```
#%% Rayden Dodd Term Project DashBoard
import pandas as pd
import numpy as np
from matplotlib.ticker import FuncFormatter
from scipy import stats            # QQ-plot
import plotly.express as px
import plotly.figure_factory as ff
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import plotly.graph_objects as go
import dash
from dash import Dash, dcc, html, Input, Output, State, dash_table
#%% Set Globabl values
np.set_printoptions(precision=2, floatmode='fixed')
title_font = {'family': 'serif', 'color': 'blue', 'size': 18}
label_font = {'family': 'serif', 'color': 'darkred', 'size': 14}
fmt_millions=FuncFormatter(lambda x,_:f'{x/1e6:.2f}M')
fmt_thousands=FuncFormatter(lambda x,_:f'{x/1e3:.2f}K')
fmt_billions=FuncFormatter(lambda x,_:f'{x/1e9:.2f}B')
periods = ['AM Peak (Open-9:30am)',
           'Midday (9:30am-3pm)',
           'PM Peak (3pm-7pm)',
           'Evening (7pm-12am)',
           'Late Night (12am-Close)']

#%% Upload dataset
dataset = 'Entries_by_Year_Full_Data_data/Entries_by_Year_Full_Data_data.csv'
mlb_schedule_dataset = 'Entries_by_Year_Full_Data_data/mlb-2024-orig.csv'

df = pd.read_csv(dataset)
mlb_schedule = pd.read_csv(mlb_schedule_dataset)
print(df.head().to_string())
print(mlb_schedule.head().to_string())
#%% Clean Data set
columns_to_remove = ["Avg Daily Tapped Entries", "SUM([NonTapped
Entries])/COUNTD([Date])", "Avg Daily Entries", "Avg Daily Entries (copy)", "Avg
Daily Entries Non-Rounded", "Avg Daily Exits Rounded", "Avg Daily NonTapped
Entries", "Data Range", "DAY_TYPE", "Tableau X 98", "Tableau Y 98", "Year", "Year
(Discrete)"]
df = df.drop(columns=[col for col in columns_to_remove if col in df.columns])
# Clean up negative entries
cols = ['Entries',
        'NonTapped Entries',
        'Tap Entries',
        'Exits',
        'Tap Exits', 'NonTapped Exits']
df["Holiday_Cat"] = df["Holiday"]
# Holiday to True or False
df['Holiday'] = df['Holiday'].map({'Yes': True, 'No': False})
# Clean up dates
df["Date"]      = pd.to_datetime(df["Date"], format='%m/%d/%Y %I:%M:%S %p')
df.rename(columns={'Year of Date': 'Year'}, inplace=True)
df["Year"]        = df["Date"].dt.year
df["Month"]       = df["Date"].dt.month_name()
df["Day of Week"] = df["Date"].dt.day_name()
df['NonTapped Exits'] = (df['Exits'] - df['Tap Exits']).clip(lower=0)
df["Year_Cat"] = df["Date"].dt.year.astype(str).astype("category")
# Clean data set for PCA pre compute pre clean
clean = df.copy()
```

```
clean[cols] = df[cols].clip(lower=0)
numeric_cols   = clean.select_dtypes(include=[np.number]).columns
X              = clean[numeric_cols].dropna().values
X_scaled       = StandardScaler().fit_transform(X)

pca            = PCA().fit(X_scaled)
explained      = pca.explained_variance_ratio_
cum_explained  = np.cumsum(explained)
pcs_global    = pca.transform(X_scaled)
print(df.head().to_string())


#%% Dash
# ----------------------------------------------------------------
# DASH APP HEADER
# ----------------------------------------------------------------
app = Dash(__name__, suppress_callback_exceptions=True)
server = app.server

# Layout
header_bar = html.Div(
    id='app-header',
    children=[
        html.H1("Metro Ridership Dashboard", className='app-title'),
        html.Img(src='/assets/wmata_logo.png', className='app-logo')
    ]
)

app.layout = html.Div([
    # Global Vars
    dcc.Store(id="cleaning-store", data={"methods": ["clip"], "threshold": 3.0}),
    dcc.Store(id="transform-store",  data={}),

    # Header + tabs
    header_bar,
    dcc.Tabs(id='main-tabs', value='tab-home', parent_className='custom-tabs',
className='custom-tabs',
        children=[
        dcc.Tab(label='Home',                     value='tab-home'),
        dcc.Tab(label='Cleaning & Outliers',      value='tab-clean'),
        dcc.Tab(label='Dimensionality Reduction', value='tab-pca'),
        dcc.Tab(label='Normality Tests',          value='tab-normality'),
        dcc.Tab(label='Data Transformation',      value='tab-transform'),
        dcc.Tab(label='Single-Variable Plots',    value='tab-single'),
        dcc.Tab(label='Multi-Variable Plots',     value='tab-multi'),
        dcc.Tab(label='Statistics',               value='tab-stats')
    ]),

    html.Div(id='tab-content', style={'padding': '25px'})
])
# Render Tabs
@app.callback(Output('tab-content', 'children'),
              Input('main-tabs', 'value'))
def render_tab(active):
    return {
        'tab-home'     : home_layout,
        'tab-clean'    : clean_layout,
        'tab-pca'      : pca_layout,
        'tab-normality': normal_layout,
```

```
        'tab-transform': transform_layout,
        'tab-single'   : dashboard_layout,
        'tab-multi'    : multi_layout,
        'tab-stats'    : stats_layout
    }[active]

# ------------------------------------------------------------------
# TABS
# ------------------------------------------------------------------


# HOME
# ------------------------------------------------------------------
home_layout = html.Div([
    html.H2("Welcome!", style={"color": "#5c79ae"}),
    dcc.Markdown("""
                    **Dataset**
                    WMATA (Washington Metropolitan Area Transit Authority) rail-
ridership records from **January 2012 to June 2025**.
