Dataset Description

- The training dataset consists of approximately 145k time series.
- Each of these time series represent a number of daily views of a different Wikipedia article, starting from July 1st, 2015 up until December 31st, 2016.
- *Article name_ wiki project_type of access_agent* is the format for Page column

	Page	2015-07-01	2015-07-02	2015-07-03	2015-07-04	2015-07-05	2015-07-06	2015-07-07	2015-07-08	2015-07-09	...
0	2NE1_zh.wikipedia.org_all-access_spider	18.0	11.0	5.0	13.0	14.0	9.0	9.0	22.0	26.0	...
1	2PM_zh.wikipedia.org_all-access_spider	11.0	14.0	15.0	18.0	11.0	13.0	22.0	11.0	10.0	...
2	3C_zh.wikipedia.org_all-access_spider	1.0	0.0	1.0	1.0	0.0	4.0	0.0	3.0	4.0	...
3	4minute_zh.wikipedia.org_all-access_spider	35.0	13.0	10.0	94.0	4.0	26.0	14.0	9.0	11.0	...
4	52_Hz_I_Love_You_zh.wikipedia.org_all-access_s...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...

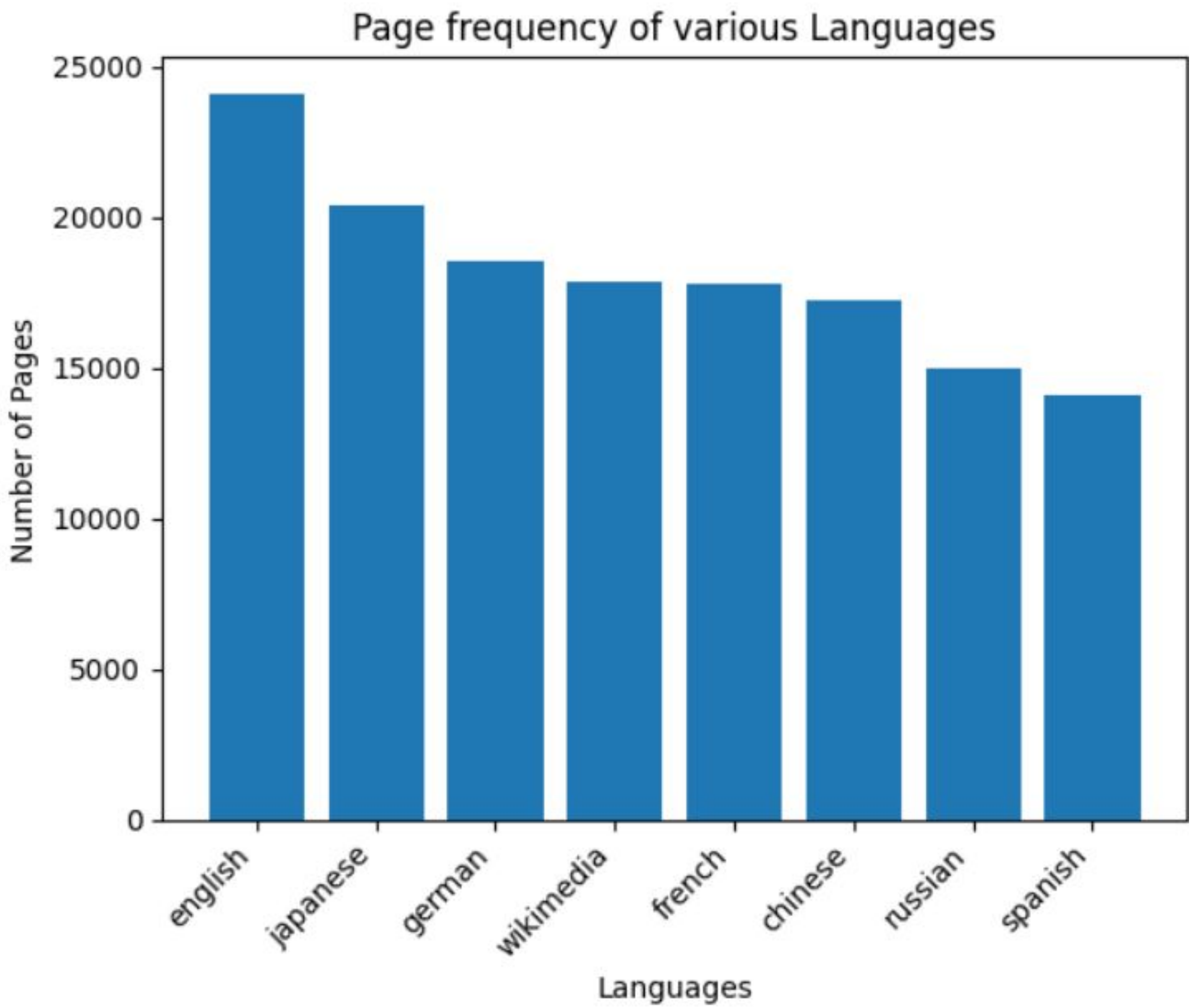
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145063 entries, 0 to 145062
Columns: 551 entries, Page to 2016-12-31
dtypes: float64(550), object(1)
memory usage: 609.8+ MB
```

Dataset Exploration | Impact on traffic by language

```
def get_language(page):
    res = re.search('[a-z][a-z].wikipedia.org',page)
    if res:
        return res[0][0:2]
    return 'na'

