Wine Quality Prediction

Parsa Hajiannejad

Summary

- In this project, I worked on predicting wine quality by classifying wines as either "good" or "bad" based on 11 properties. The dataset contained 6497 samples, and I handled issues like outliers and class imbalance using SMOTE. I scaled the features and applied transformations like Box-Cox to stabilize variance and reduce skewness.
- Implemented two classifiers (Logistic Regression & SVM) from scratch in three variants:
  - Linear
  - Explicit degree-2 polynomial
  - Approximate RBF (Random Fourier Features)
- Performed systematic 5-fold cross-validation grid search to tune learning rates, regularization, C, and γ
- Evaluated on held-out test set using Accuracy, Precision, Recall, F1, ROC/AUC, confusion matrices, and learning-curve analysis

Steps

1. **Data Preparation:**
   - I cleaned the dataset by removing duplicate rows and ensuring there were no missing values.
   - I addressed the class imbalance problem by applying SMOTE to balance the classes, ensuring fair model training.
2. **Feature Engineering and Scaling:**
   - I used all 11 physicochemical features without any additional transformations, except for scaling.
   - I applied StandardScaler to standardize the features and Box-Cox PowerTransformer to handle skewness and stabilize variance.
   - Winsorization was used to cap outliers at the 1st and 99th percentiles instead of removing them.
3. **Model Selection & Tuning:**
   - Implemented from-scratch Logistic Regression and SVM:
     1. Linear decision boundary
     2. Non-linear via explicit degree-2 polynomial features
     3. Non-linear via Random Fourier feature approximation of RBF kernel
   - Performed 5-fold cross-validation for each variant:
     1. LR: tune $\alpha \in \{0.1, 0.01\}$, $\lambda \in \{0, 0.1\}$
     2. SVM: tune $\alpha \in \{0.01, 0.001\}$, $C \in \{0.1, 1, 10\}$
     3. RBF: tune $\gamma \in \{0.01, 0.1, 1\}$, $C \in \{0.1, 1, 10\}$

Steps

4. Model Evaluation:

- 5-fold cross-validation to estimate CV accuracy for all six model variants
- Compared on test set using:
  - Confusion matrices
  - ROC curves & AUC
  - Accuracy, Precision, Recall, F1
  - Learning curves (loss vs. epoch)
  - Train vs. test accuracy to check for over/under-fitting

5. Final Results

- Best overall: Logistic Regression with polynomial features (test Accuracy ≈ 0.76, AUC ≈ 0.84)
- SVM poly close second (Accuracy ≈ 0.75, AUC ≈ 0.83)
- RBF variants underperformed relative to explicit polynomial expansion

Wine Quality Dataset Overview

- **Dataset size**: 6497 wine samples (1599 red + 4898 white)
- **Source**: UCI Machine Learning Repository
- **Features**: 11 numeric physicochemical attributes
- **Target**: Quality score (int between 3 and 9)
- **Goal**: Classify wine as “good” or “bad” based on chemical properties

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

Data Integrity Checks

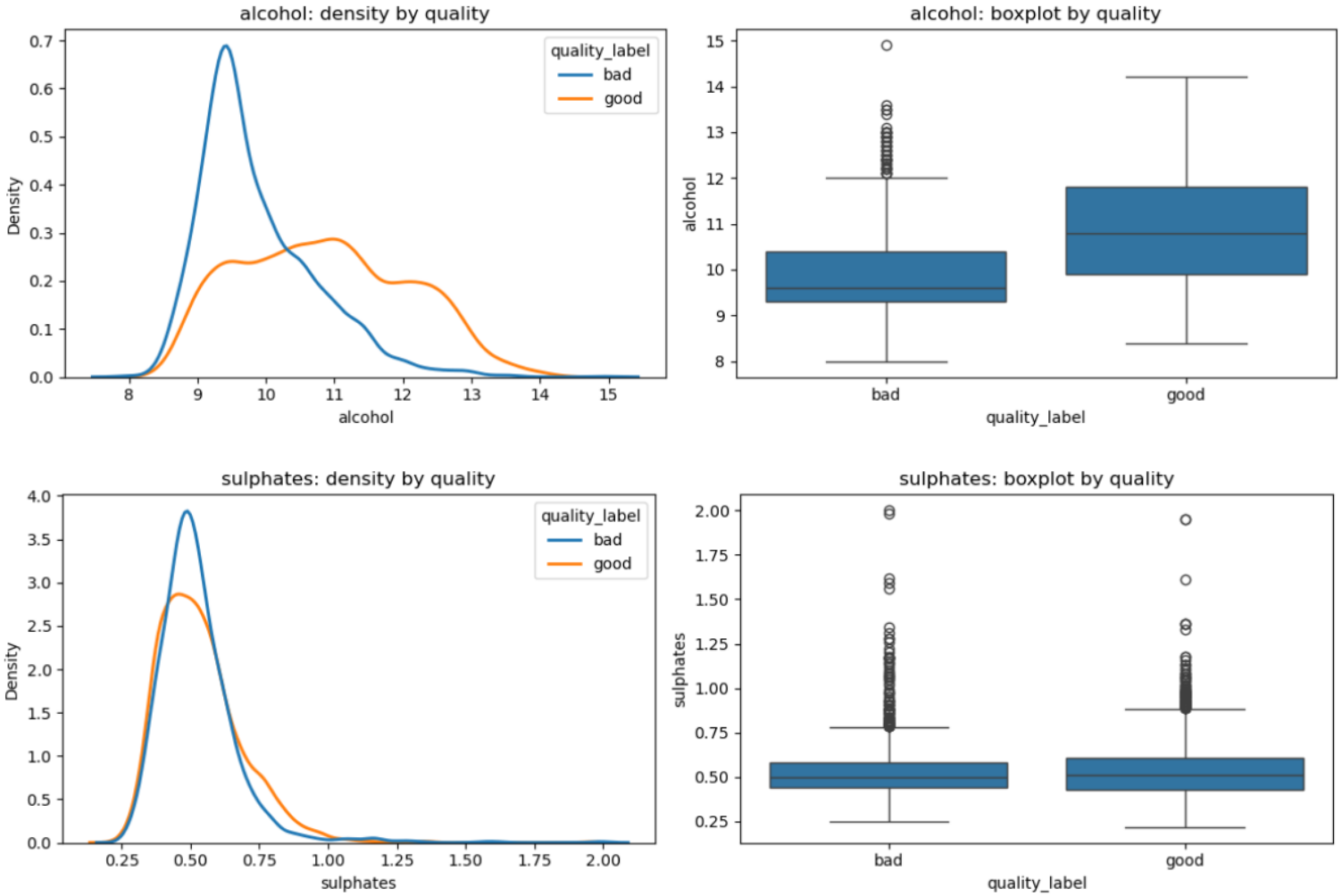
- **No missing values**
- **Duplicates found and removed:**
  - Red wine: 240 duplicates
  - White wine: 937 duplicates
  - Kept only unique rows
- **Data types checked and consistent**
  - All features are numerical (floats/integers)
  - No need for categorical encoding
- **Label harmonization**
  - Combined datasets added a type column: "red" or "white"

```
Missing values per column:
 fixed acidity           0
volatile acidity         0
citric acid              0
residual sugar           0
chlorides                0
free sulfur dioxide      0
total sulfur dioxide     0
density                  0
pH                       0
sulphates                0
alcohol                  0
quality                  0
wine_type                0
quality_label            0
dtype: int64

Number of duplicate rows: 1177
```

```
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         6497 non-null   float64
 1   volatile acidity      6497 non-null   float64
 2   citric acid           6497 non-null   float64
 3   residual sugar        6497 non-null   float64
 4   chlorides             6497 non-null   float64
 5   free sulfur dioxide   6497 non-null   float64
 6   total sulfur dioxide  6497 non-null   float64
 7   density               6497 non-null   float64
 8   pH                    6497 non-null   float64
 9   sulphates             6497 non-null   float64
 10  alcohol               6497 non-null   float64
 11  quality               6497 non-null   int64
 12  wine_type             6497 non-null   object
 13  quality_label         6497 non-null   object
dtypes: float64(11), int64(1), object(2)
memory usage: 710.7+ KB
```

