MACHINE LEARNING REPORT

Odyssefs Diamantopoulos-Pantaleon 3180049

Ilias-Marios Stogiannidis 3180178

Report

Machine Learning

Professor Panagiota Tzintza

Athens University of Economic and Business

FIRST STEPS

We were tasked to evaluate the Linear and Logistic Regression models and a Neural Network performance on the prediction of Bitcoin price data. The Bitcoin data was found here https://finance.yahoo.com/quote/BTC-USD/history?p=BTC-USD and we downloaded the data for the dates in the range of 11/9/2017 to 12/27/2021. In the Jupyter Notebook there are extended comments about the project, however we are going to extensively analyze it in this report. Firstly, we imported the dataset to our program. Then we decided to check the variables. We discovered that the Close and the Adj Close column have the exact same values.

```
In [4]: 1 (btc_data['Close'] - btc_data['Adj Close']).sum()
Out[4]: 0.0
```

We also checked that there are no null values.

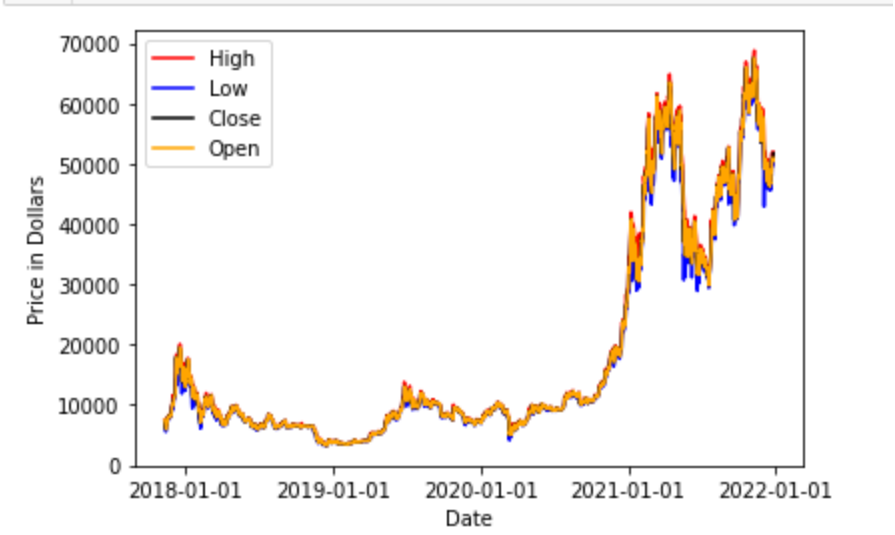
```
In [5]: 1 btc_data.isna().sum()
Out[5]: Date         0
        Open         0
        High         0
        Low          0
        Close        0
        Adj Close    0
        Volume       0
        dtype: int64
```

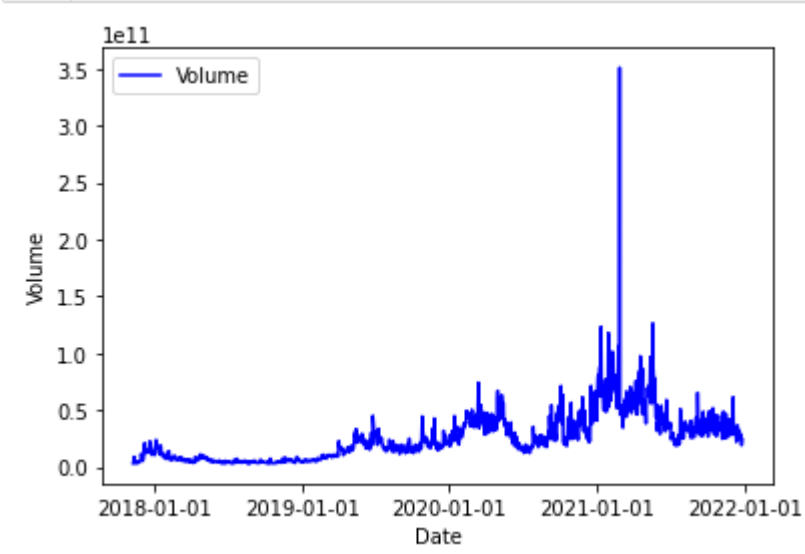
Furthermore, we checked about outliers, more specifically negative values that would not make sense to exist in our dataset.

```
In [37]: 1 btc_data.query('0>=Open or 0>=High or 0>=Low or 0>=Close or 0 >=Volume')
Out[37]:
```

Date	Open	High	Low	Close	Adj Close	Volume	PosNegClosing

We also visualized the data in two different plots. The first plot contained the Open, High, Low, Close and Adj Close data that share the same unit of measurement, while the second plot will have the Volume column.





