

```
1 !pip install --upgrade pip setuptools wheel
2 !pip cache purge
```

Requirement already satisfied: pip in /usr/local/lib/python3.12/dist-packages (25.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.12/dist-packages (80.9.0)
Requirement already satisfied: wheel in /usr/local/lib/python3.12/dist-packages (0.45.1)
Files removed: 6 (250 kB)

```
1 !pip install -q transformers torch sentencepiece openpyxl pandas sklearn
```

error: subprocess-exited-with-error

```
× python setup.py egg_info did not run successfully.
  exit code: 1
  ↳ See above for output.
```

note: This error originates from a subprocess, and is likely not a problem with pip.
Preparing metadata (setup.py) ... error

error: metadata-generation-failed

```
× Encountered error while generating package metadata.
  ↳ See above for output.
```

note: This is an issue with the package mentioned above, not pip.
hint: See above for details.

```
1 import os
2 import re
3 import pandas as pd
4 from transformers import pipeline
5 from typing import List
```

```
1 INPUT_CSV = "/content/sample_data/mobile_reviews - mobile_reviews.csv"
2 OUTPUT_XLSX = "/content/sample_data/mobile_review_model_ready.xlsx"
```

```
1 if not os.path.exists(INPUT_CSV):
2     raise FileNotFoundError(f"Input file not found: {INPUT_CSV}")
3 df = pd.read_csv(INPUT_CSV)
```

```
1 # Ensure a text column exists. Common column names: 'review', 'Review', 'text', 'review_text'
2 text_col_candidates = ['review', 'Review', 'text', 'review_text', 'Comments', 'comment', 'ReviewText']
3 text_col = None
4 for c in text_col_candidates:
5     if c in df.columns:
6         text_col = c
7         break
8 if text_col is None:
9     # fallback to first string-like column
10    for c in df.columns:
11        if pd.api.types.is_string_dtype(df[c]):
12            text_col = c
13            break
14 if text_col is None:
15    raise ValueError("No text column found in dataset. Please ensure the CSV has a review text column.")
16
17 print(f"Using review text column: {text_col}")
```

Using review text column: review_text

```

1 # 1) Attribute list and keyword mapping
2 # You can extend this dictionary to add more synonyms.
3 ATTR_KEYWORDS = {
4     "Battery": ["battery", "charge", "charging", "battery life", "drain", "power bank"],
5     "Camera": ["camera", "photo", "picture", "selfie", "image", "video", "zoom"],
6     "Design": ["design", "build", "body", "weight", "look", "stylish", "shape"],
7     "Display": ["screen", "display", "brightness", "resolution", "amoled", "lcd", "touch"],
8     "Performance": ["speed", "lag", "slow", "performance", "processor", "cpu", "fps", "stutter"],
9     "Price": ["price", "cost", "value", "expensive", "cheap", "costly", "worth"],
10    "Software": ["software", "ui", "os", "updates", "firmware", "bug", "crash"],
11    "Connectivity": ["wifi", "bluetooth", "network", "signal", "sim", "4g", "5g", "hotspot"],
12    "Speaker": ["speaker", "audio", "sound", "mic", "earpiece"],
13    "Sensor": ["fingerprint", "face unlock", "sensor", "gps", "proximity"]
14 }
15
16 # Precompile regex for attributes
17 ATTR_PATTERNS = {attr: re.compile(r'\b(' + "|".join(re.escape(w) for w in kws) + r')\b', flags=re.I)
18     for attr, kws in ATTR_KEYWORDS.items()}
19
20 def extract_attributes(text: str) -> List[str]:
21     found = []
22     if not isinstance(text, str):
23         return found
24     for attr, patt in ATTR_PATTERNS.items():
25         if patt.search(text):
26             found.append(attr)
27     return found if found else ["General"] # fallback attribute "General"
28

