Great — below is a **single, exhaustive, step-by-step implementation plan** (no code executed here) that takes you from **data generation** all the way to **final JSON output**. Each step contains minute, actionable details, expected shapes/types, formulas, and recommended hyperparameters. I’ve also emphasized **test score** as an objective signal and showed where to weight it or let the model learn its importance.

# Step-by-step implementation (data → model → output)

## Step 0 — Project setup (one-time)

* Create a project folder and a Jupyter notebook: resume\_classifier.ipynb.
* Create subfolders: data/, models/, notebooks/, src/.
* Python packages (suggest): numpy, pandas, scikit-learn, tensorflow (or pytorch), nltk/spacy, sentence-transformers (optional), shap, lime, matplotlib.
* Set random seeds for reproducibility:
  + seed = 42 (use in numpy, tf, random).

## Step 1 — Generate / Collect dataset

**Goal:** Build a dataset of JSON resume-like records plus domain requirement files and labels.

### 1.1 Data schema (each resume)

{

"skills": [...strings...],

"projects": [...strings...],

"work\_experience": [{"title":"Data Scientist","years":2}, ...],

"test\_score": 88,

"preferred\_domain": "Data Science",

"id": "candidate\_001"

}

### 1.2 Domain requirement file (per domain)

Example domain\_requirements/data\_science.json:

{

"domain": "Data Science",

"required\_skills": ["Python","Pandas","NumPy","Scikit-learn","PyTorch","Docker","Deep Learning"]

}

### 1.3 Synthetic data rules (if you lack many resumes)

* Generate N samples (start N=2000) varying skills/project titles/years/test\_scores.
* Test scores sample from realistic distribution (e.g., normal centered 65, std 20, clipped to 0–100).
* Ensure label diversity (Fit/Partial/Not Fit) by construction.

## Step 2 — Create ground truth labels (initially rule-based)

**Why:** You need labels to train supervised models. You can later replace with human labels.

**Baseline rule set (you can tune):**

* Compute skill\_match\_ratio = matched\_skills / total\_required\_skills.
* Compute normalized test\_score\_norm = test\_score / 100.

Labeling rules (example):

* Fit if (skill\_match\_ratio >= 0.70) AND (test\_score\_norm >= 0.75) AND (project\_count >= 1).
* Partial Fit if (0.40 <= skill\_match\_ratio < 0.70) OR (0.50 <= test\_score\_norm < 0.75).
* Not Fit if (skill\_match\_ratio < 0.40) OR (test\_score\_norm < 0.50).

**Note about arithmetic** (example):

* If matched\_skills = 8 and total\_required = 20 then  
  skill\_match\_ratio = 8 ÷ 20 = 0.4 (i.e., 8/20 = 0.4).
* If test\_score = 88 then  
  test\_score\_norm = 88 ÷ 100 = 0.88.

Store labels in dataset as "label": "Partial Fit".

## Step 3 — Preprocessing & helper functions

Create small modular functions.

### 3.1 Build skill vocabulary

* From all resumes and domain lists, build skill\_vocab = sorted(unique\_skills).
* skill\_vocab\_size = len(skill\_vocab).

### 3.2 Skill encoding function

* Input: candidate skills list.
* Output: binary vector of length skill\_vocab\_size where position i is 1 if skill present.

### 3.3 Matched & missing skills (per domain)

* Input: candidate skills, domain required skills.
* matched\_skills = intersection(candidate\_skills, required\_skills) (list).
* missing\_skills = required\_skills - candidate\_skills (list).
* skill\_match\_ratio = len(matched\_skills) / len(required\_skills).

### 3.4 Project & experience features

* project\_count = len(projects).
* Optional: project\_title\_embeddings — encode using sentence-transformers or tokenize + embedding.
* years\_experience = sum(item['years'] for item in work\_experience) or use max years or weighted sum based on titles.

### 3.5 Test score normalization

* test\_score\_norm = test\_score / 100 (float in [0,1]).  
  Example: 88 → 88 ÷ 100 = 0.88.

### 3.6 Numeric feature scaling

* For numeric fields (years\_experience, project\_count), use StandardScaler or MinMaxScaler on training data.
* Save scaler objects.

