Smart Parking IoT – Occupancy & Demand Forecasting

Code Documentation: Data Processing and Machine Learning Methods

AAI-530: Data Science and the Internet of Things

University of San Diego - Applied Artificial Intelligence Program

Group 10

Ved Prakash Dwivedi & Dhrub Satyam

Instructor: Prof. Anamika Singh

February 2025

Repository: https://github.com/vedpd/AAI530-Group10-smart-parking-iot-forecasting

Dataset: Harvard Dataverse – DOI: 10.7910/DVN/YLWCSU

Contents

Abstract

This document presents the complete code documentation for the Smart Parking IoT – Occupancy and Demand Forecasting project submitted for AAI-530 at the University of San Diego. The codebase is organized across six Jupyter notebooks covering the full machine learning pipeline: data sampling, data overview, data cleaning and exploratory data analysis (EDA), machine learning baseline models, Long Short-Term Memory (LSTM) time-series forecasting, and LSTM inference on the complete dataset. The dataset comprises real-world IoT parking sensor readings from the SFpark system (Harvard Dataverse, DOI: 10.7910/DVN/YLWCSU), containing over 500,000 records across 841 parking segments in San Francisco. Two primary machine learning approaches are implemented and compared: (a) traditional ML baseline models including Linear Regression, Random Forest, Gradient Boosting, and Multi-Layer Perceptron, and (b) deep learning LSTM architectures including Basic LSTM, Stacked LSTM, and Bidirectional LSTM. All code is written in Python and follows reproducible data science practices with explicit random seeds, modular functions, and documented preprocessing pipelines.

Keywords: IoT, time-series forecasting, LSTM, parking occupancy, machine learning, smart city, Python, TensorFlow, scikit-learn

Introduction

This code document accompanies the technical report for the Smart Parking IoT project developed by Group 10 for AAI-530 at the University of San Diego. The project applies applied artificial intelligence and IoT data engineering techniques to a real-world urban parking dataset collected by the SFpark sensor infrastructure in San Francisco.

The codebase is structured as a pipeline of six sequential Jupyter notebooks, each addressing a distinct phase of the data science workflow. Together, these notebooks implement (a) data ingestion and sampling, (b) initial data understanding, (c) data cleaning and exploratory data analysis, (d) machine learning baseline models, (e) LSTM deep learning forecasting, and (f) full-dataset inference using the trained LSTM model.

The two primary machine learning methods compared in this project are traditional supervised learning models (as baselines) and Long Short-Term Memory (LSTM) neural networks (as the primary deep learning approach). All implementations use Python 3 with standard scientific computing libraries including pandas, NumPy, scikit-learn, and TensorFlow/Keras.

Pipeline Overview

Table 1
Project Notebook Pipeline

No.	Phase	File	Primary Purpose
01	Data Sampling	01_data_sampling.ipynb	30% stratified sample from 5M-row source dataset
02	Data Overview	02_data_overview.ipynb	Schema inspection, temporal range, sensor analysis
03	Cleaning & EDA	03_cleaning_and_eda.ipynb	Cleaning, feature engineering, EDA visualizations

No.	Phase	File	Primary Purpose
04	ML Baseline Models	04_ml_baseline_models.ipynb	Linear Regression, RF, GBM, MLP training
05	LSTM Forecasting	05_lstm_forecasting.ipynb	Basic, Stacked, Bidirectional LSTM training
06	LSTM Inference	06_lstm_inference.ipynb	Full-dataset inference and results export

Note. All notebooks are in the /notebooks directory of the project repository. Semicolon-delimited CSV format used throughout.

Data Processing Code

Data processing is implemented across three notebooks (01–03) and encompasses data sampling from the raw source, initial data inspection, and comprehensive cleaning with feature engineering. Each notebook follows a modular design with clearly documented code cells and inline explanations.

Notebook 01: Data Sampling (01_data_sampling.ipynb)

This notebook loads the 332 MB source dataset (sfpark_filtered_136_247_100taxis.csv) containing over 5 million rows of raw IoT sensor readings and produces a 30% stratified random sample for manageable downstream analysis. The sampling uses a fixed random seed (42) to ensure reproducibility.