```

```

1 # 2) Sentiment + emotion pipelines using transformers
2 print("Loading transformers pipelines (these will download pretrained models). This may take ~1-2 minutes.")
3 sentiment_pipe = pipeline("sentiment-analysis", model="nlptown/bert-base-multilingual-uncased-sentiment")
4 # emotion model
5 emotion_pipe = pipeline("text-classification", model="bhadresh-savani/bert-base-go-emotion", return_all_scores=True)
6
7 def predict_sentiment(text: str):
8     # nlptown returns stars. We'll map 1-2 -> Negative, 3 -> Neutral, 4-5 -> Positive
9     out = sentiment_pipe(text[:512]) # truncate for long text
10    # output example: {'label': '3', 'score': 0.5}
11    label = out[0]['label']
12    try:
13        stars = int(label)
14    except:
15        stars = 3
16    if stars <= 2:
17        return "Negative", stars
18    elif stars == 3:
19        return "Neutral", stars
20    else:
21        return "Positive", stars
22
23 def predict_emotions(text: str):
24     # returns top emotions (score > threshold)
25     out = emotion_pipe(text[:512])[0] # list of dicts with label & score
26     # sort descending
27     sorted_ = sorted(out, key=lambda x: x['score'], reverse=True)
28     # choose top emotions with score > 0.15 (tunable)
29     top = [d['label'] for d in sorted_ if d['score'] > 0.15]
30     if not top:
31         top = [sorted_[0]['label']]
32     return top

```

Loading transformers pipelines (these will download pretrained models). This may take ~1-2 minutes.
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.

```
warnings.warn(  
config.json: 100%                               953/953 [00:00<00:00, 21.6kB/s]  
  
model.safetensors: 100%                         669M/669M [00:17<00:00, 63.1MB/s]  
  
tokenizer_config.json: 100%                     39.0/39.0 [00:00<00:00, 522B/s]  
  
vocab.txt:      872k/? [00:00<00:00, 4.67MB/s]  
  
special_tokens_map.json: 100%                  112/112 [00:00<00:00, 2.52kB/s]  
Device set to use cpu  
config.json:      2.10k/? [00:00<00:00, 168kB/s]  
  
pytorch_model.bin: 100%                       438M/438M [00:14<00:00, 36.3MB/s]  
  
tokenizer_config.json: 100%                    333/333 [00:00<00:00, 30.3kB/s]  
  
model.safetensors: 100%                       438M/438M [00:05<00:00, 102MB/s]  
  
vocab.txt:      232k/? [00:00<00:00, 11.4MB/s]  
  
tokenizer.json:      466k/? [00:00<00:00, 25.3MB/s]  
  
special_tokens_map.json: 100%                  112/112 [00:00<00:00, 11.1kB/s]  
Device set to use cpu  
/usr/local/lib/python3.12/dist-packages/transformers/pipelines/text_classification.py:111: UserWarning: `return_all_scores` is now deprecated, if want a similar functionality use `top_k=None` instead of `re  
warnings.warn(  

```

```
1 # 3) Run attribute extraction + sentiment+emotion per review  
2 records = []  
3 n = len(df)  
4 for idx, row in df.iterrows():  
5     text = str(row[text_col]) if pd.notnull(row[text_col]) else ""  
6     attrs = extract_attributes(text)  
7     per_attr_results = []  
8     # We run sentiment/emotion on the full review, but we will attach the result to each attribute found.  
9     # (You may later re-run on the attribute sentence span if you prefer sentence-level segmentation.)  
10    sentiment_label, sentiment_star = predict_sentiment(text)  
11    emotions = predict_emotions(text)  
12    # Compose attributes result string  
13    for a in attrs:  
14        per_attr_results.append(f"{a}:{(sentiment_label)}, {'.'.join(emotions)}")  
15    results_str = " ; ".join(per_attr_results)  
16    records.append(results_str)  
17  
18 # Add a new column preserving original frame  
19 df_out = df.copy()  
20 df_out["Attributes_Analysis"] = records
```

```
1 # Also add structured columns of aggregated Sentiment/Emotions if not present:  
2 # create numeric sentiment cols: Sentiment_Pos, Sentiment_Neg, Sentiment_Neu as floats (0/1)  
3 sent_pos = []  
4 sent_neg = []  
5 sent_neu = []  
6 pred_stars = []  
7
```

```

8 # Re-run sentiment to create numeric columns (cheap compared to emotion model)
9 for idx, row in df_out.iterrows():
10     text = str(row[text_col]) if pd.notnull(row[text_col]) else ""
11     s_label, s_star = predict_sentiment(text)
12     pred_stars.append(s_star)
13     sent_pos.append(1.0 if s_label == "Positive" else 0.0)
14     sent_neg.append(1.0 if s_label == "Negative" else 0.0)
15     sent_neu.append(1.0 if s_label == "Neutral" else 0.0)
16
17 df_out["Sentiment_Star"] = pred_stars
18 df_out["Sentiment_Pos"] = sent_pos
19 df_out["Sentiment_Neg"] = sent_neg
20 df_out["Sentiment_Neu"] = sent_neu
21

