Wine Quality Classification Library - Product Requirements Document

Project Overview

Project Name: WinePredict
Team Size: 3 members
Timeline: December 12-16, 2024 (Submission: Dec 16 11:59pm)
Presentation: December 17, 2024, 10:00-13:00
Repository: GitHub (invite: @Icedgarr)

Objective

Build a scalable, well-architected Python library for wine quality classification that handles class imbalance effectively and demonstrates best practices in software engineering and machine learning pipelines.

Dataset

Source: Kaggle Wine Quality Dataset
Type: Portuguese "Vinho Verde" red wine variants
Features: 11 physicochemical properties
Target: Wine quality (originally 0-10 scale)
Key Challenges:

- Small sample size (~1,600 samples)
- Severe class imbalance (most wines are "average" quality)

Classification Strategy

Target Classes: 4 bins (from original 0-10 scale)

- **Poor:** 0-4
- **Average:** 5-6
- **Good:** 7-8
- **Excellent:** 9-10

Primary Metric: Recall (especially for minority classes - Poor and Excellent)
Imbalance Handling: Stratified K-Fold Cross-Validation + Class Weights

---

Technical Requirements

Core Deliverables

1. **Library Architecture**
    - Modular folder structure following Python package conventions
    - Clear separation of concerns (data, models, features, utils)
    - Extensible design for adding new components
2. **End-to-End ML Pipeline**
    - Data loading with quality binning
    - Preprocessing (multiple preprocessors)
    - Feature engineering (≥5 independent features/feature sets)
    - Stratified train/test split and cross-validation
    - Model training with class weights
    - Hyperparameter tuning
    - Recall-focused evaluation metrics
3. **API Service**
    - REST API endpoint accepting JSON input
    - Single datapoint prediction with class probabilities
    - Model serving infrastructure
4. **Testing & Documentation**
    - Unit tests for preprocessing and features
    - README with contribution guidelines
    - API documentation

Proposed Library Structure

```
winepredict/
├── README.md
├── requirements.txt
├── setup.py
├── .gitignore
```

```
├── config/
│   ├── config.yaml              # Hyperparameters & pipeline config
│   └── class_mapping.yaml       # Quality score → class bins
├── data/
│   ├── raw/                     # Git-ignored, data goes here
│   └── processed/               # Transformed datasets
├── notebooks/
│   ├── demo_pipeline.ipynb      # End-to-end demo
│   └── eda.ipynb                # Exploratory analysis (optional)
├── winepredict/
│   ├── __init__.py
│   ├── data/
│   │   ├── __init__.py
│   │   ├── loader.py            # Data loading & quality binning
│   │   └── splitter.py          # Stratified splitting utilities
│   ├── preprocessing/
│   │   ├── __init__.py
│   │   ├── base.py              # Base preprocessor class
│   │   ├── scalers.py           # StandardScaler, RobustScaler
│   │   ├── outliers.py          # Outlier detection/handling
│   │   └── transformers.py      # Log, Box-Cox transformations
│   ├── features/
│   │   ├── __init__.py
│   │   ├── base.py              # Base feature class
│   │   ├── chemical_ratios.py   # Acidity ratios, SO2 ratios
│   │   ├── interactions.py      # alcohol×acidity, pH×citric_acid
│   │   ├── domain_features.py   # Preservation score, sweetness index
│   │   ├── statistical.py       # Percentile features, z-scores
│   │   └── polynomial.py        # Polynomial feature combinations
│   ├── models/
│   │   ├── __init__.py
│   │   ├── base.py              # Base model interface
│   │   └── classifiers.py       # RandomForest, XGBoost, etc.
│   ├── evaluation/
│   │   ├── __init__.py
│   │   ├── base.py              # Base metric class
│   │   └── metrics.py           # Recall, Precision, F1, Confusion Matrix
│   ├── utils/
│   │   ├── __init__.py
│   │   └── class_weights.py     # Calculate class weights
│   └── pipeline/
│       ├── __init__.py
│       └── pipeline.py          # Main pipeline orchestration
├── api/
```

```
│   ├── __init__.py
│   ├── app.py                # FastAPI application
│   ├── schemas.py            # Pydantic models for validation
│   └── models/               # Saved models directory
└── tests/
    ├── __init__.py
    ├── test_preprocessing.py
    ├── test_features.py
    └── test_data_loader.py
```

Work Distribution Strategy

Phase 1: Foundation (Dec 12-13)