                    Each row contains **Date, Station Name, Time of Day, Entries,
Exits**, and—from **2023 on**—split counts for **Non-Tapped Entries (un-paid)** and
**Tapped Entries (paid)**.

                    **Navigation:**
                     • *Data Cleaning* – choose clipping / Z-score
                     • *Cleaning & Outlier Detection* – Clipping & z-score &
IQR
                     • *Dimensionality Reduction* – PCA & Cumulative Explained
Variance
                     • *Normality Tests* – Hist+KDE & QQ-plot
                     • *Data Transformation* – log10, sqrt(x), z-score, minmax

                     • *Single-Variable Plots* – Bar, Pie, Line, Area, Box,
Histogram+KDE, Violin, Strip, Rug
                     • *Multi-Variable Plots* –  Scatter Plot Matrix, Join
Plots, 3d scatter, Contour plots
                     • *Statistics* – Dataframe Statistics by Feature and by
Station
                        """),

    html.Button("Download Dataset", id="download-button", n_clicks=0,
style={"margin-top": "15px"}),
    dcc.Download(id="download-data"),

    html.Img(src="/assets/cherry_blossom.png", className="home-img-left"),
    html.Img(src="/assets/metro_train.jpg", className="home-img-right")
])

@app.callback(
    Output("download-data", "data"),
    Input("download-button", "n_clicks"),
    prevent_initial_call=True
)
def download_csv(n_clicks):
    df = pd.read_csv("assets/mlb-2024-orig.csv")
    return dcc.send_data_frame(df.to_csv, "wmata_ridership.csv", index=False)

# Data Cleaning & Outlier Detetction
# ------------------------------------------------------------------
```

```
clean_layout = html.Div([
    html.H3("Cleaning & Outlier Options",
style={'fontFamily':'serif','color':'blue'}),

    dcc.Checklist(
        id='cleaning-methods',
        options=[
            {'label': 'Clip Negative Values', 'value': 'clip'},
            {'label': 'Z-Score Outlier Removal', 'value': 'zscore'}
        ],
        value=['clip'],
        labelStyle={'display': 'inline-block', 'margin-right': '20px'}
    ),

    html.Div([
        html.Label("Z-Score Threshold:", style={'margin-top': '10px'}),
        dcc.Slider(
            id='zscore-threshold',
            min=1.0,
            max=5.0,
            step=0.1,
            value=3.0,
            marks={i: str(i) for i in range(1, 6)}
        )
    ]),
    # BEFORE box-plot
    html.Div(
        dcc.Loading(
            type="circle",
            color="#5c79ae",
            children=dcc.Graph(id="box-before")
        ),
        style={"display": "inline-block", "width": "48%", "verticalAlign": "top"}
    ),

    # AFTER box-plot
    html.Div(
        dcc.Loading(
            type="circle",
            color="#5c79ae",
            children=dcc.Graph(id="box-after")
        ),
        style={"display": "inline-block", "width": "48%", "verticalAlign": "top"}
    ),

], style={'margin-bottom': '20px'})

@app.callback(
    Output("cleaning-store", "data"),
    Input("cleaning-methods",  "value"),
    Input("zscore-threshold",  "value"),
    prevent_initial_call=True   # avoid firing on first page load
)
def cache_cleaning(methods, threshold):
    return {"methods": methods, "threshold": threshold}

# callback that drives the two box-plots
@app.callback(
    Output('box-before', 'figure'),
```

```
    Output('box-after',  'figure'),
    Input('cleaning-store', 'data')
)
def show_cleaning_effect(cleaning_data):
    n_sample = min(100000, len(df))
    df_sampled = df.sample(n=n_sample, random_state=42)
    methods   = cleaning_data['methods']
    z_thresh  = cleaning_data['threshold']

    # BEFORE
    fig_before = px.box(df_sampled[['Entries','Exits']],
                        title='Before Cleaning',
                        points='outliers')

    # AFTER
    df_clean, _ = filter_and_group(
        df_sampled, years=None, months=None, days=None, dates=None,
        periods=None, hours=None, stations=None,
        group_mode='time',
        cleaning=methods, z_thresh=z_thresh
    )
    fig_after = px.box(df_clean[['Entries','Exits']],
                       title='After Cleaning',
                       points='outliers')

    for f in (fig_before, fig_after):
        f.update_layout(font_family='serif',
                        title_font=dict(size=18, color='blue'))
        f.update_xaxes(
            title_text="Metric",
            title_font=dict(family="serif", color="darkred", size=16)
        )
        f.update_yaxes(
            title_text="Value",
            title_font=dict(family="serif", color="darkred", size=16)
        )

    return fig_before, fig_after

# Dimensionality Reduction (PCA)
# ------------------------------------------------------------------
pca_layout = html.Div([
    html.H2("Dimensionality Reduction", style=title_font),

    html.P(
        "Use the slider to choose how many principal components to inspect. "
        "The heat-map shows correlation among the selected PCs, while the line "