Library Imports and Path Configuration

```
# Import libraries
import pandas as pd
import numpy as np
from pathlib import Path

# Define paths relative to project root
project_root = Path.cwd().parent
source_path  = project_root / 'data' / 'raw' / 'extracted' /
'sfpark_filtered_136_247_100taxis.csv'
output_path  = project_root / 'data' / 'raw' / 'smart_parking_full.csv'
```

Data Loading and 30% Sampling

```
# Load the full 100-taxi SFpark dataset (5M+ rows, semicolon-delimited)
df_full = pd.read_csv(source_path, sep=';')

# Reproducible 30% random sample
df_sample = df_full.sample(frac=0.3, random_state=42)

# Persist the sampled dataset for downstream notebooks
df_sample.to_csv(output_path, sep=';', index=False)

# Verification
df_verify = pd.read_csv(output_path, sep=';')
print(f'Sampled rows: {len(df_verify):,}  |  Columns: {df_verify.shape[1]}')
```

Notebook 02: Data Overview (02_data_overview.ipynb)

This notebook performs initial data inspection to understand dataset schema, temporal coverage, and sensor structure. Key tasks include loading the sampled dataset, parsing the timestamp column, analysing time intervals, computing occupancy rates, and characterizing the 841 parking segments.

Dataset Loading and Timestamp Parsing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path

# Load sampled dataset (semicolon delimiter)
df = pd.read_csv(data_path, sep=';')

# Convert timestamp to datetime64
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Temporal coverage
print(f'Time range: {df["timestamp"].min()} to {df["timestamp"].max()}')
print(f'Span: {(df["timestamp"].max() - df["timestamp"].min()).days} days')
```

Occupancy Rate Calculation and Sensor Analysis

```
# Calculate occupancy rate (primary target variable)
df['occupancy_rate'] = df['occupied'] / df['capacity']

# Sensor column identification
observed_cols = [c for c in df.columns if c.startswith('observed')]
diff_cols     = [c for c in df.columns if c.startswith('diff')]

# Segment-level summary
print(f'Unique segments: {df["segmentid"].nunique()}')
print(f'Avg records/segment: {len(df) / df["segmentid"].nunique():.1f}')

# Missing value report
missing = df.isnull().sum()
print(missing[missing > 0])
```

Notebook 03: Data Cleaning and EDA (03_cleaning_and_eda.ipynb)

This notebook constitutes the core data engineering phase. It applies systematic data cleaning rules, engineers a rich set of temporal and domain-specific features, and produces comprehensive exploratory visualizations. The cleaned, feature-enriched dataset is saved as cleaned_smart_parking_data.csv for use by modeling notebooks.

Data Cleaning Pipeline

```
# 1. Parse timestamp
df['timestamp'] = pd.to_datetime(df['timestamp'])

# 2. Fill missing sensor readings with 0 (no observation = no vehicle detected)
for col in observed_cols + diff_cols:
    df[col] = df[col].fillna(0)

# 3. Remove duplicate records
df = df.drop_duplicates()

# 4. Fix over-capacity anomalies (sensor errors)
df.loc[df['occupied'] > df['capacity'], 'occupied'] = \
    df.loc[df['occupied'] > df['capacity'], 'capacity']

# 5. Fix zero-capacity records with non-zero occupancy
mask = (df['capacity'] == 0) & (df['occupied'] > 0)
df.loc[mask, 'capacity'] = df.loc[mask, 'occupied']
```

Temporal Feature Engineering

```
# Calendar features
df['hour']         = df['timestamp'].dt.hour
df['day_of_week']  = df['timestamp'].dt.dayofweek   # 0=Mon, 6=Sun
df['day_of_month'] = df['timestamp'].dt.day
df['month']        = df['timestamp'].dt.month
df['quarter']      = df['timestamp'].dt.quarter