Member 1: Infrastructure & Data

- Set up GitHub repository with .gitignore, README skeleton
- Create complete folder structure
- Implement data loading module with **quality binning logic** (`data/loader.py`)

```python
def bin_quality(score):
    if score <= 4: return 'poor'
    elif score <= 6: return 'average'
    elif score <= 8: return 'good'
    else: return 'excellent'
```

- Implement stratified splitting utilities (`data/splitter.py`)
- Create `config/config.yaml` with class bins and hyperparameters
- Create `requirements.txt` with dependencies
- Add class distribution analysis to validate imbalance
- Initial commit and repository setup

Member 2: Preprocessing & Testing

- Design base preprocessor class (`preprocessing/base.py`)
- Implement 3-4 preprocessors:

    - **StandardScaler/RobustScaler** (for numerical features)
    - **Outlier handler** (IQR method or Z-score)
    - **Log/Box-Cox transformers** (for skewed distributions)
- Implement class weight calculation utility (`utils/class_weights.py`)
- Write unit tests for preprocessing (`tests/test_preprocessing.py`)
- Test with stratified splits to ensure no data leakage
- Document preprocessor interface in README

Member 3: Feature Engineering & Testing

- Design base feature class (`features/base.py`)
- Implement 5+ classification-focused feature sets:
    - **Chemical Ratios:**
        - `fixed_acidity / volatile_acidity`
        - `free_sulfur_dioxide / total_sulfur_dioxide`
    - **Interaction Features:**
        - `alcohol × pH` (preservation interaction)
        - `sulphates × alcohol` (quality indicator)
    - **Domain Features:**
        - Preservation score: `f(alcohol, sulphates, pH)`
        - Balance index: `f(acidity, sweetness, alcohol)`
    - **Statistical Features:**
        - Standardized z-scores within the dataset
        - Percentile rankings of key features
    - **Polynomial Features:**
        - Quadratic terms for alcohol, acidity, sulphates
- Write unit tests for features (`tests/test_features.py`)
- Document feature interface in README

Phase 2: Models & API (Dec 14-15)

Member 1: API Development

- Implement FastAPI application (`api/app.py`)
- Create Pydantic schemas for input validation (`api/schemas.py`)
- Create `/predict` endpoint with:
  - JSON input validation
  - Feature computation
  - Model prediction with **class probabilities**
  - Confidence scores
- Implement model loading and caching
- Add health check endpoint `/health`
- Test API with sample requests
- Document API usage in README

Member 2: Model Training & Evaluation

- Implement base model interface (`models/base.py`)
- Create classifier wrappers with **class weights built-in**:

```python
RandomForestClassifier(class_weight='balanced', ...)
XGBClassifier(scale_pos_weight=weight_ratio, ...)
```

- Implement hyperparameter tuning with **stratified CV**:
  - Use `StratifiedKFold(n_splits=5)`
  - Optimize for recall (or F1-weighted)
- Implement recall-focused evaluation metrics:
  - **Per-class recall** (especially for Poor/Excellent)
  - Weighted F1-score
  - Confusion matrix visualization
  - Classification report
- Save best model with joblib
- Document model interface and class weight approach in README

Member 3: Pipeline Orchestration

- Implement main pipeline class (`pipeline/pipeline.py`)
- Integrate full flow:
    1. Load data with quality binning
    2. Stratified train/test split
    3. Apply preprocessing (fit on train only!)
    4. Compute features
    5. Train models with class weights
    6. Hyperparameter tuning with stratified CV
    7. Evaluate on test set with recall metrics
- Create Jupyter notebook (`notebooks/demo_pipeline.ipynb`) showing:
    - EDA of class distribution
    - Full pipeline execution
    - Model performance analysis
    - Feature importance
- Ensure pipeline is configurable via `config.yaml`
- Add logging for pipeline steps

Phase 3: Integration & Documentation (Dec 15-16)

All Members (Collaborative)

- **Dec 15 Morning:** Integration testing
    - Run full pipeline end-to-end
    - Verify stratified splitting maintains class distribution
    - Confirm class weights are being applied
    - Check all unit tests pass
    - Fix integration issues
- **Dec 15 Afternoon:** Documentation
    - Complete README with:
        - Project description & classification approach
        - Installation instructions

