# Urban Mood Forecaster

A Time Series Analysis and Forecasting System for Global Sentiment Dynamics and Travel Recommendations

Time Series Course Project

**Team Members**

Shushan Meyroyan
Shushan Gevorgyan
Armen Madoyan
Davit Sargsyan

American University of Armenia
November 2025

# Contents

# Abstract

This project investigates the temporal evolution of public sentiment across countries using the Twitter Sentiment Geographical Index (TSGI) dataset. The goal is to quantify how collective mood changes over time, evaluate multiple forecasting models, and build a practical *Urban Mood Forecaster* application that can support travel decisions and urban well-being analysis.

We first construct daily sentiment time series from country-level TSGI scores, perform data cleaning and interpolation, and then apply a range of forecasting models, including classical statistical models (AR, MA, ARIMA, SARIMA, Holt–Winters), and deep learning models based on recurrent neural networks (LSTM, GRU, Bidirectional LSTM, and Stacked LSTM). Model performance is evaluated primarily with the root mean squared error (RMSE) on held-out test data.

Beyond forecasting, the system computes seasonal sentiment profiles to identify the best season to visit each country and recommends the happiest countries for a chosen season. All functionality is exposed through an interactive Streamlit web application with rich Plotly visualizations.

The results show that classical SARIMA models provide strong baselines, while Bidirectional LSTM and LSTM models achieve the lowest RMSE on global sentiment, capturing non-linear patterns and complex temporal dependencies. The developed system demonstrates how open social media sentiment data, modern time series models, and an interactive interface can be combined to support data-driven reasoning about urban mood dynamics and travel planning.

# Chapter 1

# Motivation, Problem Statement, and Project Goals

## 1.1 Motivation

Promoting subjective well-being is a central element of the United Nations Sustainable Development Goals. Traditional economic indicators such as GDP or unemployment rates measure economic activity but do not directly capture how people feel in their daily lives. Governments and researchers increasingly rely on subjective well-being (SWB) indicators as a complement to these objective metrics.

Social media offers a new source of information for approximating collective mood. Public, geotagged tweets provide time-stamped and location-specific signals that can be aggregated into sentiment indices. These indices do not perfectly reflect the mood of an entire population, because Twitter users are not a representative sample, but they do provide a fast and high-frequency proxy for emotional trends and reactions to events.

The Twitter Sentiment Geographical Index (TSGI) converts billions of tweets into sentiment scores at different spatial levels. This makes it possible to analyze how public sentiment evolves over time in different countries and to ask questions such as:

- How did global mood change during major events?
- Are some seasons consistently "happier" than others in a given country?
- Can we forecast future sentiment based on the past?

Our project uses these ideas to build *Urban Mood Forecaster*, a system that combines time series analysis, deep learning, and an interactive web interface.

## 1.2 Problem Statement

The central problem of the project can be summarized as follows: given a daily time series of sentiment scores for each country, we want to model, forecast, and interpret the dynamics of public sentiment, and use these forecasts for simple travel-related recommendations.

More concretely, we address the following questions.

First, we ask whether it is possible to model and forecast the temporal dynamics of public sentiment at the country level using standard time series models. This includes autoregressive and moving average models, combinations such as ARIMA, and seasonal models such as SARIMA that explicitly capture periodic patterns.

Second, we compare these classical models with deep learning recurrent models, particularly LSTM and GRU variants. Recurrent neural networks are designed to handle sequential data and may capture non-linear dependencies that are not easily modeled by linear ARIMA-type approaches.

Third, we use the time series structure to identify which seasons are happiest for a given country, and which countries are happiest in each season. This transforms a raw forecasting task into a more interpretable question that is relevant to travel and seasonal mood patterns.

Finally, we want to expose all of this functionality through an application that non-experts can use. The system should load the data, fit the models, produce forecasts, and present results through interactive plots and simple textual recommendations, without requiring the user to know the statistical or deep learning details.

## 1.3 Project Goals

The project goals can be grouped into five main objectives.

The first objective is to build clean daily sentiment time series for each country from the TSGI data. This involves merging yearly CSV files, converting dates, enforcing a regular daily frequency, and dealing with missing values through interpolation.

The second objective is to fit and compare multiple time series models, using RMSE as the primary evaluation metric on a held-out test set. This comparison includes both naive baselines and more sophisticated ARIMA-family models.

The third objective is to design and train recurrent neural network models that operate on sliding windows of past sentiment values to predict future values. These models include simple LSTMs, GRUs, Bidirectional LSTMs, and deeper stacked LSTMs.

The fourth objective is to develop seasonal mood profiles and travel recommendation logic. This requires mapping months to seasons, aggregating average sentiment by season and country, and using these summaries to suggest the best season to visit or the happiest countries in a given season.

The final objective is to implement a Streamlit web application, called *Urban Mood Forecaster*, that wraps all of the above in an accessible interface. The app needs to load data, run models, display interactive visualizations, and present clear explanations and recommendations.

# Chapter 2

# Dataset and Raw File Structure

## 2.1 Twitter Sentiment Geographical Index (TSGI)

The Twitter Sentiment Geographical Index (TSGI) is a dataset designed to monitor global well-being using social media sentiment. It is produced jointly by the Sustainable Urbanization Lab at MIT and the Center for Geographic Analysis at Harvard University. The idea behind TSGI is to use public, geotagged tweets, apply a sentiment analysis model, and aggregate the results at various spatial and temporal resolutions.