Feature Distributions(example)



Key Statistics

- **Symmetry vs. Skew**: Most chemistry characteristics (fixed acidity, pH, density) are approximately symmetric (mean≈median)
- **Range & Outliers**: Residual sugar and sulfur-dioxide are widely ranged

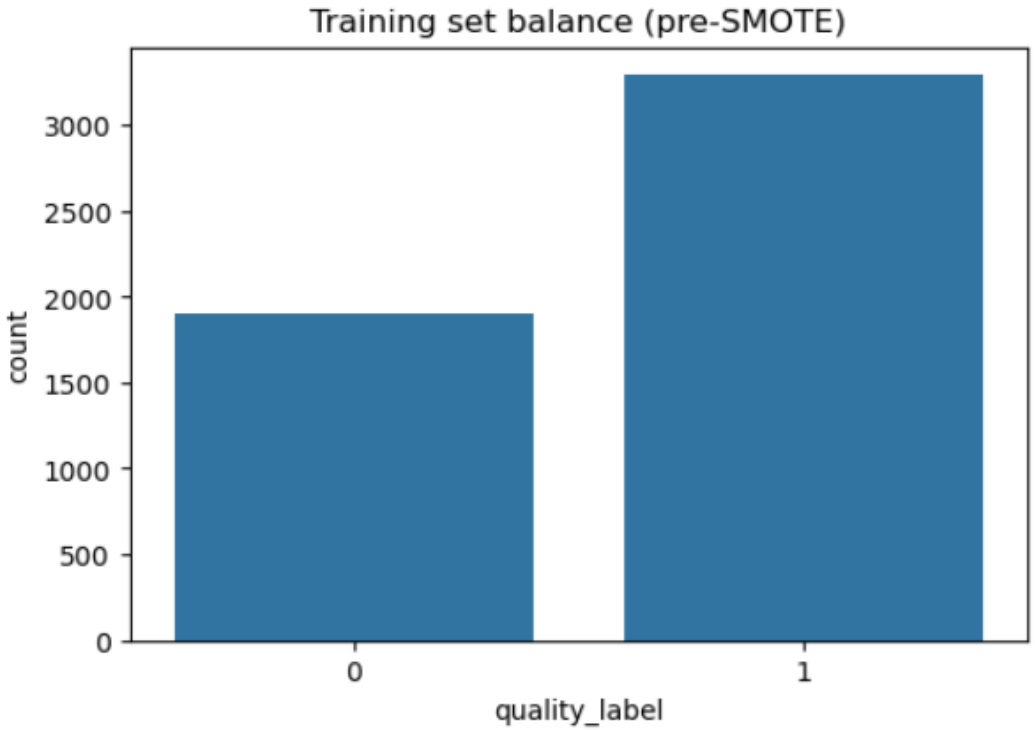
- **Quality Balance**: Quality measures range 3 to 9 with a mild bias toward superior marks (mean 5.82, median 6.00), providing approximately two-thirds “good” (≥6) vs. one-third “bad”(class imbalance)

	mean	std	min	25%	50%	75%	max
fixed acidity	7.215	1.296	3.8	6.4	7	7.7	15.9
volatile acidity	0.34	0.165	0.08	0.23	0.29	0.4	1.58
citric acid	0.319	0.145	0	0.25	0.31	0.39	1.66
residual sugar	5.443	4.758	0.6	1.8	3	8.1	65.8
chlorides	0.056	0.035	0.009	0.038	0.047	0.065	0.611
free sulfur dioxide	30.525	17.749	1	17	29	41	289
total sulfur dioxide	115.745	56.522	6	77	118	156	440
density	0.995	0.003	0.987	0.992	0.995	0.997	1.039
pH	3.219	0.161	2.72	3.11	3.21	3.32	4.01
sulphates	0.531	0.149	0.22	0.43	0.51	0.6	2
alcohol	10.492	1.193	8	9.5	10.3	11.3	14.9
quality	5.818	0.873	3	5	6	6	9