        - Usage examples
        - **Class imbalance strategy explanation**
        - Contribution guidelines
        - API documentation
    - Add comprehensive docstrings
    - Document class binning rationale
- **Dec 16 Morning:** Polish & validation
    - Test API with edge cases (poor/excellent wines)
    - Verify model returns confidence scores
    - Add error handling
    - Check recall scores meet expectations
- **Dec 16 Evening:** Final review & submission
    - Code review each other's work
    - Ensure repository is clean
    - Verify all requirements met
    - Submit via Google Classroom
    - Invite @Icedgarr to repository

Technical Specifications

Data Processing

Quality Binning Logic:

```python
CLASS_BINS = {
    'poor': (0, 4),      # 3, 4
    'average': (5, 6),   # 5, 6 (majority class)
    'good': (7, 8),      # 7, 8
    'excellent': (9, 10) # 9, 10 (rare)
}
```

Stratified Splitting:

```python
from sklearn.model_selection import StratifiedKFold, train_test_split

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Cross-validation
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

Preprocessing Requirements

- At least 3 preprocessor types
- Each preprocessor inherits from base with `fit()` and `transform()`
- **Critical:** Fit only on training data to prevent leakage
- Suggested preprocessors:
  - `RobustScaler` (handles outliers better than StandardScaler)
  - `OutlierClipper` (clip extreme values to 5th/95th percentile)
  - `LogTransformer` (for right-skewed features like residual_sugar)

Feature Engineering Requirements

Minimum 5 feature sets with classification focus:

1. **Chemical Ratios** (`features/chemical_ratios.py`)

```python
- acidity_ratio = fixed_acidity / volatile_acidity
- sulfur_ratio = free_sulfur_dioxide / total_sulfur_dioxide
- acid_balance = (citric_acid + fixed_acidity) / volatile_acidity
```

2. **Interaction Features** (`features/interactions.py`)

```python
```

```
- alcohol_pH = alcohol * pH
- alcohol_sulphates = alcohol * sulphates
- acidity_alcohol = fixed_acidity * alcohol
```

3. **Domain-Specific Features** (`features/domain_features.py`)

```
python

- preservation_score = f(alcohol, sulphates, pH, SO2)
- sweetness_index = residual_sugar / (alcohol + 1)
- balance_score = normalized combination of acidity, sweetness, alcohol
```

4. **Statistical Features** (`features/statistical.py`)

```
python

- z-scores for key features (alcohol, acidity, sulphates)
- percentile ranks within dataset
- distance from class means
```

5. **Polynomial Features** (`features/polynomial.py`)

```
python

- alcohol², pH², sulphates²
- Select top N polynomial features by correlation with target
```

Model Requirements

Primary Models (with class weights):

```
python
```

```
# Random Forest (handles imbalance well)
RandomForestClassifier(
    n_estimators=200,
    class_weight='balanced',  # Automatically adjusts for imbalance
    random_state=42
)

# XGBoost (great for small datasets)
XGBClassifier(
    scale_pos_weight=weight_ratio,  # Calculated from class distribution
    max_depth=6,
    learning_rate=0.1,
    random_state=42
)

# Gradient Boosting (alternative)
GradientBoostingClassifier(
    n_estimators=100,
    learning_rate=0.1,
    random_state=42
)
```

Class Weight Calculation:

```
python

from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(y_train),
    y=y_train
)
```

Hyperparameter Tuning:

- Use `StratifiedKFold` for cross-validation
- Optimize for **recall** or **weighted F1**
- Use `RandomizedSearchCV` for efficiency (faster than GridSearch)

```
python
```

```
from sklearn.model_selection import RandomizedSearchCV

search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_grid,
    cv=StratifiedKFold(n_splits=5),
    scoring='recall_weighted',  # or 'f1_weighted'
    n_iter=20,
    random_state=42
)
```

Evaluation Metrics (Recall-Focused)

Primary Metrics:

```
python

from sklearn.metrics import (
    recall_score,
    precision_score,
    f1_score,
    classification_report,
    confusion_matrix
)

# Per-class recall (most important)
recall_per_class = recall_score(y_test, y_pred, average=None)

# Weighted metrics
recall_weighted = recall_score(y_test, y_pred, average='weighted')
f1_weighted = f1_score(y_test, y_pred, average='weighted')