train['lang'] = train.Page.map(get_language)
from collections import Counter
print(Counter(train.lang))
```

Regex on page name to identify the language



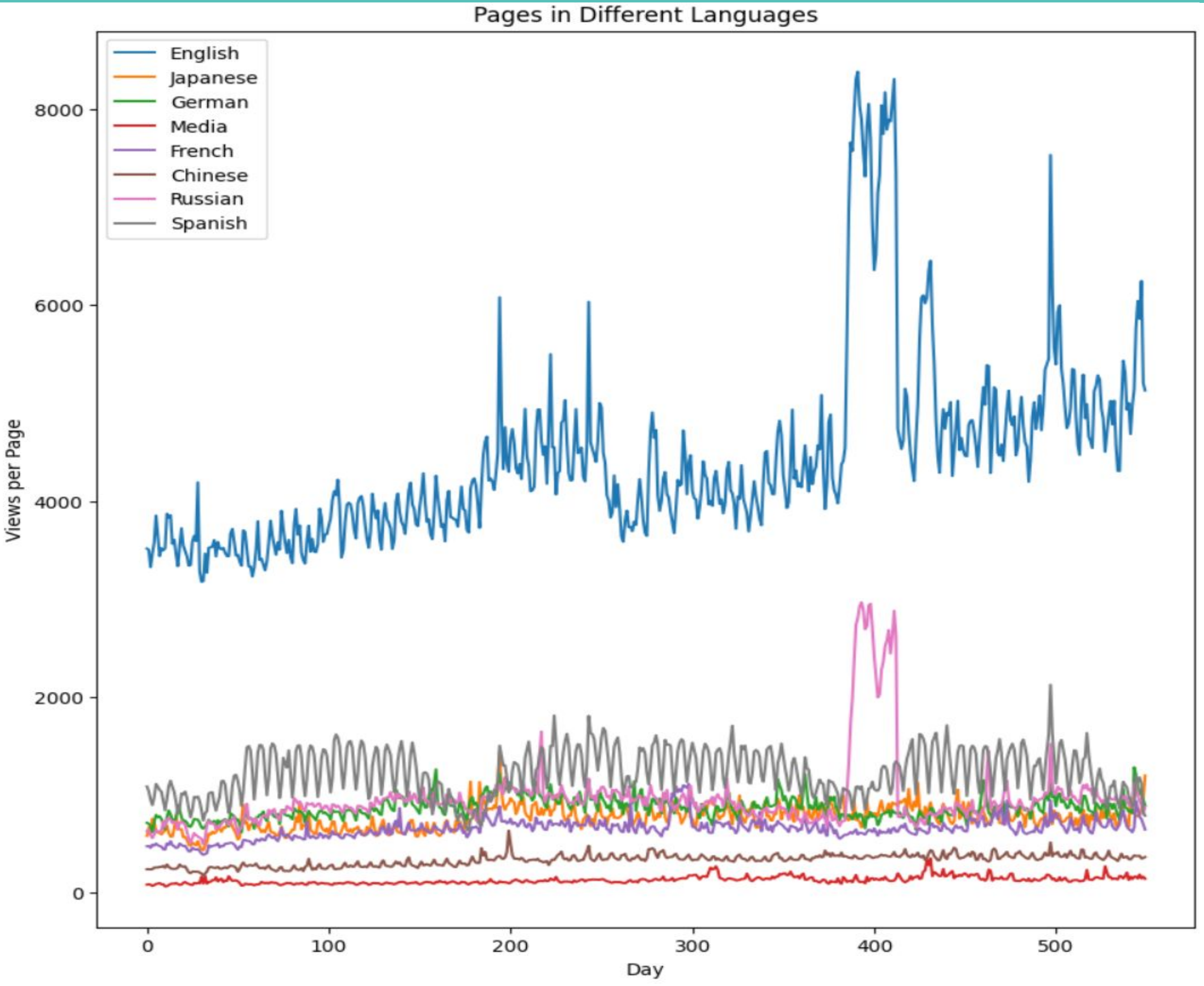
Dataset Exploration| Impact on traffic by language

- Impact on traffic by language

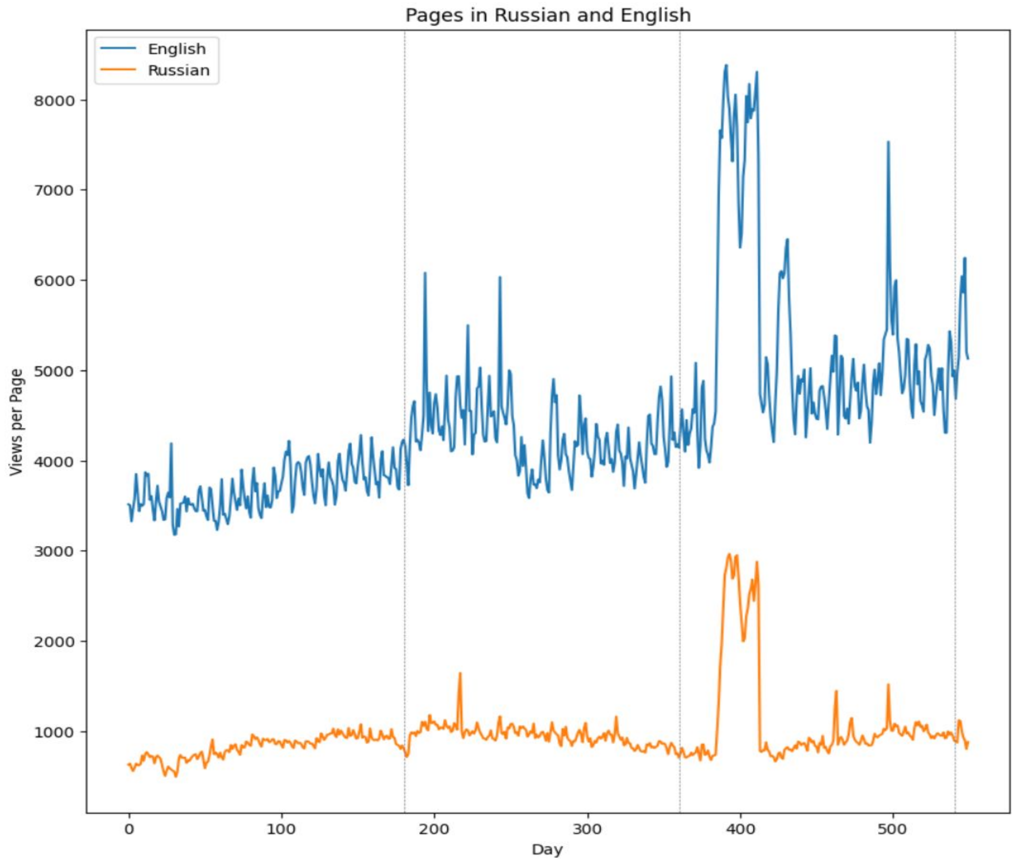
```
lang_sets = {}
lang_sets['en'] = train[train.lang=='en'].iloc[:,0:-1]
lang_sets['ja'] = train[train.lang=='ja'].iloc[:,0:-1]
lang_sets['de'] = train[train.lang=='de'].iloc[:,0:-1]
                 .
                 .
                 .
```

1. Creating a dictionary of datasets, wherein we have separate dataset for each language.
2. Taking the aggregate views per language and dividing with number of pages to get views per page

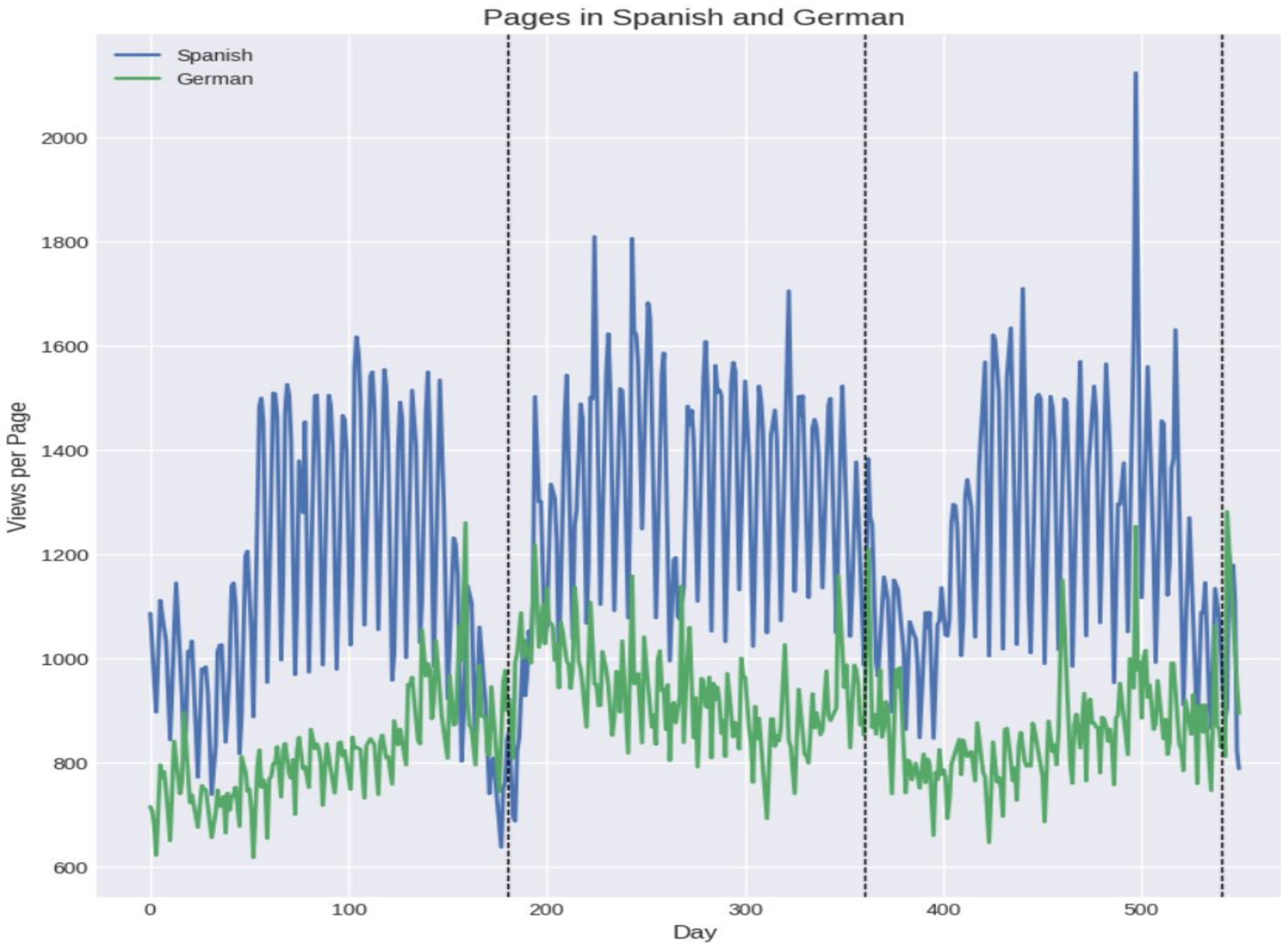
- As expected traffic on the english page is 4-5x higher than the other languages.
- Spanish has the second highest traffic on a time averaged basis followed closely by russian.



Dataset Exploration| Impact on traffic by language



1. English and russian pages show very strong correlation.
2. Notice the giant peak around day 400 for both these languages. It is Aug - 2016 start of summer olympics
3. Trend shows cycles in period of 180 days

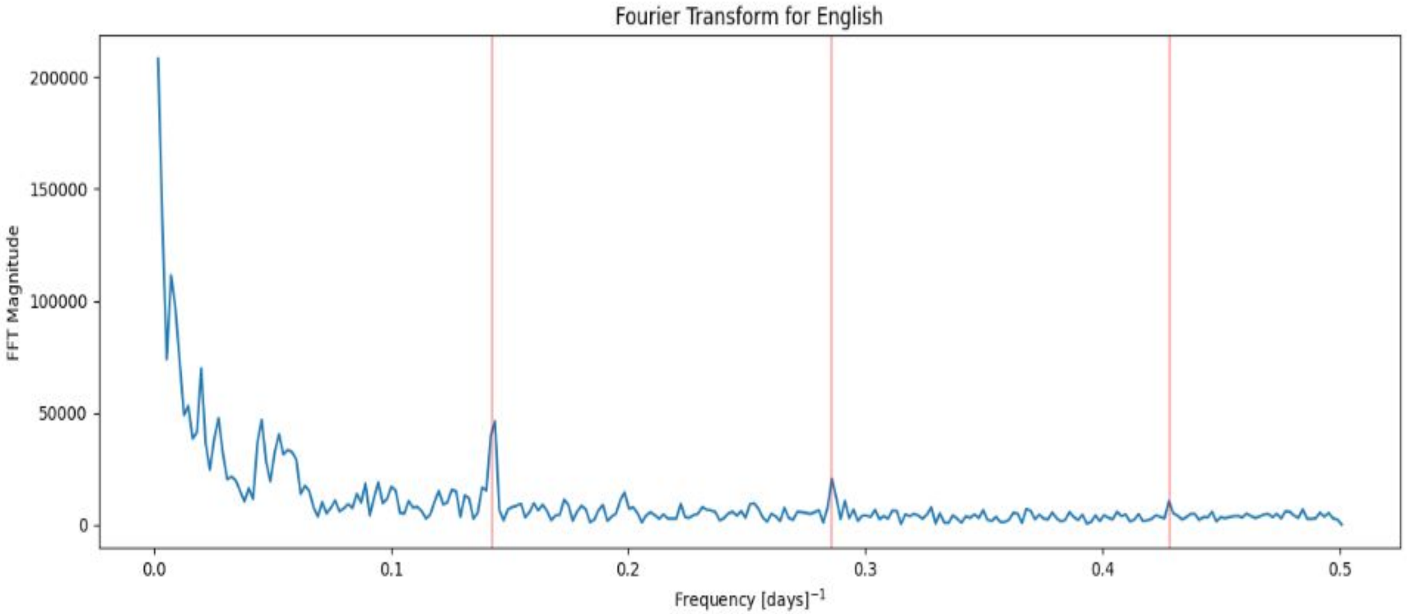


1. Spanish and German pages show very strong cyclic behaviour with 180 days period

Dataset Exploration

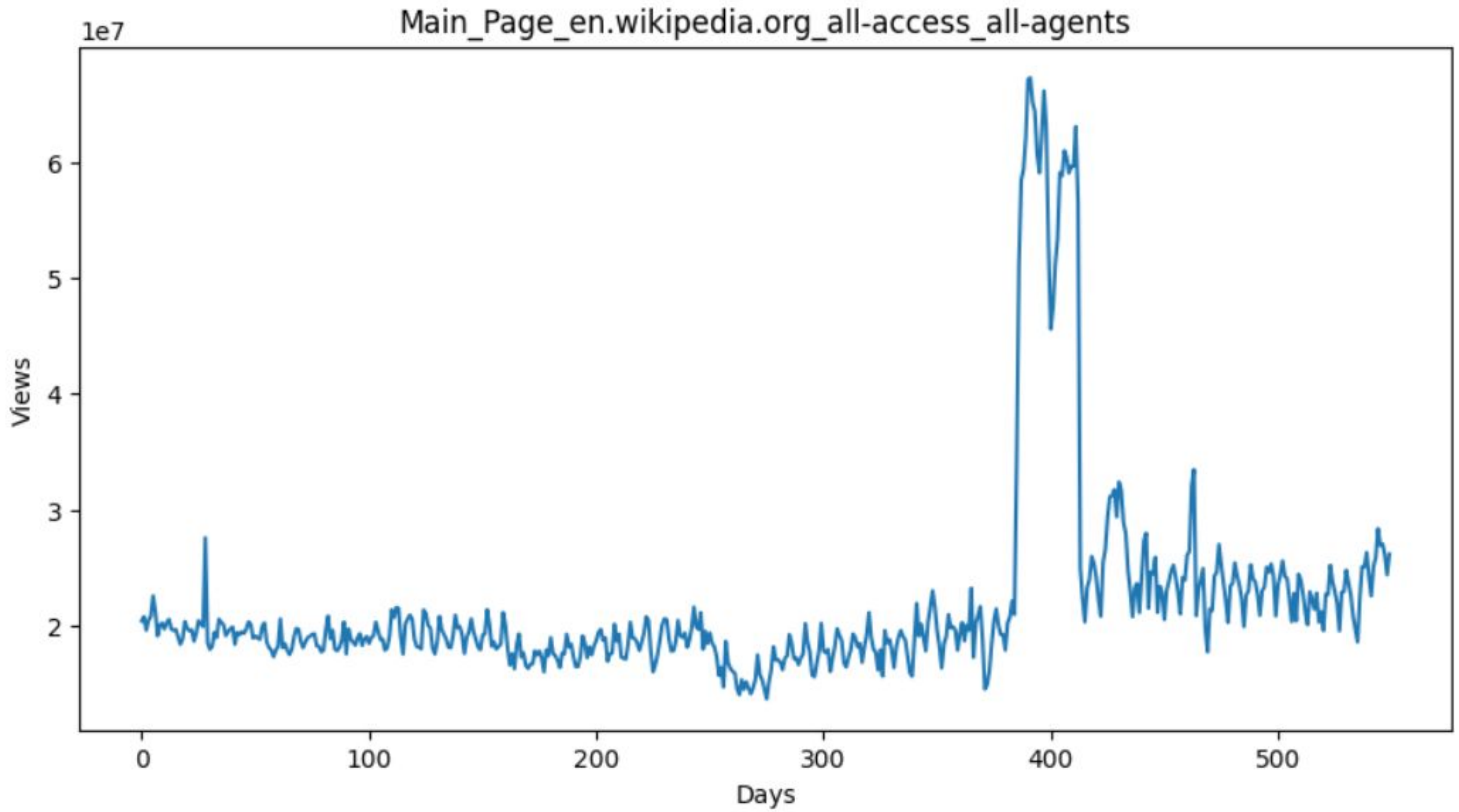
- Further we will be looking at English Language

In the FFT plot we see peaks at 1/7 and 2/7 indicating weekly and bi-weekly cycles.



Problem Definition

- For the Wikipedia English Main page, predict viewership for the next **20 timesteps(days).**
  - This gives sufficient time for inference and optimal response at the server side
  - Checks for robustness of the model
- Chosen performance metric is Mean squared Error

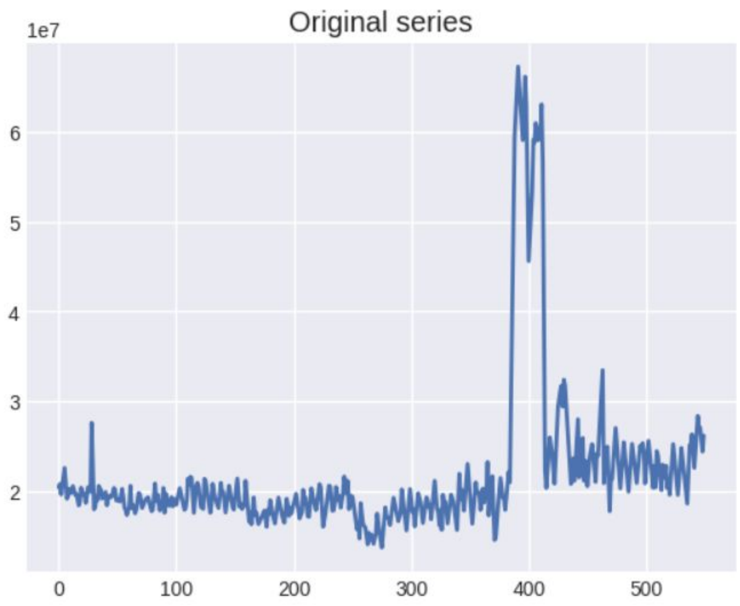


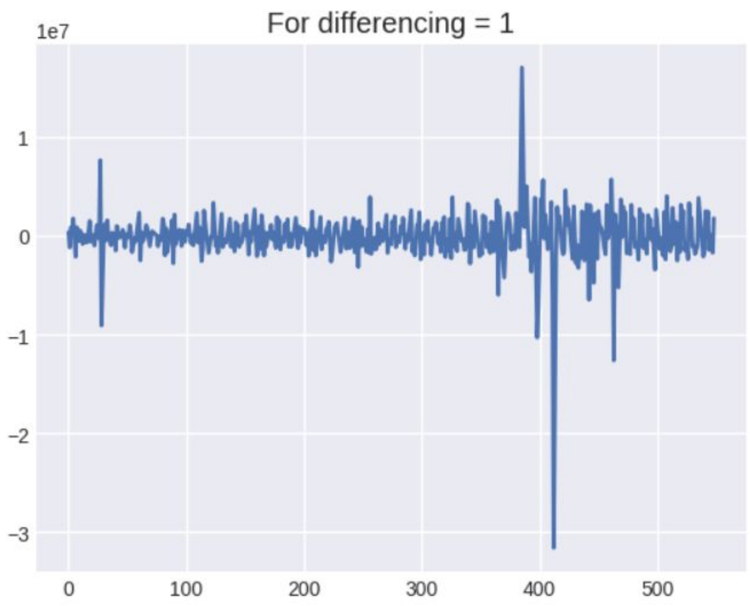
Benchmark

- For Benchmark we will be using FBprophet released by Meta.
- In addition, we will construct an ARIMA model to establish a baseline for assessing the performance of the deep learning models.
    - To apply we need to find 3 parameters
        - p → lag order
        - d → degree of differencing
        - q → order of moving average

ARIMA | Finding parameters

- Finding value of d
  - No. of times raw observations undergo differencing for stationarity
  - Stationarity refers to constant mean and variance of the data remain over time





- For d = 1 we can see constant mean but can't determine if variance is acceptable
- So we perform AD-Fuller test to determine the same

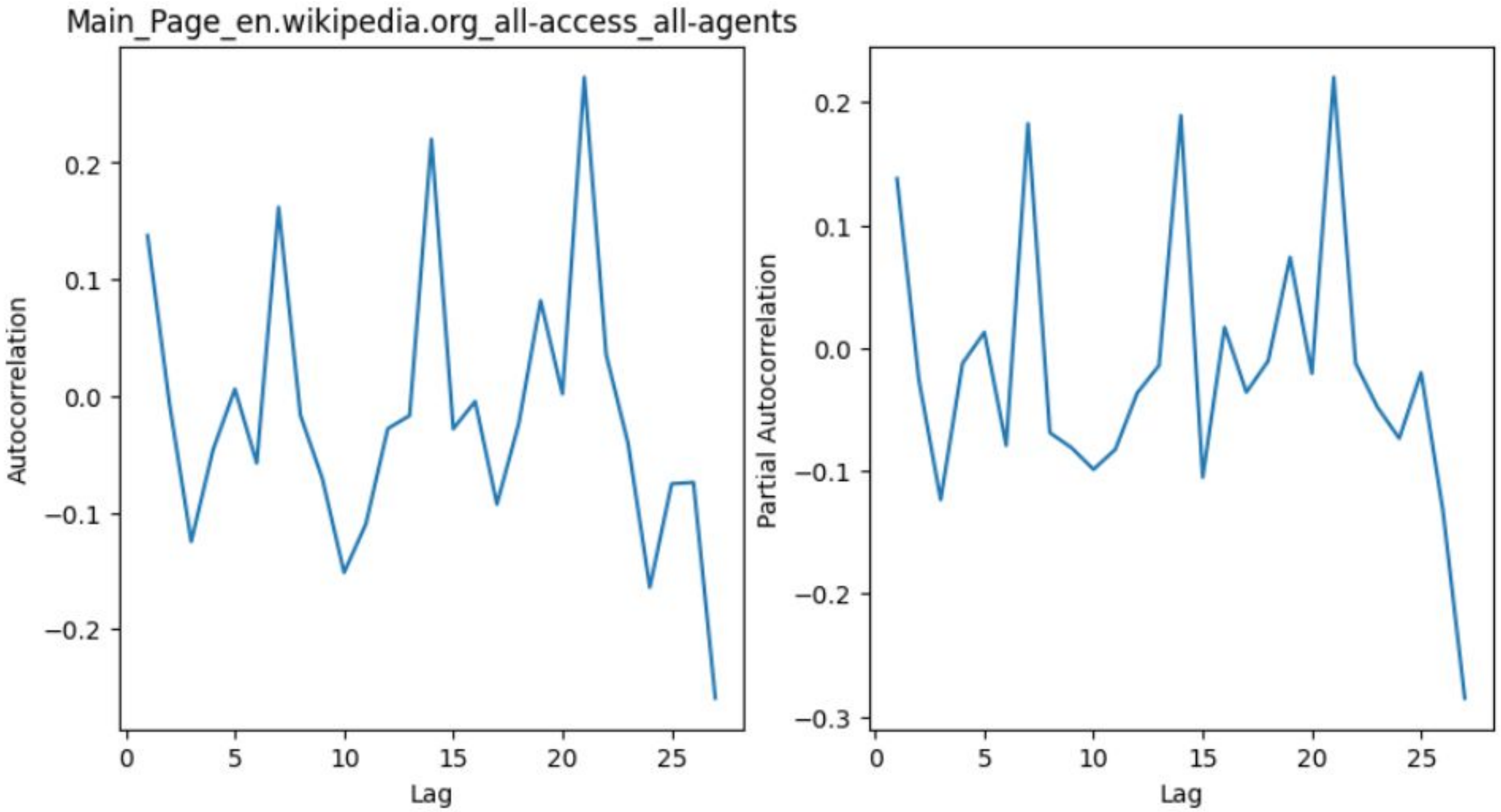
```
from statsmodels.tsa.stattools import adfuller

