ARTIFICIAL INTELLIGENCE AND ANOMALY DETECTION

Time Series Anomaly Detection With LSTM Autoencoders- an Unsupervised ML Approach

How to set-up an anomaly detection model

Sarit Maitra

Sep 16, 2020 · 7 min read ★

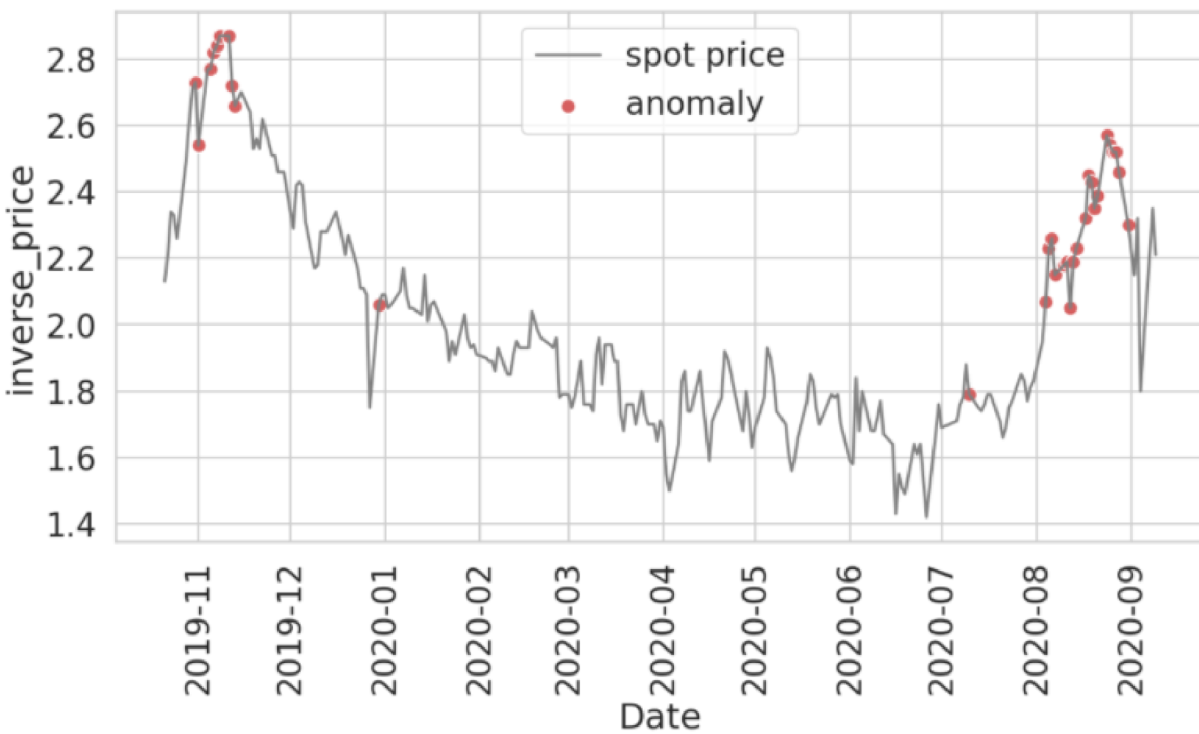


Image by Author

Anomaly here to detect that, actual results differ from predicted results in price prediction. As we are aware that, real-life data is streaming, time-series data etc., where anomalies give significant information in critical situations. In the detection of anomalies, we are interested in discovering abnormal, unusual or unexpected records and in the time series context, an anomaly can be detected within the scope of a single record or as a subsequence/pattern.

Estimating the historical data, time-series based predictive model helps us in predicting future price by estimating them with the current data. Once we have the prediction we can use that data to detect anomalies on comparing them with actuals.

Let's implement it and look at its pros and cons. Hence, our objective here is to develop an anomaly detection model for Time Series data. We will use neural-network architecture for this use case.

Let us load Henry Hub Spot Price data from EIA. We have to remember that, the order of data here is important and should be chronological as we are going to forecast the next point.

```
import os
print(os.listdir("../input"))
import warnings
warnings.filterwarnings('ignore')

print("....Data loading...."); print()
print('\033[4mHenry Hub Natural Gas Spot Price, Daily (Dollars per
Million Btu)\033[0m')
def retrieve_time_series(api, series_ID):
  series_search = api.data_by_series(series=series_ID)
  spot_price = DataFrame(series_search)
  return spot_price

def main():
  try:
    api_key = "....API KEY..."
    api = eia.API(api_key)
    series_ID = 'xxxxxx'
    spot_price = retrieve_time_series(api, series_ID)
    print(type(spot_price))
    return spot_price;
  except Exception as e:
    print("error", e)
    return DataFrame(columns=None)
spot_price = main()
spot_price = spot_price.rename({'Henry Hub Natural Gas Spot Price,
Daily (Dollars per Million Btu)': 'price'}, axis = 'columns')
spot_price = spot_price.reset_index()
spot_price['index'] = pd.to_datetime(spot_price['index'].str[:-3],
format='%Y %m%d')
```

