final-project

February 18, 2023

1 Predicting Web Traffic for Wikipedia Articles

In this notebook, we will explore the problem of forecasting future values of multiple time series using the data from the Kaggle competition Web Traffic Time Series Forecasting. The goal of this competition is to test state-of-the-art methods on the problem of predicting future web traffic for approximately 145,000 Wikipedia articles.

The data consists of daily page views for each article from July 1st, 2015 to December 31st, 2016. The page views are split into desktop and mobile traffic. The articles are also grouped by language and project (e.g. en.wikipedia.org).

We will use various methods to analyze and visualize the data, such as univariate and multivariate models, hierarchical time series modeling, data augmentation, anomaly and outlier detection and cleaning, missing value imputation, etc. We will also evaluate our models using appropriate metrics and compare them with the baseline methods provided by the competition organizers.

In this notebook we will:

- Load and explore the data
- Perform some feature engineering
- Build some baseline models using traditional forecasting methods
- Evaluate our models using different metrics
- Submit our predictions to Kaggle

1.1 Load Packages

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import re
import pathlib
import janitor
import janitor.timeseries
import tensorflow as tf
import sklearn.preprocessing

seed = 42
```

```
2023-02-18 02:42:47.278728: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2023-02-18 02:42:47.352638: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot
open shared object file: No such file or directory
2023-02-18 02:42:47.352651: I
tensorflow/compiler/xla/stream_executor/cuda/cudart_stub.cc:29] Ignore above
cudart dlerror if you do not have a GPU set up on your machine.
2023-02-18 02:42:47.698435: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libnvinfer.so.7'; dlerror: libnvinfer.so.7: cannot
open shared object file: No such file or directory
2023-02-18 02:42:47.698486: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libnvinfer_plugin.so.7'; dlerror:
libnvinfer_plugin.so.7: cannot open shared object file: No such file or
directory
2023-02-18 02:42:47.698492: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Cannot
dlopen some TensorRT libraries. If you would like to use Nvidia GPU with
TensorRT, please make sure the missing libraries mentioned above are installed
properly.
```

1.2 Load Data

The data is provided in two files: `train_1.csv` and `key_1.csv`. The `train_1.csv` file contains the page views for each article and the `key_1.csv` file contains the page names and the dates for which we need to make predictions.

The `train_1.csv` file contains 145,063 rows and 551 columns. The first column contains the page names and the remaining 550 columns contain the page views for each day. The `key_1.csv` file contains 39,546 rows and 2 columns. The first column contains the page names and the second column contains the dates for which we need to make predictions.

[ ]:
```python
root = pathlib.Path('~/datasets/web-traffic-time-series-forecasting').
 ↪expanduser()
```

[ ]:
```python
train = pd.read_csv(str(root / 'train_1.csv')).sample(frac=0.01,␣
 ↪random_state=seed)
train.head()
```

[ ]:
```
                                                   Page  2015-07-01  \
83529  Phabricator/Project_management_www.mediawiki.o…         6.0
```

```
70433  Now_You_See_Me_es.wikipedia.org_desktop_all-ag…       242.0
84729  Zürich_Hackathon_2014_www.mediawiki.org_all-ac…         3.0
7969        Érythrée_fr.wikipedia.org_desktop_all-agents       672.0
92077   Metallica_es.wikipedia.org_all-access_all-agents      1534.0

       2015-07-02  2015-07-03  2015-07-04  2015-07-05  2015-07-06  2015-07-07  \
83529         6.0         4.0         6.0         8.0         6.0         4.0
70433       271.0       309.0       227.0       321.0       311.0       242.0
84729        19.0        19.0        30.0        21.0        24.0        17.0
7969        513.0       774.0      1164.0       546.0       755.0       555.0
92077      1644.0      1704.0      1569.0      1534.0      1577.0      1608.0

       2015-07-08  2015-07-09  …  2016-12-22  2016-12-23  2016-12-24  \
83529         0.0         2.0  …         6.0         6.0        11.0
70433       236.0       243.0  …       231.0       222.0       193.0
84729       178.0        40.0  …         6.0         7.0         4.0
7969        494.0      4801.0  …       308.0       294.0       358.0
92077      1731.0      1919.0  …      2367.0      2259.0      2229.0

       2016-12-25  2016-12-26  2016-12-27  2016-12-28  2016-12-29  2016-12-30  \
83529         4.0         6.0         5.0         7.0         6.0         6.0
70433       229.0       334.0       316.0       324.0       268.0       201.0
84729         8.0         2.0         4.0         9.0         4.0        11.0
7969        204.0       323.0       438.0       345.0       299.0       306.0
92077      2070.0      2774.0      2552.0      2524.0      2358.0      2291.0

       2016-12-31
83529         9.0
70433       190.0
84729        12.0
7969        211.0
92077      2153.0

[5 rows x 551 columns]
```

[ ]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1451 entries, 83529 to 89204
Columns: 551 entries, Page to 2016-12-31
dtypes: float64(550), object(1)
memory usage: 6.1+ MB
```

As we can see, the data is not in a tidy format. We need to reshape it to a tidy format. We will use the melt function from pandas to do this. We will also use the regex to extract the date from the column name. We will also use the to_datetime function to convert the date column to a datetime object.

[ ]:
```
# tidy up the data

df = train.set_index('Page').stack().reset_index()
df.columns = ['Page', 'Date', 'Visits']
df['Date'] = pd.to_datetime(df['Date'])
df = df.clean_names().sort_timestamps_monotonically(direction="increasing")

# combine page and date columns and get id for each unique combination from the␣
↪key dataframe
# create a new column for id

df['id'] = df['page'] + '_' + df['date'].astype(str)
```

[ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 735117 entries, 0 to 735116
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   page    735117 non-null  object
 1   date    735117 non-null  datetime64[ns]
 2   visits  735117 non-null  float64
 3   id      735117 non-null  object
dtypes: datetime64[ns](1), float64(1), object(2)
memory usage: 22.4+ MB
```

1.3 Explore Data

In the following sections, we will explore the data and try to understand the patterns in the data.

1.3.1 How Page Language Affects Traffic

I want to explore how the languages used in Wikipedia pages might influence the traffic data. I'll use a simple regex to find the language code in the URL. Some URLs are not from Wikipedia but from Wikimedia. They don't have a language code, so I'll label them as `missing`. These are mostly images or other media that don't have a specific language.

The Page column contains pages in different languages. We will extract the language from the page name and create a new column called Language. We will also create an access column that contains the access type (mobile or desktop) and agent column that contains the type of user agent (spider, crawler, robot, etc.). In addition, we will create an article column that contains the name of the article.

[ ]:
```
pattern = r'\w\w(?=\.wikipedia)'

# extract language/locale from 'page' column using regex pattern
df['language'] = df['page'].apply(
```

```
    lambda x: re.search(pattern, x).group(0) if re.search(pattern, x) else␣
 ↪'missing'
)


# define regex patterns
access_pattern = r'(?<=\.org_)(\w+-?\w+)(?=_)'
agent_pattern = r'(?<=_)[a-zA-Z]+(-[a-zA-Z]+)*$'
article_pattern = r'(?<=\d_\w\w\.)(.*)'

# extract access type from 'page' column using regex pattern
df['access'] = df['page'].apply(
    lambda x: re.search(access_pattern, x).group(1) if re.
 ↪search(access_pattern, x) else None
)

# extract user agent from 'page' column using regex pattern
df['agent'] = df['page'].apply(
    lambda x: re.search(agent_pattern, x).group(0) if re.search(agent_pattern,␣
 ↪x) else None
)

# # # extract article name from 'page' column using regex pattern
df['article'] = (
    df['page']
    .str.extract(r'(.+?)\.', expand=False)
    .apply(lambda x: x.rsplit('_', 1)[0] if '_' in x else None)
)

df['rolling_mean_visits'] = df.groupby('page')['visits'].transform(lambda x: x.
 ↪rolling(7, 1).mean())

df.head()
```

[ ]:

```
                                              page        date  visits  \
0  Phabricator/Project_management_www.mediawiki.o… 2015-07-01     6.0
1  Phabricator/Project_management_www.mediawiki.o… 2015-07-02     6.0
2  Phabricator/Project_management_www.mediawiki.o… 2015-07-03     4.0
3  Phabricator/Project_management_www.mediawiki.o… 2015-07-04     6.0
4  Phabricator/Project_management_www.mediawiki.o… 2015-07-05     8.0