p_value = adfuller(data_diff)
print("\n",p_value[1])
```

```
5.481911439181588e-09
```

- This is below our threshold of 0.05 so we accept the value of d as 1.

ARIMA | Finding parameters

- Finding value of p and q
  - p - number of lag observations incorporated in the model
  - q - size of moving average window



- As we can't visually predict the parameters we will be performing Grid search.

- Point where Autocorrelation drops off is the order of moving average i.e q
- Point where Partial Autocorrelation spikes is the order of lag i.e p

ARIMA | Finding parameters

- Finding value of p and q through GridSearch

```
for p in range(7):
    for q in range(7):
        model = SARIMAX(data, order=(p,1,q)) #because adf test showed that d=1
        results = model.fit()
        # Append order and results tuple
        order_aic_bic.append((p,q,results.aic, results.bic))

order_df['total'] = order_df['AIC']+order_df['BIC']
order_df.loc[order_df.total == min(order_df['total'])]
```

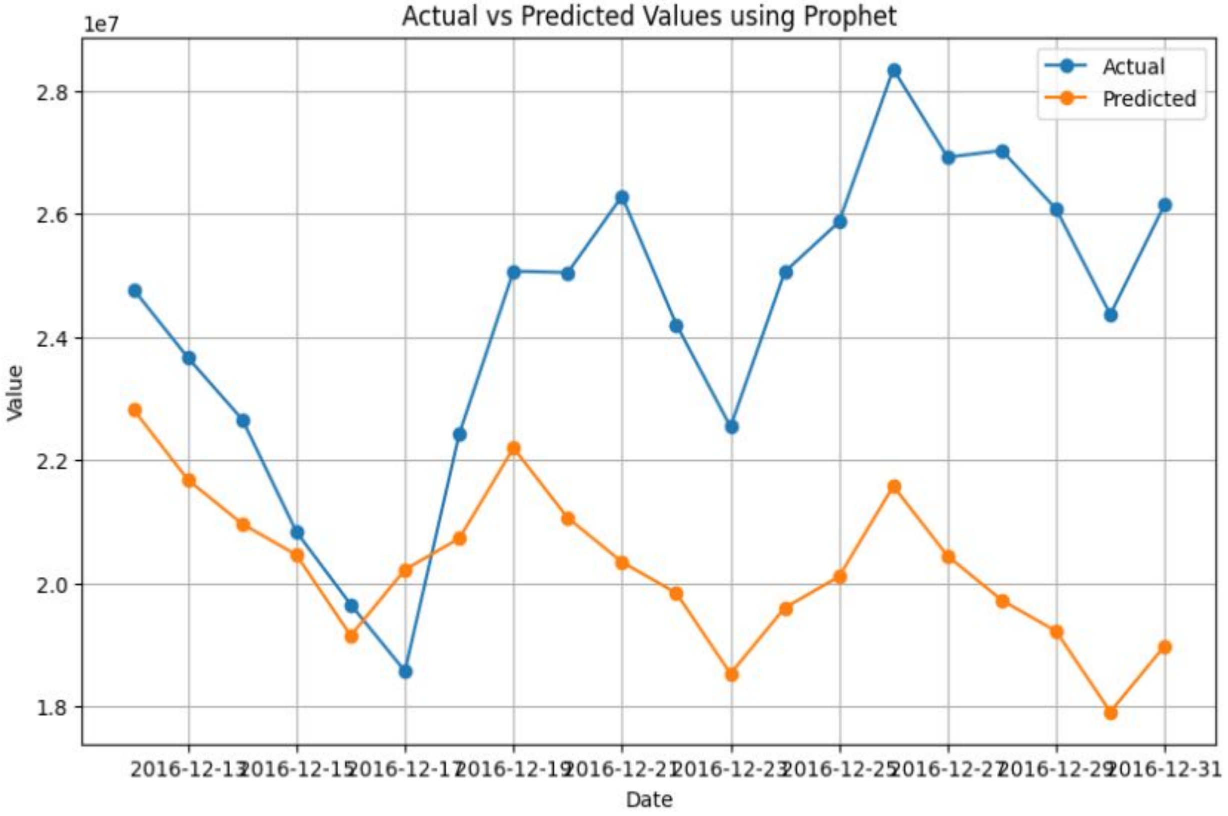
	p	q	AIC	BIC	total
33	4	5	17659.242343	17702.323327	35361.56567

- AIC and BIC are information criteria metrics used to describe fit of a model. They can be thought of as loss i.e lower the better
- We combine the two as they are of similar scale and select the parameter with minimum total

- Using these parameters (4,1,5) we make our ARIMA model and plot it against the Actual values

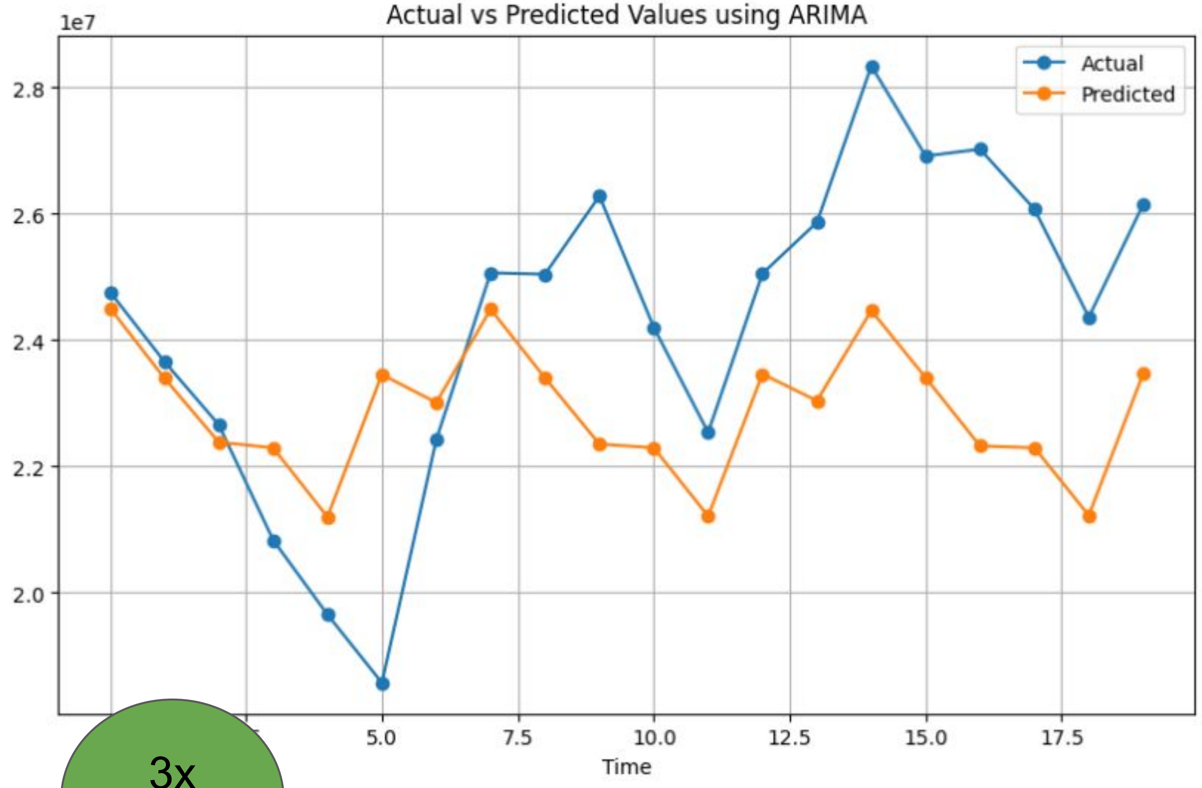
Benchmark

- FBProphet