The core motivation is that promoting well-being is an explicit goal of the United Nations, and many governments wish to complement traditional indicators with subjective well-being measures. TSGI offers one way to approximate subjective well-being by tracking how positive or negative public tweets are in different parts of the world over time.

## 2.2 Granularity and Temporal Coverage

The dataset provides sentiment indices at several spatial levels. At the coarsest level there is a global index, summarizing sentiment across the entire world. At the next level, there are country-level scores, followed by state or province level scores, and finally county or city level scores. This hierarchical structure allows users to choose a level of analysis appropriate for their question.

In this project we focus on the country level. Country-level analysis has several advantages. It simplifies the data volume compared to city-level analysis, which is important when training deep learning models, and it aligns naturally with travel scenarios: when planning a trip, people often choose a country rather than a specific county.

Temporally, the data span from 2012 to at least 2023, and are updated monthly in the Harvard Dataverse platform. Each record corresponds to a particular date, indicating the aggregated sentiment score for that date and location.

## 2.3 Field Definitions

For the country-level CSV files used in our project, the key fields are:

- `DATE`: the date of the sentiment measurement, interpreted as a daily timestamp.
- `NAME_0`: the country name, which we later rename to `country` for clarity.
- `SCORE`: a sentiment index value in the interval $[0, 1]$, where 0 represents negative overall sentiment and 1 represents positive overall sentiment.
- `N`: the number of tweets contributing to the sentiment score for that date and country.

It is important to emphasize that we do not recompute sentiment ourselves. The sentiment model and aggregation procedure are provided by the TSGI authors. We treat `SCORE` as a ready-made numeric signal that can be modeled as a time series.

## 2.4 Raw File Organization

The original data are distributed as yearly CSV files. Each file corresponds to one year of country-level sentiment data. In the project repository, we store these files under a dedicated directory, for example:

```
dataverse_files/
    Sentiment Data - Country/
        TSGI_country_2012.csv
        TSGI_country_2013.csv
        ...
```

Each file contains multiple countries, with one row per date and country. The naming convention is important because we use the year encoded in the filename later when assembling the combined dataset.

## 2.5 Design Decisions

Choosing country-level data and using the precomputed `SCORE` field are deliberate design decisions. They allow us to focus on the time series modeling and application-building aspects without needing to handle raw text or train a sentiment classifier. This separation of concerns is typical in real projects, where a downstream analysis team consumes a processed dataset prepared by another group.

# Chapter 3

# Data Preprocessing and Time Series Construction

## 3.1 Combining Yearly CSV Files

The first step in our pipeline is to combine all yearly CSV files into a single DataFrame. This is done by the helper function `load_sentiment_data(data_directory)` in `backend.py`. The function proceeds as follows.

It scans the directory for all files with the extension `.csv`. For each file, it loads the contents using `pandas.read_csv`. The filename follows a pattern such as `TSGI_country_2012.csv`, so we can extract the year `2012` from the string and store it in a new column called `year`. This step is not strictly necessary because the `DATE` column already contains the full date, but the `year` column is convenient for quality-control checks and filtering.

After all files are read, the function concatenates the individual DataFrames into a single large DataFrame using `pd.concat`. It then renames the `NAME_0` column to `country` to have a more intuitive column name. The `DATE` column is converted from string to `datetime` using `pd.to_datetime`, so that resampling and time-based indexing work properly. Finally, the combined DataFrame is sorted by `country` and `DATE`.

The result is a unified dataset with columns {`DATE`, `country`, `N`, `SCORE`, `year`} that contains all years and all countries.

## 3.2 Extracting a Single-Country Series

For analysis and forecasting, we typically work with one country at a time. The helper function `get_country_data(combined_df, country)` takes the combined DataFrame and a country name as input.

It first filters the rows where the `country` column matches the requested country name. Then it keeps only the `DATE` and `SCORE` columns, because for univariate time series forecasting we only need the sentiment values and their time stamps.

Next, it sets `DATE` as the index so that the DataFrame becomes a time-indexed series. The crucial step is calling `asfreq("D")`, which enforces a daily frequency. If some dates are missing in the original data (for example, no tweets were collected or no score was reported on a particular day), this method introduces explicit missing entries for those dates in the index.

## 3.3 Handling Missing Values

Once the time series has a daily index, we must handle missing sentiment values. Simply dropping missing days would break the time continuity and reduce the effective length of the series. Instead, we use linear interpolation.

When we call `interpolate("linear")` on the time series, pandas fills each missing value by drawing a straight line between the nearest known values before and after the gap. For example, if the series contains a value of 0.6 on day 1 and 0.8 on day 3, then the missing day 2 receives the interpolated value 0.7.

This approach assumes that sentiment changes relatively smoothly over short intervals when there are no observed data points, which is a reasonable approximation for aggregated indices. After interpolation, the series is contiguous and suitable for classical time series models and recurrent neural networks.

## 3.4 Global Series Construction

In addition to country-specific series, we also construct a global sentiment series by averaging the `SCORE` values across all countries for each day. This global series allows us to benchmark models on a single, longer series that aggregates worldwide mood. The same interpolation and resampling procedures are applied, resulting in a continuous daily global series.

