Travel Time Calculation Across Different Data Sources

Elijah Whitham-Powell, Meghana Cyanam

03-19-2025

Contents

1 Executive Summary

This project explored two primary sources of traffic data—PORTAL (managed by ODOT and WSDOT) and INRIX (a commercial provider)—to investigate travel time trends on highways in the Portland metropolitan area. Each dataset presents unique strengths and limitations, ranging from sensor coverage and data frequency to the way road segments are defined. Integrating the datasets required a robust approach to handling differences in geometry (e.g., segment boundaries), time intervals, and data quality or availability.

A core portion of the work involved "cleaning" and "harmonizing" the two datasets so that they could be combined in a single framework. This included transforming hex-encoded road segments (in PORTAL) into usable geometry objects, standardizing coordinate reference systems, normalizing timestamps across multiple years, and filtering records to ensure consistency in coverage. Ultimately, the merged dataset enables more comprehensive analyses—such as year-to-year trend detection but also potential pit falls in the data, such as not enough coverage between the segments in PORTAL and INRIX.

Key achievements include:

- *Metadata Integration:* Successfully joined PORTAL station segments with INRIX TMC (Traffic Message Channel) segments via spatial intersection. This step highlighted how different segmentation and directional labeling can lead to data mismatches.
- *Time-Series Standardization:* Unified 15-minute interval data from both sources, converting timestamps to a consistent time zone and format for seamless comparison.
- *Validation and Visualization:* Used GeoPandas plotting to confirm that station coverage areas correctly overlapped with TMC segments. This allowed identification of potential misalignments or missing station geometry.
- *Foundations for Incident Analysis:* While the scope shifted toward data-integration workflows, the cleaned and merged dataset can now be readily used to overlay incident data and evaluate how quickly each source detects and captures major congestion.

In sum, this project provides a framework for combining heterogeneous traffic datasets, paving the way for more refined analyses of congestion patterns, travel time reliability, and incident impacts across the region.

Code for this project can be found in a GitHub repo:

- `https://github.com/whitham-powell/cades-traveltime-compare`

This may continue to be updated and changed, but the final version of the code used for this report in the appendix (E) for the data processing steps as a reference.

2 Introduction

2.1 Background

Accurate and reliable travel time information is a cornerstone for effective traffic management, congestion mitigation, and infrastructure planning—particularly for agencies such as the Oregon Department of Transportation (ODOT) and research organizations like the Transportation Research and Education Center (TREC) at Portland State University. This project focuses on comparing and integrating two key traffic data sources used in Oregon's highway corridors:

- **PORTAL**, maintained by state transportation agencies, derives its travel-time metrics primarily from roadside sensors (e.g., loop detectors) that record vehicle counts and speeds as cars pass fixed locations.
- **INRIX**, a commercial provider, collects data from GPS-enabled vehicles, mobile devices, and other third-party sensors. By tapping into moving "probe" data rather than fixed detectors, INRIX can often capture a broader picture of travel conditions.

Because each source uses different measurement techniques—fixed detectors in PORTAL versus GPS-based probe data in INRIX—they can yield different estimates for the same segments in both spatial coverage and reported travel times. Understanding these differences, and how to best combine or compare the two datasets, is critical for ODOT in tasks such as corridor-performance monitoring, project evaluation, and incident response. By unifying these data sources, TREC and other research collaborators hope to support more nuanced analyses of traffic patterns, thereby improving decision-making and resource allocation for transportation agencies.

2.2 Objective

The original intent of this project was to analyze how travel times have evolved across multiple years (2019-2024) by examining three distinct datasets—PORTAL (managed by ODOT and WSDOT), INRIX, and TomTom—and to investigate what factors (e.g., sensor coverage, segmentation methods) might contribute to discrepancies among these sources. It was also envisioned to compare each source's ability to detect and respond to traffic incidents. However, the scope of this project altered due to the complexity of merging two, large, heterogeneous datasets. Consequently, this work focuses on cleaning, integrating, and comparing PORTAL and INRIX data, then discussing how differences in data availability and geographic coverage can affect travel time estimates and incident detection.

2.3 Tools

Python 3.12 and the following libraries:

- `pandas` - Data manipulation and analysis.
- `geopandas` - Extends pandas to allow spatial operations on geometric types.
- `numpy` - Mathematical functions on arrays and matrices.
- `matplotlib` - Plotting library.
- `shapely` - Geometric operations (installing geopandas should include this).

3 Data Sources

3.1 PORTAL

PORTAL is a public transportation data platform that provides access to a variety of transportation data sources. PORTAL data includes real-time transit data, historical transit data, and transit schedules. PORTAL data is used by transportation agencies, businesses, and researchers to monitor transit operations, plan transit projects, and analyze transit patterns.

The PORTAL data provided for use in this project included the following:

- **Metadata**:
  - **Stations: (stations.csv)** A table of stations with descriptive data of the stations. Primarily relied on these columns for processing and joining the two datasets:

    ```
    ['stationid', 'highwayid', 'segment_geom', 'start_date', 'end_date']
    ```

    A full list of available columns can be found in the appendix B.1.1.

    Example of the stations metadata:

    ```
       stationid  highwayid segment_geom start_date   end_date
    0       3196          6  010200...    2014-0...        NaN
    1       3001          4  010200...    2012-0...  2014-1...
    2       3001          4  010200...    2015-0...        NaN
    3       1104          4  010200...    2012-0...  2014-1...
    4       1104          4  010200...    2014-1...  2014-1...
    ```

    Where:

    * `stationid`: The unique identifier for the station.
    * `highwayid`: The unique identifier for the highway.
    * `segment_geom`: The geometry of the road segment. Stored as a hex-encoded WKB and required conversion for joining with the INRIX shapefile.
    * `start_date`: The date the station was activated.
    * `end_date`: The date the station was deactivated.
  - **Highways: (highways.csv)** A table of highways with descriptive data of the highways. Primarily relied on these columns for processing and joining the two datasets:

    ```
    ['highwayid', 'direction', 'bound', 'highwayname']
    ```

    A full list of available columns can be found in the appendix B.1.2.

    Example of the highways metadata:

    ```
       highwayid direction bound highwayname
    0        614      WEST    WB     R5 I-84
    1          0      unkn    uB         NaN
    2        615      EAST    EB     R5 I-84
    3         16     SOUTH    SB      US 99E
    4         61     SOUTH    SB        SR 7
    ```

    Where:

    * `highwayid`: The unique identifier for the highway. Used to join with the stations metadata to bolster the station metadata.

    * `direction`: The direction of the highway was used to establish what direction traffic was flowing for a given station.
    * `bound`: The bound of the highway (NB, SB, EB, WB). Further used to establish the direction of traffic flow when comparing to INRIX data.
    * `highwayname`: The name of the highway. Was used for descriptive purposes and to ensure correctness of matching INRIX data.
  - **Detectors: (detectors.csv)** A table of detectors that we did not use in this project. A full list of available columns can be found in the appendix B.1.3.

- **Traffic data:** Timeseries data of 15-minute intervals for 2019 through 2024 covering I-5, I-205, and SR-14 corridors. The data is grouped by corridor with files for for each bound, state and year. For example, I-5 Southbound in Oregon in 2023 would have this path and filename: `PORTAL/I-5 Corridor/ORI5SB_2023.csv`

```
   stationid  stationtt  stationcountreadings                starttime
0      10642   0.585000                   180  2023-12-18 12:15:00-08
1       1033   0.233333                   405  2023-03-05 07:45:00-08
2       3173   0.180000                   405  2023-06-01 00:30:00-07
3       1034   0.335000                   450  2023-03-08 19:15:00-08
4       1147   0.670000                   270  2023-11-21 00:45:00-08
```

  Where:

  - `stationid`: The unique identifier for the station.
  - `stationtt`: The travel time in minutes past the station.
  - `stationcountreadings`: The number of vehicles passing the station during the 15-minute interval.
  - `starttime`: The time of the observation (beginning of the 15-minute interval).

3.2 INRIX

INRIX is a private company that collects and analyzes traffic data. They provide a variety of data products, including real-time traffic data, historical traffic data, and traffic forecasts. INRIX data is collected from a variety of sources, including GPS data from vehicles, mobile devices, and road sensors. INRIX data is used by transportation agencies, businesses, and researchers to monitor traffic conditions, plan transportation projects, and analyze traffic patterns.

