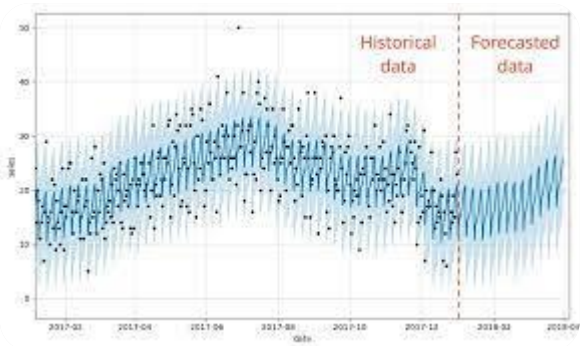


PHASE 4

ABSTRACT

Phase 4: Advance Modeling & Feature Engineering

Soumen Chatterjee

Phase 4 – 06-Mar-2022

- At this stage we will try with **Advanced Modeling and Feature Engineering**

Advanced modelling consists ideas like combining various models to obtain a better result while respecting real-world constraints and needs.

There are different ways of combining the models as stated below –
There are two approaches i.e. the **Ensemble** and **Hybrid** techniques that we can use to combine several algorithms together.

By ensemble we use multiple learning algorithms to obtain better predictive results than we could have used one algorithm. Ensemble methods falls under three categories **Bagging** (Bootstrap Aggregation) which combines the outputs of several algorithms in a way that decreases the variance of prediction by generating additional data for training from our original data set. Boosting basically increases the amount of training data.

Another ensemble approach is **Boosting** where we use the subset of our original data to produce several models then we boost their performance by combining them together using techniques such as voting.

The final ensemble method is **Stacking**, this is simply applying several algorithms to our original data in a stacked way. We train the base classifiers also referred to as weak learners then take the algorithm with the highest performance.

The second option of combining several algorithms together is to develop a **Hybrid** model. Unlike ensemble a hybrid model gives we flexibility of going beyond the ensemble techniques to create our innovative model. An example of hybrid model is in tasks of text classification where we can use Convolution Neural Network with Recurrent Neural Network to classify text. Convolution Network is used for feature extraction while Recurrent Network is for classification purpose.

Before combining algorithms we need to know the strengths and weaknesses of each algorithm we are considering to combine together.

Further explanation addition to above…

Multiple-Model Techniques
Ensemble learning is concerned with approaches that combine predictions from two or more models.

We can characterize a model as an ensemble learning technique if it has two properties, such as:

- Comprising two or more models.
- Predictions are combined.

We might also suggest that the goal of an ensemble model is to improve predictions over any contributing member. Although a lesser goal might be to improve the stability of the model, e.g. reduce the variance in the predictions or prediction errors.

Nevertheless, there are models and model architectures that contain elements of ensemble learning methods, but it is not clear as to whether they may be considered ensemble learning or not.

For example, we might define an ensemble learning technique as being composed of two or more models. The problem is that there may be techniques that have more than two models, yet do not combine their predictions. Alternatively, they may combine their predictions in unexpected ways.

For a lack of a better name, we will refer to these as “multiple-model techniques” to help differentiate them from ensemble learning methods. Yet, as we will see, the line that separates these two types of machine learning methods is not that clear.

Multiple-Model Techniques: Machine learning algorithms that are composed of multiple models and combine the techniques but might not be considered ensemble learning.
As such, it is important to review and explore multiple-model techniques that sit on the border of ensemble learning to both better understand ensemble learning and to draw upon related ideas that may improve the ensemble learning models we create.

There are predictive modeling problems where the structure of the problem itself may suggest the use of multiple models.

Typically, these are problems that can be divided naturally into sub-problems. This does not mean that dividing the problems into sub problems is the best solution for a given example; it only means that the problem naturally lends itself to decomposition.

Two examples are multi-class classification and multiple-output regression.

Multiple Models for Multi-Class Classification
Classification problems involve assigning a class label to input examples.

Binary classification tasks are those that have two classes. One decision is made for each example, either assigning it to one class or another. If modeled using probability, a single probability of the example being to one class is predicted, where the inverse is the probability for the second class, called a binomial probability distribution.

