Audience: Curious machine learning newbies of all ages

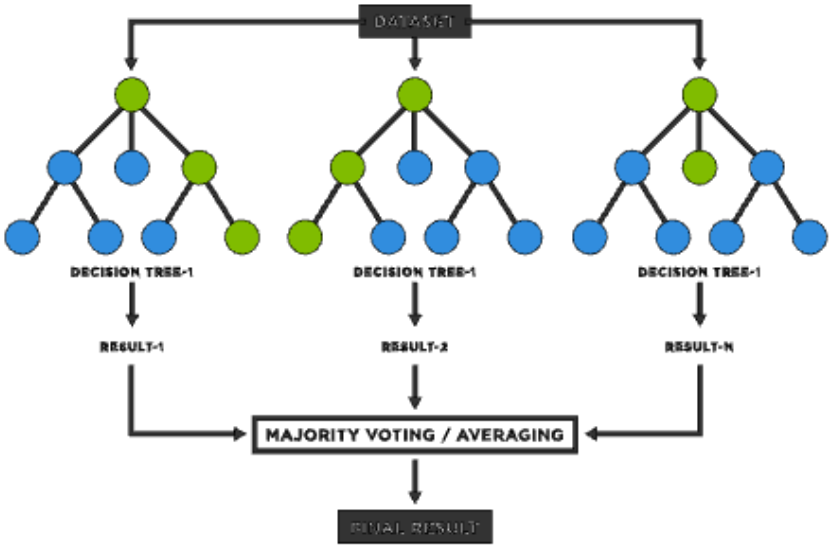
Duration: ~15min read

Which is the real money maker, Random Forest regressors or LSTM Networks?

In the field of machine learning, it is instrumental to be able to pick the right model or algorithm for the task at hand. In this article, we will be carrying out an in-depth comparison between two prominent models, the Random Forest Regressor and the LSTM model in the context of stock price predictions.

When deciding on a model to tackle this problem of equity prediction, we were inclined to implement either a Random Forest, which is the found in the field of machine learning, or an LSTM (Long short-term memory), which is a neural network found in the field of deep learning.

So, what exactly is a Random Forest Classifier? To try and illustrate the concept behind a Random Forest algorithm, imagine deciding to purchase a new car. You have some idea of the features you want in your car, but you know that your friends (who know you well) can give you some better ideas. The first friend advises you on a car model after enquiring about your positive and negative experiences on a previous car ride, for example. In this manner, your friend provides a suggestion based on some rules on what to recommend. Similarly, other friends ask you different questions and subsequently, provide you with different recommendations. In the end, you choose a car after taking an average of all the recommendations that were made to you. Thus, the random forest provides an easy to implement and flexible machine learning algorithm that can output a quite reliable result.

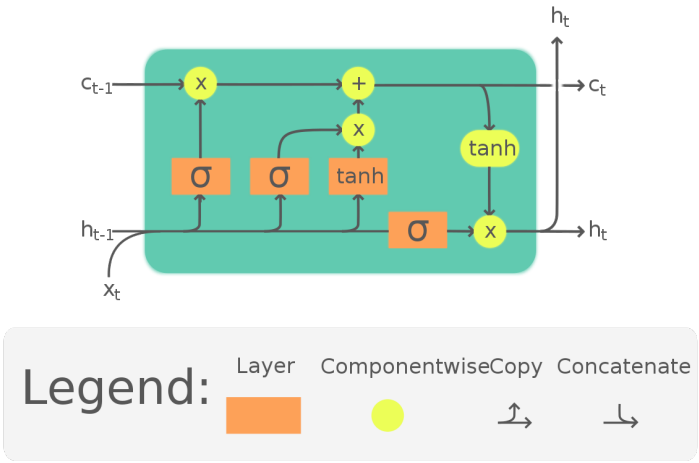


The use cases of a Random Forest can actually be found across a variety of fields: healthcare, finance, etc. For example, in the world of banking, a random forest can model the likelihood that a borrower will default on their monthly payment.

