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
```

🚀 ACCELERATED WORK STRATEGY

KEY ADVANTAGE: Reusing existing diabetes API code saves 7-9 hours of development time!

See `MIGRATION_GUIDE.md` for exact code changes needed.

Work Distribution Strategy

Phase 1: Foundation (Dec 12-13)

MEMBER 1: API Migration & Models ⚡ ACCELERATED

Day 1 Morning (Dec 12, 9am-12pm) - API MIGRATION

Task 1.0: Migrate Existing API (45 min) 🎯 HIGH PRIORITY Follow the detailed migration guide to convert diabetes API to wine API:

- Copy `api_fastapi.py` to `api/app.py`
- Update Pydantic models (WineFeatures vs PatientFeatures)
- Update FEATURES list
- Update model loading paths
- Update prediction logic for 4 classes
- Test with: `uv run uvicorn api.app:app --reload`
- Verify docs work: `http://localhost:8000/docs`

Task 1.1: Repository Setup (15 min)

- Create GitHub repository named `winepredict`

- Add .gitignore:

```
__pycache__/
*.pyc
.pytest_cache/
data/raw/
data/processed/
*.ipynb_checkpoints
api/models/*.pkl
.env
```

- Create README.md with project title and placeholder sections
- Initialize with Python .gitignore template
- Invite team members as collaborators
- Invite @Icedgarr as collaborator
- Create `dev` branch for development

Task 1.2: Folder Structure Creation (15 min)

- Create all folders from the PRD structure
- Add empty `__init__.py` files to all Python package directories
- Commit: "Initial project structure"

Task 1.3: Dependencies File (15 min)

- Create `requirements.txt` with all dependencies from PRD
- Test installation in fresh virtual environment: `pip install -r requirements.txt`
- Commit: "Add project dependencies"

Task 1.4: Configuration Files (30 min)

- Create `config/config.yaml`:

```yaml
```

```yaml
data:
  raw_path: "data/raw/winequality-red.csv"
  processed_path: "data/processed/"
  test_size: 0.2
  random_seed: 42

quality_bins:
  poor: [0, 4]
  average: [5, 6]
  good: [7, 8]
  excellent: [9, 10]

preprocessing:
  scaler_type: "robust"  # or "standard"
  outlier_method: "iqr"
  outlier_threshold: 1.5

features:
  enable_ratios: true
  enable_interactions: true
  enable_polynomial: true
  polynomial_degree: 2

model:
  cv_folds: 5
  use_class_weights: true
  random_seed: 42

hyperparameters:
  random_forest:
    n_estimators: [100, 200, 300]
    max_depth: [10, 20, 30, null]
    min_samples_split: [2, 5, 10]
  xgboost:
    n_estimators: [100, 200]
    max_depth: [3, 6, 9]
    learning_rate: [0.01, 0.1, 0.3]
```

- Commit: "Add configuration files"

Day 1 Afternoon (Dec 12, 2pm-6pm)

Task 1.5: Data Loader Implementation (2 hours)

Create `winepredict/data/loader.py`:

```python
```

```python
import pandas as pd
import yaml
from typing import Tuple, Dict
from pathlib import Path

class WineDataLoader:
    """Load and prepare wine quality dataset with quality binning."""

    def __init__(self, config_path: str = "config/config.yaml"):
        with open(config_path, 'r') as f:
            self.config = yaml.safe_load(f)
        self.quality_bins = self.config['quality_bins']

    def bin_quality(self, score: int) -> str:
        """Convert numeric quality score to categorical bins."""
        if score <= self.quality_bins['poor'][1]:
            return 'poor'
        elif score <= self.quality_bins['average'][1]:
            return 'average'
        elif score <= self.quality_bins['good'][1]:
            return 'good'
        else:
            return 'excellent'

    def load_data(self) -> Tuple[pd.DataFrame, pd.Series]:
        """Load raw data and return features and binned target."""
        # Load raw CSV
        data_path = self.config['data']['raw_path']
        df = pd.read_csv(data_path)

        # Separate features and target
        X = df.drop('quality', axis=1)
        y_numeric = df['quality']

        # Bin quality scores
        y = y_numeric.apply(self.bin_quality)

        return X, y

    def get_class_distribution(self, y: pd.Series) -> Dict[str, int]:
        """Calculate class distribution for imbalance analysis."""
        return y.value_counts().to_dict()
```

```
    def get_feature_names(self) -> list:
        """Return list of feature column names."""
        return [
            'fixed_acidity', 'volatile_acidity', 'citric_acid',
            'residual_sugar', 'chlorides', 'free_sulfur_dioxide',
            'total_sulfur_dioxide', 'density', 'pH', 'sulphates', 'alcohol'
        ]
```

- Add comprehensive docstrings
- Test manually with sample data
- Commit: "Implement data loader with quality binning"

Task 1.6: Stratified Splitting Utilities (1 hour)

Create `winepredict/data/splitter.py`:

```python
```

```
from sklearn.model_selection import train_test_split, StratifiedKFold
import pandas as pd
from typing import Tuple

class StratifiedSplitter:
    """Handle stratified train/test splits and cross-validation."""

    def __init__(self, test_size: float = 0.2, random_seed: int = 42):
        self.test_size = test_size
        self.random_seed = random_seed

    def train_test_split(
        self,
        X: pd.DataFrame,
        y: pd.Series
    ) -> Tuple[pd.DataFrame, pd.DataFrame, pd.Series, pd.Series]:
        """Perform stratified train/test split."""
        return train_test_split(
            X, y,
            test_size=self.test_size,
            stratify=y,
            random_state=self.random_seed
        )

    def get_cv_splitter(self, n_splits: int = 5) -> StratifiedKFold:
        """Return stratified K-fold cross-validator."""
        return StratifiedKFold(
            n_splits=n_splits,
            shuffle=True,
            random_state=self.random_seed
        )

    def validate_split(self, y_train: pd.Series, y_test: pd.Series) -> dict:
        """Validate that class distributions are maintained."""
        train_dist = y_train.value_counts(normalize=True).to_dict()
        test_dist = y_test.value_counts(normalize=True).to_dict()
        return {
            'train_distribution': train_dist,
            'test_distribution': test_dist
        }
```

- Test with dummy data

- Commit: "Add stratified splitting utilities"

Task 1.7: Unit Tests for Data Layer (30 min)

Create `tests/test_data_loader.py`:

```python
import pytest
import pandas as pd
from winepredict.data.loader import WineDataLoader

def test_bin_quality():
    loader = WineDataLoader()
    assert loader.bin_quality(3) == 'poor'
    assert loader.bin_quality(5) == 'average'
    assert loader.bin_quality(7) == 'good'
    assert loader.bin_quality(9) == 'excellent'

def test_load_data():
    loader = WineDataLoader()
    X, y = loader.load_data()
    assert len(X) > 0
    assert len(X) == len(y)
    assert set(y.unique()) == {'poor', 'average', 'good', 'excellent'}

def test_feature_names():
    loader = WineDataLoader()
    features = loader.get_feature_names()
    assert len(features) == 11
    assert 'alcohol' in features
```

- Run tests: `pytest tests/test_data_loader.py -v`
- Commit: "Add unit tests for data loader"

Day 2 Morning (Dec 13, 9am-12pm)

Task 1.8: Setup Script (30 min)

Create `setup.py`:

```python
```

```
from setuptools import setup, find_packages

setup(
    name="winepredict",
    version="0.1.0",
    packages=find_packages(),
    install_requires=[
        line.strip()
        for line in open('requirements.txt').readlines()
    ],
    python_requires='>=3.8',
)
```

- Test: `pip install -e .`
- Commit: "Add setup.py for package installation"

Task 1.9: README Documentation (1 hour)

Update README.md with:

- Project overview and objectives
- Installation instructions
- Quick start guide
- Data download instructions
- Team member responsibilities
- Placeholder sections for API usage and contribution guidelines

Task 1.10: Pull Request & Code Review (30 min)

- Push all work to `dev` branch
- Create PR to `main` with detailed description
- Request review from team members
- Address any feedback

MEMBER 2: Preprocessing & Model Layer

Day 1 Morning (Dec 12, 9am-1pm)

Task 2.1: Base Preprocessor Class (1 hour)

Create `winepredict/preprocessing/base.py`:

```python
```

```
from abc import ABC, abstractmethod
import pandas as pd
from typing import Any

class BasePreprocessor(ABC):
    """Abstract base class for all preprocessors."""

    def __init__(self):
        self.is_fitted = False
        self._fit_params = {}

    @abstractmethod
    def fit(self, X: pd.DataFrame, y: pd.Series = None) -> 'BasePreprocessor':
        """Fit preprocessor on training data.

        Args:
            X: Training features
            y: Training target (optional, for some preprocessors)

        Returns:
            self: Fitted preprocessor
        """
        pass

    @abstractmethod
    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """Transform data using fitted preprocessor.

        Args:
            X: Features to transform

        Returns:
            Transformed features
        """
        pass

    def fit_transform(self, X: pd.DataFrame, y: pd.Series = None) -> pd.DataFrame:
        """Fit and transform in one step."""
        return self.fit(X, y).transform(X)

    def _check_is_fitted(self):
        """Verify preprocessor has been fitted."""
        if not self.is_fitted:
```

```
            raise RuntimeError(
                f"{self.__class__.__name__} must be fitted before transform"
            )
```

- Add type hints and docstrings
- Commit: "Add base preprocessor class"

Task 2.2: Scaler Preprocessors (1.5 hours)

Create `winepredict/preprocessing/scalers.py`:

```
python
```

```
from sklearn.preprocessing import StandardScaler, RobustScaler
import pandas as pd
from .base import BasePreprocessor

class StandardScalerPreprocessor(BasePreprocessor):
    """Standardize features by removing mean and scaling to unit variance."""

    def __init__(self):
        super().__init__()
        self.scaler = StandardScaler()

    def fit(self, X: pd.DataFrame, y=None):
        """Fit scaler on training data."""
        self.scaler.fit(X)
        self.feature_names = X.columns.tolist()
        self.is_fitted = True
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """Scale features."""
        self._check_is_fitted()
        X_scaled = self.scaler.transform(X)
        return pd.DataFrame(X_scaled, columns=self.feature_names, index=X.index)


class RobustScalerPreprocessor(BasePreprocessor):
    """Scale features using statistics robust to outliers."""

    def __init__(self):
        super().__init__()
        self.scaler = RobustScaler()

    def fit(self, X: pd.DataFrame, y=None):
        """Fit robust scaler on training data."""
        self.scaler.fit(X)
        self.feature_names = X.columns.tolist()
        self.is_fitted = True
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """Scale features using robust statistics."""
        self._check_is_fitted()
```

```
        X_scaled = self.scaler.transform(X)
        return pd.DataFrame(X_scaled, columns=self.feature_names, index=X.index)
```

- Test both scalers manually
- Commit: "Implement scaler preprocessors"

Task 2.3: Outlier Handler (1 hour)

Create `winepredict/preprocessing/outliers.py`:

```python
```

```python
import pandas as pd
import numpy as np
from .base import BasePreprocessor

class OutlierClipper(BasePreprocessor):
    """Clip outliers to specified percentiles."""

    def __init__(self, lower_percentile: float = 5, upper_percentile: float = 95):
        super().__init__()
        self.lower_percentile = lower_percentile
        self.upper_percentile = upper_percentile
        self.clip_values = {}

    def fit(self, X: pd.DataFrame, y=None):
        """Calculate clip values from training data."""
        for col in X.columns:
            lower = np.percentile(X[col], self.lower_percentile)
            upper = np.percentile(X[col], self.upper_percentile)
            self.clip_values[col] = (lower, upper)

        self.is_fitted = True
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """Clip outliers to fitted percentiles."""
        self._check_is_fitted()
        X_clipped = X.copy()

        for col, (lower, upper) in self.clip_values.items():
            X_clipped[col] = X_clipped[col].clip(lower=lower, upper=upper)

        return X_clipped
```

- Commit: "Add outlier clipping preprocessor"

Day 1 Afternoon (Dec 12, 2pm-6pm)

Task 2.4: Log Transformer (1 hour)

Create `winepredict/preprocessing/transformers.py`:

```
python
```

```python
import pandas as pd
import numpy as np
from .base import BasePreprocessor

class LogTransformer(BasePreprocessor):
    """Apply log transformation to skewed features."""

    def __init__(self, features_to_transform: list = None, offset: float = 1e-6):
        super().__init__()
        self.features_to_transform = features_to_transform
        self.offset = offset  # Avoid log(0)

    def fit(self, X: pd.DataFrame, y=None):
        """Identify features to transform based on skewness."""
        if self.features_to_transform is None:
            # Auto-detect skewed features (skewness > 0.75)
            self.features_to_transform = [
                col for col in X.columns
                if X[col].skew() > 0.75
            ]

        self.is_fitted = True
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """Apply log transformation."""
        self._check_is_fitted()
        X_transformed = X.copy()

        for col in self.features_to_transform:
            X_transformed[col] = np.log(X[col] + self.offset)

        return X_transformed
```

- Commit: "Add log transformer for skewed features"

Task 2.5: Class Weight Utility (45 min)

Create `winepredict/utils/class_weights.py`:

```python
```

```python
import numpy as np
from sklearn.utils.class_weight import compute_class_weight
from typing import Dict

def calculate_class_weights(y_train: np.ndarray) -> Dict[str, float]:
    """Calculate balanced class weights for imbalanced dataset.

    Args:
        y_train: Training labels

    Returns:
        Dictionary mapping class labels to weights
    """
    classes = np.unique(y_train)
    weights = compute_class_weight(
        class_weight='balanced',
        classes=classes,
        y=y_train
    )

    return dict(zip(classes, weights))


def get_sample_weights(y_train: np.ndarray, class_weights: Dict[str, float]) -> np.ndarray:
    """Convert class weights to sample weights for training.