## 3.5 Train–Test Split

To evaluate forecasting models, we divide each time series into a training part and a test part. We use a chronological split to preserve the temporal ordering and avoid information leakage from the future.

Let the series consist of values $y_1, y_2, \ldots, y_T$. We choose a split index

$$i^\star = \lfloor 0.8T \rfloor,$$

so that the first 80% of observations are used as the training set and the remaining 20% as the test set. Formally,

$$\text{train} = \{y_t\}_{t=1}^{i^\star}, \qquad \text{test} = \{y_t\}_{t=i^\star+1}^{T}.$$

This method mimics a realistic forecasting scenario, where a model is trained on past data and evaluated on unseen future data.

## 3.6 Scaling and Sequence Construction for RNNs

Recurrent neural networks work best when inputs are scaled to a standardized range. In our project we use Min–Max scaling to map all sentiment values into the interval $[0, 1]$. This is implemented with `sklearn.preprocessing.MinMaxScaler`.

Consider a one-dimensional array of sentiment values $(y_1, \dots, y_T)$. The scaler computes

$$z_t = \frac{y_t - y_{\min}}{y_{\max} - y_{\min}},$$

where $y_{\min}$ and $y_{\max}$ are the minimum and maximum of the training data. The resulting normalized sequence $(z_1, \dots, z_T)$ lies in $[0, 1]$.

For RNNs, we convert this normalized series into overlapping input–target pairs using sliding windows. Given a window length $L$, the function `create_sequences(data, seq_length)` creates pairs

$$\mathbf{x}^{(i)} = (z_i, z_{i+1}, \dots, z_{i+L-1}), \qquad y^{(i)} = z_{i+L},$$

for $i = 1, \dots, n - L$, where $n$ is the length of the normalized sequence. Each $\mathbf{x}^{(i)}$ is an input sequence of length $L$, and $y^{(i)}$ is the target value the model must predict.

These sequences are then reshaped to the three-dimensional format $(\text{samples}, \text{timesteps}, \text{features}) = (N, L, 1)$ required by Keras RNN layers, where $N$ is the number of training samples.

The helper function `prepare_rnn_data(train, test, seq_length=30)` automates this process. It concatenates train and test values, fits the Min–Max scaler, splits scaled data back into train and test portions, and calls `create_sequences` separately on each part. The result is a consistent and normalized set of input–target pairs for training and evaluating RNNs.

# Chapter 4

# Classical Time Series Models

This chapter explains the statistical models we use as baselines and benchmarks: autoregressive models (AR), moving average models (MA), ARIMA, seasonal ARIMA (SARIMA), and Holt–Winters exponential smoothing. We also describe how these models are implemented in Python using the `statsmodels` library.

## 4.1 Autoregressive Model AR(1)

An autoregressive model of order 1, denoted AR(1), assumes that the current value of the series depends linearly on its immediate past value plus noise. The model equation is

$$y_t = \phi_0 + \phi_1 y_{t-1} + \varepsilon_t, \tag{4.1}$$

where $\phi_0$ is a constant, $\phi_1$ is the autoregressive coefficient, and $\varepsilon_t$ is white noise with zero mean and constant variance.

Intuitively, if $\phi_1$ is positive, a high value at time $t-1$ increases the expected value at time $t$; if it is negative, the series tends to reverse direction. If $|\phi_1| < 1$, the effect of a shock fades over time, which is important for stationarity.

In Python, we fit an AR(1) model using the `ARIMA` class with order $(1, 0, 0)$, because AR is a special case of ARIMA with no differencing and no moving average part:

```
from statsmodels.tsa.arima.model import ARIMA

ar_model = ARIMA(train, order=(1, 0, 0)).fit()
ar_forecast = ar_model.forecast(len(test))
```

## 4.2 Moving Average Model MA(1)

A moving average model of order 1, denoted MA(1), expresses the current value of the series as a linear combination of the current and previous noise terms:

$$y_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1}, \tag{4.2}$$

where $\mu$ is the mean of the series, $\varepsilon_t$ is white noise, and $\theta_1$ is the moving average coefficient.

In an MA model, the noise terms are not directly observed, but they are inferred during model fitting. The key idea is that correlation in the series is captured by how past shocks influence the present value.

We implement MA(1) using `ARIMA` with order $(0, 0, 1)$, because this corresponds to a pure MA model:

```
ma_model = ARIMA(train, order=(0, 0, 1)).fit()
ma_forecast = ma_model.forecast(len(test))
```

## 4.3 ARIMA(1,0,1)

The ARIMA family of models combines autoregressive terms, differencing, and moving average terms. The general ARIMA$(p, d, q)$ model is defined by

$$\Phi(B)(1 - B)^d y_t = \Theta(B)\varepsilon_t, \tag{4.3}$$

where $B$ is the backshift operator, $By_t = y_{t-1}$, $\Phi(B)$ is a polynomial of degree $p$ in $B$ representing the autoregressive part, $(1 - B)^d$ is the differencing operator applied $d$ times, and $\Theta(B)$ is a polynomial of degree $q$ in $B$ representing the moving average part.

In our experiments we use ARIMA$(1, 0, 1)$, so there is one autoregressive term, no differencing ($d = 0$), and one moving average term. This model can capture both persistence from the past value and the effect of past shocks.