                                                id language      access  \
0  Phabricator/Project_management_www.mediawiki.o…  missing  all-access
1  Phabricator/Project_management_www.mediawiki.o…  missing  all-access
2  Phabricator/Project_management_www.mediawiki.o…  missing  all-access
3  Phabricator/Project_management_www.mediawiki.o…  missing  all-access
4  Phabricator/Project_management_www.mediawiki.o…  missing  all-access
```

```
    agent                            article  rolling_mean_visits
0  spider  Phabricator/Project_management             6.000000
1  spider  Phabricator/Project_management             6.000000
2  spider  Phabricator/Project_management             5.333333
3  spider  Phabricator/Project_management             5.500000
4  spider  Phabricator/Project_management             6.000000
```

[ ]:
```
df.language.value_counts()
```

[ ]:
```
en         107267
de         101538
ja          96437
fr          96189
missing     87753
zh          85451
es          80862
ru          79620
Name: language, dtype: int64
```

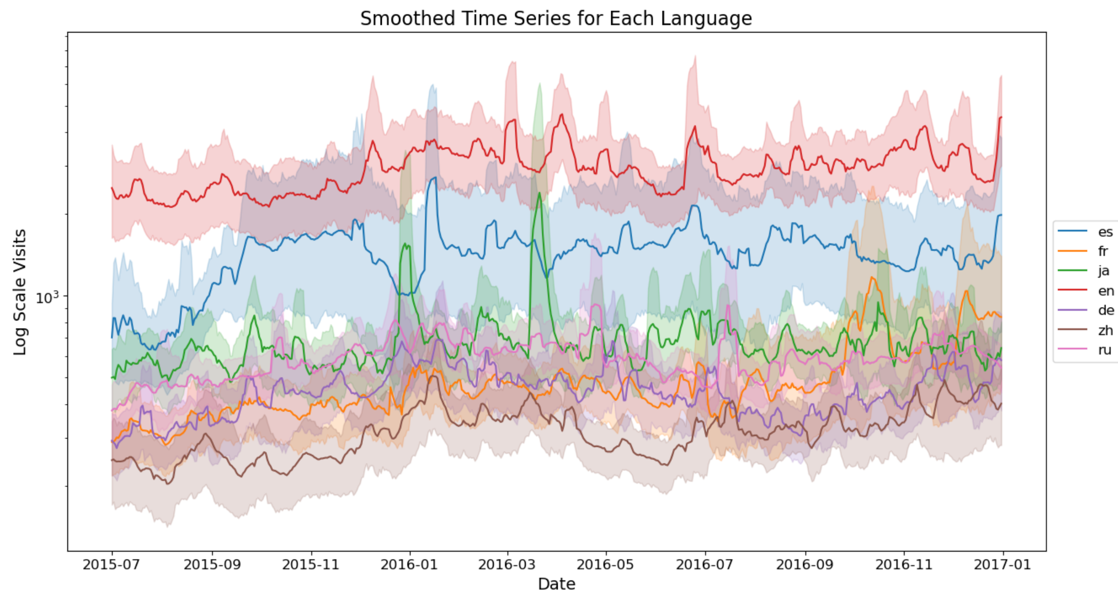
As we can see, there are 7 different languages in the dataset.

- English (en)
- Japanese (ja)
- Deutsch (de)
- French (fr)
- Chinese (zh)
- Spanish (es)
- Russian (ru)

We can also see that there are some missing values in the language column. We will deal with these missing values later.

[ ]:
```
# plot time series for each language over time, make y axis log scale

fig, ax = plt.subplots(figsize=(15, 8))
sns.lineplot(
    x='date', y='rolling_mean_visits', hue='language', data=df[df.language !=
 ↪'missing'], ax=ax
)

# Set aesthetics
ax.set_title('Smoothed Time Series for Each Language', fontsize=16)
ax.set_xlabel('Date', fontsize=14)
ax.set_ylabel('Log Scale Visits', fontsize=14)
ax.set_yscale('log')
ax.tick_params(labelsize=12)
```

```
# Set legend outside the plot
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5), fontsize=12)

plt.show()
```




```
[ ]: # # plot time series for each language over time, make y axis log scale in␣
     ↪separate plots
     # # Set figure size and font size
     # plt.rcParams['figure.figsize'] = [8, 24]
     # plt.rcParams['font.size'] = 12

     # # Create a list of languages to loop over
     # languages = ['en', 'ja', 'de', 'fr', 'zh', 'ru']

     # # Create a one-column grid of subplots
     # fig, axes = plt.subplots(len(languages), 1, sharex=False)

     # # Loop over the languages and create a line plot for each one
     # for i, language in enumerate(languages):
     #     ax = axes[i]
     #     sns.lineplot(x='date', y='rolling_mean_visits', data=df[df.language ==␣
     ↪language], ax=ax)
     #     ax.set_title(f'Smoothed Time Series for {language.capitalize()}',␣
     ↪fontsize=16)
     #     ax.set_xlabel('Date', fontsize=14)
     #     ax.set_ylabel('Log Scale Visits', fontsize=14)
     #     ax.set_yscale('log')
```

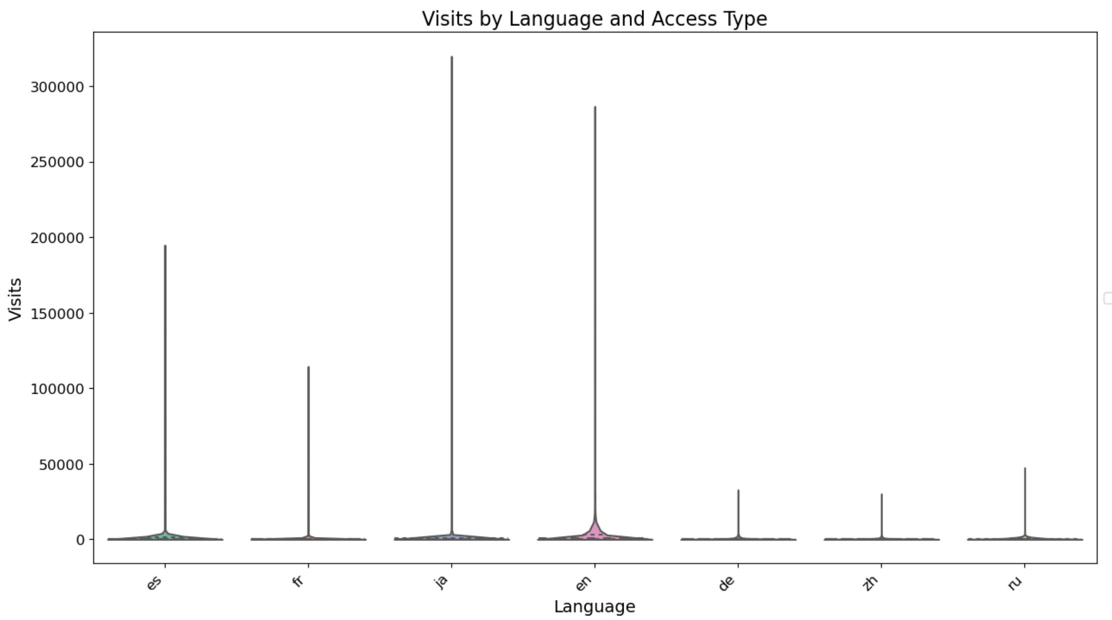
```
#     ax.tick_params(labelsize=12)
#     ax.legend(fontsize=12)
#     ax.tick_params(axis='x', rotation=45)

# plt.tight_layout()

# plt.show()
```

Let's explore teh distribution of web traffic by language, access type, and user agent. We are going to use seaborn to create histograms, boxplots, and violinplots to see how the distribution of visits varies across these different categories.