Turning our attention now to neural networks, LSTMs are essentially a type of recurrent neural network which are designed to easily decipher patterns located within long sequences of data. They are especially useful for numerical time series data such as stock price data (Open, High, Low, Close and technical indicator values). By taking both time and sequence into account, LSTMs set themselves apart from other neural networks. Consequently, the use cases of LSTMs tend to lie in time series forecasts, speech recognition, handwriting recognition, traffic forecast, etc.

To give you an analogy of how this algorithm works, try reciting the first 10 letters of the alphabet. Easy right? Now try reciting those same letters backwards. Hard isn't it. That is the intuition behind the LSTM model. Our brains are wired to remember the alphabet sequentially, hence we decipher which letter comes next based on which letter has already been said. This memory of the previous letter is stored in a component of a LSTM node called the hidden state, which is a store of historical information that is useful in predicting sequential data. However, what I have just described to you in this paragraph is the Recurrent Neural Network (RNN), not exactly a LSTM model.

The LSTM model is designed to solve a prevalent problem that a vanilla RNN suffers from, the vanishing gradient. What this essentially means is that as this hidden state stores more and more historical data, the weightage given to older data "vanishes" when predictions are being made. This results in a loss of valuable data on historical trends, compromising the accuracy of our predictions. To get around this issue, the LSTM architecture includes a component of "unchanging hidden state" which stores the full information of relatively older data, so that valuable trends from this is not lost. This enables the LSTM model to perform exceptionally well in terms of incorporating insights from older trends in data.

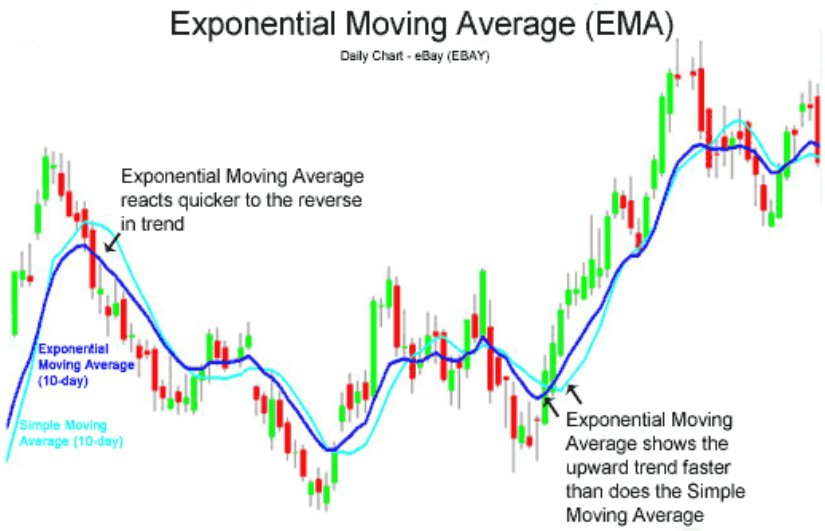


Generally, random forests tend to perform well and can be quickly put together, given their ease of implementation, flexibility as well as being computationally inexpensive. However, some neural networks can outperform random forests, but the trade-off is that they can take much

longer to develop and may require much more data. Thus, we opted to use a random forest to make predictions for our model.

To better generate and predict the stock price signal, three commonly-used technical indicators: exponential moving average, relative strength index, and volume price trend indicators were used and fed into our machine learning models for evaluation.

1. Exponential moving average (EMA)



Differing from a simple moving average indicator that seeks only to compute arithmetic mean of the stock price over a specified period, EMA is an alternative form of simple moving average indicator that assigns greater weight and hence significance to more recent stock price. This is because stock prices are proven to be more responsive and sensitive to recent news and changes instead of historical ones dated a while back. Therefore, EMA is believed to be a more effective indicator and is more prevalent in use for generating stock price signals.

