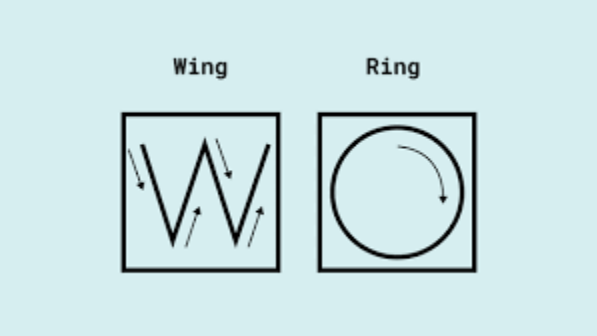
Oct 18, 2025 | Github repo link - https://github.com/ColoradoEmbeddedAI-Fall2025/lab-3-building-a-dataset-team6.git
Embedded AI
Alha, Matt Hartnett, Piyush Nagpal, Sam Walker, Shane, Zane McMorris

Getting Started



For this lab, we are using a smartphone app to gather data from the onboard accelerometer to detect gestures made with the phone. Our goal is to make two gestures recognizable by our model, and to do all the data manipulation, training, and testing with our new dataset. The two gestures we want to train are a square and a triangle.

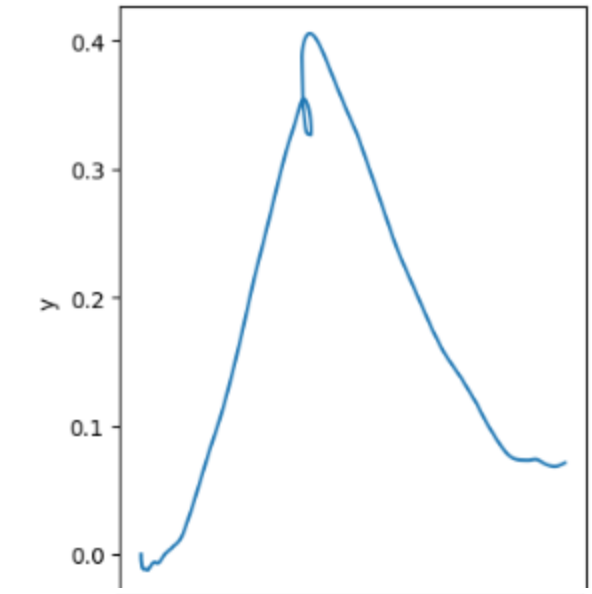
The training data, at the time of writing, consists of 10 samples each of the square being drawn and the triangle being drawn. To gather the data we settled on a common methodology to reduce the number of variables for the input information. We decided to point the phone screen straight up towards the sky, and draw the shapes clockwise at arm's length, with stroke lengths of about one foot. The data was gathered at 100Hz (10ms period), with most gestures taking four to five seconds to perform, resulting in about 400-500 datapoints for each gesture. The only exported data from the app was the calibrated accelerometer data. The reason we don't think we need the orientation data was that we gathered the data from the same orientation. A more refined version of this model could include the orientation data to enhance its capabilities and use cases.

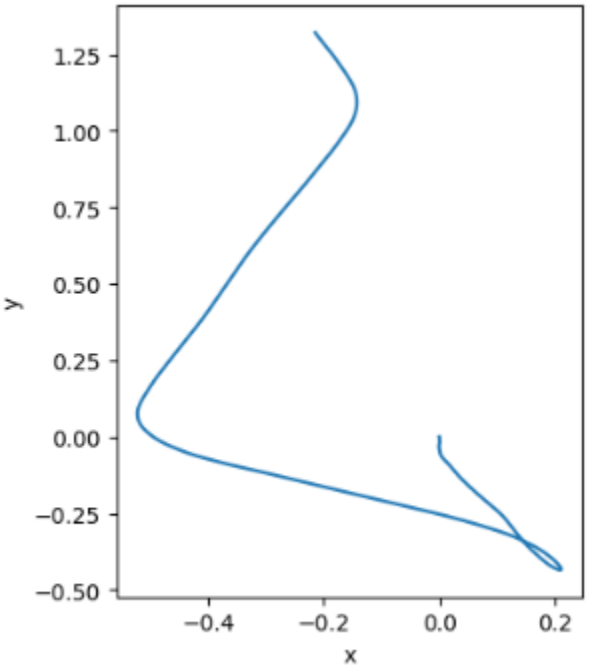
The data exported from the app is as a CSV with the following columns: UTC time, seconds since starting the measurement, and each of the three axes of movement represented as floats. The data could be cleaned up a bit by removing the unnecessary data (time data) because we just care about the motions being made by the phone, not when they happened.