More than two classes can introduce a challenge. The techniques designed for two classes can be extended to multiple classes, and sometimes, this is straightforward.

Multi-Class Classification: Assign one among more than class labels to a given input example.
Alternatively, the problem can be naturally partitioned into multiple binary classification tasks.

There are many ways this can be achieved.

For example, the classes can be grouped into multiple one-vs-rest prediction problems. A model can then be fit for each subproblem and typically the same algorithm type is used for each model. When a prediction is required for a new example, then the model that responds more strongly than the other models can assign a prediction. This is called a one-vs-rest (OvR) or one-vs-all (OvA) approach.

- OvR: A technique that splits a multi-class classification into one binary classification problem per class.

The multi-class classification problem can be divided into multiple pairs of classes, and a model fit on each. Again, a prediction for a new example can be chosen from the model that responds more strongly. This is referred to as one-vs-one (OvO).

- OvR: A technique that splits a multi-class classification into one binary classification problem per each pair of classes.

For more on one-vs-rest and one-vs-one classification, see the tutorial:

This approach of partitioning a multi-class classification problem into multiple binary classification problems can be generalized. Each class can be mapped to a unique binary string for the class with arbitrarily length. One classifier can then be fit to predict each bit in the bit string, allowing an arbitrary number of classifiers to be used.

The bit string can then be mapped to the class label with the closest match. The additional bits act like error-correcting codes, improving the performance of the approach in some cases over simpler OvR and OvO methods. This approach is referred to as Error-Correcting Output Codes, ECOC.

- ECOC: A technique that splits a multi-class classification into an arbitrary number of binary classification problems.

In each of these cases, multiple models are used, just like an ensemble. Predictions are also combined, like an ensemble method, although in a winner-take-all method

rather than a vote or weighted sum. Technically, this is a combination method, but unlike most typical ensemble learning methods.

Unlike ensemble learning, these techniques are designed to explore the natural decomposition of the prediction problem and leverage binary classification problems that may not easily scale to multiple classes.

Whereas ensemble learning is not concerned with opening up new capabilities and is typically only focused on improved predictive performance over contributing models. With techniques like OvR, OvR, and ECOC, the contributing models cannot be used to address the prediction problem in isolation, by definition.

Multiple Models for Multi-Output Regression
Regression problems involve predicting a numerical value given an input example.

Typically, a single output value is predicted. Nevertheless, there are regression problems where multiple numeric values must be predicted for each input example. These problems are referred to as multiple-output regression problems.

Multiple-Output Regression: Predict two or more numeric outputs given an input. Models can be developed to predict all target values at once, although a multi-output regression problem is another example of a problem that can be naturally divided into subproblems.

Like binary classification in the previous section, most techniques for regression predictive modeling were designed to predict a single value. Predicting multiple values can pose a problem and requires the modification of the technique. Some techniques cannot be reasonably modified for multiple values.

One approach is to develop a separate regression model to predict each target value in a multi-output regression problem. Typically, the same algorithm type is used for each model. For example, a multi-output regression with three target values would involve fitting three models, one for each target.

When a prediction is required, the same input pattern is provided to each model and the specific target for each model is predicted and together represent the vector output of the method.

Multi-Output Regression: A technique where one regression model is used for each target in a multi-output regression problem.

Another related approach is to create a sequential chain of regression models. The difference is that the output of the first model predicts the first output target value, but this value is used as part of the input to the second model in the chain in order to predict the second output target value, and so on.

As such, the chain introduces a linear dependence between the regression models, allowing the outputs of models later in the chain to be conditional on the outputs of prior models in the chain.

Regression Chain: A technique where a sequential chain of regression models is used to predict each target in a multi-output regression problem, one model later in the chain uses values predicted by models earlier in the chain.

In each case, multiple regression models are used, just like an ensemble.

A possible difference from ensembles is that the predictions made by each model are not combined directly. We could stretch the definition of “combining predictions” to cover this approach, however. For example, the predictions are concatenated in the case of multi-output regression models and indirectly via the conditional approach in chained regression.

The key difference from ensemble learning methods is that no contributing ensemble member can solve the prediction problem alone. A solution can only be achieved by combining the predictions from all members.