Moreover, we normalized the data by using the MinMax scaler and also found the correlation of the variables:

In [10]:
```
1 corr = btc.corr()
2 corr
```

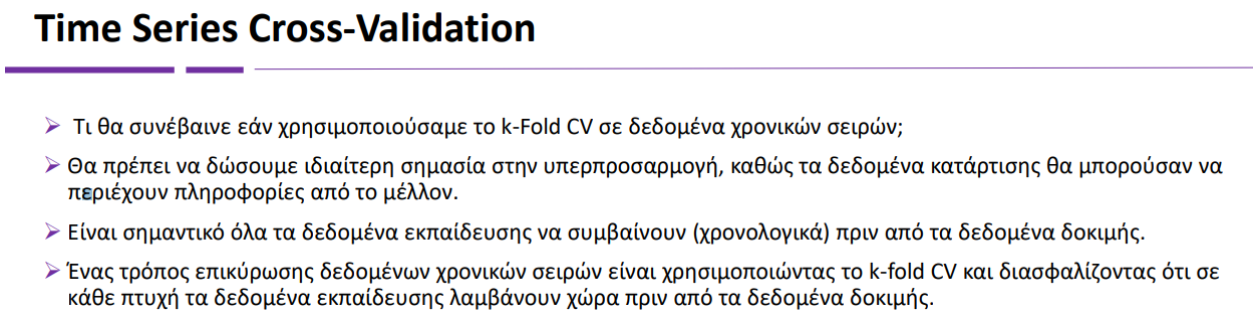
Out[10]:

	Close	Adj Close	Open	High	Volume	Low
Close	1.000000	1.000000	0.998335	0.999278	0.607540	0.999138
Adj Close	1.000000	1.000000	0.998335	0.999278	0.607540	0.999138
Open	0.998335	0.998335	1.000000	0.999333	0.609332	0.998689
High	0.999278	0.999278	0.999333	1.000000	0.614514	0.998625
Volume	0.607540	0.607540	0.609332	0.614514	1.000000	0.597926
Low	0.999138	0.999138	0.998689	0.998625	0.597926	1.000000

We see that there are extremely high correlations between almost all the variables. Also, we confirmed that the Adj Close and the Close have the same values since their correlation have 1 value.

LINEAR REGRESSION

We are going to use the Linear regression model of the sklearn.linear_model library and train it with the data we have. We have already normalized the data so we now have to do the Training and the Diagnostics part. In order to correctly train it and perform cross validation, we have to carefully decide how to split the data, because we should not train the model with data from the future in order to predict the past.



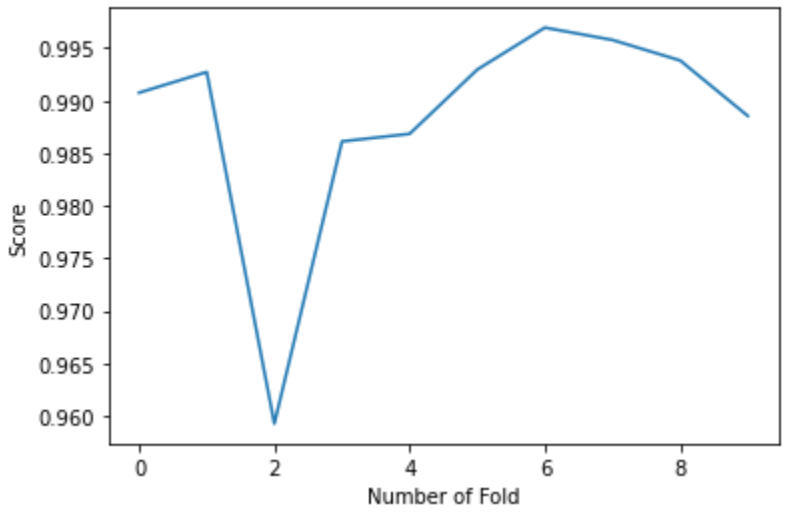
In other words, we have to make sure that the data that we train the model are before chronologically than the data that we are going to predict. In order to manage this we use the TimeSeriesSplit method and we perform 10 splits. However, we also set the max_train_size to 365 because each year has 365 days and the test_size to 100 in order to predict the next 100 days.

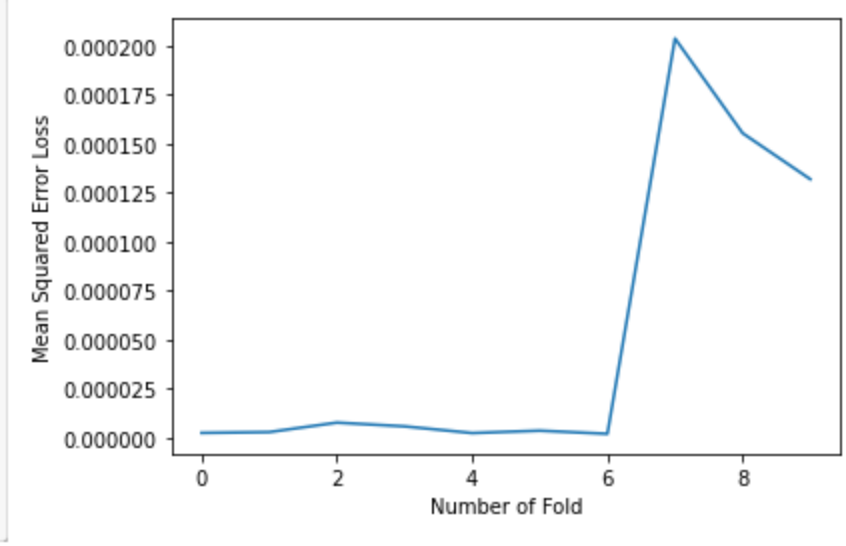
```
kf = TimeSeriesSplit(n_splits=10,max_train_size =365,test_size=100) #365 because of the dates and 100 days to predict
```

