CTA Service Cuts Problem Set Solutions

Peter Ganong, Felix Farb and Alex Lan

As of September 2025, the Chicago Transit Authority was facing a major budget shortfall which was going to require a 40% cut in service. Here is the Chicago Sun-Times coverage of the issue. Acting President Nora Leerhsen has asked you to teach her about different ways to achieve that service reduction.[1]

She has told you that she cares about both about efficiency, by which she means affecting the smallest number of riders possible, and equity. Although equity has many different definitions, she suggests that using income is sufficient. Here are the datasets you will use, some of which are stored in this repo:

- Scheduled trips from Chicago Transit Authority in General Transit Feed Specification (GTFS)
- Passengers' actual trips on CTA in Chicago from MIT Transit Lab. These were recently used in the economics paper "Optimal Urban Transportation Policy: Evidence from Chicago' '. We are grateful to one of the authors (Juan Camilo Castillo) for sharing it with us. The appendix describes the data in a bit more detail.
- Median Household income by Census tract from the 2023 5 year American Community Survey (ACS). Taken from the NHGIS website.

```
# setup
import geopandas as gpd
from os.path import join
import altair as alt
import pandas as pd
import numpy as np
import time

import warnings
warnings.filterwarnings('ignore')
alt.renderers.enable("png")
```

[1] Of course, the same dataset and questions could also be used to think about expanding service in a different budget environment.

```
alt.data_transformers.disable_max_rows()

data_path = 'data/external'

# improve graph Resolution
import tempfile
from IPython.display import SVG, display, Image
import vl_convert as vlc

def display_altair_png(chart, scale=2):
    """
    Render an Altair chart to a PNG and display it inline.

    Parameters
    ----------
    chart : altair.Chart
        Altair chart object to render.
    scale : int, optional
        Resolution scaling factor for the PNG (default = 2). Use scale=2 for
↪   standard slides. Use scale = 3-4 for dense figures or PDF exports.
    """
    png_bytes = vlc.vegalite_to_png(chart.to_dict(), scale=scale)

    # Write to a temporary PNG file and display
    with tempfile.NamedTemporaryFile(suffix=".png") as tmp:
        tmp.write(png_bytes)
        tmp.flush()
        display(Image(filename=tmp.name))
```

Problem 1 – how well-utilized is each route?

Part a – number of scheduled vehicle trips

Write a function `calculate_trip_counts()` which returns a dataframe called `routes_w_trips` with the number of scheduled bus and train trips each week along each route by collapsing `trips.txt` to the route level and attaching labels using `routes.txt`.

Required diagnostics to check that you are on track:

- Print the number of routes with schedules in `routes.txt` and the number of routes with actual trips in `trips.txt`.

- Count if there are any routes with trips that don't have schedules and vice-versa. If a route does have no trips, then fill in the number of trips as being zero.
- Expect 133 rows in the combined dataframe.

Solution

Note that we only care about the number of trips along each route, which you can calculate using the `route_id` column in `routes.txt`. We could also get a data frame of trips by simply grouping the `trips.txt` dataframe by `route_id`.

However, there are two reasons we still use `routes.txt`. First is that we also want the route names from `routes.txt`, and second is that we would miss routes with no trips.

```
def calculate_trip_counts(verbose = False):
    trips_file = join(data_path, "trips.txt")
    routes_file = join(data_path, "routes.txt")

    # Read data (ensure route_id comparable across files)
    trips_df  = pd.read_csv(trips_file, dtype={"route_id": str})
    routes_df = pd.read_csv(routes_file, dtype={"route_id": str})

    # Count trips by route_id
    trip_counts = (
        trips_df["route_id"]
        .value_counts()
        .rename_axis("route_id")
        .reset_index(name="trip_count")
    )

    # Merge routes with trip counts
    routes_w_trips = routes_df.merge(trip_counts, on="route_id", how="left")

    routes_w_trips["trip_count"] =
↪   routes_w_trips["trip_count"].fillna(0).astype(int)

    # Stats for the two different datasets
    routes_in_routes = routes_df["route_id"].nunique()
    routes_in_trips  = trips_df["route_id"].nunique()
    zero_trip_routes = int((routes_w_trips["trip_count"] == 0).sum())

    # We transform into sets due to the ease of set arithmetic
    routes_set = set(routes_df["route_id"])
    trips_set  = set(trips_df["route_id"])
    trips_not_in_routes = len(trips_set - routes_set)
```

```
    if verbose:
        print(f"  routes.txt: {routes_in_routes} distinct route_id
        ↪  (rows={len(routes_df)})")
        print(f"  trips.txt:  {routes_in_trips} distinct route_id
        ↪  (trips={len(trips_df)})")
        print(f"  Routes with zero trips (present in routes.txt only):
        ↪  {zero_trip_routes}")
        print(f"  route_id present in trips.txt but missing from routes.txt:
        ↪  {trips_not_in_routes}")
    return routes_w_trips

routes_df = calculate_trip_counts(verbose=True)
print("Final dataframe length:")
print(len(routes_df.index))
```

```
  routes.txt: 133 distinct route_id (rows=133)
  trips.txt:  133 distinct route_id (trips=93651)
  Routes with zero trips (present in routes.txt only): 0
  route_id present in trips.txt but missing from routes.txt: 0
Final dataframe length:
133
```

Part b – number of passenger trips

Next, calculate the number of passenger trips along each route using `bus_passengers.csv` and `train_passengers.csv`.