Multiple Expert Models

So far, we have looked at dividing problems into subtasks based on the structure of what is being predicted.

There are also problems that can be naturally divided into sub problems based on the input data. This might be as simple as partitions of the input feature space, or something more elaborate, such as dividing an image into the foreground and background and developing a model for each.

A more general approach for this from the field of neural networks is referred to as a mixture of experts (MoE).

The approach involves first dividing the learning task into subtasks, developing an expert model for each subtask, using a gating model to decide or teach which expert to

use for each example and the pool the outputs of the experts, and gating model together to make a final prediction.

MoE: A technique that develops an expert model for each subtask and learns how much to trust each expert when making a prediction for specific examples.

wo aspects of MoE make the method unique. The first is the explicit partitioning of the input feature space, and the second is the use of a gating network or gating model that learns which expert to trust in each situation, e.g., each input case.

Unlike the previous examples for multi-class classification and multi-output regression that divided the target into sub problems, the contributing members in a mixture of experts model can address the whole problem, at least partially or to some degree. Although an expert may not be tailored to a specific input, it can still be used to make a prediction on something outside of its area of expertise.

Also, unlike those previously reviewed methods, a mixture of experts also combines the predictions of all contributing members using a weighted sum, albeit measured by the gating network.

As such, it more closely resembles more familiar ensemble learning techniques, such as stacked generalization, known as stacking.

Hybrids Constructed From Multiple Models

Another type of machine learning that involves the use of multiple models and is loosely related to ensemble learning is hybrid models.

Hybrid models are those models that combine two or more models explicitly. As such, the definition of what does and does not constitute a hybrid model can be vague.

Hybrid Model: A technique that combines two or more different machine learning models in some way.

For example, an autoencoder neural network that learns how to compress input patterns to a bottleneck layer, the output of which is then fed to another model, such as a support vector machine, would be considered a hybrid machine learning model.

This example has two machine learning models, a neural network and a support vector machine. It just so happens that the models are linearly stacked one on top of another into a pipeline and the last model in the pipeline makes a prediction.

Consider an ensemble learning method that has multiple contributing ensemble members of different types (e.g. a logistic regression and a support vector machine) and uses voting to average their predictions. This ensemble too might be considered a hybrid machine learning model, under the broader definition.

Perhaps the key difference between ensemble learning and hybrid machine learning is the need in hybrid models to use models of differing types. Whereas in ensemble learning, contributing members to the ensemble may be of any type.