The INRIX data provided for use in this project included the following:

- **Metadata**:
  - **Shape file:** A shape file of the TMC (Traffic Message Channel) codes and their corresponding geometries. The shape file is used to map the TMC codes to the corresponding road segments and contains additional metadata about the road segments found in the TMC identification file. While additional years were provided, only the 2023 were used and tested.

    A full list of available columns in the INRIX shape file can be found in the appendix B.2.1.

    Example of the INRIX shape file as a dataframe:

```
        Tmc RoadNumb_1   Direction  \
0  114+52629       None   Eastbound
1  114+52633       None   Eastbound
2  114+52708       None  Northbound
3  114+52715       None  Northbound
4  114+52720       None   Eastbound

                                            geometry
0  LINESTRING (-123.29476 44.58176, -123.2945 44....
1  LINESTRING (-123.25326 44.58882, -123.25294 44...
2  LINESTRING (-123.30875 44.64446, -123.30863 44...
3  LINESTRING (-123.13165 44.61689, -123.13165 44...
4  LINESTRING (-123.10932 44.62391, -123.10843 44...
```

– **TMC identification files:** Similar to the PORTAL stations metadata, it contains descriptive data about the TMC codes referring to their location, and active dates. There is a single file for each year. These files were used initially, but largely abandoned in favor of the shape file. It did end up being used for finding unique TMC codes to reduce the shapefile based metadata to the 139 TMC codes we were targetting. The following columns were used initially before switching to the shape file:

```
['tmc',
 'road',
 'start_latitude',
 'start_longitude',
 'end_latitude',
 'end_longitude',
 'active_start_date',
 'active_end_date']
```

A full list of available columns in the INRIX metadata files can be found in the appendix B.2.2.

Example of the INRIX metadata:

```
        tmc road  start_latitude  start_longitude  end_latitude  \
0  114P04469  I-5        45.63780       -122.66179      45.64110
1  114P04468  I-5        45.62821       -122.66686      45.63523
2  114P04467  I-5        45.62194       -122.67272      45.62656
3  114P04466  I-5        45.61442       -122.67729      45.62137
4  114-04468  I-5        45.63781       -122.66208      45.63622

   end_longitude         active_start_date           active_end_date
0     -122.66174  2022-03-22 14:00:00-04:00  2023-03-21 14:00:00-04:00
1     -122.66266  2022-03-22 14:00:00-04:00  2023-03-21 14:00:00-04:00
2     -122.66792  2022-03-22 14:00:00-04:00  2023-03-21 14:00:00-04:00
3     -122.67309  2022-03-22 14:00:00-04:00  2023-03-21 14:00:00-04:00
4     -122.66241  2022-03-22 14:00:00-04:00  2023-03-21 14:00:00-04:00
```

- **Traffic data:** Timeseries data of 15-minute intervals for 2019 through 2024 covering the Portland metro area. The data is provided as a single file for each year. The following columns were used for processing and joining the two datasets:

```
['tmc_code', 'measurement_tstamp', 'travel_time_seconds']
```

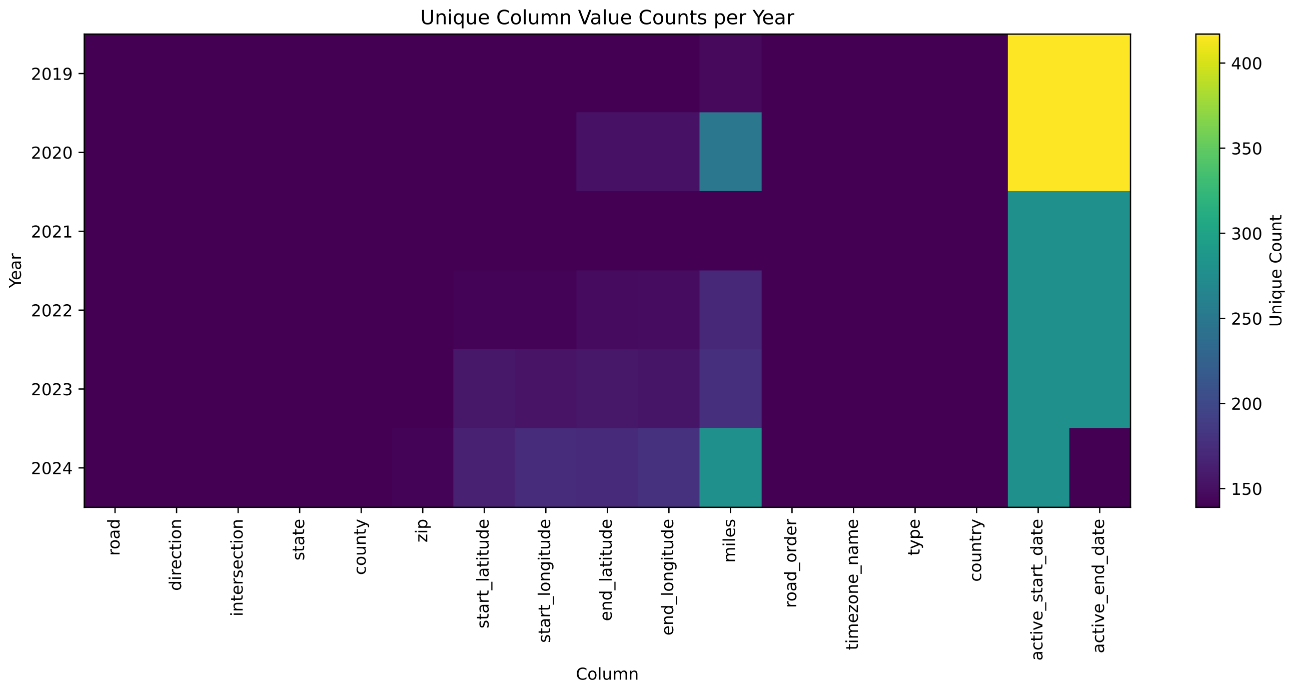
A full list of available columns in the INRIX timeseries data can be found in the appendix C.2.

Example of the INRIX timeseries data:

```
    tmc_code    measurement_tstamp  travel_time_seconds
0  114P04469  2023-01-01 00:00:00                12.54
1  114P04469  2023-01-01 00:15:00                12.82
2  114P04469  2023-01-01 00:30:00                12.99
3  114P04469  2023-01-01 00:45:00                13.32
4  114P04469  2023-01-01 01:00:00                13.22
```

3.3 Data Incosistencies

The INRIX metadata for each year provided in the TMC identification files indicated that the sensors or reading locations changed within their respective year which lead to focusing on 2023 and the shapefiles for that year.



A full table of the unique column value counts per year can be found in the appendix A.

4 Data Processing

Although our final data processing procedure evolved iteratively, below is the *recommended* order of steps for organizations or researchers aiming to replicate or extend our methods:

1. **Dataset Loading & Cleaning**
   - Load each data source (PORTAL and INRIX) into separate data structures.
   - Clean individual data sources (e.g., remove duplicates, handle NaNs, ensure consistent column naming).

- Standardize direction fields across sources (e.g., convert “WB” and “WESTBOUND” both to “WEST”).

2. **Coordinate Reference System Handling**
    - Choose a consistent coordinate reference system before performing any spatial operations. In this project, data initially appeared in Web Mercator for certain geometry columns, partly because many web maps or shapefile tools default to this projection. However, WGS 84 lat/long is recommended for more precise geospatial analysis and to avoid distortions in distance calculations.
    - Convert all geometry columns to the same coordinate reference system— WGS 84 lat/long — prior to spatial joining or plotting. This ensures that intersections or unions are calculated accurately.
3. **Spatial Join**
    - Once all geometry columns are in a consistent coordinate reference system, use GeoPandas or a similar library to perform the spatial joins. For instance, join PORTAL station segments with INRIX TMC segments by using an “intersects” or “within” predicate.
    - Validate join results by plotting the matched segments on a map to check for apparent misalignment.
4. **Time-Series Merging**
    - Convert all timestamps to a common format/time zone.
    - When merging daily or sub-hourly travel time data, align the start times, intervals, and frequencies between datasets.
    - Consider how to handle missing observations or intervals (e.g., forward fill, interpolation, or exclusion, depending on project aims).
5. **Final Data Checks**
    - Perform simple aggregations or visual checks to confirm correctness.
    - Calculate summary statistics (like average travel times or basic comparisons of differences) to catch any unexpected discrepancies introduced by the merges.

