BioHackthon

🔑 Predict embryonic stage from single-cell transcriptomic data.

Bigger picture (what's really going on)

- **Goal:** From each cell's RNA counts, guess **where it is on the embryonic timeline** (its stage).
- **Why it's hard:**
    - Each cell has **~20,000 genes** (too many, noisy).
    - Data comes from **two studies** (batch effects = they look different).
    - Some stages have **few cells** (class imbalance).
- **Winning idea:** Turn the huge gene table into a **small set of smart numbers** (features) that change in a **smooth way from early → late**. Then a **simple model** learns the mapping.
- **What the judges score:**
    - As a **number** (continuous stage): **MSE & R^2**
    - As a **label** (stage class): **Macro-F1**
    - Plus: clean preprocessing, no leakage, reproducible code.

Think of it like this: you're compressing a big, messy song (all instruments = genes) into a few sliders (loudness, tempo, bass...) that clearly increase/decrease over time. Then a simple rule can tell what part of the song (stage) you're in.

step-by-step game plan

▼ 0) Prepare your project

```
# make a clean project
mkdir -p rsg-track2 && cd rsg-track2

# (optional) create a venv or conda env
python -m venv .venv
source .venv/bin/activate  # on Windows: .venv\Scripts\act
ivate

# install deps
pip install --upgrade pip
pip install scanpy anndata scikit-learn pandas numpy scipy
joblib
```

Put your `.h5ad` in this folder (e.g., `train_adata.h5ad`).

to extract the train_data :

```


import scanpy as sc
import pandas as pd

# Load the .h5ad file
adata = sc.read_h5ad("train_adata.h5ad")

# 1. Export the main data matrix (cells × genes)
# Convert to dense first if it's sparse (can be big!)
df = pd.DataFrame(adata.X.toarray(),
                  index=adata.obs_names,
                  columns=adata.var_names)
df.to_csv("matrix.csv")

# 2. Export cell metadata
```

```
adata.obs.to_csv("cell_metadata.csv")

# 3. Export gene metadata
adata.var.to_csv("gene_metadata.csv")
```

▼ 1) Load and check the data

- Open the notebook they gave you or start a new one.
- Load the file and **check two columns exist** in `adata.obs`:
  - `stage` (the target)
  - `dataset` (the batch/study id)
- If `stage` is like text (e.g., “early”, “mid”, “late” or “CS13, CS14…”), write down the **correct order** (early→late) so you can:
  - make a **number version** (0,1,2,…) for regression
  - keep a **label version** for classification

> Paste this near the top of your notebook. If you already ran normalize/log earlier, that’s fine; this cell doesn’t break anything.

```
import scanpy as sc
import pandas as pd, numpy as np, re

# 1.1) Load
train_adata = sc.read_h5ad("train_adata.h5ad")

# 1.2) Ensure batch column is named 'dataset'
if "dataset" not in train_adata.obs.columns:
    if "origin" in train_adata.obs.columns:
        train_adata.obs["dataset"] = train_adata.obs["orig
in"].astype(str)
        print("✅ Created train_adata.obs['dataset'] from
'origin'")
    else:
        raise ValueError("Need a batch column: neither 'da
taset' nor 'origin' in .obs")

# 1.3) Ensure 'stage' exists
if "stage" not in train_adata.obs.columns:
    raise ValueError("Missing target column 'stage' in .ob
s")

print("Unique stage labels:", sorted(train_adata.obs["stag
e"].astype(str).unique()))
print("Datasets (batches):")
print(train_adata.obs["dataset"].astype(str).value_counts
())

# 1.4) Decide order. If you KNOW the correct early→late or
der, put it here:
STAGE_ORDER_MANUAL = None
# Example: STAGE_ORDER_MANUAL = ["CS13","CS14","CS15","CS1
6","CS17","CS18","CS19","CS20"]

def _num_in(s):
    m = re.search(r"(\d+\.?\d*)", str(s))
    return float(m.group(1)) if m else None

labels = sorted(train_adata.obs["stage"].astype(str).uniqu
e())

if STAGE_ORDER_MANUAL:
    STAGE_ORDER = STAGE_ORDER_MANUAL
else:
    nums = [_num_in(s) for s in labels]
    if all(n is not None for n in nums):
        STAGE_ORDER = [s for s,_ in sorted(zip(labels, num
s), key=lambda x: x[1])]
    else:
        # fallback: alphabetical (edit later if needed)
        STAGE_ORDER = labels

print("→ Stage order (early→late):", STAGE_ORDER)

# 1.5) Set ordered categorical + make numeric & label targ
ets
train_adata.obs["stage"] = pd.Categorical(
    train_adata.obs["stage"].astype(str),
    categories=STAGE_ORDER,
    ordered=True
)
train_adata.obs["stage_reg"] = train_adata.obs["stage"].ca
t.codes.astype(float)  # 0,1,2,...
train_adata.obs["stage_cls"] = train_adata.obs["stage"].as
type(str)

train_adata.obs[["stage","stage_reg","stage_cls","datase
t"]].head()
```