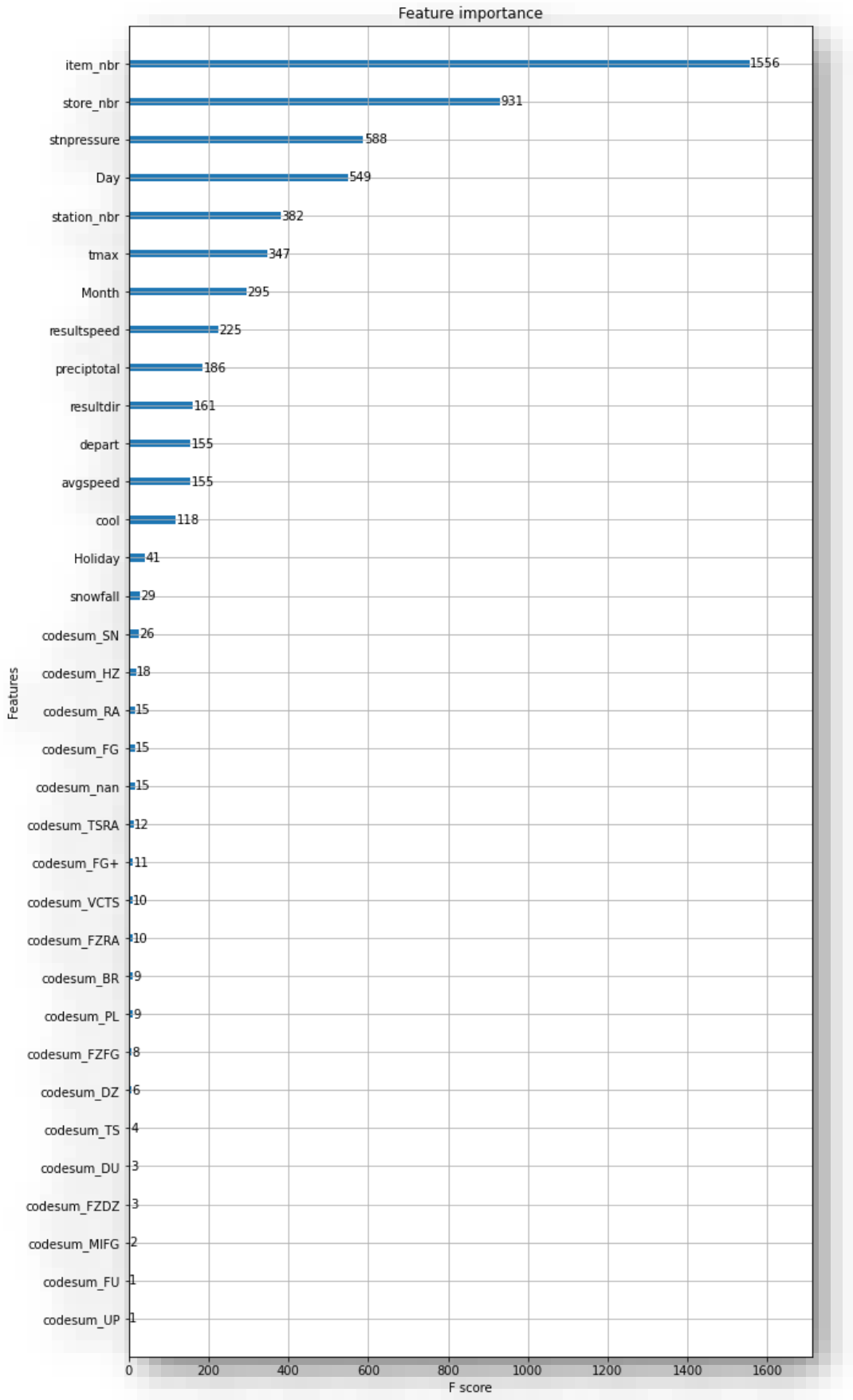
Further, it is more likely that a hybrid machine learning will graft one or more models onto another base model, which is quite different from fitting separate models and combining their predictions as we do in ensemble learning.

- Given the above detailed explanation, examples, pros, cons of combining different models through the above explained techniques, we found that the issue we are dealing with does not meet any requirement to use the combination of models.

Whereas we have used two different types of model to deal with this, one the classification problem to understand whether there is any sell or not and based on the prediction we have sent the query point to a regression model to predict the future sells.

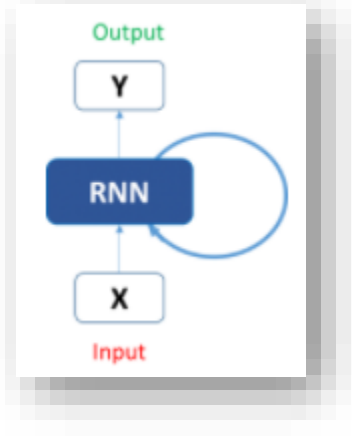
- We have also implemented the **advanced feature encoding and engineering methods**

Like we have encoded "**codesum**" variable which is categorical in nature to 29 features. Created new features like "Day" "Month" "Holiday". And below is the measure of impact for the newly engineered features and the existing one.
Below feature importance shows the newly engineered features like "Day" "Month" "Holiday" has great values whereas 29 codesum features created through one hot encoding do not have much importance.
So we removed those newly added codesum features while implementing advance modelling.