4.1 PORTAL Metadata

- Convert the station geometry from hex-encoded WKB to a shapely geometry object.

```python
from shapely.wkb import loads as wkb_loads # installing geopandas will include shapely
def wkb_to_geom(wkb_hex):
    try:
        if pd.isna(wkb_hex) or not wkb_hex.strip():
            return None
        geom = wkb_loads(bytes.fromhex(wkb_hex))
        return geom
    except (ValueError, TypeError) as e:
        print(f"Conversion error: {e}")
        return None
```

The conversion function was applied to the `segment_geom` column in the stations metadata. The total number of rows in the DataFrame and the number of successful segment conversions were printed to verify the conversion process and as you can see this does reduce the number of rows due to not every station having this geometry data.

```
Total rows in DataFrame: 1488
Successful segment conversions: 716
```

- Join the stations and highways metadata on the `highwayid` with a subset of highways columns.

```python
# Merge highways data for station direction coverage and highway names
portal_stations_df = portal_stations_df.merge(
    portal_highways_df[["highwayid", "direction", "bound", "highwayname"]],
    on="highwayid",
    how="left"
)
```

- Standardize the direction and bound columns to ensure consistency between the PORTAL and INRIX data.

```python
def standardize_direction(direction, bound=None):
    """
    Standardize direction formats between PORTAL and INRIX
    """
    direction_map = {
        # INRIX formats
        "NORTHBOUND": "NORTH",
        "SOUTHBOUND": "SOUTH",
        "WESTBOUND": "WEST",
        "EASTBOUND": "EAST",
        # PORTAL formats
        "NORTH": "NORTH",
        "SOUTH": "SOUTH",
        "WEST": "WEST",
        "EAST": "EAST",
        "NORT": "NORTH",  # Handle the truncated 'NORT'
        "CONST": None,  # Handle construction case separately
    }

    # If direction is not valid, try to use bound
    bound_map = {
        "NB": "NORTH",
        "SB": "SOUTH",
        "WB": "WEST",
        "EB": "EAST",
        "JB": None,  # Special cases
        "ZB": None,
    }

    std_direction = direction_map.get(direction.upper())
    if std_direction is None and bound is not None:
        std_direction = bound_map.get(bound.upper())
    return std_direction
```

You may not need both the direction and bound. Only the PORTAL metadata has a bound column, but it was used for verification purposes. This same function was used to standardize the direction and bound follows for both PORTAL and INRIX data.

```
portal_stations_df["standardized_direction"] = portal_stations_df.apply(
    lambda row: standardize_direction(row["direction"], row["bound"]),
    axis=1,
)
```

- The enhanced PORTAL stations metadata was then converted into a GeoDataFrame for spatial joining with the INRIX shape file.

```
# Convert to GeoDataFrame with Web Mercator CRS
the_CRS = "EPSG:3857" # This was determined using a helper function omitted in this report
portal_gdf = gpd.GeoDataFrame(portal_stations_df, geometry="segment_geom", crs=the_CRS)
```

Note: As part of the dataset joining process, the EPSG for both datasets were converted to 4326 for plotting purposes and proper joining.

```
gdf = gdf.to_crs("EPSG:4326")
```

4.2 INRIX Metadata

- The INRIX shape files were read as GeoDataFrames.

```
import geopandas as gpd
oregon_gdf = gpd.read_file("path/to/oregon_shapefile.shp")
clark_gdf = gpd.read_file("path/to/clark_shapefile.shp")
```

Since there were two shapefiles for 2023 (Oregon and Clark County, WA), they were read separately and then concatenated.

- The standardize direction function was applied to the INRIX metadata.

```
inrix_full_df["standardized_direction"] = inrix_full_df.apply(
    lambda row: standardize_direction(row["Direction"]), axis=1
)
```

- The INRIX shape file was filtered to only include the TMC codes that were present in the INRIX timeseries data.

4.3 Spatial Join of PORTAL and INRIX metadata

The PORTAL and INRIX data were spatially joined using the station and TMC geometries. The station geometry was converted from a hex-encoded WKB to a shapely geometry object. The INRIX shape file was read as a GeoDataFrame and the two datasets were joined using a spatial join. The spatial join was performed using the `sjoin()` function from the `geopandas` library. The join was performed using the `intersects` operation, which returns all records from the left GeoDataFrame that intersect with records from the right GeoDataFrame. The resulting GeoDataFrame contained the PORTAL and INRIX metadata joined on the station and TMC geometries.

4.3.1 Visualizing the geometries prior to the spatial join

The PORTAL stations and INRIX TMCs were plotted on a map to visualize the spatial relationship between the two datasets. The plot showed the locations of the stations and TMCs in the Portland metro area.

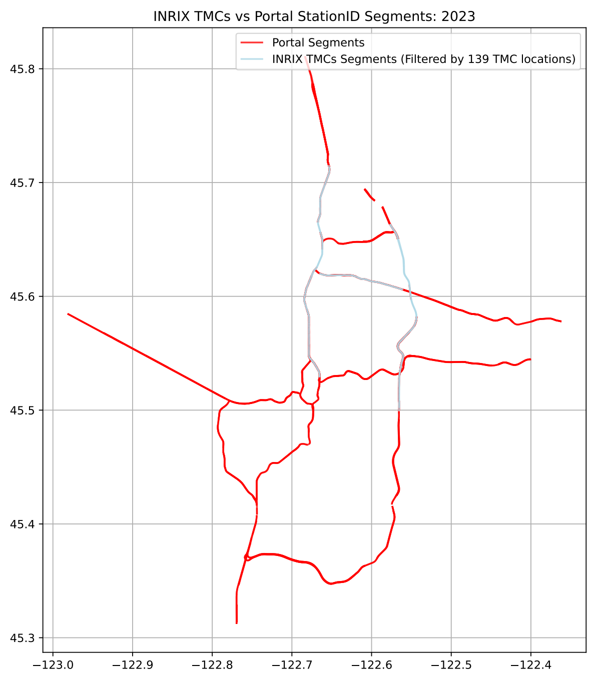
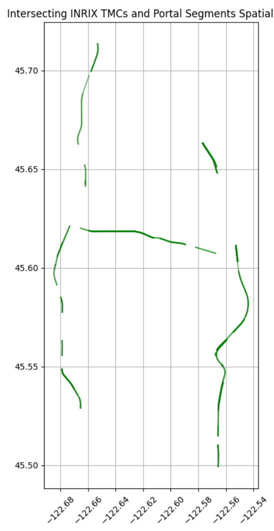

INRIX TMCs vs Portal StationID Segments: 2023
Portal Segments
INRIX TMCs Segments (Filtered by 139 TMC locations)
45.8
45.7
45.6
45.5
45.4
45.3
−123.0
−122.9
−122.8
−122.7
−122.6
−122.5
−122.4

Figure 1: Plot of the INRIX TMCs and PORTAL stations for which we have geometry data.

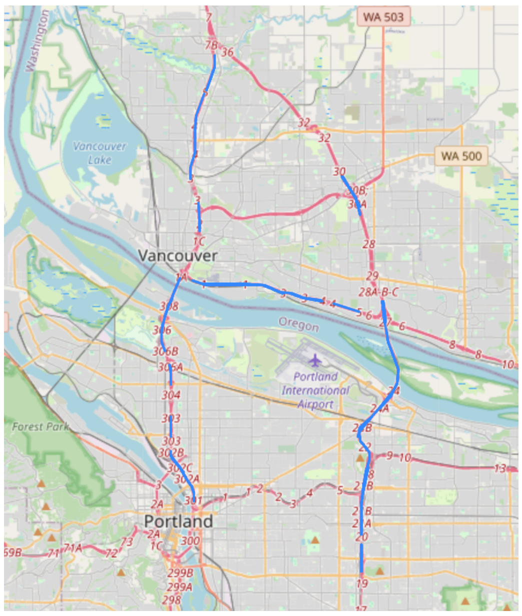
```
portal_inrix_spatial_join = inrix_filtered_by_tmc.sjoin(
    portal_gdf, how="inner", predicate="intersects", lsuffix = "inrix", rsuffix="portal"
)
```

4.3.2 Visualizing the Spatial Join

Now that the metadata has been joined, it can be visualized.



(a) Intersecting segments plot of longitude and latitudes.



(b) Spatial Join of PORTAL and INRIX Metadata on reference map.

Figure 2: Spatial Join of PORTAL and INRIX Metadata full images can be found in appendix D.