## Step 4 — Final feature vector (what you feed to model)

Two parallel branches:

1. **Skills branch**
   * skill\_vector (binary, length = V).
   * Add scalar skill\_match\_ratio as an extra numeric feature (or let model compute from skill\_vector, but adding is helpful).
2. **Numeric branch**
   * [test\_score\_norm, project\_count\_scaled, years\_experience\_scaled, skill\_match\_ratio] — a dense numeric vector.
3. **Optional text branch**
   * Project titles (or experience titles) encoded by RNN/CNN or by sentence embeddings (e.g., SBERT) → vector.
4. **Concatenate** all branches into one final vector for dense processing.

**Example final shapes (toy):**

* skill\_vector: (V,) where V = 300.
* numeric\_vector: (k,) where k = 4.
* project\_embedding: (d,) where d = 512 (if using SBERT).
* final\_vector: (V + k + d,).

## Step 5 — Model architecture (practical, detailed)

You can choose Keras (TensorFlow) or PyTorch. I’ll outline a Keras-style hybrid that works well:

### 5.1 Inputs

* skill\_input shape = (V,) (binary).
* numeric\_input shape = (k,) (float).
* project\_input shape = (d,) (optional).

### 5.2 Skills branch (dense)

* x1 = Dense(256, activation='relu')(skill\_input)
* x1 = Dropout(0.3)(x1)
* x1 = Dense(128, activation='relu')(x1)

### 5.3 Numeric branch (dense)

* x2 = Dense(32, activation='relu')(numeric\_input)
* x2 = Dense(16, activation='relu')(x2)

### 5.4 Project/text branch (if using embeddings)

* x3 = Dense(128, activation='relu')(project\_input)
* x3 = Dense(64, activation='relu')(x3)

### 5.5 Concatenate

* concat = concatenate([x1, x2, x3]) (exclude x3 if not used)
* h = Dense(128, activation='relu')(concat)
* h = Dropout(0.3)(h)
* h = Dense(64, activation='relu')(h)

### 5.6 Output

* out = Dense(3, activation='softmax')(h) for 3 classes.

### 5.7 Compile

* loss = categorical\_crossentropy
* optimizer = Adam(lr=1e-3)
* metrics = ['accuracy'] plus compute F1 during evaluation using sklearn.

**Notes / alternatives**

* For text sequences: use Embedding + Conv1D or LSTM(64) then flatten and connect.
* If V is huge (>1000), consider learning a skill\_embedding (treat skill list as tokens) and use pooling.

## Step 6 — Training procedure (exact steps & hyperparams)

1. **Train/val/test split**: 70/15/15 stratified by label.
2. **Batch size**: 32 or 64.
3. **Epochs**: up to 50 with callbacks.
4. **Callbacks**:
   * EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)
   * ModelCheckpoint(save\_best\_only=True)
5. **Class weights**: compute class\_weight = {cls: total\_samples / (num\_classes \* class\_count[cls])} (sklearn provides utility).
6. **Fit**: model.fit(X\_train, y\_train, validation\_data=(X\_val, y\_val), class\_weight=class\_weight, callbacks=..., epochs=...).

## Step 7 — Metrics & evaluation (how to measure success)

1. **Primary metrics**:
   * Accuracy, Precision, Recall, F1 (macro & weighted).
2. **Confusion matrix** for error analysis.
3. **Per-class precision/recall** to ensure Fit is not sacrificed.
4. **Calibration**:
   * Plot predicted probability histograms.
   * If calibration needed, use sklearn.calibration.CalibratedClassifierCV or temperature scaling (for NN).
5. **Threshold checks**:
   * Default: choose class = argmax(softmax\_probs).
   * You might define "uncertain" if max\_prob < 0.55 and then force human review or label as Partial Fit.

## Step 8 — Interpretability & explanation (detailed)

You must return a human readable explanation plus matched/missing skills.

### 8.1 Matched & missing skills

* Already computed in Step 3: return both lists.