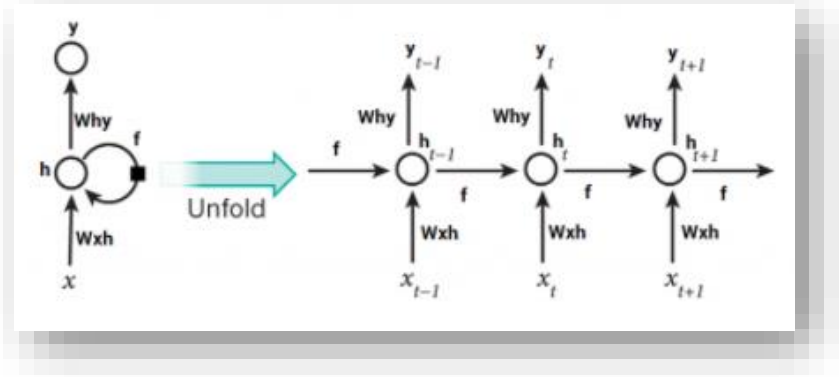

Feature importance
item_nbr
store_nbr
stnpressure
Day
station_nbr
tmax
Month
resultspeed
preciptotal
resultdir
depart
avgspeed
cool
Holiday
snowfall
codesum_SN
codesum_HZ
codesum_RA
codesum_FG
codesum_nan
codesum_TSRA
codesum_FG+
codesum_VCTS
codesum_FZRA
codesum_BR
codesum_PL
codesum_FZFG
codesum_DZ
codesum_TS
codesum_DU
codesum_FZDZ
codesum_MIFG
codesum_FU
codesum_UP
1556
931
588
549
382
347
295
225
186
161
155
155
118
41
29
26
18
15
15
15
12
11
10
10
9
9
8
6
4
3
3
2
1
1
Features
0
200
400
600
800
1000
1200
1400
1600
F score

- **Next we have tried with Advance modelling technique like neural networks – LSTM**

A brief details on Long Short-Term Memory Units (LSTMs) which we used for multivariate single step time series forecasting. We used LSTMs as it has an edge over conventional feed-forward neural networks and RNN in many ways. This is because of their property of selectively remembering patterns for long durations of time. The problem what we are solving is basically predicting future stocks based on everyday sales of past few years.
As for a specific store location the weather features, demographics are kind of same throughout the years. So we can use the past data to predict future **using Recurrent Neural Networks (RNN).**



- **Typical RNN looks like**



- **Expanded**

Recurrent Neural Networks work just fine when we are dealing with short-term dependencies. However, vanilla RNNs fail to understand the context behind an input for something that happened long before, cannot be recalled when making predictions in the present. The reason behind this is the problem of **Vanishing Gradient**.
As we know that for a conventional feed-forward neural network, the weight updating that is applied on a particular layer is a multiple of the learning rate, the error term from the previous layer and the input to that layer. Thus, the error term for a particular layer is somewhere a product of all previous layers' errors. When dealing with activation functions like the sigmoid function, the small values of its derivatives (occurring in the error function) gets multiplied multiple times as we move towards the starting layers. As a result of this, the gradient almost vanishes as we move towards the starting layers, and it becomes difficult to train these layers.

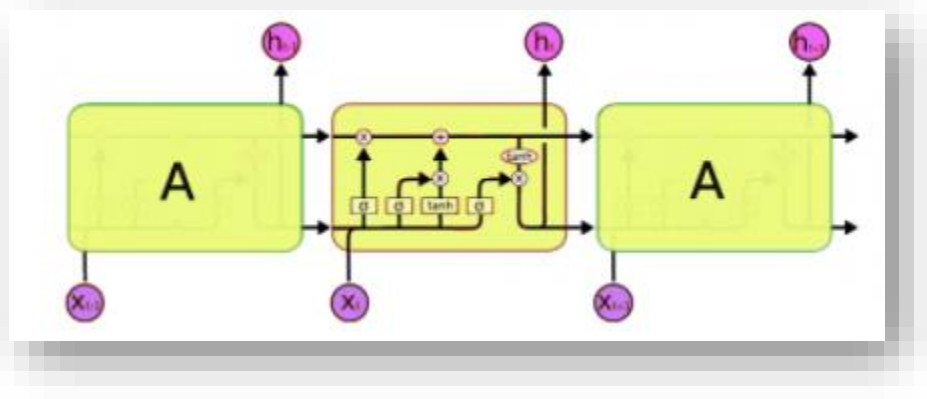
LSTMs on the other hand, make small modifications to the information by multiplications and additions. With LSTMs, the information flows through a mechanism known as cell states. This way, LSTMs can selectively remember or forget things. The information at a particular cell state has three different dependencies.