        "plot shows cumulative explained variance.",
        style={'maxWidth': '800px'}
    ),

    dcc.Slider(
        id='pca-n-components',
        min=2,
        max=len(explained),
        step=1,
        value=2,
        marks={i: str(i) for i in range(1, len(explained)+1)},
        tooltip={'placement': 'bottom'}
```

```
    ),

    html.Br(),

    html.Div(dcc.Loading(type="circle", color="#5c79ae",
children=dcc.Graph(id="pca-heatmap")),
             style={"display": "inline-block", "width": "48%", "verticalAlign":
"top"}),
    html.Div(dcc.Loading(type="circle", color="#5c79ae",
children=dcc.Graph(id="pca-cumvar")),
             style={"display": "inline-block", "width": "48%", "verticalAlign":
"top"}),
], style={'padding':'15px'})

@app.callback(
    Output("pca-heatmap", "figure"),
    Output("pca-cumvar",  "figure"),
    Input("pca-n-components", "value")
)
def update_pca_plots(k_user):
    k = int(k_user)
    k = max(2, min(k, len(explained)))
    pcs = pcs_global[:, :k]
    corr = np.corrcoef(pcs.T)
    labels = [f"PC{i+1}" for i in range(k)]

    fig_heat = go.Figure(
        data=go.Heatmap(
            z=corr,
            x=labels,
            y=labels,
            colorscale="Viridis",
            zmin=-1,
            zmax=1,
            colorbar=dict(title="ρ"),
            text=np.round(corr, 2),
            texttemplate="%{text}",
            textfont=dict(color="black", family="serif", size=14)
        )
    )
    fig_heat.update_layout(
        title=f"Correlation Matrix of First {k} PCs",
        font_family="serif",
        title_font=dict(size=18, color="blue")
    )

    fig_var = go.Figure()
    fig_var.add_trace(
        go.Scatter(
            x=list(range(1, len(cum_explained) + 1)),
            y=cum_explained,
            mode="lines+markers",
            name="Cumulative Variance"
        )
    )
    fig_var.add_shape(
        type="line",
        x0=k,
        x1=k,
```

```
        y0=0,
        y1=1,
        line=dict(dash="dash", width=2)
    )
    fig_var.add_shape(
        type="line",
        x0=1,
        x1=len(cum_explained),
        y0=0.95,
        y1=0.95,
        line=dict(color="red", dash="dot")
    )
    fig_var.update_layout(
        title="Cumulative Explained Variance",
        xaxis_title="Number of Components",
        yaxis_title="Cumulative Variance",
        yaxis_range=[0, 1.02],
        font_family="serif",
        title_font=dict(size=18, color="blue"),
        showlegend=False
    )
    fig_heat.update_xaxes(
        title_text="Principal Components",
        title_font=dict(family="serif", color="darkred", size=16)
    )
    fig_heat.update_yaxes(
        title_text="Principal Components",
        title_font=dict(family="serif", color="darkred", size=16)
    )

    fig_var.update_xaxes(
        title_text="Number of Components",
        title_font=dict(family="serif", color="darkred", size=16)
    )
    fig_var.update_yaxes(
        title_text="Cumulative Variance",
        title_font=dict(family="serif", color="darkred", size=16)
    )

    return fig_heat, fig_var

# Normality
# ------------------------------------------------------------
normal_layout = html.Div([
    html.H2("Normality Check", style=title_font),
    html.P(
        "Select a numeric feature from the dropdown to assess its normality. "
        "The left hand graph displays a histogram overlaid with a kernel-density
curve, "
        "The right hand graph is a QQ-plot",
        style={"maxWidth": "800px"}
    ),
    html.Div([
        dcc.Dropdown(
            id="normal-feature",
            options=[{"label": c, "value": c} for c in numeric_cols],
            value='Entries',
            placeholder='Please Select:',
            multi=False,
```

```
            clearable=False,
            style={"width": "40%"}
        )
    ]),

    html.Br(),
    html.Div(dcc.Loading(type="circle",
color="#5c79ae",children=dcc.Graph(id="normal-hist")),
             style={"display": "inline-block", "width": "48%", "verticalAlign":
"top"}),
    html.Div(dcc.Loading(type="circle",
color="#5c79ae",children=dcc.Graph(id="normal-qq")),
             style={"display": "inline-block", "width": "48%", "verticalAlign":
"top"}),
])

@app.callback(
    Output("normal-hist", "figure"),
    Output("normal-qq",   "figure"),
    Input("normal-feature", "value"),
    State("cleaning-store", "data")
)
def normality_plots(feature, cleaning_data):
    n_sample = min(100000, len(df))
    df_sampled = df.sample(n=n_sample, random_state=42)
    if cleaning_data is None:
        cleaning_data = {"methods": ["clip"], "threshold": 3.0}

    df_clean, _ = filter_and_group(
        df_sampled, years=None, months=None, days=None, dates=None,
        periods=None, hours=None, stations=None,
        group_mode="time",
        cleaning=cleaning_data["methods"],
        z_thresh=cleaning_data["threshold"]
    )

    series = df_clean[feature].dropna()