### 8.2 Feature contributions (explain why)

* **Rule-based template** (fast and reliable):
  + Include:
    - raw test\_score and test\_score\_norm
    - skill\_match\_ratio and len(matched\_skills)/len(required\_skills)
    - project\_count and years\_experience
    - mention top-missing-skills (first 3)
  + Example template:

"High test score (88) and covers 8/12 required skills, but lacks PyTorch, Docker. Projects: 3; Experience: 1 year. Model confidence: 0.82 → Partial Fit."

### 8.3 LIME / SHAP (advanced)

* Use **SHAP DeepExplainer** or **KernelExplainer** to get feature attributions per prediction.
* Steps:
  + Use a small subset of training data as background.
  + Run SHAP on the model for a given candidate → get feature importance list (e.g., test\_score contributed +0.12 to Fit).
  + Convert important features into bullet points in explanation: “Test score strongly favors Fit; missing PyTorch reduces score.”

### 8.4 Sensitivity test (simple but effective)

* Perturb test\_score by ±10% and observe label/confidence change.
* If small changes flip class => flag as **borderline** in explanation.

## Step 9 — Postprocessing: building final JSON output

**Final output should match your spec**:

Example pipeline to produce JSON:

1. Run class\_probs = model.predict(final\_vector).
2. pred\_idx = argmax(class\_probs); label = classes[pred\_idx].
3. confidence = float(class\_probs[pred\_idx]) (keep 2–3 decimals).
4. matched\_skills, missing\_skills from Step 3.
5. feature\_summary = { "skill\_match\_ratio": skill\_match\_ratio, "years\_experience": years\_experience, "test\_score": test\_score\_norm, "project\_count": project\_count } — keep normalized test\_score or include both raw & norm if desired.
6. explanation = construct from template + optionally append top SHAP features.

**Example numeric formatting / arithmetic**:

* If matched\_skills = 8 and required = 20, then  
  skill\_match\_ratio = 8 ÷ 20 = 0.4.  
  Put 0.4 or format as 0.40 based on preference.

**Full JSON**

{

"label": "Partial Fit",

"confidence": 0.82,

"matched\_skills": ["Python", "Pandas", ...],

"missing\_skills": ["PyTorch","Docker"],

"feature\_summary": {"skill\_match\_ratio": 0.4, "years\_experience": 1, "test\_score": 0.88, "project\_count": 3},

"explanation": "High test score (88/100) and solid foundational skills (8/20 matched). Missing PyTorch & Docker and only 1 year experience -> Partial Fit. Recommend adding a Deep Learning project with PyTorch and containerize it."

}

## Step 10 — Save model & artifacts

* Save model weights and architecture (SavedModel or model.save()).
* Save scalers, skill\_vocab, label\_encoder, and any tokenizer or embedding models (pickle or joblib).
* Save explanation templates and SHAP background dataset.

## Step 11 — Deployment (simple API)

* Create a small FastAPI or Flask app with endpoint:
  + POST /classify accepts resume JSON → returns classification JSON.
* API steps for each request:
  + Validate JSON fields and preferred domain.
  + Use skill\_vocab to encode skills.
  + Compute numeric features and scale them.
  + Build final\_vector and call model.predict.
  + Compute matched/missing list.
  + Generate explanation and return JSON.

**Security**: Validate inputs and limit request sizes.

## Step 12 — Tests, monitoring & iterative improvements

* **Unit tests** for: encoding functions, matched/missing, explanation generator.
* **Integration tests**: sample resumes → expected output shape & plausible values.
* **Monitoring** in production:
  + Track model drift (distribution of test\_scores, skill\_match\_ratio).
  + Log low-confidence predictions for human review.
* **Iterate**:
  + Collect labels from human reviews to re-train model.
  + Try alternative model families: tree-based (XGBoost) with SHAP for easier explainability or transformer-based models for richer text features.

