrategy)
    data.dropna(axis=1, thresh=round(len(data)*0.9), inplace=True)

    # Replace LookAhead features
    data.drop(labels=column_LH, inplace=True, axis=1)
    data_noLH.ta.dpo(lookhead=False)
    data_noLH.ta.ichimoku(lookhead=False)
    data_noLH.drop(labels=base_columns, inplace=True, axis=1)

    # Clean-up
    data = pd.concat([data, data_noLH], axis=1)
    data.dropna(axis=1, thresh=round(len(data)*0.9), inplace=True)
    data.dropna(axis=0, inplace=True)

    return data
```

2.4 Train-Test Splitting

The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model.
This splitting has to be done very carefully since any data leak will make every result invalid. In this research, an 80% Train - 20% Test is used, with consequent data. Furthermore, to avoid any preference made by the binary classifier, the training dataset is balanced through the elimination of samples to achieve equal 1s and 0s occurrences.

```
def preprocess_data(data, ...):
    ...
    data = data.to_numpy()
    n = len(data)
    train_x = []
    val_x = []
    test_x = []
    num_train = int(split*n)
    num_validation = int(n*validation_split) # 0 sized by default

    # Split
    train_x = data[0:num_train]
    val_x = data[num_train:num_train + num_validation]
        if(validation_split) else np.empty(shape=(0,0))
    test_x = data[num_train + num_validation:]
    ...
    # Buys vs Holds
    n_buys = train_y.sum()
    n_holds = n - n_buys

    if n_buys > n_holds:
        n_delete = n_buys - n_holds
        j = 1
    else:
        n_delete = n_holds - n_buys
        j = 0
    ...
    # Delete exceeding label
    while(n_delete > 0):
            for i in range(len(train_x)):
                if(train_y[i] == j):
                    train_x = np.delete(train_x, i, axis = 0)
                    train_y = np.delete(train_y, i, axis = 0)
                    n_delete -= 1
                    break
```

Finally, we shuffle the train data to get more variance and windowing independence:

```
shuffled = list(zip(train_x, train_y))
shuffle(shuffled)
(train_x, train_y) = zip(*shuffled)
train_x, train_y = zip(*shuffled)
```

2.5 Data Normalization

It is important to scale features before training a machine learning model. Normalization is a common way of doing this scaling: subtract the mean and divide by the standard deviation of each feature. The mean and standard deviation should only be computed using the training data so that the models have no access to the values in the validation and test sets.
It's also arguable that the model shouldn't have access to future values in the training set when training, and that this normalization should be done using moving averages. But, in the interest of simplicity, this implementation uses a simple average using the StandardScaler found in the *sklearn* module.

```
def preprocess_data(data, ...):
    ...
    for i in range(train_x.shape[1]-1) :
        X = train_x[:, i].reshape(-1,1)
        scaler = StandardScaler().fit(X)
        train_x[:, i] = scaler.transform(X).reshape(1,-1)

