How To Guide

1. Introduction: The Hybrid Classification Pipeline

This guide details the technical implementation of our HS Code Predictor, a hybrid system combining machine learning and rule-based logic to classify shipping products. The core challenge was integrating highly dimensional text data (product descriptions) with structured metadata (price, weight, country) to predict one of many possible HS tariff codes.

This document walks through the entire process, including advanced data preparation techniques like feature binning and the modular design that allows for future deep learning upgrades.

2. Data Cleaning & Preprocessing

The foundation of our system is the transformation of messy raw data (samples_clean.csv, samples_noisy.csv) into a single, unified feature vector using src/features.py.

2.1 Handling Missing Values and Text Normalization

We prioritize data imputation over deletion. Missing numeric values are filled with 0.0, and text fields are filled with empty strings. Crucially, we normalize comma-separated tags to ensure consistency.

Code Snippet: Basic Cleaning Logic (src/features.py)

```python
def _clean_basic(df: pd.DataFrame) -> pd.DataFrame:
    req = ["id", "description", "tags", "price", "weight",
           "origin", "dest", "gift", "label_id"]
    missing = [c for c in req if c not in df.columns]
    if missing:
        raise ValueError(f"Missing required columns: {missing}")

    df["price"] = pd.to_numeric(df["price"], errors="coerce").fillna(0.0)
    df["weight"] = pd.to_numeric(df["weight"], errors="coerce").fillna(0.0)

    # Fill text-like fields
    for c in ["tags", "origin", "dest", "description"]:
        df[c] = df[c].fillna("").astype(str)

    df["gift"] = pd.to_numeric(df["gift"], errors="coerce").fillna(0).astype(int)
    df["label_id"] = pd.to_numeric(df["label_id"], errors="coerce").astype(int)

    # Normalise tags: "a, b , ,c" -> "a,b,c"
    df["tags"] = (
        df["tags"].astype(str).str.split(",")
        .apply(lambda toks: ",".join([t.strip() for t in toks if t.strip()]))
    )
    return df
```

2.2 Numerical Feature Binning (Non-Trivial Technique)

Instead of relying solely on Z-score normalization for price and weight, which can be sensitive to outliers, we converted continuous numerical data into ordered categorical bins. This helps the downstream KNN and ID3 models treat specific price/weight ranges as distinct features.

Bin Definitions (src/features.py):

```python
def bin_price(x: pd.Series) -> pd.Series:
    bins = [0, 25, 50, 100, 200, 10**9]
    labels = ["0-25", "25-50", "50-100", "100-200", "200+"]
    return pd.cut(x, bins=bins, labels=labels, right=True, include_lowest=True)


def bin_weight(x: pd.Series) -> pd.Series:
    bins = [0, 0.25, 0.5, 1, 2, 10**9]
    labels = ["0-0.25", "0.25-0.5", "0.5-1", "1-2", "2+"]
    return pd.cut(x, bins=bins, labels=labels, right=True, include_lowest=True)
```

These binned features (price_bin and weight_bin) are then one-hot encoded, increasing robustness.

3. Feature Engineering & Vectorization Pipeline

All features are merged into a single, sparse feature matrix using sklearn.compose.ColumnTransformer. This ensures that disparate data types are transformed correctly before modeling.

Code Snippet: ColumnTransformer Pipeline Construction

```python
    pre = ColumnTransformer(
        transformers=[
            ("tags", tag_vec, "tags"),
            ("desc", desc_vec, "description"),
            ("price_bin", cat_enc, ["price_bin"]),
            ("weight_bin", cat_enc, ["weight_bin"]),
            ("odg", cat_enc, ["origin", "dest", "gift"]),
        ],
        remainder="drop",
        sparse_threshold=1.0,
    )
```

4. Model Training and Selection (KNN)

We found that the K-Nearest Neighbors (KNN) classifier with Cosine Distance performed best on our data. This choice is critical because our feature vectors are high-dimensional and sparse due to the TF-IDF components. Cosine distance measures similarity by angle, ignoring the magnitude difference common in sparse text data.

Training Command-Line Workflow (CLI for Reproducibility):

To train the optimal model, we use the modular pipeline script:

python -m src.features

python -m src.pipeline --dataset clean --model knn --k 3 --metric cosine

This sequence saves the final model as artifacts/models/clean_knn_k3_cosine.joblib.

5. Evaluation and Error Analysis

5.1 Key Metrics

Since minor classification errors are common in a hierarchical system like HS codes, we primarily use Top-3 Accuracy alongside Macro-F1 Score. Top-3 accuracy ensures that if the correct label is among the model's top three most likely predictions, it is considered a useful output for a human operator.