In order to evaluate the model, we use the score method and the mean squared error function. The score method simply calculates the correct guessed/total guessed and the mean squared error is the best loss function. An example of one run is the following:

```
1 of TimeSeriesSplit 10
Score: 0.9907528792808834
Mean squared error: 2.2613461109581828e-06
2 of TimeSeriesSplit 10
Score: 0.9927226304747312
Mean squared error: 2.6739667829362036e-06
3 of TimeSeriesSplit 10
Score: 0.9592762081592875
Mean squared error: 7.50445157832767e-06
4 of TimeSeriesSplit 10
Score: 0.9861227803495584
Mean squared error: 5.595529491778814e-06
5 of TimeSeriesSplit 10
Score: 0.9868412970618526
Mean squared error: 2.1851490249901053e-06
6 of TimeSeriesSplit 10
Score: 0.992954254791606
Mean squared error: 3.3933837497469295e-06
7 of TimeSeriesSplit 10
Score: 0.9969458212707586
Mean squared error: 1.7838096589038757e-06
8 of TimeSeriesSplit 10
Score: 0.995771516687892
Mean squared error: 0.00020357100644073124
9 of TimeSeriesSplit 10
Score: 0.9938222110137763
Mean squared error: 0.00015522649339473843
10 of TimeSeriesSplit 10
Score: 0.9885469941353374
Mean squared error: 0.00013174246602562563
```

```
plt.show()
```





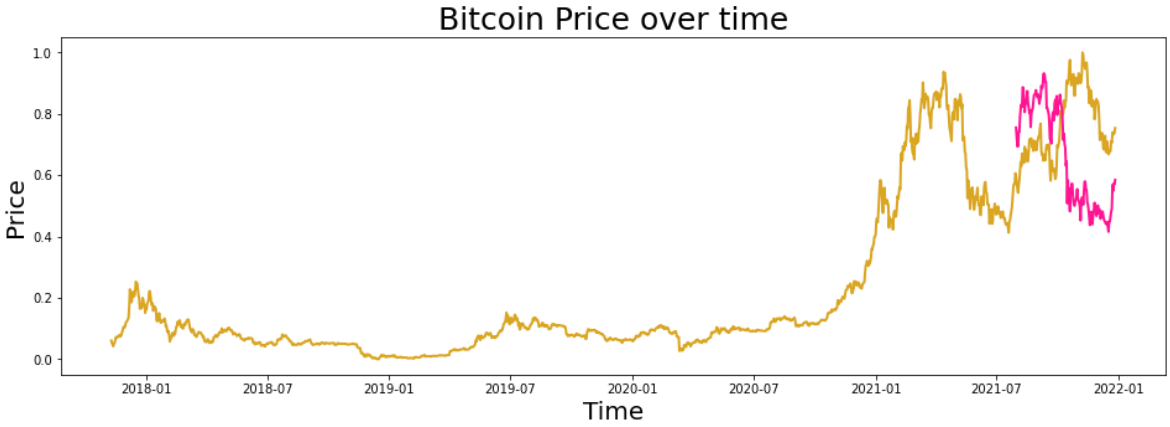
We see that the score is extremely high (0.95-0.99) and the loss is very low. We were expecting to have great performance since according to https://www.ncbi.nlm.nih.gov/pmc/articles/PMC374386/ the Correlation quantifies the strength of the linear relationship between a pair of variables, whereas regression expresses the relationship in the form of an equation. In other words, since our variables had extremely high correlation then the linear model would be suitable for predictions on this dataset.

Additionally, we decided to predict the 10% last values of the dataset and plot them together with the actual values. More specifically, the model has 1510 dates, which means that we took the last 151 days. We shifted the dataset by the number of days and then we predicted the last days with the help of our model:

```
In [15]: 1 future_set = btc.shift(periods=150).tail(150)
```

```
In [16]: 1 prediction = lr.predict(future_set[['Open','High','Volume','Low']])
```

Finally, we plotted the results predicted along with the actual prices to see how our model performed:



LOGISTIC REGRESSION

The logistic regression method is used in order to classify data into categories. In our dataset we do not have certain classes, so we have to create some. The two classes that we are going to create are going to represent if the bitcoin price rose higher during the day or dropped. In order to do that we are going to create a new column

> Creating a new column that contains the difference of the openning with the closing price.

```
1 btc_data['PosNegClosing'] = btc_data['Open']- btc_data['Close']
```

Then we are going to put false to the negative values and true to the positive values.