    for i in range(test_x.shape[1]-1):
        X = test_x[:, i].reshape(-1,1)
        scaler = StandardScaler().fit(X)
        test_x[:, i] = scaler.transform(X).reshape(1,-1)
```

3 Ensemble Model

Once all the data has been processed, enriched and normalized it has to be fed to our model. The dataset has the format (samples, window_size, features) and it's too big and too complex to be used by an SVM. To solve this problem, the Support Vector Machine will be used as a stacking model that will use features extracted from various Neural Networks.

3.1 Activation Ensemble

As shown in [10], using a Support Vector Machine as a stacking classifier of neural network activation improves accuracy with respect to common ensemble techniques such as Majority Voting or Sum Rule.
This method uses numerical activations found in deep layers of neural networks as a single feature vector to train and test a single Support Vector Machine.

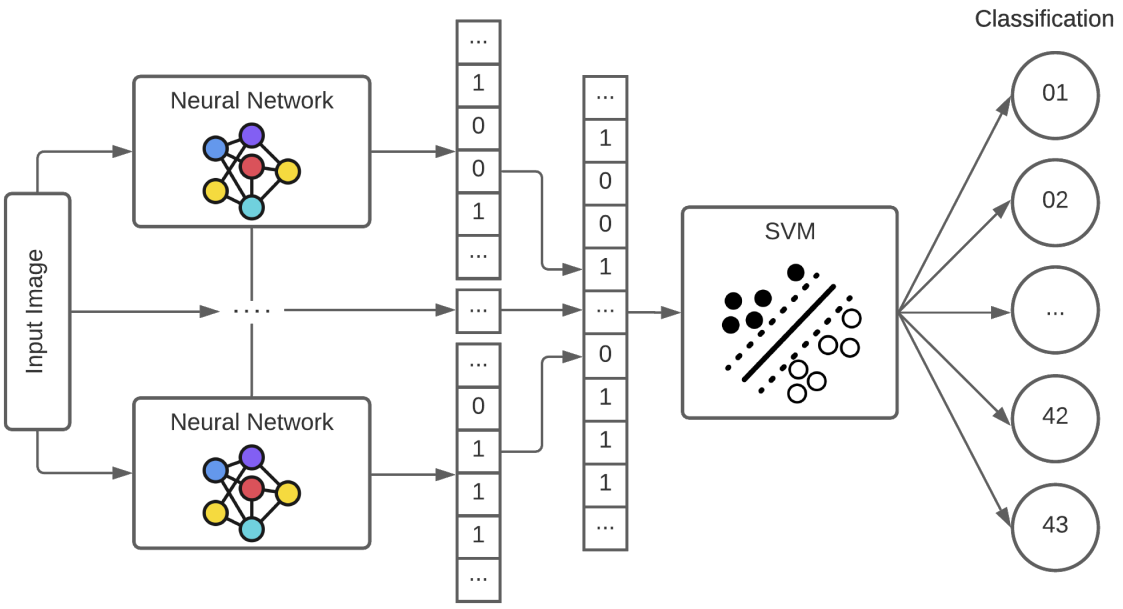


Figure 3: Activation Ensemble

In Figure, the simple process of Activation Ensemble on an Image Classification task.

3.2 Neural Networks