We'll visualize this with an example. Let's take the example of predicting stock prices for a particular stock. The stock price of today will depend upon:

The trend that the stock has been following in the previous days, maybe a downtrend or an uptrend.
The price of the stock on the previous day, because many traders compare the stock's previous day price before buying it.
The factors that can affect the price of the stock for today. This can be a new company policy that is being criticized widely, or a drop in the company's profit, or maybe an unexpected change in the senior leadership of the company.
These dependencies can be generalized to any problem as:

1. The previous cell state (i.e. the information that was present in the memory after the previous time step)
2. The previous hidden state (i.e. this is the same as the output of the previous cell)
3. The input at the current time step (i.e. the new information that is being fed in at that moment)

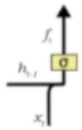
Architecture of LSTMs



A typical LSTM network is comprised of different memory blocks called cells
(The rectangles that we see in the image). There are two states that are being transferred to the next cell; the cell state and the hidden state. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called gates.

Forget Gate

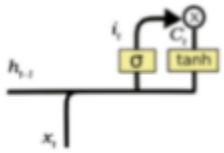
A forget gate is responsible for removing information from the cell state. The information that is no longer required for the LSTM to understand things or the information that is of less importance is removed via multiplication of a filter. This is required for optimizing the performance of the LSTM network.



This gate takes in two inputs; h_t-1 and x_t.

h_t-1 is the hidden state from the previous cell or the output of the previous cell and x_t is the input at that particular time step. The given inputs are multiplied by the weight matrices and a bias is added. Following this, the sigmoid function is applied to this value. The sigmoid function outputs a vector, with values ranging from 0 to 1, corresponding to each number in the cell state. Basically, the sigmoid function is responsible for deciding which values to keep and which to discard. If a ‘0’ is output for a particular value in the cell state, it means that the forget gate wants the cell state to forget that piece of information completely. Similarly, a ‘1’ means that the forget gate wants to remember that entire piece of information. This vector output from the sigmoid function is multiplied to the cell state.

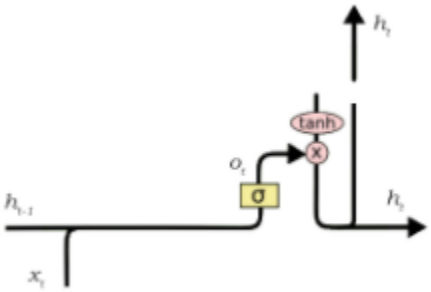
Input Gate



The input gate is responsible for the addition of information to the cell state. This addition of information is basically three-step process as seen from the diagram above.

1. Regulating what values need to be added to the cell state by involving a sigmoid function. This is basically very similar to the forget gate and acts as a filter for all the information from h_t-1 and x_t.
2. Creating a vector containing all possible values that can be added (as perceived from h_t-1 and x_t) to the cell state. This is done using the tanh function, which outputs values from -1 to +1.
3. Multiplying the value of the regulatory filter (the sigmoid gate) to the created vector (the tanh function) and then adding this useful information to the cell state via addition operation.

Output Gate



The functioning of an output gate can again be broken down to three steps:

1. Creating a vector after applying tanh function to the cell state, thereby scaling the values to the range -1 to +1.
2. Making a filter using the values of h_t-1 and x_t, such that it can regulate the values that need to be output from the vector created above. This filter again employs a sigmoid function.
3. Multiplying the value of this regulatory filter to the vector created in step 1, and sending it out as an output and also to the hidden state of the next cell.