4.4 Timeseries Data

Once a spatial join dataframe was created to provide a mapping that related the PORTAL stations to the INRIX TMCs, the timeseries data was processed. The timeseries data was read in, typed and joined to the spatial join dataframe to provide a unified dataset.

4.4.1 Timestamp normalization

```python
# Function to ensure timezone offsets include minutes (e.g. "-08" becomes "-08:00")
def normalize_timezone(timestamp):
    return re.sub(r"([-+]\d{2})$", r"\1:00", timestamp)

# Function to insert default microseconds if not present
def add_default_microseconds(timestamp):
    if "." not in timestamp:
        match = re.search(r"([-+]\d{2}:\d{2})$", timestamp)
        if match:
            tz = match.group(1)
            timestamp = timestamp[: match.start()] + ".000000" + tz
    return timestamp

# Combined normalization function
def normalize_timestamp(timestamp):
    if pd.isna(timestamp):
        return None
    timestamp = normalize_timezone(timestamp)
    timestamp = add_default_microseconds(timestamp)
    return timestamp
```

4.4.2 Processing the PORTAL Timeseries Data

- The initial shape and data types of the PORTAL timeseries data:

```
Portal data shape: (1283929, 4)
Portal data initial dtypes:
stationid                int64
stationtt              float64
stationcountreadings     int64
starttime               object
dtype: object
```

- New data types for the PORTAL data:

```
Portal time series data final dtypes:
stationid                                           string[python]
stationtt                                                  Float64
stationcountreadings                                         Int64
starttime               datetime64[ns, America/Los_Angeles]
dtype: object
```

- The first 5 rows of the processed data and converted starttime column:

```
  stationid  stationtt  stationcountreadings                 starttime
0     10578       40.5                   180 2023-04-19 12:45:00-07:00
1     10580      43.95                   180 2023-05-18 21:45:00-07:00
2     10578      39.75                   180 2023-01-09 04:15:00-08:00
3       230       40.8                   180 2023-03-23 05:30:00-07:00
4       232       22.2                  1080 2023-03-03 10:00:00-08:00
```

4.4.3 Processing the INRIX Timeseries Data

A similar process of converting data types and timestamps was done for the INRIX timeseries data.

- The initial shape and data types of the INRIX timeseries data:

```
INRIX data shape: (4870004, 8)
INRIX data initial dtypes:
tmc_code                    object
measurement_tstamp          object
speed                      float64
historical_average_speed   float64
reference_speed            float64
travel_time_seconds        float64
confidence_score           float64
cvalue                     float64
dtype: object
```

- The new data types for the INRIX data:

```
INRIX data final dtypes:
tmc_code                                        string[python]
measurement_tstamp          datetime64[ns, America/Los_Angeles]
```

```
speed                                                        Float64
historical_average_speed                                     Float64
reference_speed                                              Float64
travel_time_seconds                                          Float64
confidence_score                                             Float64
cvalue                                                       Float64
dtype: object
```

- The first 5 rows and a minimal set of columns of the processed data and converted measurement_tstamp column:

```
    tmc_code        measurement_tstamp  travel_time_seconds
0  114P04469 2023-01-01 00:00:00-08:00                12.54
1  114P04469 2023-01-01 00:15:00-08:00                12.82
2  114P04469 2023-01-01 00:30:00-08:00                12.99
3  114P04469 2023-01-01 00:45:00-08:00                13.32
4  114P04469 2023-01-01 01:00:00-08:00                13.22
```

4.4.4 Joining the Timeseries Data with the Spatial Join Data

The metadata obtained from the spatial join was initially filtered to include only PORTAL stations active during 2023, while it was assumed that the INRIX TMC data remained active for the entire year. However, we did not verify whether the locations of the matched TMCs were consistent throughout the year, which may introduce errors. To address this, it is necessary to ensure that the active dates for both the TMC data and the PORTAL stations are aligned with the period under analysis and that their locations are not significantly different. The following would be the high-level steps for joining the metadata and then joining it with the timeseries data.

1. **Data Loading:** Import the segment-converted PORTAL data and the INRIX shapefile data.

2. **Spatial and Temporal Join:** Merge the datasets based on the intersection of their geometries as well as the matching active dates. Note that this join may produce duplicate TMC–station pairs for each active date.

3. **Timeseries Merging:** Integrate the timeseries data from both sources by matching timestamps, while ensuring that data is extracted only during the overlapping active periods of the TMCs and the stations.

Alternatively, you could choose the following simpler options,

- Ignore TMCs or stations that were not active the entire year or the entire period of interest.
- Drop TMC–station pairs in which there is evidence that either changed location during the period of interest.

Careful joining of the timeseries and metadata: This was order dependent to avoid duplication of rows.

```
inrix_enriched = pd.merge(
    inrix_data_filtered_df,
    sjoined_portal_inrix_df[["Tmc", "stationid"]],
    left_on="tmc_code",
    right_on="Tmc",
    how="left",
)

merged_df = pd.merge(
    portal_data_filtered_df,
    inrix_enriched,
    left_on=["stationid", "starttime"],
    right_on=["stationid", "measurement_tstamp"],
    how="inner",
    suffixes=("_portal", "_inrix"),
)

merged_df["diff"] = merged_df["stationtt"] - merged_df["travel_time_seconds"]
```

The first 10 rows of the merged data:

```
  stationid        Tmc                 starttime
measurement_tstamp  \
0       230  114P04403 2023-04-21 23:15:00-07:00 2023-04-21
23:15:00-07:00
1       230  114P04403 2023-04-10 16:30:00-07:00 2023-04-10
16:30:00-07:00
2      3106  114P04401 2023-04-07 03:15:00-07:00 2023-04-07
03:15:00-07:00
3      3106  114P04401 2023-04-07 03:15:00-07:00 2023-04-07
03:15:00-07:00
4      3106  114P04401 2023-04-07 03:15:00-07:00 2023-04-07
03:15:00-07:00
5      3106  114+04401 2023-04-07 03:15:00-07:00 2023-04-07
03:15:00-07:00
6      3106  114+04401 2023-04-07 03:15:00-07:00 2023-04-07
03:15:00-07:00
7      3106  114+04401 2023-04-07 03:15:00-07:00 2023-04-07
03:15:00-07:00
8      3106  114P04401 2023-04-15 15:30:00-07:00 2023-04-15
15:30:00-07:00
9      3106  114P04401 2023-04-15 15:30:00-07:00 2023-04-15
15:30:00-07:00

   stationtt  travel_time_seconds   diff
0      41.55                 37.4   4.15
1     106.65                93.58  13.07
2       10.0                15.42  -5.42
3       10.0                15.42  -5.42
4       10.0                15.42  -5.42
5       10.0                21.82 -11.82
6       10.0                21.82 -11.82
7       10.0                21.82 -11.82
8       10.2                16.01  -5.81
9       10.2                16.01  -5.81
```

Where:

- `stationid`: The unique identifier for the PORTAL station.
- `Tmc`: The unique identifier for the INRIX TMC.
- `starttime`: The time of the observation (beginning of the 15-minute interval) for the PORTAL data.
- `measurement_tstamp`: The time of the observation (beginning of the 15-minute interval) for the INRIX data.
- `stationtt`: The travel time in minutes past the station for the PORTAL data.
- `travel_time_seconds`: The travel time in seconds for the INRIX data.

There are more columns available in the merged data, but this would be considered the bear minimum to compare travel times between the two datasources. These columns should also confirm that the time periods are correctly matched between the two time series. A full set of available columns can be found in the appendix C.3.

5 Applications of the Joined Dataset

With the core data cleaning and merging process now largely established, the combined dataset can be leveraged for analyses—some of which aligns directly with our initial objectives. Although the project's scope shifted toward establishing robust data-integration workflows, the effort invested in standardizing geometry, timestamps, and travel-time metrics between PORTAL and INRIX data now positions us to accomplish more extensive travel-time comparisons and incident-focused studies.

5.1 Revisiting the Original Objectives

1. **Evaluating Travel Time Trends (2019–2024)**
    - The merged PORTAL–INRIX dataset can be used to investigate how travel times evolve over multiple years. By filtering for specific corridors (e.g., I-205) or time ranges, one can visualize and quantify how speeds and congestion have changed—whether seasonally, annually, or in response to policy or infrastructure changes.
2. **Identifying Factors Underlying Data Differences**

    Because the final dataset preserves metadata on sensor coverage and segmentation, one can examine whether apparent discrepancies that arise from:

    - Sensor placement and lane coverage (e.g., stations vs. TMC segments).
    - Variations in data availability (e.g., missing intervals or shorter active date ranges).
    - Differences in road-segment definitions (e.g., one data source splitting a highway segment differently than another).