> Then putting to all negative values the value 0 and to all posivite the value 1 in order to have a column with binary data.

```
1 btc_data['PosNegClosing'].mask(btc_data['PosNegClosing']>0,1,inplace=True)
```

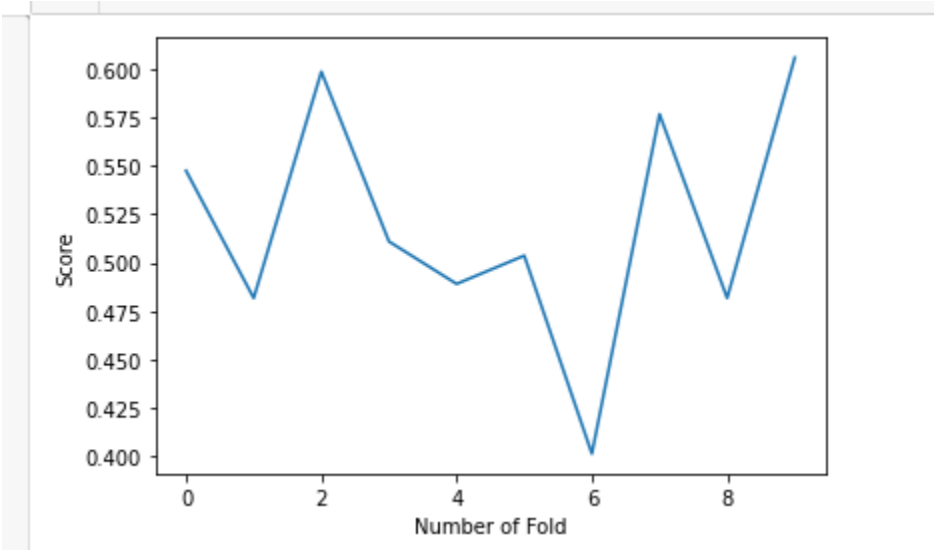
```
1 btc_data['PosNegClosing'].mask(btc_data['PosNegClosing']<0,0,inplace=True)
```

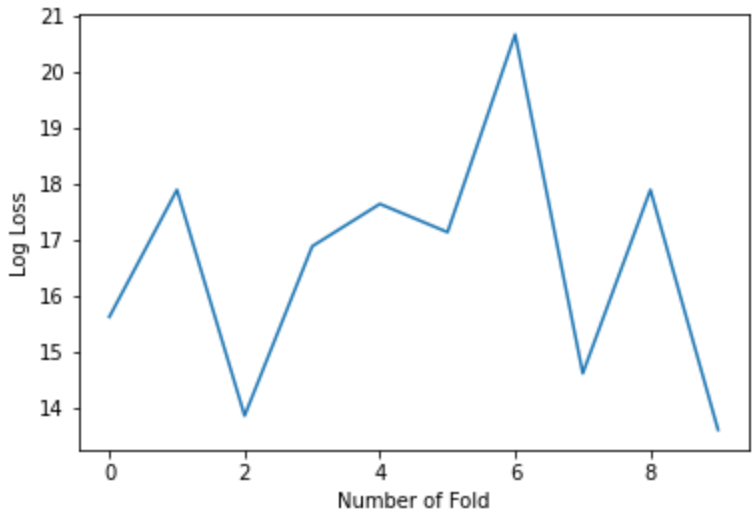
In order to train the model we are going to use the TimeSeriesSplit for the same reasons that were stated in the Linear regression part and we are going to evaluate using the same score method as in the Linear regression and the log_loss function.

```
kf = TimeSeriesSplit(n_splits=10,max_train_size =365,test_size=10) #the bigger the test_size teh worst the performance
scores =[]
```

One thing that we have to mention is that we chose for the model to predict fewer days because we noticed that its performance was getting worser the more days we asked it to predict. Let's now see an example of a run:

```
1 of TimeSeriesSplit 10
Score: 0.4
Log Loss: 20.72326583694641
2 of TimeSeriesSplit 10
Score: 0.8
Log Loss: 6.907835238725157
3 of TimeSeriesSplit 10
Score: 0.6
Log Loss: 13.815830396936352
4 of TimeSeriesSplit 10
Score: 0.6
Log Loss: 13.815830396936352
5 of TimeSeriesSplit 10
Score: 1.0
Log Loss: 9.992007221626413e-16
6 of TimeSeriesSplit 10
Score: 0.7
Log Loss: 10.361872797702265
7 of TimeSeriesSplit 10
Score: 0.7
Log Loss: 10.361872797702265
8 of TimeSeriesSplit 10
Score: 0.8
Log Loss: 6.907915198468176
9 of TimeSeriesSplit 10
Score: 0.4
Log Loss: 20.72326583694641
10 of TimeSeriesSplit 10
Score: 0.7
Log Loss: 10.361632918473205
```





As we can see the Logistic regression method is not performing well. This is to be expected since according to this thread in Stack exchange (https://stats.stackexchange.com/questions/250376/feature-correlation-and-their-effect-of-logistic-regression) and the lesson of regression methods of PennState (https://online.stat.psu.edu/stat501/lesson/12/12.3) the higher the correlation of the data the worse the results of the logistic regression.