2) Clean the counts so cells are comparable

- **Normalize** per cell to total 10,000 counts (CP10k).
- Take **log1p** (log makes numbers behave).
- **Batch correct** with **ComBat** using `obs["dataset"]` (removes study differences).
- Pick **Highly Variable Genes (HVGs)** (e.g., 3,000).
- **Scale** genes to z-scores (mean 0, sd 1).

(Tip: do this with Scanpy; all functions exist: `normalize_total`, `log1p`, `pp.combat`, `pp.highly_variable_genes`, `pp.scale`.)

- A log scale is used in gene expression analysis to make very large differences easier to handle and transforms the numbers using a log scale so differences are easier to see and interpret.

- Next, it removes hidden distortions between different experimental batches so that all samples are on an even playing field.
- ComBat helps correct for any artificial differences caused by these conditions so that the data can be fairly and accurately compared, solves batch differences (sometimes it overcorrects but biology)

> This is exactly your plan: normalize → log1p → ComBat → HVGs → scale.
>
> If you already did normalize/log1p in your notebook, you can keep them (they're idempotent).

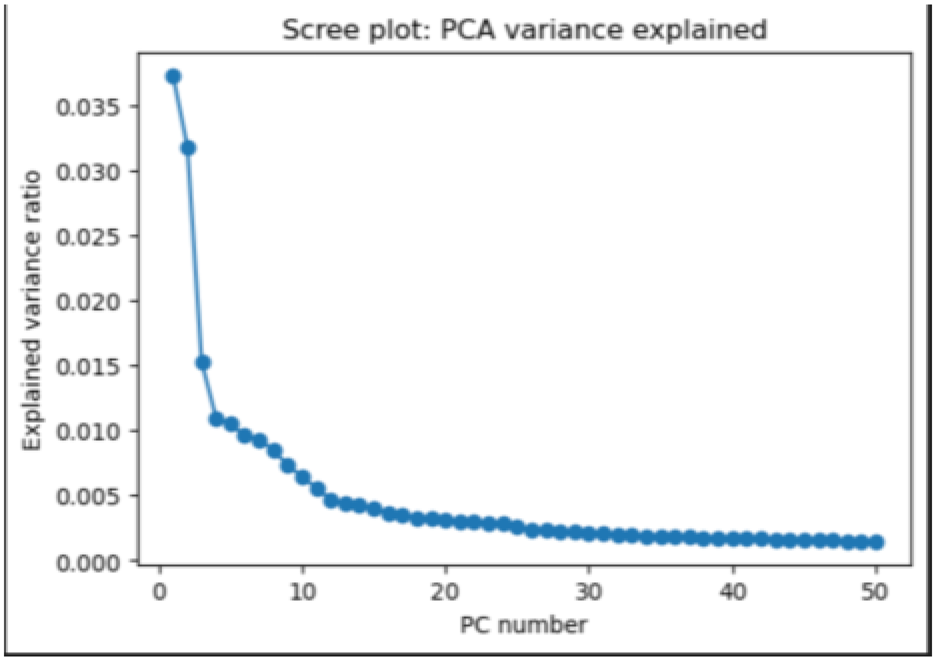
```
# 2.1) Basic QC + normalize + log
sc.pp.filter_cells(train_adata, min_genes=200)
sc.pp.normalize_total(train_adata, target_sum=1e4)
sc.pp.log1p(train_adata)

# 2.2) Batch correction (on log data)
sc.pp.combat(train_adata, key="dataset")

# 2.3) Select HVGs and scale
sc.pp.highly_variable_genes(
    train_adata,
    n_top_genes=3000,
    flavor="seurat_v3",
    batch_key="dataset"
)
train_adata_hvg = train_adata[:, train_adata.var["highly_v
ariable"]].copy()
sc.pp.scale(train_adata_hvg, max_value=10)
```