time	seconds_elap	z	y	x
1.7605E+18	0.12857935	0	0	0
1.7605E+18	0.14854443	0.04586124	-0.107564	-0.0698555
1.7605E+18	0.15852686	0.02740765	-0.10021	-0.0276057
1.7605E+18	0.16850928	-0.0128803	-0.0153232	-0.0028821
1.7605E+18	0.17849194	0.0766592	0.04658866	-0.0041811
1.7605E+18	0.18847437	0.09456539	0.07390976	0.02827662
1.7605E+18	0.19845679	-0.0625772	0.0738523	0.05026668
1.7605E+18	0.20843945	0.03647995	0.05747128	-0.0114633
1.7605E+18	0.21842188	-0.0451269	-0.0342925	-0.142259
1.7605E+18	0.2284043	-0.2893057	-0.1016068	-0.2185206
1.7605E+18	0.23838672	-0.4240065	-0.1529427	-0.1827606
1.7605E+18	0.24836938	-0.48139	-0.0965455	-0.0565979
1.7605E+18	0.25835205	-0.5334501	0.06357217	0.10899371

Processing the sensor data

Importing the .csv into the google collab notebook and into a Pandas dataframe (DF) was an easy process. The time data could be dropped from the DF easily leaving us with just acceleration data. We leveraged some code from the magic wand example to double-integrate the data and then plot it. This showed that the data we collected was not perfect and that more pre-processing was going to be needed to not only get the shapes we wanted out of the acceleration data, but to get coherent data for the AI to train on.

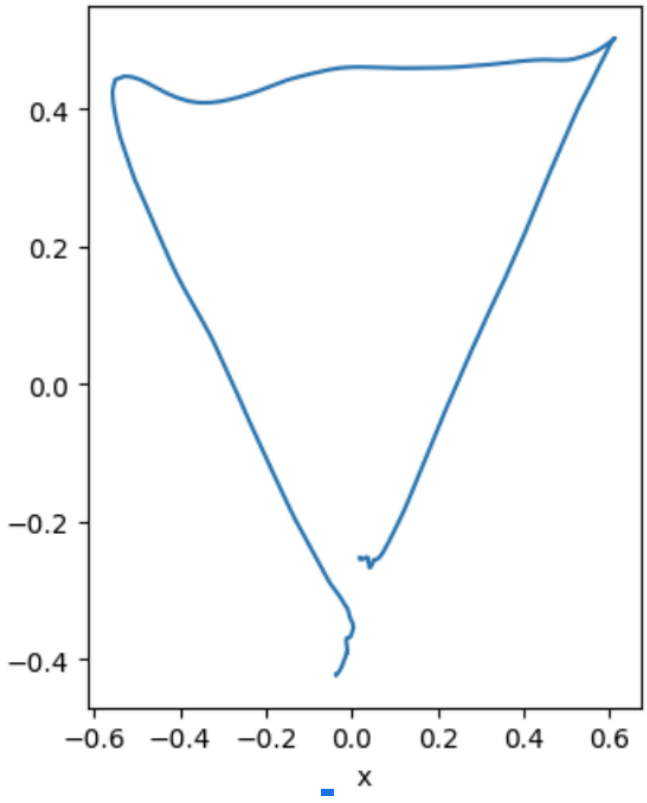




The figures on the right are two examples of the best plots we were able to get without any preprocessing, just double integration of the acceleration data. Some ideas for why the data doesn't match our expectations are that there may be some delay in starting the recording that cannot be measured simply, the orientation of the phone changes throughout the gesture so the acceleration data alone cannot represent the path through space, or that the signal to noise ratio (SNR) is not high enough for the gestures we recorded. If the start time of the recording is not instantaneous, we can start the recording and wait a good amount of time and then trim the uninteresting pause at the start, ensuring that the gesture was captured completely at the cost of an additional preprocessing step. If the orientation of the phone is changing substantially, we can involve the orientation data in our calculations to get a more true representation of reality. If the SNR is the concern, then we can increase the acceleration of the phone at each vertex, increasing the SNR. An additional measure we can take to address the SNR is to use a low-pass, or band-pass filter to remove the high frequency noise, ideally leaving us with the regular movements of the gesture.

We experimented with using different filters to augment the data, using both low-pass and band-pass filters with varying degrees of success. One sample would look very accurate while another would not look any better. With this we decided to approach it from another angle, but with keeping this lesson in mind.