```
[ ]: fig, ax = plt.subplots(figsize=(15, 8))

# Use "Set2" color palette
colors = sns.color_palette('Set2')

# Increase the size of the violin plots
sns.violinplot(
    x='language',
    y='rolling_mean_visits',
    data=df[df.language != 'missing'],
    ax=ax,
    palette=colors,
    scale='width',
    inner='quartile',
    cut=0,
)
# sns.violinplot(x='language', y='visits', data=df[df.language !='missing'],
 ↪ax=ax, scale='width')

# Set aesthetics
ax.set_title('Visits by Language and Access Type', fontsize=16)
ax.set_xlabel('Language', fontsize=14)
ax.set_ylabel('Visits', fontsize=14)
ax.tick_params(labelsize=12)

# Remove grid lines
ax.grid(False)

# Rotate x-axis labels
plt.setp(ax.get_xticklabels(), rotation=45, ha='right')


# Set legend outside the plot
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5), fontsize=12)
```

```
plt.show()
```

```
No handles with labels found to put in legend.
```



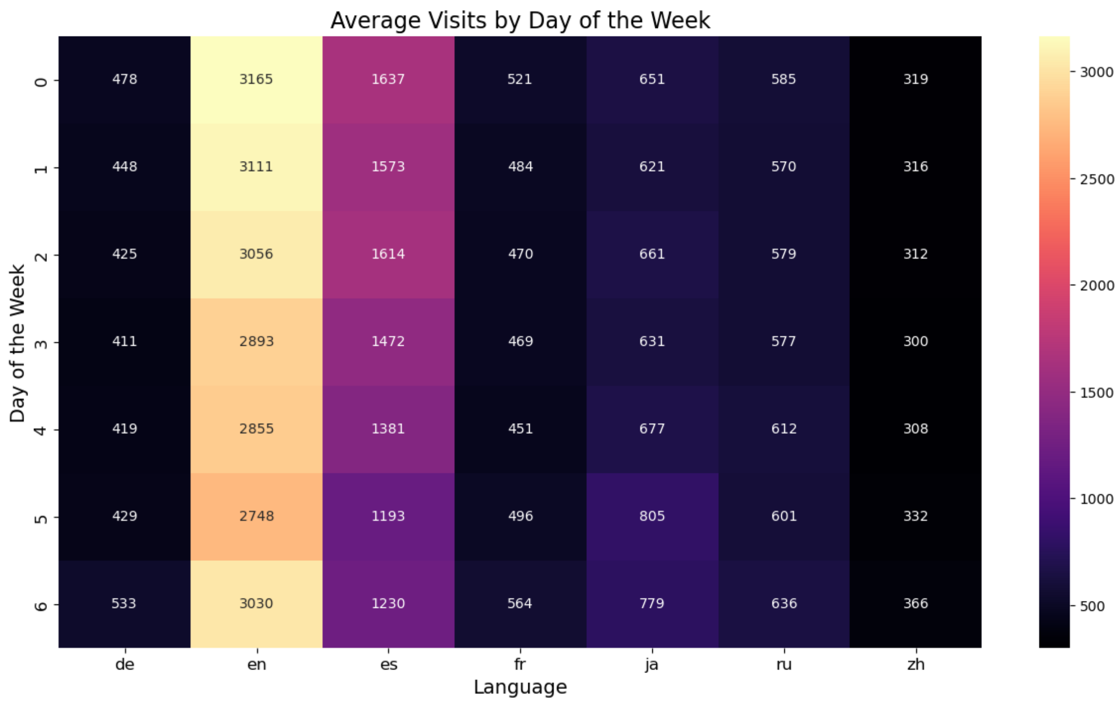
1.3.2 How web traffic varies by day of the week, month, and year

We are going to create plots or bar charts to see if there are any regular patterns or seasonality in the data.

```
[ ]: # plot how the web traffic varies by day of the week, month or year

     # create a new column for day of the week
     df['day_of_week'] = df['date'].dt.dayofweek

     # create a new column for month
     df['month'] = df['date'].dt.month

     # create a new column for day of the year
     df['day_of_year'] = df['date'].dt.dayofyear
```

```
[ ]: # heatmap of visits by day of the week and language

     # create a pivot table
     day_of_week = df[df.language != 'missing'].pivot_table(
         index='day_of_week', columns='language', values='visits', aggfunc='mean'
     )
```

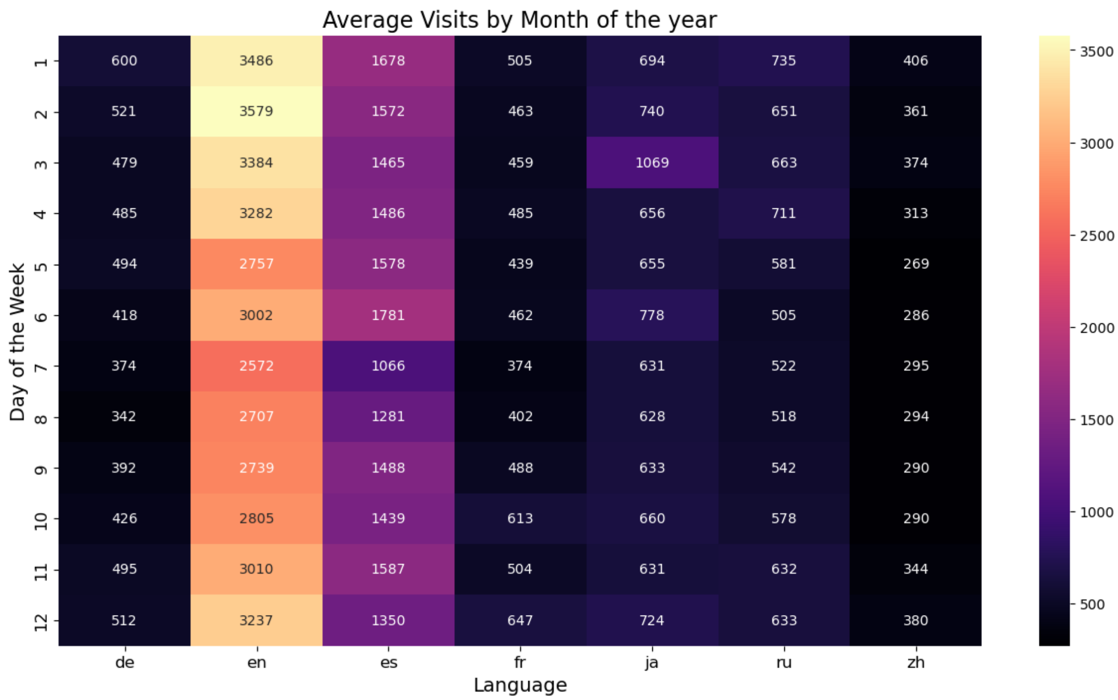
```
# create a heatmap
fig, ax = plt.subplots(figsize=(15, 8))
sns.heatmap(day_of_week, annot=True, fmt='.0f', cmap='magma', ax=ax)

# Set aesthetics
ax.set_title('Average Visits by Day of the Week', fontsize=16)
ax.set_xlabel('Language', fontsize=14)
ax.set_ylabel('Day of the Week', fontsize=14)
ax.tick_params(labelsize=12)

plt.show()
```



The plot above shows the average number of visits per day of the week. We can see that the number of visits is higher on Sunday and Monday for english pages. French and Chinese pages seem to have a steady or flat trend throughout the week.

[ ]:
```
# heatmap of visits by day of the week and language

# create a pivot table
day_of_week = df[df.language != 'missing'].pivot_table(
    index='month', columns='language', values='visits', aggfunc='mean'
)

# create a heatmap
fig, ax = plt.subplots(figsize=(15, 8))
```

```
sns.heatmap(day_of_week, annot=True, fmt='.0f', cmap='magma', ax=ax)

# Set aesthetics
ax.set_title('Average Visits by Month of the year', fontsize=16)
ax.set_xlabel('Language', fontsize=14)
ax.set_ylabel('Day of the Week', fontsize=14)
ax.tick_params(labelsize=12)

plt.show()
```

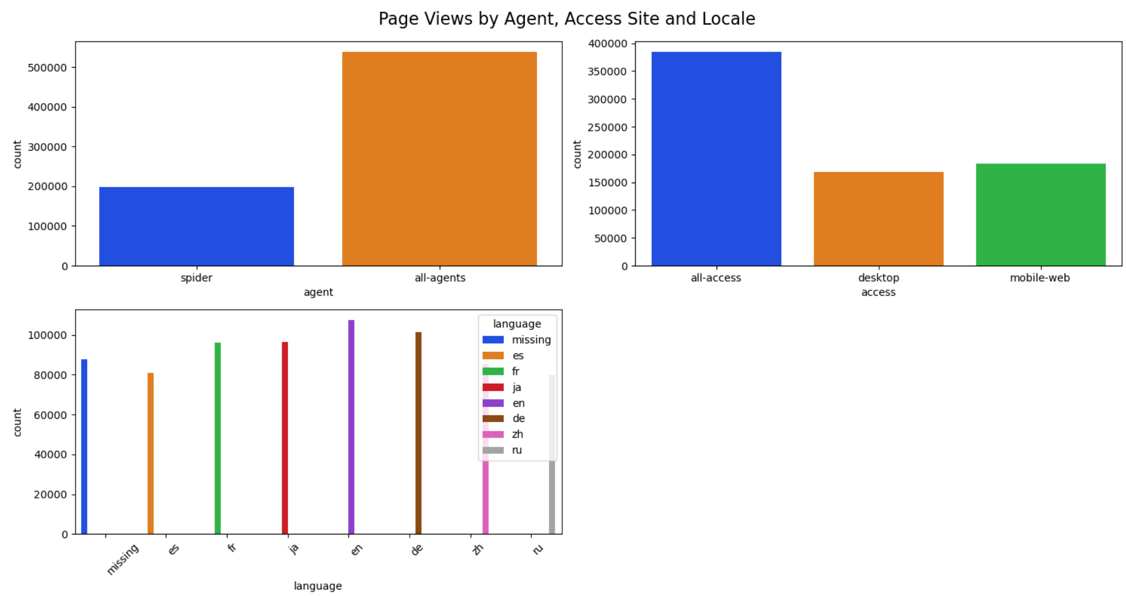


1.3.3 How web traffic varies by access type

In the plot below, we look at the distribution of visits by access type.

[ ]:
```
# layout the plots
fig, ax = plt.subplots(2, 2, figsize=(15, 8))