Required diagnostics to check that you are on track:

- When first reading each dataset, print the number of rows. Each row corresponds to one boarding. Many journeys require multiple boardings.
- Which dataset has journeys that mix different modes (e.g. a train boarding followed by a bus boarding) within a single journey?
- Combine the two datasets. You should have 1236508 rows in the combined dataframe of bus and train passenger trips.

Then, calculate each route's utilization rate (passenger trips count divided by scheduled trips count).[2]

[2]One slightly awkward thing: the utilization rates are *relative*. What this means is that we do not know (and indeed Juan Camilo who gave us the data said he did not know) how many days of data contribute to this data file. This is the kind of thing that happens very frequently with data—you save just the columns that you need for your analysis and columns that a future user might want are lost. Fortunately, even without

- You will need to find a suitable variable to construct the number of passengers on each route.
- Merge diagnostics: print which scheduled routes are missing in the passenger routes data, and which passenger routes are missing in the scheduled routes data.
- Examine the list you printed carefully. Are there any examples where the same route has different names in the schedule data and the passenger data? If needed, harmonize the names.
- Produce matched data frame `utilization_df`. You should see 126 matched routes.

```
# Function that loads train and bus boarding files and then creates a dataset
 ↪  combining the two together
def read_passengers_df()
    # ...
    return passengers_df

# Function to count the number of passengers who take each route. Takes in a
 ↪  combined dataframe of bus and train passengers
def calculate_passenger_counts(passengers_df):
    # ...
    return passenger_count_df

def calculate_utilization_rates(passengers_df):

    passenger_count_df = calculate_passenger_counts(passengers_df)
    # ...

    return utilization_df
```

Solution

`train_passengers.csv` has journeys that mix different modes. To count passengers per route, we group by `line_string` in the passenger data, which corresponds to `route_id` in the routes data.

```
def read_passengers_df():

    trains_file = join(data_path, "train_passengers.csv")
    buses_file = join(data_path, "bus_passengers.csv")
    # Load trips
```

knowing the dates used for the data and therefore not knowing the absolute utilization rate, we still are able to figure out which routes are more or less in use, which is the information the CTA chief needs to decide which routes should see cuts in service.

```
    train_passengers_df = pd.read_csv(trains_file)
    print("Number of train passenger trips: ",len(train_passengers_df))

    bus_passengers_df = pd.read_csv(buses_file)
    print("Number of bus passenger trips: ", len(bus_passengers_df))

    passengers_df = pd.concat([train_passengers_df, bus_passengers_df],
↪   ignore_index=True)
    passengers_df.drop(columns=['mode_type'], inplace=True)
    print("Total number of passenger trips: ",len(passengers_df))
    return passengers_df


def calculate_passenger_counts(passengers_df):

    passenger_count_df = (
        passengers_df
        .groupby("line_string", as_index=False, sort=False)
        .agg(
            passenger_count=("line_string", "size")
        )
    )

    return passenger_count_df

passengers_df = read_passengers_df()
```

```
Number of train passenger trips:  622722
Number of bus passenger trips:  613786
Total number of passenger trips:  1236508
```

Before merging the passenger data and the routes data, let's do some diagnostics to see which routes are missing in either dataset:

```
# Merge diagnostics
passenger_count_df = calculate_passenger_counts(passengers_df)
trips_count_df = calculate_trip_counts()

routes_set = set(trips_count_df['route_id'].dropna())
passengers_set = set(passenger_count_df['line_string'].dropna())
```

```
unmatched_routes = routes_set - passengers_set
unmatched_passengers = passengers_set - routes_set

print(f"Routes with no passenger data: {len(unmatched_routes)}")
if unmatched_routes:
    unmatched_df =
 ↪  trips_count_df[trips_count_df['route_id'].isin(unmatched_routes)][['route_id',
 ↪  'route_long_name']]
    print(unmatched_df.to_string(index=False))

print(f"Passenger lines not in routes: {len(unmatched_passengers)}")
if unmatched_passengers:
    print(f"  {sorted(unmatched_passengers)[:10]}")
```

```
Routes with no passenger data: 13
route_id               route_long_name
      X4         Cottage Grove Express
      N5         South Shore Night Bus
      10  Museum of Science & Industry
     120 Ogilvie/Streeterville Express
     121   Union/Streeterville Express
     125           Water Tower Express
     201                 Central/Ridge
     206           Evanston Circulator
       P                  Purple Line
       Y                  Yellow Line
       G                   Green Line
     Org                  Orange Line
     Brn                   Brown Line
Passenger lines not in routes: 6
  ['5', 'Brown', 'Green', 'Orange', 'Purple', 'Yellow']
```

The passenger data uses full names for train lines (e.g. `'Brown'`) while routes data uses abbreviations (e.g. `'Brn'`). We also need to map `'5'` to `'N5'` (the South Shore Night Bus). Rename these before merging:

```
line_string_rename = {
    'Brown': 'Brn',
    'Green': 'G',
    'Orange': 'Org',
    'Purple': 'P',
```

```
    'Yellow': 'Y',
    '5': 'N5',
}
passengers_df['line_string'] =
 ↪  passengers_df['line_string'].replace(line_string_rename)
```

With passenger trips and trip counts for each route, we can now calculate utilization rates:

```
def calculate_utilization_rates(passengers_df):

    passenger_count_df = calculate_passenger_counts(passengers_df)
    trips_count_df = calculate_trip_counts()

    utilization_df = (
        trips_count_df[["route_id", "trip_count","route_long_name"]]
        .merge(passenger_count_df, how="inner",
            left_on="route_id", right_on="line_string")
    )

    utilization_df["utilization_rate"] = (
        utilization_df["passenger_count"] / utilization_df["trip_count"]
    )

    return utilization_df

utilization_df = calculate_utilization_rates(passengers_df)
```

Problem 2 – which routes have the lowest utilization? (efficiency)

First, calculate the 40th percentile of trips by utilization rate. For this calculation, assume each trip has the same utilization rate as the route as a whole.