```

```

1 # For emotions, create one-hot-ish mean columns for a small set of emotions (Joy, Anger, Sadness, Fear, Disgust, Surprise)
2 emotion_list = ["joy", "anger", "sadness", "fear", "disgust", "surprise"]
3 # run emotion pipe once and create flags
4 emotion_flags = {e: [] for e in emotion_list}
5 for idx, row in df_out.iterrows():
6     text = str(row[text_col]) if pd.notnull(row[text_col]) else ""
7     ems = predict_emotions(text)
8     ems_lower = [e.lower() for e in ems]
9     for e in emotion_list:
10         emotion_flags[e].append(1.0 if e in ems_lower else 0.0)
11
12 for e in emotion_list:
13     colname = "Emotion_" + e.capitalize()
14     df_out[colname] = emotion_flags[e]
15
16 # Add Attr_<name> binary columns for model-ready features
17 for attr in ATTR_KEYWORDS.keys():
18     col = "Attr_" + attr
19     df_out[col] = df_out["Attributes_Analysis"].apply(lambda s: 1.0 if f"{attr}:" in s else 0.0)
20
21 # If there is a 'Rating' column, keep it but don't change here
22 # Save to Excel (keep original columns order + new cols appended)
23 df_out.to_excel(OUTPUT_XLSX, index=False)
24 print(f"Saved processed file to: {OUTPUT_XLSX}")

```

Saved processed file to: /content/sample_data/mobile_review_model_ready.xlsx

Second Part of the Problem

```
1 !pip install -q statsmodels pandas openpyxl numpy scipy
```

```

1 import os
2 import pandas as pd
3 import numpy as np
4 import statsmodels.api as sm
5 import statsmodels.formula.api as smf
6 from statsmodels.discrete.count_model import ZeroInflatedPoisson, ZeroInflatedNegativeBinomialP
7 import warnings
8 warnings.filterwarnings("ignore")

```

```

1 INPUT_XLSX = "/content/sample_data/mobile_review_model_ready.xlsx"
2 if not os.path.exists(INPUT_XLSX):

```

```
3 raise FileNotFoundError(f"Required file not found: {INPUT_XLSX}. Run gen_attribute_sentiment_excel.py first.")
```

```
1 df = pd.read_excel(INPUT_XLSX)
2 print("Loaded:", INPUT_XLSX, "shape:", df.shape)
3
4 # ----- 1) Prepare Rating (round to integers) -----
5 if "Rating" not in df.columns:
6     # attempt common alternatives
7     for c in df.columns:
8         if c.lower().startswith("rating"):
9             df["Rating"] = df[c]
10            break
11 else:
12     raise ValueError("No Rating column found in excel. Please ensure sheet has 'Rating'.")
13
```

Loaded: /content/sample_data/mobile_review_model_ready.xlsx shape: (300, 26)

```
1 # Disallow NA ratings
2 df = df[df["Rating"].notna()].copy()
3 df["Rating_Rounded"] = df["Rating"].round().astype(int)
4 df["Rating_Rounded"] = df["Rating_Rounded"].clip(lower=0) # ensure non-negative
5
```

```
1 print("Rating distribution (rounded):")
2 print(df["Rating_Rounded"].value_counts().sort_index())
```

Rating distribution (rounded):

Rating_Rounded

```
1 43
2 53
3 45
4 66
5 93
```

Name: count, dtype: int64

```
1 # zeros proportion
2 zero_count = (df["Rating_Rounded"] == 0).sum()
3 zero_prop = zero_count / len(df)
4 print(f"Zeros: {zero_count} / {len(df)} ({zero_prop:.3f})")
```

Zeros: 0 / 300 (0.000)

```
1 # ----- 2) Build predictor formula -----
2 # Use any Sentiment_*, Emotion_*, Attr_* columns that exist
3 predictors = [c for c in df.columns if c.startswith("Sentiment_") or c.startswith("Emotion_") or c.startswith("Attr_")]
4 if not predictors:
5     raise ValueError("No Sentiment_/Emotion_/Attr_ predictors found in the file.")
6
7 formula_rhs = " + ".join(predictors)
8 formula = "Rating_Rounded ~ " + formula_rhs
9 print("Using formula:", formula)
10
```