    These insights can help understand why one data source might capture fluctuations in travel time more consistently than another.

3. **Incident Analysis for Data Source Reliability**

   Incorporating the newly available incident data (which includes fields for incident location, lane closures, and duration) will allow testing how effectively each data source detects and reflects disruptions. By overlaying incident timestamps and locations onto the combined travel-time dataset, one could measure:

   - Lag time between when an incident occurs and when a data source's reported travel time rises significantly.
   - Completeness of detection (e.g., whether certain sources miss shorter-lived disruptions).
   - Recovery time once lanes reopen.

5.2 Leveraging Incident Data for Advanced Analyses

Incident data (detailing the when, where, and severity of traffic blockages) paves the way for sophisticated statistical and machine-learning methods, including:

1. **Changepoint Analysis**

   - Techniques such as changepoint detection can identify moments in time when the average speed or travel time distribution shifts. For instance, if a multi-lane blockage occurs, one might see abrupt changes in speed or volume data. Researchers can then directly link these "changepoints" to incident data, shedding light on how quickly each source's travel-time measurements respond to lane closures or severe congestion.

2. **Maximum Mean Discrepancy (MMD)**

   - MMD is a non-parametric method for comparing two probability distributions. In the context of our project, it could be applied to compare the distribution of travel-time readings before vs. after an incident, or to measure how different data sources respond to similar incidents. By sampling data from each source during the same time windows, analysts could quantify whether the distributions of travel times differ significantly, thereby revealing how much each data source "agrees" on the severity and duration of an incident's impact.

A Data Inconsistencies

A.1 INRIX Metadata Inconsistencies Tables

(a) Unique column value counts per year columns 1 to 5

	road	direction	intersection	state	county
2019	139	139	139	139	139
2020	139	139	139	139	139
2021	139	139	139	139	139
2022	139	139	139	139	139
2023	139	139	139	139	139
2024	139	139	139	139	139

(b) Unique column value counts per year columns 6 to 10

	zip	start_latitude	start_longitude	end_latitude	end_longitude
2019	139	139	139	139	139
2020	139	139	139	153	153
2021	139	139	139	139	139
2022	139	142	142	147	148
2023	139	157	154	157	155
2024	142	165	174	172	179

(c) Unique column value counts per year columns 10 to 15

	miles	road_order	timezone_name	type	country
2019	146	139	139	139	139
2020	249	139	139	139	139
2021	140	139	139	139	139
2022	171	139	139	139	139
2023	176	139	139	139	139
2024	278	139	139	139	139

(d) Unique column value counts per year columns 16 to end

	active_start_date	active_end_date
2019	417	417
2020	417	417
2021	278	278
2022	278	278
2023	278	278
2024	278	139

B Metadata Columns

B.1 Portal Metadata

B.1.1 Stations

All available columns in the stations metadata:

```
['stationid',
 'highwayid',
 'milepost',
 'locationtext',
 'length',
 'upstream',
 'downstream',
 'numberlanes',
 'length_mid',
 'downstream_mile',
 'upstream_mile',
 'agencyid',
 'opposite_stationid',
 'segment_geom',
 'station_geom',
 'start_date',
 'end_date',
 'lon',
 'lat',
 'detectortype',
 'detectorlocation',
 'agency',
 'region',
 'active_dates',
 'id',
 'station_location_id']
```

B.1.2 Highways

All available columns in the highways metadata:

```
['highwayid',
 'direction',
 'highwaylength',
 'highwayname',
 'startmp',
 'endmp',
 'bound',
 'oppositehighwayid',
 'milepost_direction',
 'end_location',
 'start_location',
 'show_on_site']
```

B.1.3 Detectors

All available columns in the detectors metadata:

```
['detectorid',
 'highwayid',
 'enabledflag',
 'locationtext',
 'detectortype',
 'detectorclass',
 'detectortitle',
 'detectorstatus',
 'lanenumber',
 'stationid',
 'rampid',
 'controllerid',
 'end_date',
 'start_date',
 'milepost',
 'agency_lane',
 'atms_id',
 'active_dates']
```

B.2 INRIX Metadata

B.2.1 Shape file

All available columns in the shape file:

```
['MVVersionI',
 'FRC',
 'Tmc',
 'Type',
 'RoadNumber',
 'RoadNumb_1',
 'RoadName',
 'FirstName',
 'LinearTMC',
 'Country',
 'State',
 'County',
 'Zip',
 'Direction',
 'StartLat',
 'StartLong',
 'EndLat',
 'EndLong',
 'Miles',
 'geometry']
```

B.2.2 TMC Identification Files

All available columns in the TMC identification files:

```
['tmc',
 'road',
 'direction',
 'intersection',
 'state',
 'county',
 'zip',
 'start_latitude',
 'start_longitude',
 'end_latitude',
 'end_longitude',
 'miles',
 'road_order',
 'timezone_name',
 'type',
 'country',
 'active_start_date',
 'active_end_date']
```

C Timeseries Data Columns

C.1 PORTAL Timeseries Data

All available columns in the PORTAL timeseries data:

```
['stationid', 'stationtt', 'stationcountreadings', 'starttime']
```

C.2 INRIX Timeseries Data

All available columns in the INRIX timeseries data:

```
['tmc_code',
 'measurement_tstamp',
 'speed',
 'historical_average_speed',
 'reference_speed',
 'travel_time_seconds',
 'confidence_score',
 'cvalue']
```

C.3 Time and Spatial Joined Columns

All available columns in the time and spatial joined data:

```
['stationid',
 'stationtt',
 'stationcountreadings',
 'starttime',
 'tmc_code',
 'measurement_tstamp',
 'speed',
 'historical_average_speed',
 'reference_speed',
 'travel_time_seconds',
 'confidence_score',
 'cvalue',
 'Tmc',
 'diff']
```