```
Mean Squared Error: 22740395714770.99
Error percent at 1st step: 7.877016194077989 %
Error percent at 20th step: 27.404859378893313 %
Max error is 27.404859378893313 %
```

- ARIMA



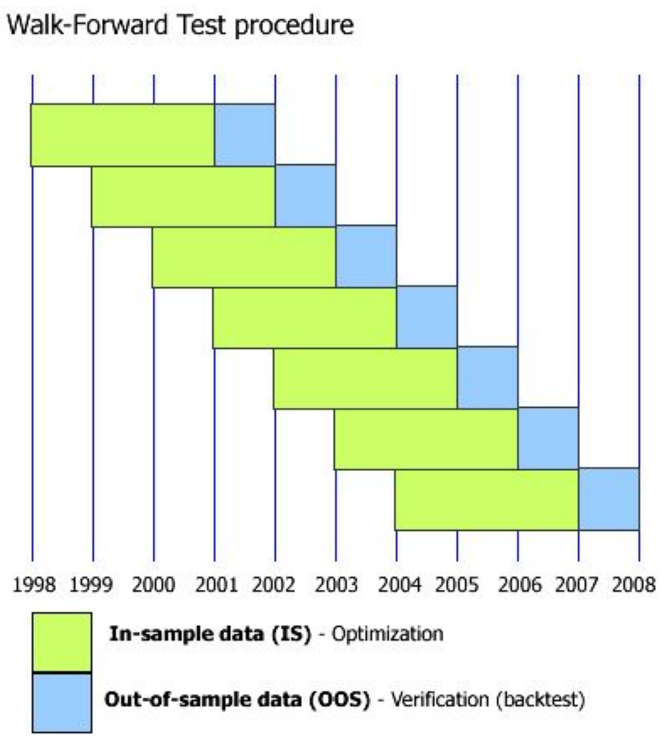
3x better

```
Mean Squared Error: 7203327963531.891
Error percent at 1st step: 1.032681172174155 3 %
Error percent at 20th step: 10.245091979780042 %
Max error is 17.395344549761337 %
```

Neural Network Approaches

- Vector Model
- Encoder-Decoder Model
  - Simple attention
  - Simplified Multi-headed attention
  - CNN filters to replicate attention

Data Preparation

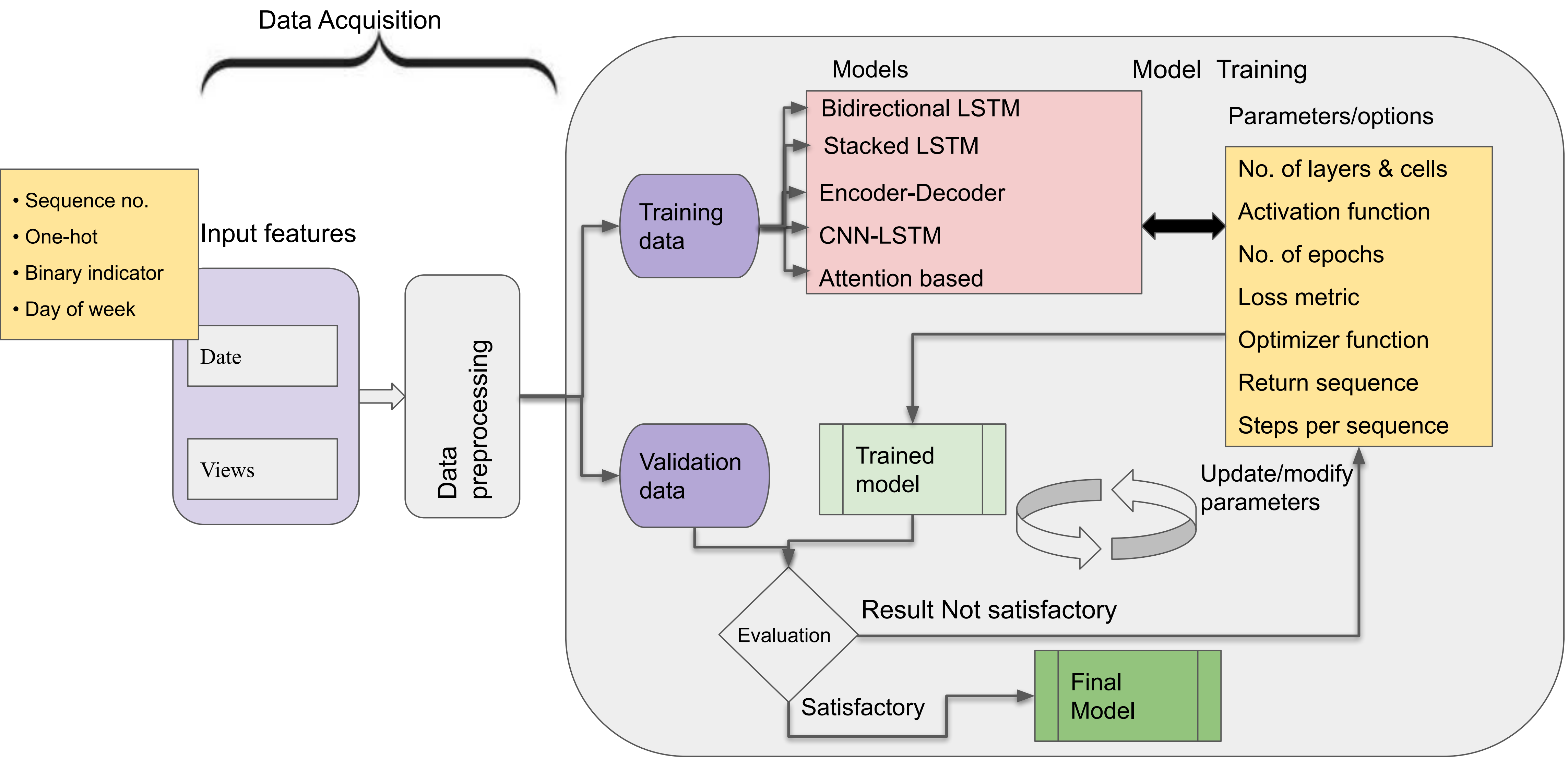


Source : stackoverflow

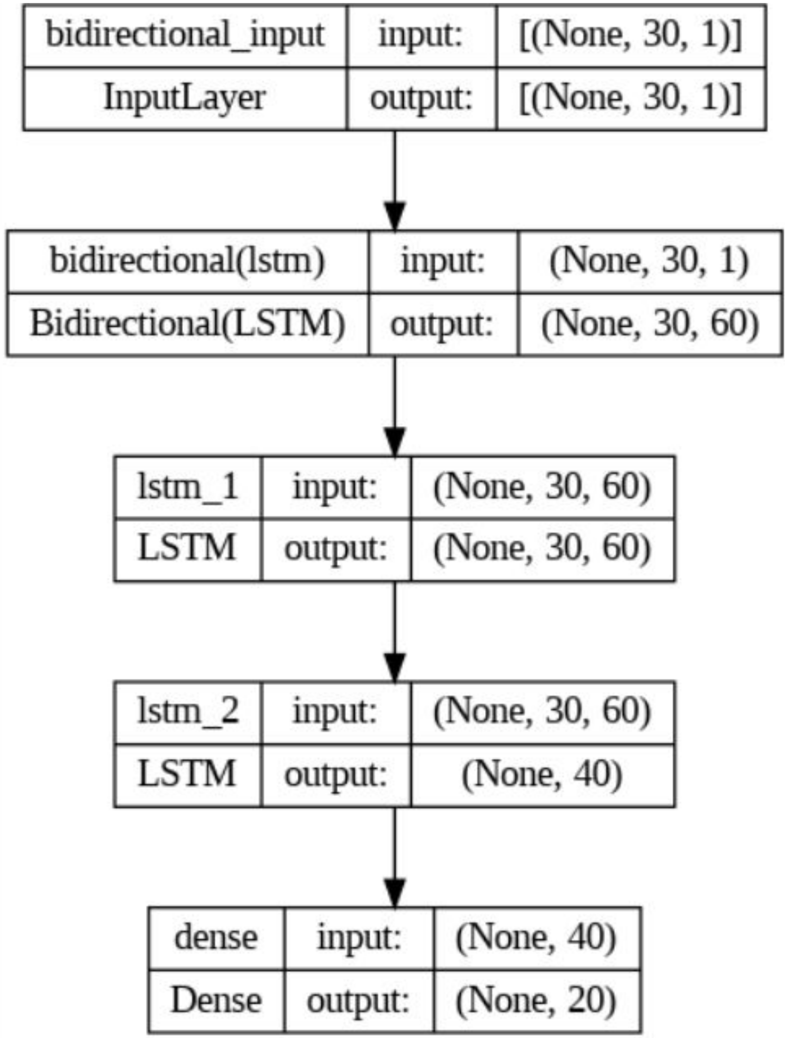
- look_back = 30 timesteps (Input)
- forecast_horizon = 20 timesteps (Target)
- 550 days data results in 481 sequences