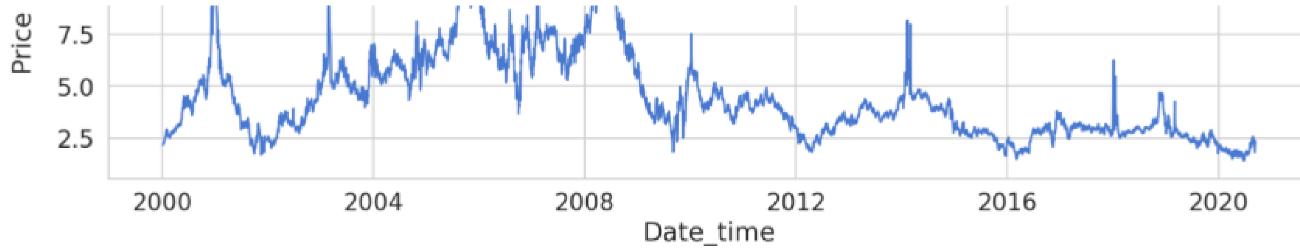
```
spot_price['Date']= pd.to_datetime(spot_price['index'])
spot_price.set_index('Date', inplace=True)
spot_price = spot_price.loc['2000-01-01':,['price']]
spot_price = spot_price.astype(float)
print(spot_price)
```

```
....Data loading....

Henry Hub Natural Gas Spot Price, Daily (Dollars per Million Btu)
<class 'pandas.core.frame.DataFrame'>
            price
Date
2000-01-04   2.16
2000-01-05   2.17
2000-01-06   2.18
2000-01-07   2.19
2000-01-10   2.20
...           ...
2020-09-02   2.15
2020-09-03   2.32
2020-09-04   1.80
2020-09-08   2.35
2020-09-09   2.21

[5217 rows x 1 columns]
```

Raw data visualization

```
print('Historical Spot price visualization:')
plt.figure(figsize = (15,5))
plt.plot(spot_price)
plt.title('Henry Hub Spot Price (Daily frequency)')
plt.xlabel ('Date_time')
plt.ylabel ('Price ($/Mbtu)')
plt.show()
```

Historical Spot price visualization:




```
print('Missing values:', spot_price.isnull().sum())
# checking missing values
spot_price = spot_price.dropna()
# dropping missing valies
print('....Dropped Missing value row....')
print('Rechecking Missing values:', spot_price.isnull().sum())
# checking missing values
```

```
Missing values: price    1
dtype: int64
....Dropped Missing value row....
Rechecking Missing values: price    0
dtype: int64
```

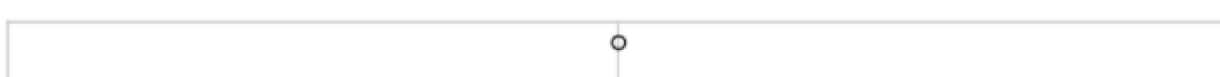
```
1   spot_price.describe().transpose()
```

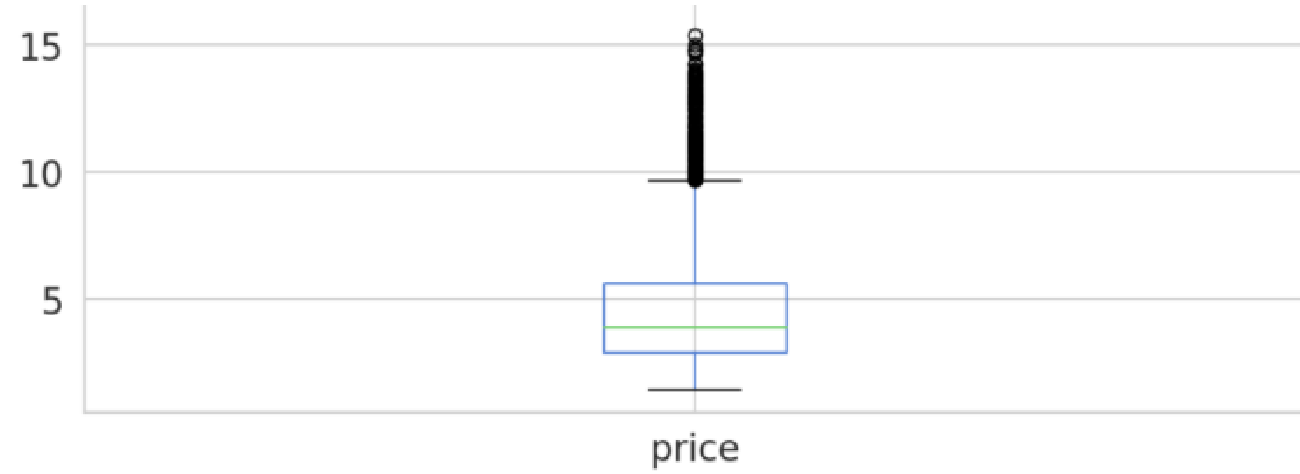
	count	mean	std	min	25%	50%	75%	max
price	5216.0	4.45351	2.205964	1.42	2.87	3.86	5.59	18.48

