**AI/ML Hackathon Survival Guide (One‑Day Prep)**

**0) Mindset & Game Plan (first 60–90 minutes)**

* **Scope fast, then iterate**: Pick the smallest slice that proves value end‑to‑end (data → baseline → metric → tiny demo). Polish later.
* **Timebox**: E.g., *90 min problem framing/EDA*, *3–4 h baseline + first demo*, *remainder iterate, eval, and UI*.
* **Assign roles**: PM/presenter, data wrangler, modeler, integrator/UI, tester. You’re the **AI/ML lead**.
* **Keep it reproducible**: Single repo, env.yml/requirements.txt, seed set, notebooks/, src/, data/.
* **Think deployment early**: CPU-only fallback, minimal dependencies, small models, quantization if needed.

**1) Problem Framing Checklist**

* **User & pain point**: Who benefits? What decision/action changes?
* **Prediction/generation task**: Classification, regression, ranking, retrieval, summarization, Q&A, CV (cls/det/seg), anomaly detection.
* **Success metric** (pick one primary): Accuracy/F1/ROC‑AUC/PR‑AUC, MAE/MAPE/RMSE, mAP/IoU, latency, hallucination rate.
* **Data availability**: Size, labels, imbalance, leakage risks, licensing/PII.
* **Constraints**:
  + Latency (e.g., <200 ms API),
  + Memory (<1 GB),
  + Offline vs cloud,
  + Hardware (CPU vs GPU),
  + Cost.
* **Baselines**:
  + Tabular → Logistic Reg / Random Forest / XGBoost.
  + Vision → Transfer learning (ResNet18/MobileNetV2).
  + Text → TF‑IDF + linear, or zero‑/few‑shot LLM, or RAG.
* **Deliverable**: Minimal UI (Streamlit) + API (FastAPI) + short demo deck.

**2) Data Hygiene & Splits (what judges listen for)**

* **Train/Val/Test** (or K‑fold). Keep test untouched until final.
* **Stratify** for classification; **grouped splits** for user/time leakage.
* **Leakage watchlist**: Timestamps after event, duplicate IDs, target leaks in features, future data in training, augmentation in both train/test.
* **Imbalance**: Metrics (F1/PR‑AUC), class weights, resampling (SMOTE cautiously), threshold tuning.
* **Preprocessing**: Missing values, outlier handling, categorical encoding, scaling numeric.

**3) Metrics Cheat‑Sheet**

* **Classification**: Accuracy (balanced data), F1 (imbalanced), ROC‑AUC (ranking ability), PR‑AUC (rare positives), confusion matrix, calibration.
* **Regression**: MAE (robust), RMSE (penalizes big errors), MAPE (business‑friendly: beware zeros), R² (explanatory power).
* **Vision**: mAP@[.5:.95] for detection, **IoU / mIoU / Dice** for segmentation.
* **NLP Gen**: Exact match, ROUGE/BLEU (summaries/translations), human eval rubric, retrieval hit‑rate for RAG, hallucination rate.

**Talking point**: *“We chose F1 over accuracy due to class imbalance and tuned the decision threshold on the validation set to maximize F1 while keeping recall ≥ 0.8.”*

**4) Quick Baselines (copy‑paste)**

**4.1 Tabular Classification (scikit‑learn pipeline)**

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report, f1\_score

# Load

df = pd.read\_csv('data/train.csv')

X = df.drop('target', axis=1)

y = df['target']

# Simple type split

cat\_cols = X.select\_dtypes(include=['object', 'category']).columns

num\_cols = X.select\_dtypes(include='number').columns

pre = ColumnTransformer([

('num', StandardScaler(with\_mean=False), num\_cols),

('cat', OneHotEncoder(handle\_unknown='ignore'), cat\_cols)

])

clf = Pipeline([

('pre', pre),

('model', LogisticRegression(max\_iter=1000, class\_weight='balanced'))

])

X\_tr, X\_va, y\_tr, y\_va = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)

clf.fit(X\_tr, y\_tr)

pred = clf.predict(X\_va)

print(classification\_report(y\_va, pred))

print('F1:', f1\_score(y\_va, pred, average='weighted'))

**Upgrade paths**: try RandomForestClassifier, XGBClassifier, or CatBoostClassifier (handles categorical nicely).

**4.2 Image Classification (PyTorch transfer learning)**

import torch, torchvision as tv

from torch import nn

from torchvision import transforms

from torch.utils.data import DataLoader

from torchvision.datasets import ImageFolder

# Data

train\_tfms = transforms.Compose([

transforms.Resize(256),

transforms.CenterCrop(224),

transforms.RandomHorizontalFlip(),

transforms.ToTensor()

])

val\_tfms = transforms.Compose([

transforms.Resize(256), transforms.CenterCrop(224), transforms.ToTensor()

])

train\_ds = ImageFolder('data/images/train', transform=train\_tfms)

val\_ds = ImageFolder('data/images/val', transform=val\_tfms)

train\_dl = DataLoader(train\_ds, batch\_size=32, shuffle=True)

val\_dl = DataLoader(val\_ds, batch\_size=64)