After lots of trial and error with different combinations of accelerometer, gravity, orientation, and gyroscope data, we found that using acceleration and orientation together nets us a very promising image of our movement. The figure to the right shows the result of combining acceleration and orientation data. With this revelation in mind, we recollected all of our training data because we incorrectly presumed that we wouldn't need orientation data if we kept the phone steady enough.



We have pasted the main script that adjusts the data below. It looks more complicated than it is. Essentially it just double integrates the accelerometer data in order to get the position and then uses the orientation data to correct any odd changes to the accelerometer data. The reason it is so key to include the orientation data is because any small rotation in your phone while collecting the data causes the accelerometer data to be incorrect when converting to position. Any small error in the accelerometer data causes very big errors in position because the errors propagate throughout the integration process.

```
# Simple & sane: Accelerometer + Orientation → world-frame linear accel → integrate to X-Z position
import numpy as np
import pandas as pd
from pathlib import Path
from scipy.integrate import cumulative_trapezoid as cumtrapz

PLANE  = "xz"        # use world X-Z (you said +x left, +z up)
SCALE  = 0.35        # fit your plot limits
G      = 9.80665

def _read_xyz_time(fp, fs_fallback=100.0):
    df = pd.read_csv(fp)
    cols = {c.lower(): c for c in df.columns}
    if "seconds_elapsed" in cols:
```

```
        t = pd.to_numeric(df[cols["seconds_elapsed"]],
errors="coerce").to_numpy(float)
    elif "time" in cols:
        t = pd.to_numeric(df[cols["time"]],
errors="coerce").to_numpy(float)
    else:
        t = np.arange(len(df), dtype=float) / fs_fallback
    x = pd.to_numeric(df[cols["x"]], errors="coerce").to_numpy(float)
    y = pd.to_numeric(df[cols["y"]], errors="coerce").to_numpy(float)
    z = pd.to_numeric(df[cols["z"]], errors="coerce").to_numpy(float)
    return t, x, y, z

def _read_orientation(fp):
    """Return (t, qw, qx, qy, qz) from Orientation.csv which may have
quaternion or yaw/pitch/roll."""
    df = pd.read_csv(fp)
    cols = {c.lower(): c for c in df.columns}
    # time
    if "seconds_elapsed" in cols:
        t = pd.to_numeric(df[cols["seconds_elapsed"]],
errors="coerce").to_numpy(float)
    elif "time" in cols:
        t = pd.to_numeric(df[cols["time"]],
errors="coerce").to_numpy(float)
    else:
        t = np.arange(len(df), dtype=float) / 100.0

    # quaternion present?
    if all(k in cols for k in ("qw","qx","qy","qz")):
        qw = pd.to_numeric(df[cols["qw"]],
errors="coerce").to_numpy(float)
        qx = pd.to_numeric(df[cols["qx"]],
errors="coerce").to_numpy(float)
        qy = pd.to_numeric(df[cols["qy"]],
errors="coerce").to_numpy(float)
        qz = pd.to_numeric(df[cols["qz"]],
errors="coerce").to_numpy(float)
    elif all(k in cols for k in ("w","x","y","z")):
        qw = pd.to_numeric(df[cols["w"]], errors="coerce").to_numpy(float)
        qx = pd.to_numeric(df[cols["x"]], errors="coerce").to_numpy(float)
```