▼ 3) Make a small set of useful numbers per cell (features)

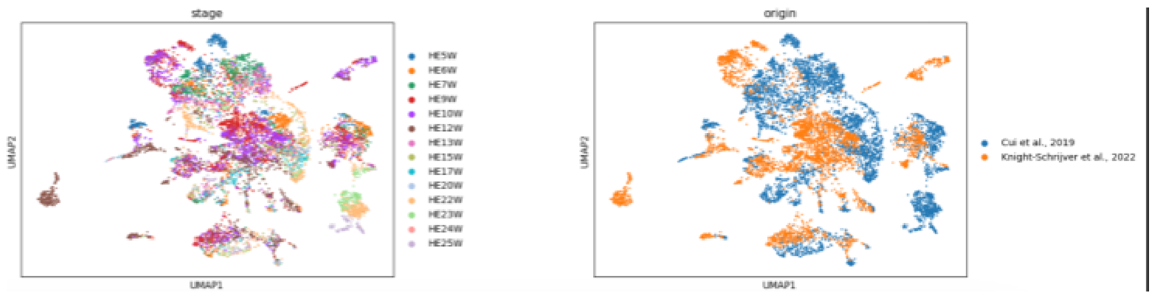
You want ~25–30 numbers total:

A) 12 data-driven numbers

- Run **PCA** and keep **PC1...PC12**



- The UMAP visualizations show that cells cluster primarily according to developmental stage while still maintaining robust integration across datasets from different studies. Distinct but partially overlapping stage-specific populations are evident, and the main biological groups are consistently observed in both datasets, indicating minimal batch effect and reliable dataset harmonization. This suggests successful integration and biological consistency, enabling meaningful downstream analysis of cell states and transitions across stages.



B) 4 simple biology scores (averages)

- **Muscle score:** TNNT2, TTN, MYH6, MYH7

- **Division score:** MKI67, TOP2A
- **Glycolysis score:** HK2, PFKM, ALDOA, ENO1, PKM, LDHA
- **OXPHOS score:** NDUFS1, SDHB, UQCRC1, COX4I1, ATP5F1A

  (If a gene isn't in your data, just skip it.)

  ⇒ better solution : we tried to enrich the data with gseapy

  We'll map your 4 biology modules to **Hallmark** sets:

  - Muscle → `HALLMARK_MYOGENESIS`
  - Division → `HALLMARK_E2F_TARGETS` (cell-cycle/proliferation)
  - Glycolysis → `HALLMARK_GLYCOLYSIS`
  - OXPHOS → `HALLMARK_OXIDATIVE_PHOSPHORYLATION`

C) 2 tiny ratios

- **EnergyRatio** = Glycolysis / OXPHOS
- **MaturityRatio** = MYH6 / MYH7

  (Add a tiny 0.001 to numerator/denominator to avoid divide-by-zero.)

Now you have a **feature table**: rows = cells, columns = those ~26–30 numbers.

We'll do **20 PCs + 4 module scores + 2 ratios** (my plan).

We compute module scores/ratios **on the log+ComBat data (before HVG cut)** so genes aren't missing.