Using formula: Rating_Rounded ~ Sentiment_Star + Sentiment_Pos + Sentiment_Neg + Sentiment_Neu + Emotion_Joy + Emotion_Anger + Emotion_Sadness + Emotion_Fear + Emotion_Disgust + Emotion_Surprise + Attr_Batte

```
1 # Convert boolean-like columns to numeric if needed
2 for p in predictors:
3     df[p] = pd.to_numeric(df[p], errors='coerce').fillna(0.0)
```

```
1 # ----- 3) Fit models -----
2 results = {}
```

```
1 # Poisson GLM
2 print("Fitting Poisson GLM...")
3 poisson = smf.glm(formula=formula, data=df, family=sm.families.Poisson()).fit()
4 results['Poisson'] = poisson
```

Fitting Poisson GLM...

```
1 # Negative Binomial GLM
2 print("Fitting Negative Binomial GLM...")
3 nb = smf.glm(formula=formula, data=df, family=sm.families.NegativeBinomial()).fit()
4 results['NegBin'] = nb
```

Fitting Negative Binomial GLM...

```
1 # --- Robust ZIP fit (replacement for the failing block) ---
2 #import numpy as np
3 #from statsmodels.discrete.count_model import ZeroInflatedPoisson
4 #import warnings
5
6 print("Fitting Zero-Inflated Poisson (ZIP)")
7
8 # y (endog)
9 y = df["Rating_Rounded"].values
10
11 # Build exog dataframe (no const yet)
12 exog_df = df[predictors].astype(float).copy()
13
14 # 1) Drop zero / near-zero variance columns
15 stds = exog_df.std(axis=0, ddof=0)
16 near_zero = stds[stds < 1e-8].index.tolist()
17 if near_zero:
18     print("Dropping near-constant columns:", near_zero)
19     exog_df.drop(columns=near_zero, inplace=True)
20
21 # 2) Drop one of any near-perfectly correlated pairs (abs corr > 0.999)
22 if exog_df.shape[1] > 1:
23     corr = exog_df.corr().abs()
24     upper = corr.where(np.triu(np.ones(corr.shape), k=1).astype(bool))
25     to_drop_corr = [col for col in upper.columns if any(upper[col] > 0.999)]
26     if to_drop_corr:
27         print("Dropping highly correlated columns:", to_drop_corr)
28         exog_df.drop(columns=to_drop_corr, inplace=True)
29
30 # Ensure we have at least one predictor (if none, fall back to intercept-only model)
31 if exog_df.shape[1] == 0:
32     print("No valid predictors remain after cleaning. Fitting intercept-only ZIP.")
33     exog_clean = pd.DataFrame({'const': np.ones(len(df))})
34     exog_infl = None # will let statsmodels handle inflation with default
35 else:
36     exog_clean = exog_df.copy()
37     # add constant for the count model exog
38     exog = sm.add_constant(exog_clean, has_constant='add').astype(float)
39     # Use exog_infl WITHOUT a constant to reduce collinearity/identifiability issues
40     exog_infl = exog_clean.astype(float)
41
42 # Try fitting with a couple of solvers, catching singular matrix errors
```

```

43 zip_model = None
44 fit_errors = []
45 try:
46     if exog_df.shape[1] == 0:
47         zip_model = ZeroInflatedPoisson(endog=y, exog=None, exog_infl=None, inflation='logit').fit(method='newton', maxiter=100, disp=False)
48     else:
49         zip_model = ZeroInflatedPoisson(endog=y, exog=exog, exog_infl=exog_infl, inflation='logit').fit(method='newton', maxiter=100, disp=False)
50     print("ZIP fitted with 'newton'.")
51 except np.linalg.LinAlgError as e:
52     fit_errors.append(("newton", str(e)))
53     warnings.warn(f"ZIP newton solver failed: {e}. Trying 'bfgs'...")
54     try:
55         zip_model = ZeroInflatedPoisson(endog=y, exog=exog, exog_infl=exog_infl, inflation='logit').fit(method='bfgs', maxiter=200, disp=False)
56         print("ZIP fitted with 'bfgs'.")
57     except Exception as e2:
58         fit_errors.append(("bfgs", str(e2)))
59         warnings.warn(f"ZIP bfgs solver failed: {e2}. Trying 'nm'...")
60         try:
61             zip_model = ZeroInflatedPoisson(endog=y, exog=exog, exog_infl=exog_infl, inflation='logit').fit(method='nm', maxiter=400, disp=False)
62             print("ZIP fitted with 'nm'.")
63         except Exception as e3:
64             fit_errors.append(("nm", str(e3)))
65             zip_model = None
66
67 if zip_model is None:
68     print("All attempts to fit ZIP failed. Errors:", fit_errors)
69     raise RuntimeError("ZeroInflatedPoisson fit failed after clean-up and solver retries. "
70                        "Inspect predictors for multicollinearity or provide fewer predictors.")
71 else:
72     results['ZIP'] = zip_model
73     print("ZIP model stored in results['ZIP'].")
74