We have tried using LSTMS with 3 different models –
a)

```
model = Sequential()
model.add(InputLayer((30, 14)))
model.add(LSTM(64))
model.add(Dense(8, 'relu'))
model.add(Dense(1, 'linear'))
```

Below is what it interpret:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 64)                20224

 dense_2 (Dense)             (None, 8)                 520

 dense_3 (Dense)             (None, 1)                 9

=================================================================
Total params: 20,753
Trainable params: 20,753
Non-trainable params: 0
_________________________________________________________________
```

Below is the Error analysis of Model 1:

Epoch 30/30
2666/2666 [==============================] - 11s 4ms/step - loss: 867.7957 - **root_mean_squared_error: 29.4584** - val_loss: 717.9977 - **val_root_mean_squared_error: 26.7955**

2. Model 2:

```
model2 = tf.keras.Sequential()
model2.add(tf.keras.layers.LSTM(128,input_shape=(30,14),return_sequences=True))
model2.add(tf.keras.layers.LeakyReLU(alpha=0.5))
model2.add(tf.keras.layers.LSTM(128,return_sequences=True))
model2.add(tf.keras.layers.LeakyReLU(alpha=0.5))
model2.add(tf.keras.layers.Dropout(0.3))
model2.add(tf.keras.layers.LSTM(64,return_sequences=True))
model2.add(tf.keras.layers.Dropout(0.3))
model2.add(tf.keras.layers.Dense(1))
```

Below is what it interpret:

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_5 (LSTM)               (None, 30, 128)           73216

 leaky_re_lu_2 (LeakyReLU)   (None, 30, 128)           0

 lstm_6 (LSTM)               (None, 30, 128)           131584

 leaky_re_lu_3 (LeakyReLU)   (None, 30, 128)           0

 dropout_2 (Dropout)         (None, 30, 128)           0

 lstm_7 (LSTM)               (None, 30, 64)            49408

 dropout_3 (Dropout)         (None, 30, 64)            0

 dense_5 (Dense)             (None, 30, 1)             65

=================================================================
Total params: 254,273
Trainable params: 254,273
Non-trainable params: 0
_________________________________________________________________
```

Below is the Error analysis of Model 2:

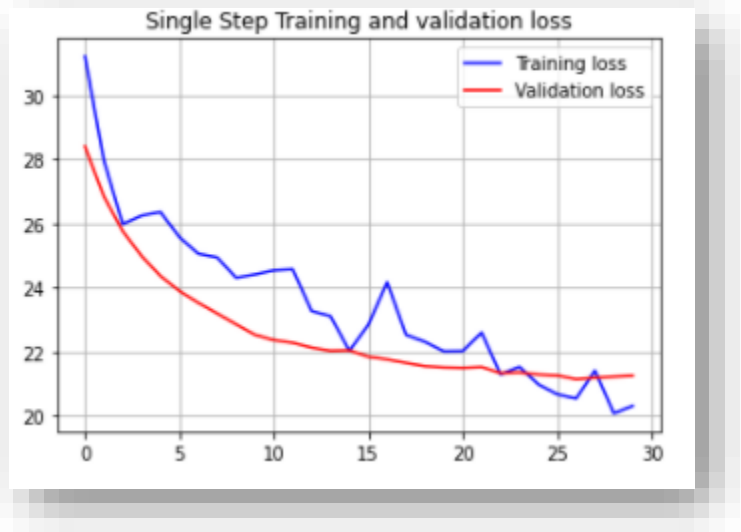
Epoch 30/30
2666/2666 [==============================] - 23s 9ms/step - loss: 1083.9133 - **root_mean_squared_error: 32.9230** - val_loss: 803.0638 - **val_root_mean_squared_error: 28.3410**