The common characteristic of different types of market manipulation is that, the unexpected pattern or behavior in data.

```
# Generate Boxplot
print('Box plot visualization:')
spot_price.plot(kind='box', figsize = (10,4))
plt.show()
```

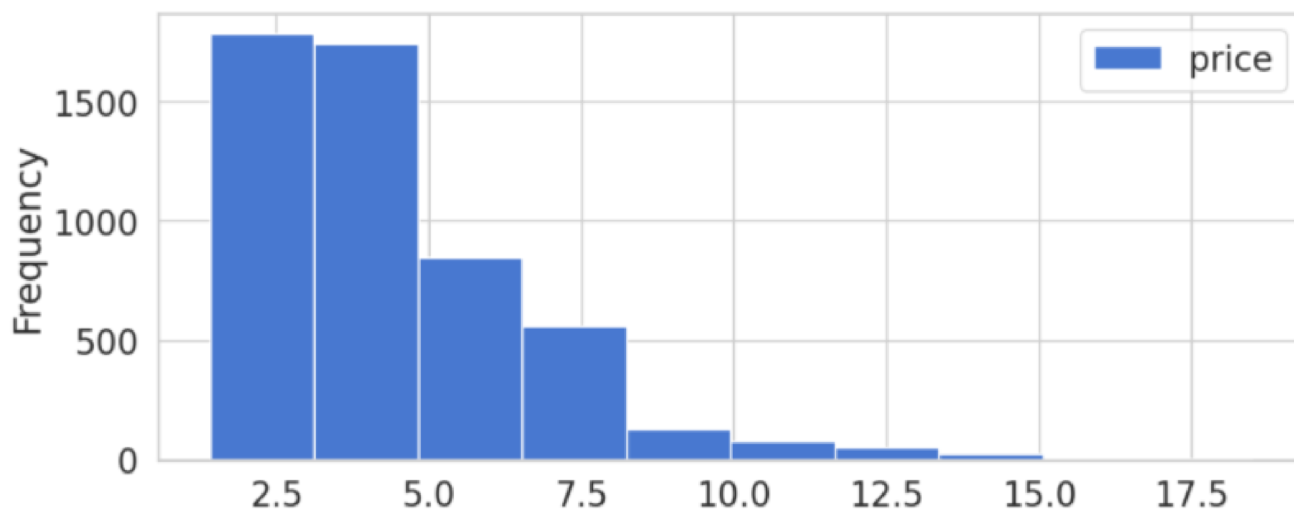
```
Box plot visualization:
```



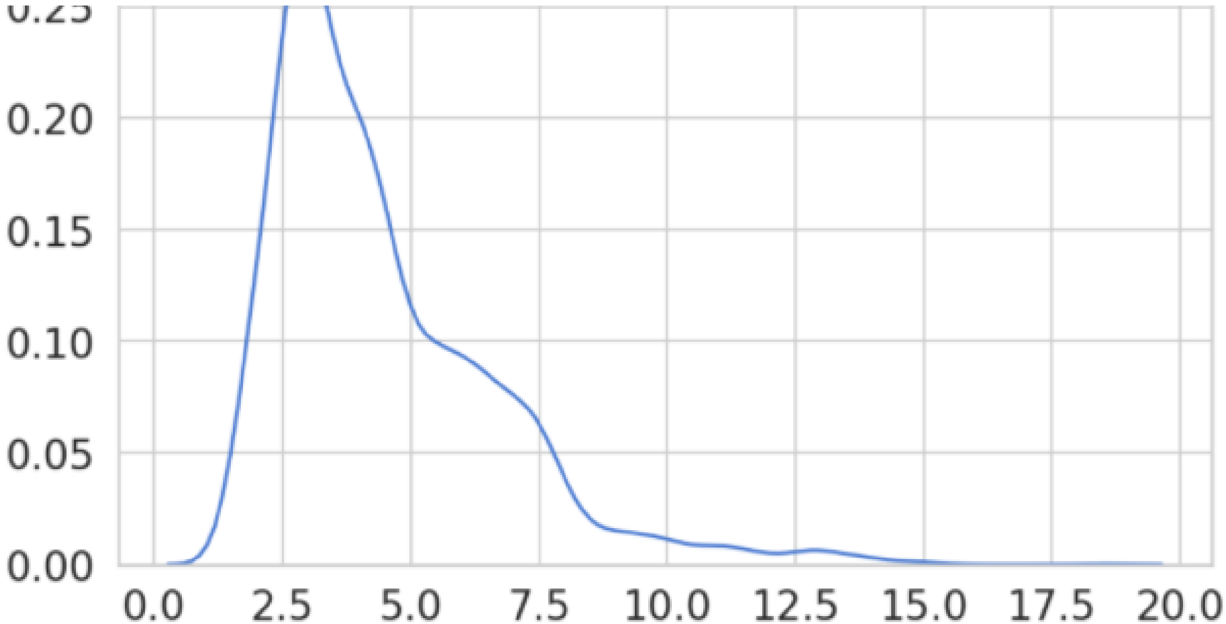



```
# Generate Histogram
print('Histogram visualization:')
spot_price.plot(kind='hist', figsize = (10,4) )
plt.show()
```

```
Histogram visualization:
```




```
fig, ax1 = plt.subplots(ncols=1, figsize = (8,5))
ax1.set_title('Price data before scaling')
sns.kdeplot(spot_price['price'], ax=ax1)
plt.show()
```