## Quick recommended hyperparameters & checklist

* Skill branch dense sizes: 256 → 128
* Numeric branch: 32 → 16
* Final Dense: 128 → 64
* Dropout: 0.3
* Optimizer: Adam lr = 1e-3 (reduce to 1e-4 if unstable)
* Batch size: 32
* Epochs: 50 with EarlyStopping(patience=5)
* Use class\_weight computed from class frequencies

## Example “what to run in notebook” workflow (section titles)

1. Setup & imports
2. Load data & domain files
3. Build vocab & label dataset
4. Feature engineering functions (implement and test)
5. Convert all records → feature matrices (X\_skills, X\_numeric, X\_projects) and y
6. Train/val/test split (stratified)
7. Build model and train (with checkpoints)
8. Evaluate & visualize metrics
9. Implement classify\_resume(input\_json) and test on held-out examples
10. Add SHAP/LIME explanations and finalize explanation templates
11. Save artifacts & demo REST API (optional cell)

## Final notes & practical tips

* **Test score is important**: include it as a scalar numeric feature—either let the model learn its weight or create a composite score (model is preferred).
* **Don’t overtrust skill lists** — that’s why test\_score\_norm and project evidence are key. Use perturbation tests and SHAP to check how much the model depends on test\_score vs skill\_vector.
* **Start simple**: begin with skill binary + numeric features + MLP. Add text/RNN/CNN and SHAP later.
* **Collect human labels** over time for better supervised performance.

If you want, I can now:

* (A) Generate a **notebook outline with markdown + code cell placeholders** (complete skeleton).
* (B) Produce the **data preprocessing & feature engineering code** next (ready-to-run).
* (C) Produce the **Keras model code** and training cell.

Pick A, B, or C and I’ll produce the requested code/outline next.

Perfect catch 🔥 — you pointed out **two very important pipeline steps I skipped**:

1. **Data cleaning after generation**
2. **Balancing labels (normalization if one class dominates, e.g., Fit >> others)**

Let’s go **deep into just those 2 steps** with practical details, math, and workflow.

# 🧹 Step A — Data Cleaning after Generation (Deep Dive with Fuzzy/Embedding Matching)

### A.1 Skill normalization (improved)

**Problem:**  
Raw skills can be written differently but mean the same (e.g., "Python", "python3", "py", "PYTHON", "pytorch deep learning", "DL"). If we treat them as unique tokens, the model sees "python3" ≠ "python". That fragments the feature space.

**Solution:**  
Instead of a static alias map:

1. **Lowercasing + stripping** basic text normalization.  
   "Python3 " → "python3".
2. **Embedding-based similarity check** (using models like sentence-transformers/all-MiniLM-L6-v2):
   * Create a **canonical skill vocabulary** (curated or extracted from domain requirements). Example: ["python", "pandas", "numpy", "tensorflow", "docker", ...].
   * For each candidate’s skill, compute embedding vector.
   * Find cosine similarity to vocabulary embeddings.
   * If similarity ≥ 0.8 → map to that canonical skill.
   * Else → keep as-is (rare/novel skill).

**Math:**  
Cosine similarity between skill vector uu and vocab vector vv:

sim(u,v)=u⋅v∥u∥∥v∥\text{sim}(u, v) = \frac{u \cdot v}{\|u\| \|v\|}

If sim(u,v)≥0.8\text{sim}(u,v) ≥ 0.8, treat as the same skill.

**Example:**

* + "pyTorch DL" → embedding matches closest to "pytorch" with similarity 0.87 → map → "pytorch".
  + "sql database mgmt" → similarity with "sql" = 0.91 → map → "sql".

1. **Fuzzy string matching fallback** (Levenshtein distance):
   * Use fuzzy matching only if embedding similarity < 0.8.
   * Example: "javasript" → fuzzy ratio with "javascript" = 92% → map.
2. **Remove duplicates** post-mapping.

✅ Result: Skills list standardized without manual alias maps.

### A.2 Project title cleaning

1. Lowercase, strip punctuation.
2. Remove stopwords: "and", "the", "project", "using".
   * "Image Classification using CNN Project" → "image classification cnn".
3. Keep **important tokens** (NLP, CNN, RNN, GAN, LSTM).
4. Drop empty/too-short titles (length < 3 tokens).

✅ Example:  
"NLP Project 2" → "nlp".