Second, make a histogram of trips (not routes) with 30 bins of the utilization rate. Suggestions and questions:

- Feel free to use pandas or altair manipulation functions – whatever you find easier.
- You will notice some trips with far higher utilization rates that will mess with the histogram if you don't winsorize the data. What do those trips correspond to?
- Taking the 40th percentile of trip-weighted utilization as a cutoff, color the bars below the 40th percentile cutoff in red. Given the binning structure, it is impossible to color exactly 40% of the trips red. If you use utilization bins of width 1, you will find that you can color either 37.9% or 42.5% of the trips red.

What are the 5 routes with lowest utilization rates?

Here is a skeleton of functions to write:

```
# Function to return the qth percentile of trips by utilization (we will use
 ↪  the 40th percentile)
def weighted_quantile(values, weights, q):

    return p40_utilization


def make_hist(utilization_df):

    return figure
```

Solution

Let's first look at the top 10 routes by utilization rate:

```
# Top utilization rate routes
top_routes = (
    utilization_df[['route_id', 'route_long_name', 'utilization_rate',
 ↪  'trip_count', 'passenger_count']]
    .sort_values('utilization_rate', ascending=False)
    .head(10)
)
print(top_routes.to_string(index=False))
```

route_id	route_long_name	utilization_rate	trip_count	passenger_count
Org	Orange Line	186.620968	372	69423
Red	Red Line	147.117757	1070	157416
G	Green Line	91.511299	708	64790
Blue	Blue Line	86.981250	1120	97419
N5	South Shore Night Bus	78.731707	82	6456
Brn	Brown Line	60.595238	420	25450
Pink	Pink Line	57.042169	332	18938
30	South Chicago	49.708126	603	29974
169	69th/UPS Express	37.181818	22	818
56	Milwaukee	32.394057	774	25073

The outlier trips correspond to trains. A single train carries far more passengers than a bus. The advantages of buses over trains are not reflected in these data.

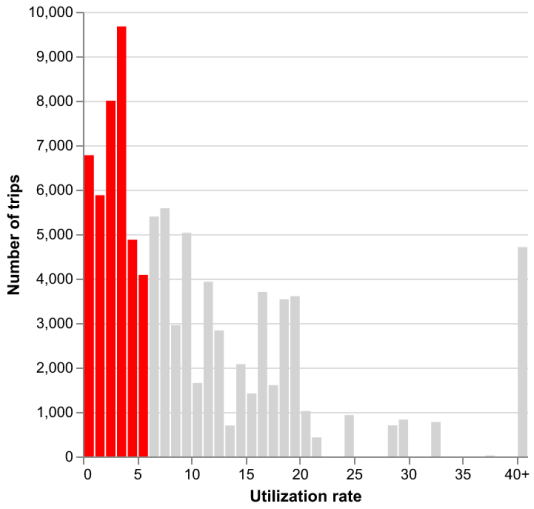
Here, we color bins red until at least 40% of trips have been highlighted.

```
def weighted_quantile(values, weights, q):
    sorter = np.argsort(values)
    values = np.asarray(values)[sorter]
    weights = np.asarray(weights)[sorter]
    cumsum = weights.cumsum()
    cutoff = q * weights.sum()
    return values[cumsum >= cutoff][0]

p40_utilization = weighted_quantile(
    utilization_df["utilization_rate"],
    utilization_df["trip_count"],
    0.40,
)
print(f"Weighted 40th percentile utilization: {p40_utilization:.4f}")

def make_hist(utilization_df):
    df = utilization_df.copy()
    df["utilization_rate"] = pd.to_numeric(df["utilization_rate"],
 ↪  errors="coerce")
    df["trip_count"] = pd.to_numeric(df["trip_count"], errors="coerce")
    df = df.dropna(subset=["utilization_rate", "trip_count"])

    step = 1  # 1-unit bins from 0..40, with a final 40+ bin

    alt.data_transformers.disable_max_rows()

    base = (
        alt.Chart(df)
        #cap utilization rates at 40
        .transform_calculate(
            util_capped="datum.utilization_rate > 40 ? 40 :
 ↪  datum.utilization_rate"
        )
        #construct 40 bins
        .transform_bin(
            ["util_bin", "util_bin_end"],
            field="util_capped",
            bin=alt.Bin(extent=[0, 41], step=step),
        )
        #count the number of trips in each bin
```

```
        .transform_aggregate(
            sum_trips="sum(trip_count)",
            groupby=["util_bin", "util_bin_end"],
        )
        #count the number of trips total
        .transform_joinaggregate(total_trips="sum(sum_trips)")
        .transform_window(
            cum_trips="sum(sum_trips)",
            sort=[{"field": "util_bin"}],
        )
        # share_prev = cumulative share before this bin (so the crossing bin
        ↪ is included)
        .transform_calculate(
            share="datum.cum_trips / datum.total_trips",
            share_prev="(datum.cum_trips - datum.sum_trips) /
↪ datum.total_trips"
        )
        .mark_bar()
        .encode(
            x=alt.X(
                "util_bin:Q",
                bin="binned",
                title="Utilization rate",
                axis=alt.Axis(labelExpr="datum.value == 40 ? '40+' :
↪ datum.label"),
            ),
            x2=alt.X2("util_bin_end:Q"),
            y=alt.Y("sum_trips:Q", title="Number of trips"),
            fill=alt.condition(
                "datum.share_prev < 0.4",  # inclusive until >40% is colored
                alt.value("red"),
                alt.value("lightgray"),
            ),
        )
    )

    note_text = (
        "Note: bars colored red until the bottom 40% of trips have been
        ↪ colored.\n"
        "Utilization rate should only be interpreted in relative terms; we do
        ↪ not know "
        "how many days of data contribute to the passenger counts (see
        ↪ footnote 2)."
```

```
    )
    caption = (
        alt.Chart(pd.DataFrame({"note": [note_text]}))
        .mark_text(align="left", baseline="top", fontSize=11, lineBreak="\n")
        .encode(x=alt.value(5), y=alt.value(0), text="note:N")
        .properties(height=50)
    )

    final = alt.vconcat(base, caption).configure_view(strokeOpacity=0)
    return final

utilization_hist = make_hist(utilization_df)
display_altair_png(utilization_hist)
```

```
Weighted 40th percentile utilization: 5.7217
```