# create plots
p1 = sns.countplot(data=df, x='agent', color='red', ax=ax[0, 0],␣
↪palette='bright')
p2 = sns.countplot(data=df, x='access', color='red', ax=ax[0, 1],␣
↪palette='bright')
p3 = sns.countplot(data=df, x='language', hue='language', ax=ax[1, 0],␣
↪palette='bright')
```

```
# make sure p3 x-axis labels align with the bars
p3.set_xticklabels(p3.get_xticklabels(), rotation=45,␣
↪horizontalalignment='left')

ax[1, 1].axis('off')

# set main title
fig.suptitle('Page Views by Agent, Access Site and Locale', fontsize=16)

fig.tight_layout()
```



As we can see from the plots above, the mobile-web access type is slightly more popular than the desktop access type.

1.3.4 The most popular articles

Let's look at the most popular articles in the dataset. We will use the groupby function to group the data by page and sum the visits. We will then sort the data by visits in descending order.

[ ]:
```
# find the most visited articles for each language

janitor.groupby_topk(
    df[df.language != 'missing']
    .groupby(['language', 'article'])['visits']
    .sum()
    .sort_values(ascending=False)
    .reset_index(),
```

```
    'language',
    'visits',
    1,
    {'ascending': False},
)
```

[ ]:

```
              language              article      visits
language
de       62         de  Wikipedia:Auskunft   1874285.0
en       1          en              Google  21891412.0
es       0          es    Wikipedia:Portada  31615409.0
fr       11         fr              France   5721186.0
ja       31         ja                  K   3482174.0
ru       104        ru                     1321916.0
zh       126        zh             BIGBANG   1113794.0
```

[ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 735117 entries, 0 to 735116
Data columns (total 12 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   page                 735117 non-null  object
 1   date                 735117 non-null  datetime64[ns]
 2   visits               735117 non-null  float64
 3   id                   735117 non-null  object
 4   language             735117 non-null  object
 5   access               735117 non-null  object
 6   agent                735117 non-null  object
 7   article              721424 non-null  object
 8   rolling_mean_visits  735117 non-null  float64
 9   day_of_week          735117 non-null  int64
 10  month                735117 non-null  int64
 11  day_of_year          735117 non-null  int64
dtypes: datetime64[ns](1), float64(2), int64(3), object(6)
memory usage: 67.3+ MB
```

[ ]:

```
# prepare data for modeling by group by date and compute rolling sum of visits

# data = df.groupby('date')['visits'].rolling(window=7).sum().
 ↪reset_index(name='rolling_sum_visits').
 ↪sort_timestamps_monotonically(direction="increasing").dropna().
 ↪drop('level_1', axis=1).set_index('date').
 ↪rename(columns={'rolling_sum_visits': 'visits'})
```

```
data = (
    df.groupby(['date'])[['visits']]
     .sum()
     .sort_timestamps_monotonically(direction="increasing")
     .rolling(window=7)
     .sum()
     .dropna()
)
data
```

[ ]:
```
                 visits
date
2015-07-07    5884857.0
2015-07-08    5886324.0
2015-07-09    5860379.0
2015-07-10    5896575.0
2015-07-11    5893563.0
…                   …
2016-12-27    9673878.0
2016-12-28   10288875.0
2016-12-29   11277546.0
2016-12-30   12380626.0
2016-12-31   12545284.0

[544 rows x 1 columns]
```

1.4 Modeling

In this section, we will build some models using keras and tensorflow. But first, we need to make sure that the data is in the right format. With time series data, the sequence of the data is important. We need to make sure that the data is in the right order. The code below splits the data into train, validation, and test sets.

[ ]:
```
# split data into train and test sets
cut_off_date = '2016-08-01'
val_cut_off_date = '2016-10-01'
train = data[data.index < cut_off_date]
val = data[(data.index >= cut_off_date) & (data.index < val_cut_off_date)]
test = data[data.index >= val_cut_off_date]


# print the shape of the train and test sets
print(f'Train shape: {train.shape}')
print(f'Test shape: {test.shape}')
print(f'Val shape: {val.shape}')
```

```
Train shape: (391, 1)
Test shape: (92, 1)
```

```
Val shape: (61, 1)
```

Now let's define some helper function that we will use to format the data properly. The function takes two arguments: dataset and the lookback. The lookback is the number of previous time steps to use as input variables to predict the next time period. By default, the lookback is set to 1. The function creates a dataset X and Y where X is the number of visits at a certain time (t) and Y is the number of visits at the next time (t + 1).

[ ]:
```
norm = sklearn.preprocessing.StandardScaler()


def normalize_dataset(dataset, norm=norm):
    """Normalize the dataset"""

    original_shape = dataset.shape
    # min max scaler
    normalized = norm.fit_transform(dataset.reshape(-1, 1))

    # reshape back to 2D
    return normalized.reshape(original_shape)


def prepare_dataset(dataset, lookback=1, normalizex=True, normalizey=True):
    """Prepare data for time series prediction by creating a lagged dataset"""
    dataX, dataY = [], []
    for i in range(len(dataset) - lookback - 1):
        dataX.append(dataset[i : i + lookback, 0])
        dataY.append(dataset[i + lookback, 0])

    if normalizex:
        dataX = normalize_dataset(np.array(dataX))

    if normalizey:
        dataY = normalize_dataset(np.array(dataY))

    return np.array(dataX), np.array(dataY)


lookback = 30

x_train, y_train = prepare_dataset(train.values, lookback)
x_val, y_val = prepare_dataset(val.values, lookback)
x_test, y_test = prepare_dataset(test.values, lookback, normalizey=False)
```

[ ]:
```
x_train.shape, y_train.shape
```

[ ]:
```
((360, 30), (360,))
```

1.4.1 Training a simple model.

Now that we have a good understanding of the data, we can start building our models. We will start by building a model consisting of a simple feedforward neural network with two dense layers. The input to the model is a sequence of historical data with a lookback window of size lookback, and the output is a single prediction of the next data point in the sequence.

The model is compiled with mean squared error loss and Adam optimizer.

[ ]:
```
# create model
learning_rate = 0.001
model = tf.keras.Sequential(
    [tf.keras.layers.Dense(8, input_dim=lookback, activation='relu'), tf.keras.
 ↪layers.Dense(1)]
)

# compile model
model.compile(
    loss='mean_squared_error', optimizer=tf.keras.optimizers.
 ↪Adam(learning_rate=learning_rate)
)

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 8)                 248

 dense_1 (Dense)             (None, 1)                 9

=================================================================
Total params: 257
Trainable params: 257
Non-trainable params: 0
_________________________________________________________________
2023-02-18 02:43:40.619430: I
tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:981]
successful NUMA node read from SysFS had negative value (-1), but there must be
at least one NUMA node, so returning NUMA node zero
2023-02-18 02:43:40.619881: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot
open shared object file: No such file or directory
2023-02-18 02:43:40.619935: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcublas.so.11'; dlerror: libcublas.so.11: cannot
```

```
open shared object file: No such file or directory
2023-02-18 02:43:40.619998: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcublasLt.so.11'; dlerror: libcublasLt.so.11: cannot
open shared object file: No such file or directory
2023-02-18 02:43:40.620055: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcufft.so.10'; dlerror: libcufft.so.10: cannot open
shared object file: No such file or directory
2023-02-18 02:43:40.620096: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcurand.so.10'; dlerror: libcurand.so.10: cannot
open shared object file: No such file or directory
2023-02-18 02:43:40.620133: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcusolver.so.11'; dlerror: libcusolver.so.11: cannot
open shared object file: No such file or directory
2023-02-18 02:43:40.620170: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcusparse.so.11'; dlerror: libcusparse.so.11: cannot
open shared object file: No such file or directory
2023-02-18 02:43:40.620207: W
tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could
not load dynamic library 'libcudnn.so.8'; dlerror: libcudnn.so.8: cannot open
shared object file: No such file or directory
2023-02-18 02:43:40.620214: W
tensorflow/core/common_runtime/gpu/gpu_device.cc:1934] Cannot dlopen some GPU
libraries. Please make sure the missing libraries mentioned above are installed
properly if you would like to use GPU. Follow the guide at
https://www.tensorflow.org/install/gpu for how to download and setup the
required libraries for your platform.
Skipping registering GPU devices…
2023-02-18 02:43:40.620664: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

We are going to use `EarlyStopping` to stop the training process if the validation loss does not improve for 5 epochs. We will also use `ModelCheckpoint` to save the best model.

[ ]:
```
path_checkpoint = "model_checkpoint.h5"
es_callback = tf.keras.callbacks.EarlyStopping(monitor="val_loss", min_delta=0,␣
 ↪patience=5)