# Model

model = tv.models.resnet18(weights=tv.models.ResNet18\_Weights.DEFAULT)

model.fc = nn.Linear(model.fc.in\_features, len(train\_ds.classes))

# Train

opt = torch.optim.AdamW(model.parameters(), lr=1e-3)

loss\_fn = nn.CrossEntropyLoss()

device = 'cuda' if torch.cuda.is\_available() else 'cpu'

model.to(device)

for epoch in range(5):

model.train()

for xb, yb in train\_dl:

xb, yb = xb.to(device), yb.to(device)

opt.zero\_grad(); loss = loss\_fn(model(xb), yb); loss.backward(); opt.step()

# quick val

model.eval(); correct = total = 0

with torch.no\_grad():

for xb, yb in val\_dl:

xb, yb = xb.to(device), yb.to(device)

pred = model(xb).argmax(1)

correct += (pred==yb).sum().item(); total += yb.size(0)

print(f'Epoch {epoch+1}: val acc = {correct/total:.3f}')

**Vision notes**: Use small backbones (ResNet18/MobileNetV2). For tiny data, **freeze most layers** and train the head only; heavy augmentations help.

**4.3 Simple RAG (no external APIs) with TF‑IDF**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

# docs: list of strings

# queries: list of questions

vectorizer = TfidfVectorizer(stop\_words='english', ngram\_range=(1,2))

X = vectorizer.fit\_transform(docs)

def retrieve(query, topk=5):

q = vectorizer.transform([query])

sims = cosine\_similarity(q, X).ravel()

idx = np.argsort(-sims)[:topk]

return [(int(i), float(sims[i])) for i in idx]

# Answer = top passages concatenated (or pass to a small local model if allowed)

**Upgrade**: If allowed, swap TF‑IDF for sentence-transformers embeddings + FAISS; optionally call a small LLM for answer synthesis.

**5) Model Iteration Playbook (what to say you did)**

1. **EDA**: class balance, missingness, leakage scan, simple charts.
2. **Baseline**: trivial heuristic (e.g., predict majority class) → simple model.
3. **Feature work** (tabular): target‑aware encoding, text fields → TF‑IDF, date parts, counts, log transforms.
4. **Regularization & tuning**: Grid/Random search on key params; avoid overfitting.
5. **Validation**: Fixed split or K‑fold; early stopping; calibration if needed.
6. **Error analysis**: Confusion matrix slices; stratify by user/region/time.
7. **Threshold tuning**: Optimize metric on val.
8. **Packaging**: Save artifact (.pkl/ONNX), write predict() wrapper, add API/UI.

**6) LLM Options: Prompt → RAG → Fine‑tune (when to choose what)**

* **Prompt‑only**: Fastest if task is formatting/rewriting or simple Q&A with public knowledge; depends on API/latency/cost; poorer control.
* **RAG**: When domain knowledge is in your docs; updateable without retraining; controllable citations. **Default choice** for hackathons.
* **Fine‑tune**: When outputs must follow narrow style or tasks are not solvable by retrieval; costs time & data; consider LoRA/QLoRA on small models.

**Key LLM talking points**

* Tokenization, context window, temperature/top‑p, system vs user prompts, few‑shot examples, tool calling (functions), safety/guardrails.
* Hallucinations & mitigations: retrieval grounding, citations, refusal policy, constrained decoding, eval with golden answers.

**7) Deployment‑Friendly Tricks**

* **Small models**: Prefer distilled/quantized models (8‑bit/4‑bit) for CPU demos.
* **ONNX / TorchScript**: Export models for faster inference.
* **Batching & caching**: Cache repeated embeddings/inference.
* **Latency**: Pre‑load model at app start, avoid cold starts.
* **Conf limits**: Add confidence scores & fallbacks (rule‑based) to avoid risky outputs.

**8) Minimal API & UI Snippets**

**FastAPI prediction endpoint**

from fastapi import FastAPI

import joblib

import pandas as pd

app = FastAPI()

model = joblib.load('artifacts/model.pkl')

@app.post('/predict')

def predict(payload: dict):

X = pd.DataFrame([payload])

yhat = model.predict(X)[0]

return {"prediction": int(yhat)}

**Streamlit demo app**

# streamlit run app.py

import streamlit as st

import joblib, pandas as pd

st.title('Demo Predictor')

model = joblib.load('artifacts/model.pkl')

with st.form('form'):

feature\_a = st.number\_input('feature\_a')

feature\_b = st.text\_input('feature\_b')

submitted = st.form\_submit\_button('Predict')

if submitted:

X = pd.DataFrame([{"feature\_a": feature\_a, "feature\_b": feature\_b}])

st.write('Prediction:', int(model.predict(X)[0]))

**9) Common CV Tasks (sound confident)**

* **Classification**: Transfer learning; class weights/augment; metric = accuracy/F1.
* **Detection**: Try a lightweight model (YOLOv5n/v8n or SSD‑lite); metric = mAP.
* **Segmentation**: U‑Net/DeepLab; metric = IoU/Dice; handle class imbalance with focal loss.
* **Data aug**: flips, rotations, color jitter, random crops; keep val/test clean.

