Tram Aux-Energy Project

What this project does

I simulate tram trips and look at the 'auxiliary' energy — lights, ventilation fans, and heating/cooling (HVAC). For every trip I compare two ways of running the tram:

• Fixed system: simple, constant settings (brighter at night, steady ventilation, HVAC that doesn't care about how many people are inside).

• Sensor-based system: smarter and flexible — it dims lights when the sun is up, turns fans down when the tram is empty, and accounts for how far the outdoor temperature is from comfy inside.

For each trip I compute energy (kWh) for both systems and then calculate "% savings" if we used the sensor-based setup. Later, I train a small ML model so I can estimate that % quickly for any new trip.

How the synthetic dataset is built

Every row in the dataset is one tram trip. I randomly pick realistic conditions and then compute energies:

• Season (winter/spring/summer/autumn) with weights, because energy isn't spread evenly across the year.

• Hour of day → I turn that into a simple daylight flag (1 for daylight, 0 for dark) using season-specific daylight windows.

• Trip duration in minutes (clipped between 10 and 60) and converted to hours for energy.

• Outside temperature (°C) drawn from a season-typical range (e.g., winter cooler, summer warmer).

• Passenger count (0–200) with peaks in morning and evening rush hours; from that I also get occupancy %.

The one identity I use everywhere is:

Energy (kWh) = Power (kW) × Time (hours)

So most of the formulas are just: work out a sensible 'kW' and then multiply by trip duration in hours.

Lighting

Fixed system (no sensors):

lighting_kWh_fixed = (L_BASE_DAY if daylight = 1 else L_BASE_NIGHT) × duration_hours

• Daytime uses a lower baseline (sun helps). Night uses a higher baseline.

Sensor-based lighting:

lighting_kW_sensor = max(L_MIN + α_occ × passengers − β_day × daylight , L_MIN)
lighting_kWh_sensor = lighting_kW_sensor × duration_hours
Why this makes sense:
• There's a safety minimum L_MIN (we never go below this).
• More people → slightly more light (+ α per passenger) so people can move safely/read.
• If it's daylight, we dim by β (daylight harvesting).

Ventilation

Fixed system:

vent_kWh_fixed = VENT_FIXED × duration_hours
• A steady fan baseline, regardless of occupancy.

Sensor-based ventilation:

vent_kW_sensor = max(VENT_MIN_LOW + γ_occ × passengers , VENT_MIN_LOW)
vent_kWh_sensor = vent_kW_sensor × duration_hours
Why it's reasonable:
• More people → more fresh air needed (indoor air quality scales with headcount).
• When empty, keep a lower minimum to save energy, but never below the safety minimum.

HVAC (heating/cooling)

First define the temperature gap from comfy inside: ΔT = | COMFORT_TEMP (21°C) − outside_temp |

Fixed system:

hvac_kW_fixed = A_fixed × ΔT + B_fixed
if outside is "extreme" (below EXTREME_LOW or above EXTREME_HIGH):
  hvac_kW_fixed *= EXTREME_MULT_FIXED
hvac_kWh_fixed = hvac_kW_fixed × duration_hours
Intuition:
• The farther outside is from comfort, the harder HVAC has to work (proportional to ΔT).
• There is always some baseline B_fixed for fans/pumps.
• On very cold/hot days, there's a small extra penalty (defrost cycles, inefficiencies).

Sensor-based HVAC:

hvac_kW_sensor = A_sens × ΔT + B_sens + η_occ × passengers
if outside is "extreme":
  hvac_kW_sensor *= EXTREME_MULT_SENS (this multiplier is smaller than the fixed one)
hvac_kWh_sensor = hvac_kW_sensor × duration_hours
Intuition:
• Slightly lower slope and baseline (A_sens < A_fixed, B_sens < B_fixed) to reflect smarter control.

• People add a bit of load (heat and moisture) so we add η per passenger.
• Extreme-day bump still applies but is smaller because the system can throttle better.

What "% savings" means

On each trip I compare the two totals (fixed vs sensor):
total_fixed = light_fixed + vent_fixed + hvac_fixed
total_sensor = light_sensor + vent_sensor + hvac_sensor

pct_savings = 100 × (1 − total_sensor / total_fixed)
• If sensor uses less, the fraction is < 1 → positive savings.
• If equal → 0%.
• If sensor used more (rare in this setup) → negative.

Example

Imagine: winter, hour=7, duration=30 min (0.5 h), outside=2°C, passengers=140. I mark daylight=1 or 0 based on winter daylight window. Then I compute each subsystem's kW and multiply by 0.5 h. Finally I add up totals for fixed and for sensor and plug them into the savings formula. This exact calculation is what the CLI calls the "Formula (true)" number.

Training the ML model (Random Forest)

I treat % savings as a number to predict (regression). Inputs are season, daylight, duration, outside temp, passenger count, and occupancy. Since 'season' is text, I first one-hot encode it (turn it into several 0/1 columns). I use a scikit-learn Pipeline so preprocessing and the model stay together.

Why Random Forest?
• It handles non-linear patterns like "very cold + very crowded".
• It doesn't need feature scaling and is quite robust on tabular data.
• It gives feature importances, so we can see which inputs the model relied on.
Settings I used: 300 trees, each split looks at a random subset of features (sqrt rule), and leaves must have a handful of trips so the model doesn't overfit tiny quirks.

I split data 80/20 into train and test. After fitting, I report MAE (average miss), RMSE (like MAE but punishes big misses more), and R^2 (how much variance in % savings the model explains). I also save the whole pipeline to a file so I can load it later without retraining.

The CLI predictor

When you run predict_cli.py, it asks for five things: season, hour, duration, outside temperature, and passengers. It turns season+hour into a daylight flag and computes occupancy %. Then it does two computations:
1) ML model — it loads the saved Random Forest pipeline and predicts % savings for that single-row DataFrame.

2) Formula (true) — it runs the exact physics formulas (same ones used to generate the dataset).
Because the model was trained on synthetic data from those formulas, the two numbers should be close. With real data in the future, the ML number could learn patterns that go beyond the simple rules.

How to run

1) Create a virtual environment and install requirements.
2) Run generate_dataset.py to create synthetic_trips.csv.
3) Run train_model.py to fit the forest and save the model file.
4) Run predict_cli.py to try a scenario and compare ML vs formula.

Limitations

In this prototype, the comfort setpoint is fixed at 21 °C, but optimal cabin temperature is context-dependent (adaptive comfort): it should vary with outside weather, solar gain, humidity, crowding, and even clothing, so a smarter controller would adjust the setpoint by season and conditions. The daylight windows are hand-picked and may not match actual sunrise/sunset for a given date, latitude, and DST; a proper solar-position calculation would be more accurate. The physics coefficients (kW per °C, baselines, per-passenger terms, extreme multipliers) are stylized rather than fleet-calibrated, and the ventilation/lighting rules simplify IAQ standards and dimming behavior. Passenger peaks, duration bounds, and seasonal temperature distributions are also rough approximations, so absolute kWh and %-savings numbers are indicative rather than exact; the trends are more trustworthy than the precise values. These are synthetic assumptions meant to be realistic for learning: season weights, daylight windows, and temperature ranges are simple but plausible. The point is to show the full workflow end-to-end.

Next steps

By plugging in real telemetry, weather feeds, and route patterns, we can retrain and compare different models (e.g., gradient boosting) while keeping the same clean interface and explainability.