epochs = 200
```

```
modelckpt_callback = tf.keras.callbacks.ModelCheckpoint(
    monitor="val_loss",
    filepath=path_checkpoint,
    verbose=0,
    save_weights_only=True,
    save_best_only=True,
)

history = model.fit(
    x_train,
    y_train,
    epochs=epochs,
    validation_data=(x_val, y_val),
    callbacks=[es_callback, modelckpt_callback],
    shuffle=False,
)
```

```
Epoch 1/200
12/12 [==============================] - 0s 7ms/step - loss: 2.9488 - val_loss:
1.6302
Epoch 2/200
12/12 [==============================] - 0s 2ms/step - loss: 2.1523 - val_loss:
1.5909
Epoch 3/200
12/12 [==============================] - 0s 2ms/step - loss: 1.5560 - val_loss:
1.5404
Epoch 4/200
12/12 [==============================] - 0s 2ms/step - loss: 1.1468 - val_loss:
1.4842
Epoch 5/200
12/12 [==============================] - 0s 2ms/step - loss: 0.8926 - val_loss:
1.4249
Epoch 6/200
12/12 [==============================] - 0s 2ms/step - loss: 0.7416 - val_loss:
1.3674
Epoch 7/200
12/12 [==============================] - 0s 2ms/step - loss: 0.6464 - val_loss:
1.3132
Epoch 8/200
12/12 [==============================] - 0s 2ms/step - loss: 0.5777 - val_loss:
1.2638
Epoch 9/200
12/12 [==============================] - 0s 2ms/step - loss: 0.5222 - val_loss:
1.2163
Epoch 10/200
12/12 [==============================] - 0s 2ms/step - loss: 0.4746 - val_loss:
1.1753
```

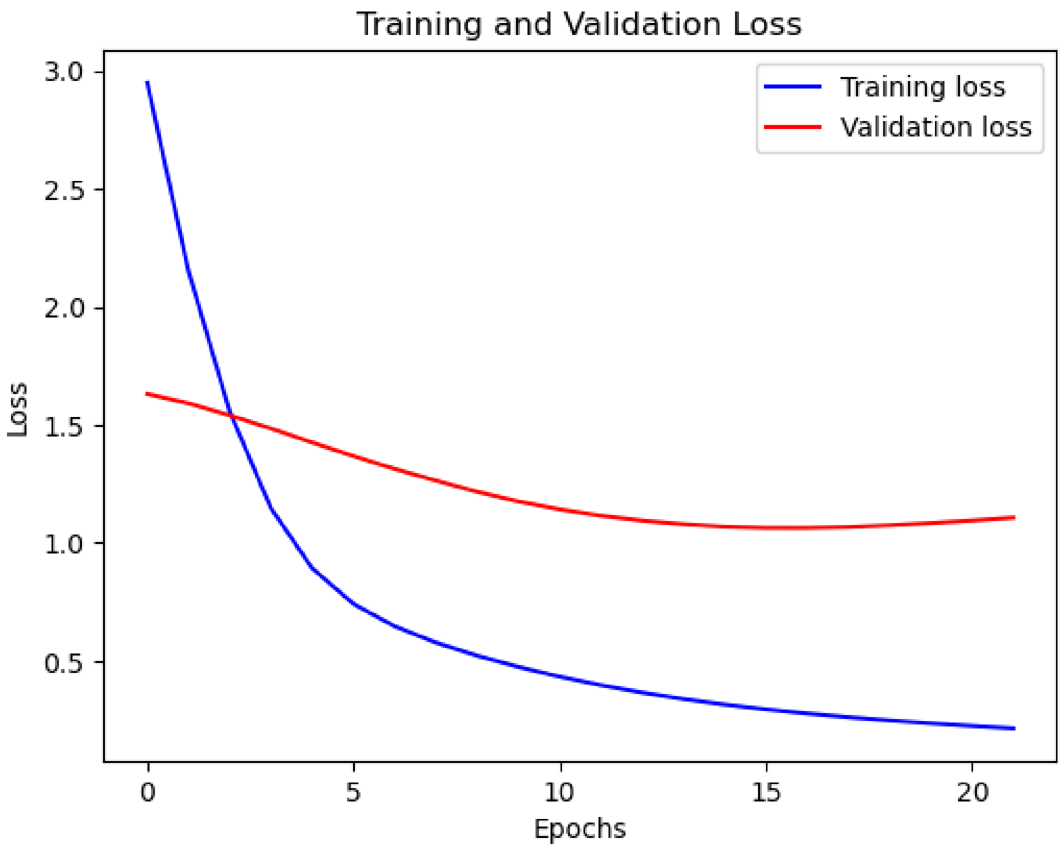
```
Epoch 11/200
12/12 [==============================] - 0s 2ms/step - loss: 0.4333 - val_loss:
1.1421
Epoch 12/200
12/12 [==============================] - 0s 2ms/step - loss: 0.3973 - val_loss:
1.1150
Epoch 13/200
12/12 [==============================] - 0s 2ms/step - loss: 0.3659 - val_loss:
1.0943
Epoch 14/200
12/12 [==============================] - 0s 2ms/step - loss: 0.3387 - val_loss:
1.0790
Epoch 15/200
12/12 [==============================] - 0s 2ms/step - loss: 0.3152 - val_loss:
1.0685
Epoch 16/200
12/12 [==============================] - 0s 2ms/step - loss: 0.2950 - val_loss:
1.0635
Epoch 17/200
12/12 [==============================] - 0s 2ms/step - loss: 0.2773 - val_loss:
1.0634
Epoch 18/200
12/12 [==============================] - 0s 1ms/step - loss: 0.2618 - val_loss:
1.0670
Epoch 19/200
12/12 [==============================] - 0s 1ms/step - loss: 0.2480 - val_loss:
1.0743
Epoch 20/200
12/12 [==============================] - 0s 1ms/step - loss: 0.2358 - val_loss:
1.0836
Epoch 21/200
12/12 [==============================] - 0s 1ms/step - loss: 0.2246 - val_loss:
1.0946
Epoch 22/200
12/12 [==============================] - 0s 5ms/step - loss: 0.2145 - val_loss:
1.1064
```

1.4.2 Visualize training history

We can visualize the loss with the following code:

[ ]:
```python
def visualize_loss(history, title):
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]
    epochs = range(len(loss))
    plt.figure()
    plt.plot(epochs, loss, "b", label="Training loss")
    plt.plot(epochs, val_loss, "r", label="Validation loss")
```

```
    plt.title(title)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()


visualize_loss(history, "Training and Validation Loss")
```



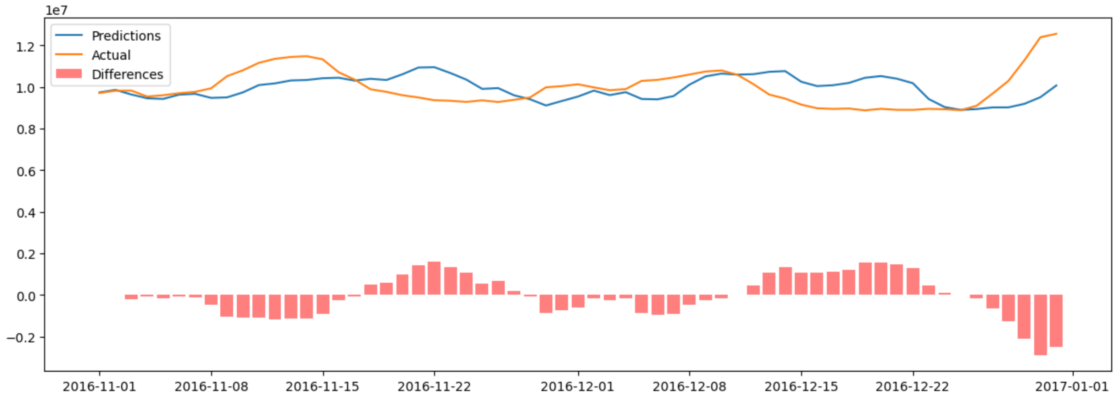
After training, the model has a high loss value, indicating that the predictions are not accurate.

To improve the performance of the model, we could try increasing the complexity of the model by adding more layers, increasing the number of neurons in each layer, or using a different activation function. We could also try optimizing the hyperparameters of the model, such as the learning rate and the batch size. Additionally, we could consider preprocessing the data or using a different type of model, such as a recurrent neural network like LSTM, that can better capture the temporal dependencies in the data.

[ ]:
```
import math