In time series forecasting, multiple typologies of neural models are employed. To maximize the chance of obtaining a more accurate SVM, we have to aim to get the most statistically independent components. One way of doing that is by operating with different neural network models.
In the following subsections, we'll introduce the used ones.

3.2.1 Fully Connected Network

Fully connected neural networks (FCNNs) are a type of artificial neural network where the architecture is such that all the nodes, or neurons, in one layer, are connected to the neurons in the next layer.

These networks represent the simplest and most common model of implementing a neural network, but they are still used in time series forecasting [5] and thanks to the Universal Approximation Theorem of Neural Network, if the data can be described by a function (hopefully it can) we can exploit them and obtain high accuracies.

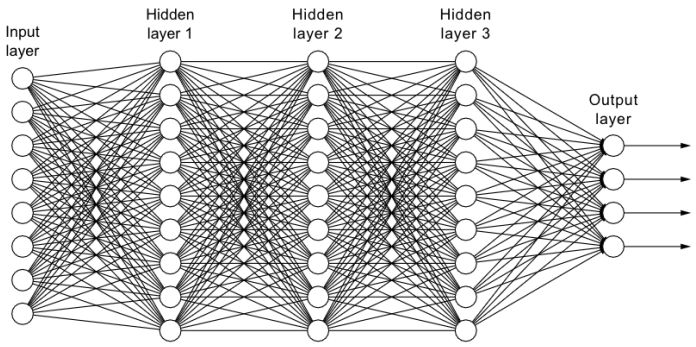


Figure 4: Fully Connected Network

3.2.2 Convolutional Neural Network

Convolutional neural networks have their roots in image processing. It was first published in LeNet to recognize the MNIST handwritten digits. However, convolutional neural networks are not limited to handling images. CNNs can also be used in time series forecasting [2][7]