D Spatial Join Visualizations

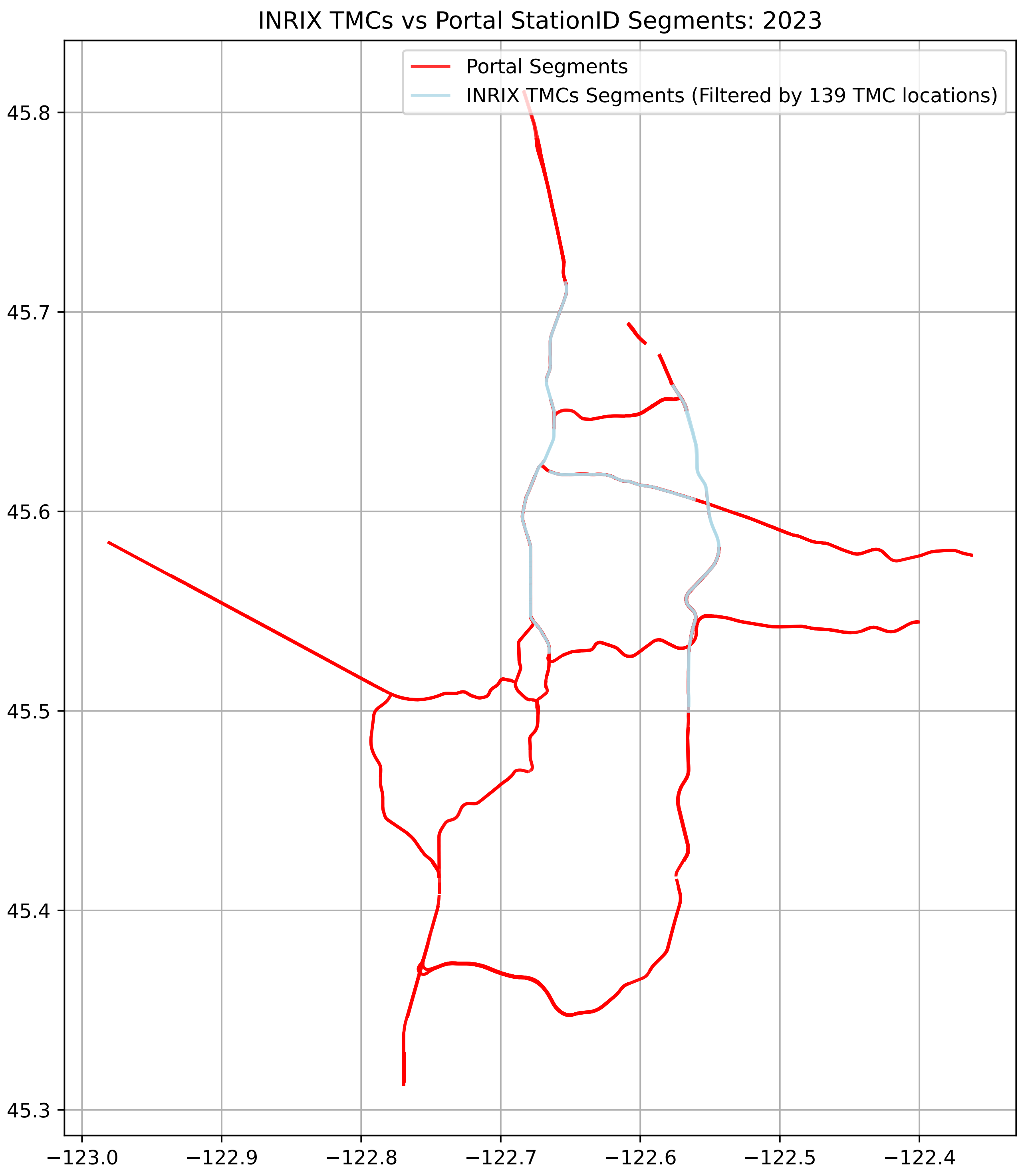


Figure 3: Plot of the INRIX TMCs and PORTAL stations for which we have geometry data in 2023.

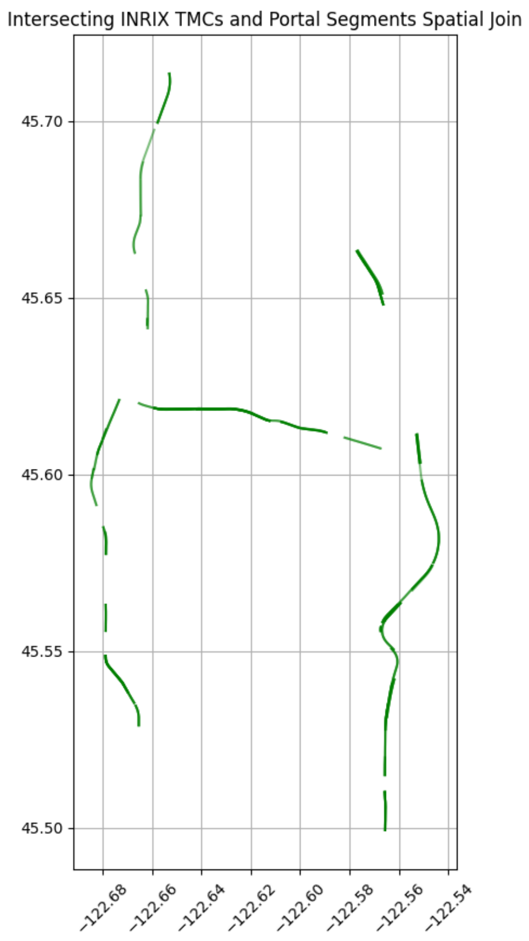


Figure 4: Intersecting segments plot of longitude and latitudes.

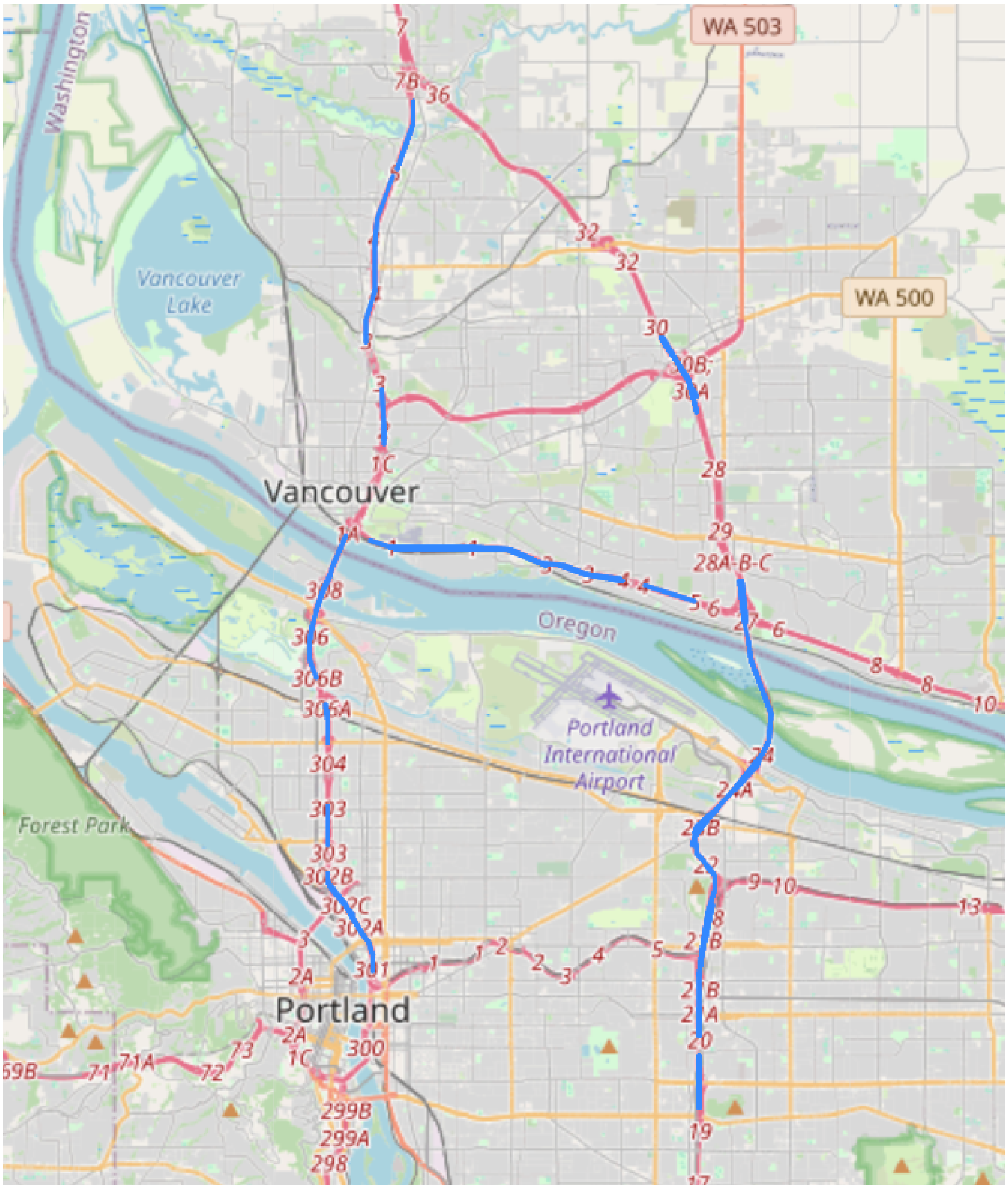


Figure 5: Spatial Join of PORTAL and INRIX Metadata on reference map.

E Code

The code samples are also included in the appendix for reference and context. The code is availabe in the GitHub repository https://github.com/whitham-powell/cades-traveltime-compare. The code on the repository includes the helper class and may continue to be updated.