```
def create_dataset(data, data2 , look_back=30, forecast_horizon=20):
    X, y = [], []
    for i in range(len(data) - look_back - forecast_horizon + 1):
        X.append(data.iloc[i:(i + look_back)].values)
        y.append(data2.iloc[(i + look_back):(i + look_back + forecast_horizon)].values)
    return np.array(X), np.array(y)
```

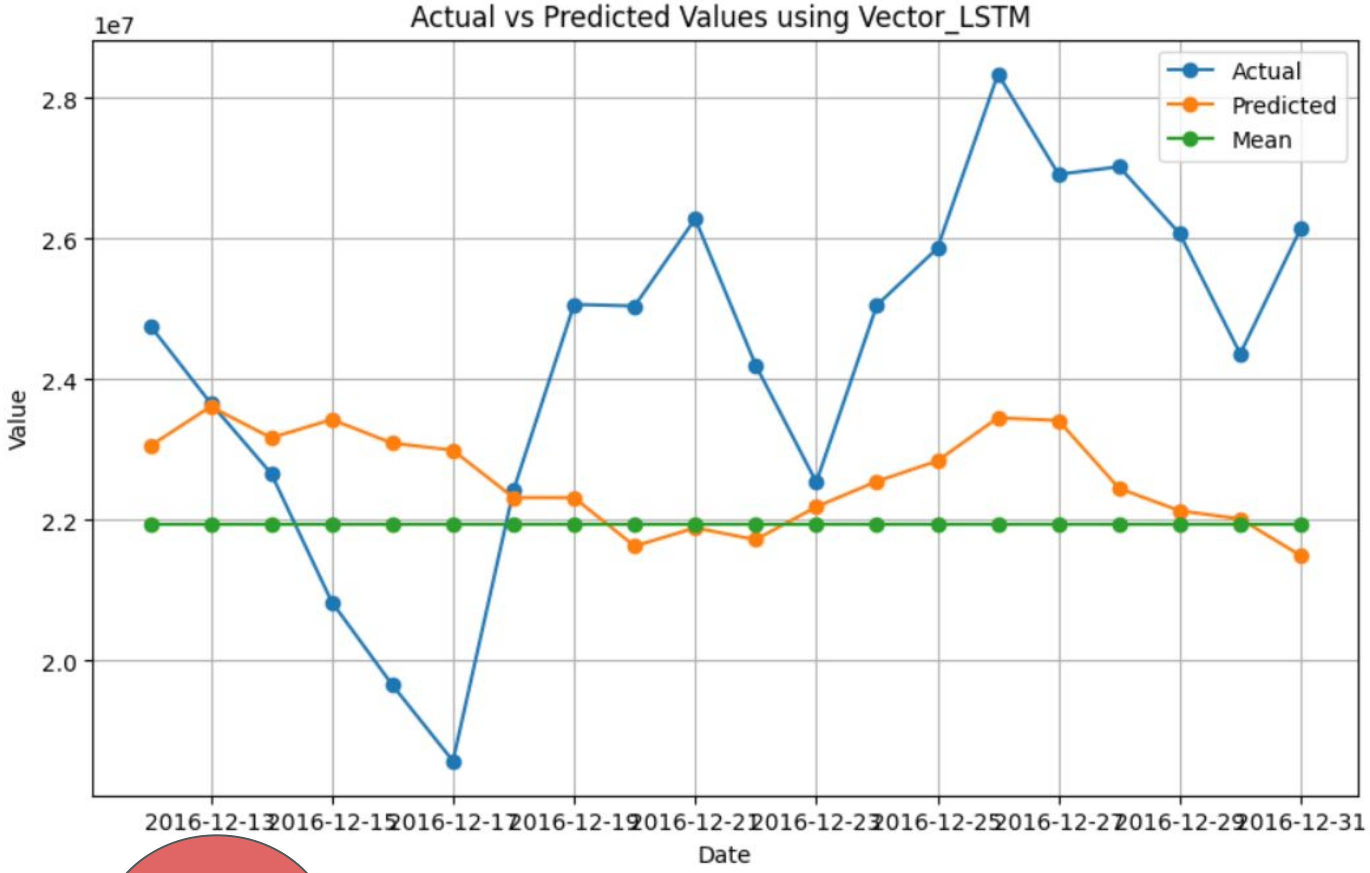
Model evaluation



Vector Model



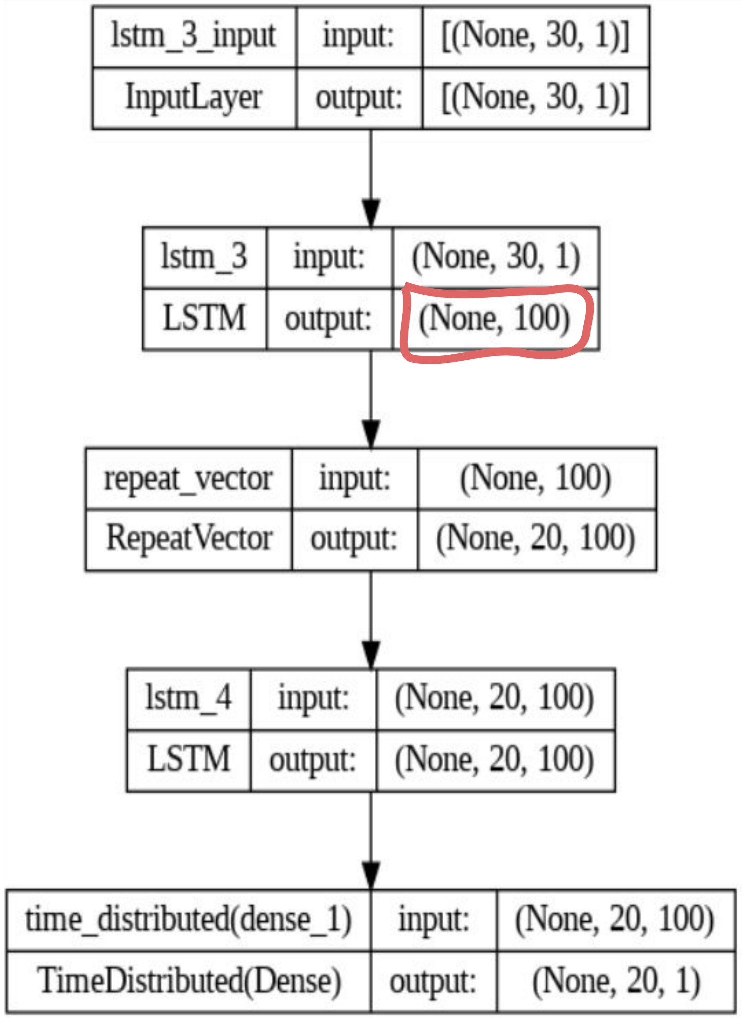
Model Architecture



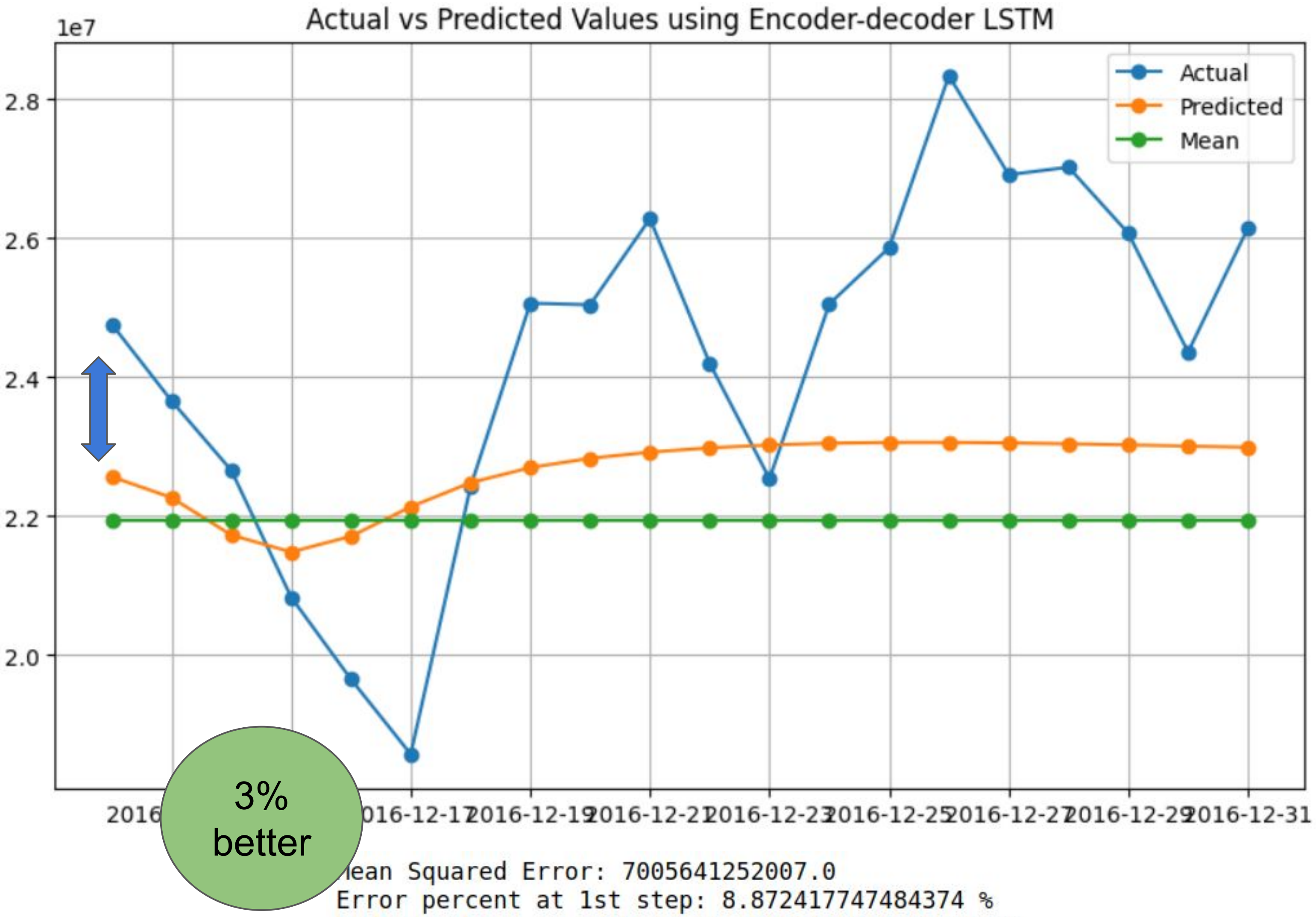
40% worse

```
Mean Squared Error: 10066278997481.6
Error percent at 1st step: 6.910681117051346 %
Error percent at 20th step: 17.802676452434728 %
Max error is 23.797674364514094 %
```

Encoder-Decoder Model

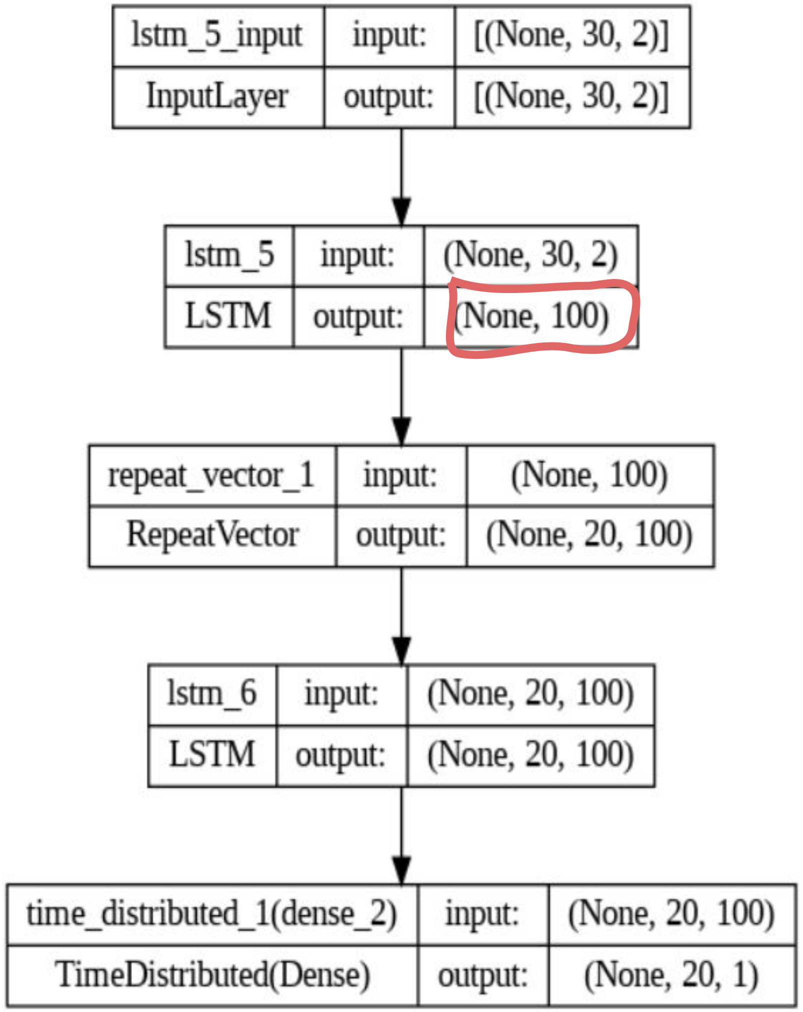


Model Architecture

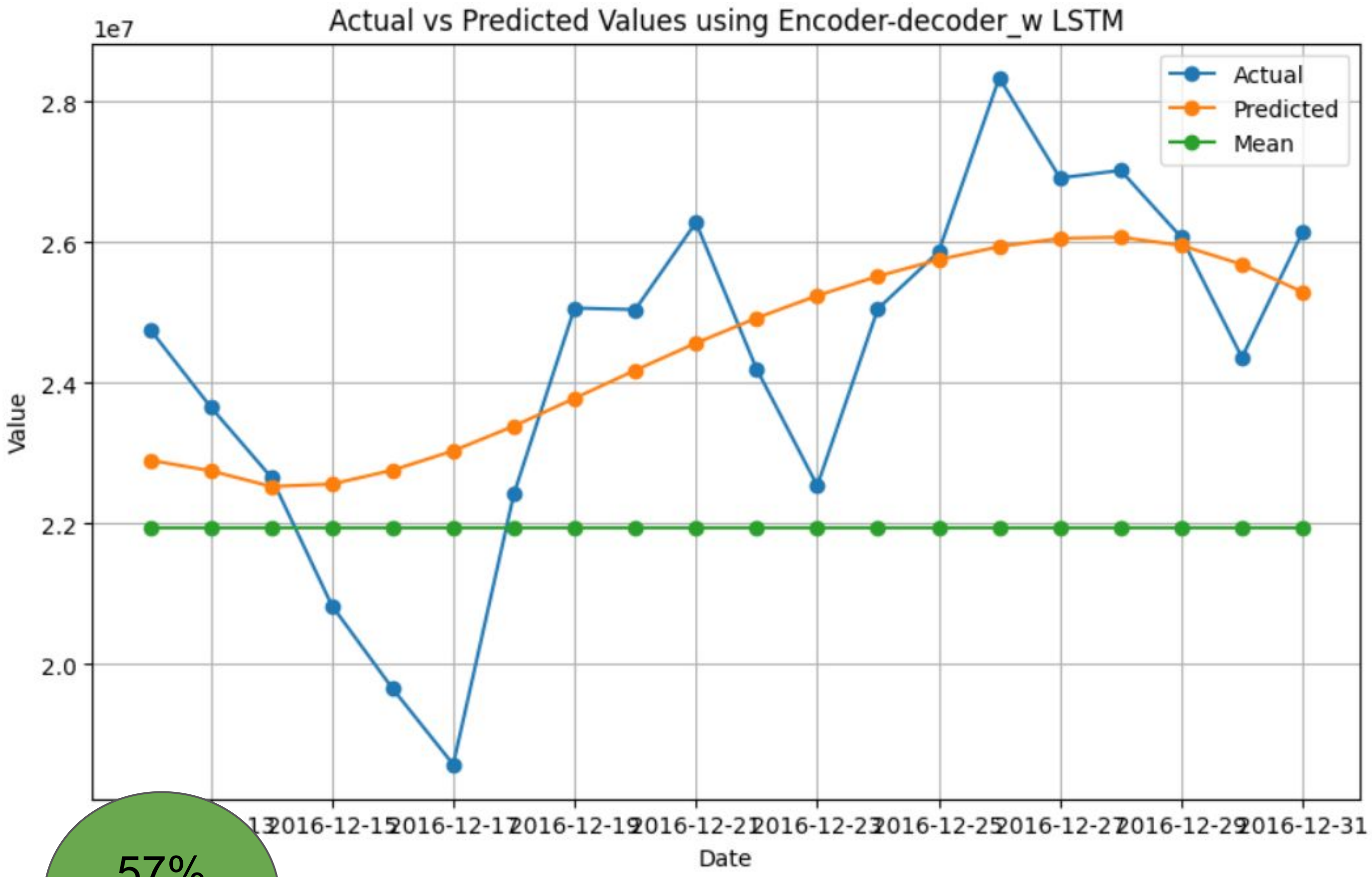