Note: bars colored red until the bottom 40% of trips have been colored.
Utilization rate should only be interpreted in relative terms; we do not know how many days of data contribute to the passenger counts (see footnote 2).

The 5 routes with lowest utilization rates are:

```
# lowest utilization rate routes
lowest_routes = (
    utilization_df[['route_id', 'route_long_name', 'utilization_rate',
↪   'trip_count', 'passenger_count']]
    .sort_values('utilization_rate', ascending=True)
```

```
    .head(5)
)
print(lowest_routes.to_string(index=False))
```

```
route_id           route_long_name  utilization_rate  trip_count  passenger_count
     171   U. of Chicago/Hyde Park          0.006757         148                1
     130             Museum Campus          0.010471         191                2
     124                 Navy Pier          0.020599         534               11
     143 Stockton/Michigan Express          0.041667          72                3
     134  Stockton/LaSalle Express          0.045455          88                4
```

Problem 3 – income of riders on each route (equity)

Next, we want to get information on the income distribution of CTA riders and how it varies by route. The passenger data do not have information on income (they are anonymized), but they do have information on where they got on. Assume that passenger income equals median income in the Census tract where they boarded.[3]

Part a.

1. Make the passenger data into a spatial data frame using the provided longitude and latitude columns.
2. Spatially join the passenger observations to census tracts, and use this to assign an income to each passenger.
3. Assign each passenger into one of five equally-sized groups by tract-level income quintile and add this quintile as a column to `passengers_inc_df`.

Details:

- Compute the number of unique longitude and latitude points. You should find that there are 353 points.
- Do the spatial join using each *unique* point instead of every observation. This cuts down the number of points to match to just a few hundred points which therefore speeds up computation.
- 7 points have negative reported income. Be sure to drop the points. Can you figure out why those locations have negative reported income? Google could be helpful here.
- Because the longitudes and latitudes are in degrees instead of meters, we should set them to a *geographic* coordinate reference system (CRS). For this question, use `EPSG:4326`, a geographic CRS.
- Print the CRS of the gdf with the census tracts, and compare it to the CRS that you chose for the passenger data. Then transform the CRS of one to the other before performing the spatial join.

Here is a skeleton of functions to write:

```
def rider_income_calc(passengers_df):
    tracts_path = os.path.join(in_path,
↪   "illinois_tract_income/illinois_tract_income.shp")
    tracts = gpd.read_file(tracts_path)

    return passengers_inc_df
```

[3] Obviously, this is a heroic assumption. It helps make the problem set manageable.