The implementation is straightforward:

```
arima_model = ARIMA(train, order=(1, 0, 1)).fit()
arima_forecast = arima_model.forecast(len(test))
```

## 4.4 Seasonal ARIMA: SARIMA(1,0,1)(0,1,1,7)

Many time series exhibit seasonality, meaning patterns that repeat at regular intervals. In our daily sentiment series, we observe weekly patterns (for example, weekends may have different sentiment than weekdays). To model such patterns, we use seasonal ARIMA, abbreviated SARIMA.

A SARIMA$(p, d, q)(P, D, Q)_s$ model incorporates both non-seasonal and seasonal components. The general form is

$$\Phi(B)\Phi_s(B^s)(1 - B)^d(1 - B^s)^D y_t = \Theta(B)\Theta_s(B^s)\varepsilon_t, \tag{4.4}$$

where $s$ is the seasonal period, $\Phi_s$ and $\Theta_s$ are seasonal polynomials of orders $P$ and $Q$, and $D$ is the seasonal differencing order.

In our project we use a model with non-seasonal order $(1, 0, 1)$, seasonal order $(0, 1, 1)$, and seasonal period $s = 7$ to capture weekly effects. This configuration is written as SARIMA$(1, 0, 1)(0, 1, 1)_7$. It was selected based on a grid search over small ranges of parameters, evaluated using RMSE on the test set.

We fit this model using the `SARIMAX` class:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

sarima_model = SARIMAX(
    train,
    order=(1, 0, 1),
    seasonal_order=(0, 1, 1, 7)
).fit(disp=False)

sarima_forecast = sarima_model.forecast(len(test))
```

## 4.5 Holt–Winters Exponential Smoothing

Holt–Winters exponential smoothing is another classical method for forecasting time series that exhibit trend and seasonality. It maintains three components: level, trend, and seasonality. For an additive model with period $s$, the update equations are

$$\ell_t = \alpha(y_t - s_{t-s}) + (1-\alpha)(\ell_{t-1} + b_{t-1}), \tag{4.5}$$
$$b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1}, \tag{4.6}$$
$$s_t = \gamma(y_t - \ell_t) + (1-\gamma)s_{t-s}, \tag{4.7}$$
$$\hat{y}_{t+h} = \ell_t + hb_t + s_{t+h-s(k+1)}, \tag{4.8}$$

where $\ell_t$ is the level, $b_t$ is the trend, $s_t$ is the seasonal component, $\alpha, \beta, \gamma$ are smoothing parameters, and $\hat{y}_{t+h}$ is the forecast $h$ steps ahead.

Intuitively, the model updates its estimate of the current level, trend, and seasonal pattern by blending new observations with previous estimates. The smoothing parameters control how quickly the model adapts to new information.

We implement Holt–Winters using `ExponentialSmoothing` from `statsmodels`:

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

hw_model = ExponentialSmoothing(
    train,
    trend="add",
    seasonal="add",
    seasonal_periods=7
)
hw_fit = hw_model.fit(optimized=True)
hw_forecast = hw_fit.forecast(len(test))
```

## 4.6 Baseline Forecasts

In addition to the above models, we use simple baselines for comparison. A naive forecast uses the last observed value of the training set as the prediction for all future time points.

A mean forecast uses the average of the training data. A rolling mean forecast uses a fixed-length moving average of the most recent observations.

These baselines are important because they define a lower bound: any reasonable model should perform better than a naive or mean forecast. If a complex model does not outperform these baselines, it may be overfitting or unnecessary.

# Chapter 5

# Stationarity Tests, Hyperparameter Selection, and Evaluation

## 5.1 Stationarity and Its Importance

Many time series models, especially AR and ARIMA-type models, assume that the series is stationary, meaning that its statistical properties (mean, variance, autocorrelation) do not change over time. Pure trend and strong non-stationarity can violate these assumptions and make such models unreliable.

To assess stationarity, we use two complementary statistical tests: the Augmented Dickey–Fuller (ADF) test and the KPSS test. Using both is helpful because they have different null hypotheses and failure modes.

## 5.2 Augmented Dickey–Fuller Test

The Augmented Dickey–Fuller test checks the null hypothesis that a time series has a unit root, which implies non-stationarity. The test is based on the regression

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \sum_{i=1}^{k} \phi_i \Delta y_{t-i} + \varepsilon_t, \tag{5.1}$$

where $\Delta y_t = y_t - y_{t-1}$ is the first difference, $t$ is time, and $k$ is the number of lagged differences included to control for autocorrelation in the residuals.

If the estimated coefficient $\gamma$ is significantly negative, the test rejects the null hypothesis of a unit root, suggesting that the series is stationary. A small $p$-value (e.g., below 0.05) indicates stationarity.

In our global sentiment series, we obtain a small ADF $p$-value, which suggests that the series is stationary or at least does not clearly exhibit a unit root.

## 5.3 KPSS Test

The KPSS test has the opposite null hypothesis: it tests whether a series is stationary around a mean or a deterministic linear trend. A large test statistic relative to critical values leads to rejection of the stationarity null in favor of the alternative that the series has a unit root.