```
ean Squared Error: 7005641252007.0
Error percent at 1st step: 8.872417747484374 %
Error percent at 20th step: 12.074277406027027 %
Max error is 19.180268838879297 %
```

Encoder-Decoder Model with day feature



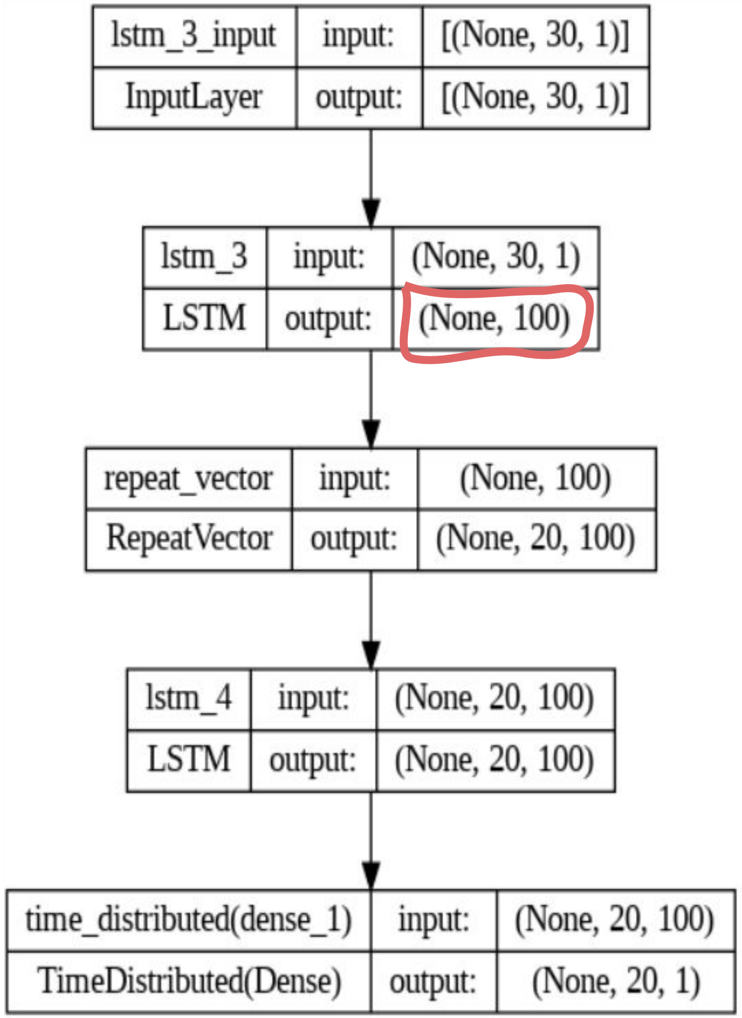
Model Architecture



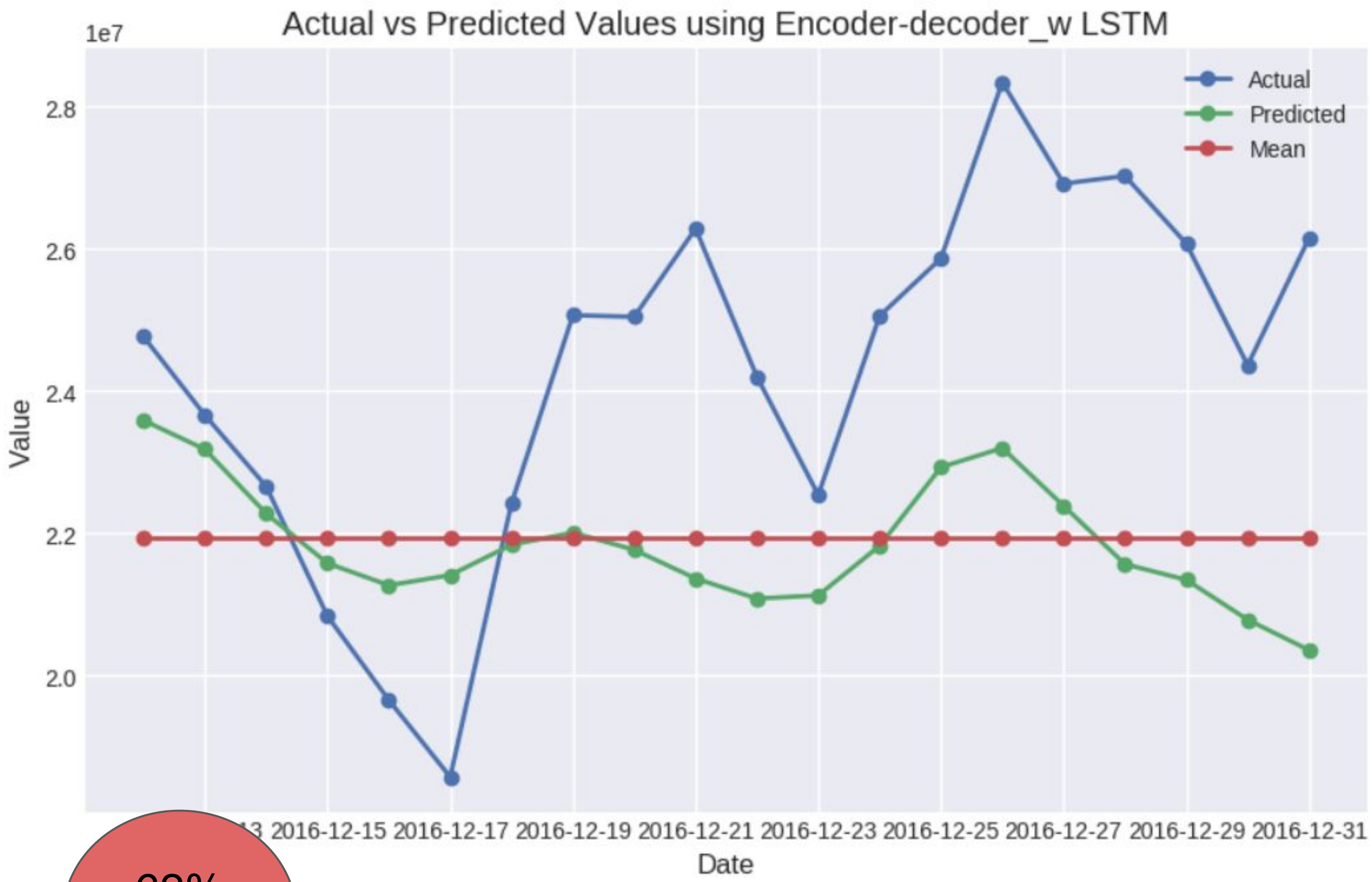
57% better

```
Mean Squared Error: 3055056713356.4
Error percent at 1st step: 7.519503173808409 %
Error percent at 20th step: 3.265900662114898 %
Max error is 24.025385839957824 %
```

Encoder-Decoder Model | <u>MAE loss</u>



Model Architecture



62% worse

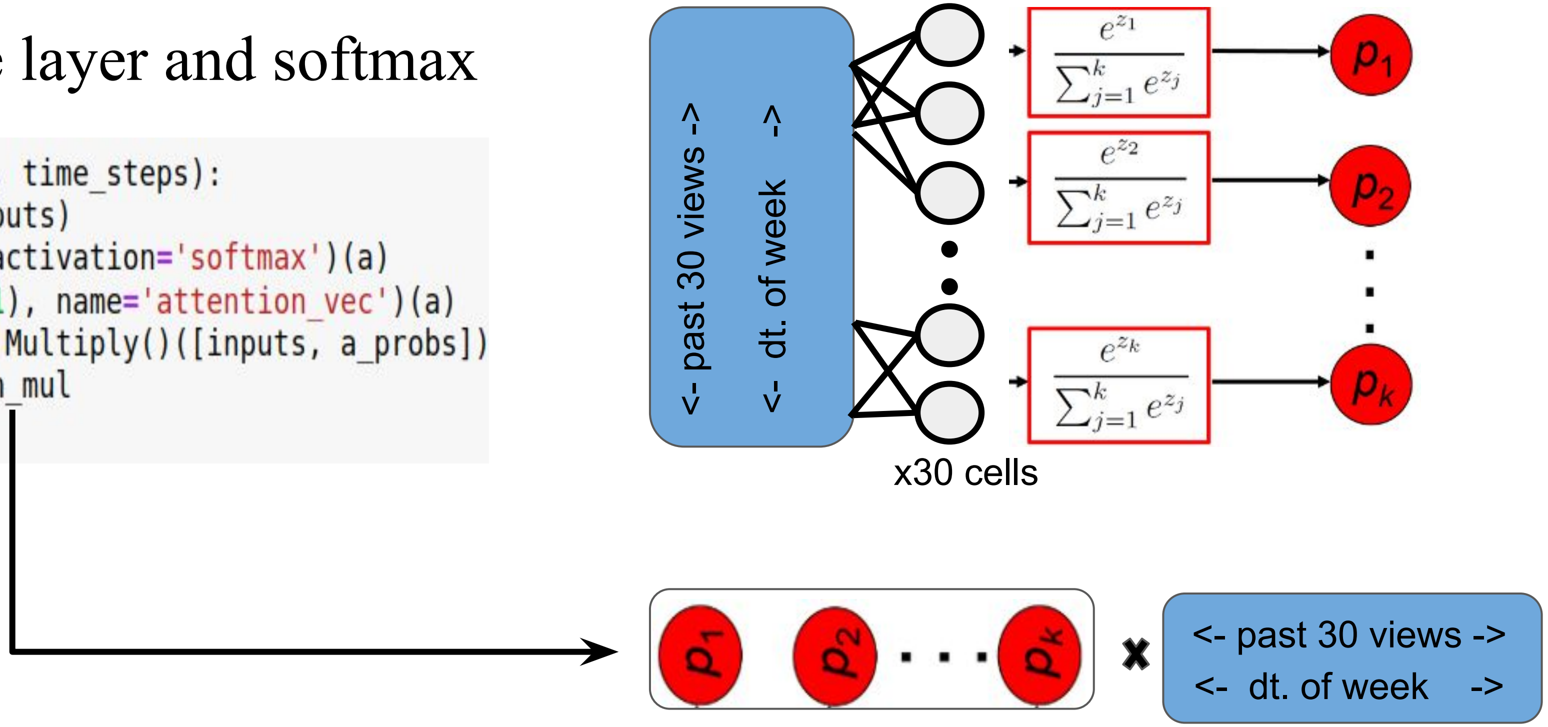
```
an Squared Error: 11696356076578.8
rror percent at 1st step: 4.755382552937186 %
Error percent at 20th step: 22.14985043713962 %
Max error is 22.14985043713962 %
```

Attention to input

- Simple Attention

Using 1 dense layer and softmax