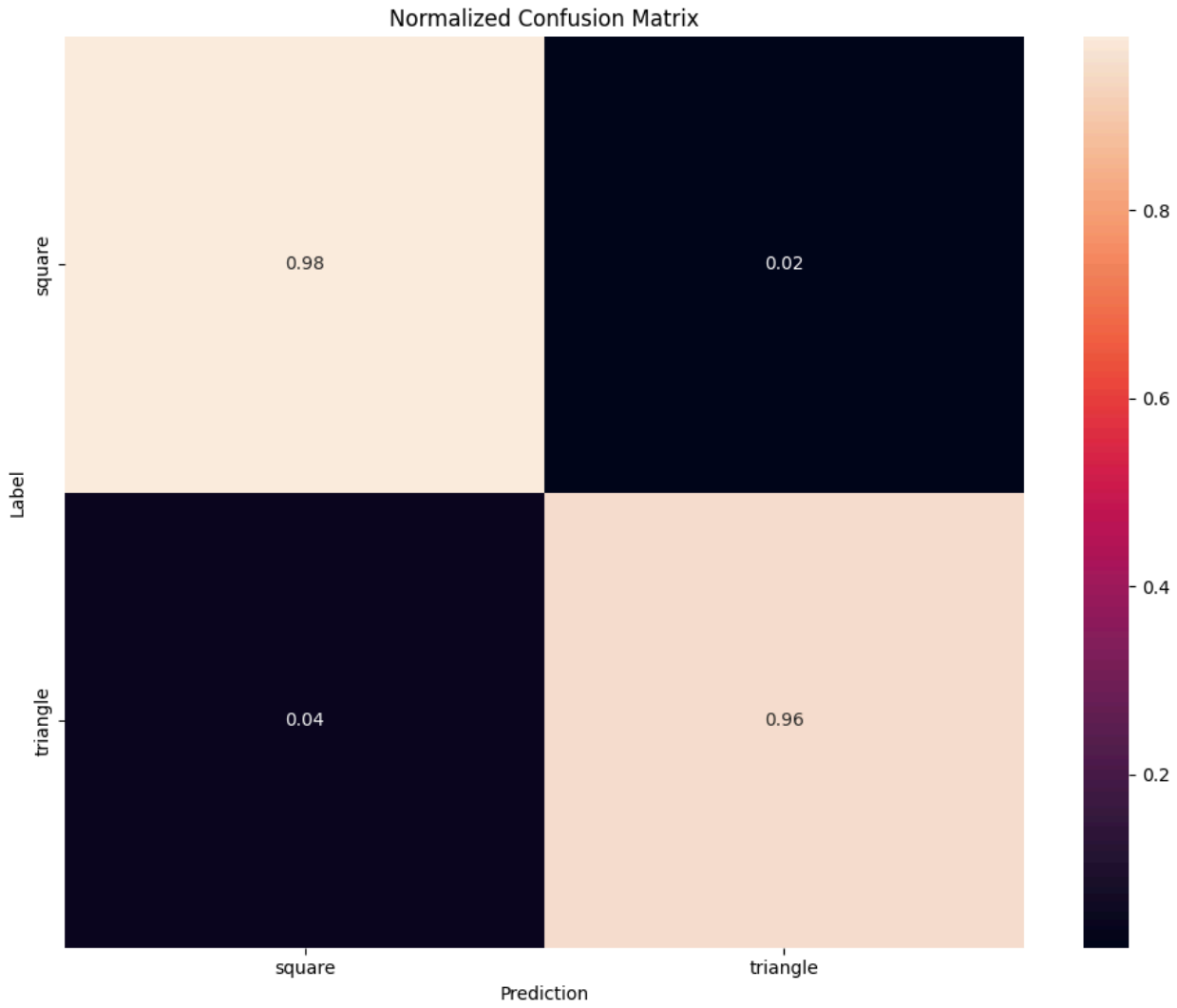
```
        qy = pd.to_numeric(df[cols["y"]], errors="coerce").to_numpy(float)
        qz = pd.to_numeric(df[cols["z"]], errors="coerce").to_numpy(float)
    else:
        # yaw/pitch/roll (deg): expect columns like 'yaw','pitch','roll'
or 'azimuth','pitch','roll'
        yaw  = pd.to_numeric(df[cols.get("yaw", cols.get("azimuth"))],
errors="coerce").to_numpy(float)
        pitch= pd.to_numeric(df[cols["pitch"]],
errors="coerce").to_numpy(float)
        roll = pd.to_numeric(df[cols["roll"]],
errors="coerce").to_numpy(float)
        # ZYX euler -> quaternion
        r, p, y = np.deg2rad(roll), np.deg2rad(pitch), np.deg2rad(yaw)
        cy, sy = np.cos(y*0.5), np.sin(y*0.5)
        cp, sp = np.cos(p*0.5), np.sin(p*0.5)
        cr, sr = np.cos(r*0.5), np.sin(r*0.5)
        qw = cr*cp*cy + sr*sp*sy
        qx = sr*cp*cy - cr*sp*sy
        qy = cr*sp*cy + sr*cp*sy
        qz = cr*cp*sy - sr*sp*cy
    return t, qw, qx, qy, qz

def _quat_to_rot(qw, qx, qy, qz):
    """Quaternion (w,x,y,z) → 3x3 rotation matrix. Maps device->world."""
    # normalize
    s = np.sqrt(qw*qw + qx*qx + qy*qy + qz*qz) + 1e-12
    w, x, y, z = qw/s, qx/s, qy/s, qz/s
    xx, yy, zz = x*x, y*y, z*z
    xy, xz, yz = x*y, x*z, y*z
    wx, wy, wz = w*x, w*y, w*z
    R = np.array([
        [1-2*(yy+zz),   2*(xy-wz),     2*(xz+wy)],
        [2*(xy+wz),     1-2*(xx+zz),   2*(yz-wx)],
        [2*(xz-wy),     2*(yz+wx),     1-2*(xx+yy)],
    ], dtype=float)
    return R

def _accori_to_world_pos(acc_fp, ori_fp, label, plane=PLANE, scale=SCALE):
    # read
    t_a, ax, ay, az = _read_xyz_time(acc_fp)
```

```
    t_o, qw, qx, qy, qz = _read_orientation(ori_fp)

    # heuristic: if accel magnitude median ~1, assume 'g' and convert to
m/s^2
    mag = np.median(np.sqrt(ax*ax + ay*ay + az*az))
    if 0.6 < mag < 1.4:  # around 1 g
        ax, ay, az = ax*G, ay*G, az*G

    # interp quaternion to accel timestamps
    qw_i = np.interp(t_a, t_o, qw)
    qx_i = np.interp(t_a, t_o, qx)
    qy_i = np.interp(t_a, t_o, qy)
    qz_i = np.interp(t_a, t_o, qz)
    # renormalize
    qn = np.sqrt(qw_i*qw_i + qx_i*qx_i + qy_i*qy_i + qz_i*qz_i) + 1e-12
    qw_i, qx_i, qy_i, qz_i = qw_i/qn, qx_i/qn, qy_i/qn, qz_i/qn

    # rotate accel to world and subtract gravity
    N = len(t_a)
    awx = np.empty(N); awy = np.empty(N); awz = np.empty(N)
    for i in range(N):
        R = _quat_to_rot(qw_i[i], qx_i[i], qy_i[i], qz_i[i])  #
device->world
        a_world = R @ np.array([ax[i], ay[i], az[i]])
        a_world[2] -= G
        awx[i], awy[i], awz[i] = a_world

    # integrate (trapezoid) with simple end-velocity correction