# Cyclical (sinusoidal) encodings for ML continuity
df['hour_sin']  = np.sin(2 * np.pi * df['hour']        / 24)
df['hour_cos']  = np.cos(2 * np.pi * df['hour']        / 24)
df['day_sin']   = np.sin(2 * np.pi * df['day_of_week'] / 7)
df['day_cos']   = np.cos(2 * np.pi * df['day_of_week'] / 7)
df['month_sin'] = np.sin(2 * np.pi * df['month']       / 12)
df['month_cos'] = np.cos(2 * np.pi * df['month']       / 12)

# Binary indicators
df['is_weekend']   = (df['day_of_week'] >= 5).astype(int)
df['is_rush_hour'] = ((df['hour'].between(7, 9)) |
                      (df['hour'].between(16, 18))).astype(int)
```

```
# Time period categories
df['time_period'] = pd.cut(df['hour'],
                           bins=[-1, 6, 12, 18, 24],
                           labels=['Night', 'Morning', 'Afternoon', 'Evening'])
```

Parking-Specific Feature Engineering

```
# Core occupancy metrics
df['occupancy_rate']   = df['occupied'] / df['capacity']
df['occupancy_rate']   = df['occupancy_rate'].fillna(0)
df['available_spaces'] = df['capacity'] - df['occupied']

# Occupancy level categories
df['occupancy_level'] = pd.cut(df['occupancy_rate'],
                               bins=[-0.1, 0.25, 0.5, 0.75, 1.1],
                               labels=['Low', 'Medium', 'High', 'Full'])

# Sensor reliability metrics
df['total_observed']     = df[observed_cols].fillna(0).sum(axis=1)
df['avg_observed']       = df[observed_cols].fillna(0).mean(axis=1)
df['sensor_variance']    = df[observed_cols].fillna(0).var(axis=1)
df['sensor_reliability'] = df[observed_cols].notna().sum(axis=1) /
len(observed_cols)
```

Following cleaning and feature engineering, the dataset is visualized through a 3x3 grid of matplotlib plots including occupancy rate distribution, capacity distribution, hourly and daily patterns, occupancy level pie chart, scatter of capacity vs. occupied spaces, and sensor reliability histogram.

Machine Learning Method 1: Baseline Models

Notebook 04 (04_ml_baseline_models.ipynb) implements four traditional supervised learning baseline models to predict parking occupancy rates. These models serve as benchmarks against which the LSTM deep learning approach is evaluated. Data leakage is explicitly addressed by excluding the 'occupied' column from the feature set, ensuring that models learn from temporal and sensor patterns rather than from the target variable's direct determinants.

Feature Selection and Data Preparation

```
from sklearn.model_selection import train_test_split, TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor

# Feature set (NOTE: 'occupied' excluded to prevent data leakage)
feature_cols = ['hour', 'day_of_week', 'month', 'capacity',
                'hour_sin', 'hour_cos', 'day_sin', 'day_cos',
                'month_sin', 'month_cos']

X = df[feature_cols].fillna(df[feature_cols].median())
y = df['occupancy_rate']

# Remove rows with NaN in target
valid = ~(X.isnull().any(axis=1) | y.isnull())
X, y  = X[valid], y[valid]

# 80/20 train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Feature scaling (required for Linear Regression and MLP)
scaler         = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled  = scaler.transform(X_test)
```

Baseline Model 1: Linear Regression

```
# Train Linear Regression (uses scaled features)
lr_model = LinearRegression()
lr_model.fit(X_train_scaled, y_train)
y_pred_lr = lr_model.predict(X_test_scaled)
```

```python
# Evaluation metrics
lr_mse = mean_squared_error(y_test, y_pred_lr)
lr_mae = mean_absolute_error(y_test, y_pred_lr)
lr_r2  = r2_score(y_test, y_pred_lr)

# Feature coefficients for interpretability
feature_importance_lr = pd.DataFrame({
    'feature':     feature_cols,
    'coefficient': lr_model.coef_
}).sort_values('coefficient', key=abs, ascending=False)
```

Baseline Model 2: Random Forest Regressor

```python
# Train Random Forest (uses unscaled features, tree-based)
rf_model = RandomForestRegressor(
    n_estimators=100,
    random_state=42,
    n_jobs=-1              # parallelise across all CPU cores
)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)

rf_mse = mean_squared_error(y_test, y_pred_rf)
rf_mae = mean_absolute_error(y_test, y_pred_rf)
rf_r2  = r2_score(y_test, y_pred_rf)