Detecting anomalous subsequence

Here, the goal is identifying an anomalous subsequence within a given long time series (sequence).

> Anomaly detection is based on the fundamental concept of modeling what is normal in order to discover what is not....Dunning & Friedman

Pre-processing

We'll use 95% of the data and train our model on it:

```
train_size = int(len(spot_price) * 0.95)
test_size = len(spot_price) - train_size
train, test = spot_price.iloc[0:train_size], spot_price.iloc[train_size:len(spot_price)]
print('Train shape:',train.shape)
print('Test shape:', test.shape)
```

```
Train shape: (4955, 1)
Test shape: (261, 1)
```

Next, we'll re-scale the data using the training data and apply the same transformation to the test data. I have used Robust scaler as shown below:

```
# data standardization
robust = RobustScaler(quantile_range=(25, 75)).fit(train[['price']])
train['price'] = robust.transform(train[['price']])
test['price'] = robust.transform(test[['price']])
```

Finally, we'll split the data into sub-sequences with the help of a helper function.

```
# helper function
def create_dataset(X, y, time_steps=1):
  a, b = [], []
  for i in range(len(X) - time_steps):
     v = X.iloc[i:(i + time_steps)].values
     a.append(v)
     b.append(y.iloc[i + time_steps])
  return np.array(a), np.array(b)

# We'll create sequences with 30 days of historical data

n_steps = 30

# reshape to 3D [n_samples, n_steps, n_features]

X_train, y_train = create_dataset(train[['price']], train['price'],
n_steps)
X_test, y_test = create_dataset(test[['price']], test['price'],
n_steps)
print('X_train shape:', X_train.shape)
print('X_test shape:', X_test.shape)
```

```
X_train shape: (4925, 30, 1)
X_test shape: (231, 30, 1)
```

LSTM Autoencoder in Keras

The sequence autoencoder is similar to sequence to sequence learning. It employs a recurrent network as an encoder to read in an input sequence into a hidden representation. Then, the representation is fed to a decoder recurrent network to reconstruct the input sequence itself.

Here, our Autoencoder should take a sequence as input and outputs a sequence of the same shape. We have a total of 5219 data points in the sequence and our goal is to find anomalies. We are trying to find out when data points are abnormal.

> If we can predict a data point at time 't' based on the historical data until 't-1', then we have a way of looking at an expected value compared to an actual value to see if we are within the expected range of values for time 't'.

We can compare *y_pred* with the actual value (*y_test*). The difference between *y_pred* and *y_test* gives the error, and when we get the errors of all the points in the sequence, we end up with a distribution of just errors. To accomplish this, we will use a sequential model using Keras. The model consists of a LSTM layer and a dense layer. The LSTM layer takes as input the time series data and learns how to learn the values with respect to time. The next layer is the dense layer (fully connected layer). The dense layer takes as input the output from the LSTM layer, and transforms it into a fully connected manner. Then, we apply a sigmoid activation on the dense layer so that the final output is between 0 and 1.

We also use the 'adam' optimizer and the 'mean squared error' as the loss function.

Issue with Sequences

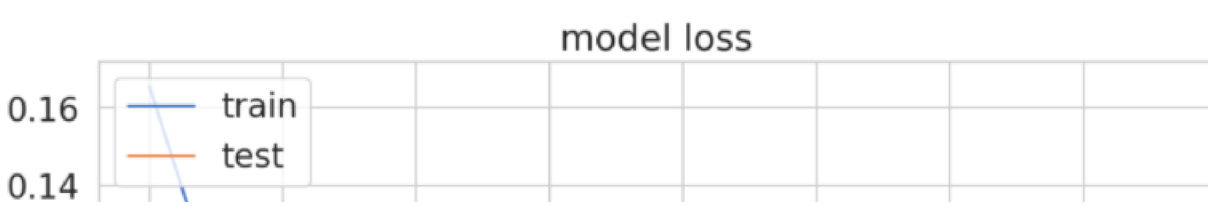
- ML algorithms, and neural networks are designed to work with fixed length inputs.
- Temporal ordering of the observations can make it challenging to extract features suitable for use as input to supervised learning models.