### A.3 Work experience cleaning

1. Normalize job titles using **embedding similarity** to a canonical set:
   * Canonical: ["data scientist", "data analyst", "machine learning engineer", "intern", ...].
   * "Sr. Data Scientist" → embedding similarity with "data scientist" = 0.94 → map.
2. Ensure years is numeric:
   * "2.5 yrs" → 2.5.
   * "N/A" → 0.
   * Negative → clamp to 0.

✅ Example:  
{"title": "Sr. Data Analyst", "years": "3 yrs"} → {"title": "data analyst", "years": 3}

### A.4 Test score cleaning

1. Clamp to [0,100].
   * If 120 → 100.
   * If -5 → 0.
2. Convert consistently to float (normalized):

normalized\_score=score100\text{normalized\\_score} = \frac{\text{score}}{100}

✅ Example:  
88 → 0.88

### A.5 Remove corrupted records

Drop record if:

1. **Empty core fields**: no skills **and** no projects **and** no test score.
2. **Missing domain**: "preferred\_domain" = "".
3. **Duplicates**: same skills, projects, and test\_score across resumes.

✅ Example:  
If resume = {skills: [], projects: [], test\_score: null} → DROP.

### 🚀 Final Output after Cleaning:

* All skills standardized semantically (not just by alias).
* Job titles mapped to a controlled set.
* Test scores in clean [0,1] range.
* No corrupted or duplicate entries.
* Dataset consistent, ready for **feature engineering**.

# ⚖️ Step B — Label Balancing (Class Normalization)

**Why?** If most generated resumes fall into Fit, the model will **overfit** and predict Fit for everyone. We need to balance classes.

## ****B.1 Analyze label distribution****

Example:

| **Label** | **Count** |
| --- | --- |
| Fit | 1000 |
| Partial Fit | 300 |
| Not Fit | 200 |

Clearly **imbalanced**.

## ****B.2 Balancing strategies****

### Option 1: **Undersampling majority**

* Randomly reduce Fit samples down to match the smallest class.
* Problem: Lose valuable data (not good if dataset is small).

### Option 2: **Oversampling minority (preferred for resumes)**

* Duplicate or synthesize new samples of Partial Fit and Not Fit until they match Fit.
* Example with above distribution:
  + Target count = 1000 (max class).
  + Duplicate Partial Fit from 300 → 1000.
  + Duplicate Not Fit from 200 → 1000.

Now all classes = 1000 each.

### Option 3: **Weighted loss (model-level balancing)**

* Instead of changing dataset size, adjust model loss.
* Formula for class weight:

class\_weight[c]=Nnc×K\text{class\\_weight}[c] = \frac{N}{n\_c \times K}

where:

* + NN = total samples
  + ncn\_c = number of samples in class cc
  + KK = number of classes

Example:

* Total N=1500N = 1500, classes = 3
* nfit=1000n\_{fit} = 1000, npartial=300n\_{partial} = 300, nnot=200n\_{not} = 200

class\_weight["Fit"]=1500/(1000×3)=0.5class\\_weight["Fit"] = 1500 / (1000 × 3) = 0.5 class\_weight["Partial"]=1500/(300×3)=1.67class\\_weight["Partial"] = 1500 / (300 × 3) = 1.67 class\_weight["NotFit"]=1500/(200×3)=2.5class\\_weight["Not Fit"] = 1500 / (200 × 3) = 2.5

Feed these weights into model training (model.fit(..., class\_weight=class\_weight)).

### Option 4: **Hybrid (best practice)**

1. Light oversampling of minority classes.
2. Add class weights in training.

This ensures dataset isn’t too skewed **and** loss penalizes imbalance.

✅ **Result after balancing**:

* Training dataset has ~equal representation of all 3 labels.
* Model won’t always predict Fit.
* Evaluation metrics (F1 per class) become more reliable.

# 🔑 Summary of the 2 Added Steps

1. **Data Cleaning**
   * Normalize skills (python3 → python).
   * Standardize projects, experience, and job titles.
   * Clamp test scores to 0–100.
   * Drop invalid or duplicate resumes.
2. **Label Balancing**
   * Detect imbalance.
   * Apply oversampling/undersampling.
   * Or use **class weights** in loss function.
   * (Best: hybrid of oversampling + class weights).