# Gini-based feature importance
feature_importance_rf = pd.DataFrame({
    'feature':    feature_cols,
    'importance': rf_model.feature_importances_
}).sort_values('importance', ascending=False)
```

Baseline Model 3: Gradient Boosting Regressor

```python
# Train Gradient Boosting (sequential ensemble of decision trees)
gb_model = GradientBoostingRegressor(
    n_estimators=100,
    random_state=42
)
gb_model.fit(X_train, y_train)
y_pred_gb = gb_model.predict(X_test)

gb_mse = mean_squared_error(y_test, y_pred_gb)
gb_mae = mean_absolute_error(y_test, y_pred_gb)
gb_r2  = r2_score(y_test, y_pred_gb)
```

Baseline Model 4: Multi-Layer Perceptron

```python
# Train Neural Network (MLP Regressor - shallow feed-forward)
nn_model = MLPRegressor(
    hidden_layer_sizes=(100, 50),   # two hidden layers
    activation='relu',
    solver='adam',
    max_iter=500,
    random_state=42
)
nn_model.fit(X_train_scaled, y_train)
y_pred_nn = nn_model.predict(X_test_scaled)

nn_mse = mean_squared_error(y_test, y_pred_nn)
nn_mae = mean_absolute_error(y_test, y_pred_nn)
nn_r2  = r2_score(y_test, y_pred_nn)
```

Time Series Cross-Validation

```
# Sort by timestamp to preserve temporal ordering
df_sorted = df.sort_values('timestamp')
X_ts = df_sorted[feature_cols].fillna(df_sorted[feature_cols].median())
y_ts = df_sorted['occupancy_rate']

tscv = TimeSeriesSplit(n_splits=5)
rf_ts_scores = []

for train_idx, test_idx in tscv.split(X_ts):
    X_tr, X_te = X_ts.iloc[train_idx], X_ts.iloc[test_idx]
    y_tr, y_te = y_ts.iloc[train_idx], y_ts.iloc[test_idx]
    rf_ts = RandomForestRegressor(n_estimators=50, random_state=42, n_jobs=-1)
    rf_ts.fit(X_tr, y_tr)
    rf_ts_scores.append(r2_score(y_te, rf_ts.predict(X_te)))

print(f'Time-series CV R2: {np.mean(rf_ts_scores):.4f} +/-
{np.std(rf_ts_scores):.4f}')
```

Baseline Model Performance Summary

Table 2 summarises the performance metrics for all four baseline models evaluated on the 20% held-out test set. Metrics include Mean Squared Error (MSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2).

Table 2
Baseline ML Model Performance on 20% Test Set

Model	MSE	MAE	R^2	Notes
Linear Regression	0.0773	0.2300	0.082	Interpretable baseline
Random Forest	0.0672	0.2094	0.202	Best baseline
Gradient Boosting	0.0680	0.2126	0.192	Competitive
MLP Neural Network	0.0688	0.2140	0.184	Shallow deep learning

Note. MSE = Mean Squared Error; MAE = Mean Absolute Error; R^2 = Coefficient of Determination. All models trained on 80% chronological split. 'occupied' feature excluded to prevent data leakage.

Machine Learning Method 2: LSTM Time-Series Forecasting

Notebooks 05 and 06 implement and deploy Long Short-Term Memory (LSTM) neural networks for sequential time-series forecasting of parking occupancy rates. LSTM networks are a specialized class of Recurrent Neural Networks (RNNs) designed to capture long-range temporal dependencies through learnable gating mechanisms. Three LSTM architectures are implemented and compared: Basic LSTM, Stacked LSTM, and Bidirectional LSTM.

Data Preprocessing for LSTM

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,
ModelCheckpoint
from sklearn.preprocessing import MinMaxScaler

np.random.seed(42)
tf.random.set_seed(42)

# Feature set for LSTM (includes cyclical encodings)
features = ['occupancy_rate', 'hour', 'day_of_week', 'month',
            'is_weekend', 'is_rush_hour',
            'hour_sin', 'hour_cos', 'day_sin', 'day_cos']

# Aggregate to hourly granularity per segment
df_hourly = df.groupby(
    [pd.Grouper(key='timestamp', freq='H'), 'segmentid']
).agg({'occupancy_rate': 'mean', 'capacity': 'mean',
       'hour': 'first', 'day_of_week': 'first', 'month': 'first',
       'is_weekend': 'first', 'is_rush_hour': 'first',
       'hour_sin': 'first', 'hour_cos': 'first',
       'day_sin': 'first', 'day_cos': 'first'}).reset_index()

# Select most-data-rich segment for single-segment LSTM
target_segment = df_hourly['segmentid'].value_counts().index[0]
df_seg = df_hourly[df_hourly['segmentid'] ==
target_segment].sort_values('timestamp')

# MinMax scaling to [0, 1]
scaler_features = MinMaxScaler(feature_range=(0, 1))
scaler_target   = MinMaxScaler(feature_range=(0, 1))
features_scaled = scaler_features.fit_transform(df_seg[features])
target_scaled   = scaler_target.fit_transform(df_seg[['occupancy_rate']])
```