```

Fitting Zero-Inflated Poisson (ZIP)

Dropping near-constant columns: ['Sentiment_Star', 'Sentiment_Pos', 'Sentiment_Neg', 'Sentiment_Neu', 'Emotion_Joy', 'Emotion_Anger', 'Emotion_Fear', 'Emotion_Disgust', 'Emotion_Surprise', 'Attr_Connectivity']

ZIP fitted with 'newton'.

ZIP model stored in results['ZIP'].

```

1 # Zero-Inflated Negative Binomial (ZINB - P variant)
2 print("Fitting Zero-Inflated Negative Binomial (ZINB-P)...")
3 zinb = ZeroInflatedNegativeBinomialP(endog=y, exog=exog, exog_infl=exog, inflation='logit').fit(method='newton', maxiter=100, disp=False)
4 results['ZINB'] = zinb

```

Fitting Zero-Inflated Negative Binomial (ZINB-P)...

```

1 # ----- 4) Zero-truncated approximation -----
2 # There is no direct built-in ZeroTruncated Poisson in statsmodels; approximate by fitting
3 # Poisson/NegBin on data where Rating_Rounded > 0 (i.e., remove zeros) – this is an approximation.
4 df_pos = df[df["Rating_Rounded"] > 0].copy()
5 if len(df_pos) < 10:
6     print("Too few positive counts for zero-truncated approximation.")
7 else:
8     formula_pos = "Rating_Rounded ~ " + formula_rhs
9     print("Fitting Poisson on positive-only (approx Zero-Truncated Poisson)...")
10    pt = smf.glm(formula=formula_pos, data=df_pos, family=sm.families.Poisson()).fit()
11    results['PosOnly_Poisson'] = pt
12
13    print("Fitting NegBin on positive-only (approx Zero-Truncated NegBin)...")
14    ptnb = smf.glm(formula=formula_pos, data=df_pos, family=sm.families.NegativeBinomial()).fit()
15    results['PosOnly_NegBin'] = ptnb

```

Fitting Poisson on positive-only (approx Zero-Truncated Poisson)...

Fitting NegBin on positive-only (approx Zero-Truncated NegBin)...

```
1 # ----- 5) Compare models (AIC/BIC/LogLik) -----
2 comp = []
3 for name, res in results.items():
4     try:
5         aic = res.aic
6         bic = res.bic
7         llf = res.llf
8     except Exception:
9         # for discrete models, attributes are different
10        aic = getattr(res, "aic", np.nan)
11        bic = getattr(res, "bic", np.nan)
12        llf = getattr(res, "llf", np.nan)
13    comp.append({"Model": name, "AIC": aic, "BIC": bic, "LogLike": llf, "Nparams": getattr(res, "df_model", np.nan)})
14 comp_df = pd.DataFrame(comp).sort_values("AIC")
15 print("\nModel comparison sorted by AIC:")
16 print(comp_df)
17
18 comp_df.to_csv("/content/sample_data/model_comparison.csv", index=False)
19 print("Saved model comparison to /content/sample_data/model_comparison.csv")
```

Model comparison sorted by AIC:

| | Model | AIC | BIC | LogLike | Nparams |
|---|-----------------|--------------|--------------|---------------|---------|
| 0 | Poisson | 1120.133650 | -1456.062372 | -551.066825 | 8.0 |
| 4 | PosOnly_Poisson | 1120.133650 | -1456.062372 | -551.066825 | 8.0 |
| 1 | NegBin | 1428.552181 | -1606.413901 | -705.276090 | 8.0 |
| 5 | PosOnly_NegBin | 1428.552181 | -1606.413901 | -705.276090 | 8.0 |
| 2 | ZIP | 22408.820622 | 22471.784924 | -11187.410311 | 8.0 |
| 3 | ZINB | NaN | NaN | NaN | 8.0 |