Solution

Many Census/ACS-derived files use large negative placeholders like -666666666 to mean "no estimate available / suppressed (insufficient data or disclosure avoidance)." One of these points (41.98, -87.90) corresponds to O'Hare International Airport. The others likely fall in census tracts covering non-residential areas such as industrial zones, large parks, cemeteries, or institutional group quarters (e.g., prisons, hospitals). These areas have zero or minimal residential population, so there are no households to survey for income data.

```
def rider_income_calc(passengers_df):

    tracts_path = join(data_path,
↪   "illinois_tract_income/illinois_tract_income.shp")
    tracts = gpd.read_file(tracts_path)

    psgrs = passengers_df.copy()
    psgrs = psgrs[pd.notna(psgrs['start_lat']) &
↪   pd.notna(psgrs['start_lon'])]

    # Use income at the location where the trip begins (order == 1)
    subtrip_1 = psgrs.loc[psgrs["order"].eq(1)]

    # Deduplicate by coordinates for efficient spatial join
    coord_cols = ["start_lat", "start_lon"]
    unique_coords = (
        subtrip_1[coord_cols]
        .drop_duplicates()
        .reset_index(drop=True)
    )
    print("Number of unique lat-longs:", unique_coords.shape[0])

    # Build GeoDataFrame for unique points in WGS84, then project to tracts
     ↪  CRS
    unique_points = gpd.GeoDataFrame(
        unique_coords,
        geometry=gpd.points_from_xy(unique_coords["start_lon"],
↪   unique_coords["start_lat"]),
        crs="EPSG:4326",
    )

    print("Tracts CRS:", tracts.crs.to_epsg())

    # Transforming CRS
```

```
    unique_points = unique_points.to_crs(tracts.crs)
    unique_joined = gpd.sjoin(unique_points, tracts, how="left", op="within")

    # Get rid of sjoin index artifact
    if "index_right" in unique_joined.columns:
        unique_joined = unique_joined.drop(columns=["index_right"])

    # Merge tract attributes back to all trips by coordinate match
    psgrs_inc = psgrs.merge(
        unique_joined[coord_cols + ["geometry", "GEOID", "ASQPE001"]],
        on=coord_cols,
        how="left",
    )

    out_gdf = gpd.GeoDataFrame(psgrs_inc, geometry="geometry",
↪   crs=tracts.crs)

    # Make sure income is numeric
    out_gdf["income"] = pd.to_numeric(out_gdf["ASQPE001"], errors="coerce")

    # Report and drop negative income points
    neg_income = out_gdf[out_gdf["income"] < 0]
    if len(neg_income) > 0:
        neg_coords = neg_income[coord_cols].drop_duplicates()
        print(f"Points with negative income: {len(neg_coords)}")
        print(neg_coords.to_string(index=False))

    out_gdf = out_gdf[out_gdf["income"] > 0]

    # Columns to keep in outputs
    out_gdf = out_gdf.loc[:, ~out_gdf.columns.duplicated()]

    return out_gdf

passengers_inc_df = rider_income_calc(passengers_df)

passengers_inc_df["income_quintile"] = (
      pd.qcut(passengers_inc_df["income"],
       q=5,
       labels=False,
       duplicates="drop").add(1)
      .astype("Int64")
```

```
  )

print(passengers_inc_df["income_quintile"].value_counts().sort_index())
```

```
Number of unique lat-longs: 353
Tracts CRS: None
Points with negative income: 7
 start_lat  start_lon
 41.875664 -87.673622
 41.786610 -87.737877
 41.809719 -87.596733
 41.910355 -87.638817
 41.809761 -87.593063
 41.792953 -87.739746
 41.977665 -87.904221
1    255729
2    233706
3    233621
4    247371
5    234017
Name: income_quintile, dtype: Int64
```

Part b.

Collapse `passengers_inc_df` to compute the number of passengers from each income quintile on each route. Then repeat the utilization rate calculation from problem 1. Note that now you will have six utilization rates – one for each income quintile and an overall rate.

Diagnostics:

- Add a test in your code to verify that the number of passengers from each income quintile sums to the total number of passengers with non-missing income.
- The set of passengers with non-missing income is smaller than the set of all passengers which you computed in problem 1 for two reasons. How much smaller and why?
- How many route-quintile cells do not have any passengers (e.g. the 76 bus does not have any low-income riders)? Be sure to fill these in as zero.
- Do you think it makes more sense to store the data as a wide format (one row per route with columns for each quintile) or a long format (one row per route-quintile)? Make a guess now and feel free to revisit as you keep working on the problem set.
- Sanity check your results: which route has the most bottom income quintile passengers? Which route has the most top income quintile passengers?

Here is a skeleton of functions to write:

```
def calculate_passenger_counts_quintile(passengers_inc_df):

    return passenger_count_df

def calculate_utilization_rates_quintile(passengers_inc_df):
     passenger_count_df =
 ↪  calculate_passenger_counts_quintile(passengers_inc_df)
    trips_count_df = calculate_trip_counts()

    return utilization_quintile_df
```

Solution

There are two sources of missing incomes: passengers without longitude/latitude data, and census tracts with missing income. The latter is much rarer, and those tracts with missing income are coded to -666666666. There are many more observations with missing incomes, however.

Sanity check: The red train has the most bottom income quintile passengers. The blue train has the most top income quintile passengers.