Target Variable and Class Imbalance

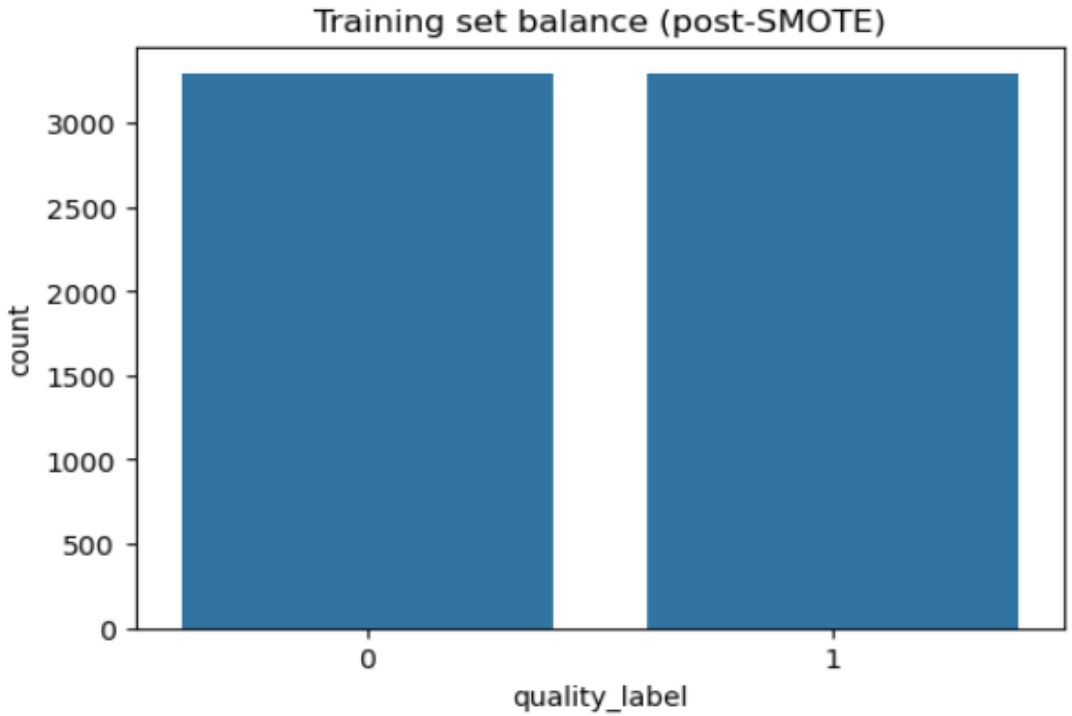
- **Target definition**
  - Quality scores (integer: 3–9) grouped into binary classes
  - Label 1 (“good”): quality ≥ 6
  - Label 0 (“bad”): quality < 6

- **Class imbalance issue:**
  - Majority of wines labeled “good” → imbalance may bias models



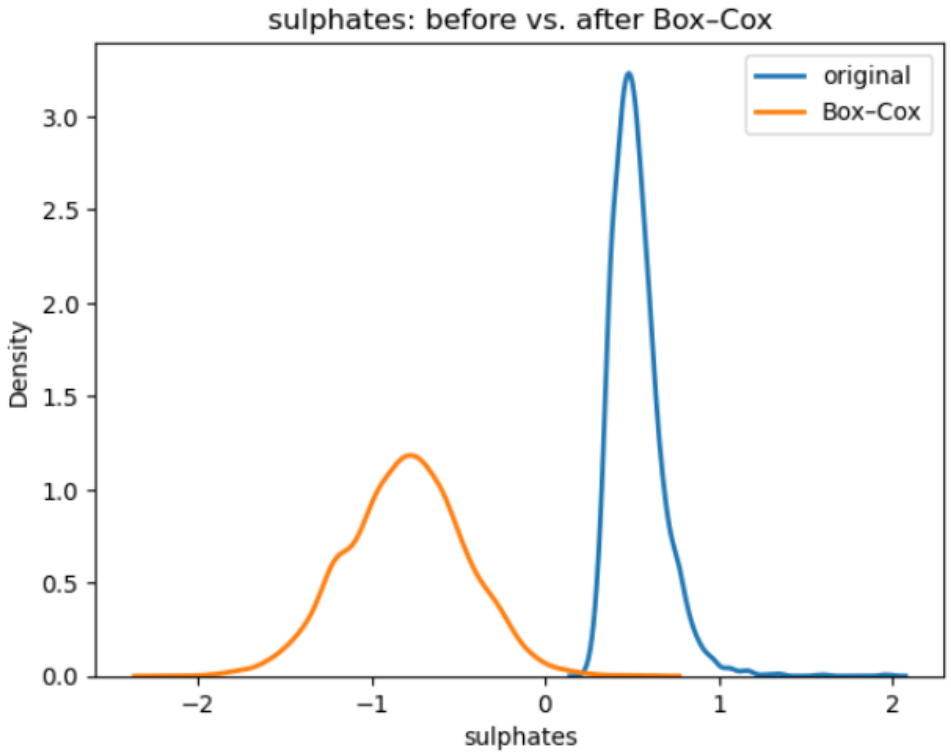
Addressing Class Imbalance (SMOTE)

- **Technique used:** SMOTE (Synthetic Minority Oversampling Technique)
  - Only applied on the **training set**, not test set
- **Oversampling parameters:**
  - k_neighbors = 5
  - Balanced both classes to the majority class size
- **Effectiveness:**
  - Transformed the skewed distribution into a balanced one
  - Essential for fair model training and evaluation



Feature Engineering and Scaling Pipeline

- **Feature selection:**
  - Used all 11 physicochemical variables (no derived features or domain-specific encoding)
- **Feature scaling strategy:**
  - Built a pipeline
  - StandardScaler: to center and scale features (mean = 0, std = 1)
  - Box-Cox PowerTransformer: Reduces skewness and stabilizes variance
- **Why this matters:**
  - Logistic regression and SVM are sensitive to feature scales
  - Helps gradient-based optimization converge faster
- **Implementation:**
  - Combined via make_pipeline() and fit only on training data (no leakage)

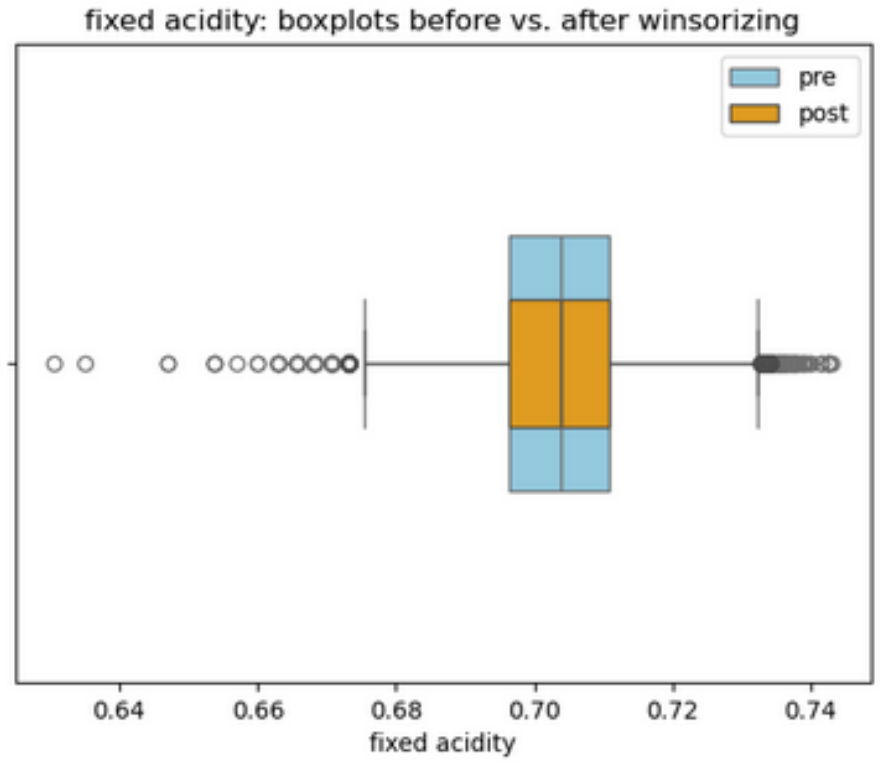


Outlier handling

- **Winsorization** is a technique where extreme values are capped at a certain percentile to reduce their influence on the dataset.(I have chosen 1st and 99st percentile)