2. Relative Strength Index (RSI)

RSI is a momentum indicator that takes into account both the price and volume changes. It helps to forecast future price trends through tracking whether a stock is oversold or overbought. Given how the stock markets are driven by the force of demand and supply, RSI is often considered an effective price trend indicator. As shown in the graph above, it is displayed as an oscillator that fluctuates between the upper and lower bound, typically set at the 70 and 30 mark. Any fluctuation beyond the upper bound or below the lower bound indicates an overbought or oversold position respectively

3. Volume Price Trend (VPT)

Similarly tracking demand and supply of a stock, VPT differs from RSI by accounting for both the price and volume trends in its computation. VPT defines volume changes as the fundamental force behind price changes and therefore, cumulative volume is provided to help investors better understand the money flow behind each stock

Preparing our data for prediction

```
import yfinance as yf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pandas_ta as ta
print('Packages successfully imported')
```

Fig 3.3 Importing required packages

These are the packages we will be using for our comparative analysis between RF Regressors and LSTM models. We will be using the yfinance API for our dataset, which includes standard stock price metrics like open price, close price, volume and adjusted close. To incorporate our technical indicators, we will be using the pandas_ta library, which comes with in-built technical indicator functions to make the computational task easier.

```
dow_reduced = ['AAPL', 'BA', 'CSCO', 'GS', 'HD', 'INTC', 'JNJ', 'KO',
'DIS', 'V'] # Stocks to analyse
 stocks_df = yf.download(tickers = dow_reduced, period = '10y', interval =
                  '1d', group_by='ticker', threads=True)
stocks_df.head()
```

Fig 3.4 Pulling required data

```
                # Including a columns for EMA, RSI and PVT
for ticker in dow_reduced:
 stocks_df[ticker, 'EMA'] = ta.ema(close=stocks_df[ticker, 'Adj Close'],
length=30) # Taking a 30-day moving average
 stocks_df[ticker, 'RSI'] = ta.rsi(close=stocks_df[ticker, 'Adj Close']) #
             Calculating the Relative strength for each stock
 stocks_df[ticker, 'PVT'] = ta.pvt(close = stocks_df[ticker, 'Adj Close'],
volume = stocks_df[ticker, 'Volume']) # Calculating the price-volume trend
                               for each stock
```

Fig 3.5 Calculating the required indicators

After pulling the required data (10years of data on 10 stocks), we calculate the technical indicator values for each stock. The data for the first stock being analysed (Apple) is this. (This dataframe is obtained after some projection steps to get only the columns we are interested in).

	AAPL			
	Adj Close	EMA	RSI	PVT
Date				
2012-04-09	19.456692	17.789045	74.209994	1.533994e+10
2012-04-10	19.218462	17.881266	68.207627	1.425055e+10
2012-04-11	19.149967	17.963118	66.541160	1.400228e+10
2012-04-12	19.045065	18.032921	63.963600	1.366575e+10
2012-04-13	18.508673	18.063614	52.718493	1.124461e+10

Fig 3.6 Apple's indicator and adjusted close data

Next, in order for the prediction step to work, we include a new column called the Next Adjusted Close, which is essentially the adjusted close price for the next row of data (next day). The next day's adjusted price becomes the target variable for our models to predict.

```
# Performing some data processing
def process_data(df, ticker_list):
   for ticker in ticker_list:
     df[ticker, 'Next Adj Close'] = df[ticker, 'Adj Close']
     df[ticker, 'Next Adj Close'] = df[ticker, 'Next Adj Close'].shift(-1)
     # Drop columns that are NaN
     df = df.dropna()
   return df
```

Fig 3.7 Code to produce this new target column

With this function, our final transformed data should be looking like this,

	AAPL				
	Adj Close	EMA	RSI	PVT	Next Adj Close
Date					
2022-02-04	172.389999	169.744912	54.228849	5.564257e+09	171.660004
2022-02-07	171.660004	169.868466	52.924970	5.531544e+09	174.830002
2022-02-08	174.830002	170.188565	57.683178	5.669729e+09	176.279999
2022-02-09	176.279999	170.581561	59.690217	5.728852e+09	172.119995
2022-02-10	172.119995	170.680815	52.061203	5.514419e+09	168.639999

Fig 3.8 Final transformed data for one stock

Random Forest Example

To start building our Random Forest regression model, we import the RandomForestRegressor class from the ensemble module of the Scikit-Learn library. We also import the train_test_split function from sklearn.model_selection to split our dataset into train and test subsets. With data preprocessing completed, building the first iteration of our model is simple and only requires a few lines of code.