    Args:
        y_train: Training labels
        class_weights: Dictionary of class weights

    Returns:
        Array of sample weights
    """
    return np.array([class_weights[label] for label in y_train])
```

- Add unit tests
- Commit: "Add class weight calculation utilities"

Task 2.6: Preprocessing Unit Tests (1.5 hours)

Create `tests/test_preprocessing.py`:

python

```
import pytest
import pandas as pd
import numpy as np
from winepredict.preprocessing.scalers import (
    StandardScalerPreprocessor,
    RobustScalerPreprocessor
)
from winepredict.preprocessing.outliers import OutlierClipper
from winepredict.preprocessing.transformers import LogTransformer

# Sample data fixture
@pytest.fixture
def sample_data():
    np.random.seed(42)
    return pd.DataFrame({
        'feature1': np.random.randn(100),
        'feature2': np.random.exponential(2, 100),
        'feature3': np.random.randn(100) * 10
    })

def test_standard_scaler_fit_transform(sample_data):
    scaler = StandardScalerPreprocessor()
    X_scaled = scaler.fit_transform(sample_data)

    # Check mean ~ 0 and std ~ 1
    assert np.allclose(X_scaled.mean(), 0, atol=0.1)
    assert np.allclose(X_scaled.std(), 1, atol=0.1)

def test_robust_scaler_fit_transform(sample_data):
    scaler = RobustScalerPreprocessor()
    X_scaled = scaler.fit_transform(sample_data)

    # Check shape preserved
    assert X_scaled.shape == sample_data.shape

def test_outlier_clipper(sample_data):
    clipper = OutlierClipper(lower_percentile=10, upper_percentile=90)
    clipper.fit(sample_data)
    X_clipped = clipper.transform(sample_data)

    # Check outliers are clipped
    for col in sample_data.columns:
        lower, upper = clipper.clip_values[col]
```

```
        assert X_clipped[col].min() >= lower
        assert X_clipped[col].max() <= upper

def test_log_transformer_auto_detect(sample_data):
    transformer = LogTransformer()
    transformer.fit(sample_data)
    X_transformed = transformer.transform(sample_data)

    # feature2 is exponential (skewed), should be transformed
    assert 'feature2' in transformer.features_to_transform
    assert X_transformed['feature2'].skew() < sample_data['feature2'].skew()

def test_preprocessor_not_fitted_error(sample_data):
    scaler = StandardScalerPreprocessor()

    with pytest.raises(RuntimeError, match="must be fitted"):
        scaler.transform(sample_data)
```

- Run all tests: `pytest tests/test_preprocessing.py -v`
- Commit: "Add comprehensive preprocessing tests"

Day 2 Morning (Dec 13, 9am-12pm)

Task 2.7: Preprocessing Module Integration (30 min)

Update `winepredict/preprocessing/__init__.py`:

```python
from .base import BasePreprocessor
from .scalers import StandardScalerPreprocessor, RobustScalerPreprocessor
from .outliers import OutlierClipper
from .transformers import LogTransformer

__all__ = [
    'BasePreprocessor',
    'StandardScalerPreprocessor',
    'RobustScalerPreprocessor',
    'OutlierClipper',
    'LogTransformer',
]
```

Task 2.8: README Documentation - Preprocessors (1 hour)

Add to README.md:

```markdown
## Preprocessing

### Available Preprocessors

1. **StandardScalerPreprocessor**: Standardizes features (mean=0, std=1)
2. **RobustScalerPreprocessor**: Scales using median and IQR (robust to outliers)
3. **OutlierClipper**: Clips extreme values to percentiles
4. **LogTransformer**: Applies log transformation to skewed features

### Adding a New Preprocessor

1. Create a new 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
6. Update this README

Example:
[Include code example]
```

Task 2.9: Pull Request (30 min)

- Push preprocessing work to `dev`
- Create PR with description
- Request team review

MEMBER 3: Feature Engineering Layer

Day 1 Morning (Dec 12, 9am-1pm)

Task 3.1: Base Feature Class (1 hour)

Create `winepredict/features/base.py`:

```python
```

```python
from abc import ABC, abstractmethod
import pandas as pd

class BaseFeature(ABC):
    """Abstract base class for all feature engineering classes."""

    def __init__(self, feature_names: list = None):
        self.feature_names = feature_names or []

    @abstractmethod
    def compute(self, X: pd.DataFrame) -> pd.DataFrame:
        """Compute features from input data.

        Args:
            X: Input features DataFrame

        Returns:
            DataFrame with computed features
        """
        pass

    def get_feature_names(self) -> list:
        """Return list of feature names this class generates."""
        return self.feature_names

    def __call__(self, X: pd.DataFrame) -> pd.DataFrame:
        """Allow calling instance directly: features = feature_obj(X)"""
        return self.compute(X)
```

- Commit: "Add base feature class"

Task 3.2: Chemical Ratio Features (1 hour)

Create `winepredict/features/chemical_ratios.py`:

```python
```

```
import pandas as pd
import numpy as np
from .base import BaseFeature

class ChemicalRatioFeatures(BaseFeature):
    """Compute chemistry-based ratio features."""

    def __init__(self):
        feature_names = [
            'acidity_ratio',
            'sulfur_ratio',
            'acid_balance',
            'acidity_per_density'
        ]
        super().__init__(feature_names)

    def compute(self, X: pd.DataFrame) -> pd.DataFrame:
        """Compute chemical ratio features."""
        features = pd.DataFrame(index=X.index)

        # Fixed to volatile acidity ratio (higher = better balance)
        features['acidity_ratio'] = (
            X['fixed_acidity'] / (X['volatile_acidity'] + 1e-6)
        )

        # Free to total sulfur dioxide ratio (preservation indicator)
        features['sulfur_ratio'] = (
            X['free_sulfur_dioxide'] / (X['total_sulfur_dioxide'] + 1e-6)
        )

        # Overall acid balance (citric + fixed over volatile)
        features['acid_balance'] = (
            (X['citric_acid'] + X['fixed_acidity']) / (X['volatile_acidity'] + 1e-6)
        )

        # Acidity concentration (normalized by density)
        features['acidity_per_density'] = (
            X['fixed_acidity'] / X['density']
        )

        return features
```

- Test manually with sample data

- Commit: "Implement chemical ratio features"

Task 3.3: Interaction Features (1 hour)

Create `winepredict/features/interactions.py`:

```python
```

```python
import pandas as pd
from .base import BaseFeature

class InteractionFeatures(BaseFeature):
    """Compute interaction features between important variables."""

    def __init__(self):
        feature_names = [
            'alcohol_ph',
            'alcohol_sulphates',
            'alcohol_acidity',
            'sulphates_acidity',
            'density_alcohol',
        ]
        super().__init__(feature_names)

    def compute(self, X: pd.DataFrame) -> pd.DataFrame:
        """Compute multiplicative interaction features."""
        features = pd.DataFrame(index=X.index)