```
def attention_layer(inputs, time_steps):
    a = Permute((2, 1))(inputs)
    a = Dense(time_steps, activation='softmax')(a)
    a_probs = Permute((2, 1), name='attention_vec')(a)
    output_attention_mul = Multiply()([inputs, a_probs])
    return output_attention_mul
```



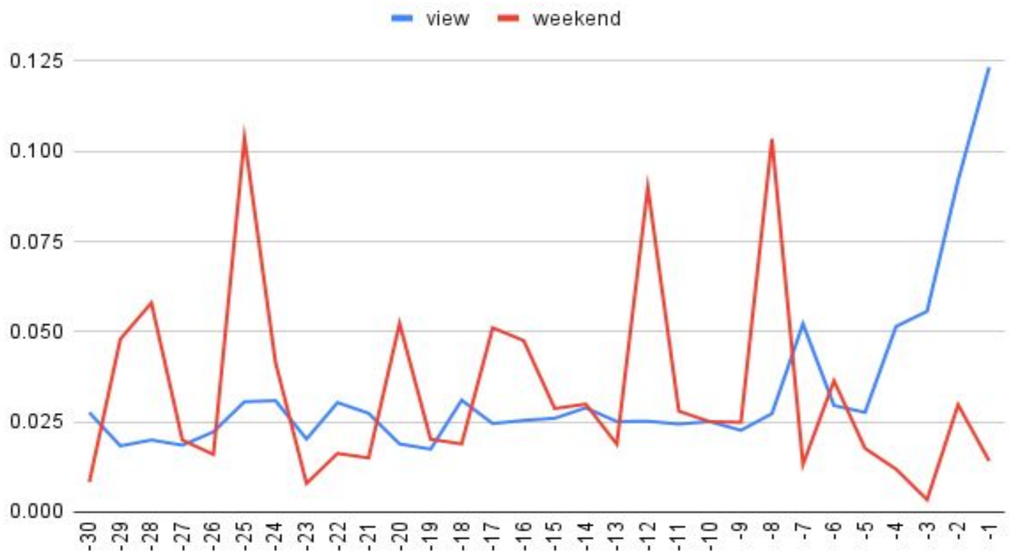
Attention to input

- Simple Attention

Using 1 dense layer and softmax

```
def attention_layer(inputs, time_steps):
    a = Permute((2, 1))(inputs)
    a = Dense(time_steps, activation='softmax')(a)
    a_probs = Permute((2, 1), name='attention_vec')(a)
    output_attention_mul = Multiply()([inputs, a_probs])
    return output_attention_mul
```

Overfitting



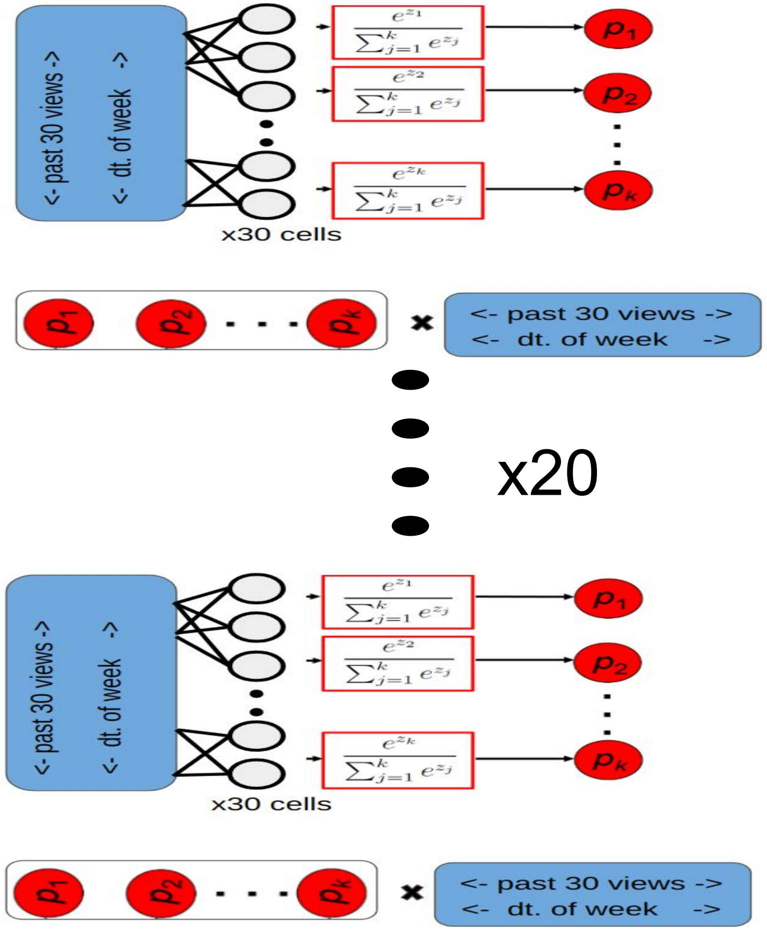
Attention_weights

Attention to input

- Simplified Multi Headed Attention

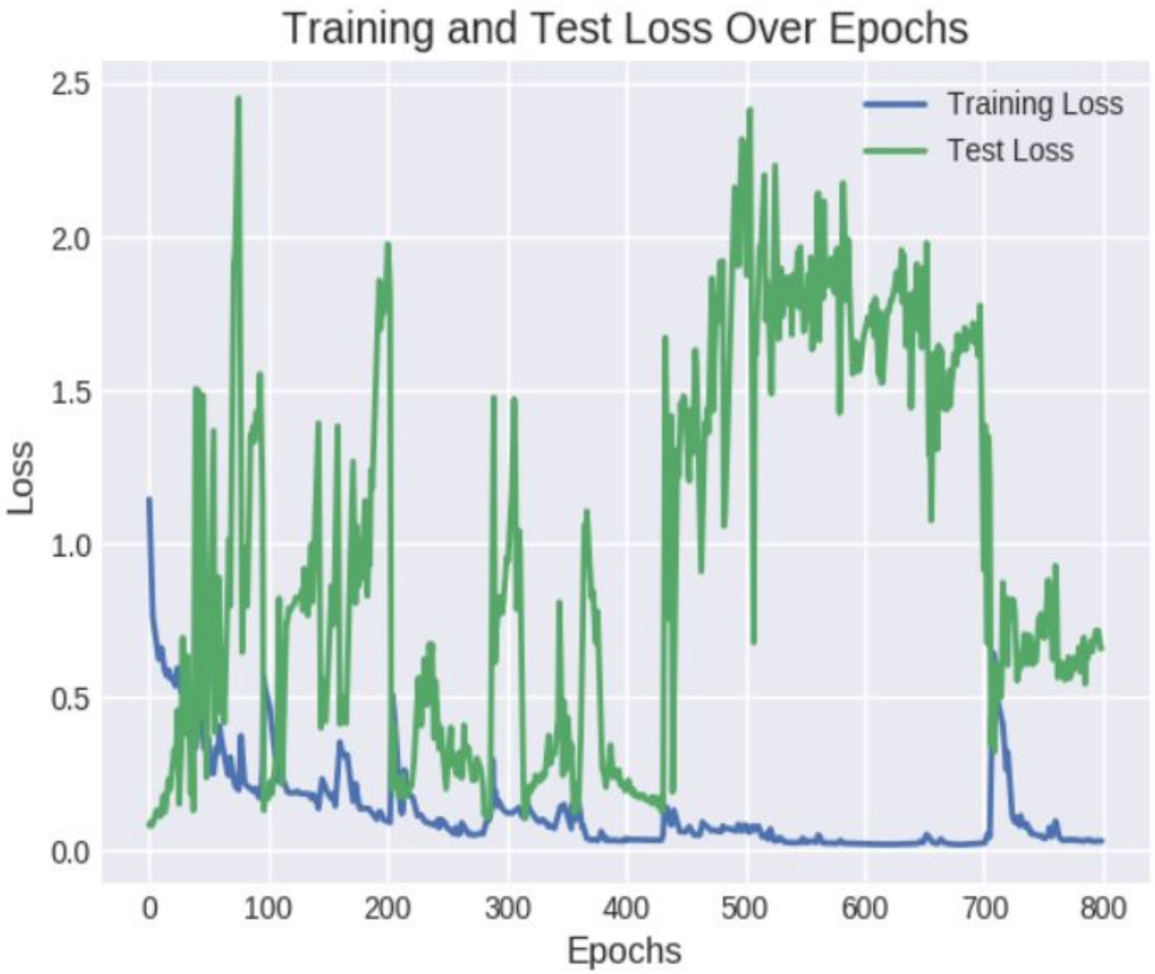
  Creating separate attention vector for each timestep

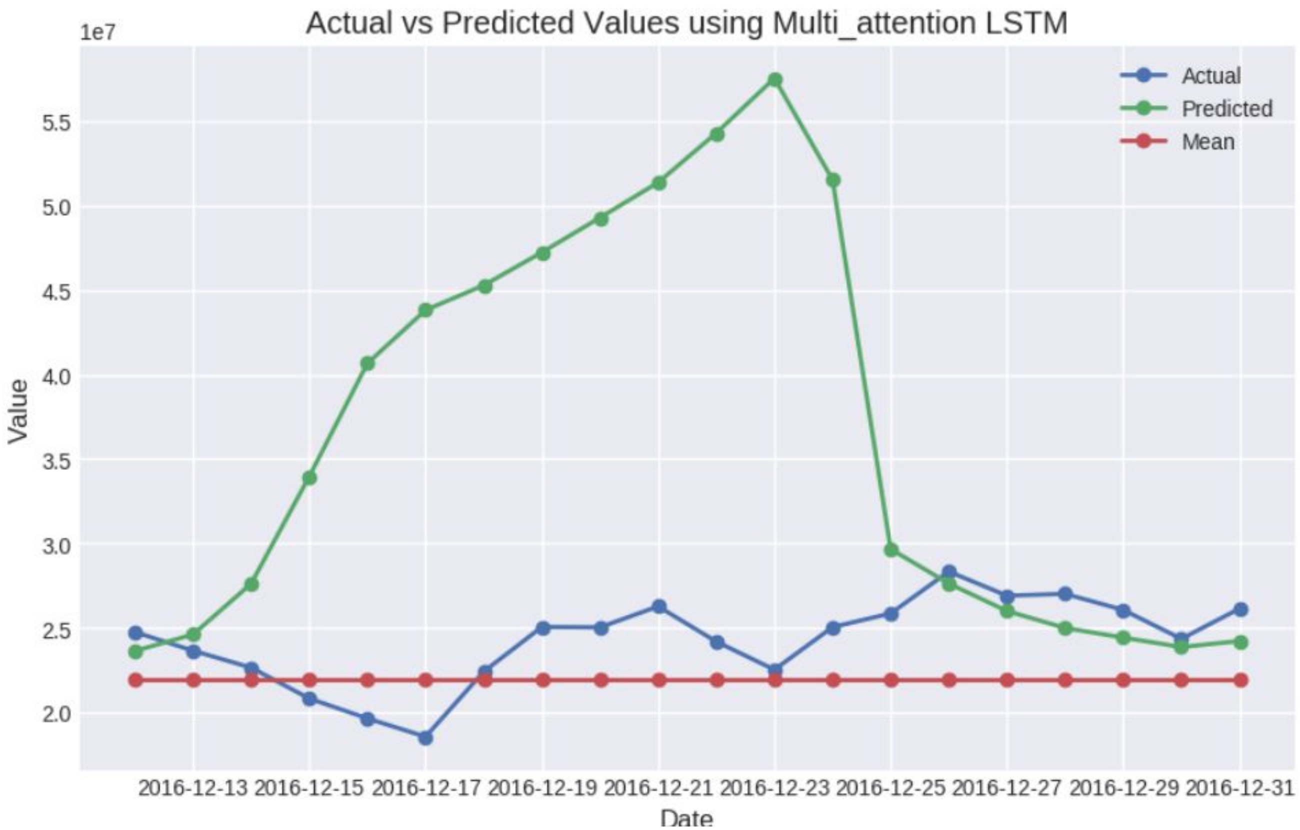
```python
def multi_head_attention_layer(inputs, time_steps, num_heads, layer_name_prefix):
    # inputs.shape = (batch_size, time_steps, input_dim)
    attention_heads = []
    for i in range(num_heads):
        # Create unique names for layers in each head
        dense_layer_name = f"{layer_name_prefix}_dense_head_{i}"
        permute_layer_name = f"{layer_name_prefix}_permute_head_{i}"
        multiply_layer_name = f"{layer_name_prefix}_multiply_head_{i}"