In our case, the KPSS test indicates possible non-stationarity, especially related to level shifts and seasonal patterns. This result, combined with the ADF result, motivates using seasonal differencing in SARIMA models, where $(1 - B^s)$ is applied or equivalently $D > 0$.

## 5.4 SARIMA Hyperparameter Grid Search

Choosing ARIMA and SARIMA orders is a model selection problem. We perform a simple grid search over small ranges:

$$p, d, q \in \{0, 1\}, \qquad P, D, Q \in \{0, 1\}, \qquad s = 7.$$

For each combination of parameters $(p, d, q, P, D, Q)$, we fit a SARIMAX model on the training data, forecast the test period, and compute RMSE between predictions and actual values. We keep track of the RMSE and select the configuration with the lowest error.

This procedure identifies SARIMA$(1, 0, 1)(0, 1, 1)_7$ as the best-performing model among the tested configurations for our global sentiment series. This model balances complexity and forecasting performance.

## 5.5 Evaluation Metrics

The main evaluation metric for all models is the root mean squared error (RMSE). Given a set of test values $\{y_t\}_{t=1}^n$ and corresponding predictions $\{\hat{y}_t\}_{t=1}^n$, RMSE is defined as

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}\bigl(y_t - \hat{y}_t\bigr)^2}. \tag{5.2}$$

RMSE penalizes larger errors more heavily than small errors due to the square, and it has the same units as the original series. Lower RMSE indicates better predictive performance.

We also consider mean absolute error (MAE),

$$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n}|y_t - \hat{y}_t|, \tag{5.3}$$

which is easier to interpret directly as an average absolute deviation. RMSE is more sensitive to large mistakes, while MAE treats all errors linearly.

When comparing SARIMA models, we also look at information criteria such as the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC), which penalize model complexity. Lower AIC and BIC indicate a better trade-off between fit and parsimony. However, in the final comparison across very different model families, RMSE on a held-out test set is the decisive metric.

# Chapter 6

# Recurrent Neural Network Models

This chapter describes the recurrent neural network (RNN) models used in the project: LSTM, GRU, Bidirectional LSTM, and Stacked LSTM. We focus on how they work, why they are appropriate for time series, and how they are implemented with TensorFlow and Keras.

## 6.1 Packages for Deep Learning

The deep learning components rely on the TensorFlow ecosystem. We use `tensorflow` or `tensorflow-macos` depending on the hardware. The Keras API provides high-level classes such as `Sequential` for model composition and layers including `LSTM`, `GRU`, `Bidirectional`, `Dense`, and `Dropout`. Training is supported by callbacks such as `EarlyStopping` and `ReduceLROnPlateau`.

The Min–Max scaling step described earlier uses `sklearn.preprocessing.MinMaxScaler`, which is crucial to normalize inputs for stable training.

## 6.2 LSTM Cell: Equations and Intuition

Long Short-Term Memory (LSTM) networks address the problem of vanishing and exploding gradients in traditional RNNs by introducing a memory cell and gating mechanisms. At each time step $t$, the LSTM cell maintains a hidden state $h_t$ and a cell state $c_t$. Given the input $x_t$ and previous states $h_{t-1}$ and $c_{t-1}$, the update equations are:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f), \tag{6.1}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i), \tag{6.2}$$

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c), \tag{6.3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \tag{6.4}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o), \tag{6.5}$$

$$h_t = o_t \odot \tanh(c_t), \tag{6.6}$$

where $\sigma$ is the logistic sigmoid function, $\odot$ denotes element-wise multiplication, and $W_\cdot, b_\cdot$ are trainable weights and biases.

The forget gate $f_t$ decides how much of the previous cell state to keep. The input gate $i_t$ decides how much new information to write. The candidate cell state $\tilde{c}_t$ encodes new information, and the output gate $o_t$ controls how much of the cell state to expose as the hidden state. This structure allows the cell to retain information over long sequences and selectively update its memory.

## 6.3 LSTM Architecture Used

Our main LSTM architecture, implemented in `backend.py`, is a two-layer LSTM network with dropout regularization, followed by a dense output layer. The structure is:

- First LSTM layer with 50 units, ReLU activation, and `return_sequences=True`, which returns the full sequence for the next layer.
- Dropout layer with rate 0.2 to reduce overfitting.
- Second LSTM layer with 50 units, ReLU activation, and `return_sequences=False`, which returns only the final hidden state.
- Another dropout layer with rate 0.2.
- Dense output layer with 1 unit to predict the next sentiment value.

The model is compiled with mean squared error (MSE) as the loss function and Adam as the optimizer. MSE is appropriate because we are predicting a continuous numeric value.

## 6.4 GRU Cell: Equations and Differences from LSTM

Gated Recurrent Units (GRUs) simplify the LSTM architecture by using fewer gates and merging some states. A GRU cell operates with an update gate $z_t$ and a reset gate $r_t$. The equations are:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \tag{6.7}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r), \tag{6.8}$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h), \tag{6.9}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{6.10}$$

The update gate controls how much of the previous hidden state is kept, whereas the reset gate determines how much of the past to forget when computing the candidate hidden state $\tilde{h}_t$.

GRUs generally have fewer parameters than LSTMs and can be faster to train while still capturing long-term dependencies. In our implementation, the GRU architecture mirrors the LSTM one: two GRU layers with 50 units each, dropout layers in between, and a final dense layer.