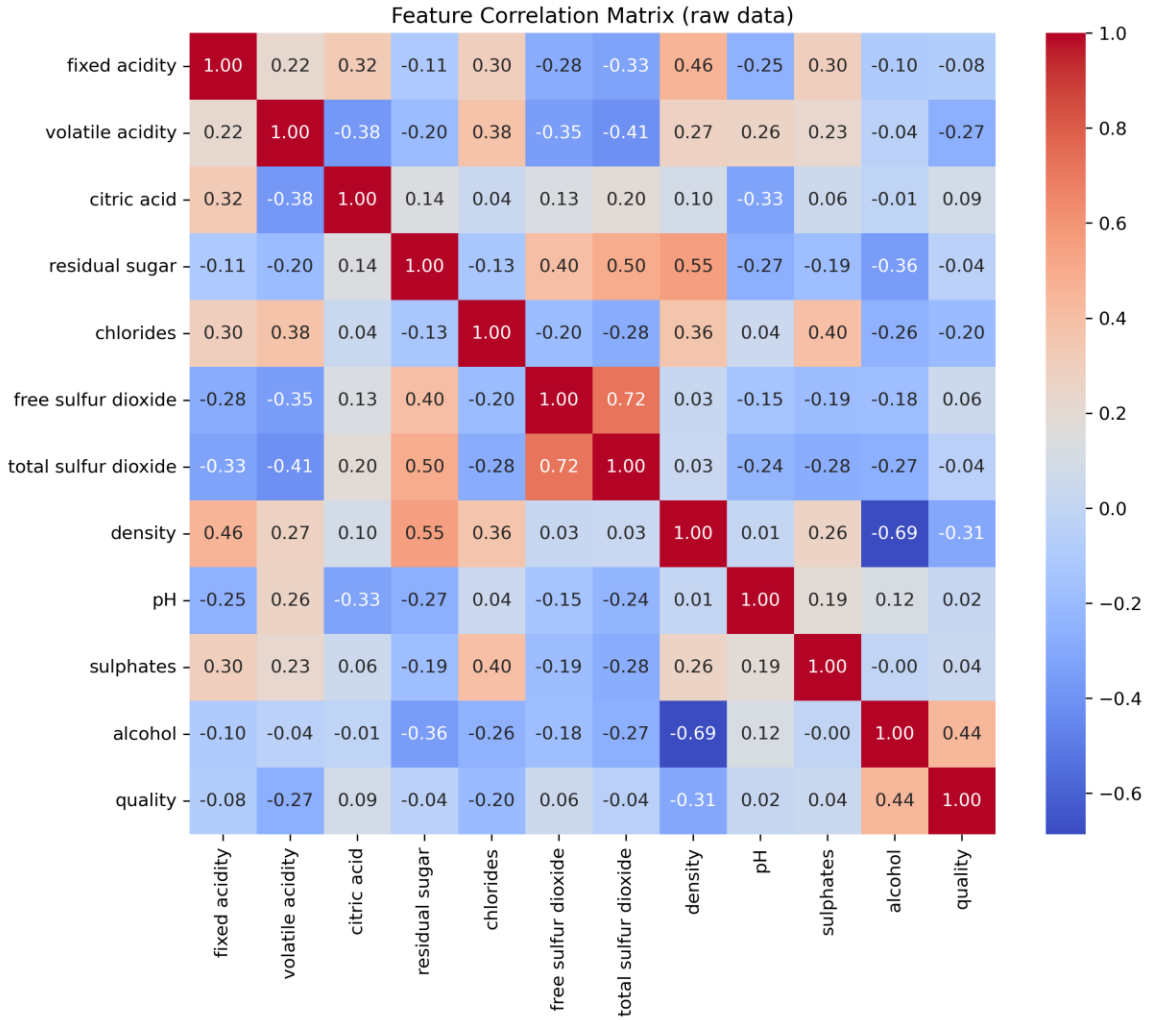
- As an example, I have shown the impact of this method on a feature(fixed acidity)

- winsorization successfully reduces outliers, resulting in a more stable and consistent distribution without removing data points



Correlation Matrix & Feature Insights

- **Correlation heatmap** computed using Pearson coefficients
- Detected **strong correlations** among several features
- **Multicollinearity** observed in certain clusters
- Considered for potential feature reduction, but kept all 11 for interpretability



Train/Test Split

- **Split ratio:**
  - 80 % train (5,197 samples)
  - 20 % test (1,300 samples)
- **Stratification:**
  - ensures the “good”/“bad” ratio is preserved in both sets
- **Random seed:**
  - 42 for reproducibility

Split	Samples	Good(%)	Bad(%)
Train	5197	63.3	36.7
Test	1300	63.3	36.7

Logistic Regression: Notations

- **Sigmoid:** $\sigma(z) = \frac{1}{1 + e^{-z}}$
- **Prediction Probabilities:** $\hat{y}^{(i)} = \sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}}$
- **Gradients with L2 Regularization:** $\frac{\partial L}{\partial w} = \frac{1}{n}\left(X^T(\hat{y} - y) + \lambda w\right)$ $\frac{\partial L}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})$
- **Gradient Descent Updates:** $w := w - \eta \cdot \frac{\partial L}{\partial w}, \quad b := b - \eta \cdot \frac{\partial L}{\partial b}$
- **Log-loss with Regularization:** $\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right] + \frac{\lambda}{2n}\sum_{j=1}^{d} w_j^2$
- **Probabilities & Binary Prediction:** $P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{x} \cdot \mathbf{w} + b)$ $\hat{y} = \begin{cases} 1 & \text{if } \hat{p} \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$

Logistic Regression: Designed Pseudo-Code

Input:

- Data matrix X of shape (n_samples × n_features)
- Label vector y of length n_samples (0/1)
- Learning rate α
- Number of epochs T
- Regularization strength λ

Initialize:

$w \leftarrow$ zero vector of length n_features

$b \leftarrow 0$

loss_history $\leftarrow$ empty list

For epoch = 1 to T do:

1. Compute linear scores:

$z \leftarrow X \cdot w + b$

2. Apply sigmoid to get predicted probabilities:

$y\_pred \leftarrow 1 / (1 + \exp(-z))$

3. Compute errors:

$error \leftarrow y\_pred - y$

4. Compute gradients (with L2 regularization on weights):

$dw \leftarrow (X^T \cdot error + \lambda \cdot w) / n\_samples$

$db \leftarrow mean(error)$

5. Update parameters by gradient descent:

$w \leftarrow w - \alpha \cdot dw$

$b \leftarrow b - \alpha \cdot db$

6. Compute regularized log-loss:

$loss \leftarrow -mean[y \cdot \log(y\_pred) + (1-y) \cdot \log(1-y\_pred)]$

$loss \leftarrow loss + (\lambda / (2 \cdot n\_samples)) \cdot sum(w^2)$

Append loss to loss_history

End

Return model parameters {w, b} and loss_history

predict_proba(X_new):

$z\_new \leftarrow X\_new \cdot w + b$

return $1 / (1 + \exp(-z\_new))$

predict(X_new, threshold = 0.5):

probs $\leftarrow$ predict_proba(X_new)

return array where each entry = 1 if prob ≥ threshold else 0

Support Vector Machine: Notations

- **Label Conversion:** $y \in \{0, 1\} \Rightarrow y' \in \{-1, +1\}$
- **Margin Computation:** $\text{margin}_i = y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$
- **Hinge Loss + Regularization:** $\mathcal{L} = \frac{1}{2}\|\mathbf{w}\|^2 + C \cdot \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b))$
- **Gradient:** $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \frac{C}{n}\sum_{i \in \mathcal{I}} y^{(i)}\mathbf{x}^{(i)} \quad \frac{\partial \mathcal{L}}{\partial b} = -\frac{C}{n}\sum_{i \in \mathcal{I}} y^{(i)}$
- **Parameter Update:** $\mathbf{w} := \mathbf{w} - \eta \cdot \nabla_w \mathcal{L}, \quad b := b - \eta \cdot \nabla_b \mathcal{L}$

SVM: Designed Pseudo-Code

Input:

- Feature matrix X (n_samples × n_features)
- Labels y ∈ {0,1} (length n_samples)
- Learning rate α
- Epochs T
- Regularization coefficient C

Preprocessing:

1. Convert labels y′ ∈ {−1,+1}:

for i in 1…n_samples:

y′[i] ← (y[i] == 1) ? +1 : −1

Initialize:

w ← zero vector of length n_features

b ← 0

loss_history ← empty list

For epoch = 1 to T do:

1. Compute raw scores:

scores ← X · w + b

2. Compute margins:

margins ← y′ * scores

3. Identify misclassified or margin-violating samples:

mask ← (margins < 1)

4. Compute gradients:

Gradient of regularization term ½‖w‖² is w

Hinge loss derivative adds −y′[i]·x[i] for each masked sample

dw ← w − (C / n_samples) * sum_over_i(mask)[y′[i] * X[i]]

db ← − (C / n_samples) * sum_over_i(mask)[y′[i]]

5. Update parameters:

w ← w − α * dw

b ← b − α * db

6. Compute and record loss:

hinge_losses ← max(0, 1 − margins)

loss ← ½ * (w · w) + C * mean(hinge_losses)

append loss to loss_history

End

Return:

- Weight vector w
- Bias b
- loss_history

Prediction:

decision_function(X_new):

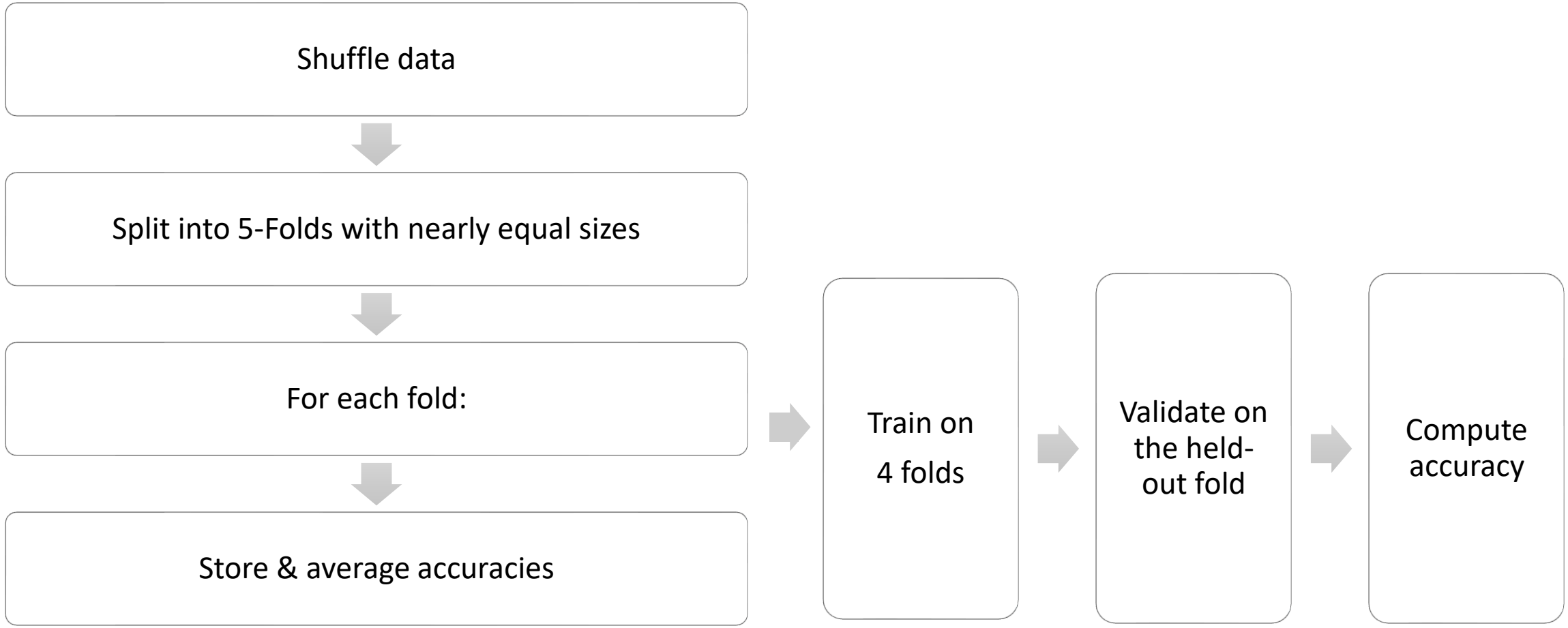
return X_new · w + b

predict(X_new):

scores ← decision_function(X_new)

return (scores ≥ 0) ? 1 : 0

5-Fold Cross-Validation Diagram



5-Fold Cross-Validation Utility

Why Cross Validation?