# Full classification report
print(classification_report(y_test, y_pred,
      target_names=['poor', 'average', 'good', 'excellent']))
```

Metric Classes to Implement:

- `RecallMetric` - per-class and weighted recall
- `PrecisionMetric` - per-class and weighted precision
- `F1Metric` - per-class and weighted F1
- `ConfusionMatrixMetric` - visualization and raw matrix

- `ClassificationReportMetric` - comprehensive report

Success Criteria:

- Recall > 70% for "average" class (majority)
- Recall > 50% for "poor" and "excellent" classes (minority)
- Weighted F1 > 0.70

API Specifications

Endpoint: `POST /predict`

Request Format:

```json
{
  "fixed_acidity": 7.4,
  "volatile_acidity": 0.7,
  "citric_acid": 0.0,
  "residual_sugar": 1.9,
  "chlorides": 0.076,
  "free_sulfur_dioxide": 11.0,
  "total_sulfur_dioxide": 34.0,
  "density": 0.9978,
  "pH": 3.51,
  "sulphates": 0.56,
  "alcohol": 9.4
}
```

Response Format:

```json
```

```
{
  "predicted_class": "average",
  "confidence": 0.73,
  "probabilities": {
    "poor": 0.05,
    "average": 0.73,
    "good": 0.19,
    "excellent": 0.03
  },
  "model_version": "v1.0",
  "timestamp": "2024-12-16T10:30:00Z"
}
```


Additional Endpoint: GET /health


```
json

{
  "status": "healthy",
  "model_loaded": true,
  "model_version": "v1.0"
}
```

Best Practices Checklist

Code Quality

- [ ] DRY: No code duplication
- [ ] KISS: Simple, readable implementations
- [ ] Loose coupling: Modules are independent
- [ ] Type hints for public methods
- [ ] Comprehensive docstrings
- [ ] Consistent naming conventions (PEP 8)

Software Engineering

- [ ] Unit tests with >70% coverage for core logic
- [ ] All tests pass before merging
- [ ] Git commits are atomic and well-described
- [ ] No hardcoded paths or values

- [ ] Configuration via YAML

Machine Learning (Classification-Specific)

- [ ] Reproducible results (set random seeds)
- [ ] **Stratified** train/test split
- [ ] **Stratified K-Fold** cross-validation
- [ ] **Class weights** applied to all models
- [ ] No data leakage (fit on train only!)
- [ ] Hyperparameter tuning on validation set
- [ ] **Recall-focused evaluation**
- [ ] Class distribution analysis in EDA

Contribution Guidelines (for README)

Adding a New Preprocessor

1. Create class inheriting from `BasePreprocessor`
2. Implement `fit()` and `transform()` methods
3. Ensure `fit()` only uses training data
4. Add to `preprocessing/__init__.py`
5. Write unit tests in `tests/test_preprocessing.py`
6. Update README with usage example

Adding a New Feature

1. Create class inheriting from `BaseFeature`
2. Implement `compute()` method
3. Consider class separability when designing features
4. Add to `features/__init__.py`
5. Write unit tests in `tests/test_features.py`
6. Document feature rationale in README

Adding a New Model

1. Create wrapper class inheriting from `BaseModel`

2. Implement `fit()`, `predict()`, `predict_proba()` methods
3. **Ensure class weights are supported and enabled by default**
4. Implement `tune_hyperparameters()` with stratified CV
5. Add to `models/__init__.py`
6. Update pipeline configuration

Adding a New Metric

1. Create class inheriting from `BaseMetric`
2. Implement `calculate()` method
3. Support both per-class and averaged metrics
4. Add to `evaluation/__init__.py`
5. Update evaluation module

Presentation Structure (10-15 minutes)

Live Demo Flow (No Slides Needed)

1. Project Overview (2 min)

- Dataset: Portuguese Vinho Verde red wines
- Challenge: Predict quality with severe class imbalance
- Solution: 4-class classification (poor/average/good/excellent)
- Strategy: Stratified CV + class weights for recall optimization

2. Architecture Walkthrough (4 min)

- **VS Code:** Show folder structure
- **Key Design Decisions:**
  - Base classes for extensibility
  - Quality binning logic
  - Stratified splitting utilities
  - Class weight integration

- **How to extend:** Quick demo of adding a new feature or preprocessor

3. Pipeline Demonstration (5 min)

- **Jupyter Notebook:** Run end-to-end pipeline live
- **Show:**
  - Class distribution (visualize imbalance)
  - Preprocessing steps
  - Feature engineering output
  - Model training with class weights
  - **Evaluation results with per-class recall**
  - Confusion matrix highlighting minority class performance

4. API Demo (2 min)

- **Terminal/Postman:** Live API call
- Show JSON request/response with probabilities
- Demonstrate prediction for different quality wines

5. Team Collaboration (2 min)

- How work was divided (infrastructure/preprocessing/features)
- Git workflow and parallel development
- Contribution guidelines for new team members

Key Talking Points:

- "We handled class imbalance using class weights rather than SMOTE for simplicity and speed"
- "Stratified cross-validation ensures minority classes are represented in each fold"
- "Our recall metric prioritizes catching rare but important quality levels"
- "The architecture allows easy addition of new features, preprocessors, and models through base classes"

Dependencies