4. Model 3 (Best predicting)

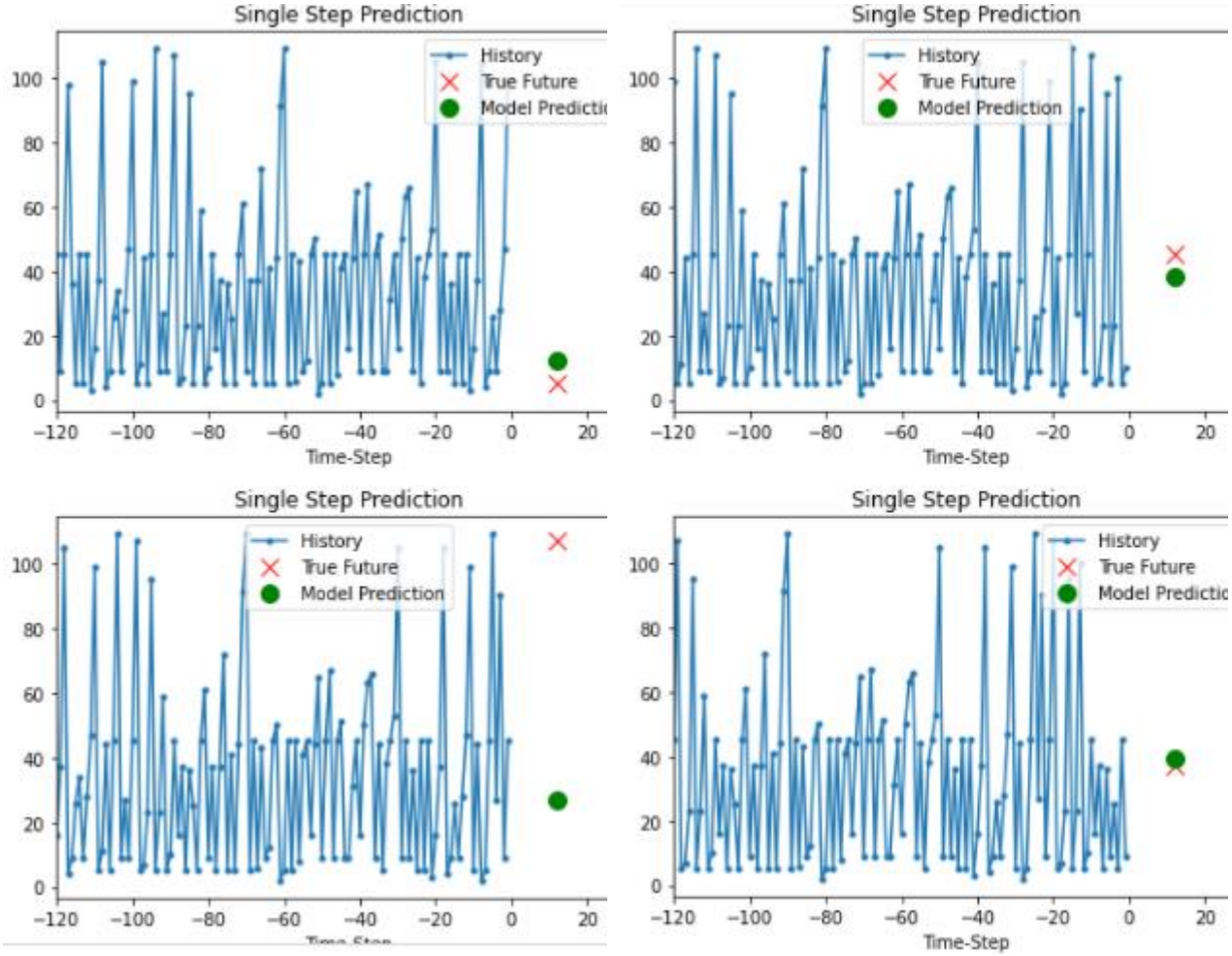
```
# Model
single_step_model = tf.keras.models.Sequential()
single_step_model.add(tf.keras.layers.LSTM(32,
                                           input_shape=x_train_single.shape[-2:]))
single_step_model.add(tf.keras.layers.Dense(1))

single_step_model.compile(optimizer=tf.keras.optimizers.RMSprop(), loss='mae')

single_step_history = single_step_model.fit(train_data_single, epochs=EPOCHS,
                                            steps_per_epoch=STEPS_PER_EPOCH,
                                            validation_data=val_data_single,
                                            validation_steps=50)
```

Below is the Error analysis of Model 2:

```
Epoch 30/30
200/200 [==============================] - 1s 6ms/step - loss: 20.3042 -
val_loss: 21.2426
```



Visualizing the prediction:



By this we are done with Phase 4.

References

1. https://machinelearningmastery.com/multiple-model-machine-learning/#:~:text=Hybrid%20Model%3A%20A%20technique%20that,learning%20models%20in%20some%20way.
2. https://www.quora.com/How-do-I-combine-machine-learning-models
3. https://analyticsindiamag.com/a-complete-guide-to-lstm-architecture-and-its-use-in-text-classification/
4. https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/
5. https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/