## 6.5 Bidirectional LSTM

A Bidirectional LSTM processes input sequences in both forward and backward directions. In the forward pass, it reads the sequence from beginning to end; in the backward pass, it reads from end to beginning. At each time step, the forward and backward hidden states are concatenated to form a richer representation:

$$\overrightarrow{h_t} = \text{LSTM}_{\text{fwd}}(x_1, \dots, x_t), \qquad \overleftarrow{h_t} = \text{LSTM}_{\text{bwd}}(x_T, \dots, x_t),$$

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}].$$

While true online forecasting cannot use future inputs, bidirectional models are still useful when we train on fixed windows of past data: within each window, the backward pass can exploit structure inside the window.

The implementation uses the `Bidirectional` wrapper:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Bidirectional, Dense, Dropout

model = Sequential([
    Bidirectional(LSTM(50, activation='relu', return_sequences=True),
                  input_shape=(seq_length, 1)),
    Dropout(0.2),
    Bidirectional(LSTM(50, activation='relu', return_sequences=False)),
    Dropout(0.2),
    Dense(1)
])
```

## 6.6 Stacked LSTM

The stacked LSTM is a deeper architecture intended to increase representational power. It consists of three LSTM layers with decreasing numbers of units, each followed by dropout. The idea is that the first layer can capture low-level patterns, while subsequent layers learn more abstract features.

A typical configuration in our project is:

- First LSTM layer with 100 units and `return_sequences=True`.
- Dropout layer with rate 0.2.
- Second LSTM layer with 50 units and `return_sequences=True`.
- Dropout layer with rate 0.2.
- Third LSTM layer with 25 units and `return_sequences=False`.
- Dropout layer with rate 0.2.

- Dense layer with 1 output unit.

While deeper models can, in principle, capture more complex patterns, they are also more prone to overfitting, especially when data are limited. In our experiments, the stacked LSTM performs worse than simpler RNN models on the global series, which suggests that the extra capacity is not necessary and may overfit the noise.

## 6.7 Training Procedure and Callbacks

All RNN models share a common training setup. The loss function is mean squared error, and we also monitor mean absolute error as a secondary metric. The optimizer is Adam, which adapts the learning rate during training. We use a batch size of 32 and train for up to 50 epochs in experiments, though in the web app we may reduce the number of epochs to speed up training.

To prevent overfitting and speed up convergence, we use two callbacks:

- `EarlyStopping` monitors the validation loss and stops training if it does not improve for a specified number of epochs (patience). It can also restore the model weights to the best observed during training.
- `ReduceLROnPlateau` reduces the learning rate when the validation loss stops improving, allowing the optimizer to take smaller steps and refine the solution.

We use a validation split of 20% of the training sequences, so the model can be evaluated on data that are not used for weight updates.

## 6.8 Model Saving and Loading

For practical use in the application, we save trained models to disk and reload them when needed. For each country and model type, we store:

- `model.h5`: the Keras model weights and architecture.
- `scaler.pkl`: the fitted Min–Max scaler used for normalization.
- `metadata.json`: basic metadata such as the country name, model type, sequence length, and TensorFlow version.

These files are placed in directories such as:

```
models/lstm_models/<country_slug>/
models/gru_models/<country_slug>/
models/bilstm_models/<country_slug>/
models/stacked_lstm_models/<country_slug>/
```

When the application needs a forecast for a given country and RNN model type, it first tries to load an existing model. If a saved model is not found, it trains a new one on the full series, saves it, and then uses it for forecasting. This strategy avoids retraining models from scratch on every request.

# Chapter 7

# Application Architecture and Seasonal Analysis

## 7.1 Overview of the Urban Mood Forecaster App

The *Urban Mood Forecaster* is implemented as a Streamlit web application (`app.py`) that uses `backend.py` for data loading, preprocessing, modeling, and forecasting. The goal is to provide an interface that lets users select a country, view its sentiment history, run forecasts with different models, and obtain season-based travel recommendations.

The app is organized into three main modes: forecasting sentiment, finding the best season to visit a country, and listing the happiest countries for a chosen season.

## 7.2 Frontend Technologies and Styling

The frontend is built with Streamlit. We use `st.set_page_config` to set the page title and layout. To make the app visually appealing, we use custom CSS injected via `st.markdown` with the `unsafe_allow_html=True` flag. This CSS can set a background image and custom fonts for the title and headings.

For plots, we use Plotly, specifically `plotly.express` and `plotly.graph_objects`. Plotly provides interactive charts with tooltips, zooming, and panning, which are well-suited to time series visualization.

## 7.3 Data Loading and Caching

Loading multiple CSV files and constructing combined DataFrames can be slow, so we cache the data using Streamlit's `@st.cache_data` decorator:

```
@st.cache_data
def load_data_cached(data_directory):
    return load_sentiment_data(data_directory)
```

The first time the app runs, it loads the data from disk and builds the combined DataFrame. On subsequent runs within the same session, the cached result is reused, leading to a faster user experience.

## 7.4 Navigation and Mode Selection

At the top of the app, we provide three large buttons representing the three modes: "Forecast Sentiment", "Best Season to Visit", and "Happiest Countries by Season". When the user clicks a button, we store the selected option in `st.session_state.option`. The main body of the app then checks this option and renders the corresponding section.

This simple navigation scheme avoids complex multi-page logic while still giving users clear choices.

## 7.5 Forecast Sentiment Mode

In the forecasting mode, the user first selects a country from a dropdown list, which is populated with all unique country names in the data. The app calls `get_country_data` to obtain the daily sentiment series for the selected country and displays this historical series as a line chart.

The time series is split into training and test sets (80/20). The user then chooses a forecast horizon, such as 7 days, 30 days, or 365 days into the future. There is also an option to include recurrent neural network models in the comparison. If TensorFlow is not available on the machine, the app shows a warning and falls back to classical models only.

When the user clicks the "Run Forecast" button, the app calls a function such as `evaluate_models`, which fits AR, MA, ARIMA, SARIMA, and optionally RNN models on the training set, forecasts the test period, and computes RMSE for each model. It then presents a table of RMSE scores and highlights the best model.

Based on the best model name (for example, "SARIMA" or "Bidirectional LSTM"), the app chooses an internal model type string (such as `"sarima"` or `"bilstm"`) and passes it to `forecast_future`. This function refits the chosen model on the full historical series and produces predictions for the requested horizon, as well as confidence intervals.

The app displays a Plotly figure that overlays the historical series and future predictions. For SARIMA, the confidence intervals come from the model's `conf_int()` method. For RNN models, we approximate intervals by using a fixed percentage of the prediction standard deviation, acknowledging that this is not a full probabilistic treatment.

## 7.6 Best Season to Visit a Country

In the second mode, the focus shifts from day-level predictions to seasonal patterns. The user selects a country, and the app uses a function such as `get_seasonal_scores` to compute the average sentiment for each season.

We define seasons by mapping months to seasons:

- Winter: December, January, February.
- Spring: March, April, May.
- Summer: June, July, August.
- Autumn: September, October, November.

For the selected country, we group the daily series by season and compute the mean `SCORE` in each group. The resulting seasonal scores are sorted from highest to lowest. The app displays a table of seasonal averages, highlights the best season, and provides a textual recommendation such as “Summer is the happiest season to visit this country based on average sentiment.”

To make the pattern more intuitive, the app also shows a bar chart of seasonal scores using Plotly, with seasons on the x-axis and average sentiment on the y-axis.

## 7.7 Happiest Countries by Season

The third mode answers the question: given a season, which countries are happiest on average during that season? The user selects a season from a dropdown. The app then filters the combined dataset to include only months belonging to that season. It groups the data by `country` and computes the average `SCORE` for each country in that season.

The top $k$ countries (for example, top 15) are selected and displayed in a table with their average sentiment scores. This immediately shows which countries tend to have the highest sentiment during the chosen season.

An additional feature is the continent breakdown. Using `pycountry` and `pycountry_convert`, we map each country to a continent (for example, Europe, Asia, Africa). We count how many of the top countries belong to each continent and display the distribution in a bar chart. This visualization helps answer questions such as “Are the happiest winter destinations mostly in Europe or Asia?”.

# Chapter 8

# Results, Discussion, Limitations, and Conclusion

## 8.1 Model Comparison Results

Using the global average sentiment series as a benchmark, we evaluate all classical models and recurrent neural network models. The main outcome is that Bidirectional LSTM achieves the lowest RMSE, closely followed by the standard LSTM. GRU performs slightly worse but still outperforms classical baselines. The selected SARIMA$(1, 0, 1)(0, 1, 1)_7$ model achieves an RMSE that is higher than the best RNN models but better than naive, mean, and Holt–Winters baselines.

The deeper stacked LSTM overfits the data and obtains significantly higher RMSE than simpler RNNs. This confirms that more complex models are not always better; they need enough data and regularization to avoid memorizing noise.

These quantitative results suggest that recurrent models can capture non-linear temporal patterns in global sentiment that linear SARIMA models miss. However, the performance gap, while statistically significant, is not enormous, which raises questions about the trade-off between interpretability and predictive accuracy.

## 8.2 Forecast Visualizations

The forecast plots produced by the app are crucial for interpretation. For each country, the user sees a line plot with the historical daily sentiment series and an extension corresponding to the forecast horizon. The model's predictions appear as a curve, and the uncertainty is represented as a shaded band.

For SARIMA, the uncertainty band is derived from the model's estimated variance and directly corresponds to a confidence interval under the model assumptions. For RNNs, we approximate the band by taking a fixed percentage of the predicted standard deviation; this is a heuristic and not a full probabilistic forecast.

Visually, these plots allow users to see whether the model expects sentiment to remain stable, increase, or decrease, and how confident it is in those predictions. A narrow band suggests more confidence, while a wide band indicates more uncertainty.

## 8.3 Seasonal Sentiment and Travel Recommendations

The seasonal analysis reveals that many countries exhibit higher average sentiment in summer months, which aligns with intuitive expectations: better weather, vacation periods, and more outdoor activities often contribute to more positive mood. However, some countries show different patterns, with peaks in spring or autumn. The "Best Season to Visit" mode surfaces these non-obvious patterns and can suggest, for example, that autumn is the happiest season to visit a particular country.