        # Alcohol × pH (preservation interaction)
        features['alcohol_ph'] = X['alcohol'] * X['pH']

        # Alcohol × Sulphates (quality indicator)
        features['alcohol_sulphates'] = X['alcohol'] * X['sulphates']

        # Alcohol × Acidity (balance indicator)
        features['alcohol_acidity'] = X['alcohol'] * X['fixed_acidity']

        # Sulphates × Acidity (preservation and taste)
        features['sulphates_acidity'] = X['sulphates'] * X['fixed_acidity']

        # Density-Alcohol interaction (body indicator)
        features['density_alcohol'] = X['density'] * X['alcohol']

        return features
```

- Commit: "Implement interaction features"

Day 1 Afternoon (Dec 12, 2pm-6pm)

Task 3.4: Domain-Specific Features (1.5 hours)

Create `winepredict/features/domain_features.py`:

```python
```

```
import pandas as pd
import numpy as np
from .base import BaseFeature

class DomainFeatures(BaseFeature):
    """Domain-specific wine quality indicators."""

    def __init__(self):
        feature_names = [
            'preservation_score',
            'sweetness_index',
            'balance_score',
            'complexity_score',
            'body_indicator'
        ]
        super().__init__(feature_names)

    def compute(self, X: pd.DataFrame) -> pd.DataFrame:
        """Compute domain-specific features."""
        features = pd.DataFrame(index=X.index)

        # Preservation score: higher alcohol, sulphates, lower pH
        features['preservation_score'] = (
            X['alcohol'] * X['sulphates'] * (14 - X['pH'])
        )

        # Sweetness index: residual sugar relative to alcohol
        features['sweetness_index'] = (
            X['residual_sugar'] / (X['alcohol'] + 1)
        )

        # Balance score: normalized combination of key factors
        features['balance_score'] = (
            (X['alcohol'] / X['alcohol'].max()) +
            (X['citric_acid'] / X['citric_acid'].max()) +
            (1 - X['volatile_acidity'] / X['volatile_acidity'].max())
        ) / 3

        # Complexity score: variety of chemical components
        features['complexity_score'] = (
            X['citric_acid'] * X['sulphates'] * X['alcohol']
        )
```

```
    # Body indicator: density relative to alcohol (fuller body)
    features['body_indicator'] = (
        X['density'] / (X['alcohol'] + 1)
    )

    return features
```

- Commit: "Add domain-specific wine features"

Task 3.5: Statistical Features (1 hour)

Create `winepredict/features/statistical.py`:

```
python
```

```
import pandas as pd
import numpy as np
from scipy import stats
from .base import BaseFeature

class StatisticalFeatures(BaseFeature):
    """Statistical transformations and aggregations."""

    def __init__(self):
        feature_names = [
            'alcohol_zscore',
            'acidity_zscore',
            'sulphates_zscore',
            'alcohol_percentile',
            'quality_indicators_mean',
        ]
        super().__init__(feature_names)

    def compute(self, X: pd.DataFrame) -> pd.DataFrame:
        """Compute statistical features."""
        features = pd.DataFrame(index=X.index)

        # Z-scores for key quality indicators
        features['alcohol_zscore'] = stats.zscore(X['alcohol'])
        features['acidity_zscore'] = stats.zscore(X['fixed_acidity'])
        features['sulphates_zscore'] = stats.zscore(X['sulphates'])

        # Percentile ranking of alcohol content
        features['alcohol_percentile'] = (
            X['alcohol'].rank(pct=True) * 100
        )

        # Mean of normalized quality indicators
        quality_cols = ['alcohol', 'sulphates', 'citric_acid']
        normalized = X[quality_cols].apply(
            lambda x: (x - x.min()) / (x.max() - x.min())
        )
        features['quality_indicators_mean'] = normalized.mean(axis=1)

        return features
```

- Commit: "Implement statistical features"

Task 3.6: Polynomial Features (1 hour)

Create `winepredict/features/polynomial.py`:

```python
```

```python
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
from .base import BaseFeature

class PolynomialFeatureGenerator(BaseFeature):
    """Generate polynomial features for key variables."""

    def __init__(self, degree: int = 2, interaction_only: bool = False):
        self.degree = degree
        self.interaction_only = interaction_only
        self.poly = None
        self.key_features = [
            'alcohol', 'volatile_acidity', 'sulphates',
            'citric_acid', 'pH'
        ]
        super().__init__()

    def fit(self, X: pd.DataFrame):
        """Fit polynomial transformer on key features."""
        X_subset = X[self.key_features]
        self.poly = PolynomialFeatures(
            degree=self.degree,
            interaction_only=self.interaction_only,
            include_bias=False
        )
        self.poly.fit(X_subset)
        self.feature_names = [
            f'poly_{name}'
            for name in self.poly.get_feature_names_out(self.key_features)
        ]
        return self

    def compute(self, X: pd.DataFrame) -> pd.DataFrame:
        """Generate polynomial features."""
        if self.poly is None:
            self.fit(X)

        X_subset = X[self.key_features]
        poly_features = self.poly.transform(X_subset)

        return pd.DataFrame(
            poly_features,
            columns=self.feature_names,
```

```
        index=X.index
    )
```

- Commit: "Add polynomial feature generator"

Task 3.7: Feature Engineering Tests (1.5 hours)

Create `tests/test_features.py`:

```
python
```

```python
import pytest
import pandas as pd
import numpy as np
from winepredict.features.chemical_ratios import ChemicalRatioFeatures
from winepredict.features.interactions import InteractionFeatures
from winepredict.features.domain_features import DomainFeatures
from winepredict.features.statistical import StatisticalFeatures
from winepredict.features.polynomial import PolynomialFeatureGenerator

@pytest.fixture
def sample_wine_data():
    """Create sample wine data for testing."""
    np.random.seed(42)
    n_samples = 100

    return pd.DataFrame({
        'fixed_acidity': np.random.uniform(4, 16, n_samples),
        'volatile_acidity': np.random.uniform(0.1, 1.5, n_samples),
        'citric_acid': np.random.uniform(0, 1, n_samples),
        'residual_sugar': np.random.uniform(0.5, 15, n_samples),
        'chlorides': np.random.uniform(0.01, 0.6, n_samples),
        'free_sulfur_dioxide': np.random.uniform(1, 70, n_samples),
        'total_sulfur_dioxide': np.random.uniform(6, 290, n_samples),
        'density': np.random.uniform(0.99, 1.01, n_samples),
        'pH': np.random.uniform(2.7, 4.0, n_samples),
        'sulphates': np.random.uniform(0.3, 2.0, n_samples),
        'alcohol': np.random.uniform(8, 15, n_samples),
    })

def test_chemical_ratios(sample_wine_data):
    feature_gen = ChemicalRatioFeatures()
    features = feature_gen.compute(sample_wine_data)

