# Event Based Data Pipeline for Earnings Announcements

## Group Details

- **Ryaan Zaheed** (NetID: **rz346**)

- **Harsh Choksi** (NetID: **hnc28**)

GitHub repository: https://github.com/itzGlitxh/EarningsProject/

## Project Definition

The goal of this project is to build a full event based pipeline around quarterly earnings for large U.S. stocks.

The pipeline should be able to

- Download daily prices and earnings data for a universe of tickers
- Store that data in a clean relational database
- Build event windows around each earnings announcement
- Engineer features that describe price behavior and earnings surprise
- Train machine learning models that try to predict the sign of the short term price reaction after earnings

The final outcome is not just a model. The deliverable is a reproducible system that can be re-run on fresh data with one set of scripts.

## Introduction

Earnings announcements are natural "events" in equity markets. Prices and trading activity usually change around these dates, but the underlying data is scattered across APIs, spreadsheets and scripts. It is easy to misalign dates or mix up which prices belong to which earnings event.

This project designs a small but realistic research platform around this problem. The main ideas are

1. Build a single pipeline from raw data to database to modeling dataset
2. Treat each earnings release as an event with a seven day window from day minus three to day plus three
3. Train simple models that use only information available before the announcement to predict whether the price move from day zero to day one will be positive or negative
4. Evaluate the models with a strict time based split so that training data is always in the past and test data is always in the future

The project focuses on a group of large cap U.S. stocks from 2020 to 2025. This includes big technology names like AAPL and MSFT, large banks like JPM and BAC, plus several other liquid names.

The results show that the best model achieves 53.4 percent accuracy on future earnings compared with a 52.4 percent majority class baseline. This is only a one point improvement, which is not enough to claim a real edge after costs, but it does show that the pipeline is working and that the evaluation is honest.

## Methodology

We split the work into three main components: data pipeline, database and machine learning.

### 1. Data pipeline

All folder paths, tickers and date ranges are stored in `config.py`. The key settings are

- Price history from 2020/01/01 to 2025/12/31
- Event window length of three trading days on each side of the announcement
- Reaction horizon from day zero to day one

**Price data**
`download_prices.py` uses `yfinance` to download daily OHLCV data for every ticker over the full date range. Each ticker is saved as `data/raw/prices/TICKER.csv`.

**Earnings data**
`download_earnings.py` uses `yfinance.Ticker(ticker).earnings_dates` to pull historical earnings dates with

- Announcement date

- Actual EPS
- Estimated EPS
- Surprise percent

The script then

- Standardizes column names
- Computes surprise level as `eps_actual` - `eps_estimate`
- Filters to the same date range as prices
- Drops rows where EPS values are missing
- Writes `data/raw/earnings/TICKER.csv`

At this stage the project has consistent raw CSVs for prices and earnings that can be used on any machine without a paid API key.

**2. Database**

`load_to_db.py` creates a small SQLite database to keep the data organized. There are three main tables

- `companies` with one row per ticker
- `daily_prices` with one row per ticker per trading day
- `earnings_announcements` with one row per ticker per earnings date

`companies` holds `company_id`, ticker and optional name and sector. `daily_prices` and `earnings_announcements` both store `company_id` as a foreign key.

The script

1. Creates the tables if they do not exist
2. Inserts companies using the ticker list
3. Loads each price CSV into `daily_prices`
4. Loads each earnings CSV into `earnings_announcements`

This keeps the raw data in one place and avoids repeated CSV parsing later.

**3. Event windows and features**

`build_event_windows.py` reads from the database and produces a processed event window file.

On the price side the script

- Joins `daily_prices` with `companies` to get tickers
- Computes daily log returns from adjusted close
- Computes rolling volatility over several windows
- Measures average volume and relative volume

  Calculates simple technical indicators like moving averages, RSI and intraday range

On the earnings side it

- Joins `earnings_announcements` with `companies`
- For each ticker tracks previous surprise level and surprise percent
- Counts streaks of consecutive beats
- Computes the historical beat rate up to each event

For each earnings announcement the script

- Aligns the announcement date to the first trading date on or after that calendar date
- Takes all price rows from three trading days before that date to three days after it
- Attaches the earnings features to every row for that event
- Computes cumulative log return within the window and converts it to a cumulative simple return

All of this is saved in `data/processed/event_windows.csv`. Each row is identified by an `earnings_id` and a relative day index between minus three and plus three.

## 4. Machine learning

`modeling.py` converts the event windows into a modeling dataset and trains classifiers.

### Label definition and feature selection

The model predicts the sign of the price move from the event day to the next trading day. For each event the script

- Sums log returns for `window_rel_day` equal to 0 and 1
- Converts this to a simple `reaction_return`
- Sets `reaction_positive` to 1 if `reaction_return` is greater than zero, otherwise 0

Features are taken mainly from the day before earnings, plus some pre earnings changes. They include

- Earnings surprise level and surprise percent
- Previous surprise percent, beat streak and historical beat rate
- Changes in volume and volatility before the event
- RSI and price relative to moving averages
- Interaction terms that mix surprise with momentum or volatility

The script saves this table as `modeling_dataset.csv`.

**Time based train and test split**

To avoid look ahead bias, the dataset is sorted by `announcement_date` and split by time. The first 80 percent of events are used for training and the last 20 percent for testing. The logic is similar to:

```
df = df.sort_values("announcement_date").reset_index(drop=True)

split_idx = int(len(df) * 0.8)

train = df.iloc[:split_idx]

test = df.iloc[split_idx:]

X_train = train[feature_cols]

y_train = train["reaction_positive"]

X_test = test[feature_cols]

y_test = test["reaction_positive"]
```

The script also computes a baseline accuracy from the majority class in the test set.