E.1 Segment Conversion and Spatial Join Code Demo

demo_sjoin_portal_inrix_meta.py

```python
import struct

import geopandas as gpd
import matplotlib.pyplot as plt
import pandas as pd
from shapely.wkb import loads as wkb_loads

from datafiles import (
    PortalInrixDataFiles,
)  # This is the helper class to manage the file paths


def standardize_direction(direction, bound=None):
    """
    Standardize direction formats between PORTAL and INRIX
    """
    direction_map = {
        # INRIX formats
        "NORTHBOUND": "NORTH",
        "SOUTHBOUND": "SOUTH",
        "WESTBOUND": "WEST",
        "EASTBOUND": "EAST",
        # PORTAL formats
        "NORTH": "NORTH",
        "SOUTH": "SOUTH",
        "WEST": "WEST",
        "EAST": "EAST",
        "NORT": "NORTH",  # Handle the truncated 'NORT'
        "CONST": None,  # Handle construction case separately
    }

    # If direction is not valid, try to use bound
    bound_map = {
        "NB": "NORTH",
        "SB": "SOUTH",
        "WB": "WEST",
        "EB": "EAST",
        "JB": None,  # Special cases
        "ZB": None,
    }

    std_direction = direction_map.get(direction.upper())
    if std_direction is None and bound is not None:
        std_direction = bound_map.get(bound.upper())

    return std_direction


def check_ewkb_srid(wkb_hex):
    # Convert hex to bytes
    wkb_bytes = bytes.fromhex(wkb_hex)

    # Check endianness (byte order)
    endian = ">" if wkb_bytes[0] == 0 else "<"

    # Check if it's EWKB (has SRID)
    has_srid = bool(struct.unpack(endian + "I", wkb_bytes[1:5])[0] & 0x20000000)

    if has_srid:
        # SRID is stored after the type indicator
        srid = struct.unpack(endian + "I", wkb_bytes[5:9])[0]
        return srid
    return None


def wkb_to_geom(wkb_hex):
    try:
        if pd.isna(wkb_hex) or not wkb_hex.strip():
            return None
        geom = wkb_loads(bytes.fromhex(wkb_hex))
        # print(f"ekwb_srid: {check_ewkb_srid(wkb_hex)}") # Uncomment to check SRID
        return geom
    except (ValueError, TypeError) as e:
        print(f"Conversion error: {e}")
        return None


# The datafiles object is a helper class to manage the file paths and
# the file summary methods should confirm it found the files expected.
# example_datafiles.get_* methods return the file paths and could be replace
# with the actual file paths if you prefer.

example_datafiles = PortalInrixDataFiles(data_root="../data/CADES_DATA")
example_datafiles.portal_file_summary()
example_datafiles.inrix_file_summary()
```

```python
# Load CSVs
portal_stations_df = pd.read_csv(example_datafiles.get_portal_meta("stations"))
portal_highways_df = pd.read_csv(example_datafiles.get_portal_meta("highways"))
# pd.read_csv() can be replaced with pd.read_sql() if you have a database connection
# However, the SQL query would need to be written to extract the same data as the CSVs
# and there is additional setup required for the database connection as seen in the
# pandas documentation.
# - https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html#pandas.read_sql

# Convert hex WKB to geometries
portal_stations_df["segment_geom"] = portal_stations_df["segment_geom"].apply(
    wkb_to_geom
)
portal_stations_df["station_geom"] = portal_stations_df["station_geom"].apply(
    wkb_to_geom
)

# After your wkb_to_geom conversions
print(f"Total rows in DataFrame: {len(portal_stations_df)}")
print(
    f"Successful segment conversions: {portal_stations_df['segment_geom'].notna().sum()}"
)
print(
    f"Successful station conversions: {portal_stations_df['station_geom'].notna().sum()}"
)

# Merge highways data for additional station direction coverage and highway names
portal_stations_df = portal_stations_df.merge(
    portal_highways_df[["highwayid", "direction", "bound", "highwayname"]],
    on="highwayid",
    how="left",
)

# Standardize direction formats
portal_stations_df["standardized_direction"] = portal_stations_df.apply(
    lambda row: standardize_direction(row["direction"], row["bound"]),
    axis=1,
)

# Convert to GeoDataFrame with Web Mercator CRS
the_CRS = "EPSG:3857"
portal_gdf = gpd.GeoDataFrame(portal_stations_df, geometry="segment_geom", crs=the_CRS)

# Load INRIX TMC Shapefiles - Oregon and Clark County, WA
oregon_gdf = gpd.read_file(
    "data/CADES_DATA/CADES_INRIX/INRIX_TMC_Shapefile-2023_Oregon/Oregon_2301TMC/OREGON.shp"
```

```
)

clark_gdf = gpd.read_file(
    "data/CADES_DATA/CADES_INRIX/INRIX_TMC_Shapefile-2023_ClarkCountyWA/ClarkCountyWA2301TMC.shp
)

# Convert all to a common CRS ("EPSG:4326" lon/lat)
oregon_gdf = oregon_gdf.to_crs("EPSG:4326")
clark_gdf = clark_gdf.to_crs("EPSG:4326")
portal_gdf = portal_gdf.to_crs("EPSG:4326")

# Specify the year for INRIX data
year_str = "2023"

# Load INRIX TMC Identification data this is used to find the unique TMCs
# to filter to the same which are the given 139 TMCs of interest.
tmc_identification_df = pd.read_csv(example_datafiles.get_inrix_meta(year_str))

# Merge Oregon and Clark County TMCs into 1 dataframe
inrix_full_df = pd.concat([oregon_gdf, clark_gdf], ignore_index=True)
inrix_full_df["standardized_direction"] = inrix_full_df.apply(
    lambda row: standardize_direction(row["Direction"]), axis=1
)
unique_tmc_ids = tmc_identification_df.tmc.unique().tolist()
inrix_filtered_by_tmc = inrix_full_df[inrix_full_df["Tmc"].isin(unique_tmc_ids)]

portal_inrix_spatial_join = inrix_filtered_by_tmc.sjoin(
    portal_gdf, how="inner", predicate="intersects", lsuffix="inrix", rsuffix="portal"
)

# Visualize the segments to ensure it looks correct
fig, ax = plt.subplots(figsize=(12, 10))
portal_gdf.plot(
    ax=ax,
    color="red",
    edgecolor="red",
    alpha=0.1,
    markersize=10,
    label="Portal Segments",
)
inrix_filtered_by_tmc.plot(
    ax=ax,
    color="lightblue",
    edgecolor="lightblue",
    alpha=0.5,
    markersize=1,
    label="INRIX TMCs Segments (Filtered by 139 TMC locations)",
)

plt.title("INRIX TMCs vs Portal StationID Segments: 2023")
plt.legend()
plt.grid(True)
plt.show()

# Visualize the spatial join to ensure it looks correct
fig, ax = plt.subplots(figsize=(12, 10))
portal_inrix_spatial_join.plot(
    ax=ax, color="green", edgecolor="green", alpha=0.5, markersize=10
)
plt.title("Intersecting INRIX TMCs and Portal Segments Spatial Join")
plt.grid(True)
plt.xticks(rotation=45)
plt.show()

# List of columns to save to the file (can be modified as needed)
columns_of_interest = [
    "Tmc",
    "Direction",
    "stationid",
    "highwayid",
    "RoadNumb_1",
    "direction",
    "bound",
    "standardized_direction_portal",
    "standardized_direction_inrix",
    "StartLat",
    "StartLong",
    "milepost",
    "lon",
    "lat",
    "highwayname",
]

# Select columns of interest
portal_inrix_spatial_join[columns_of_interest]

# Save the spatial join to a CSV
# - comment out this line if you don't want to save to file
portal_inrix_spatial_join.to_csv("data/portal_inrix_spatial_join.csv")
```