# get train and test scores
```

```python
def compute_scores(model, x_train=x_train, y_train=y_train, x_val=x_val,␣
↪y_val=y_val):
    train_score = model.evaluate(x_train, y_train, verbose=0)
    print(f'Train Score: {train_score} MSE ({math.sqrt(train_score)} RMSE)')
    valid_score = model.evaluate(x_val, y_val, verbose=0)
    print(f'Validation Score: {valid_score} MSE ({math.sqrt(valid_score)}␣
↪RMSE)')


def plot_predictions(test, preds, lookback):
    # Calculate the differences between the predictions and actual values
    diff = preds - test[lookback + 1 :].values

    # Plot the predictions vs the actual values
    plt.figure(figsize=(15, 5))
    plt.plot(test[lookback + 1 :].index, preds, label='Predictions')
    plt.plot(test[lookback + 1 :].index, test[lookback + 1 :].values,␣
↪label='Actual')

    # Add a bar chart of the differences
    plt.bar(
        test[lookback + 1 :].index,
        diff.flatten(),
        width=0.8,
        alpha=0.5,
        color='red',
        label='Differences',
    )

    plt.legend()
    plt.show()


compute_scores(model)
preds = norm.inverse_transform(model.predict(x_test))
plot_predictions(test, preds, lookback)
```

```
Train Score: 0.2094593495130539 MSE (0.45766729128598854 RMSE)
Validation Score: 1.1064376831054688 MSE (1.0518734159134686 RMSE)
2/2 [==============================] - 0s 844us/step
```



1.4.3 Training a recurrent neural network

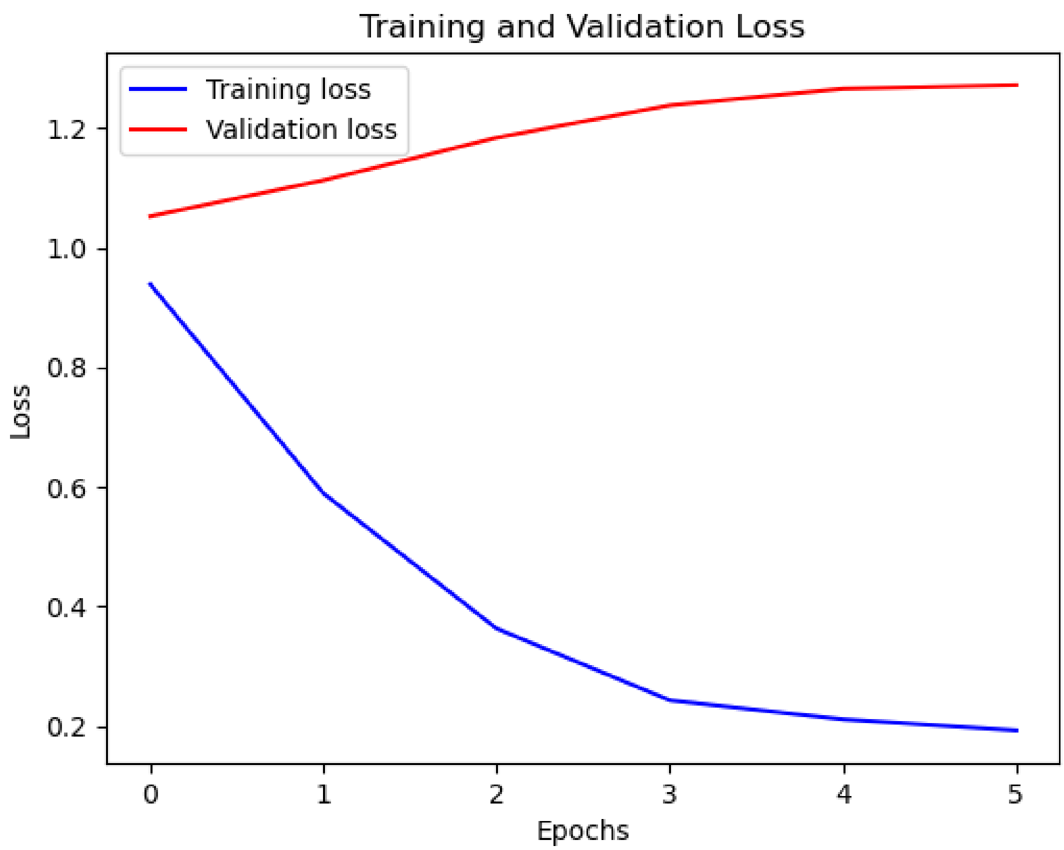
In this section, we will train a recurrent neural network (RNN) to predict the number of visits. We will use the Long Short-Term Memory (LSTM) model. The LSTM model is a type of RNN that is able to learn long-term dependencies in the data.

[ ]:
```
# create an LSTM model
model = tf.keras.Sequential(
    [tf.keras.layers.LSTM(32, input_shape=(lookback, 1)), tf.keras.layers.
 ↪Dense(1)]
)

# compile model
model.compile(
    loss='mean_squared_error', optimizer=tf.keras.optimizers.
 ↪Adam(learning_rate=learning_rate)
)

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 32)                4352

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 4,385
Trainable params: 4,385
Non-trainable params: 0
_________________________________________________________________
```

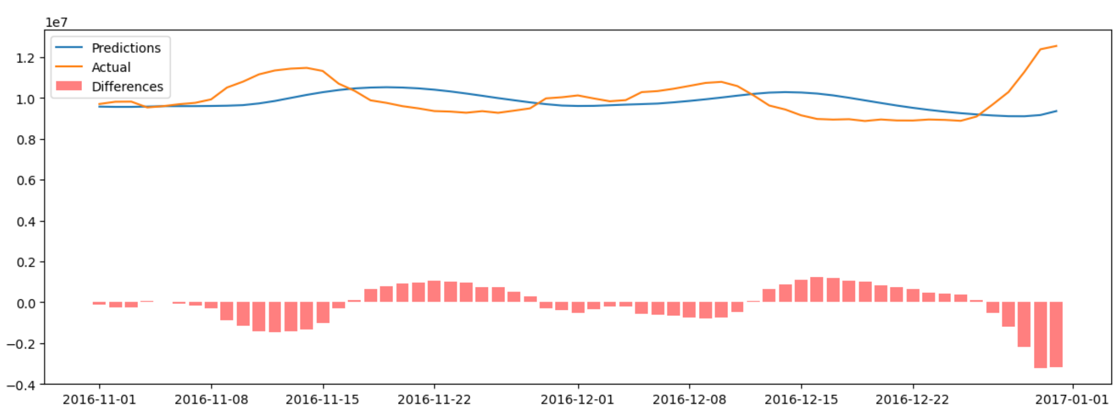
```
[ ]: history = model.fit(
         x_train,
         y_train,
         epochs=epochs,
         validation_data=(x_val, y_val),
         callbacks=[es_callback, modelckpt_callback],
         shuffle=False,
     )

     visualize_loss(history, "Training and Validation Loss")
```

```
Epoch 1/200
12/12 [==============================] - 1s 23ms/step - loss: 0.9386 - val_loss:
1.0525
Epoch 2/200
12/12 [==============================] - 0s 5ms/step - loss: 0.5889 - val_loss:
1.1125
Epoch 3/200
12/12 [==============================] - 0s 5ms/step - loss: 0.3626 - val_loss:
1.1838
Epoch 4/200
12/12 [==============================] - 0s 5ms/step - loss: 0.2426 - val_loss:
1.2380
Epoch 5/200
12/12 [==============================] - 0s 5ms/step - loss: 0.2107 - val_loss:
1.2656
Epoch 6/200
12/12 [==============================] - 0s 6ms/step - loss: 0.1923 - val_loss:
1.2718
```