Firstly, we initialize a RandomForestRegressor object and assign the argument of n_estimators to an arbitrary value of 1000, which represents the number of trees in the forest. Next, we train our RandomForestRegressor with the training data. With that, the first iteration of our model is completed and we can now test our model using the test data by using the RandomForestRegressor predict method.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

# Split data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(df, target,
test_size=0.3)
```

```
# First run of random forest regressor
rf = RandomForestRegressor(n_estimators=1000)
rf.fit(x_train, y_train)
y_pred = rf.predict(x_test)
```

We can also use the get_params method of RandomForestRegressor to have a closer inspection of the various hyperparameters used for our estimator. In essence, hyperparameters are factors controlling the learning process and cannot be estimated by the model from the given data.

```
print(rf.get_params())
```

A rough sensing of the quality of our model's prediction can be obtained via some of the functions available in the sklearn.metrics module. In this case, we have chosen to use the mean squared error (MSE) metric. With Intel's stock data, our Random Forest model appears to have performed relatively well with a MSE of approximately 0.645.

```
from sklearn import metrics
print(f"MSE of initial RF model for
{ticker}:{metrics.mean_squared_error(y_test, y_pred)}")
```

At this stage, we should wonder if we are able to improve the performance of our model and generate better predictions. Indeed, this may be possible through hyperparameter tuning, which allows us to find an optimal set of hyperparameters to maximize model performance and prevent the problem of overfitting. There are numerous methods available to perform the tuning of hyperparameters. Well-known ones include Randomized Search, Grid Search and Bayesian Optimization.

In our example, we have decided on the Randomized Search approach using RandomizedSearchCV from sklearn.model_selection. With Random Search, a search space of bounded hyperparameter values is defined as a grid, and then points are randomly sampled in this grid, each corresponding to different combinations of hyperparameters. A k-fold cross-validation is then performed to evaluate the performance of the model under different hyperparameters to yield the best set of hyperparameters.

Using the get_params method, we see that the RandomForestRegressor is defined by many hyperparameters, such as n_estimators, bootstrap, max_leaf_nodes, and min_impurity_decrease. It will be time-consuming to attempt to optimize every hyperparameter. Hence, in view of striking a balance between model performance and run-time efficiency, we have gathered a smaller list of six hyperparameters to tune by learning from the experience of

similar projects available online. They are n_estimators, max_features, max_depth, min_samples_split, min_samples_leaf, and bootstrap. We define the search grid over ranges of feasible values in a dictionary as follows:

```
from sklearn.model_selection import RandomizedSearchCV

# n_estimators: Number of trees in random forest
# max_features: Number of features to consider at every split
# max_depth: Maximum number of levels in tree
# min_samples_split: Minimum number of samples required to split a node
# min_samples_leaf: Minimum number of samples required at each leaf node
# bootstrap: Method of selecting samples for training each tree

# Create the search grid
 search_grid = {'n_estimators': [int(x) for x in np.linspace(start = 100,
                        stop = 2000, num = 10)],
                'max_features': ['auto', 'sqrt'],
                'max_depth': [int(x) for x in np.linspace(10, 100, num =
                             10)] + [None],
                'min_samples_split': [2, 5, 10],
                'min_samples_leaf': [1, 2, 4],
                'bootstrap': [True, False]}
```

A base model rf_tuned is first created using RandomizedSearchCV. One of the crucial arguments that is passed is n_iter, which determines the number of parameter settings to be sampled. According to scikit-learn's documentation, this is the factor that trades off runtime with the quality of the machine learning solution. Another key argument that is passed is cv, which indicates the type of cross-validation to be used. In this project, we will use the default of 10 candidate samples and 5-fold cross validation. Furthermore, we have set the n_jobs argument to -1 so that we use all processors and speed up processing.

Following that, the base model is fitted with the training dataset to obtain the optimal hyperparameters, which can then be viewed by calling the best_params_ attribute. We should compare this set of hyperparameters to the ones used to create the initial Random Forest Regressor to observe how it has been modified to create an apparently more optimal solution.