NEURAL NETWORK

We decided to create a Recurrent neural network. To do this we have to first declare some values. More specifically:

```
# number of past observations to be considered for the LSTM training and prediction
n_past = 50

# number of future datapoints to predict (if higher than 1, the model switch to Multi-Step)
n_future = 1

# activation function used for the RNN (softsign, relu, sigmoid)
activation = 'relu'

# dropout for the hidden layers
dropout = 0.2

# number of hidden layers
n_layers = 2

# number of neurons of the hidden layers
n_neurons = 50

# features to be considered for training (if only one is Close, then its Univariate, if more, then it's Multivariate)
features = ['Close', 'Volume']
#features = ['Close']

# number of inputs features (if higher than 1, )
n_features = len(features)

# patience for the early stopping (number of epochs)
patience = 25

# optimizer (adam, RMSprop)
optimizer='adam'
```

As suggested, we used the 50 past days to predict the next 1 day. We also used the activation method “Relu” and we included 3 hidden layers with 50 neurons in each layer. We also decided to keep only the Close column since we want to predict based on the closing values and the Volume column since it contains different information.

```
:
1 df = btc_data.set_index('Date')[features]
2 df = df.set_index(pd.to_datetime(df.index))
3 df.dropna(inplace=True)
```

Then we normalized the data as suggested with the MinMaxScaler

```
# scale
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
print('training_set_scaled.shape: ', training_set_scaled.shape)
training_set_scaled
```

```
ining_set_scaled.shape:  (1510, 2)

ay([[0.06073083, 0.00086937],
    [0.05256296, 0.00656405],
    [0.0485129 , 0.00570333],
    ...,
    [0.73360872, 0.04627854],
    [0.73951041, 0.05183451],
    [0.75376818, 0.06094721]])
```

Then we have to create our training datasets. To do that, just like before, we have to ensure that no future data is used to predict past data. Therefore, we are going to match 50 past days to 1 future day with the following code:

```
# creating a data structure with 50 timesteps and 1 output
X_train = []
y_train = []

for i in range(n_past, len(training_set_scaled) - n_future + 1):
    X_train.append(training_set_scaled[i-n_past:i, :])
    y_train.append(training_set_scaled[i:i+n_future, 0])

X_train, y_train = np.array(X_train), np.array(y_train)
X_train.shape, y_train.shape
```

Then we start building the model:

```
# Building the RNN

# Initialising the RNN
regressor = Sequential()

# Input layer
regressor.add(LSTM(units=n_past, return_sequences=True, activation=activation, input_shape=(X_train.shape[1], n_features)))
#regressor.add(LSTM(units=neurons, return_sequences=True, activation=activation, input_shape=(X_train.shape[1], 1)))

# Hidden layers
for _ in range(n_layers):
    regressor.add(Dropout(dropout))
    regressor.add(LSTM(units=n_neurons, return_sequences=True, activation=activation))

# Last hidden layer (changing the return_sequences)
regressor.add(Dropout(dropout))
regressor.add(LSTM(units=n_neurons, return_sequences=False, activation=activation))

# Adding the output layer
regressor.add(Dense(units=n_future))

# Compiling the RNN
regressor.compile(optimizer=optimizer, loss='mean_squared_error')

# Model summary
regressor.summary()
```

The model contains an input layer that gets the training data that we previously created. Then we use 2 hidden layers that contain the Dropout and the LSTM filters(https://keras.io/api/layers/recurrent_layers/lstm/) and then have a final hidden layer that will set the return sequence to false in order to return only the last output. Lastly, we are going to have the output layer that will give us the final verdict.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 50, 50)            10600

 dropout_3 (Dropout)         (None, 50, 50)            0

 lstm_5 (LSTM)               (None, 50, 20)            5680

 dropout_4 (Dropout)         (None, 50, 20)            0

 lstm_6 (LSTM)               (None, 50, 20)            3280

 dropout_5 (Dropout)         (None, 50, 20)            0

 lstm_7 (LSTM)               (None, 20)                3280

 dense_1 (Dense)             (None, 1)                 21

=================================================================
Total params: 22,861
Trainable params: 22,861
Non-trainable params: 0
```

In case the training goes on for a long time we have set an early stopping mechanism that stops the process as soon as val_loss reaches the minimum value in order to avoid overfitting.

```
# Adding early stopping
early_stop = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=patience)
```

Then we are going to fit the data to the model.

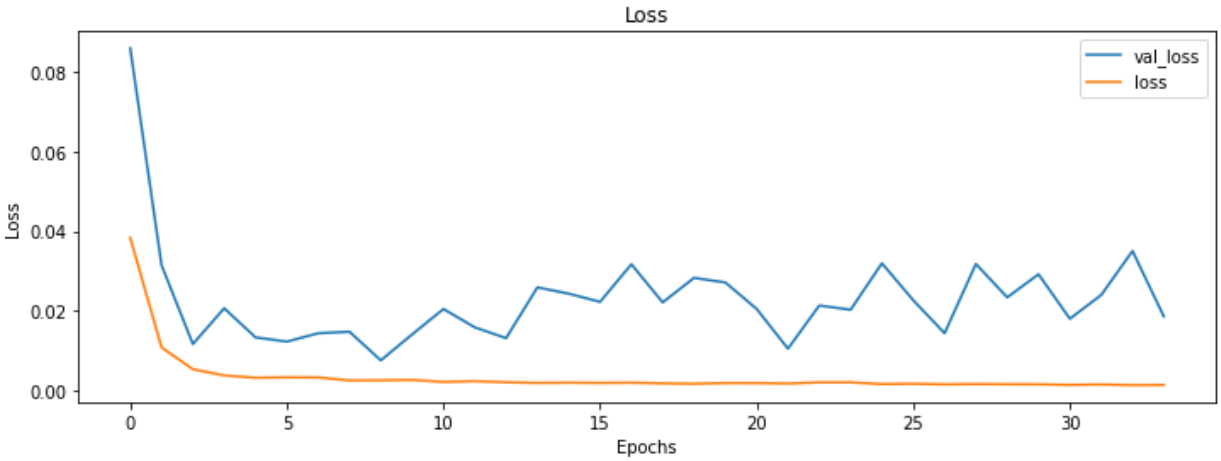
```
# Fitting the RNN to the Training set
res = regressor.fit(X_train, y_train
                    , batch_size=50
                    , epochs=50
                    , validation_split=0.1
                    , callbacks=[early_stop]
                   )
```

The batch size is 50 as suggested and we have put a few more epochs, because we noticed better performance.

An example of a run is the following:

```
42/42 [==============================] - 2s 48ms/step - loss: 0.0022 - val_loss: 0.0205
Epoch 12/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0024 - val_loss: 0.0159
Epoch 13/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0021 - val_loss: 0.0132
Epoch 14/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0020 - val_loss: 0.0259
Epoch 15/50
42/42 [==============================] - 2s 46ms/step - loss: 0.0020 - val_loss: 0.0243
Epoch 16/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0020 - val_loss: 0.0223
Epoch 17/50
42/42 [==============================] - 2s 48ms/step - loss: 0.0020 - val_loss: 0.0317
Epoch 18/50
42/42 [==============================] - 2s 46ms/step - loss: 0.0019 - val_loss: 0.0221
Epoch 19/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0018 - val_loss: 0.0283
Epoch 20/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0019 - val_loss: 0.0271
Epoch 21/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0019 - val_loss: 0.0205
Epoch 22/50
42/42 [==============================] - 2s 49ms/step - loss: 0.0018 - val_loss: 0.0105
Epoch 23/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0021 - val_loss: 0.0214
Epoch 24/50
42/42 [==============================] - 2s 46ms/step - loss: 0.0021 - val_loss: 0.0203
Epoch 25/50
42/42 [==============================] - 2s 48ms/step - loss: 0.0017 - val_loss: 0.0319
Epoch 26/50
42/42 [==============================] - 2s 49ms/step - loss: 0.0018 - val_loss: 0.0227
Epoch 27/50
42/42 [==============================] - 2s 48ms/step - loss: 0.0016 - val_loss: 0.0144
Epoch 28/50
42/42 [==============================] - 2s 49ms/step - loss: 0.0017 - val_loss: 0.0318
Epoch 29/50
42/42 [==============================] - 2s 48ms/step - loss: 0.0016 - val_loss: 0.0234
Epoch 30/50
42/42 [==============================] - 2s 46ms/step - loss: 0.0016 - val_loss: 0.0292
Epoch 31/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0015 - val_loss: 0.0180
Epoch 32/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0016 - val_loss: 0.0240
Epoch 33/50
42/42 [==============================] - 2s 48ms/step - loss: 0.0014 - val_loss: 0.0350
Epoch 34/50
42/42 [==============================] - 2s 47ms/step - loss: 0.0015 - val_loss: 0.0186
Epoch 00034: early stopping
```

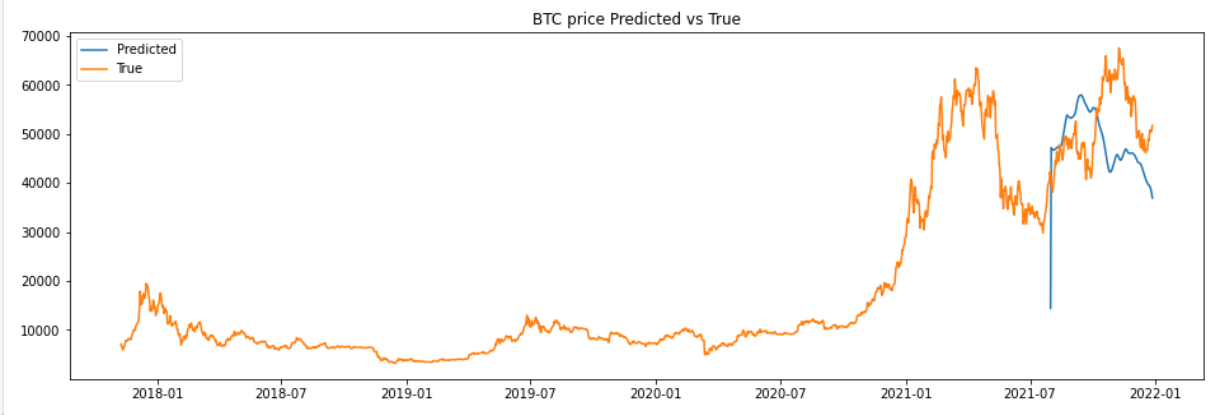
As we can see the program stopped at the 34th epoch because the val_loss reached a minimum value.



Now we have to predict the future bitcoin price based on the 50 past days and plot it. In order to do that we need to define a new function, the dummy_invscaler, that correctly performs inverse scale on the data. More specifically, since the scaler was trained into 2 features, it needs two features to perform the inverse scaler. For that purpose, this function will create a dummy array and concatenate it to the y_pred/y_true. That dummy of ones will be dropped after performing the inverse_transform.