This solution uses wide format (one row per route, with columns `passengers_q1` through `passengers_q5`), but long format (one row per route-quintile) is arguably the better choice. Long format avoids repetitive column-looping patterns like `for q in range(1, 6): df[f"col_q{q}"] = ...`, works naturally with `groupby("income_quintile")` for per-quintile aggregations, and integrates more cleanly with Altair for plotting without needing to melt first. The main advantage of wide format is that all quintiles for a single route are visible in one row, which can be convenient for inspection – but you can always pivot from long to wide at the display stage.

```
def calculate_passenger_counts_quintile(passengers_inc_df):

    total_counts = (
        passengers_inc_df
        .groupby("line_string", as_index=False, sort=False)
        .agg(
            median_income=("income", "median"),
            passenger_count=("line_string", "size")
        )
    )
```

```
    # Passenger counts per income quintile
    group_cols = ["line_string", "income_quintile"]

    # Count rows (each row = one passenger)
    by_quintile = (
        passengers_inc_df
        .groupby(group_cols, as_index=False, sort=False)
        .size()
        .rename(columns={"size": "passenger_count"})
    )

    # Ensure all quintiles 1..5 appear as columns (even if qcut dropped some)
    expected_quintiles = range(1, 6)

    # Pivot to wide so each quintile is its own column
    by_quintile_wide = (
        by_quintile
        .pivot(index="line_string", columns="income_quintile",
↪   values="passenger_count")
        .reindex(columns=expected_quintiles, fill_value=0)
        .fillna(0)
    )
    # Rename columns to passengers_q1 .. passengers_q5 and restore index as a
     ↪  column
    by_quintile_wide = by_quintile_wide.astype(int)
    by_quintile_wide.columns = [f"passengers_q{int(c)}" for c in
↪   by_quintile_wide.columns]
    by_quintile_wide = by_quintile_wide.reset_index()

    # Merge overall totals with quintile breakdowns
    passenger_count_df = total_counts.merge(by_quintile_wide,
↪   on="line_string", how="left")
    # Fill any remaining NA in passengers_q*
    passenger_cols = [f"passengers_q{i}" for i in range(1, 6)]
    passenger_count_df[passenger_cols] =
↪   passenger_count_df[passenger_cols].fillna(0).astype(int)

    return passenger_count_df

def calculate_utilization_rates_quintile(passengers_inc_df):

    passenger_count_df =
↪   calculate_passenger_counts_quintile(passengers_inc_df)
```

```
    trips_count_df = calculate_trip_counts()

    utilization_quintile_df = (
        trips_count_df[["route_id", "trip_count", "route_long_name"]]
        .merge(
            passenger_count_df,
            how="inner",
            left_on="route_id",
            right_on="line_string"
        )
    )


    # Overall utilization
    utilization_quintile_df["utilization_rate"] = (
        utilization_quintile_df["passenger_count"] /
 ↪  utilization_quintile_df["trip_count"]
    )

    # utilization rates by quintile
    for q in range(1, 6):
        col = f"passengers_q{q}"
        utilization_quintile_df[f"utilization_q{q}"] = (
            utilization_quintile_df[col] /
 ↪  utilization_quintile_df["trip_count"]
        )

    return utilization_quintile_df

utilization_quintile_df =
 ↪  calculate_utilization_rates_quintile(passengers_inc_df)
```

```
# Diagnostics

passenger_count_df = calculate_passenger_counts_quintile(passengers_inc_df)

# Verify that the number of passengers in each quintile sums to the total
 ↪  number of passengers with non-missing income
quintile_cols = [f"passengers_q{i}" for i in range(1, 6)]
passengers_in_quintiles = passenger_count_df[quintile_cols].sum().sum()
total_with_income = passenger_count_df["passenger_count"].sum()
```

```
print("Sum of passengers in each quintile:", passengers_in_quintiles)
print("Total passengers with non-missing income:", total_with_income)
assert passengers_in_quintiles == total_with_income, \
    f"Mismatch: quintile sum ({passengers_in_quintiles}) != total with income
     ↪  ({total_with_income})"
print("Test passed: quintile sums match total with non-missing income.")

# How much smaller is the set of passengers with non-missing income compared
 ↪  to all passengers?
print(f"\nTotal number of passengers: {len(passengers_df)}")
print(f"Number of passengers with non-missing income: {total_with_income}")

# How many route-quintile cells have zero passengers?
zero_cells = (passenger_count_df[quintile_cols] == 0).sum().sum()
total_cells = len(passenger_count_df) * len(quintile_cols)
print(f"\nRoute-quintile cells with zero passengers: {zero_cells} out of
 ↪  {total_cells}")
```

```
Sum of passengers in each quintile: 1204313
Total passengers with non-missing income: 1204313
Test passed: quintile sums match total with non-missing income.

Total number of passengers: 1236508
Number of passengers with non-missing income: 1204313

Route-quintile cells with zero passengers: 32 out of 630
```

```
# Sanity check: which route has the most Q1 and Q5 passengers?
most_q1 =
 ↪  passenger_count_df.loc[passenger_count_df["passengers_q1"].idxmax(),
 ↪  ["line_string", "passengers_q1"]]
most_q5 =
 ↪  passenger_count_df.loc[passenger_count_df["passengers_q5"].idxmax(),
 ↪  ["line_string", "passengers_q5"]]
print(f"Most bottom quintile passengers: {most_q1['line_string']}
 ↪  ({int(most_q1['passengers_q1'])})")
print(f"Most top quintile passengers: {most_q5['line_string']}
 ↪  ({int(most_q5['passengers_q5'])})")
```

```
Most bottom quintile passengers: Red (39290)
```

```
Most top quintile passengers: Blue (18719)
```

Problem 4

Take the two variables you have lovingly constructed in the last two problems and help Nora understand how they are related for each route. For this problem you will start from a data frame with utilization (efficiency, problem 2) and median rider income (equity) for each route.

1. Compute a route's income as the median of the income of all the riders who use that route (as estimated by tract of origin). This is slightly different from and simpler than what you did in problem 3[^4]
2. Make a scatter plot with route utilization on the y-axis and the median rider income on the x-axis. Find three routes which you think your readers might be particularly interested in knowing the income and utilization levels for before deciding whether to cut service. Annotate them using red and label them with the `route_long_name` column we got from `routes.txt`.
3. Draw a horizontal line at the 40th percentile of trip-weighted utilization (which you calculated already in problem 2). This shows which routes would be cut if you prioritized only efficiency and ignored equity.
4. The plot is hard to read because the low utilization routes are all tightly compressed. Make a second version of the plot where the y-axis is on a reverse log scale (we didn't cover this in class, but it is easy to find via the online altair documents). Compare the two versions of the plot. For which audiences would you want to use the regular y-axis and for which audiences would you want to use the reverse-log y-axis?