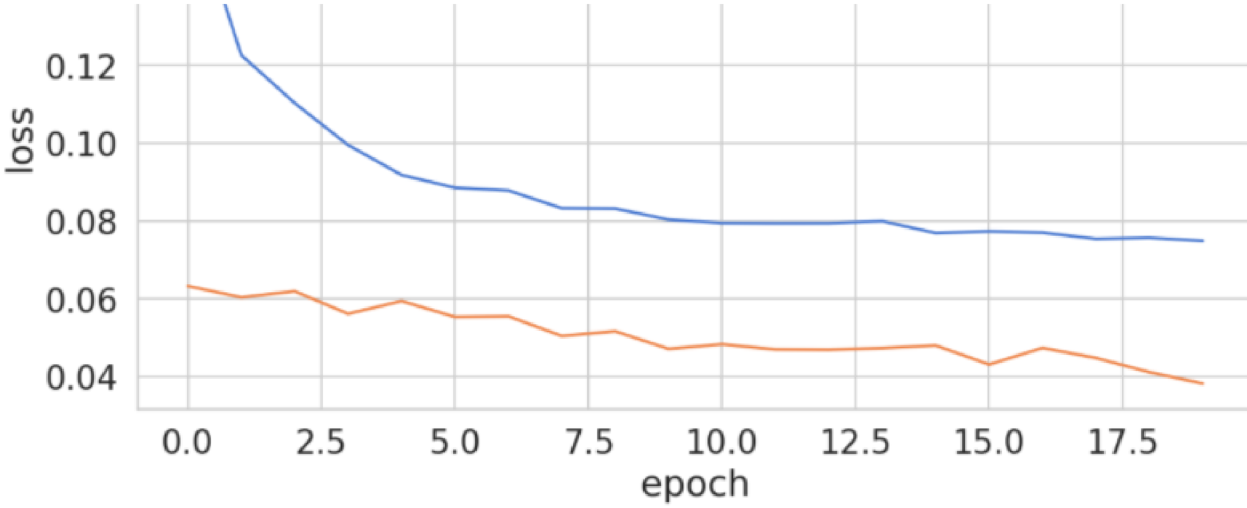
```
units = 64; dropout = 0.20; optimizer = 'adam'; loss = 'mae';
epochs = 20;


model = keras.Sequential()
model.add(keras.layers.LSTM(units=units, input_shape =
(X_train.shape[1], X_train.shape[2])))
model.add(keras.layers.Dropout(rate=dropout))
model.add(keras.layers.RepeatVector(n=X_train.shape[1]))
model.add(keras.layers.LSTM(units=units, return_sequences=True))
model.add(keras.layers.Dropout(rate=dropout))
```

```
model.add(keras.layers.Dropout(rate=dropout))
model.add(keras.layers.TimeDistributed(keras.layers.Dense(units=
X_train.shape[2])))
model.compile(loss= loss, optimizer=optimizer)
history = model.fit(X_train, y_train, epochs=epochs, batch_size=32,
validation_split=0.1, shuffle=False)
```

```
Epoch 7/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0879 - val_loss: 0.0555
Epoch 8/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0833 - val_loss: 0.0504
Epoch 9/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0832 - val_loss: 0.0516
Epoch 10/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0804 - val_loss: 0.0471
Epoch 11/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0794 - val_loss: 0.0483
Epoch 12/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0794 - val_loss: 0.0469
Epoch 13/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0794 - val_loss: 0.0469
Epoch 14/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0799 - val_loss: 0.0473
Epoch 15/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0769 - val_loss: 0.0480
Epoch 16/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0773 - val_loss: 0.0431
Epoch 17/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0770 - val_loss: 0.0473
Epoch 18/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0754 - val_loss: 0.0447
Epoch 19/20
139/139 [==============================] - 1s 7ms/step - loss: 0.0756 - val_loss: 0.0411
Epoch 20/20
139/139 [==============================] - 1s 6ms/step - loss: 0.0749 - val_loss: 0.0382
```

```
# history for loss
plt.figure(figsize = (10,5))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```





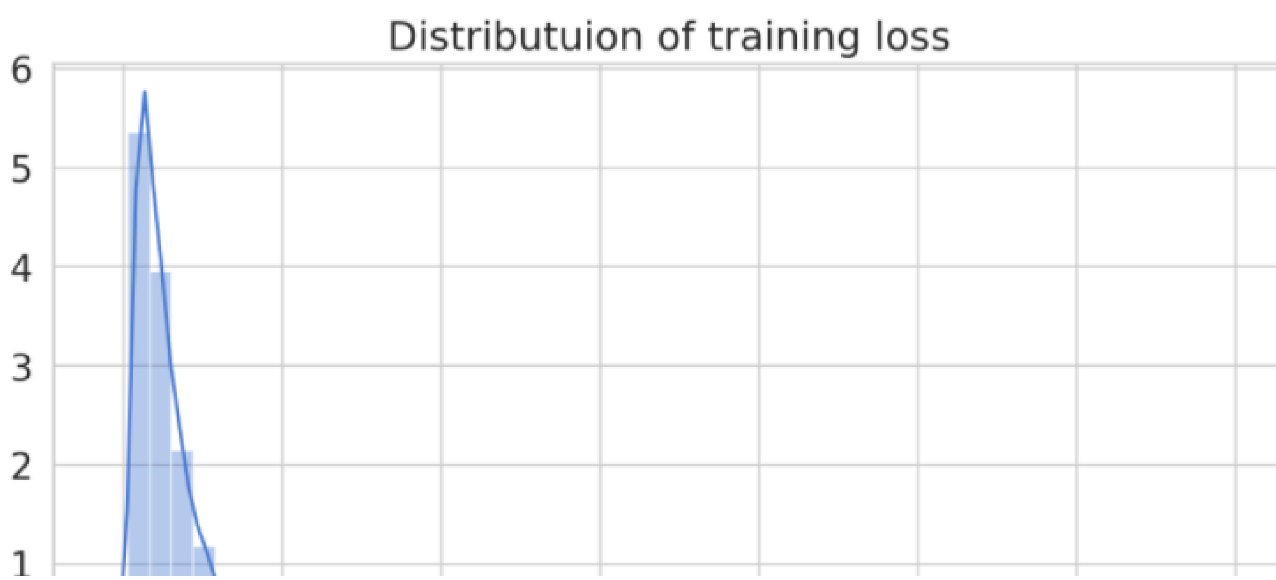
Evaluation

Once the model is trained, we can predict using test data set and compute the error (mae). Let's start with calculating the Mean Absolute Error (MAE) on the training data.

MAE on train data:

```
train_pred = model.predict(X_train)
train_loss = (np.mean(np.abs(train_pred - X_train), axis=1))
avg_loss = train_loss.mean()
print('Training loss:', avg_loss); print()

plt.figure(figsize = (10,5))
sns.distplot(train_loss, bins=50, kde=True);
plt.title('Distributuion of training loss')
plt.show()
```

```
Training loss: 0.17053443755839884
```



Accuracy metrics on test data:

```
# MAE on the test data:
y_pred = model.predict(X_test)
print('Predict shape:', y_pred.shape); print();
mae = np.mean(np.abs(y_pred - X_test), axis=1)
# reshaping prediction
pred = y_pred.reshape((y_pred.shape[0] * y_pred.shape[1]),
y_pred.shape[2])
print('Prediction:', pred.shape); print();
print('Test data shape:', X_test.shape); print();
# reshaping test data
X_test = X_test.reshape((X_test.shape[0] * X_test.shape[1]),
X_test.shape[2])
print('Test data:', X_test.shape); print();
# error computation
errors = X_test - pred
print('Error:', errors.shape); print();
# rmse on test data
RMSE = math.sqrt(mean_squared_error(X_test, pred_reshape))
print('Test RMSE: %.3f' % RMSE);
```

```
Predict shape: (231, 30, 1)

Prediction: (6930, 1)

Test data shape: (231, 30, 1)

Test data: (6930, 1)

Errors: (6930, 1)

Test RMSE: 0.099
```

RMSE is 0.099, which is low, and this is also evident from the low loss from the training phase after 20 epochs: loss: 0.0749— val_loss: 0.0382. Though this might be a good prediction where the error is low but the anomalous behavior in the actuals cant be identified using this.

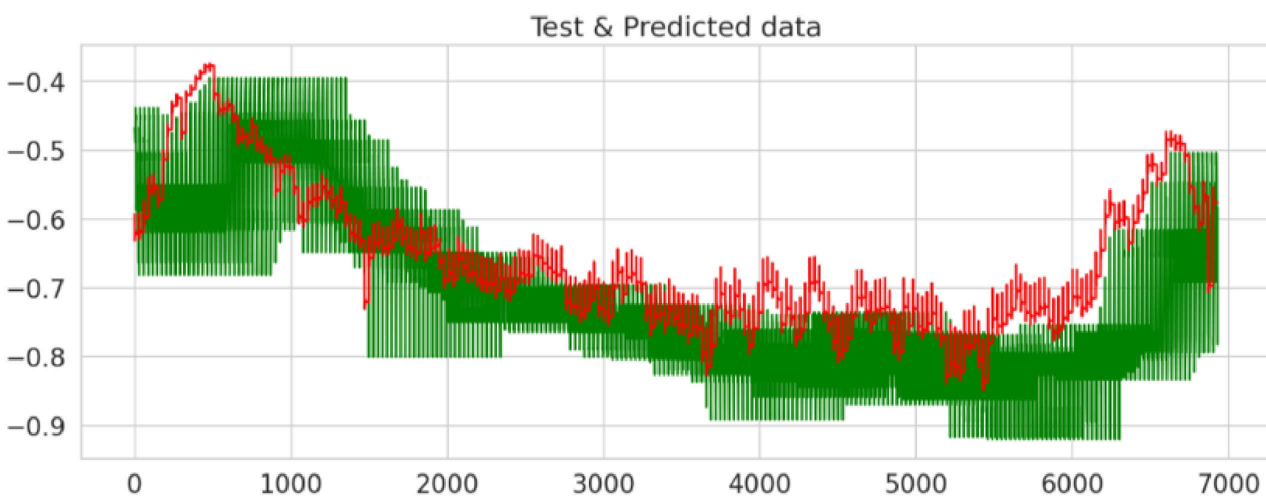
Threshold computation:

```
dist = np.linalg.norm(X_test - pred, axis=1);


scores = dist.copy();
print('Score:', scores.shape);
scores.sort();
cut_off = int(0.80 * len(scores));
print('Cutoff value:', cut_off);
threshold = scores[cut_off];
print('Threshold value:', threshold);
```

```
Score: (6930,)
Cutoff value: 5544
Threshold value: 0.1182294107865598
```

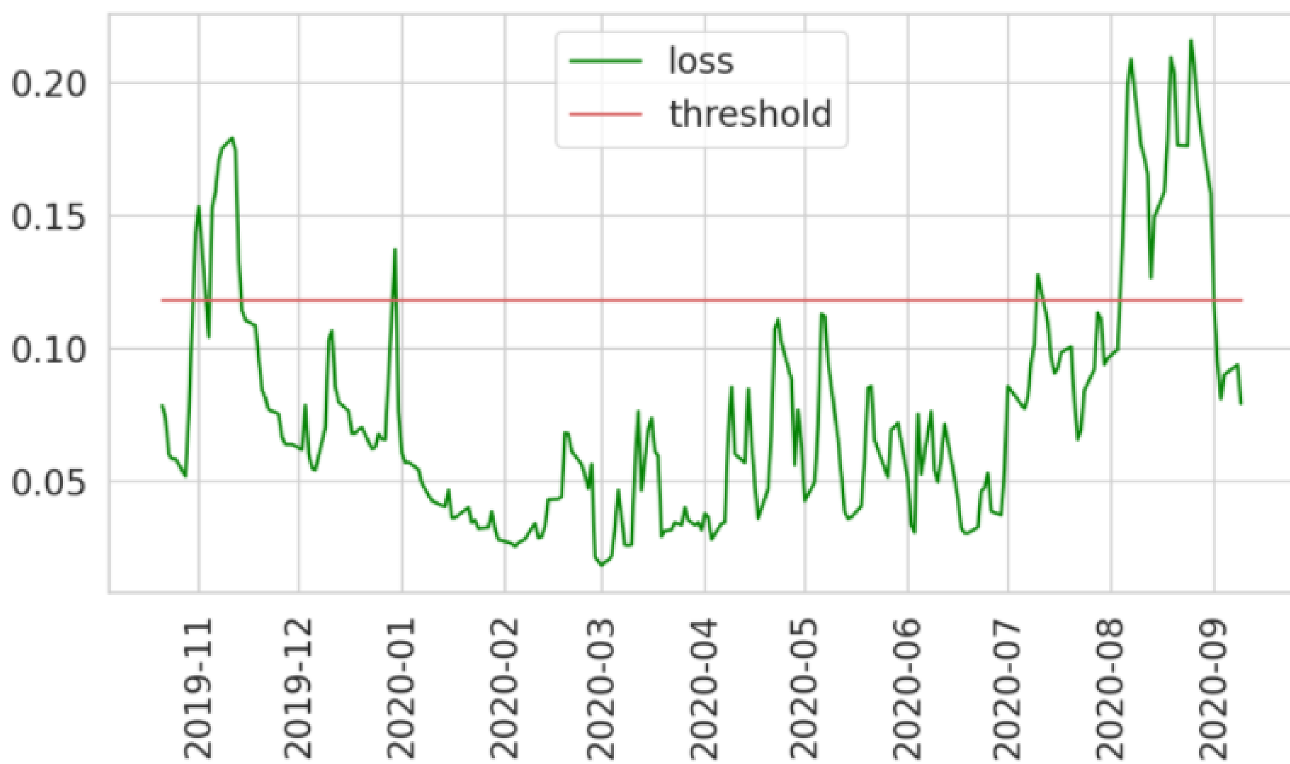
Objective is that, anomaly will be detected when the error is larger than selected threshold value.