    hist_fig = px.histogram(
        series,
        marginal="box",
        histnorm="probability density",
        opacity=0.75,
        title=f"Histogram + KDE — {feature}"
    )
    hist_fig.update_traces(marker_line_width=1)
    hist_fig.add_trace(
        go.Scatter(
            x=np.linspace(series.min(), series.max(), 300),
            y=stats.gaussian_kde(series)(np.linspace(series.min(), series.max(),
300)),
            mode="lines",
            name="KDE",
            line=dict(width=2)
        )
    )
    hist_fig.update_layout(
        font_family="serif",
        title_font=dict(size=18, color="blue")
```

```
    )
    hist_fig.update_xaxes(
        title_text=feature,
        title_font=dict(family="serif", color="darkred", size=16)
    )
    hist_fig.update_yaxes(
        title_text="Density",
        title_font=dict(family="serif", color="darkred", size=16)
    )

    theory, sample = stats.probplot(series, dist="norm", plot=None)[0]
    qq_fig = px.scatter(
        x=theory,
        y=sample,
        labels={"x": "Theoretical Quantiles", "y": "Sample Quantiles"},
        title=f"QQ-Plot — {feature}"
    )
    qq_fig.add_shape(
        type="line",
        x0=theory.min(), y0=sample.min(),
        x1=theory.max(), y1=sample.max(),
        line=dict(color="red", dash="dash")
    )
    qq_fig.update_layout(
        font_family="serif",
        title_font=dict(size=18, color="blue")
    )
    qq_fig.update_xaxes(
        title_font=dict(family="serif", color="darkred", size=16)
    )
    qq_fig.update_yaxes(
        title_font=dict(family="serif", color="darkred", size=16)
    )

    return hist_fig, qq_fig

# Transform
# ------------------------------------------------------------
transform_layout = html.Div([
    html.H2("Data Transformation", style=title_font),

    dcc.Markdown(
        """
    Pick a numeric feature and apply a transform.
    Click **Apply** to commit the change to the working dataset.
    Click **Reset All** to restore the original dataset.
    """,
        style={"maxWidth": "800px"}
    ),

    html.Div([
        html.Label("Feature:"),
        dcc.Dropdown(
            id="trans-feature",
            options=[{"label": c, "value": c}
                     for c in df.select_dtypes(include=[np.number]).columns],
            value='Entries',
            clearable=False,
            searchable=True,
```

```
            style={"width": "40%"}
        )
    ]),

    html.Div([
        html.Label(["Method:" ,  dcc.Tooltip(
            children="Choose a transform: log, root, z-score or min-max.",
            direction="right",
            style={"fontFamily": "serif"}
        )]),
        dcc.Dropdown(
            id="trans-method",
            options=[
                {"label": "Log (log10(x+1))",      "value": "log"},
                {"label": "Square-root (sqr(x))",  "value": "sqrt"},
                {"label": "Standardise (Z-score)", "value": "zscore"},
                {"label": "Min–Max 0-1",           "value": "minmax"}
            ],
            value="log",
            clearable=False,
            searchable=True,
            style={"width": "40%"}
        ),
    ], style={"margin-top": "10px"}),

    html.Br(),
    html.Button("Apply transform", id="apply-transform", n_clicks=0,
                style={"margin-right": "20px"}),
    html.Button("Reset ALL", id="reset-data", n_clicks=0,
                style={"background": "crimson", "color": "white"}),

    html.Br(), html.Br(),
    html.Div(
        dcc.Loading(type="circle", color="#5c79ae",children=dcc.Graph(id="trans-
before")),
        style={"display": "inline-block", "width": "48%", "verticalAlign": "top"}),

    html.Div(dcc.Loading(type="circle",
color="#5c79ae",children=dcc.Graph(id="trans-after")),
        style={"display": "inline-block", "width": "48%", "verticalAlign": "top"}),
])

@app.callback(
    Output("transform-store", "data", allow_duplicate=True),
    Input("apply-transform", "n_clicks"),
    Input("reset-data", "n_clicks"),
    State("trans-feature", "value"),
    State("trans-method", "value"),
    State("transform-store", "data"),
    prevent_initial_call=True
)
def mutate_transform_store(n_apply, n_reset, feature, method, current_map):
    current_map = {} if current_map is None else current_map.copy()

    # Handle None defaults
    n_apply = n_apply or 0
    n_reset = n_reset or 0

    # Determine trigger
```

```
    ctx = dash.callback_context
    if not ctx.triggered:
        return current_map
    trigger = ctx.triggered[0]['prop_id'].split('.')[0]

    # Avoid triggering on tab change or load
    if n_apply == 0 and n_reset == 0:
        return current_map

    if trigger == "apply-transform" and feature:
        current_map[feature] = method
    elif trigger == "reset-data":
        current_map.clear()

    return current_map

@app.callback(
    Output("trans-before", "figure"),
    Output("trans-after",  "figure"),
    Input("trans-feature",  "value"),