▼ Version 1 (just 4 scores)

```
import scanpy as sc
import numpy as np
import pandas as pd
import gseapy as gp

import matplotlib.pyplot as plt

# 3.1) PCA (keep first 20 components) on scaled HVGs
```

```
sc.tl.pca(train_adata_hvg, n_comps=50)
pc = pd.DataFrame(
    train_adata_hvg.obsm["X_pca"][:, :20],
    index=train_adata_hvg.obs_names,
    columns=[f"PC{i+1}" for i in range(20)]
)

def _intersect(genes, varnames):
    return [g for g in genes if g in varnames]


#Get the explained variance ratio per PC from AnnData
explained_var = train_adata_hvg.uns['pca']['variance_ra
tio']

Plot the scree plot
plt.figure(figsize=(6,4))
plt.plot(range(1, len(explained_var)+1), explained_var,
marker='o')
plt.xlabel('PC number')
plt.ylabel('Explained variance ratio')
plt.title('Scree plot: PCA variance explained')
plt.show()

# 3.2a) Run GSEA/ORA on top cluster marker genes or HVG
s
# Example: use top 200 HVGs
gene_list = list(train_adata.var[train_adata.var["highl
y_variable"]].index)[:200]

enr = gp.enrichr(
    gene_list=gene_list,
    gene_sets='GO_Biological_Process_2021',  # or anoth
er set, e.g., 'KEGG_2021_Human'
    organism='Human',
    outdir=None,  # don't save files
    cutoff=0.05
)

# 3.2b) Expand core module gene lists using enrichment 
results if relevant genes are found
def expand_module_with_enrichment(base_genes, enr, rele
vant_terms):
    # Find genes from top enriched GO terms/pathways re
levant to this module
    extra_genes = []
    for idx, row in enr.results.iterrows():
        term = row['Term']
        if any(r in term for r in relevant_terms):
            extra_genes.extend([g for g in row['Gene
s'].split(';') if g not in base_genes])
    return list(set(base_genes + extra_genes))

MUSCLE   = ["TNNT2","TTN","MYH6","MYH7"]
MUSCLE = expand_module_with_enrichment(MUSCLE, enr, ["m
uscle", "myofibril", "sarcomere"])
DIVISION = ["MKI67","TOP2A"]
DIVISION = expand_module_with_enrichment(DIVISION, enr,
["cell cycle", "mitosis"])
GLYCO    = ["HK2","PFKM","ALDOA","ENO1","PKM","LDHA"]
GLYCO = expand_module_with_enrichment(GLYCO, enr, ["gly
colysis"])
OXPHOS   = ["NDUFS1","SDHB","UQCRC1","COX4I1","ATP5F1
A"]
OXPHOS = expand_module_with_enrichment(OXPHOS, enr, ["o
xidative phosphorylation"])

# 3.2c) Add core and new module scores
core_modules = [
    ("MuscleScore", MUSCLE),
    ("DivisionScore", DIVISION),
    ("GlycolysisScore", GLYCO),
    ("OXPHOSScore", OXPHOS),
]

# Auto-add NEW top enriched pathways as modules (top 2
not in core set)
additional_modules = []
core_terms = ["muscle","cell cycle","mitosis","glycolys
is","oxidative phosphorylation","myofibril","sarcomer
e"]
added_terms = set()
for idx, row in enr.results.iterrows():
    # Add only if term doesn't overlap with already pre
sent ones
    if all(core not in row["Term"].lower() for core in
core_terms):
        genes = [g for g in row['Genes'].split(';') if
g in train_adata.var_names]
        if len(genes) >= 2:
            additional_modules.append((row["Term"].repl
ace(" ","_") + "Score", genes))
            added_terms.add(row["Term"])
        if len(added_terms) == 2:  # Only add top 2 new
modules
            break

for name, genes in core_modules + additional_modules:
    g = _intersect(genes, train_adata.var_names)
    if len(g) >= 2:
        sc.tl.score_genes(train_adata, gene_list=g, sco
re_name=name, use_raw=False)
    else:
        train_adata.obs[name] = 0.0  # fallback if gene
s missing

# 3.3) Ratios
eps = 1e-3
```

```
def safe_gene(adata, g):
    if g in adata.var_names:
        x = adata[:, g].X
        x = x.A.astype(float).ravel() if hasattr(x,
"A") else np.asarray(x).ravel()
        return x
    return np.zeros(adata.n_obs, dtype=float)

myh6 = safe_gene(train_adata, "MYH6")
myh7 = safe_gene(train_adata, "MYH7")

ratios = pd.DataFrame(index=train_adata.obs_names)
ratios["EnergyRatio"]   = np.log(train_adata.obs["Glyco
lysisScore"] + eps) - np.log(train_adata.obs["OXPHOSSco
re"] + eps)
ratios["MaturityRatio"] = np.log(myh6 + eps) - np.log(m
yh7 + eps)

# 3.4) Assemble feature table X (including new modules)
module_names = [name for name, _ in core_modules + addi
tional_modules]
X = pc.join(train_adata.obs[module_names]).join(ratios)

# targets
y_reg = train_adata.obs["stage_reg"].values.astype(floa
t)
y_cls = train_adata.obs["stage_cls"].values.astype(str)

print(X.shape, "features; columns:", list(X.columns))
```