```python
def dummy_invscaler(y, n_features):
    '''
    Since the scaler was trained into 2 features, it needs two features to perform the inverse scaleer.
    For that purpose, this function will create a dummy array and concatenate it to the y_pred/y_true.
    That dummy of ones will be drop after performing the inverse_transform.
    INPUTS: array 'y', shape (X,)
    '''
    y = np.array(y).reshape(-1,1)
    if n_features>1:
        dummy = np.ones((len(y), n_features-1))
        y = np.concatenate((y, dummy), axis=1)
        y = sc.inverse_transform(y)
        y = y[:,0]
    else:
        y = sc.inverse_transform(y)
    return y
```

The last thing that we are going to do is predict the values of the last 500 days and compare it to the actual prices by plotting them both.



As we can see the predictions are not entirely accurate but for a little bit they seem to follow the trend. This makes sense because we only feed the Close prices to the neural network. The other models have access to more information.

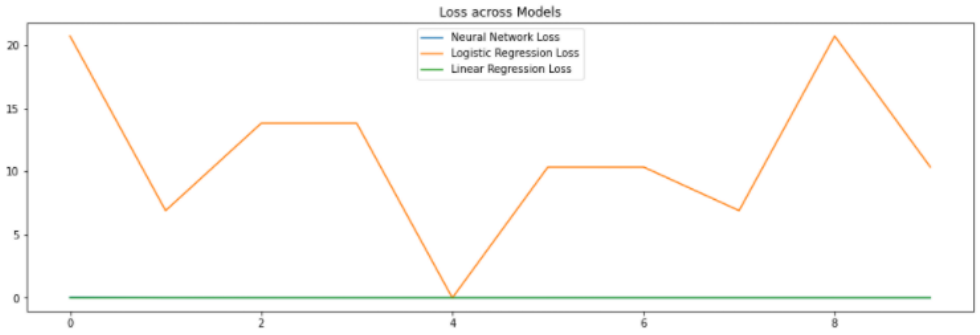
CONCLUSION

In the following table we are putting the approximate values that were generated during our test for the Score and Loss metrics:

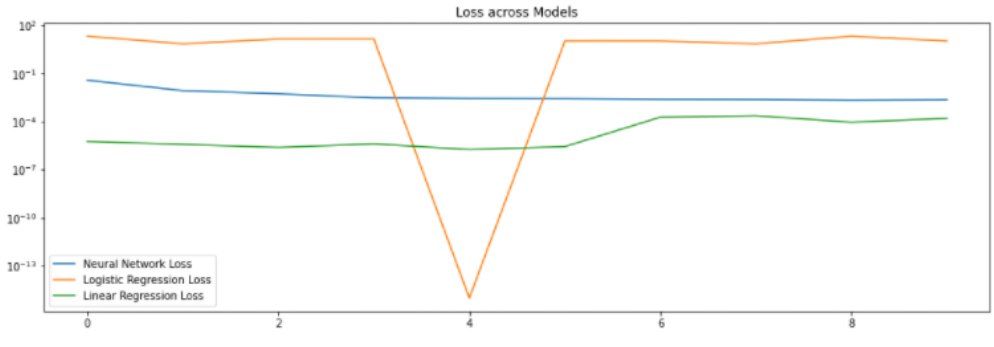
Metric/Model	Score	Loss
Linear Regression	0.97-0.99	0.0001- 0.00000226
Logistic Regression	0.5-0.8	6-20
Neural Network	-	0.15 – 0.02

We can also add the following graph:

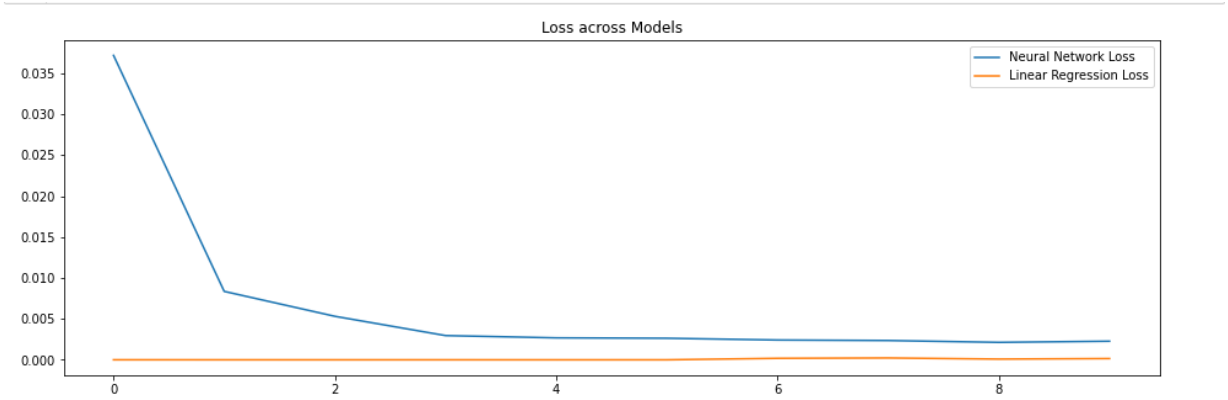
```
In [113]:
plt.figure(figsize=(16,5))
plt.plot(history['loss'],label='Neural Network Loss')
plt.plot(losses,label='Logistic Regression Loss')
plt.plot(losseslinear,label='Linear Regression Loss')
ax.set_xlabel('Recursion Number')
ax.set_ylabel('Loss')
plt.title('Loss across Models')
plt.legend()
plt.show()
```



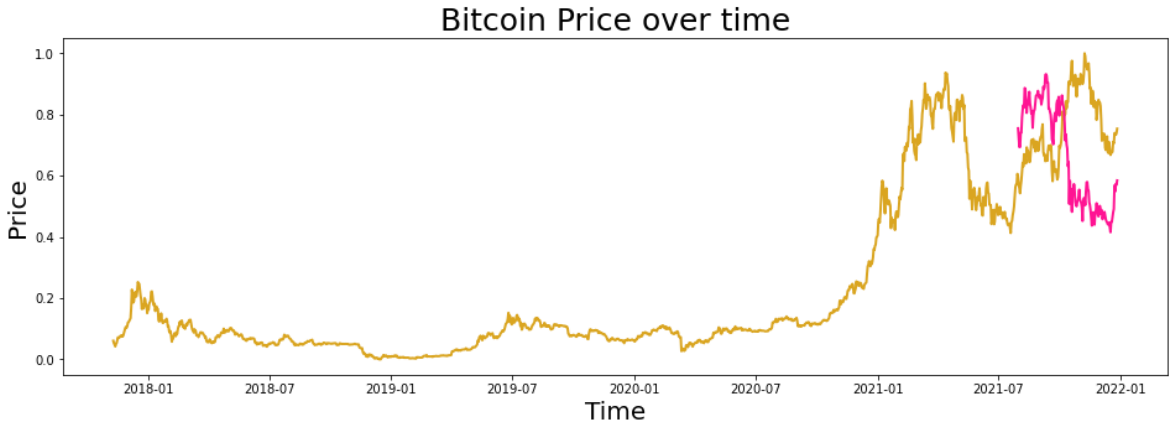