    # Check all features generated
    assert len(features.columns) == len(feature_gen.feature_names)
    assert 'acidity_ratio' in features.columns
    assert 'sulfur_ratio' in features.columns

    # Check no NaN or inf values
    assert not features.isnull().any().any()
    assert not np.isinf(features.values).any()

def test_interaction_features(sample_wine_data):
```

```
    feature_gen = InteractionFeatures()
    features = feature_gen.compute(sample_wine_data)

    assert len(features.columns) == len(feature_gen.feature_names)
    assert 'alcohol_ph' in features.columns

    # Verify interaction calculation
    expected = sample_wine_data['alcohol'] * sample_wine_data['pH']
    pd.testing.assert_series_equal(
        features['alcohol_ph'],
        expected,
        check_names=False
    )

def test_domain_features(sample_wine_data):
    feature_gen = DomainFeatures()
    features = feature_gen.compute(sample_wine_data)

    assert 'preservation_score' in features.columns
    assert 'balance_score' in features.columns

    # Check balance score is between 0 and 1
    assert features['balance_score'].between(0, 1).all()

def test_statistical_features(sample_wine_data):
    feature_gen = StatisticalFeatures()
    features = feature_gen.compute(sample_wine_data)

    # Z-scores should have mean ~ 0, std ~ 1
    assert np.abs(features['alcohol_zscore'].mean()) < 0.1
    assert np.abs(features['alcohol_zscore'].std() - 1) < 0.1

    # Percentiles should be 0-100
    assert features['alcohol_percentile'].between(0, 100).all()

def test_polynomial_features(sample_wine_data):
    feature_gen = PolynomialFeatureGenerator(degree=2)
    features = feature_gen.compute(sample_wine_data)

    # Should generate more features than input
    assert len(features.columns) > len(feature_gen.key_features)

    # Check shape matches input
    assert len(features) == len(sample_wine_data)
```

```
def test_feature_get_names():
    feature_gen = ChemicalRatioFeatures()
    names = feature_gen.get_feature_names()

    assert isinstance(names, list)
    assert len(names) > 0
```

- Run all tests: `pytest tests/test_features.py -v --cov=winepredict/features`
- Commit: "Add comprehensive feature engineering tests"

Day 2 Morning (Dec 13, 9am-12pm)

Task 3.8: Feature Module Integration (30 min)

Update `winepredict/features/__init__.py`:

```python
from .base import BaseFeature
from .chemical_ratios import ChemicalRatioFeatures
from .interactions import InteractionFeatures
from .domain_features import DomainFeatures
from .statistical import StatisticalFeatures
from .polynomial import PolynomialFeatureGenerator

__all__ = [
    'BaseFeature',
    'ChemicalRatioFeatures',
    'InteractionFeatures',
    'DomainFeatures',
    'StatisticalFeatures',
    'PolynomialFeatureGenerator',
]
```

Task 3.9: README Documentation - Features (1 hour)

Add to README.md:

```markdown
```

Feature Engineering

Available Feature Sets

1. ****ChemicalRatioFeatures****: Acidity ratios, sulfur ratios, balance indicators
2. ****InteractionFeatures****: Multiplicative interactions between key variables
3. ****DomainFeatures****: Wine-specific quality indicators
4. ****StatisticalFeatures****: Z-scores, percentiles, aggregations
5. ****PolynomialFeatureGenerator****: Polynomial and interaction terms

Feature Design Philosophy

Features are designed to capture:
- Chemical balance and preservation
- Wine body and complexity
- Quality indicators from domain knowledge
- Non-linear relationships

Adding a New Feature Set

[Include detailed example with code]

Task 3.10: Pull Request (30 min)

- Push feature engineering work
- Create PR with examples
- Request team review

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, Testing & Documentation (Dec 15-16)

ALL MEMBERS: Collaborative Integration (Dec 15)

Morning Session (Dec 15, 9am-12pm)

Task ALL.1: Pre-Integration Prep (All Members, 30 min each)

- Pull latest `dev` branch
- Review each other's PRs
- Note any blockers or dependencies
- Create shared Slack/Discord channel for real-time coordination

Task ALL.2: Integration Testing (All Members Together, 2 hours)

Member 1 Leads: API Integration

1. Ensure all team members have run `python run_pipeline.py` to generate model
2. Start API server: `uvicorn api.app:app --reload`
3. Test endpoints together:

```bash
```

```
# Health check
curl http://localhost:8000/health

# Prediction test
python api/test_api.py
```

4. Debug any model loading issues
5. Verify feature generation matches training pipeline

Member 2 Leads: Model Validation

1. Run full pipeline with both model types:

```
python

# Test Random Forest
pipeline.run_full_pipeline(model_type='random_forest')

# Test XGBoost
pipeline.run_full_pipeline(model_type='xgboost')
```

2. Compare results and choose best model
3. Verify recall metrics meet targets (>0.50 for minority classes)
4. If metrics are low, troubleshoot:
    - Check class weight calculation
    - Verify stratified splitting
    - Review feature importance
    - Adjust hyperparameter grid

Member 3 Leads: Pipeline Integration

1. Run Jupyter notebook end-to-end
2. Verify all visualizations render correctly
3. Test with different config settings:
    - Disable polynomial features
    - Change scaler type

- Adjust CV folds

4. Ensure pipeline is reproducible (same random seed = same results)

Task ALL.3: Unit Test Pass (All Members, 30 min)

```bash
# Run all tests
pytest tests/ -v --cov=winepredict --cov-report=html

# Check coverage
open htmlcov/index.html  # or xdg-open on Linux
```

- Fix any failing tests together
- Aim for >70% coverage on core modules
- Commit: "Fix integration issues and ensure all tests pass"

Afternoon Session (Dec 15, 2pm-6pm)

MEMBER 1: API Documentation & Testing

Task 1.1: Comprehensive API Documentation (1.5 hours)

Update `api/README.md`:

```markdown
```

Wine Quality Prediction API

Overview
REST API for predicting wine quality from physicochemical properties. Supports real-time inference with confidence scores.

Quick Start

Installation
```bash
pip install -r requirements.txt
```

Running the API
```bash
# From project root
cd api
uvicorn app:app --reload --port 8000
```

API available at: http://localhost:8000

Interactive Documentation
- Swagger UI: http://localhost:8000/docs
- ReDoc: http://localhost:8000/redoc

API Endpoints

1. GET /
Root endpoint with API info.

2. GET /health
Health check and model status.

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

3. POST /predict

Predict wine quality from features.

Request Body:
```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:
```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": "2024-12-15T14:30:00Z"
}
```

Example Usage

Python
```python
import requests

url = "http://localhost:8000/predict"
wine = {
    "fixed_acidity": 7.4,
```

```
    "volatile_acidity": 0.7,
    # ... other features
}

response = requests.post(url, json=wine)
print(response.json())
```

curl
```bash
curl -X POST "http://localhost:8000/predict" \
  -H "Content-Type: application/json" \
  -d @api/sample_inputs/average_wine.json
```