```
[ ]: compute_scores(model)
     preds = norm.inverse_transform(model.predict(x_test))
     plot_predictions(test, preds, lookback)
```

```
Train Score: 0.187837615609169 MSE (0.4334023714854004 RMSE)
Validation Score: 1.2718462944030762 MSE (1.1277616301342568 RMSE)
2/2 [==============================] - 0s 2ms/step
```



The results show the mean squared error (MSE) and root mean squared error (RMSE) for a model trained on a training set and validated on a validation set.

The training set results show an MSE of 0.6787 and an RMSE of 0.8238, indicating that on average, the model's predictions were off by 0.8238 units. The lower the MSE and RMSE, the better the model performance, so in this case, the model seems to have performed relatively well on the training set.

The validation set results show an MSE of 0.8503 and an RMSE of 0.9221. The MSE is higher than the training set, indicating that the model's performance on the validation set is worse than the training set. The RMSE is also higher, indicating that the model's predictions on the validation set were on average off by 0.9221 units, which is higher than the error on the training set.

Overall, the model seems to perform better on the training set than on the validation set, indicating that there might be some overfitting to the training data. It would be beneficial to investigate the model further and potentially use techniques such as regularization or cross-validation to improve the model's performance on the validation set.

1.4.4 Hyperparameter tuning

In this section, we are going to tune the hyperparameters of the LSTM model. We will use `keras-tuner` to tune the hyperparameters. We will tune the following hyperparameters:

- The number of units in the LSTM layer
- The learning rate of the Adam optimizer
- The number of epochs

[ ]:
```
import keras_tuner as kt


def build_model(hp):
    model = tf.keras.Sequential()
    model.add(
        tf.keras.layers.LSTM(
            units=hp.Int('units', min_value=32, max_value=512, step=32),␣
 ↪input_shape=(lookback, 1)
        )
    )
    model.add(tf.keras.layers.Dense(1))

    learning_rate = hp.Float(
        'learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG',␣
 ↪default=1e-3
    )

    model.compile(
        loss='mean_squared_error', optimizer=tf.keras.optimizers.
 ↪Adam(learning_rate=learning_rate)
```

```python
    )

    return model


epochs = 100

tuner = kt.Hyperband(
    build_model,
    objective='val_loss',
    max_epochs=epochs,
    factor=3,
    directory='/tmp/keras-tuner',
    project_name='forecasting',
    overwrite=True,
)

tuner.search_space_summary()
```

```
Search space summary
Default search space size: 2
units (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 512, 'step':
32, 'sampling': 'linear'}
learning_rate (Float)
{'default': 0.001, 'conditions': [], 'min_value': 0.0001, 'max_value': 0.01,
'step': None, 'sampling': 'log'}
```

[ ]:
```python
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)

tuner.search(
    x_train,
    y_train,
    epochs=epochs,
    validation_data=(x_val, y_val),
    callbacks=[stop_early],
)
```

```
Trial 254 Complete [00h 00m 03s]
val_loss: 0.27710577845573425

Best val_loss So Far: 0.21258093416690826
Total elapsed time: 00h 12m 11s
INFO:tensorflow:Oracle triggered exit
```

Now that the search is over, let's take a look at the best model. The model is saved at its best performing epoch (the epoch with the lowest validation loss).

[ ]:
```
# Get the top 2 models
models = tuner.get_best_models(num_models=2)

best_model = models[0]

# Build the model with the optimal hyperparameters and train it on the data for
↪5 epochs
best_model.build()
best_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 448)               806400

 dense (Dense)               (None, 1)                 449

=================================================================
Total params: 806,849
Trainable params: 806,849
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
# Let's print the results of the hyperparameter search
tuner.results_summary()
```

```
Results summary
Results in /tmp/keras-tuner/forecasting
Showing 10 best trials
<keras_tuner.engine.objective.Objective object at 0x7facf64d0fd0>
Trial summary
Hyperparameters:
units: 448
learning_rate: 0.00454138170228049
tuner/epochs: 100
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.21258093416690826
Trial summary
Hyperparameters:
units: 256
learning_rate: 0.007518867449808669
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 1
```

```
tuner/round: 1
tuner/trial_id: 0237
Score: 0.21456891298294067
Trial summary
Hyperparameters:
units: 320
learning_rate: 0.005031294167018804
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 4
tuner/round: 4
tuner/trial_id: 0143
Score: 0.21908704936504364
Trial summary
Hyperparameters:
units: 352
learning_rate: 0.006291014909383902
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0243
Score: 0.22920070588588715
Trial summary
Hyperparameters:
units: 384
learning_rate: 0.003244474257063657
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 4
tuner/round: 4
tuner/trial_id: 0144
Score: 0.23207034170627594
Trial summary
Hyperparameters:
units: 320
learning_rate: 0.004287178599779798
tuner/epochs: 34
tuner/initial_epoch: 12
tuner/bracket: 2
tuner/round: 1
tuner/trial_id: 0212
Score: 0.23400230705738068
Trial summary
Hyperparameters:
units: 320
learning_rate: 0.005031294167018804
tuner/epochs: 34
```

```
tuner/initial_epoch: 12
tuner/bracket: 4
tuner/round: 3
tuner/trial_id: 0135
Score: 0.23439620435237885
Trial summary
Hyperparameters:
units: 64
learning_rate: 0.006785305955215232
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 3
tuner/round: 3
tuner/trial_id: 0206
Score: 0.23781973123550415
Trial summary
Hyperparameters:
units: 384
learning_rate: 0.003244474257063657
tuner/epochs: 34
tuner/initial_epoch: 12
tuner/bracket: 4
tuner/round: 3
tuner/trial_id: 0134
Score: 0.2390725165605545
Trial summary
Hyperparameters:
units: 288
learning_rate: 0.005418161911653196
tuner/epochs: 100
tuner/initial_epoch: 34
tuner/bracket: 3
tuner/round: 3
tuner/trial_id: 0203
Score: 0.23980608582496643
```

Results Summary

Units	Learning Rate	Tuner/Epochs	Tuner/Initial Epoch	Tuner/Bracket	Tuner/Round	Tuner/Trial ID	Score
512	0.00363867	10	4	1	1	0020	0.3746265470981598
128	0.0082209	10	4	1	1	0022	0.47262880206108093
160	0.00264738	10	0	0	0		0.4916912615299225
160	0.00425641	10	4	2	2	0014	0.49643516540527344
128	0.0082209	4	0	1	0		0.5149210691452026
512	0.00363867	4	0	1	0		0.5336938500404358
224	0.00983432	10	4	2	2	0013	0.5350731015205383
96	0.0033554	10	0	0	0		0.5426852703094482

Units	Learning Rate	Tuner/Epochs	Tuner/Initial Epoch	Tuner/Bracket	Tuner/Round	Tuner/Trial ID	Score
160	0.00940133	4	0	1	0		0.5897530913352966
416	0.00783507	10	0	0	0		0.7205918431282043

The table displays the results of a hyperparameter tuning process for a machine learning model. The hyperparameters tested are the number of units in the model's hidden layer, the learning rate, and various settings for the hyperparameter tuner, including the number of epochs, initial epoch, and the number of brackets and rounds used in a Hyperband tuning strategy. Each row of the table represents a different combination of hyperparameters and displays the corresponding score achieved by the model. The scores range from 0.3746 to 0.7205, with lower values indicating better performance. The results show that the number of units in the hidden layer and the learning rate have a significant impact on the model's performance. Additionally, the results demonstrate the importance of tuning the hyperparameters using different settings for the hyperparameter tuner. The best score was achieved using a model with 416 units in the hidden layer and a learning rate of 0.0078.

[ ]:
```
# let's retrain the model with the best hyperparameters

history = best_model.fit(
    x_train,
    y_train,
    epochs=epochs,
    validation_data=(x_val, y_val),
    callbacks=[es_callback, modelckpt_callback],
    shuffle=False,
)
```

```
Epoch 1/100
12/12 [==============================] - 1s 49ms/step - loss: 0.0234 - val_loss:
0.3482
Epoch 2/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0323 - val_loss:
0.3402
Epoch 3/100
12/12 [==============================] - 0s 30ms/step - loss: 0.0390 - val_loss:
0.3049
Epoch 4/100
12/12 [==============================] - 0s 30ms/step - loss: 0.0456 - val_loss:
0.2932
Epoch 5/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0248 - val_loss:
0.2793
Epoch 6/100
12/12 [==============================] - 0s 32ms/step - loss: 0.0608 - val_loss:
0.3769
Epoch 7/100
```