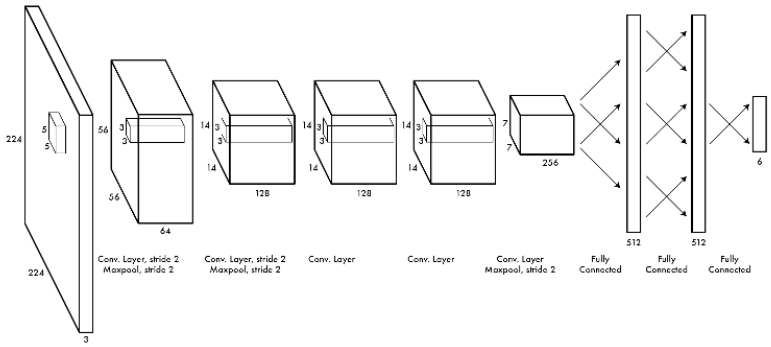


Figure 5: Convolutional Network

3.2.3 LSTM - Long Short Time Memory

The Long Short-Term Memory (LSTM) model is the elegant Recurrent neural network's variant. A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed or undirected graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior.
LSTM uses the purpose-built LSTM's memory cell to represent the long-term dependencies in time series data [12]. In addition, LSTM is introduced to solve the vanishing gradient problem of RNN in case of long-term context memorization is required [11].

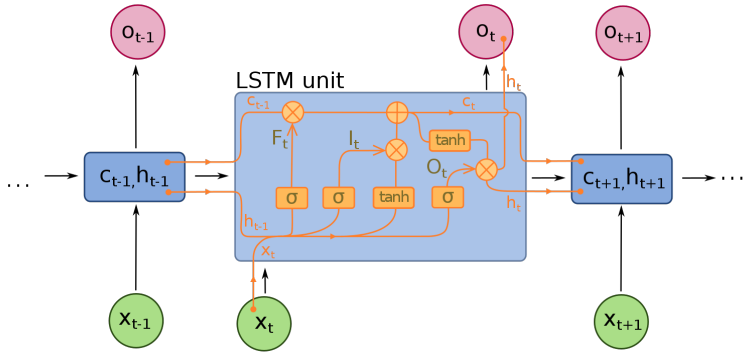


Figure 6: LSTM Network

Following, the topology of the networks implemented with window=24 and time-frame=1h.

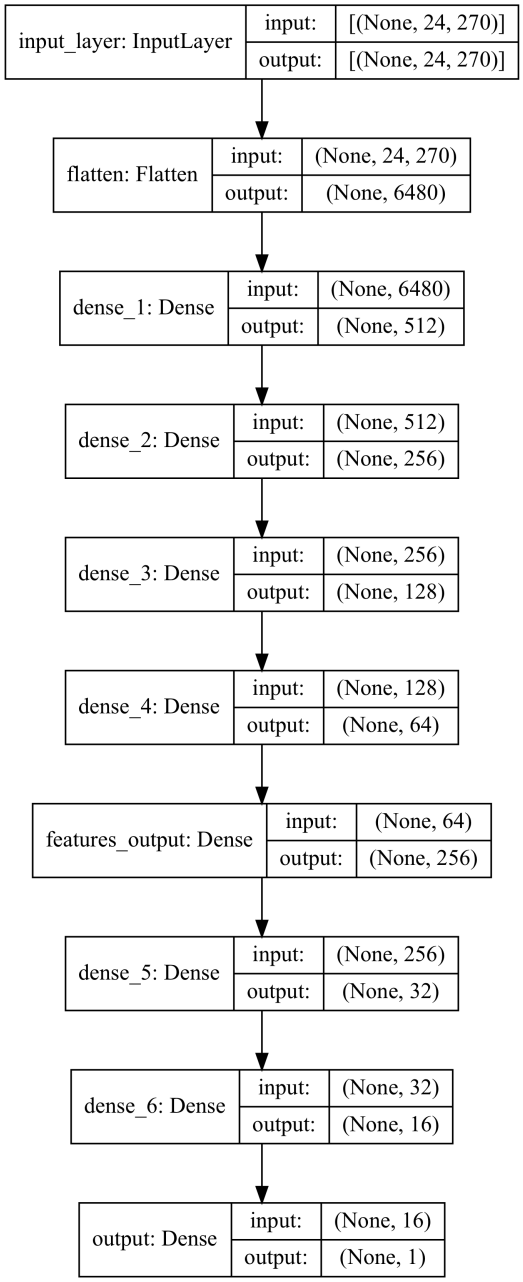


Figure 7: Fully Connected Model

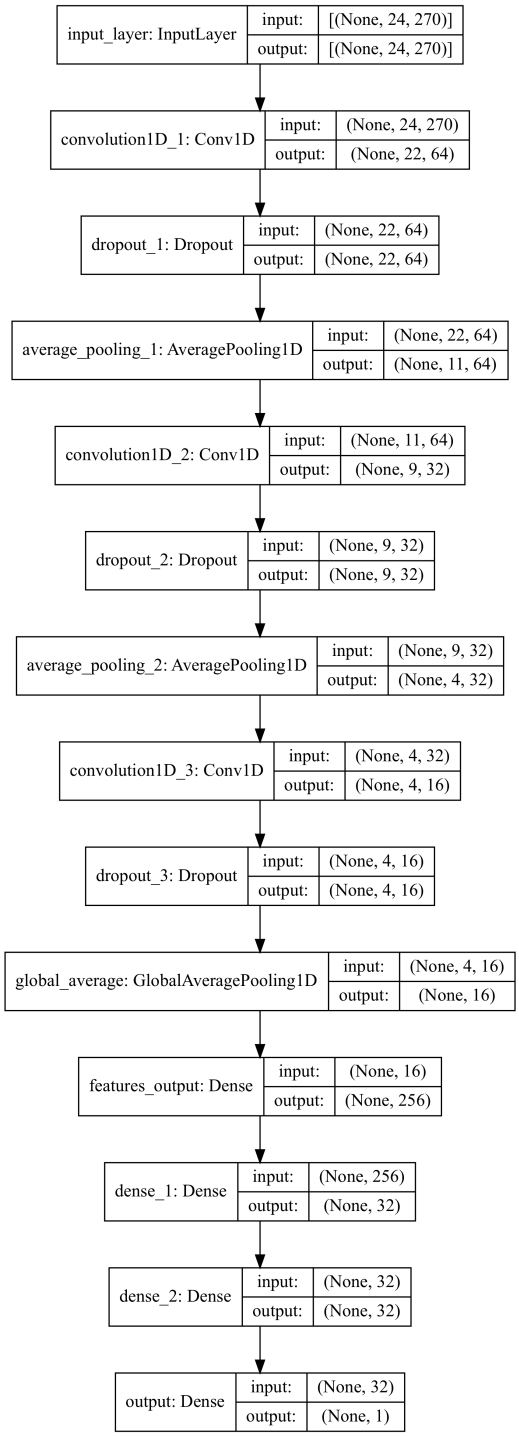


Figure 8: Convolutional Model

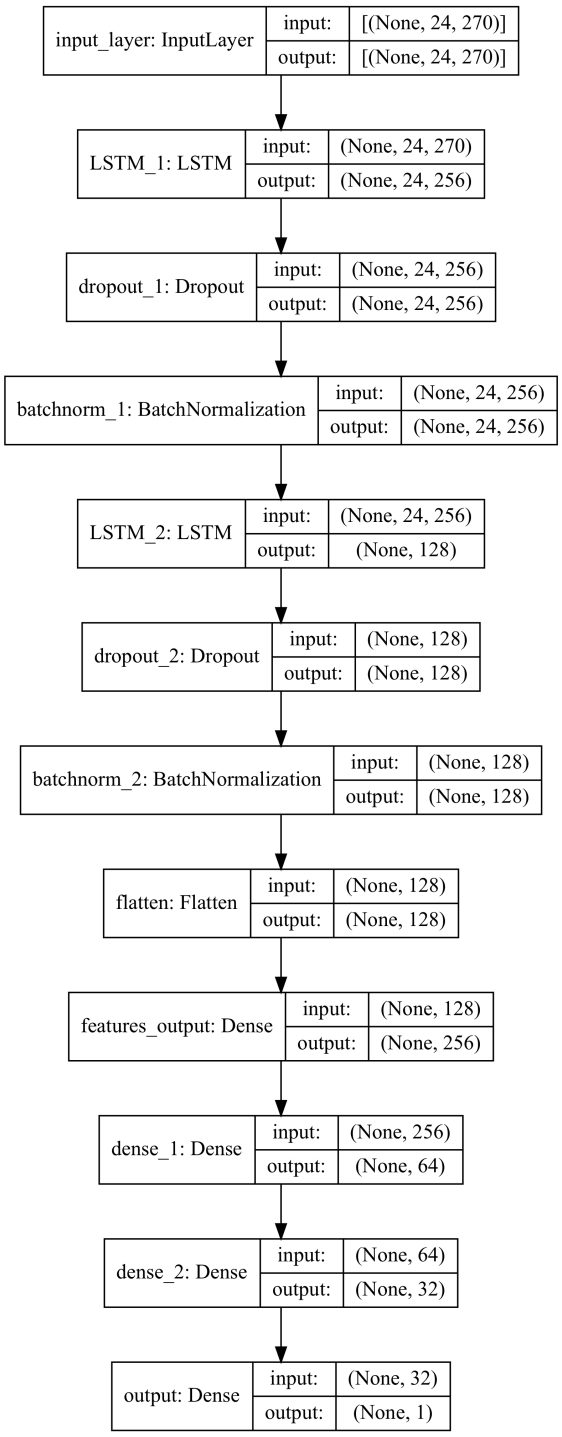


Figure 9: LSTM Model

3.3 Feature Extraction

Once every model has been trained and tested, we use the layered structure of Tensorflow neural network implementation to define 3 new models, 'feature_extractor' models.