- **Reliable performance**

  Reduces variance vs a single train/test split
- **Hyperparameter guard**

  Prevents overfitting our tuning to one held-out set
- **Data efficiency**

  Each sample is used for validation exactly once
- **Reproducibility**

  Used np.random.seed(42) for consistent fold splits

Tuning metric: $mean(cv\_acc) = \frac{1}{5}\sum_{i=1}^{5} acc_i$

```python
def cross_val_accuracy(model_cls, X, y, cv=5, **model_kwargs):
    # convert pandas inputs to NumPy
    X_arr = X.values if hasattr(X, 'values') else np.asarray(X)
    y_arr = y.values if hasattr(y, 'values') else np.asarray(y)

    n = len(y_arr)
    indices = np.arange(n)
    np.random.seed(42)
    np.random.shuffle(indices)

    # split indices into folds
    fold_sizes = (n // cv) * np.ones(cv, dtype=int)
    fold_sizes[:n % cv] += 1

    accuracies = []
    start = 0
    for size in fold_sizes:
        end = start + size
        val_idx   = indices[start:end]
        train_idx = np.concatenate((indices[:start], indices[end:]))

        X_train_fold = X_arr[train_idx]
        y_train_fold = y_arr[train_idx]
        X_val_fold   = X_arr[val_idx]
        y_val_fold   = y_arr[val_idx]

        model = model_cls(**model_kwargs)
        model.fit(X_train_fold, y_train_fold)
        preds = model.predict(X_val_fold)
        accuracies.append((preds == y_val_fold).mean())
        start = end

    return np.array(accuracies)
```

Hyperparameter Grid Search

Logistic Regression Grid

- **Learning rates (α):** {0.1, 0.01}
  - 0.1 for faster convergence
  - 0.01 for more stable, smaller steps
- **Regularization (λ):** {0.0, 0.1}
  - 0.0 (no penalty) to test baseline fit
  - 0.1 to prevent overfitting
- **Fixed:**
  - epochs = 500

Linear SVM Grid

- **Learning rates (α):** {0.01, 0.001}
  - 0.01 to converge quickly on hinge-loss
  - 0.001 to ensure stability with large C
- **Penalty (C):** {0.1, 1.0, 10.0}
  - 0.1 enforces strong regularization (wider margin)
  - 1.0 is default balance
  - 10.0 allows more margin violations (tighter fit)
- **Fixed:**
  - epochs = 500

Procedure: evaluate mean CV accuracy over each combo; pick the highest

Logistic Regression: Mean CV Accuracy per (α, λ)

Table of Results

Learning Rate (α)	Regularization (λ)	Mean CV Accuracy
0.1	**0.0**	**0.7439**
0.1	0.1	0.7439
0.01	0.0	0.7277
0.01	0.1	0.7277

Chosen LR hyperparameters: α = 0.1, λ = 0.0

- **Learning rate impact**:
  - α = 0.1 outperforms α = 0.01 by ~1.6 , so larger step size converges more effectively for our problem.
- **Model behavior**:
  - No signs of over-regularization here (λ up to 0.1), but too small α underfits
- **Best setting**:
  - α = 0.1, λ = 0.0 (Mean CV = 0.7439)
- **Effect of regularization**:
  - Identical scores at λ = 0.0 and λ = 0.1 → small L_2 penalty has no impact at this learning rate.

Linear SVM: Mean CV Accuracy per (α, C)

Table of Results

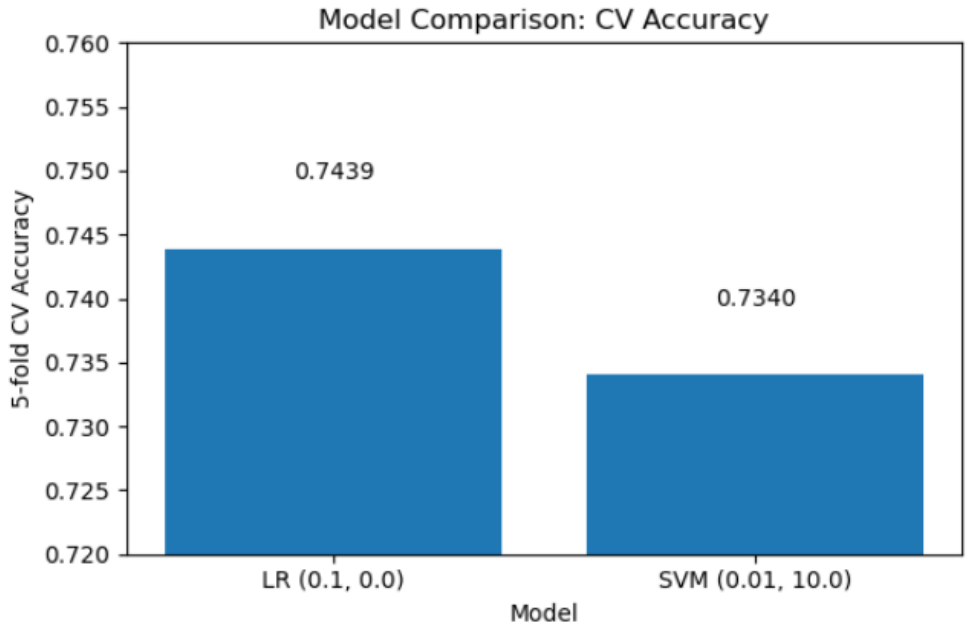
Learning Rate (α)	Penalty(C)	Mean CV Accuracy
0.01	0.1	0.6909
0.01	1.0	0.7040
0.01	**10.0**	**0.7340**
0.001	0.1	0.6910
0.001	1.0	0.6912
0.001	10.0	0.7277

Chosen SVM hyperparameters: α = 0.01, C = 10.0

- **Effect of C:**
  - Increasing C from 0.1 → 1.0 → 10.0 improves accuracy at α = 0.01
  - At α = 0.001, the boost from C=0.1→10.0 is smaller
- **Learning rate impact:**
  - α = 0.01 consistently outperforms α = 0.001 across C values by ~0.02–0.03 points, so a larger step size is needed to properly optimize the hinge-loss.
- **Bias–variance trade-off:**
  - Low C (0.1) underfits (strong regularization → lower scores), very high C (10.0) gives best fit without clear overfitting in CV

Chosen Hyperparameters

Model	Parameters	Mean CV accuracy
Linear Regression	α=0.1 λ=0.0	0.7439
Support Vector Machines	α=0.01 C=10.0	0.7340



Kernel Trick & Non-Linear Mapping

- **Why kernels?**
  - Linear models fail on non-linearly separable data.
  - Map to higher dimensions to carve out complex boundaries.

- **Explicit vs. implicit mapping:**
  - Explicit builds $\phi(x) \in \mathbb{R}^P$ with $P \gg p$.
  - Kernel trick computes $K(x,x') = \langle \phi(x), \phi(x') \rangle$ without ϕ.

- **Dual formulation of SVM:**
  - Optimization depends only on inner products $K(x_i, x_j)$.
  - Memory/time cost: $O(n^2)$ for Gram matrix.