Sequence Generation

```
def create_sequences(data, target, sequence_length, forecast_horizon=1):
    '''
    Convert time-series array into (X, y) pairs for LSTM training.
    Args:
        data:             2D array of scaled features [timesteps, features]
        target:           2D array of scaled target   [timesteps, 1]
        sequence_length:  look-back window (hours)
        forecast_horizon: steps ahead to forecast
    Returns:
        X: (N, sequence_length, n_features)
        y: (N, forecast_horizon)
    '''
    X, y = [], []
    for i in range(len(data) - sequence_length - forecast_horizon + 1):
        X.append(data[i : i + sequence_length])
        y.append(target[(i + sequence_length):(i + sequence_length +
forecast_horizon)])
    return np.array(X), np.array(y)


SEQUENCE_LENGTH  = 24  # 24-hour look-back window
FORECAST_HORIZON =  1  # 1-hour ahead prediction


X, y = create_sequences(features_scaled, target_scaled,
                        SEQUENCE_LENGTH, FORECAST_HORIZON)


# Chronological 70/15/15 train/val/test split
n         = len(X)
train_end = int(n * 0.70)
val_end   = int(n * 0.85)
X_train, y_train = X[:train_end],        y[:train_end]
X_val,   y_val   = X[train_end:val_end], y[train_end:val_end]
X_test,  y_test  = X[val_end:],          y[val_end:]
```

LSTM Architecture 1: Basic LSTM

```
def build_basic_lstm(input_shape, output_dim=1):
    '''Single-layer LSTM for 1-step occupancy forecasting.'''
    model = Sequential([
        LSTM(50, activation='relu', input_shape=input_shape,
             return_sequences=False),
        Dropout(0.2),
        Dense(25, activation='relu'),
        Dropout(0.1),
        Dense(output_dim, activation='sigmoid')  # sigmoid: output in [0, 1]
    ])
    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='mse',
        metrics=['mae']
    )
    return model


input_shape = (SEQUENCE_LENGTH, X_train.shape[2])
basic_lstm  = build_basic_lstm(input_shape, FORECAST_HORIZON)
```

LSTM Architecture 2: Stacked LSTM

```
def build_stacked_lstm(input_shape, output_dim=1):
    '''Two-layer stacked LSTM for deeper temporal representation.'''
    model = Sequential([
        LSTM(64, activation='relu', input_shape=input_shape,
             return_sequences=True),   # pass sequences to next LSTM
        Dropout(0.2),
        LSTM(32, activation='relu', return_sequences=False),
        Dropout(0.2),
        Dense(25, activation='relu'),
        Dropout(0.1),
        Dense(output_dim, activation='sigmoid')
    ])
    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='mse',
        metrics=['mae']
    )
    return model

stacked_lstm = build_stacked_lstm(input_shape, FORECAST_HORIZON)
```