```
keras.Model(
    inputs=model[i].inputs,
    outputs=model[i].layers[-4].output
))
```

Here, these new models have the input layer equal to the models trained, but the output is chosen to be a Dense layer with 256 output neurons, with the idea of concatenating this numerical 256-vectors.

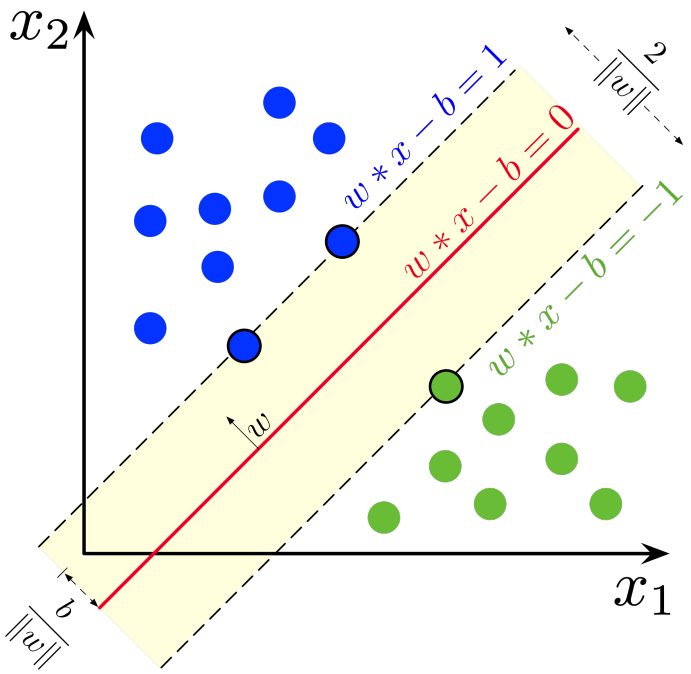


Figure 10: SVM working on bi-dimensional vectors

4 Experiments

In this section, we evaluate the performance of the proposed SVM architecture for stock pricing prediction.
`project.ipynb` explains all the lines of code used in this chapter.

4.1 Baseline

Like any good research, we have to start from what we can achieve with a 'naive' model.
Let's train an SVM model from a flattened representation of the dataset.