```
# Random sampling parameters, searching across 30 different parameter
settings,
# using 3-fold cross validation, and use all cores for maximum processing
speed
```

```
rf_tuned = RandomizedSearchCV(
    estimator = RandomForestRegressor(), param_distributions=random_grid,
n_iter=10, cv=None,
    verbose=2, random_state=42, n_jobs=-1)

# Fit new model and predict
rf_tuned.fit(x_train, y_train)
print(rf_tuned.best_params_)
```

After tuning, the refitted estimator can be called upon via the best_estimator_ attribute. We can proceed to use the predict function on this RandomizedSearchCV instance to get a new set of predictions (that should be of a higher quality than the previous one produced). To test this out, we once again use MSE as a way to measure the quality of the predictions. As it turns out, tuning the hyperparameters of our Random Forest Regressor helped to yield better predictions, based on the criteria of MSE. The tuned model recorded a MSE of approximately 0.595, using the same set of Intel's stock data.

```
new_y_pred = rf_tuned.best_estimator_.predict(x_test)
print(f"MSE of hyperparameter-tuned RF model for {ticker}:
{metrics.mean_squared_error(y_test, new_y_pred)}")
```

```
data = {'Adj Close': y_test, 'Predicted Adj Close': y_pred, 'Predicted Adj
Close (tuned)': new_y_pred}
results = pd.DataFrame(data = data)

results["Diff"] = results["Predicted Adj Close"] - results["Adj Close"]
results["Diff (tuned)"] = results["Predicted Adj Close (tuned)"] -
results["Adj Close"]
results
```

It is to be noted, however, that MSE is not the main evaluation method that will be used in this project. MSE is used in the above as a convenient proxy to determine the quality of the Random Forest Regression models generated earlier. Subsequently, we will introduce our main methods of evaluation, namely, accuracy score and ROC curve.

To get a better sense of how the predicted prices generated by both Random Forest Regression models compare to the actual adjusted close prices, the various data can be put together in a dataframe. We also create 2 new columns, Diff and Diff (tuned). The former refers to the difference between the predicted adjusted closing price and the actual adjusted closing price of

INTC using our initial model, whereas the latter refers to the corresponding difference using the hyperparameter-tuned model.

Date	Adj Close	Predicted Adj Close	Predicted Adj Close (tuned)	Diff	Diff (tuned)
2016-12-20	32.304958	32.397875	32.371874	0.092916	0.066916
2013-01-31	16.345083	16.256008	16.280730	-0.089075	-0.064354
2019-06-21	44.411400	44.180624	44.079562	-0.230775	-0.331838
2016-06-10	27.682030	27.513807	27.496531	-0.168223	-0.185499
2018-04-25	47.978477	46.586390	46.665135	-1.392087	-1.313342
...	...	...	...	...	...
2018-11-30	46.165314	45.392735	45.175001	-0.772578	-0.990313
2015-01-28	28.085638	27.608322	28.457819	-0.477316	0.372181
2012-08-31	18.496059	18.821161	18.876043	0.325101	0.379983
2022-02-11	47.580002	47.197159	47.670411	-0.382842	0.090410
2020-12-24	45.533482	46.758753	45.959663	1.225271	0.426181

750 rows × 5 columns

LSTM example

Now its time to see what kind of results are produced by the LSTM model. Unlike the Random Forest example, the splitting of the data into train and test sets is slightly different. Since LSTM models are designed to handle time series data well, we do not split the data into train and test sets randomly, instead we design our own function to split them sequentially as follows.

Next, if you have had prior experience with making forecasts on time series data, it may seem intuitive to import the TimeseriesGenerator from the keras.preprocessing library to batch up the input and target data. But however, that is only done for univariate time series analysis, where the model is making forecasts purely based on past price trends. Since we are using a host of other features in the form of technical indicators, our forecasting is multivariate and hence the batching is done a little differently. We combine the train-test splitting and batching of the sequential data into a single function as shown below.

