TECHNICAL DOCUMENTATION

PREDICTION OF HOTEL OCCUPANCY

Matija Kukić, Ivan Džanija, Valentina Vidović, Dora Baričević

Table of Contents

App

We put together FastAPI, Streamlit and Docker Compose to make a web app. FastAPI handles the behind-the-scenes stuff efficiently, making our APIs strong and adaptable, which is super important for machine learning. Streamlit makes it easy for users to interact with and see data, so everyone can understand it better. Docker Compose helps us manage everything so our app runs smoothly.

FastAPI

FastAPI is a modern, fast (as the name suggests), web framework for building APIs with Python 3.7+ based on standard Python type hints. It is built on top of Starlette for the web parts and Pydantic for the data parts. FastAPI is known for its high performance due to its asynchronous capabilities and automatic generation of interactive API documentation using Swagger UI and ReDoc. It's often chosen for its ease of use, typing support, and performance, making it a popular choice for building APIs, especially in the data science community.

Streamlit

Streamlit is an open-source Python library that makes it easy to create interactive web apps for data science and machine learning projects. It allows you to build interactive web applications directly from Python scripts, without the need for HTML, CSS, or JavaScript. Streamlit is designed to be simple and intuitive, enabling data scientists to quickly prototype and share their work with others. It provides various widgets and components for building interactive elements such as sliders, buttons, and plots, and it automatically updates the app in real-time as the underlying Python code changes.

Docker

Docker is a platform that allows you to package, distribute and run applications in lightweight, isolated containers. Containers are similar to virtual machines but more lightweight and portable, as they share the host operating system's kernel and only contain the application and its dependencies. Docker uses a declarative syntax called Dockerfile to define the environment and dependencies required for running an application. Docker Compose is a tool that allows you to define and run multi-container Docker applications using a YAML file. It simplifies the process of managing complex applications with multiple services by defining them in a single configuration file and enabling easy orchestration of containers.

Instructions

To start the app, follow these steps:
Use Docker Compose to build and run the Docker containers. Open your terminal and enter the command: docker-compose up --build
This command will launch two services: one for the frontend and one for the backend.

After the containers are up and running, open your web browser and visit http://localhost:8501 to see the app. You can also check out the API service at http://localhost:8000.

Model implementation

First we opened the give train data set and started cleaning it

Cleaning

```python
train_df = pd.read_parquet(path)
train_df["date_from"] = pd.to_datetime(train_df["date_from"])
train_df["reservation_date"] = pd.to_datetime(train_df["reservation_date"])
train_df["date_to"] = pd.to_datetime(train_df["date_to"])
train_df["stay_date"] = pd.to_datetime(train_df["stay_date"])
train_df["cancel_date"] = pd.to_datetime(train_df["cancel_date"])
#print(df.head())
#df.info()
#df.describe()
b = list((train_df["date_to"]-train_df["date_from"])/np.timedelta64(1,"D"))
del train_df["night_number"]
for i in range(len(b)):
    if b[i] == 0:
        b[i] = 1
train_df["stay_nights"] = b
train_df["price_per_night"] = train_df["price"] / train_df["stay_nights"]
#f = open("data_u_txt.txt","w+")
#f.write(df.head().to_string())
#f.close()
```

We opened the parquet with the given train data set, and calculated the price per night and added it as a column in the data frame.

```python
df = data1
df = df[df["date_from"] >= df["reservation_date"]]
df = df[ (df["reservation_date"] <= df["cancel_date"] ) | (df["cancel_date"].isna())]
df = df[df["adult_cnt"] > 0]
df = df[ (df["cancel_date"] < df["date_to"]) | (df["cancel_date"].isna())  ]
df.drop_duplicates(subset = "reservation_id", inplace = True, keep = "first")
df = df[df["reservation_status"] == "Checked-out"]
df["guest_count"] = df["adult_cnt"] + df["children_cnt"]
df["nightly_price_per_guest"] = df["price_per_night"] / df["adult_cnt"]
prices_df = df[df["price_per_night"] > 150]
prices_df = prices_df[prices_df["nightly_price_per_guest"] < 6500]
#prices_df.to_parquet("train_data_cleaned.parquet")
return prices_df
```

Then we cleaned the data from incorrect data (eg. reservation date must be before cancel date) and deleted some outliers that were statistically insignificant(guests who payed less than 150 of the given currency and those who payed more than 6500)

Model training and saving

```
path = Path.cwd().parent.parent
filePath = str(path) + "/data_cleanup/second_dataset/train_data_price_corrected.parquet"
df = pd.read_parquet(filePath)
df.info()
```

Here we open the parquet file where data cleaning was finished and store it as a data frame.