▼ Version 2 (multiple scores)

```
import scanpy as sc
import numpy as np
import pandas as pd
import gseapy as gp

# 3.1) PCA (keep first 20 components) on scaled HVGs
sc.tl.pca(train_adata_hvg, n_comps=50)
pc = pd.DataFrame(
    train_adata_hvg.obsm["X_pca"][:, :20],
    indexce=train_adata_hvg.obs_names,
    columns=[f"PC{i+1}" for i in range(20)]
)

def _intersect(genes, varnames):
    return [g for g in genes if g in varnames]

# 3.2a) Run GSEA/ORA on top cluster marker genes or HVG
s
# Example: use top 200 HVGs
gene_list = list(train_adata.var[train_adata.var["highl
y_variable"]].index)[:200]

enr = gp.enrichr(
    gene_list=gene_list,
    gene_sets='GO_Biological_Process_2021',  # or anoth
er set, e.g., 'KEGG_2021_Human'
    organism='Human',
    outdir=None,  # don't save files
    cutoff=0.05
)

# 3.2b) Expand core module gene lists using enrichment
results if relevant genes are found
def expand_module_with_enrichment(base_genes, enr, rele
vant_terms):
    # Find genes from top enriched GO terms/pathways re
levant to this module
    extra_genes = []
    for idx, row in enr.results.iterrows():
        term = row['Term']
        if any(r in term for r in relevant_terms):
            extra_genes.extend([g for g in row['Gene
s'].split(';') if g not in base_genes])
    return list(set(base_genes + extra_genes))

MUSCLE   = ["TNNT2","TTN","MYH6","MYH7"]
MUSCLE = expand_module_with_enrichment(MUSCLE, enr, ["m
uscle", "myofibril", "sarcomere"])
DIVISION = ["MKI67","TOP2A"]
DIVISION = expand_module_with_enrichment(DIVISION, enr,
["cell cycle", "mitosis"])
GLYCO    = ["HK2","PFKM","ALDOA","ENO1","PKM","LDHA"]
GLYCO = expand_module_with_enrichment(GLYCO, enr, ["gly
colysis"])
OXPHOS   = ["NDUFS1","SDHB","UQCRC1","COX4I1","ATP5F1
A"]
OXPHOS = expand_module_with_enrichment(OXPHOS, enr, ["o
xidative phosphorylation"])

# 3.2c) Add core and new module scores
core_modules = [
    ("MuscleScore", MUSCLE),
    ("DivisionScore", DIVISION),
    ("GlycolysisScore", GLYCO),
    ("OXPHOSScore", OXPHOS),
]

# Auto-add NEW top enriched pathways as modules (top 2
not in core set)
additional_modules = []
core_terms = ["muscle","cell cycle","mitosis","glycolys
is","oxidative phosphorylation","myofibril","sarcomer
e"]
added_terms = set()
for idx, row in enr.results.iterrows():
    # Add only if term doesn't overlap with already pre
sent ones
    if all(core not in row["Term"].lower() for core in
core_terms):
        genes = [g for g in row['Genes'].split(';') if
g in train_adata.var_names]
        if len(genes) >= 2:
            additional_modules.append((row["Term"].repl
ace(" ","_") + "Score", genes))
            added_terms.add(row["Term"])
        if len(added_terms) == 2:  # Only add top 2 new
modules
            break

for name, genes in core_modules + additional_modules:
    g = _intersect(genes, train_adata.var_names)
    if len(g) >= 2:
        sc.tl.score_genes(train_adata, gene_list=g, sco
re_name=name, use_raw=False)
    else:
        train_adata.obs[name] = 0.0  # fallback if gene
s missing

# 3.3) Ratios
eps = 1e-3
def safe_gene(adata, g):
    if g in adata.var_names:
        x = adata[:, g].X
        x = x.A.astype(float).ravel() if hasattr(x,
"A") else np.asarray(x).ravel()
        return x
    return np.zeros(adata.n_obs, dtype=float)

myh6 = safe_gene(train_adata, "MYH6")
myh7 = safe_gene(train_adata, "MYH7")
```

```
ratios = pd.DataFrame(index=train_adata.obs_names)
ratios["EnergyRatio"]   = np.log(train_adata.obs["Glyco
lysisScore"] + eps) - np.log(train_adata.obs["OXPHOSSco
re"] + eps)
ratios["MaturityRatio"] = np.log(myh6 + eps) - np.log(m
yh7 + eps)

# 3.4) Assemble feature table X (including new modules)
module_names = [name for name, _ in core_modules + addi
tional_modules]
X = pc.join(train_adata.obs[module_names]).join(ratios)

# targets
y_reg = train_adata.obs["stage_reg"].values.astype(floa
t)
y_cls = train_adata.obs["stage_cls"].values.astype(str)

print(X.shape, "features; columns:", list(X.columns))
```