    Input("trans-method",   "value"),
    State("cleaning-store", "data"),
    State("transform-store","data")
)
def transform_plots(feature, method, cleaning_data, trans_map):
    if cleaning_data is None:
        cleaning_data = {"methods": ["clip"], "threshold": 3.0}

    df_clean, _ = filter_and_group(
        df,
        years=None, months=None, days=None, dates=None,
        periods=None, hours=None, stations=None,
        group_mode="time",
        cleaning=cleaning_data["methods"],
        z_thresh=cleaning_data["threshold"],
        trans_map=None)

    s_before = df_clean[feature].dropna()

    if method == "log":
        s_after = np.log10(s_before + 1)
        mlabel  = "log10(x+1)"
    elif method == "sqrt":
        s_after = np.sqrt(np.clip(s_before, 0, None))
        mlabel  = "sqrt(x)"
    elif method == "zscore":
        s_after = (s_before - s_before.mean()) / s_before.std(ddof=1)
        mlabel  = "Z-score"
    else:
        s_after = (s_before - s_before.min()) / (s_before.max() - s_before.min())
        mlabel  = "Min–Max 0-1"

    fig_before = px.histogram(s_before, opacity=0.75, title=f"Original —
{feature}")
    fig_after  = px.histogram(s_after, opacity=0.75, title=f"Transformed
({mlabel})")

    for f in (fig_before, fig_after):
        f.update_layout(font_family="serif", title_font=dict(size=18,
```

```
color="blue"))
        f.update_xaxes(title_font=dict(family="serif", color="darkred", size=16))
        f.update_yaxes(title_font=dict(family="serif", color="darkred", size=16))

    return fig_before, fig_after

def apply_transforms(df_raw: pd.DataFrame, trans_map: dict[str, str]) ->
pd.DataFrame:
    df_work = df_raw.copy()
    for col, method in trans_map.items():
        if col not in df_work.columns:
            continue
        s = df_work[col]
        if method == "log":
            df_work[col] = np.log10(s + 1)
        elif method == "sqrt":
            df_work[col] = np.sqrt(np.clip(s, 0, None))
        elif method == "zscore":
            df_work[col] = (s - s.mean()) / s.std(ddof=1)
        elif method == "minmax":
            df_work[col] = (s - s.min()) / (s.max() - s.min())
    return df_work
# Single Variable
# ------------------------------------------------------------
dashboard_layout = html.Div([
    html.H2("Single-Feature"),
    html.Div([
        html.Div([
            html.Label("Feature (y-axis):"),
            dcc.Dropdown(
                options=[{"label": f, "value": f} for f in cols],
                value="Entries",
                id="feature-select"
            ),
        ], style={"width": "15%", "display": "inline-block"}),

        html.Div([
            html.Label("Years:"),
            dcc.Dropdown(
                options=[{"label": y, "value": y} for y in
sorted(df["Year"].unique())],
                multi=True, id="year-filter"
            )
        ], style={"width": "13%", "display": "inline-block"}),

        html.Div([
            html.Label("Months:"),
            dcc.Dropdown(
                options=[{"label": m, "value": m} for m in df["Month"].unique()],
                multi=True, id="month-filter"
            )
        ], style={"width": "13%", "display": "inline-block"}),

        html.Div([
            html.Label("Days of Week:"),
            dcc.Dropdown(
                options=[{"label": d, "value": d} for d in df["Day of
Week"].unique()],
                multi=True, id="day-filter"
```

```
            )
        ], style={"width": "13%", "display": "inline-block"}),
        html.Div([
            html.Label("Dates:"),
            dcc.Dropdown(
                options=[{"label": str(d), "value": d} for d in
sorted(df["Date"].unique())],
                multi=True, id="date-filter"
            )
        ], style={"width": "13%", "display": "inline-block"}),
        html.Div([
            html.Label("Time Periods:"),
            dcc.Dropdown(
                options=[{"label": p, "value": p} for p in df["Time
Period"].unique()],
                multi=True, id="period-filter"
            )
        ], style={"width": "13%", "display": "inline-block"}),
        html.Div([
            html.Label("Hours:"),
            dcc.Dropdown(
                options=[{"label": p, "value": p} for p in df["Hour"].unique()],
                multi=True, id="hour-filter"
            )
        ], style={"width": "13%", "display": "inline-block"}),
        html.Div([
            html.Label("Stations:"),
            dcc.Dropdown(
                options=[{"label": s, "value": s} for s in df["Station
Name"].unique()],
                multi=True, id="station-filter", searchable=True
            )
        ], style={"width": "20%", "display": "inline-block"}),

        html.Div([
            html.Label("Group by Station:"),
            dcc.RadioItems(
                options=[
                    {"label": "Yes", "value": "station"},
                    {"label": "No", "value": "time"}
                ],