Here is a skeleton of functions to write:

```python
def plot_utilization_vs_income(
    df: pd.DataFrame,
    log_scale: bool,
    income_col: str = "median_income",
    util_col: str = "utilization_rate",
    add_cutoff: bool = True,
    filename: str = "utilization_plot.html",
    highlight_ids=("1", "2", "3"),
):
```

Solution

To find routes worth highlighting, we construct a combined ranking of all routes. Each route is ranked separately by median rider income (highest first) and by utilization rate (lowest first). We then combine these into a single score with 60% weight on the income rank and 40%

on the utilization rank. Routes that score highest are those with high-income riders and low utilization. These are policy-relevant because cutting them would protect lower-income riders while removing underused service.

```
# Compute median income per route from passenger-level data
route_income = (
    passengers_inc_df
    .groupby("line_string", as_index=False)
    .agg(median_income=("income", "median"))
)

# Merge with utilization rates
route_summary = utilization_df.merge(route_income, on="line_string",
 ↪  how="inner")

# Combined ranking: 60% weight on income (high = interesting), 40% on
 ↪  utilization (low = interesting)
route_summary["income_rank"] =
 ↪  route_summary["median_income"].rank(ascending=False)
route_summary["utilization_rank"] = route_summary["utilization_rate"].rank()
route_summary["combined_rank"] = 0.6 * route_summary["income_rank"] + 0.4 *
 ↪  route_summary["utilization_rank"]

route_sorted = route_summary.sort_values("combined_rank")

print("=== Top 5 routes by combined rank (60% income, 40% utilization) ===")
print(
    route_sorted[["route_long_name", "median_income", "utilization_rate",
 ↪  "combined_rank"]]
    .head(5)
    .to_string(index=False)
)
```

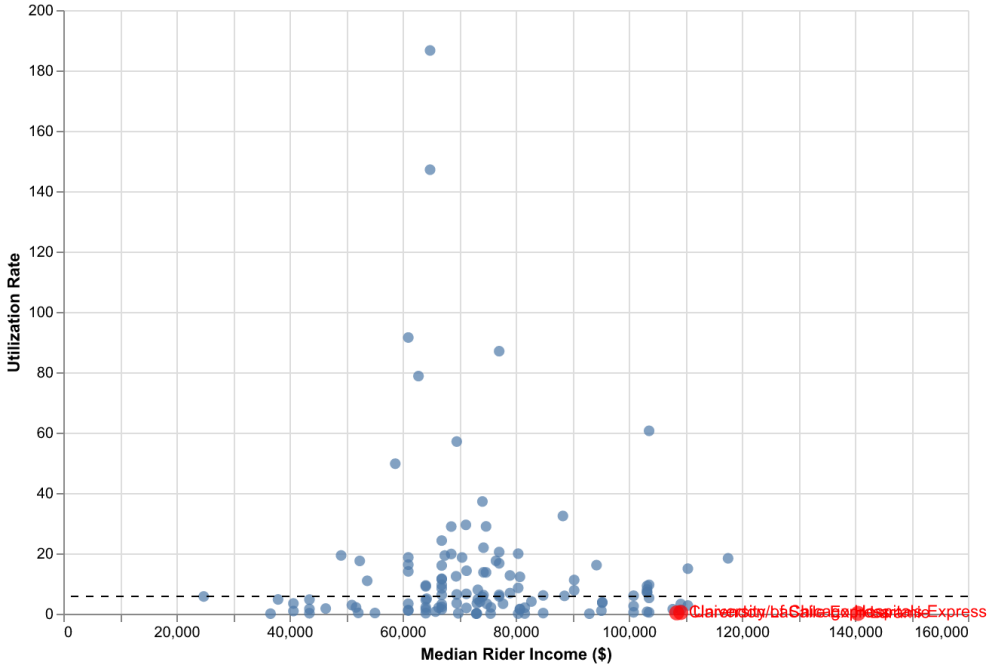
```
=== Top 5 routes by combined rank (60% income, 40% utilization) ===
                      route_long_name  median_income  utilization_rate  combined_rank
                              Laramie       140556.0          0.113462            4.2
University of Chicago Hospitals Express       109211.0          0.421053           10.4
             Clarendon/LaSalle Express       108500.5          0.280000           11.2
                              Harrison       109211.0          1.242647           14.0
                              Lincoln       103611.0          0.438119           14.1
```

I would choose Laramie, University of Chicago Hospitals Express and Clarendon/LaSalle

Express.