You now have exactly what you planned: **~26–30 numbers per cell**.

▼ 4) Split fairly

Problem

If you split cells **randomly**, the train and test cells might come from the **same study**.

That’s like letting the model “peek” at the answers → not fair.

Solution: GroupKFold

- You tell Python: *“Keep all cells from the same study together.”*
- Then in each round:
    - Train on some studies
    - Test on a completely different study

👉 This checks if the model works on **new unseen studies**, not just memorizing one.

- Use **GroupKFold (5 folds)** with `groups = adata.obs["dataset"]`.

  This means train/validation are from **different studies**, so no cheating.

- Prepare two targets:
  - `y_reg` = numeric stage (0,1,2,… in correct order) → for **MSE/R^2**
  - `y_cls` = categorical labels → for **Macro-F1**

  stratified k-folds were used to ensure that each fold maintains the same proportion of samples from each class as in the entire dataset, which is especially important for classification problems with imbalanced classes

---

▼ 5) Train two simple models

- **Regression:** `Ridge` (try `alpha` in {0.1, 1, 10}).

  Report **MSE** and **R^2** (average over folds).

- **Classification:** `LogisticRegression` with `class_weight="balanced"` (try `C` in {0.5, 1, 2}).

  Report **Macro-F1** (+ show a confusion matrix if you can).

(Why this works: PCA + biology scores + ratios track early→late; Ridge & Logistic are classical and stable.)

Models we tried: Elastic Net, Ridge regression, Random Forest
Various issues, especially with the confusion matrix.

We ultimately ended up HistGradientBoostingClassifier. It is especially good with very big data, can handle missing values automatically, and often achieves high accuracy by learning from the mistakes of earlier trees in the sequence. In gene expression analysis or other biological data, it helps scientists predict categories (such as disease types or cell stages) based on the input features by identifying subtle and complex relationships