        # Attention mechanism for each head
        a = Permute((2, 1))(inputs)
        a = Dense(time_steps, activation='softmax', name=dense_layer_name)(a)
        a_probs = Permute((2, 1), name=permute_layer_name)(a)
        attention_head = Multiply(name=multiply_layer_name)([inputs, a_probs])
        attention_heads.append(attention_head)

    # Concatenate all heads' outputs
    output_attention_mul = Concatenate(name=f"{layer_name_prefix}_concatenate")(attention_heads)
    return output_attention_mul
```



Attention to input

- Simple Multi Headed Attention



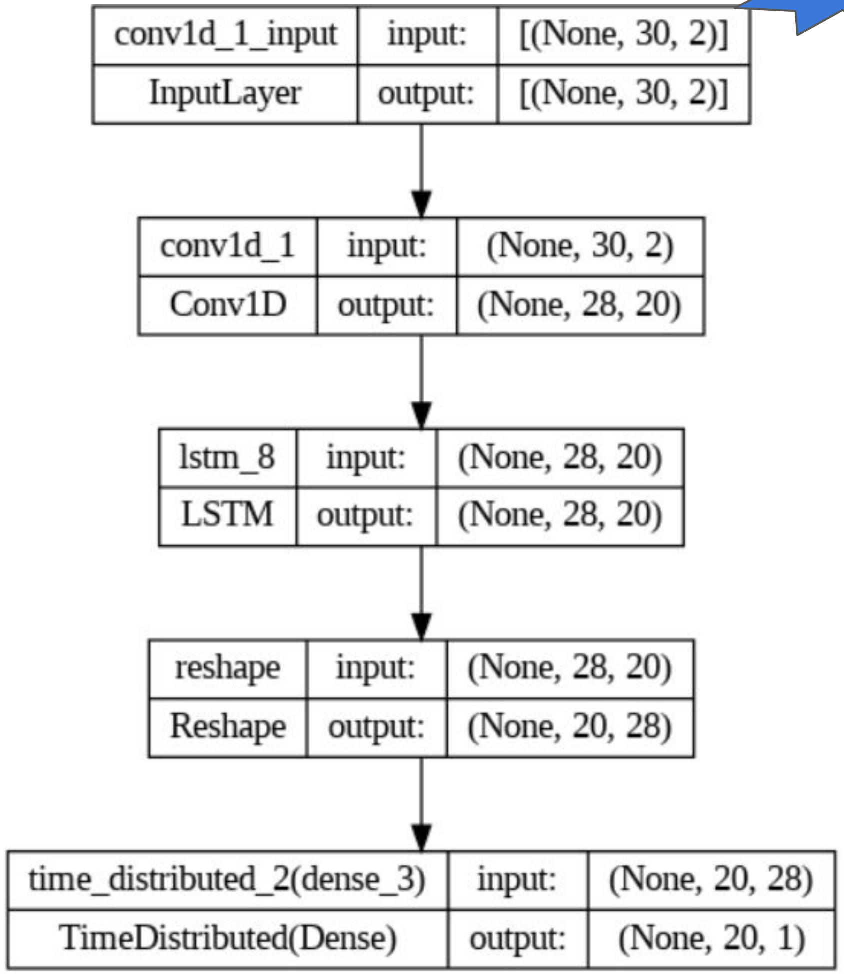


Overfitting

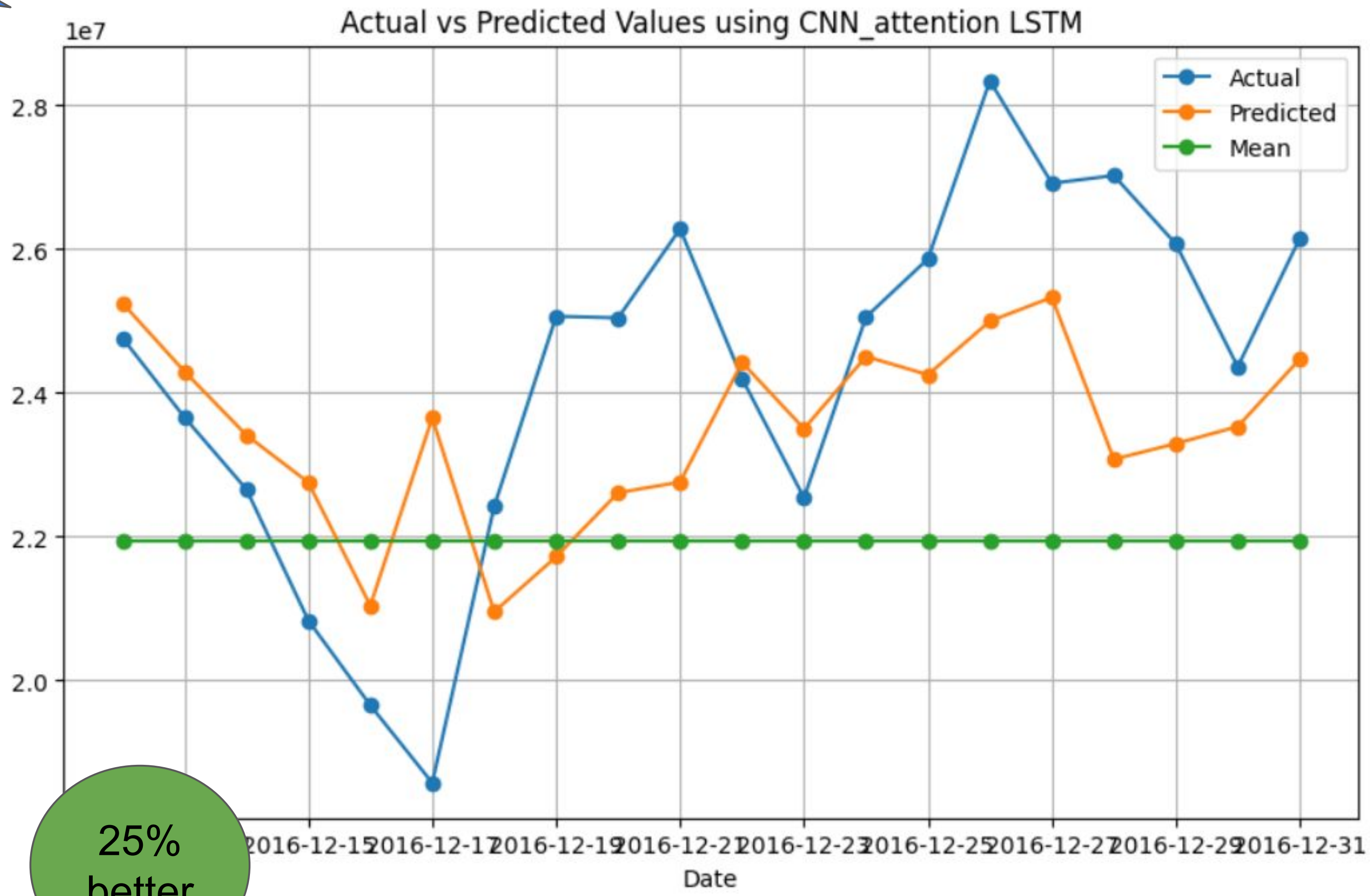
Attention to input

- Using CNN layers

Pattern



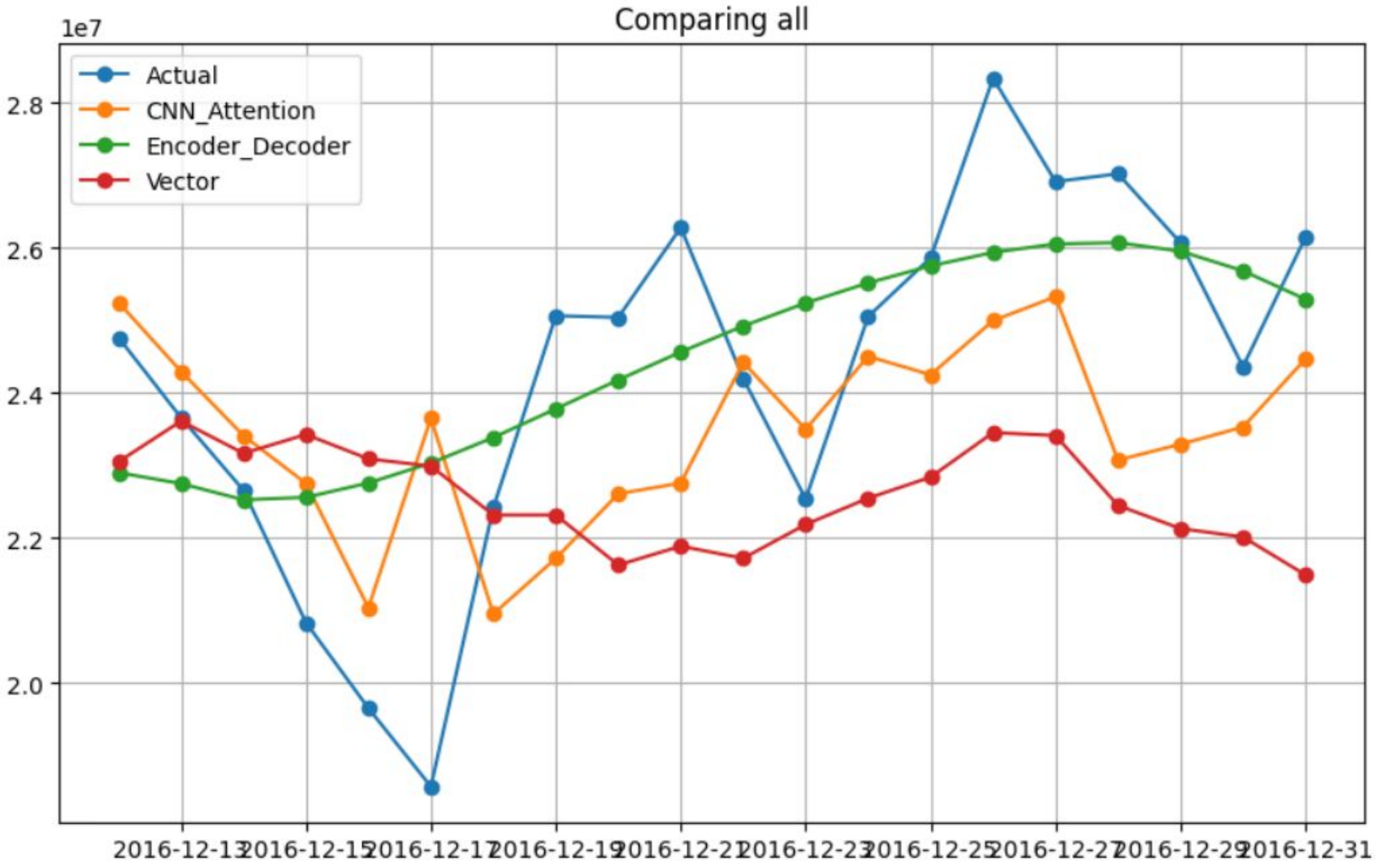
Model Architecture



25% better

```
Mean Squared Error: 5436690644659.6
Error percent at 1st step: -1.9609290455962398 %
Error percent at 20th step: 6.426193347951819 %
Max error is 27.350681917639236 %
```

Comparing all DL approaches



Model	ARIMA	FB-Prophet	LSTM Vector	Encoder-Decoder	CNN_Attention
Performance (1/MSE)	1	0.31	0.72	2.35	1.32

Thank you !