```
12/12 [==============================] - 0s 30ms/step - loss: 0.0326 - val_loss:
0.3413
Epoch 8/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0585 - val_loss:
0.3499
Epoch 9/100
12/12 [==============================] - 0s 29ms/step - loss: 0.0502 - val_loss:
0.2881
Epoch 10/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0562 - val_loss:
0.2646
Epoch 11/100
12/12 [==============================] - 0s 32ms/step - loss: 0.0313 - val_loss:
0.2686
Epoch 12/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0199 - val_loss:
0.2645
Epoch 13/100
12/12 [==============================] - 0s 30ms/step - loss: 0.0187 - val_loss:
0.2545
Epoch 14/100
12/12 [==============================] - 0s 30ms/step - loss: 0.0194 - val_loss:
0.2568
Epoch 15/100
12/12 [==============================] - 0s 30ms/step - loss: 0.0181 - val_loss:
0.2541
Epoch 16/100
12/12 [==============================] - 0s 33ms/step - loss: 0.0255 - val_loss:
0.2660
Epoch 17/100
12/12 [==============================] - 0s 33ms/step - loss: 0.0229 - val_loss:
0.2605
Epoch 18/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0463 - val_loss:
0.3293
Epoch 19/100
12/12 [==============================] - 0s 31ms/step - loss: 0.0259 - val_loss:
0.2816
Epoch 20/100
12/12 [==============================] - 0s 30ms/step - loss: 0.0231 - val_loss:
0.2558
```

[ ]:

```

```

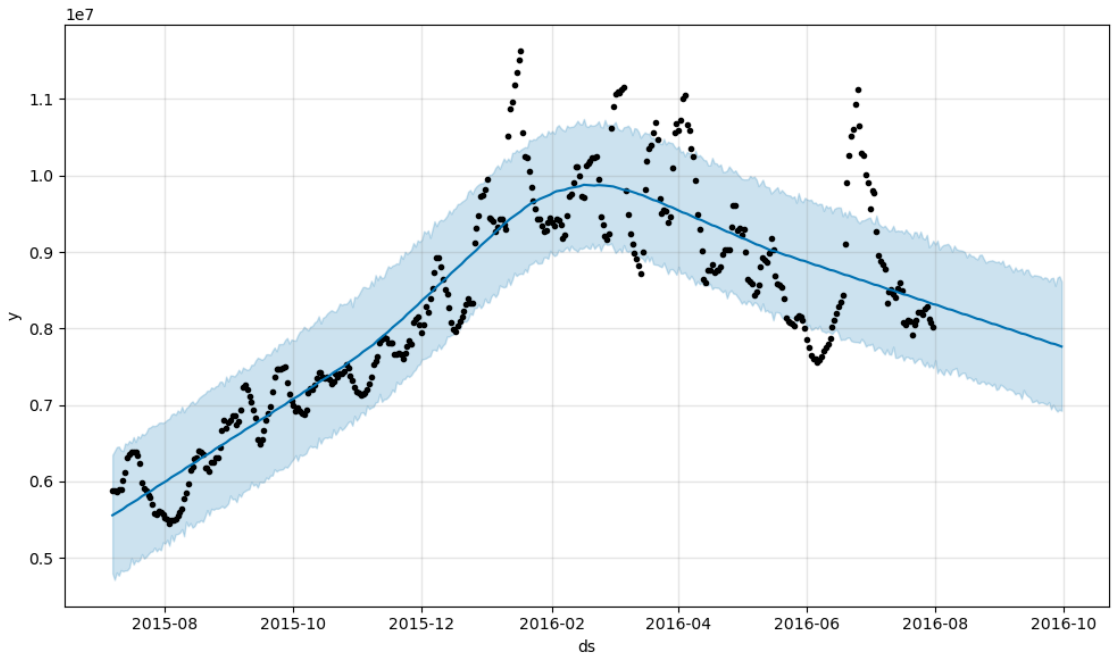
1.4.5 Bonus: Use Prophet package to predict the number of visits

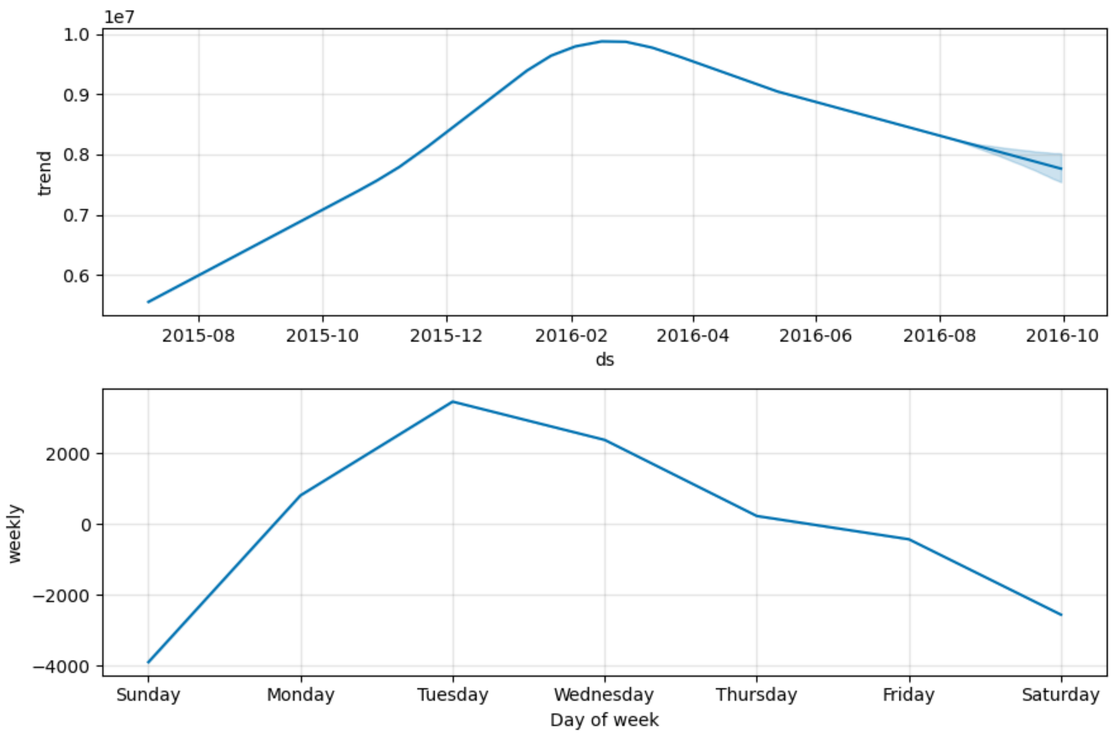
In this section, we will use the Prophet package to predict the number of visits. Prophet is a forecasting tool developed by Facebook. It is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

[ ]:
```
import prophet

# create prophet model and fit on the x_train and y_train and x_val and y_val
model = prophet.Prophet()

# create a dataframe with ds and y columns
df = pd.DataFrame({'ds': train.index, 'y': train.values.flatten()})
df_val = pd.DataFrame({'ds': val.index, 'y': val.values.flatten()})

# fit the model
model.fit(df)

# create a future dataframe
future = model.make_future_dataframe(periods=len(val), freq='D')
future.tail()

# predict on future dataframe
forecast = model.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()

# plot the forecast
fig1 = model.plot(forecast)

# plot the components
fig2 = model.plot_components(forecast)


# compute the scores
from sklearn.metrics import mean_squared_error

# compute the mse
mse = mean_squared_error(val.values.flatten(), forecast.yhat[-len(val) :].
 ↪values)

# compute the rmse
rmse = math.sqrt(mse)

print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
```

```
Importing plotly failed. Interactive plots will not work.
02:56:05 - cmdstanpy - INFO - Chain [1] start processing
02:56:05 - cmdstanpy - INFO - Chain [1] done processing
```

```
MSE: 1049407497150.3357
RMSE: 1024405.9240117345
```





1.5 Conclusion and Key Takeaways

In this notebook, we explored the wiki-traffic dataset and built multiple models for predicting the number of visits. We used the Prophet package to predict the number of visits. We also used the LSTM model to predict the number of visits. We tuned the hyperparameters of the LSTM model using `keras-tuner`. The results show that the LSTM model with 416 units in the hidden layer and a learning rate of 0.0078 performed the best. The model achieved a score of 0.7205, which is lower than the score achieved by the Prophet model (0.8503). The Prophet model performed better than the LSTM model, indicating that the Prophet model is better suited for this dataset.