```
plt.figure(figsize= (14,5))
plt.plot(X_test, color = 'green')
plt.plot(pred, color = 'red')
plt.title("Test & Predicted data")
plt.show()
```




```
score = DataFrame(index=test[n_steps:].index)
score['loss'] = mae
score['threshold'] = threshold
score['anomaly'] = score['loss'] > score['threshold']
```

```
score['anomaly'] = score['loss'] > score['threshold']
score['price'] = test[TIME_STEPS:].price

plt.figure(figsize = (10,5))
plt.plot(score.index, score['loss'], color = 'green', label='loss')
plt.plot(score.index, score['threshold'], color = 'r', label='threshold')
plt.xticks(rotation=90)
plt.legend();
```



Looks like we're thresholding extreme values quite well. Let's create a dataframe using only those:

Anomalies report format:

```
anomalies = score[score['anomaly'] == True]
x = DataFrame(anomalies.price)
x = DataFrame(robust.inverse_transform(x))
x.index = anomalies.index
x.rename(columns = {0: 'inverse_price'}, inplace = True)
anomalies = anomalies.join(x, how = 'left')
anomalies = anomalies.drop(columns=['price'], axis=1)
anomalies.tail(10)
```

Date	loss	threshold	anomaly	inverse_price
2020-08-18	0.178424	0.118229	True	2.45
2020-08-19	0.209752	0.118229	True	2.43
2020-08-20	0.203537	0.118229	True	2.35
2020-08-21	0.176636	0.118229	True	2.39
2020-08-24	0.176374	0.118229	True	2.57
2020-08-25	0.216156	0.118229	True	2.54
2020-08-26	0.206829	0.118229	True	2.52
2020-08-27	0.192255	0.118229	True	2.52
2020-08-28	0.182474	0.118229	True	2.46
2020-08-31	0.158133	0.118229	True	2.30

Inverse test data

```
test_inv = DataFrame(robust.inverse_transform(test[n_steps:]))
test_inv.index = test[n_steps:].index
test_inv.rename(columns = {0: 'price'}, inplace = True)
test_inv
```

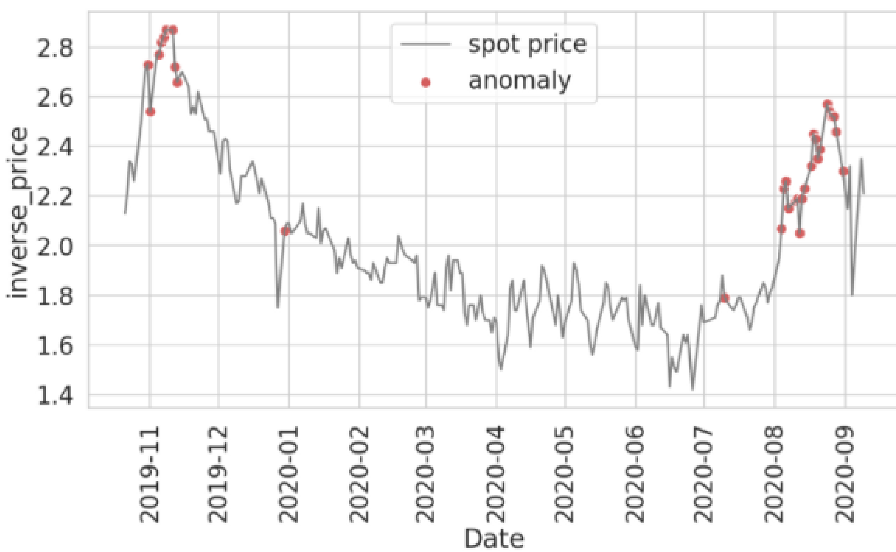
Date	price
2019-10-21	2.13
2019-10-22	2.21
2019-10-23	2.34
2019-10-24	2.33
2019-10-25	2.26
...	...
2020-09-02	2.15

2020-09-03	2.32
2020-09-04	1.80
2020-09-08	2.35
2020-09-09	2.21

231 rows × 1 columns

Finally, let's look at the anomalies found in the testing data:

```
plt.figure(figsize = (10,5))
plt.plot(test_inv.index, test_inv.price, color = 'gray', label='spot price');
sns.scatterplot(anomalies.index, anomalies['inverse_price'], color=sns.color_palette()[3], s=55, label='anomaly')
plt.xticks(rotation=90)
plt.legend(loc='upper center');
```



The red dots are the anomalies here and are covering most of the points with abrupt changes to the existing spot price. The threshold values can be changed as per the parameters we choose, especially the cutoff value. If we play around with some of the parameters we used, such as number of time steps, threshold cutoffs, epochs of the neural network, batch size, hidden layer etc., we can expect a different set of results.

With this we conclude a brief overview of finding anomalies in time series with respect to stock trading.

Conclusion

Though the stock market is highly efficient, it is impossible to prevent historical and long term anomalies. Investors may use anomalies to earn superior returns is a risk since the anomalies may or may not persist in the future. However, every report metric needs to be validated with parameters fine-tuned so that anomalies are detected when using prediction for detecting anomalies. Also for metrics with different distribution of data a different approach in identifying anomalies needs to be followed.

Connect me here.

Note: The programs described here are experimental and should be used with caution for any commercial purpose. All such use at your own risk….by Author.

Artificial Intelligence | Neural Network Algorithm | Anomaly Detection | Stock Market | Machine Learning