                value="time",
                id="group-by-station-toggle",
                labelStyle={'display': 'inline-block', 'margin-right': '10px'}
            )
        ], style={"margin-top": "15px"})
    ]),

    html.Hr(),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="bar-plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="pie-
chart")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="line-
plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="area-
plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="box-plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="hist-
plot")),
```

```
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="kde-plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="violin-
plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="strip-
plot")),
    dcc.Loading(type="circle", color="#5c79ae", children=dcc.Graph(id="rug-plot"))
])

# Shared filter and grouping logic
def filter_and_group(df, years, months, days, dates, periods, hours, stations,
group_mode, cleaning, z_thresh, trans_map=None):
    df_ = df.copy()

    # Apply cleaning
    if 'clip' in cleaning:
        for col in ['Entries', 'Exits', 'Tap Entries', 'Tap Exits', 'NonTapped
Entries', 'NonTapped Exits']:
            df_[col] = df_[col].clip(lower=0)

    if 'zscore' in cleaning:
        numeric_cols = ['Entries', 'Exits', 'Tap Entries', 'Tap Exits', 'NonTapped
Entries', 'NonTapped Exits']
        for col in numeric_cols:
            zscores = np.abs(stats.zscore(df_[col]))
            df_ = df_[zscores < z_thresh]

    # Transform
    if trans_map:
        df_ = apply_transforms(df_, trans_map)

    if years:
        df_ = df_[df_['Year'].isin(years)]
    if months:
        df_ = df_[df_['Month'].isin(months)]
    if days:
        df_ = df_[df_['Day of Week'].isin(days)]
    if dates:
        df_ = df_[df_['Date'].isin(dates)]
    if periods:
        df_ = df_[df_['Time Period'].isin(periods)]
    if hours:
        df_ = df_[df_['Hour'].isin(hours)]
    if stations and group_mode != "station":
        df_ = df_[df_['Station Name'].isin(stations)]
    elif stations and group_mode == "station" and len(stations) > 1:
        df_ = df_[df_['Station Name'].isin(stations)]

    if group_mode == "station":
        group = "Station Name"
    elif df_['Year'].nunique() > 1:
        group = "Year"
    elif df_['Month'].nunique() > 1:
        group = "Month"
    elif df_['Day of Week'].nunique() > 1:
        group = "Day of Week"
    elif df_['Date'].nunique() > 1:
        group = "Date"
    elif df_['Hour'].nunique() > 1:
        group = "Hour"
```

```
    else:
        group = None

    return df_, group

@app.callback(
    Output("bar-plot", "figure"),
    Output("pie-chart", "figure"),
    Output("line-plot", "figure"),
    Output("area-plot", "figure"),
    Output("box-plot", "figure"),
    Output("hist-plot", "figure"),
    Output("kde-plot", "figure"),
    Output("violin-plot", "figure"),
    Output("strip-plot", "figure"),
    Output("rug-plot", "figure"),
    Input("feature-select", "value"),
    Input("year-filter", "value"),
    Input("month-filter", "value"),
    Input("day-filter", "value"),
    Input("date-filter", "value"),
    Input("period-filter", "value"),
    Input("hour-filter", "value"),
    Input("station-filter", "value"),
    Input("group-by-station-toggle", "value"),
    State("cleaning-store", "data"),
    State("transform-store", "data")
)
def update_all(feature, years, months, days, dates, periods, hours, stations,
group_mode, cleaning_data, trans_map):
    cleaning = cleaning_data["methods"]
    z_thresh = cleaning_data["threshold"]
    df_, group = filter_and_group(df, years, months, days, dates, periods, hours,
stations, group_mode, cleaning, z_thresh, trans_map)

    if not group:
        return [px.bar(title="Insufficient data for grouping")] * 10

    grouped = df_.groupby(group)[feature].sum().reset_index()

    common_layout = dict(
        font_family="serif",
        font_color="darkred",
        title_font=dict(family="serif", size=24, color="blue")
    )

    bar = px.bar(grouped, x=group, y=feature, color=group, title=f"{feature} by
{group}",text_auto='.2f')
    bar.update_layout(**common_layout, xaxis_title=group, yaxis_title=feature,
yaxis_showgrid=True, showlegend=True)
    bar.update_traces(marker_line_width=1.5)

    explode = [0.1 if v == grouped[feature].max() else 0 for v in grouped[feature]]
    pie = px.pie(grouped, names=group, values=feature, title=f"{feature} Share by
{group}", hole=0.3)
    pie.update_traces(pull=explode)
    pie.update_layout(**common_layout, showlegend=True)

    line = px.line(grouped, x=group, y=feature, title=f"{feature} Trend by
```

```
{group}", markers=True)
    line.update_layout(**common_layout, xaxis_title=group, yaxis_title=feature,
xaxis_showgrid=True, yaxis_showgrid=True, showlegend=True)

    grouped['cumulative'] = grouped[feature].cumsum()