Sample Inputs
Pre-made examples in `api/sample_inputs/`:
- `poor_wine.json` - Low quality wine
- `average_wine.json` - Average quality wine
- `good_wine.json` - Good quality wine
- `excellent_wine.json` - Excellent quality wine

Error Handling

Validation Errors (422)
```json
{
  "detail": [
    {
      "loc": ["body", "alcohol"],
      "msg": "ensure this value is less than or equal to 15",
      "type": "value_error.number.not_le"
    }
  ]
}
```

Model Not Loaded (503)
```json
{
  "detail": "Model not loaded. Please check server logs."
}
```

Model Information
- **Model Type:** Random Forest / XGBoost (with class weights)
- **Training Data:** Portuguese Vinho Verde red wines
- **Classes:** poor, average, good, excellent
- **Metrics:** Optimized for recall on minority classes

Task 1.2: Create Sample Input Files (30 min)

api/sample_inputs/poor_wine.json:

```
json

{
  "fixed_acidity": 8.9,
  "volatile_acidity": 1.2,
  "citric_acid": 0.0,
  "residual_sugar": 1.5,
  "chlorides": 0.15,
  "free_sulfur_dioxide": 5.0,
  "total_sulfur_dioxide": 15.0,
  "density": 0.998,
  "pH": 3.8,
  "sulphates": 0.4,
  "alcohol": 8.5
}
```

api/sample_inputs/excellent_wine.json:

```
json

{
  "fixed_acidity": 7.0,
  "volatile_acidity": 0.3,
  "citric_acid": 0.5,
  "residual_sugar": 2.0,
  "chlorides": 0.05,
  "free_sulfur_dioxide": 30.0,
  "total_sulfur_dioxide": 80.0,
  "density": 0.995,
  "pH": 3.3,
  "sulphates": 0.9,
  "alcohol": 13.0
}
```

Create similar files for `average_wine.json` and `good_wine.json`.

Task 1.3: API Testing Suite (1 hour)

Enhance `api/test_api.py`:

```python
```

```python
import requests
import json
import sys
from pathlib import Path

BASE_URL = "http://localhost:8000"

def load_sample(filename):
    """Load sample wine data from file."""
    path = Path("api/sample_inputs") / filename
    with open(path) as f:
        return json.load(f)

def test_health():
    """Test health endpoint."""
    print("="*50)
    print("TEST 1: Health Check")
    print("="*50)

    response = requests.get(f"{BASE_URL}/health")
    result = response.json()

    print(f"Status Code: {response.status_code}")
    print(json.dumps(result, indent=2))

    assert response.status_code == 200
    assert result['model_loaded'] == True
    print("✓ Health check passed\n")

def test_predict_all_samples():
    """Test predictions on all sample wines."""
    print("="*50)
    print("TEST 2: Predictions on Sample Wines")
    print("="*50)

    samples = [
        "poor_wine.json",
        "average_wine.json",
        "good_wine.json",
        "excellent_wine.json"
    ]

    for sample_file in samples:
```

```
        print(f"\n{sample_file}:")
        wine_data = load_sample(sample_file)

        response = requests.post(
            f"{BASE_URL}/predict",
            json=wine_data
        )

        result = response.json()
        print(f"  Predicted: {result['predicted_class']}")
        print(f"  Confidence: {result['confidence']:.2%}")
        print(f"  Probabilities: {json.dumps(result['probabilities'], indent=4)}")

        assert response.status_code == 200

    print("\n✓ All sample predictions passed\n")

def test_invalid_inputs():
    """Test API validation with invalid inputs."""
    print("="*50)
    print("TEST 3: Input Validation")
    print("="*50)

    invalid_cases = [
        {
            "name": "Alcohol too high",
            "data": {"alcohol": 50, "pH": 3.5},  # Missing required fields
        },
        {
            "name": "Negative value",
            "data": load_sample("average_wine.json") | {"alcohol": -5}
        },
        {
            "name": "Missing required field",
            "data": {k: v for k, v in load_sample("average_wine.json").items() if k != "alcohol"}
        }
    ]

    for case in invalid_cases:
        print(f"\n{case['name']}:")
        response = requests.post(
            f"{BASE_URL}/predict",
            json=case['data']
        )
```

```
        print(f"  Status Code: {response.status_code}")
        assert response.status_code == 422  # Validation error
        print(f"  ✓ Correctly rejected")

    print("\n✓ Input validation passed\n")

def test_response_structure():
    """Verify response has correct structure."""
    print("="*50)
    print("TEST 4: Response Structure")
    print("="*50)

    wine_data = load_sample("average_wine.json")
    response = requests.post(f"{BASE_URL}/predict", json=wine_data)
    result = response.json()

    required_fields = [
        "predicted_class",
        "confidence",
        "probabilities",
        "model_version",
        "timestamp"
    ]

    for field in required_fields:
        assert field in result, f"Missing field: {field}"
        print(f"  ✓ {field}: present")

    # Check probabilities sum to ~1
    prob_sum = sum(result['probabilities'].values())
    assert 0.99 <= prob_sum <= 1.01, f"Probabilities sum to {prob_sum}"
    print(f"  ✓ Probabilities sum to {prob_sum:.4f}")

    print("\n✓ Response structure valid\n")

def run_all_tests():
    """Run complete API test suite."""
    try:
        test_health()
        test_predict_all_samples()
        test_invalid_inputs()
        test_response_structure()
```

```
        print("="*50)
        print("ALL TESTS PASSED ✓")
        print("="*50)
        return 0
    except Exception as e:
        print(f"\n✗ TEST FAILED: {e}")
        return 1

if __name__ == "__main__":
    sys.exit(run_all_tests())
```

- Run tests: `python api/test_api.py`
- Commit: "Add comprehensive API testing suite"

Task 1.4: Update Main README - API Section (1 hour)

Add API section to main `README.md`

MEMBER 2: Model Documentation & Metrics Analysis

Task 2.1: Model Performance Documentation (1.5 hours)

Create `docs/MODEL_PERFORMANCE.md`:

```
markdown
```

Model Performance Report

Dataset Summary
- **Total Samples:** ~1,600
- **Features:** 11 physicochemical properties
- **Target:** 4-class quality (poor, average, good, excellent)
- **Class Distribution:**
  - Poor: ~X%
  - Average: ~Y% (majority)
  - Good: ~Z%
  - Excellent: ~W%

Class Imbalance Strategy
We addressed class imbalance using:
1. **Stratified Cross-Validation** - Maintains class distribution in each fold
2. **Class Weights** - Penalizes misclassification of minority classes more heavily
3. **Recall-Focused Optimization** - Prioritizes catching minority class samples

Model Comparison

Random Forest (with class_weight='balanced')
- **Hyperparameters:**
  - n_estimators: [best value]
  - max_depth: [best value]
  - min_samples_split: [best value]

- **Performance:**
  - Weighted Recall: X.XX
  - Weighted F1: X.XX
  - Accuracy: X.XX

- **Per-Class Recall:**
  - Poor: X.XX
  - Average: X.XX
  - Good: X.XX
  - Excellent: X.XX

XGBoost (with scale_pos_weight)
[Similar structure]

Final Model Selection
Selected Model: [Random Forest / XGBoost]

Rationale:
- Best weighted recall: X.XX
- Good minority class recall (>0.50 for poor/excellent)
- Robust to small dataset size
- Fast inference time

Feature Importance
Top 10 most important features:
1. [Feature name]: X.XX importance
2. ...

