TTC Transit Service Reliability Analysis

Predicting delay-prone times/stations on Toronto's subway network using real City of Toronto Open Data, with a full ETL pipeline, comparative ML modeling, and an interactive Power BI dashboard.

23,701 real TTC subway delay events (2024–2026) | SQL star schema | Python ETL | Random Forest vs Gradient Boosting | 3-page Power BI dashboard

Why this project

I built this to demonstrate an end-to-end data science workflow on a real, messy, public dataset — not a pre-cleaned Kaggle CSV. It touches SQL schema design, Python ETL, data-quality problem solving, comparative ML modeling, and BI dashboard delivery, and doubles as a candidate contribution to Data for Good Toronto.

Data sources

Source	What	Where
TTC Subway Delay Data	Real delay events, 2024–2026	Toronto Open Data
TTC Subway Delay Codes	Code → description lookup	Same dataset page
Environment Canada	Daily weather (Toronto City station)	climate.weather.gc.ca

Repo structure

```
ttc-transit-reliability/
├── ttc_pipeline.py          <- ETL + feature engineering + model, one file
├── sql/schema.sql           <- star schema documentation
├── notebooks/01_eda.ipynb   <- exploratory analysis, charts, model comparison
├── dashboard/
│   ├── powerbi_data/        <- CSV exports Power BI reads from
│   └── screenshots/         <- dashboard page screenshots
├── data/
│   ├── raw/                 <- drop downloaded CSV/XLSX files here (gitignored)
│   └── processed/           <- SQLite DB + trained model (gitignored)
└── requirements.txt
```

Methodology — step by step

1. ETL & a real data-quality problem

TTC's raw delay data truncates the station name field at **22 characters** at the source (a genuine quirk in the open dataset — confirmed by testing with `pandas.set_option('display.max_colwidth', None)`, which showed the same cut-off text). Combined with between-station segments (`"SPADINA TO WILSON STAT"`), platform modifiers (`"(IN TUNNEL)"`), and inconsistent prefixes (`"APPROACHING KIPLING"`), the raw station field had **535 distinct messy values** for what should be ~70 real stations.

I built `normalize_station_name()` — a matcher against the real list of TTC subway stations using earliest-position, longest-match search (not naive prefix matching, since the station name doesn't always appear first in the string). Tested against the actual 535 raw values:

- **First pass: 92.3%** matched
- **After adding typo corrections, renamed-station aliases (e.g. Cedarvale → Eglinton West), and interchange-station special cases** (Bloor-Yonge, Sheppard-Yonge): **97.2%** matched
- **On the full real dataset: 99.6%** matched (71 clean stations, 0.4% genuinely unmatched — infrastructure/maintenance locations like “Gunn Building” and line-level references like “Line 1”, correctly left unmatched rather than force-mapped)

2. Star schema + SQLite

`fact_delay_events` at the center, with `dim_station`, `dim_date`, `dim_delay_code`, and `fact_weather_daily` (joined by date) as supporting tables — see `sql/schema.sql`.

3. Feature engineering

- Standard categorical features: hour, day of week, season, station, line, bound, mode
- **Rush-hour flag** (7-9am, 4-6pm), motivated by the EDA finding that these hours carry the highest delay *volume*
- **Weather features** (mean temp, precipitation) joined from Environment Canada daily data
- **Rolling 7-day / 30-day per-station delay frequency** — built with an explicit leakage safeguard: the rolling window is shifted forward one day before summing, so a given event’s own day never contributes to its own feature value. Verified directly: the first date in the dataset correctly shows 0 for both windows (no prior history exists to leak from).

4. Modeling

Framed as binary classification: will a given (station, hour, day, season, ...) combination see a delay >= 5 minutes?

Random Forest vs Gradient Boosting, trained on identical features for a fair comparison — Gradient Boosting selected as the primary model after consistently outperforming Random Forest by a small but real margin across every feature-set iteration (~0.62-0.65 ROC-AUC range for both).

5. The real finding: a feature ceiling, not a model problem

Across five separate experiments — baseline features, model swap, adding weather, adding a rush-hour flag, adding rolling per-station frequency — **ROC-AUC stayed consistently in the 0.62–0.65 range**, and `station_name` and `hour` together accounted for the large majority of feature importance in every run. Adding weather didn’t add *new* signal so much as **replace** the coarser `season` feature with a more precise version of the same information (season’s importance dropped from ~0.09 to ~0.01 once real temperature was available).

Conclusion: this class of feature (schedule/location/weather) has hit a real, honest ceiling. Meaningfully improving on ~0.64 AUC would likely require a different kind of data entirely — live vehicle positions or ridership counts — not more derived features from the same delay log.

6. EDA findings (see `notebooks/01_eda.ipynb`)

- **Hour of day:** volume peaks at rush hour (8am, 4-5pm), but the *longest* individual delays happen overnight (4-5am) — frequent-but-short vs. rare-but-severe.
- **Day of week:** Sunday has the fewest delay events but the **highest** average delay length (~9.9 min vs ~7.5-8 min weekdays) — reduced Sunday service likely means less redundancy when something does go wrong.
- **Root causes:** the top 3 delay codes (Disorderly Patron, Passenger Assistance Alarm, Door Monitoring) are all **passenger/human-behavior related, not mechanical failure**.
- **Season:** winter has both the most delay events and the longest average delays of any season.
- **Station-level nuance:** Kipling, Bloor-Yonge, and Kennedy lead in raw delay *count*, but that’s confounded with being major interchange stations with heavier traffic. Victoria Park stands out differently — lower volume but the highest average delay *length* of the top 15, arguably a more meaningful “problem station” signal.
- **Rolling trend:** Kipling Station shows a genuine repeating seasonal pattern in its 30-day rolling delay count, peaking every January-February across both years in the dataset.

Power BI dashboard

Three pages, built on the same star-schema CSVs exported by `ttc_pipeline.py --powerbi-export`:

1. **Overview** — KPI cards (total events, avg delay, delay-prone rate), delay volume by hour and day of week, season slicer
2. **Root Cause** — top delay causes by frequency and by average duration, with a station slicer for per-station drill-down
3. **Model Insights** — feature importance chart, Random Forest vs Gradient Boosting comparison, interactive rolling-delay trend line by station

See `dashboard/screenshots/` for page images, or open `Dashboard_TTC_Data.pbix` directly in Power BI Desktop.

How to run

```
pip install -r requirements.txt

# 1. Download TTC Subway Delay Data + delay codes lookup from Toronto
#    Open Data, and (optionally) Environment Canada weather CSVs.
#    Drop them all into data/raw/

# 2. Run the full pipeline (ETL + model training)
python ttc_pipeline.py

# Or run steps individually:
python ttc_pipeline.py --etl-only
python ttc_pipeline.py --model-only
python ttc_pipeline.py --powerbi-export   # export CSVs for Power BI
```

If no files are present in `data/raw/`, the pipeline auto-generates a small synthetic sample so it can be smoke-tested end-to-end before real data arrives.

Limitations & next steps

- Station-level delay counts are confounded with train frequency/traffic at that station — a true “problem station” ranking would need to normalize by scheduled service volume, which isn’t in this dataset.
- The ~0.4% of unmatched station values are infrastructure/maintenance locations, correctly excluded rather than force-mapped — but this means a small number of real delay events aren’t attributed to any station.
- Next step under consideration: extending the pipeline to bus and streetcar delay data (same TTC Open Data structure), which would also make the currently-unused `mode` feature meaningful.

Tech stack

Python (pandas, scikit-learn, sqlite3) - SQL (star schema) - Power BI (DAX, relationships, interactive dashboards) - Jupyter