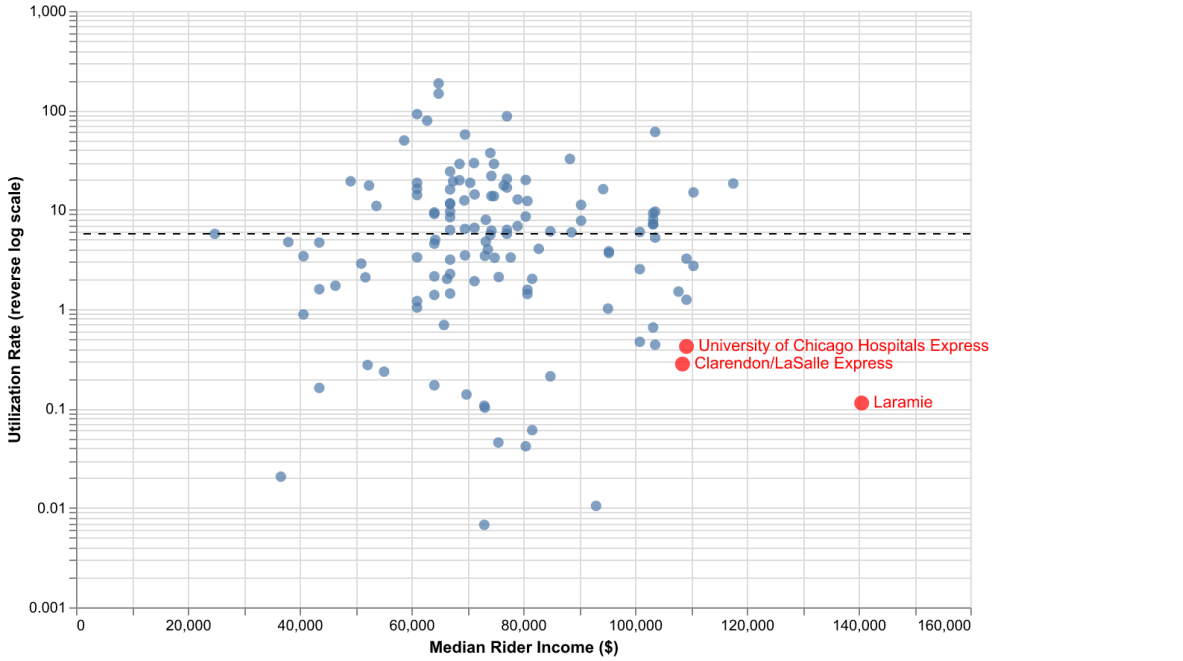
```
highlight_names = [
    "Laramie",
    "University of Chicago Hospitals Express",
    "Clarendon/LaSalle Express",
]

def plot_utilization_vs_income(
    df: pd.DataFrame,
    log_scale: bool,
    income_col: str = "median_income",
    util_col: str = "utilization_rate",
    add_cutoff: bool = True,
    highlight_names: list = highlight_names,
):
    df = df.copy()
    df["highlight"] = df["route_long_name"].isin(highlight_names)

    # Base scatter: all routes in grey
    y_title = "Utilization Rate (reverse log scale)" if log_scale else
↪   "Utilization Rate"
    base = alt.Chart(df[~df["highlight"]]).mark_circle(size=50).encode(
        x=alt.X(f"{income_col}:Q", title="Median Rider Income ($)"),
        y=alt.Y(f"{util_col}:Q", title=y_title,
                scale=alt.Scale(type="log") if log_scale else alt.Scale()),
        tooltip=["route_long_name", income_col, util_col],
    ).properties(
        width=600,
        height=400,
    )

    # Highlighted routes in red
    highlight_pts = alt.Chart(df[df["highlight"]]).mark_circle(
        size=100, color="red"
    ).encode(
        x=alt.X(f"{income_col}:Q"),
        y=alt.Y(f"{util_col}:Q",
                scale=alt.Scale(type="log") if log_scale else alt.Scale()),
        tooltip=["route_long_name", income_col, util_col],
    )
```

```
    # Labels for highlighted routes
    labels = alt.Chart(df[df["highlight"]]).mark_text(
        align="left", dx=8, color="red", fontSize=11
    ).encode(
        x=alt.X(f"{income_col}:Q"),
        y=alt.Y(f"{util_col}:Q",
                scale=alt.Scale(type="log") if log_scale else alt.Scale()),
        text="route_long_name:N",
    )

    chart = base + highlight_pts + labels

    # Add 40th percentile cutoff line
    if add_cutoff:
        cutoff_line = alt.Chart(
            pd.DataFrame({"y": [p40_utilization]})
        ).mark_rule(strokeDash=[5, 5], color="black").encode(
            y="y:Q"
        )
        chart = chart + cutoff_line

    util_note = (
        alt.Chart(pd.DataFrame({
            "note": ["Utilization rate should only be interpreted in relative
             ↪  terms; we do not know how many days of data contribute to the
             ↪  passenger counts (see footnote 2)."]
        }))
        .mark_text(align="left", baseline="top", fontSize=11)
        .encode(x=alt.value(5), y=alt.value(0), text="note:N")
        .properties(height=28)
    )

    return alt.vconcat(chart, util_note).configure_view(strokeOpacity=0)

# Regular y-axis
display_altair_png(plot_utilization_vs_income(
    route_summary, log_scale=False
))
```



Utilization rate should only be interpreted in relative terms; we do not know how many days of data contribute to the passenger counts (see footnote 2).

```
# Reverse log y-axis
display_altair_png(plot_utilization_vs_income(
    route_summary, log_scale=True
))
```



Utilization rate should only be interpreted in relative terms; we do not know how many days of data contribute to the passenger counts (see footnote 2).

The two plots show the same data but with different y-axis scales:

1. Regular y-axis: Most routes cluster near zero utilization, making it hard to distinguish between low-utilization routes. However, the high-utilization outliers (mainly train lines) are clearly visible.
2. Reverse log scale y-axis: Spreads out the low-utilization routes, making differences between them visible. Routes below the 40th percentile cutoff are easier to identify and compare.

Audience recommendations:

- Use the regular y-axis for general audiences or policymakers who need to understand the magnitude of utilization differences between different routes.
- Use the reverse log scale for technical analysts or planners who need to make decisions about which low-utilization routes to cut. It makes it easier to compare routes near the cutoff threshold.