E.2 Timeseries Data Processing Code Demo

demo_process_timeseries.py

```python
import re
import pandas as pd
import matplotlib.pyplot as plt

from datafiles import PortalInrixDataFiles


# Function to ensure timezone offsets include minutes (e.g. "-08" becomes "-08:00")
def normalize_timezone(timestamp):
    return re.sub(r"([-+]\d{2})$", r"\1:00", timestamp)


# Function to insert default microseconds if not present
def add_default_microseconds(timestamp):
    if "." not in timestamp:
        match = re.search(r"([-+]\d{2}:\d{2})$", timestamp)
        if match:
            tz = match.group(1)
            timestamp = timestamp[: match.start()] + ".000000" + tz
    return timestamp


# Combined normalization function
def normalize_timestamp(timestamp):
    if pd.isna(timestamp):
        return None
    timestamp = normalize_timezone(timestamp)
    timestamp = add_default_microseconds(timestamp)
    return timestamp


# Specify the route name, year, and bounds, and the start and end timestamps
# of the data to be processed
year = 2023
year_str = str(year)
route_name = "I205"
adjusted_route_name = re.sub(r"([A-Za-z]+)(\d+)", r"\1-\2", route_name)
bound_1 = "NB"
bound_2 = "SB"

start_dt = pd.Timestamp(
    year=year,
    month=10,
    day=1,
    hour=0,
    minute=0,
    second=0,
    tz="America/Los_Angeles",
)
end_dt = pd.Timestamp(
    year=year,
    month=12,
    day=31,
    hour=23,
    minute=59,
    second=59,
    tz="America/Los_Angeles",
)

meta_columns_of_interest = [
    "Tmc",
    # "LinearTMC",
    "Direction",
    "stationid",
    "highwayid",
    "RoadNumb_1",
    "direction",
    "Miles",
    "bound",
    "standardized_direction_portal",
    "standardized_direction_inrix",
    "StartLat",
    "StartLong",
    "EndLat",
    "EndLong",
    "milepost",
    # "length",
    "lon",
    "lat",
    "highwayname",
    "start_date",
    "end_date",
    "active_dates",
]

# Initialize the data files object as in the sjoin script again with
# caveat that this can be replaced with the actual file paths if preferred.
example_datafiles = PortalInrixDataFiles(data_root="../data/CADES_DATA")
example_datafiles.portal_file_summary()
example_datafiles.inrix_file_summary()
```

```
# The data can be accessed from a database directly.
# pd.read_csv() can be replaced with pd.read_sql() if you have a database connection
# However, the SQL query would need to be written to extract the same data as the CSVs
# and there is additional setup required for the database connection as seen in the
# pandas documentation.
# - https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html#pandas.read_sql

# Load the spatial join of the portal and INRIX data created
# in the demo_sjoin_portal_inrix_meta.py script
sjoined_portal_inrix_df = pd.read_csv("data/portal_inrix_spatial_join.csv")

# Ensure the data types are correctly set and normalize timestamps
sjoined_portal_inrix_df = sjoined_portal_inrix_df.convert_dtypes()
sjoined_portal_inrix_df["stationid"] = sjoined_portal_inrix_df["stationid"].astype(
    "string"
)
sjoined_portal_inrix_df["start_date"] = sjoined_portal_inrix_df["start_date"].apply(
    normalize_timestamp
)
sjoined_portal_inrix_df["end_date"] = sjoined_portal_inrix_df["end_date"].apply(
    normalize_timestamp
)

# Convert the start and end dates to datetime objects and convert to Pacific time
sjoined_portal_inrix_df["start_date"] = pd.to_datetime(
    sjoined_portal_inrix_df["start_date"],
    utc=True,
    errors="coerce",
).dt.tz_convert("America/Los_Angeles")
sjoined_portal_inrix_df["end_date"] = pd.to_datetime(
    sjoined_portal_inrix_df["end_date"],
    utc=True,
    errors="coerce",
).dt.tz_convert("America/Los_Angeles")


print(f"Processing {route_name} {year_str} {bound_1} and {bound_2} data")
print(
    f"Portal data files: {example_datafiles.get_portal_data(route_name, year_str, bound_1)}"
)
print(f"Building dataframe (concat 1) {bound_1}")

# Load the portal data files for the specified route, year, and bounds
# and concatenate them into a single DataFrame.
bound_1_portal_data_df = pd.concat(
    [
        pd.read_csv(file)
        for file in example_datafiles.get_portal_data(route_name, year_str, bound_1)
    ]
)

print(f"Building dataframe (concat 2) {bound_2}")
bound_2_portal_data_df = pd.concat(
    [
        pd.read_csv(file)
        for file in example_datafiles.get_portal_data(route_name, year_str, bound_2)
    ]
)

print(f"Building dataframe (concat ) {bound_1} and {bound_2}")
portal_data_df = pd.concat([bound_1_portal_data_df, bound_2_portal_data_df])

print(f"Portal data shape: {portal_data_df.shape}")
print(f"Portal data initial dtypes:\n{portal_data_df.dtypes}, converting...")

# Convert the data types and normalize the timestamps
portal_data_df = portal_data_df.convert_dtypes()
portal_data_df["stationid"] = portal_data_df["stationid"].astype("string")

print(f"Processing timestamp")
portal_data_df["starttime"] = portal_data_df["starttime"].apply(normalize_timestamp)
portal_data_df["starttime"] = pd.to_datetime(
    portal_data_df["starttime"],
    utc=True,
    errors="coerce",
).dt.tz_convert("America/Los_Angeles")
portal_data_df["stationtt"] = (
    portal_data_df["stationtt"] * 60
)  # convert from minutes to seconds

print(
    f"dtype and timezone conversion complete \n Portal data final dtypes:\n{portal_data_df.dtype
)
print(portal_data_df.head(5))

print(f"INRIX data files: {example_datafiles.get_inrix_data(year_str)}")
inrix_data_df = pd.read_csv(example_datafiles.get_inrix_data(year_str))

print(f"INRIX data shape: {inrix_data_df.shape}")
print(f"INRIX data initial dtypes:\n{inrix_data_df.dtypes}, converting...")
inrix_data_df = inrix_data_df.convert_dtypes()

print(f"Processing timestamp")
old_inrix_timestamps = inrix_data_df["measurement_tstamp"].copy()
```

```
inrix_data_df["measurement_tstamp"] = pd.to_datetime(
    inrix_data_df["measurement_tstamp"],
).dt.tz_localize("America/Los_Angeles", ambiguous=True)

print(
    f"dtype and timezone conversion complete \n INRIX data final dtypes:\n{inrix_data_df.dtypes}
)
print(inrix_data_df.head(5))

# Filter metadata and data by route name and year
portal_inrix_meta_df_filtered = sjoined_portal_inrix_df[meta_columns_of_interest][
    (sjoined_portal_inrix_df.RoadNumb_1 == adjusted_route_name)
    & (sjoined_portal_inrix_df.start_date.dt.year <= year)
    & (
        (sjoined_portal_inrix_df.end_date.dt.year >= year)
        | (
            sjoined_portal_inrix_df.end_date.isna()
        )  # It was assumed that if the end date is NA, the segment is still active
    )
]

# Filter INRIX data by TMCs of interest and timestamp range (start_dt to end_dt)
tmcs_of_interest = portal_inrix_meta_df_filtered["Tmc"].unique().tolist()
print(f"INRIX TMCs of interest: {tmcs_of_interest}")

inrix_data_filtered_df = inrix_data_df[
    (inrix_data_df.tmc_code.isin(tmcs_of_interest))
    & inrix_data_df["measurement_tstamp"].between(start_dt, end_dt, inclusive="both")
]

print(
    f"INRIX data filtered by TMC codes of interest and timestamp: {start_dt} to {end_dt} inclusi
)
print(inrix_data_filtered_df.head())

# Filter PORTAL data by station IDs of interest and timestamp range (start_dt to end_dt)
station_ids_of_interest = portal_inrix_meta_df_filtered["stationid"].unique().tolist()
print(f"PORTAL station ids of interest: {station_ids_of_interest}")

portal_data_filtered_df = portal_data_df[
    (portal_data_df.stationid.isin(station_ids_of_interest))
    & portal_data_df["starttime"].between(start_dt, end_dt, inclusive="both")
]
print(
    f"PORTAL data filtered by station IDs of interest and timestamp: {start_dt} to {end_dt} incl
)
print(portal_data_filtered_df.head())
```

```
# Merge the filtered portal and INRIX dataframes with the spatial join metadata
# Done in this order to preserve timestamp ordering
# Step 1 - Merge INRIX data with spatial join metadata
inrix_enriched = pd.merge(
    inrix_data_filtered_df,
    sjoined_portal_inrix_df[["Tmc", "stationid"]],
    left_on="tmc_code",
    right_on="Tmc",
    how="left",
)

print(inrix_enriched[["tmc_code", "Tmc", "stationid"]].head())
# Step 2 - Merge portal data with spatial join metadata
merged_df = pd.merge(
    portal_data_filtered_df,
    inrix_enriched,
    left_on=["stationid", "starttime"],
    right_on=["stationid", "measurement_tstamp"],
    how="inner",
    suffixes=("_portal", "_inrix"),
)

# Calculate the difference between the station travel time and the INRIX travel time
# Not strictly necessary, but can be useful for analysis
merged_df["diff"] = merged_df["stationtt"] - merged_df["travel_time_seconds"]

# Select the minimal set of columns used to compare travel times
merged_df[
    [
        "stationid",
        "Tmc",
        "starttime",
        "measurement_tstamp",
        "stationtt",
        "travel_time_seconds",
    ]
].head(10)

merged_df.to_csv(
    f"{route_name}_{bound_1}_{bound_2}__{start_dt:%Y%m%d}-{end_dt:%Y%m%d}_merged.csv",
    index=False,
)
```