LSTM Architecture 3: Bidirectional LSTM

```
def build_bidirectional_lstm(input_shape, output_dim=1):
    '''Bidirectional LSTM processes sequences forward AND backward,
    capturing dependencies from both past and future context within
    the input window to improve performance on structured patterns.
    '''
    model = Sequential([
        Bidirectional(
            LSTM(64, activation='relu', return_sequences=True),
            input_shape=input_shape
        ),
        Dropout(0.2),
        Bidirectional(
            LSTM(32, activation='relu', return_sequences=False)
        ),
        Dropout(0.2),
        Dense(25, activation='relu'),
        Dropout(0.1),
        Dense(output_dim, activation='sigmoid')
    ])
    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='mse',
        metrics=['mae']
    )
    return model

bidirectional_lstm = build_bidirectional_lstm(input_shape, FORECAST_HORIZON)
```

Training Configuration and Callbacks

```
# Shared training callbacks (applied to all three LSTM models)
def get_callbacks(model_name):
    return [
        EarlyStopping(
            monitor='val_loss',
            patience=10,
            restore_best_weights=True
        ),
        ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.5,
            patience=5,
            min_lr=1e-5
        ),
        ModelCheckpoint(
            f'best_{model_name}.h5',
            monitor='val_loss',
            save_best_only=True
        ),
    ]

# Example: training the Stacked LSTM
history_stacked = stacked_lstm.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
    batch_size=32,
    callbacks=get_callbacks('stacked_lstm'),
    verbose=1
)
```

LSTM Evaluation and Inverse Transform

```
def evaluate_lstm(model, X_test, y_test, scaler_target):
    '''Evaluate LSTM model and return metrics on original scale.'''
    y_pred_scaled = model.predict(X_test)
    # Inverse transform to occupancy rate scale [0, 1]
    y_test_orig = scaler_target.inverse_transform(y_test.reshape(-1, 1))
    y_pred_orig = scaler_target.inverse_transform(y_pred_scaled.reshape(-1, 1))
    mse = mean_squared_error(y_test_orig, y_pred_orig)
    mae = mean_absolute_error(y_test_orig, y_pred_orig)
    r2  = r2_score(y_test_orig, y_pred_orig)
    return mse, mae, r2, y_test_orig, y_pred_orig

mse_basic,   mae_basic,   r2_basic,   *_ = evaluate_lstm(basic_lstm,
X_test, y_test, scaler_target)
mse_stacked, mae_stacked, r2_stacked, *_ = evaluate_lstm(stacked_lstm,
X_test, y_test, scaler_target)
mse_bi,      mae_bi,      r2_bi,      *_ = evaluate_lstm(bidirectional_lstm,
X_test, y_test, scaler_target)
```

LSTM Performance Summary

Table 3 presents the performance of the three LSTM architectures on the 15% chronological test set, alongside the best-performing baseline model (Random Forest) for direct comparison.

Table 3

LSTM vs. Baseline Performance Comparison

Model	MSE	MAE	R²	Architecture
Random Forest (best baseline)	0.0672	0.2094	0.202	100 trees, Gini importance
Basic LSTM	–	–	–	50 units, Dropout 0.2, Sigmoid output
Stacked LSTM	–	–	–	64+32 units, 2 layers, Dropout
Bidirectional LSTM	–	–	–	64+32 BiLSTM units, Dropout

Note. LSTM metrics (–) are populated from actual training runs in 05_lstm_forecasting.ipynb. All splits are chronological to prevent temporal leakage.

LSTM Model Inference on Complete Dataset

Notebook 06 (06_lstm_inference.ipynb) loads the persisted best LSTM model and applies it to the complete hourly-aggregated dataset for the target parking segment. The inference pipeline generates predicted occupancy rates for every valid 24-hour look-back window, assigns train/test split labels, and exports results as structured CSV files for dashboard integration.