Confusion Matrix Analysis
[Describe confusion patterns]
- Model correctly identifies X% of average wines
- Most errors are between adjacent classes (e.g., good vs average)
- Rare classes (poor, excellent) have X% recall

Recommendations for Future Improvement
1. **Data Collection:**
   - Collect more samples of poor/excellent wines
   - Target 200+ samples per class

2. **Feature Engineering:**
   - Explore temporal features (wine age)
   - Include grape variety information
   - Experiment with ensemble of binned vs fine-grained predictions

3. **Model Architecture:**
   - Try ensemble methods (stacking, blending)
   - Experiment with ordinal classification (since quality is ordered)
   - Consider two-stage classification (binary then multi-class)

4. **Class Imbalance:**
   - If recall remains low, try SMOTE for synthetic oversampling
   - Experiment with cost-sensitive learning

Task 2.2: Hyperparameter Tuning Analysis (1 hour)

Create notebook section or script analyzing:

- CV scores across different hyperparameters
- Tuning time vs performance tradeoff
- Visualize hyperparameter importance

Task 2.3: Model Checkpoints (30 min)

- Save multiple model versions
- Document which model is in production
- Create model versioning system

Task 2.4: Update README - Model Section (1 hour)

MEMBER 3: Final Documentation & Presentation Prep

Task 3.1: Comprehensive README Update (2 hours)

Complete main `README.md`:

```markdown
```

WinePredict: Wine Quality Classification Library

[![Python 3.8+](https://img.shields.io/badge/python-3.8+-blue.svg)](https://www.python.org/downloads/)
[![License: MIT](https://img.shields.io/badge/License-MIT-yellow.svg)](https://opensource.org/licenses/MIT)

Production-ready library for classifying Portuguese Vinho Verde wine quality using machine learning.

🎯 Project Overview

This project implements a complete machine learning pipeline for wine quality prediction, addressing:
- **Severe class imbalance** (using stratified CV + class weights)
- **Small dataset** (using robust preprocessing and regularization)
- **Recall optimization** (prioritizing minority class detection)

Dataset: [Kaggle Wine Quality Dataset](https://www.kaggle.com/datasets/yasserh/wine-quality-dataset)

Classification Strategy
- **Input:** 11 physicochemical wine properties
- **Output:** 4 quality classes
  - Poor (scores 0-4)
  - Average (scores 5-6)
  - Good (scores 7-8)
  - Excellent (scores 9-10)

🚀 Quick Start

Installation
```bash
git clone https://github.com/[username]/winepredict.git
cd winepredict
pip install -r requirements.txt
pip install -e .
```

Run Pipeline
```bash
python run_pipeline.py
```

Start API
```bash
cd api
uvicorn app:app --reload
```

```

### Run Tests
```bash
pytest tests/ -v --cov=winepredict
```

📁 Project Structure

[Include the full folder tree from PRD]

🔧 Core Components

1. Data Loading & Preprocessing
- **Quality Binning:** Converts 0-10 scores to 4 classes
- **Stratified Splitting:** Maintains class distribution
- **Preprocessing Pipeline:**
  - Outlier clipping (5th-95th percentile)
  - Log transformation (skewed features)
  - Robust scaling (handles outliers)

2. Feature Engineering
Five independent feature sets adding 20+ features:
- **Chemical Ratios:** Acidity balance, sulfur ratios
- **Interactions:** Alcohol×pH, sulphates×acidity
- **Domain Features:** Preservation score, balance index
- **Statistical:** Z-scores, percentiles
- **Polynomial:** Quadratic terms of key features

3. Model Training
- **Algorithms:** Random Forest, XGBoost, Gradient Boosting
- **Class Imbalance Handling:**
  - `class_weight='balanced'` for sklearn models
  - `scale_pos_weight` for XGBoost
- **Hyperparameter Tuning:** RandomizedSearchCV with stratified CV
- **Evaluation:** Recall-focused (weighted and per-class)

4. REST API
FastAPI service for real-time predictions with confidence scores.

📊 Performance Metrics

Final Model: [Model Type]
```

Metric	Value
Weighted Recall	X.XX
Weighted F1	X.XX
Accuracy	X.XX

Per-Class Recall:
- Poor: X.XX
- Average: X.XX
- Good: X.XX
- Excellent: X.XX

[See detailed analysis in `docs/MODEL_PERFORMANCE.md`]

🍼 Usage Examples

Training a New Model
```python
from winepredict.pipeline import WinePredictionPipeline

pipeline = WinePredictionPipeline()
results = pipeline.run_full_pipeline(
    model_type='random_forest',
    tune=True
)
```

Making Predictions
```python
import requests

wine = {
    "fixed_acidity": 7.4,
    "volatile_acidity": 0.7,
    "alcohol": 9.4,
    # ... other features
}

response = requests.post(
    "http://localhost:8000/predict",
    json=wine
)

print(response.json())
```

```
# {
#   "predicted_class": "average",
#   "confidence": 0.73,
#   "probabilities": {...}
# }
```

Jupyter Notebook Demo

See `notebooks/demo_pipeline.ipynb` for a complete walkthrough.

🤝 Contributing

Adding a New Preprocessor

1. Create class inheriting from `BasePreprocessor`:

```python
from winepredict.preprocessing.base import BasePreprocessor

class MyPreprocessor(BasePreprocessor):
    def fit(self, X, y=None):
        # Learn parameters from training data
        self.is_fitted = True
        return self

    def transform(self, X):
        # Apply transformation
        return X_transformed
```

2. Add to `winepredict/preprocessing/__init__.py`
3. Write unit tests in `tests/test_preprocessing.py`
4. Update this README

Adding a New Feature Set

1. Create class inheriting from `BaseFeature`:

```python
from winepredict.features.base import BaseFeature

class MyFeatures(BaseFeature):
    def __init__(self):
        super().__init__(['feature1', 'feature2'])

    def compute(self, X):
```

```
        # Generate features
        return feature_df
```

2. Add to `winepredict/features/__init__.py`
3. Write unit tests in `tests/test_features.py`
4. Update this README

Adding a New Model

1. Create class inheriting from `BaseModel`:
```python
from winepredict.models.base import BaseModel

class MyModel(BaseModel):
    def _create_model(self, **kwargs):
        # Return sklearn-compatible model
        return MyClassifier(**kwargs)
```