```
# Creating validation and training data
train_df = aapl_df[:2114]
validation_df = aapl_df[2114:]
```

```
# Performing a train-test split
def train_test_split(ticker_df):
    for col in ticker_df.columns:
        ticker_df[col] = ticker_df[col].pct_change()
        ticker_df.dropna(inplace=True)
        ticker_df[col] = preprocessing.scale(ticker_df[col].values)

    ticker_df.dropna(inplace=True)
    # Creating sequential data
    sequential_data = []
    prev_days = deque(maxlen=pred_seq_len)

    for row in ticker_df.values:
        prev_days.append([n for n in row[:-1]])
        if len(prev_days) == pred_seq_len:
            sequential_data.append([np.array(prev_days), row[-1]])

    random.shuffle(sequential_data)

    # Get the train_X and train_Y portions of the data
    X = []
    y = []

    for seq, target in sequential_data:
        X.append(seq)
        y.append(target)

    return np.array(X), y
```

```
# Defining variables to use for the prediction step
pred_seq_len = 7 # Use past 7 days data to predict 8th day
pred_len = 1
```

To explain the code on how the batching of input data is done, first we run a for loop to run through all the training data we have. We will be creating batches of 7 rows of data (pred_seq_len) and 1 predicted value (pred_len). Essentially what this means is that 1 week of prior data will be used to predict the adjusted close price of the 8th day. The feature data is added to the train_X set, and the target values are added to the train_Y set. Once this is done, it is crucial to check the shape of the train_X and train_Y matrices to avoid running into errors during model fitting.

Now it is time to define the blueprint of our LSTM model. We will be adding a total of 5 layers in our neural network, three LSTM layer with 100 nodes, a Dense layer with 32 nodes and a final output dense layer with a single output node. We will use the rectified linear unit to . Once that is done, we will start preparing our input and target data to be fed to the neural network.

```
# Defining some useful constants
epochs = 10
batch_size = 50

# Creating the model
model = Sequential()
model.add(LSTM(100, input_shape=(train_X.shape[1:]),
return_sequences=True))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(LSTM(100, input_shape=(train_X.shape[1:]),
return_sequences=True))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(LSTM(100, input_shape=(train_X.shape[1:])))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(1, 'relu'))

# Defining the optimizer
opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)
```

Finally, we feed these input data lists into our LSTM model and make the predictions. Upon transforming the predictions back into their original “stock price” form, the predictions are as follows, in the order of predictions.head(), and predictions.tail().

Date	Adj Close	Predicted Adj Close	Diff
2012-05-14	-0.009047	-0.009100	0.000053
2012-05-15	-0.012817	-0.013920	0.001103
2012-05-16	-0.029227	-0.028323	-0.000904
2012-05-17	0.000491	0.000678	-0.000187
2012-05-18	0.058260	0.062390	-0.004130

Date	Adj Close	Predicted Adj Close	Diff
2022-03-02	0.020588	0.021423	-0.000835
2022-03-03	-0.001981	-0.000994	-0.000987
2022-03-04	-0.018408	-0.021345	0.002937
2022-03-07	-0.023718	-0.023638	-0.000080
2022-03-08	-0.011676	-0.020987	0.009311

After introducing all the models, it is also important to evaluate the models to understand them better. The first metric we use to evaluate the models is the accuracy score, which measures how many predictions the model gets right among all predictions. The formula of accuracy is the number of correct predictions divided by the total number of predictions. The Accuracy score will show how good the prediction is. The accuracy score of our model:

Moreover, the ROC curve is curve which shows the plot of the True Positive rate to the False Positive rate and the Area under the Curve is used for interpretation. This shows how the TP and FP will vary under different threshold for the classification model. The Area under the curve shows how likely the model will make a random positive example ranked higher than a random negative example. The fact that showing the predictions' ranking instead of showing the magnitude is a key strength of this ROC curve.

The results for predictions for both the Random Forest Regressor and LSTM models are comparable, despite the LSTM model taking into account the additional time series factor when forecasting the stock price. Hence, we can conclude that the Random Forest Regressor with some hyperparameter tuning yields comparable results to the baseline LSTM model.

Moreover, it is also important to apply these models in the context of the financial market. Technical analysis is used to generate alpha, so various models need to be demonstrated in place to understand the past stock price and make valid predictions.

References:

https://medium.com/@maryamuzakariya/project-predict-stock-prices-using-random-forest-regression-model-in-python-fbe4edf01664

https://neptune.ai/blog/hyperparameter-tuning-in-python-complete-guide

https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

https://scikit-learn.org/stable/

https://towardsdatascience.com/3-reasons-to-use-random-forest-over-a-neural-network-comparing-machine-learning-versus-deep-f9d65a154d89

https://builtin.com/data-science/random-forest-algorithm

https://wiki.pathmind.com/lstm