```
# Flatten training set
base_x = train_x.reshape(train_x.shape[0], -1)
classifier = svm.SVC()
classifier.fit(base_x, train_y[:,0])
```

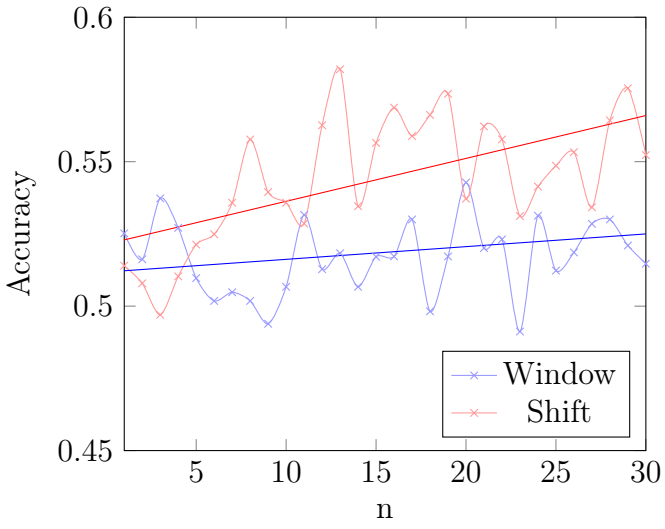
Afterwards, let's test this naive SVM on the testing dataset. The accuracy obtained is listed in the table below.

Accuracy	54.20%

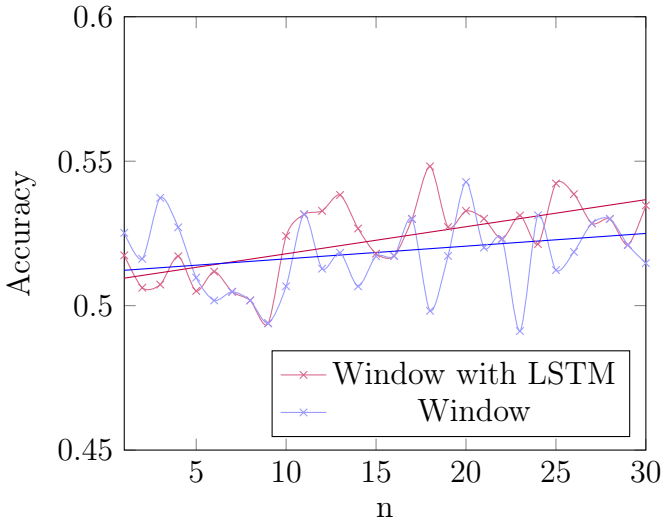
4.2 Hyper-Parameter Tuning

In this time series forecasting, we have two important hyperparameters to be tuned: window size and time-shift. On one hand, the former impacts the overall training time, but a bigger window size may carry more information to be understood by the model. The latter, on the other hand, it's a bit tricky but bigger time shifts may be influenced by trends that can be caught by our model from the multiple features.

With that in mind, we trained a simple one-hidden-layer neural network and tested its accuracy on different values of those hyperparameters with the following results.



From the above graph, we can observe that having a larger shift indeed results in higher accuracy, but the model accuracy seems to be independent from the window's value. That might be caused from our test model being too simple to exploit its benefit. To see if a better model will make good use of a time window, let's train a simple LSTM model:



The red line, the interpolation of the results on window size using an LSTM as the model, has a better slope, so we can assume that using a window size of 20 will improve our chances to achieve better results.

4.3 Network Training

Training network is by far the longest process of the entire research, even for simpler models shown in Neural Networks. There are multiple factors to be considere, such as *learning rate*, *batch size*, *optimizer*, etc.

In order to get the maximum variance and best parameters, 'experiments.py' implements the entire training-testing-evaluating process defined with a batch of parameters and a number of iterations. Once per iteration, networks are trained with some random combination of hyperparameters (training converges only with learning-rate in $[0.001, 0.01]$). Sample output is the following:

```
---------------------- ITERATION 1 ----------------------

Learning Rate: 0.01, Batch Size: 128, Epochs: 40


----------- Train -----------
Accuracy FullyConnected Network: 0.6972
Accuracy LSTM Network: 0.9575
Accuracy CNN Network: 0.6265


----------- Test -----------
Accuracy FullyConnected Network: 0.5233
Accuracy LSTM Network: 0.5497
Accuracy CNN Network: 0.5248


----------- SVM -----------
Accuracy: 0.5368
```

After n iterations, a recap follows:

```
---------------------- RECAP ----------------------


----------- Train -----------
FullyConnected Network:
     Max: 0.7397, Min: 0.6425, Avg: 0.6822, Std: 0.0256
LSTM Network:
     Max: 0.9960, Min: 0.9346, Avg: 0.9691, Std: 0.0148
CNN Network:
     Max: 0.6644, Min: 0.5866, Avg: 0.6276, Std: 0.0174

----------- Test -----------
FullyConnected Network:
     Max: 0.5591, Min: 0.4842, Avg: 0.5265, Std: 0.0178
LSTM Network:
     Max: 0.5695, Min: 0.5142, Avg: 0.5384, Std: 0.0137
CNN Network:
     Max: 0.5467, Min: 0.5004, Avg: 0.5252, Std: 0.0122

----------- SVM -----------
     Max: 0.5418, Min: 0.5036, Avg: 0.5225, Std: 0.0094
```

These are the results obtained after 30 iterations of the entire phase with a fixed learning rate of 0.01 and batch size of 256 samples:

Model	**Accuracy**			
	Max	Min	Avg	Dev
FC	72.48%	64.10%	68.12%	0.02332
LSTM	99.11%	94.35%	97.43%	0.01642
CNN	64.20%	59.84%	62.11%	0.01951

Table 1: Network Training

As Table 1 clearly states, either we get a big overfit over the training samples, or we've found a recipe for getting rich.
Testing these trained networks over the testing set, we get these results:

Model	**Accuracy**			
	Max	Min	Avg	Dev
FC	54.11%	50.01%	52.72%	0.016102
LSTM	57.23%	52.25%	54.23%	0.01181
CNN	54.28%	50.78%	52.28%	0.01351

Table 2: Network Testing

It's was overfit after all!
This is of course an indicator of the complexity of the matter at hand.

4.4 SVM Training

Once the models are trained, we now transform those models by removing the final outer layers, with the following procedure:

```
feature_extractor = keras.Model(
    inputs=model.inputs,
    outputs=model.layers[-4].output
)
```

Subsequently, we feed the entire training set to the 3 feature extractor and we store the outputs, concatenating them. Putting it all together becomes:

```
for i in range(num_nets):
    features.append(keras.Model(
        inputs=model[i].inputs,
        outputs=model[i].layers[-4].output
    ))
    features_x.append(features[i](train_x).numpy())

features_conc = np.concatenate((features_x), axis=1)
```

Returning the augmented training dataset with shape: $(n, 768)$, where n is the number of samples.

Next, we've implemented a sklearn.GridSearch to fit the best SVM estimator, as stated in the official guide, this is what GridSearch does:

> *Exhaustive search over specified parameter values for an estimator. Important members are fit, predict.*
> *GridSearchCV implements a "fit" and a "score" method.*
> *The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.*