**Quote**: *“We froze the backbone and trained the classification head to avoid overfitting, then un‑froze top layers for fine‑tuning once the head stabilized.”*

**10) NLP Tasks (quick hits)**

* **Text cls**: TF‑IDF + LogisticReg baseline; upgrade to small transformer.
* **NER**: spaCy baseline; upgrade to fine‑tuned transformer.
* **Summarization/Q&A**: RAG with chunking (e.g., 512–1k tokens), overlap 10–20%, cosine similarity; evaluate with exact match/ROUGE + manual spot checks.
* **Toxicity/PII filters**: rule‑based regex + classifier.

**11) Evaluation & Error Analysis Phrases**

* “We used **stratified K‑fold** to ensure class balance across folds.”
* “**Confidence‑threshold** tuned on validation to meet precision ≥ X at recall Y.”
* “Top **error slices** were short texts and rare classes; we added class‑weighted loss and domain synonyms to improve recall.”

**12) Reproducibility & MLOps Lite**

* **Seeds**: numpy, torch, and dataloader settings for determinism.
* **Tracking**: Simple CSV log or MLflow; record dataset hash, code commit, params, metric.
* **Artifacts**: artifacts/model.pkl, artifacts/label\_encoder.pkl.
* **Versioning**: Git branching: main (stable), dev (in‑progress), feature branches.

**13) Ethics, Safety, and IP (judges notice)**

* **Data rights**: Only use datasets you are allowed to (licenses!).
* **Privacy**: Strip PII; anonymize; don’t upload sensitive data to third‑party APIs.
* **Fairness**: Check performance across cohorts; avoid harmful automation without human‑in‑the‑loop.
* **Transparency**: Clearly state limitations and failure modes.

**14) Presentation Blueprint (5–7 minutes)**

1. **Problem & Impact** (30–45s): Who cares? What changes?
2. **Approach** (60–90s): Data, model choice, metric rationale.
3. **Demo** (2–3m): Live or recorded.
4. **Results** (60s): Before/after, baseline vs final, metric table.
5. **Engineering** (45s): Architecture, API/UI, deployment constraints.
6. **Ethics & Next steps** (30–45s): Risks, roadmap.

**One‑slide architecture**: Data ingestion → Preprocess → Model/RAG → API → UI. Include boxes for caching, logging, metrics.

**15) Handy Templates**

**Problem Framing Canvas**

* Users & pain points:
* Task type & constraints:
* Primary metric (+ target threshold):
* Data sources & licenses:
* Baseline & upgrade plan:
* Demo plan (UI/API):

**Experiment Log**

| **Run** | **Commit** | **Data ver** | **Params** | **Metric** | **Notes** |
| --- | --- | --- | --- | --- | --- |

**README.md Skeleton**

# Project Name

## Quickstart

- python -m venv .venv && source .venv/bin/activate

- pip install -r requirements.txt

- streamlit run app.py

## Data

- Put raw files in data/raw, processed in data/processed.

## Train

- python src/train.py

## Serve

- uvicorn api:app --port 8000

## Demo

- Streamlit UI at /app.py

## Notes

- Metric: F1 (val) with stratified split; seed=42.

**16) Troubleshooting Checklist**

* Metric not improving → check leakage, wrong target, bad split, data types.
* Overfitting → more regularization, dropout, data aug, early stopping, smaller model.
* Underfitting → increase capacity, better features, longer training.
* Unstable results → fix seed, batch sizes, learning rate; ensure deterministic ops.
* Slow inference → smaller/quantized model, batch requests, cache embeddings.

**17) Packing List (practical!)**

* Cables, extension board, hotspot, power bank.
* Shared repo access, API keys (if permitted), dummy data for offline demos.
* Recorded demo as fallback, screenshots.

**18) Quick “Confident Sounding” Glossary**

* **Bias–variance trade‑off**: Complexity vs generalization.
* **Regularization**: L1/L2/Dropout to prevent overfitting.
* **Cross‑validation**: Reliable estimate of out‑of‑sample performance.
* **Calibration**: Making predicted probabilities match reality.
* **Vector DB/FAISS**: Fast nearest‑neighbor search for retrieval.
* **Quantization**: Reduce precision (e.g., 8‑bit) to speed inference.
* **IoU/Dice**: Overlap metrics for segmentation quality.
* **mAP**: Mean Average Precision for detection; area under precision–recall curve across IoU thresholds.

**19) Minimal requirements.txt (safe defaults)**

pandas

numpy

scikit-learn

fastapi

uvicorn

streamlit

torch

torchvision

joblib

Add xgboost, catboost, or sentence-transformers only if needed.

**Final Tip**

If you can **show a working demo + a clear metric gain over a trivial baseline**, speak to **impact** and **constraints**, and handle **ethics & failure modes**, you’ll look like you’ve done this before. Good luck — you’ve got this! 💪