Mistakes we made: binned labels, label leakage, unstratified kfolds

Stratified k-folds were used instead of group k-folds because the priority was to maintain balanced class proportions in every fold, which is crucial for classification tasks, especially when classes are imbalanced

```
import numpy as np, pandas as pd
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, f1_score, classif
ication_report, confusion_matrix
from sklearn.ensemble import HistGradientBoostingClassifier

X_ = X.fillna(0)

# Encode labels
le = LabelEncoder()
y_enc = le.fit_transform(y_cls)

# Stage order (ordinal codes) for expected-code smoothing
cats = list(train_adata.obs["stage"].cat.categories)  # order
ed categories set earlier
label2code = {lab:i for i,lab in enumerate(cats)}
codes_vec  = np.array([label2code[l] for l in le.classes_])
# aligned with proba columns

def round_clip(v, lo=0, hi=None):
    if hi is None: hi = len(cats)-1
    v = np.rint(v).astype(int)
    return np.clip(v, lo, hi)

def off_by_one_acc(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred) <= 1)

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=
42)

y_true_all, y_arg_all, y_exp_all = [], [], []
```

```
print("=== 5-Fold Cross-Validation ===")
for fold, (tr, va) in enumerate(skf.split(X_, y_enc), 1):
    clf = HistGradientBoostingClassifier(
        learning_rate=0.06, max_depth=None, max_bins=255, ran
dom_state=42
    )
    clf.fit(X_.iloc[tr], y_enc[tr])

    # Argmax prediction
    pred_arg = clf.predict(X_.iloc[va])

    # Expected-code prediction
    proba = clf.predict_proba(X_.iloc[va])
    exp_code = (proba * codes_vec).sum(axis=1)
    pred_codes = round_clip(exp_code)
    pred_labels = np.array([cats[c] for c in pred_codes])
    pred_exp = le.transform(pred_labels)

    y_true_all.append(y_enc[va])
    y_arg_all.append(pred_arg)
    y_exp_all.append(pred_exp)

    acc_arg = accuracy_score(y_enc[va], pred_arg)
    f1_arg  = f1_score(y_enc[va], pred_arg, average="macro")
    off1    = off_by_one_acc(y_enc[va], pred_arg)
    print(f"[Fold {fold}] Argmax Acc={acc_arg:.3f} | F1={f1_a
rg:.3f} | Off±1={off1:.3f}")

y_true = np.concatenate(y_true_all)
y_hat_arg = np.concatenate(y_arg_all)
y_hat_exp = np.concatenate(y_exp_all)

print("\n=== Overall (OOF held-out) — Argmax ===")
print(f"Accuracy:  {accuracy_score(y_true, y_hat_arg):.3f}")
print(f"Macro-F1:  {f1_score(y_true, y_hat_arg, average='macr
o'):.3f}")
print(f"Off-by-1:  {off_by_one_acc(y_true, y_hat_arg):.3f}")

print("\n=== Overall (OOF held-out) — Expected-code (rounded)
===")
print(f"Accuracy:  {accuracy_score(y_true, y_hat_exp):.3f}")
print(f"Macro-F1:  {f1_score(y_true, y_hat_exp, average='macr
o'):.3f}")

# Pretty report for argmax (main result)
print("\nClassification report (OOF, Argmax):")
print(classification_report(le.inverse_transform(y_true),
                            le.inverse_transform(y_hat_arg), d
igits=3))

print("\nConfusion matrix (OOF, Argmax):")
print(confusion_matrix(le.inverse_transform(y_true),
                       le.inverse_transform(y_hat_arg)))

# ---- Final fit on ALL data & save predictions ----
final_clf = HistGradientBoostingClassifier(
    learning_rate=0.06, max_depth=None, max_bins=255, random_
state=42
).fit(X_, y_enc)

# Argmax labels
pred_all_arg = le.inverse_transform(final_clf.predict(X_))

# Expected-code labels
proba_all = final_clf.predict_proba(X_)
exp_code_all = (proba_all * codes_vec).sum(axis=1)
round_code_all = round_clip(exp_code_all)
round_label_all = np.array([cats[c] for c in round_code_all])

# Save
out = X_.copy()
```

```
out["pred_label_argmax"]   = pred_all_arg
out["pred_label_expcode"]  = round_label_all
out["pred_stage_exp_code"] = exp_code_all
out.to_csv("final_ml_results_exact_stage_hgb_final.csv", inde
x=True)
```

▼ 6) Fit on all training data and save

- Save:
    - the **preprocessing/feature builder** you used (so you can do the **exact same steps** on new data) → `preprocess.pkl`
    - the **regression model** → `model_reg.pkl`
    - the **classification model** → `model_clf.pkl`
- Use `joblib.dump()` for saving.

(If you wrapped your cleaning+feature steps in a small sklearn Transformer/Pipeline, saving is easy. If not, save the pieces you need: HVG list, PCA loadings, gene lists you used, etc.)