5.2 Confusion Matrix Visualization

Evaluation also includes generating confusion matrices (confusion_*.png) to visually inspect which classes are frequently confused. This informed our decision to use the dedicated src/rules.py module for specific, often-misclassified items.

6. Integrating Business Rules (Non-Trivial Technique)

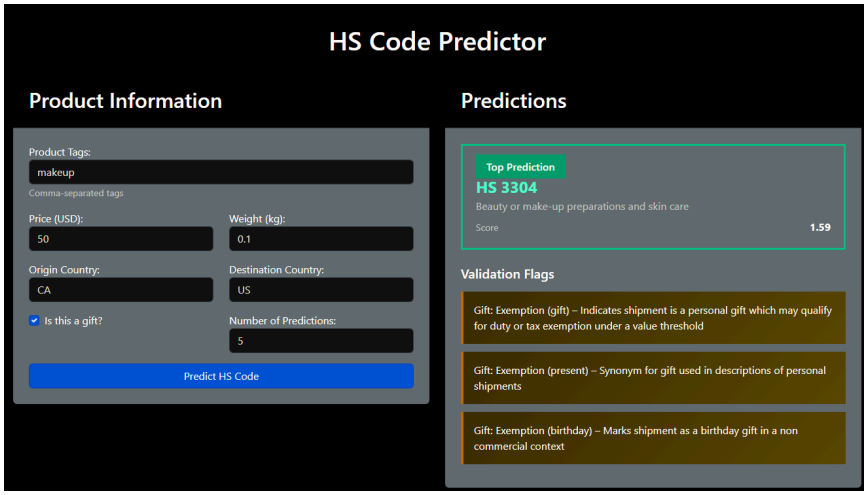
The system incorporates a Rule Engine (src/rules.py) which acts as a hybrid layer after the ML prediction.

Function: This engine checks the product's attributes (especially the 'Gift' flag and keywords in the description) against predefined rules loaded from data/rules_gift.csv and data/rules_restricted.csv.

Example: If the user checks "Is this a gift?" and the item is expensive, the rule engine generates a Validation Flag warning the operator about potential duty exemptions or restrictions, regardless of the ML model's HS code prediction.

7. Deployment: The Web Interface

The web application (built using Django) serves as the user interface for real-time inference.



Web Integration Logic (web/views.py):

1. Loads the pre-fitted preprocessor.joblib and model.joblib.
2. Maps users form data into a Pandas DataFrame.
3. Calls preprocessor.transform() to vectorize the input.
4. Calls model.predict_proba() to get confidence scores.
5. Passes the top predictions and the raw user input to the src/rules.py engine.
6. Renders the results, including prediction cards and any validation flags.

Sample Django Template Structure for Predictions:

This HTML template structure demonstrates how the results, including the top prediction badge and confidence score, are dynamically displayed:

```
{% for pred in results.predictions %}
<div class="prediction-item {% if forloop.first %}top{% endif %}">
  {% if forloop.first %}
  <span class="badge-top">Top Prediction</span>
  {% endif %}

  <!-- Heading: HS code (fallback to label_id if needed) -->
  <div class="hs-code">
    {% if pred.hs_code %}
      HS {{ pred.hs_code }}
    {% else %}
      Label {{ pred.label_id }}
    {% endif %}
  </div>

  <!-- Description: show clean title, fallback to full text -->
  {% if pred.title %}
    <div class="text-muted mb-2">{{ pred.title }}</div>
  {% elif pred.full_text %}
    <div class="text-muted mb-2">{{ pred.full_text }}</div>
  {% endif %}

  <div class="d-flex justify-content-between align-items-center">
    <span class="small text-muted">Score</span>
    <span class="fw-bold">
      {{ pred.confidence|floatformat:2 }}
    </span>
  </div>
</div>
{% endfor %}
```

8. Troubleshooting

Missing model

- Cause: The trained model file (clean_knn_k3_cosine.joblib) is missing from the artifacts/models/ directory, preventing the web app or pipeline from loading the predictor.
- Fix: Re-run the feature generation and pipeline training steps to recreate the model artifact:

python -m src.features

python -m src.pipeline --dataset clean --model knn --k 3 --metric cosine

Missing preprocessor

- Cause: The feature transformation file (preprocessor.joblib) is missing from the artifacts/ directory. This artifact is essential for transforming new data inputs into the exact feature vector shape the model was trained on.
- Fix: Re-run the feature generation script, which is responsible for fitting and saving the preprocessor:

python -m src.features

9. Licensing