    vx = cumtrapz(awx, t_a, initial=0.0)
    vy = cumtrapz(awy, t_a, initial=0.0)
    vz = cumtrapz(awz, t_a, initial=0.0)
    ramp = np.linspace(0.0, 1.0, N)
    vx -= ramp * vx[-1]; vy -= ramp * vy[-1]; vz -= ramp * vz[-1]
    px = cumtrapz(vx, t_a, initial=0.0)
    py = cumtrapz(vy, t_a, initial=0.0)
    pz = cumtrapz(vz, t_a, initial=0.0)

    # choose plane, center & scale for your plot
    if plane == "xz": X, Y = px, pz
    elif plane == "xy": X, Y = px, py
```

```
    elif plane == "yz": X, Y = py, pz
    else: raise ValueError("plane must be 'xz','xy','yz'")
    X -= X.mean(); Y -= Y.mean()
    s = max(np.std(X), np.std(Y), 1e-9)
    X, Y = (X/s)*scale, (Y/s)*scale
    pts = [{"x": float(x), "y": float(y)} for x, y in zip(X, Y)]
    return {"strokePoints": pts, "label": label, "filename": str(acc_fp)}

# ---- Build strokes: use only Accelerometer.csv + Orientation.csv in each
session ----
strokes = []
for label_dir in sorted(Path(ROOT).glob("*")):
    if not label_dir.is_dir(): continue
    for session_dir in sorted(label_dir.glob("*")):
        if not session_dir.is_dir(): continue
        acc = next((p for p in session_dir.glob("*.csv") if
"accelerometer" in p.name.lower()), None)
        ori = next((p for p in session_dir.glob("*.csv") if "orientation"
in p.name.lower()), None)
        if acc and ori:
            strokes.append(_accori_to_world_pos(acc, ori, label_dir.name,
plane=PLANE))
```

We choose to use position data to train the model rather than the straight accelerometer data for two reasons. One was because Pete Warden's example used positional data and if we could get our data to be positional then we could easily follow along with his tutorial and training the model from the positional data. The second reason is because the model will be much more accurate if we can first validate the data we are receiving. It is really difficult to look at the accelerometer data as a human and say, yes that is a square vs a triangle etc. But by plotting the data we could confirm that the data was accurate and the model should work on the fed data. This helped out a lot because it was very easy to get a high accuracy model after we had spent time making sure the data being fed was great. There are some obvious ways to increase accuracy without an even larger dataset such as using artificial jitter and noise generation to make the model more resilient. However, our model was very accurate using even just a few (20-30) samples so this was unnecessary.

After building and verifying and optimizing the model we were able to get this confusion matrix output which looks great:



As shown here, our accuracy was very high, >97%.