Saved model comparison to /content/sample_data/model_comparison.csv

Model Comparison Table

| Model | AIC | BIC | LogLike | Nparams |
|-----------------|-------------|--------------|--------------|---------|
| Poisson | 1120.13365 | -1456.062372 | -551.0668251 | 8 |
| PosOnly_Poisson | 1120.13365 | -1456.062372 | -551.0668251 | 8 |
| NegBin | 1428.552181 | -1606.413901 | -705.2760904 | 8 |
| PosOnly_NegBin | 1428.552181 | -1606.413901 | -705.2760904 | 8 |
| ZIP | 22408.82062 | 22471.78492 | -11187.41031 | 8 |
| ZINB | | | | 8 |

```
1 # Pick best model by AIC
2 best_model_name = comp_df.iloc[0]["Model"]
3 print("Best model by AIC:", best_model_name)
```

Best model by AIC: Poisson

```
1 # Save best model summary
2 best_res = results[best_model_name]
3 with open("/content/sample_data/best_model_summary.txt", "w") as f:
4     f.write(best_res.summary().as_text())
5 print("Saved best model summary to /content/sample_data/best_model_summary.txt")
```



```

1 # ---- 6) Explore weights used in "Problem Score" (optimize) ----
2 # If summary/aggregates exist in dataframe (Brand/Attribute), compute problem score per your example:
3 if ("Brand" in df.columns) and ("Attribute" in df.columns):
4     agg = df.groupby(["Brand", "Attribute"]).agg(
5         Avg_Rating=("Rating_Rounded", "mean"),
6         Count=("Rating_Rounded", "size"),
7         Sentiment_Neg=("Sentiment_Neg", "mean"),
8         Emotion_Anger=("Emotion_Anger", "mean") if "Emotion_Anger" in df.columns else ("Emotion_Anger", lambda s: 0.0),
9         Emotion_Disgust=("Emotion_Disgust", "mean") if "Emotion_Disgust" in df.columns else ("Emotion_Disgust", lambda s: 0.0),
10        Emotion_Sadness=("Emotion_Sadness", "mean") if "Emotion_Sadness" in df.columns else ("Emotion_Sadness", lambda s: 0.0)
11    ).reset_index()
12    # Fill any missing emotion columns with zeros
13    for col in ["Emotion_Anger", "Emotion_Disgust", "Emotion_Sadness"]:
14        if col not in agg.columns:
15            agg[col] = 0.0
16
17    # Grid search weights w1 (neg), w2 (anger+disgust), w3 (sadness) such that w1+w2+w3=1, w_i >= 0
18    print("Optimizing problem-score weights by correlation with Avg_Rating (we look for negative correlation).")
19    best_corr = 0.0
20    best_weights = (0.6, 0.3, 0.1)
21    for w1 in np.linspace(0,1,11):
22        for w2 in np.linspace(0,1-w1,11):
23            w3 = 1.0 - w1 - w2
24            agg["Problem_Score"] = w1*agg["Sentiment_Neg"] + w2*(agg["Emotion_Anger"] + agg["Emotion_Disgust"]) + w3*agg["Emotion_Sadness"]
25            # We want Problem_Score to be negatively correlated with Avg_Rating (higher problem -> lower rating)
26            corr = agg["Problem_Score"].corr(agg["Avg_Rating"])
27            if np.isnan(corr):
28                continue
29            if corr < best_corr: # more negative is better
30                best_corr = corr
31                best_weights = (w1, w2, w3)
32    print("Best weights (w_sent_neg, w_anger+disgust, w_sadness):", best_weights, "corr:", best_corr)
33    agg["Problem_Score"] = best_weights[0]*agg["Sentiment_Neg"] + best_weights[1]*(agg["Emotion_Anger"] + agg["Emotion_Disgust"]) + best_weights[2]*agg["Emotion_Sadness"]
34    agg.to_csv("/content/sample_data/brand_attribute_problem_score.csv", index=False)
35    print("Saved brand-attribute problem score file to /content/sample_data/brand_attribute_problem_score.csv")
36 else:
37     print("Brand/Attribute columns not present; skipping problem score optimization.")
38

```

Brand/Attribute columns not present; skipping problem score optimization.

```

1 print("All done. Check /content/sample_data for outputs:")
2 print(" - model_comparison.csv")
3 print(" - best_model_summary.txt")
4 print(" - brand_attribute_problem_score.csv (if available)")

```

All done. Check /content/sample_data for outputs:

- model_comparison.csv
- best_model_summary.txt
- brand_attribute_problem_score.csv (if available)