Model Loading and Inference Pipeline

```
import joblib
from tensorflow.keras.models import load_model

# Load saved model artefacts
with open(models_dir / 'best_lstm_metadata.pkl', 'rb') as f:
    lstm_metadata = pickle.load(f)

best_model      = load_model(models_dir /
f"best_{lstm_metadata['model_name']}_model.h5")
scaler_features = joblib.load(models_dir / 'lstm_feature_scaler.pkl')
scaler_target   = joblib.load(models_dir / 'lstm_target_scaler.pkl')

SEQUENCE_LENGTH = lstm_metadata['sequence_length']
features        = lstm_metadata['features']
target_segment  = lstm_metadata['target_segment']
```

```
def create_inference_sequences(data, sequence_length):
    '''Create overlapping windows for full-dataset inference.'''
    return np.array([data[i : i + sequence_length]
                     for i in range(len(data) - sequence_length + 1)])

# Scale features using fitted scaler (no re-fitting)
features_scaled = scaler_features.transform(df_inference[features])
X_inference     = create_inference_sequences(features_scaled, SEQUENCE_LENGTH)

# Batch inference
predictions_scaled = best_model.predict(X_inference, verbose=0)

# Inverse transform to original occupancy rate scale
predictions = scaler_target.inverse_transform(predictions_scaled)
```

```
# Align predictions with original DataFrame rows
results_df = df_target.copy()
results_df['predicted_occupancy_rate'] = np.nan
```

```
for i, idx in enumerate(range(SEQUENCE_LENGTH, SEQUENCE_LENGTH +
len(predictions))):
    if idx < len(results_df):
        results_df.iloc[idx,
            results_df.columns.get_loc('predicted_occupancy_rate')] =
predictions[i, 0]

# Add train/test split labels (80/20 chronological)
split_point = int(len(results_df) * 0.8)
results_df['data_split'] = 'train'
results_df.iloc[split_point:,
    results_df.columns.get_loc('data_split')] = 'test'

# Final metrics on valid predictions
valid = results_df.dropna(subset=['predicted_occupancy_rate'])
mse = mean_squared_error(valid['occupancy_rate'],
valid['predicted_occupancy_rate'])
mae = mean_absolute_error(valid['occupancy_rate'],
valid['predicted_occupancy_rate'])
r2  = r2_score(valid['occupancy_rate'], valid['predicted_occupancy_rate'])

# Persist inference results
results_df.to_csv(output_dir / 'lstm_inference_complete_results.csv', index=False)
valid.to_csv(output_dir / 'lstm_predictions_only.csv', index=False)
```

The inference module also generates four diagnostic visualizations: (a) a time-series comparison of actual vs. predicted occupancy over the first 500 time points, (b) a scatter plot of actual vs. predicted values, (c) a histogram of prediction residuals, and (d) a scatter plot differentiating train vs. test set performance.

Software Dependencies and Environment

All notebooks were developed and tested in Python 3.10. The following packages are required and are specified in requirements.txt in the project repository.

```
# requirements.txt - core dependencies
pandas>=1.5.0
numpy>=1.23.0
matplotlib>=3.6.0
seaborn>=0.12.0
scikit-learn>=1.2.0
tensorflow>=2.11.0
keras>=2.11.0
plotly>=5.11.0
statsmodels>=0.13.0
joblib>=1.2.0
pathlib
pickle
```

All library imports are placed at the top of each notebook in accordance with PEP 8 conventions. Warning suppression (warnings.filterwarnings('ignore')) is applied consistently to reduce output verbosity. Random seeds are set globally (np.random.seed(42), tf.random.set_seed(42)) to ensure reproducibility of all stochastic processes.

References

Dwivedi, V. P., & Satyam, D. (2024). Smart parking IoT – occupancy & demand forecasting [GitHub repository]. University of San Diego. https://github.com/vedpd/AAI530-Group10-smart-parking-iot-forecasting

Harvard Dataverse. (2023). SFpark pilot study sensor data (Version 2) [Dataset]. Harvard Dataverse. https://doi.org/10.7910/DVN/YLWCSU

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. https://www.jmlr.org/papers/v12/pedregosa11a.html

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., & Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5–32. https://doi.org/10.1023/A:1010933404324

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. The Annals of Statistics, 29(5), 1189–1232. https://doi.org/10.1214/aos/1013203451

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Rio, J. F., Wiebe, M., Peterson, P., & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585, 357–362. https://doi.org/10.1038/s41586-020-2649-2

McKinney, W. (2010). Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a