    area = px.area(grouped, x=group, y='cumulative', title=f"Cumulative {feature}
Area Chart by {group}")
    area.update_layout(**common_layout, xaxis_title=group, yaxis_title=feature,
xaxis_showgrid=True, yaxis_showgrid=True,showlegend=True)

    n_sample = min(50000, len(df_))
    df_sampled = df_.sample(n=n_sample, random_state=42)
    box = px.box(df_sampled, x=group, y=feature, title=f"{feature} Box Chart by
{group} (Sampled 50k)")
    box.update_layout(**common_layout, xaxis_title=group, yaxis_title=feature,
xaxis_showgrid=True, yaxis_showgrid=True,showlegend=True)

    hist = px.histogram(df_sampled, x=feature, marginal="violin", opacity=0.7,
title=f"Histogram with KDE: {feature} (Sampled 50k)")
    hist.update_layout(**common_layout, xaxis_title=feature, yaxis_title="Count",
xaxis_showgrid=True, yaxis_showgrid=True)

    hist_data = [df_sampled[feature].dropna()]
    group_labels = [feature]
    kde = ff.create_distplot(hist_data, group_labels, show_hist=False,
show_rug=True)
    kde.update_layout(**common_layout, title=dict(text=f"KDE of {feature}", x=0.5,
xanchor='center'), xaxis_title=feature, yaxis_title="Density", xaxis_showgrid=True,
yaxis_showgrid=True)

    violin = px.violin(df_sampled, y=feature, box=True, points="all",
title=f"Violin Plot of {feature} (Sampled 50k)")
    violin.update_layout(**common_layout, yaxis_title=feature, xaxis_showgrid=True,
yaxis_showgrid=True)

    strip = px.strip(df_sampled, y=feature, title=f"Strip Plot of {feature}
(Sampled 50k)")
    strip.update_layout(**common_layout, yaxis_title=feature, xaxis_showgrid=True,
yaxis_showgrid=True)

    rug = px.scatter(df_sampled, x=feature, y=[0] * len(df_sampled), title=f"Rug
Plot of {feature} (Sampled 50k)")
    rug.update_layout(**common_layout, xaxis_title=feature, yaxis_visible=False,
xaxis_showgrid=True)
    return bar, pie, line, area, box, hist, kde, violin, strip, rug

@app.callback(
    Output("date-filter", "options"),
    Input("year-filter", "value"),
    Input("month-filter", "value"),
    Input("day-filter", "value"),
    Input("period-filter", "value"),
    Input("station-filter", "value"),
)
def update_date_options(years, months, days, periods, stations):
    df_ = df.copy()
    if years:
        df_ = df_[df_["Year"].isin(years)]
    if months:
```

```
        df_ = df_[df_["Month"].isin(months)]
    if days:
        df_ = df_[df_["Day of Week"].isin(days)]
    if periods:
        df_ = df_[df_["Time Period"].isin(periods)]
    if stations:
        df_ = df_[df_["Station Name"].isin(stations)]
    options = sorted(df_["Date"].unique())
    return [{"label": str(d), "value": d} for d in options]

@app.callback(
    Output("hour-filter", "options"),
    Input("year-filter", "value"),
    Input("month-filter", "value"),
    Input("day-filter", "value"),
    Input("period-filter", "value"),
    Input("station-filter", "value"),
    Input("date-filter", "value")
)
def update_hour_options(years, months, days, periods, stations, dates):
    df_ = df.copy()
    if years:
        df_ = df_[df_["Year"].isin(years)]
    if months:
        df_ = df_[df_["Month"].isin(months)]
    if days:
        df_ = df_[df_["Day of Week"].isin(days)]
    if periods:
        df_ = df_[df_["Time Period"].isin(periods)]
    if stations:
        df_ = df_[df_["Station Name"].isin(stations)]
    if dates:
        df_ = df_[df_["Date"].isin(dates)]
    options = sorted(df_["Hour"].unique())
    return [{"label": str(h), "value": h} for h in options]

# Multi Variable plots
# ------------------------------------------------------------------------
multi_layout = html.Div([
    html.H2("Multi-Variable Plots", style=title_font),

    html.P("Select 2 or more numeric features to explore multi-variable
relationships.",
           style={"maxWidth": "800px"}),

    html.Div([
        html.Label("Select Features:"),
        dcc.Dropdown(
            id="multi-features",
            options=[{"label": col, "value": col} for col in numeric_cols],
            multi=True,
            value=["Entries", "Exits", "Hour"],
            placeholder="Choose 2+ numeric features",
            style={"width": "50%"}
        )
    ]),

    html.Br(),
```

```
    html.Div([
        html.Label("Color by (optional):"),
        dcc.Dropdown(
            id="multi-color",
            options=[{"label": col, "value": col} for col in
df.select_dtypes(include=["object", "category"]).columns],
            placeholder="Select a categorical feature for color (3D only)",
            value='Time Period',
            style={"width": "50%"}
        )
    ]),
    html.Br(),

    dcc.Loading(type="circle", color="#5c79ae", children=html.Div(id="multi-
plots"))
])

@app.callback(
    Output("multi-plots", "children"),
    Input("multi-features", "value"),
    Input("multi-color", "value"),
)
def update_multi_plots(selected, color_by):
    if not selected or len(selected) < 2:
        return html.P("Please select at least two features.")