- **Common kernels:**
  - Polynomial: $(\gamma \cdot x \cdot x' + r)^d$
  - RBF: $\exp(-\gamma \|x - x'\|^2)$
  - Sigmoid: $\tanh(\gamma \cdot x \cdot x' + r)$

Degree-2 Polynomial Expansion & CV Results

```
def polynomial_features(X):

    X_arr = X.values if hasattr(X, 'values') else np.asarray(X)
    n, m = X_arr.shape
    expanded = [X_arr]

        for i in range(m):
            for j in range(i, m):
                prod = (X_arr[:, i] * X_arr[:, j]).reshape(n, 1)
                expanded.append(prod)

    return np.hstack(expanded)
```

Model	α,λ or C grid	Best Params	Mean CV
Poly LR	α∈{0.01,0.1} λ∈{0,0.1}	α=0.10 λ=0.0	0.7614
Poly SVM	α∈{0.001,0.01} C∈{0.1,1,10}	α=0.01 C=10	0.7509

Model	Fold Accuracies
Poly LR	0.773 | 0.783 | 0.752 | 0.750 | 0.749
Poly SVM	0.741 | 0.773 | 0.748 | 0.755 | 0.737

RBF-Kernel via Random Fourier Features

```
def rbf_random_features(X, D, γ):

    rng = np.random.RandomState(42)

    W = rng.normal(0,√(2γ), size= (X.shape[1],D) )

    b = rng.uniform(0,2π,size=D)

    return √(2/D)*cos(X.dot(W)+b)

X_rff = rbf_random_features(X_train_scaled, D=200, γ=0.1)
```

Model	**γ grid × C grid**	**Best Params**	**Mean CV**
RBF LR	γ∈{0.01,0.1,1} C∈{0.1,1,10}	γ=0.1 C=0.1	0.7160
RBF SVM	γ∈{0.01,0.1,1} C∈{0.1,1,10}	γ=0.1 C=10	0.6638

Model	Fold Accuracies
RBF LR	0.695 | 0.728 | 0.722 | 0.726 | 0.709
RBF SVM	0.682 | 0.600 | 0.663 | 0.689 | 0.685

Linear vs. Poly vs. RBF: Performance & Costs

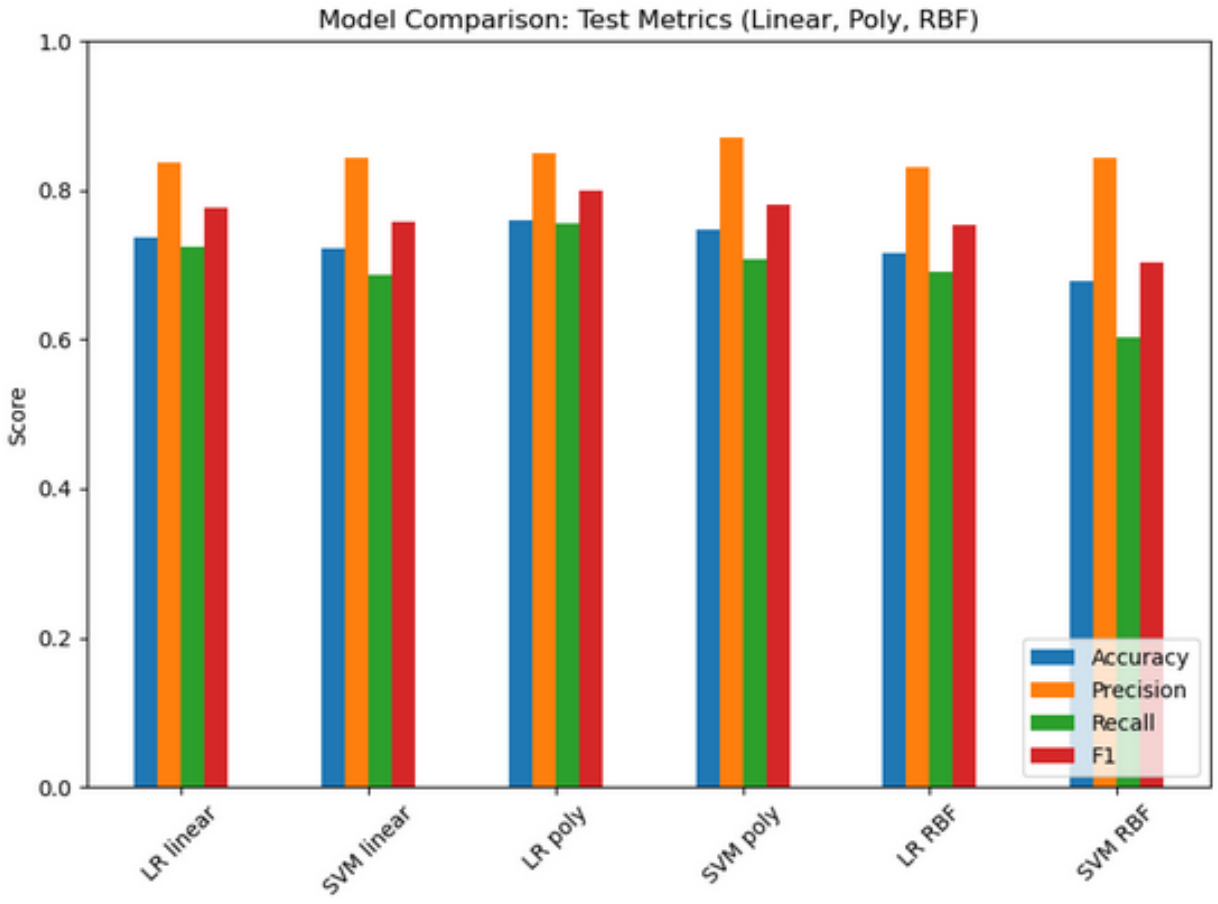
- **Accuracy gains:**
  - LR: 0.590 → 0.761 (poly) → 0.716 (RBF)
  - SVM: 0.636 → 0.751 (poly) → 0.664 (RBF)

- **Compute complexity:**
  - Linear: $O(n \cdot p)$
  - Poly-explicit: $O(n \cdot P)$, $P \approx p^2/2$
  - Kernel (RBF): $O(n^2)$ to build Gram or $O(n \cdot D)$ for RFF

- **Memory trade-off:**
  - Explicit $\phi(x)$: store X_poly ($n \times P$)
  - Kernel matrix: store $n \times n$ Gram
  - RFF: store X_rff ($n \times D$) with $D \ll n$