When executed this code gives us a data frame where for each date in the set we have the amount of rooms filled on that given day(we delete the date column and turn it into a list for easier plotting).

```
# Apply logarithm function to guest_count data
room_occupancy["room_count_log"] = np.log(room_occupancy["room_cnt"])

# Seasonal differencing
seasonal_difference = room_occupancy['room_count_log'].diff(periods=365)

# Remove NaN values resulted from differencing
seasonal_difference = seasonal_difference.dropna()
```

Here we take the log of the number of rooms occupied and differentiate the data with the number of occupied rooms from a year ago(365 periods).

```
test = room_occupancy.iloc[2:368]
train = room_occupancy.iloc[368:-1]
print(test.shape, train.shape)
```

Here we take a test set and a train set on which we will be training and testing the model on, train set is the data from 2009 and the test set is the data from 2008.

```
test_list =  test["room_cnt"].tolist()
train_list = train["room_cnt"].tolist()
mid_point_test = list()
date_list = test.index.tolist()
for i in range(len(test)):
    mid_point_test.append((test_list[i] + train_list[i]) / 2)
```

Now we create the mid_point_test list that allows us to take both years into account for the quantile that will be used for the conformal prediction.

```
mod = ARIMA(seasonal_difference_df["room_cnt"], order = (0,0,2), seasonal_order = (1,0,1,7))
mod = mod.fit()
mod.summary()
```

Here is the code that trains our ARIMA model with the parameters that we calculated to give us the best results.

```
start = len(seasonal_difference_df)
end = len(test) + len(seasonal_difference_df) - 1

pred = mod.predict(start = start, end = end)

original_train = room_occupancy.iloc[369:]
print(len(pred), len(test))
prediction_list = list()
for i, data in enumerate(train["room_count_log"]):
    prediction_list.append(data)
for i, prediction in enumerate(pred):
    prediction_list[i] += prediction
for i in range(len(prediction_list)):
    prediction_list[i] = math.exp(prediction_list[i])
print(prediction_list)
# Now, 'reverted_values' contains the predicted values in their original scale
```

Here we tested our prediction on the test set. Manual re-differentiation and returning of the predictions value from its logarithmic value is visible in this code.

```python
#residual calulation
residuals = list()
for i in range(len(mid_point_test)):
    residuals.append(mid_point_test[i]-predic
#residuals = mid_point_test - prediction_list
#residuals_list = residuals.values.tolist()


residuals_list = residuals
residuals_sorted = np.sort(residuals_list)

p_values = [(np.sum(residuals_list >= r) + 1) / (len(residuals_list) + 1) for r in residuals_sorted]
alpha = 0.05

upper_quantile = np.quantile(residuals_sorted, 1 - alpha / 2)
lower_quantile = np.quantile(residuals_sorted, alpha / 2)

def predict_new_point(prediction_point):
    upper_bound = prediction_point + upper_quantile
    lower_bound = prediction_point + lower_quantile
    return lower_bound, upper_bound

lower_list = list()
upper_list = list()
for i in prediction_list:
    lo, hi = predict_new_point(i)
    lower_list.append(lo)
    upper_list.append(hi)
```

Here we calculated the residuals of our prediction and the test set, then calculated the quantiles that we used for the conformal prediction.

```
import joblib
joblib.dump(mod, 'arima_trained.pkl')
```

We saved the trained model as a pickle file that can be opened and used for prediction in predict.py that is used by our docker deployment implementation.

Prediction making

```
model = jl.load('arima_trained.pkl')
df = pd.read_parquet('test.parquet')

#... grouping by day...

df['date_from'] = pd.to_datetime(df['date_from'])
df['date_to'] = pd.to_datetime(df['date_to'])
date1 = df["date_from"].min()
date2 = df["date_to"].max()
start = (date1 - datetime(2010,1,2)).days
end = int((date2-date1).days) + start - 1
#print(start,end,date1,date2)
pred = model.predict(start+366,end+366)
```

The beginning of prediction.py looks something like this. First we open our model and the parquet where the test data is contained. The part left out in the commented line is where we take the data frame and group the data by date and get room_cnt or the number of rooms occupied on a given date from the test set the exact same as we did in model training, the room_cnt is only used to compare the data we predict later. Then we extract the starting date from the test data and the end date and give our models prediction. The rest of the code is exactly the same as the model training code, and a plot comparing our conformal prediction and the test data is given at the end.

References

[1] Python - https://docs.python.org/3/

[2] FastAPI - https://fastapi.tiangolo.com/

[3] Streamlit - https://docs.streamlit.io/

[4] Docker - https://docs.docker.com/