The "Happiest Countries by Season" mode provides a different perspective. Instead of focusing on a single country, it reveals which countries have the highest sentiment in a chosen season and how they are distributed across continents. For instance, we may find that in winter, top countries are concentrated in certain regions, which could be due to climate, holidays, or cultural factors.

These recommendations are not meant to be definitive travel advice but rather illustrations of how sentiment data can be used to generate hypothesis and exploration guides.

## 8.4 Interpretation and Strengths

Our experiments suggest that public sentiment, as measured by TSGI, is relatively stable over long periods but exhibits meaningful short-term fluctuations. Weekly patterns and seasonal effects are clearly present, justifying the use of SARIMA with weekly seasonality and seasonal aggregation.

Recurrent neural networks, especially Bidirectional LSTM, achieve the best predictive accuracy for global sentiment and for many individual countries. Their ability to model non-linear relationships and long-term dependencies likely contributes to this performance. At the same time, SARIMA remains an attractive option because it is interpretable, computationally cheaper, and easier to explain in a policy context.

A major strength of the project is the integration of multiple components: a real, high-quality dataset with explicit well-being motivation; classical time series methods; modern deep learning; and an interactive, user-friendly web interface. The backend is structured so that models can be trained, saved, and reused in a modular way, and the frontend presents results in a way that non-experts can understand.

## 8.5 Limitations

Despite these strengths, there are important limitations. The most fundamental limitation is that sentiment scores are computed from Twitter users, who do not represent the entire population. Certain demographics, age groups, or regions may be underrepresented, and the behavior of Twitter users can change over time.

Our analysis is univariate per country: we model each country's sentiment series independently, without considering cross-country dependencies or external variables. Incorporating additional information such as major events, economic indicators, or mobility data could improve both predictive performance and interpretability.

Recurrent neural networks require substantial training time, especially without hardware acceleration such as GPUs. In a real-time system with many countries, this could become a bottleneck. Moreover, our confidence intervals for RNN forecasts are only approximate and not derived from a fully probabilistic model.

## 8.6 Future Work

Several extensions suggest themselves for future work. First, we could integrate Facebook Prophet or other modern forecasting libraries as additional baselines. Prophet is designed for time series with strong seasonality and could offer a robust alternative.

Second, we could build SARIMAX models that include exogenous variables, such as public holidays, economic indicators, or known events (e.g., elections, pandemics). This would help connect changes in sentiment to specific external drivers.

Third, we could explore transformer-based time series models and sequence-to-sequence architectures, which have shown promise in other domains. These models might capture even more subtle patterns at the cost of increased complexity.

Finally, we could extend the app to city-level data and incorporate changepoint detection algorithms to automatically annotate periods where sentiment trends change significantly, potentially corresponding to real-world shocks.

## 8.7 Conclusion

In this project, we developed *Urban Mood Forecaster*, a complete pipeline for analyzing and forecasting public sentiment using the Twitter Sentiment Geographical Index. Starting from raw yearly CSV files, we implemented data loading, cleaning, interpolation, and the construction of daily sentiment series for each country.

We evaluated a spectrum of models, from naive baselines and classical ARIMA-type models to deep recurrent neural networks. SARIMA with weekly seasonality provided a strong baseline, while Bidirectional LSTM and LSTM models achieved the best RMSE, illustrating the power of RNNs for capturing non-linear temporal patterns.

Beyond model performance, we turned the analysis into a practical tool: a Streamlit web application that allows users to explore sentiment history, obtain forecasts, and receive season-based travel recommendations. This system demonstrates how open data, time series methods, and interactive visualization can be combined to support reasoning about global mood dynamics.

Overall, our work highlights both the promise and the challenges of using social media sentiment as a proxy for subjective well-being. With further refinement, integration of additional data sources, and more sophisticated models, systems like Urban Mood Forecaster could become useful components of urban analytics and decision support, helping researchers, policymakers, and citizens better understand the emotional landscape of our world.

# Appendix: Python Packages and Environment

For completeness, we list the main Python packages used in this project.

The core scientific stack consists of `numpy` for numerical operations and `pandas` for data frames, time series handling, and resampling. Visualization is handled by `plotly.express` and `plotly.graph_objects`, which provide interactive plots.

Time series modeling uses `statsmodels`, specifically `ARIMA` and `SARIMAX` for ARIMA and SARIMA models, and `ExponentialSmoothing` for Holt–Winters exponential smoothing. Stationarity tests such as ADF and KPSS are also available in `statsmodels` and are used in exploratory analysis.

Machine learning utilities from `sklearn` include `mean_squared_error` for computing RMSE and `MinMaxScaler` for scaling RNN inputs. Deep learning relies on `tensorflow` or `tensorflow-macos` (for Apple Silicon), along with Keras components `Sequential`, `LSTM`, `GRU`, `Bidirectional`, `Dense`, `Dropout`, `EarlyStopping`, and `ReduceLROnPlateau`.

The web application is built with `streamlit`. Geographical utilities use `pycountry` and `pycountry_convert` to map country names to ISO codes and continents. Miscellaneous packages include `os`, `json`, `pickle`, `datetime`, `warnings`, `base64`, and `difflib` for file handling, serialization, and various helper functions.

All experiments and the app were run on a MacBook with an Apple M4 chip. RNN models use the TensorFlow build optimized for Apple Silicon. When TensorFlow is unavailable or fails to load, the application disables RNN models and falls back to classical time series models, ensuring that the app remains functional in a wide range of environments.