2. Add to `winepredict/models/__init__.py`
3. Add hyperparameter grid to `config/config.yaml`
4. Update this README

Adding a New Metric

1. Create class inheriting from `BaseMetric`:
```python
from winepredict.evaluation.base import BaseMetric

class MyMetric(BaseMetric):
    def calculate(self, y_true, y_pred, **kwargs):
        # Calculate metric
        return score
```

2. Add to `winepredict/evaluation/__init__.py`
3. Update `ModelEvaluator` to include new metric

📝 Configuration

Edit `config/config.yaml` to customize:
- Quality class bins
- Preprocessing methods

- Feature engineering settings
- Model hyperparameters
- CV folds and random seed

🧪 Testing

```bash
# Run all tests
pytest tests/ -v

# Run with coverage
pytest tests/ --cov=winepredict --cov-report=html

# Run specific test file
pytest tests/test_features.py -v
```

📚 Documentation

- [API Documentation](api/README.md)
- [Model Performance Analysis](docs/MODEL_PERFORMANCE.md)
- [Demo Notebook](notebooks/demo_pipeline.ipynb)

🎓 Team

- **Member 1:** Infrastructure, Data, API
- **Member 2:** Preprocessing, Models, Evaluation
- **Member 3:** Feature Engineering, Pipeline

📄 License

MIT License - see LICENSE file for details

🙏 Acknowledgments

- Dataset: UCI Machine Learning Repository
- Course: Computing for Data Science
- Professor: [Name]

Task 3.2: Create Presentation Outline (1 hour)

Create `PRESENTATION_OUTLINE.md`:

```
markdown
```

Wine Quality Prediction - Presentation Outline

****Duration:**** 10-15 minutes

Slide 1: Title (30 seconds)

- Project name: WinePredict
- Team members
- Date

Slide 2: Problem Statement (1 minute)

- Predict wine quality from chemical properties
- Key challenges:
  - Severe class imbalance (show distribution chart)
  - Small dataset (~1,600 samples)
  - Need high recall on minority classes

Slide 3: Dataset Overview (1 minute)

- 11 physicochemical features
- Original 0-10 quality scale → 4 bins
- Class distribution visualization

LIVE DEMO SECTION (7-9 minutes)

Demo 1: Architecture (2 minutes)

****Screen:**** VS Code with folder structure

- Walk through modular design
- Show base classes (BasePreprocessor, BaseFeature, BaseModel)
- Explain extensibility

Demo 2: Pipeline Execution (3 minutes)

****Screen:**** Jupyter notebook

- Load data and show imbalance
- Run preprocessing steps
- Show feature engineering output
- Train model with class weights
- Display evaluation metrics

Demo 3: API (2 minutes)

****Screen:**** Terminal + Postman

- Start API server
- Make prediction request
- Show JSON response with probabilities
- Demonstrate health check

Demo 4: Results Analysis (2 minutes)
Screen: Jupyter notebook
- Confusion matrix
- Per-class recall breakdown
- Feature importance plot

Slide 4: Technical Approach (2 minutes)
- Class imbalance solution: Stratified CV + class weights
- Why NOT SMOTE: Simpler, faster, no synthetic data
- Recall-focused optimization

Slide 5: Key Results (1 minute)
- Model performance table
- Highlight minority class recall
- Compare to baseline

Slide 6: Architecture & Scalability (1 minute)
- How to add new components
- Contribution guidelines
- Testing strategy

Slide 7: Team Collaboration (1 minute)
- Work division strategy
- Git workflow
- Parallel development approach

Q&A (2-3 minutes)

Backup Slides (if needed)
- Detailed hyperparameter tuning results
- Feature correlation analysis
- Alternative approaches considered

Task 3.3: Notebook Final Polish (1 hour)

- Add executive summary at top
- Ensure all cells run without errors
- Add clear section headers
- Include takeaways and conclusions

Task 3.4: Create Quick Reference Guide (30 min)

Create QUICK_REFERENCE.md:

markdown

Quick Reference Guide

Common Commands
```bash
# Train new model
python run_pipeline.py

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

# Run tests
pytest tests/ -v

# Run notebook
jupyter notebook notebooks/demo_pipeline.ipynb
```

File Locations
- Config: `config/config.yaml`
- Trained models: `api/models/`
- Test data: `data/raw/`
- Results: `notebooks/`

Key Configuration
```yaml
# Change model type
model: 'random_forest'  # or 'xgboost'

# Adjust CV folds
cv_folds: 5

# Enable/disable features
features:
  enable_polynomial: true
```

Troubleshooting
Model not loading in API:
- Run `python run_pipeline.py` first
- Check `api/models/` directory exists

Low recall on minority classes:
- Check class distribution in data

- Verify class_weight='balanced' is set
- Increase n_estimators

Tests failing:
- Update dependencies: `pip install -r requirements.txt --upgrade`
- Delete `__pycache__` folders

FINAL DAY: Polish & Submission (Dec 16)

Morning (Dec 16, 9am-12pm)

ALL MEMBERS: Final Testing (2 hours)

1. Fresh clone of repository
2. Run complete setup from README
3. Test all commands in Quick Reference
4. Verify API works end-to-end
5. Run test suite and ensure 100% pass rate

Task: Final Code Review (1 hour)

- Remove any commented-out code
- Fix any TODO comments
- Ensure consistent code style
- Add missing docstrings
- Remove debug print statements

Afternoon (Dec 16, 2pm-6pm)

Task: Documentation Final Pass (2 hours)

- Proofread all README files
- Fix typos and formatting
- Ensure all links work
- Add missing attribution
- Update any outdated information

Task: Presentation Dry Run (1 hour)

- Practice demo flow
- Test screen sharing
- Ensure all commands work
- Time the presentation
- Prepare backup screenshots

Task: Repository Polish (30 min)

- Add LICENSE file (MIT)
- Create .github/ISSUE_TEMPLATE
- Add badges to README
- Ensure .gitignore is complete

Task: Submission Preparation (30 min)

```bash
# Final commit
git add .
git commit -m "Final submission - Wine Quality Prediction Library"
git push origin main

# Create release tag
git tag -a v1.0 -m "Final submission for Dec 17 presentation"
git push origin v1.0

# Verify repository
# - All files present
# - README renders correctly
# - @Icedgarr has access
```

Task: Submit via Google Classroom (11:59pm deadline)

- Submit repository link
- Include team member names
- Add any submission notes

PRESENTATION DAY (Dec 17, 10:00-13:00)

Pre-Presentation Checklist

- [ ] Laptop fully charged
- [ ] API server tested locally
- [ ] Jupyter notebook runs without errors
- [ ] Screen sharing tested
- [ ] Backup screenshots ready
- [ ] All team members know their parts
- [ ] Questions prepared for Q&A

During Presentation

- **Member 1:** Architecture demo + API
- **Member 2:** Model training + results analysis
- **Member 3:** Feature engineering + pipeline flow

Post-Presentation

- Note any feedback from professor
- Update repository based on suggestions
- Celebrate! 🎉

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