```
In [112]:
plt.figure(figsize=(16,5))
plt.plot(history['loss'],label='Neural Network Loss')
plt.plot(losses,label='Logistic Regression Loss')
plt.plot(losseslinear,label='Linear Regression Loss')
ax.set_xlabel('Recursion Number')
ax.set_ylabel('Loss')
plt.yscale('log')
plt.title('Loss across Models')
plt.legend()
plt.show()
```

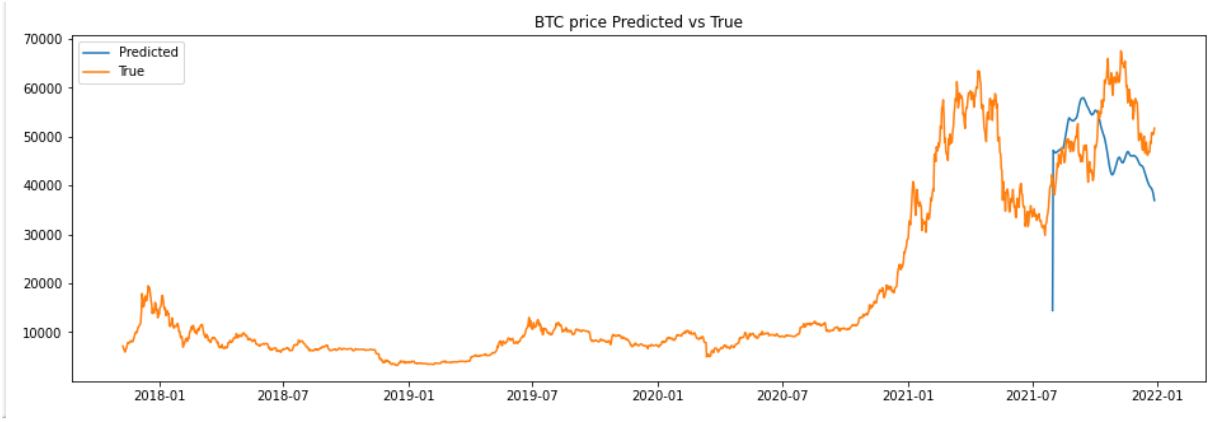


This graph compares the losses documented across models in a specific run and helps us understand which method has the lower loss. As we can see in the first diagram logistic regression has significantly bigger loss than the other two methods, however, because of the size difference we cannot accurately see the loss for the other two models. That is why we scaled it with log in the second plot in order to see which method is better. Therefore, we can see that although the logistic regression model at the 4th fold has lower loss than the others, it is generally above them. Also, the Linear regression model has slightly better results than the neural network.



According to these results the Logistic regression performed the worst, so we will discard it and we are going to look closer on the prediction performance of the other two models. These two models have the following plots:





In these plots we can see what our models predicted for the last 150 days, compared to what the actual price was. The first plot is the one created with Linear regression while the second one was created by our Neural network. We can reasonably infer that the prices predicted by the Linear regression model were closer to reality and that is why we are going to choose it as the best model out of the 3.