Three models are trained

- Logistic regression with standardized features
- Random forest classifier with shallow trees and class weight balancing
- Gradient boosting classifier with tuned learning rate and depth

For each model the script prints accuracy and a confusion matrix, then compares all models to the baseline.

**Overfitting check**

`check_overfitting.py` reuses the same features but creates two different splits

- A proper time based split, similar to the main script
- A random split that ignores time order

It prints accuracies for both and highlights the gap. The random split generally looks better, which illustrates how easy it is to overstate performance if time order is ignored.

## Results

### 1. Data coverage

After running the pipeline the SQLite database contains

- Daily prices from early 2020 through late 2025 for every ticker
- Several hundred earnings announcements for these firms

The modeling dataset then collapses each event window into a single row. The final time based split uses

- **Training period**: 2020/01/14 to 2024/10/15
- **Testing period**: 2024/10/15 to 2025/12/03

The test set contains **191** earnings events from the later part of the sample. Training uses several hundred earlier events.

### 2. Classification accuracy

The key numerical result from `modeling.py` is:

| **Model** | **Accuracy** | **vs Baseline** |
|---|---|---|
| Baseline (majority) | 52.4% | N/A |
| Logistic regression | 53.4% | +1.0% |

| | | |
|---|---|---|
| Random forest | 49.2% | -3.1% |
| Gradient boosting | 50.3% | -2.1% |

(Here the baseline is the accuracy of always predicting the more common class in the test set.)

The most important points are:

- The best model is logistic regression at 53.4 percent
- This is only 1.0 percentage point above the 52.4 percent baseline
- Both random forest and gradient boosting perform **slightly worse** than the baseline

So the data and feature set contain very limited predictive power for short term earnings reactions in this universe.

**3. Honest assessment of predictive value**

The script also prints an “Honest Assessment” section. In plain language:

- The model was tested on 191 real future earnings that the model did not see during training
- Training used only data from 2020 through 2023, then stopped in October 2024
- Even with this clean setup, the best model only beats baseline by 1 percentage point

This is not enough to claim real predictive value. Reasons:

1. The test set is small, which means random noise can easily move accuracy by a few percentage points
2. Market conditions change over time, so a signal that works in early years can fade in later periods
3. Any trading strategy would face transaction costs and slippage that would likely remove such a small edge
4. The labels are coarse (up vs down) and do not account for the size of moves, which throws away information

The conclusion of the project is that for large cap U.S. stocks it is very difficult to beat a simple baseline, at least with this kind of straightforward fundamental and technical feature set.

**4. Overfitting and evaluation choice**

`check_overfitting.py` shows that when the same data is split randomly, the measured accuracy jumps several points above the time based accuracy. In other words, the models appear to perform much better when they are allowed to mix past and future events in both the train and test set.

This happens because

- Future information leaks into the training process through the random split
- The model is evaluated on data that is too similar to what it has already seen

The time based split is stricter and more realistic. It deliberately uses only past earnings to predict later ones. The fact that accuracy drops when using this split is actually a good sign in terms of honesty, even though it makes the result less exciting.

**5. What we learned**

Even though the final numbers are not impressive, the project still gives useful insights:

- A clean event based pipeline around earnings can be built from free data sources and a small set of Python scripts
- Reproducible storage in SQLite makes it easy to rebuild features or try new models without touching the raw downloads
- The choice of evaluation method matters as much as the choice of model
- Large, liquid stocks seem close to informationally efficient at this horizon when using public earnings data and simple technical signals

In practice this suggests that a more promising research path would need either richer features (for example, options data, analyst revisions or intraday prices) or a different prediction target such as volatility or direction over longer windows.

## Contributions

### Harsh Choksi (Hnc28)

- Set up the overall repository structure and `config.py`
- Implemented and refactored the scripts that download prices and earnings from `yfinance`
- Wrote most of `build_event_windows.py` including the construction of event windows and many of the price based features
- Helped design the feature set and the label definition for the modeling dataset

- Ran the full pipeline several times, checked outputs and contributed to writing this report

**Ryaan Zaheed (rz346)**

- Designed the SQLite schema for `companies`, `daily_prices` and `earnings_announcements`

- Implemented `load_to_db.py` and verified that all joins and foreign keys behaved as expected
- Took the lead on `modeling.py`, including the time based split, baseline calculation and training of logistic regression, random forest and gradient boosting models
- Wrote `check_overfitting.py` and summarized the impact of random vs time based splits
- Helped interpret the results and drafted the discussion in the Results section

**Joint work**

- Wrote the original project proposal and interim report and kept the scope updated as the code evolved
- Chose the ticker universe and adjusted the date range so that the sample contained enough earnings events
- Debugged scripts together, checked that outputs were sensible and documented how to run each part of the pipeline

## References

1. GitHub repository for this project
   *Event Based Data Pipeline for Earnings Announcements*
   https://github.com/itzGlitxh/EarningsProject/
2. `yfinance` Python library - Used to download daily OHLCV data and earnings dates for all selected tickers.
3. SQLite and SQLAlchemy documentation - Used as references for database design, table creation and query patterns.
4. scikit learn documentation - Used for logistic regression, random forest, gradient boosting and evaluation metrics.