    plot_cols = selected.copy()
    if color_by and color_by not in plot_cols:
        plot_cols.append(color_by)

    df_ = df[plot_cols].dropna().sample(n=100000, random_state=42)

    title_style = {"title_font": dict(family=title_font["family"],
size=title_font["size"], color=title_font["color"])}
    axis_style  = {"title_font": dict(family=label_font["family"],
size=label_font["size"], color=label_font["color"])}

    x, y = selected[:2]
    color_arg = {"color": df_[color_by]} if color_by else {}

    fig1 = px.scatter(df_, x=x, y=y, title=f"Scatter: {x} vs {y}", **color_arg)
    fig2 = px.density_contour(df_, x=x, y=y, title=f"Contour: {x} vs {y}",
**color_arg)
    fig3 = px.scatter(df_, x=x, y=y, marginal_x="histogram",
marginal_y="histogram", title=f"Joint Plot: {x} vs {y}", **color_arg)

    for fig in (fig1, fig2, fig3):
        fig.update_layout(**title_style)
        fig.update_xaxes(**axis_style)
        fig.update_yaxes(**axis_style)

    graphs = [dcc.Graph(figure=fig1), dcc.Graph(figure=fig2),
dcc.Graph(figure=fig3)]

    if len(selected) >= 2:
        fig4 = px.scatter_matrix(df_, dimensions=selected, color=color_by if
color_by else None, title="Scatter Matrix")
        fig4.update_layout(**title_style)
        graphs.append(dcc.Graph(figure=fig4))
```

```
    if len(selected) >= 3:
        fig5 = px.scatter_3d(df_, x=selected[0], y=selected[1], z=selected[2],
color=color_by if color_by else None,
                             title=f"3D Scatter: {selected[0]}, {selected[1]},
{selected[2]}")
        fig5.update_traces(marker=dict(size=3))
        fig5.update_layout(**title_style)
        graphs.append(dcc.Graph(figure=fig5))

    return graphs


# Statistics tab
# ------------------------------------------------------------------------
stats_layout = html.Div([
    html.H2("Statistics", style=title_font),

    html.Div([
        html.Label("Feature:"),
        dcc.Dropdown(
            id="stats-feature",
            options=[{"label": c, "value": c}
                     for c in df.select_dtypes(include=[np.number]).columns],
            value="Entries",
            clearable=False,
            searchable=True,
            style={"width": "40%"}
        )
    ]),

    html.Br(),

    html.H3("Overall Statistics", style={"color": "#5c79ae"}),

    # summary cards (filled by callback)
    html.Div(id="stats-summary", style={"display": "flex", "gap": "25px"}),

    html.Br(),

    html.H3("Statistics by Station", style={"color": "#5c79ae"}),

    dash_table.DataTable(
        id="stats-table",
        style_cell={
            "fontFamily": "serif",
            "padding": "6px"
        },
        style_header={
            "backgroundColor": "#5c79ae",
            "color": "white",
            "fontWeight": "bold"
        },
        sort_action="native",
        page_action="native",
        page_size=20
    )
])
```

```
@app.callback(
    Output("stats-summary", "children"),
    Output("stats-table",   "data"),
    Output("stats-table",   "columns"),
    Input("stats-feature",  "value"),
    State("cleaning-store", "data"),
    State("transform-store","data")
)
def update_stats(feature, cleaning_data, trans_map):
    if cleaning_data is None:
        cleaning_data = {"methods": ["clip"], "threshold": 3.0}

    df_work, _ = filter_and_group(
        df,
        years=None, months=None, days=None, dates=None,
        periods=None, hours=None, stations=None,
        group_mode="time",
        cleaning=cleaning_data["methods"],
        z_thresh=cleaning_data["threshold"],
        trans_map=trans_map
    )

    s = df_work[feature].dropna()

    stats_dict = {
        "Count":     len(s),
        "Mean":      s.mean(),
        "Median":    s.median(),
        "Std Dev":   s.std(ddof=1),
        "Variance":  s.var(ddof=1),
        "Min":       s.min(),
        "Max":       s.max()
    }

    cards = [
        html.Div([
            html.H4(k, style={"margin": "0", "color": "#5c79ae"}),
            html.P(f"{v:,.2f}" if k != "Count" else f"{int(v):,}",
                   style={"fontSize": "22px", "margin": "0",
                          "color": "darkred"})
        ], style={
            "border": "1px solid #ddd",
            "borderRadius": "8px",
            "padding": "12px",
            "minWidth": "110px",
            "textAlign": "center",
            "boxShadow": "2px 2px 3px rgba(0,0,0,0.1)"
        }) for k, v in stats_dict.items()
    ]

    tbl = (df_work
           .groupby("Station Name")[feature]
           .agg(["count", "mean", "median", "std", "var", "min", "max"])
           .reset_index()
           .rename(columns={
               "count":  "Count",
               "mean":   "Mean",
               "median": "Median",
```

```
                "std":    "Std Dev",
                "var":    "Variance",
                "min":    "Min",
                "max":    "Max"
            })
            .round(2))

    columns = [{"name": c, "id": c} for c in tbl.columns]
    data    = tbl.to_dict("records")

    return cards, data, columns

# ---------------------------------------------------------------------
# MAIN
# ---------------------------------------------------------------------
if __name__ == '__main__':
    app.run(
        port=8035
    )
```