▼ 7) Make the required script (predict.py)

Your script must:

1. **Read**: input `.h5ad` path and output `.csv` path (from command-line args).
2. **Load**: `preprocess.pkl`, `model_reg.pkl`, `model_clf.pkl`.
3. **Transform**: apply the **same** cleaning + feature steps to the new data.
4. **Predict**:
    - `stage_pred_continuous` = regression output
    - `stage_pred_class` = classifier output (map ints back to labels if needed)
5. **Write**: `predictions.csv` with columns:
    - `cell_id` (use `adata.obs_names`)
    - `stage_pred_continuous`

    - `stage_pred_class`

  Test it on a small subset to be sure it runs start-to-finish.

▼ 8) Zip and submit (2–5 min)

- Include:
    - `predict.py`
    - your `.pkl` files
    - your notebook or `train.py`
    - `requirements.txt`
    - a short `README.md` with “how to run” (one command).

▼ 9) Pitch outline (10 slides, super short)

1. Problem & data (2 datasets, class imbalance, batch effects)
2. Constraints (classical ML only, metrics)
3. Pipeline picture (clean → features → models)
4. Cleaning (normalize, log, ComBat, HVGs, scale)
5. Features (20 PCs + 4 scores + 2 ratios)
6. Models (Ridge, Logistic w/ balanced)
7. CV setup (GroupKFold by dataset)
8. Results (MSE/R^2, Macro-F1, confusion matrix)
9. Reproducibility (artifacts, predict.py)
10. Lessons & next steps (more gene sets, ordinal model, calibration)

These metrics define the main aspects of model performance in multi-class classification tasks:

- **Accuracy (Argmax Acc):** The proportion of samples where the model’s top-predicted class (the one with the highest probability) matches the true class label, measuring overall prediction correctness.machinelearningmastery

- **Macro-F1 (F1):** The harmonic mean of precision and recall, averaged equally across all classes. This is crucial for class-imbalanced problems, reflecting the model's ability to correctly classify each category regardless of size.machinelearningmastery
- **Off-by-1 Accuracy:** The percentage of predictions that are either exactly correct or off by only one class label, useful in ordered or semi-ordinal problems to capture nearly-correct predictions.
- **Expected-code (rounded) Acc & F1:** These aggregate "rounded" predictions based on expected value outputs, yielding accuracy and F1 metrics that might decrease if class boundaries are fuzzy or not well captured by the model's output probabilities.
- **Precision, Recall, F1-score by Class:** These break down how well individual stages (e.g., HE10W, HE12W) are predicted. Precision measures how many predicted as a class are correct; recall measures how many true members of a class are found; F1-score balances these two.

All these values in a perfect model are 1

The confusion matrix provides a detailed summary of how a classification model performs by comparing the model's predictions with the actual class labels. It breaks down the prediction results into a table, showing for each actual class how many times it was predicted as every possible class. This reveals not only the number of correct predictions (diagonal entries) but also the specific patterns and types of errors made—such as which classes are most often confused with each other.

For multi-class problems, each row represents the true labels while each column shows the predicted labels, making it easy to see errors, misclassifications, and where the model might need improvement. The confusion matrix helps to compute other key metrics like accuracy, precision, and recall, and it is especially valuable for assessing performance on class-imbalanced datasets or

identifying systematic biases in model prediction

Alternative models we were considering but ran out of time

Early Embryogenesis Prediction Tool (Nature Methods, 2024)

- **Purpose**: Specifically built for cell type and developmental stage annotation in human embryo single-cell RNA-seq.
- **Reference Integration**: Combines six major published human embryo datasets, from zygote to gastrula.
- **High Accuracy**: Outperforms SingleR, scMap, and scType; reports accuracy of 0.982 for cluster-level annotation.
- **Features**: Standardized UMAP embedding, stabilized cell type/stage annotation; easily project your query dataset.
- **Best Use**: Annotate and benchmark your predictions for stage and lineage using their stabilized reference and tool.

TL;DR

- Turn each cell into ~30 **simple numbers** (20 PCs + 4 scores + 2 ratios).
- **Fix batch**, **split by dataset**, **train Ridge & Logistic**.
- **Save** preprocess + models, **ship** `predict.py`, **report** MSE/R^2 & Macro-F1.

Questions?