```
txt
```

```
# Core ML
pandas>=1.5.0
numpy>=1.24.0
scikit-learn>=1.2.0
xgboost>=1.7.0
imbalanced-learn>=0.10.0  # For SMOTE if needed later

# API
fastapi>=0.100.0
uvicorn>=0.23.0
pydantic>=2.0.0

# Visualization (for confusion matrix, etc.)
matplotlib>=3.7.0
seaborn>=0.12.0

# Testing
pytest>=7.4.0
pytest-cov>=4.1.0

# Utilities
pyyaml>=6.0
joblib>=1.3.0
python-multipart>=0.0.6  # For FastAPI file uploads
```

Risk Mitigation

Key Risks:

1. **Low recall on minority classes:** Mitigate with class weights and stratified CV
2. **Data leakage:** Mitigate by fitting preprocessors only on training data
3. **Integration issues:** Mitigate by daily check-ins and early integration testing
4. **API complexity:** Use FastAPI for simplicity; dedicated team member
5. **Time constraints:** Parallel work on independent modules; phase 3 buffer time

Class Imbalance Specific Risks:

- **Risk:** Model only predicts "average" class
    - **Mitigation:** Use class weights, monitor per-class recall, tune threshold if needed

- **Risk:** Overfitting on rare classes with SMOTE
    - **Mitigation:** Start with class weights (simpler), only add SMOTE if needed

Communication Protocol:

- Daily async check-in (Slack/Discord)
- GitHub Issues for task tracking
- Pull requests require 1 approval
- Tag blockers immediately

Success Criteria

Minimum Viable Product (MVP)

- ✅ Complete library structure with all base classes
- ✅ Working end-to-end pipeline with stratified CV
- ✅ Quality binning (4 classes) implemented
- ✅ 5+ feature sets for classification
- ✅ Model training with class weights
- ✅ Hyperparameter tuning with stratified CV
- ✅ Recall-focused evaluation metrics
- ✅ Unit tests for preprocessing and features
- ✅ Working API endpoint with probabilities
- ✅ README with contribution guidelines

Performance Targets

- **Weighted Recall:** > 0.70
- **Per-Class Recall:**
    - Average: > 0.70 (majority class)
    - Poor: > 0.50 (minority)
    - Good: > 0.60

    - Excellent: > 0.50 (minority)
- **Weighted F1:** > 0.70

Stretch Goals (If Time Permits)

- Feature importance analysis visualization
- ROC curves for each class (one-vs-rest)
- Probability calibration plots
- Model comparison dashboard
- Docker containerization for API
- Automated testing via GitHub Actions

Timeline Summary

Date	Phase	Key Deliverables	Owner
Dec 12-13	Foundation	Repo setup, quality binning, stratified splitting, preprocessors, features	All (parallel)
Dec 14-15	Pipeline	API, models with class weights, pipeline orchestration, notebook	All (parallel)
Dec 15-16	Integration	Testing, documentation, recall validation, polish	All (collaborative)
Dec 16 11:59pm	**Submission**	Repository via Google Classroom	-
Dec 17 10:00-13:00	**Presentation**	Live demo and Q&A	All

Quick Start Commands (For README)

```bash
```

```
# Clone repository
git clone <repo-url>
cd winepredict

# Install dependencies
pip install -r requirements.txt

# Run pipeline
jupyter notebook notebooks/demo_pipeline.ipynb

# Run tests
pytest tests/ -v --cov=winepredict

# Start API
cd api
uvicorn app:app --reload

# Make prediction
curl -X POST "http://localhost:8000/predict" \
  -H "Content-Type: application/json" \
  -d @sample_input.json
```

Dataset URL: https://www.kaggle.com/datasets/yasserh/wine-quality-dataset
Repository URL: [To be created]
Last Updated: December 12, 2024