```
param_grid = {'C': [0.01, 0.1, 1, 10, 100],
              'kernel': ['rbf', 'linear','sigmoid']}
classifier = GridSearchCV(svm.SVC(), param_grid)
```

Most of the time, the best classificator had this characteristics:

```
SVC(C=0.01, class_weight='balanced', kernel='linear')
```

Where the most important characteristic is the linear kernel. This usually happens when the feature space is big enough to avoid the need for a non-linear kernel. Testing these classificators, we obtained these results:

Model	**Accuracy**			
	Max	Min	Avg	Dev
SVM	54.25%	50.10%	52.12%	0.01068

Although SVM reached very high accuracies on the training set (∼97%), in the testing set, as expected, no fireworks.

5 Conclusion

What can we derive from the obtained results?

Once again, trying to predict the market ended in a failure. Around 50%of accuracy in a binary classification problem is the mathematical equivalent of random guesses. Although Support Vector Machine is a robust model for binary classification, the task they were assigned to was simply too complex.

Stock prediction has always been a target for many years, now with machine learning tools more than ever. But it seems that every approach fails, why is that?
There is no answer, as E. Dijkstra said: "Program testing can be used to show the presence of bugs, but never to show their absence!", meaning that we can just keep trying new solutions, hoping to find one, if there is one.
Still, one of the most famous hypotheses on the market predictability, the *Efficient Market Hypothesis* [9], states that stock prices are a function of public knowledge and reasonable expectations, and newly revealed information about a company's future is reflected in the present stock price virtually immediately. This would indicate that all publicly available information about a company, including its price history, is already included in the stock's current price. As a result, stock price variations reflect the release of new information, market changes in general, or random movements around the value that represents the current information set. Burton Malkiel, in his influential 1973 work A Random Walk Down Wall Street, claimed that stock prices could therefore not be accurately predicted by looking at price history.
Several empirical tests (such as ours) support the notion that the theory applies generally.

References

[1] John Bollinger. "Using bollinger bands". In: *Stocks & Commodities* 10.2 (1992), pp. 47–51.

[2] Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. "Conditional time series forecasting with convolutional neural networks". In: *arXiv preprint arXiv:1703.04691* (2017).

[3] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[4] Harris Drucker et al. "Support vector regression machines". In: *Advances in neural information processing systems* 9 (1997), pp. 155–161.

[5] CR Gent and CP Sheppard. "Predicting time series by a fully connected neural network trained by back propagation". In: *Computing & Control Engineering Journal* 3.3 (1992), pp. 109–112.

[6] FR Johnston et al. "Some properties of a simple moving average when applied to forecasting a time series". In: *Journal of the Operational Research Society* 50.12 (1999), pp. 1267–1271.

[7] Irena Koprinska, Dengsong Wu, and Zheng Wang. "Convolutional neural networks for energy time series forecasting". In: *2018 international joint conference on neural networks (IJCNN)*. IEEE. 2018, pp. 1–8.

[8] AJ Lawrance and PAW Lewis. "An exponential moving-average sequence and point process (EMA1)". In: *Journal of Applied Probability* 14.1 (1977), pp. 98–113.

[9] Burton G Malkiel. "Efficient market hypothesis". In: *Finance*. Springer, 1989, pp. 127–134.

[10] Filippo Manzardo. "Classificazione di segnali stradali tramite ensemble di reti neurali". In: (2021).

[11] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. PMLR. 2013, pp. 1310–1318.

[12] Alaa Sagheer and Mostafa Kotb. "Time series forecasting of petroleum production using deep LSTM recurrent networks". In: *Neurocomputing* 323 (2019), pp. 203–213.