Model Comparison – Test Metrics (Linear, Poly, RBF)

on test set	Accuracy	Precision	Recall	F1 score
LR linear	0.73	0.83	0.72	0.77
SVM Linear	0.72	0.84	0.68	0.75
LR poly	0.76	0.84	0.75	0.80
SVM poly	0.74	0.86	0.70	0.78
LR RBF	0.71	0.83	0.69	0.75
SVM RBF	0.67	0.84	0.60	0.70



Model Comparison – Test Metrics (Linear, Poly, RBF)

Highest overall accuracy:

- Logistic Regression with degree-2 polynomial features (≈ 0.76), closely followed by SVM poly (≈ 0.75)

Recall:

- Recall (sensitivity) peaks for LR poly (≈ 0.76) and dips for SVM RBF (≈ 0.60), showing some RBF models miss more “good” wines

Precision:

- Presicion is strongest for the two SVM variants on polynomial data (≈ 0.88), indicating very few false-positives

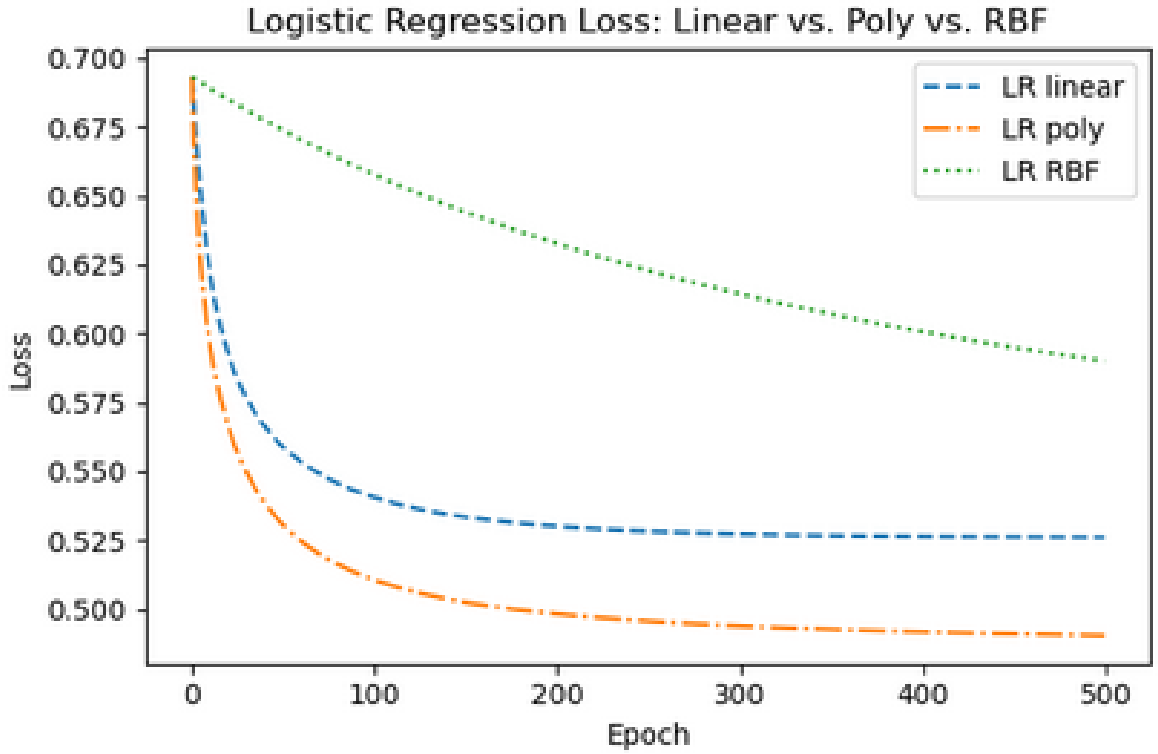
F1-score:

- F1-score balances recall and precision: best for LR poly (~ 0.80), worst for SVM RBF (~ 0.70)

- Polynomial expansion has the most balanced gains across metrics
- RBF offers no clear advantage here

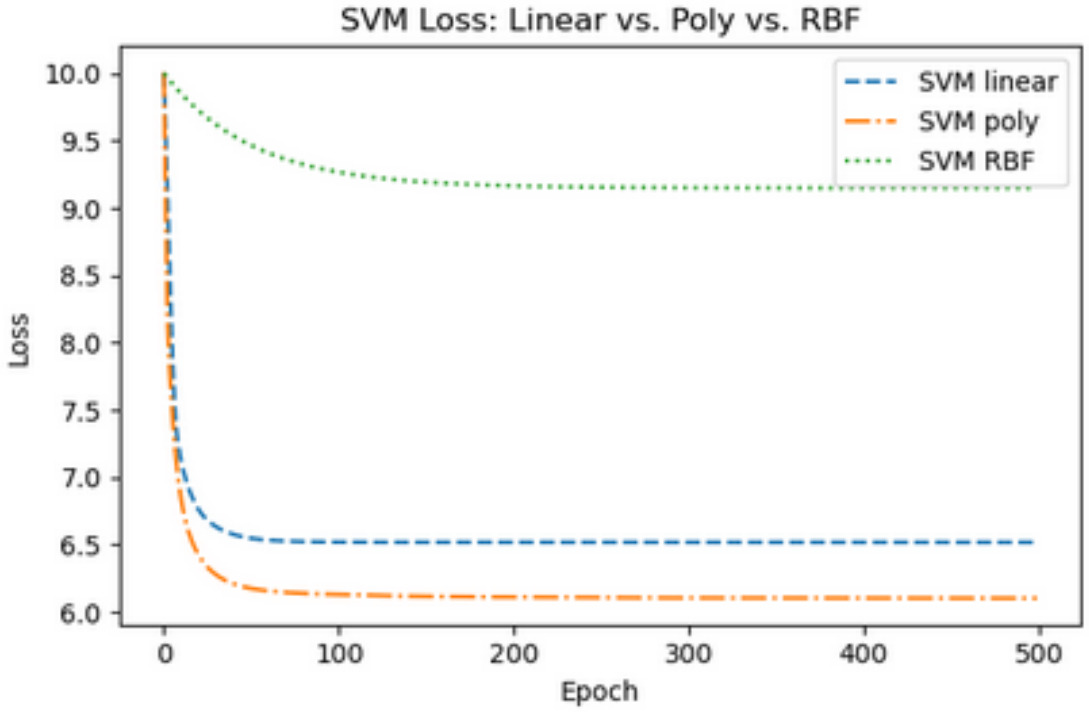
Logistic Regression Loss

- Poly LR converges fastest and to the lowest loss (~ 0.49), probably because of its greater capacity
- Linear LR flattens around loss = 0.53
- RBF LR converges more slowly and remains at a higher final loss (~ 0.58), it's under-fitting relative to poly.
- Degree-2 features give the best trade-off between model complexity and convergence
- the random-feature RBF approximation isn't capturing enough nonlinearity.

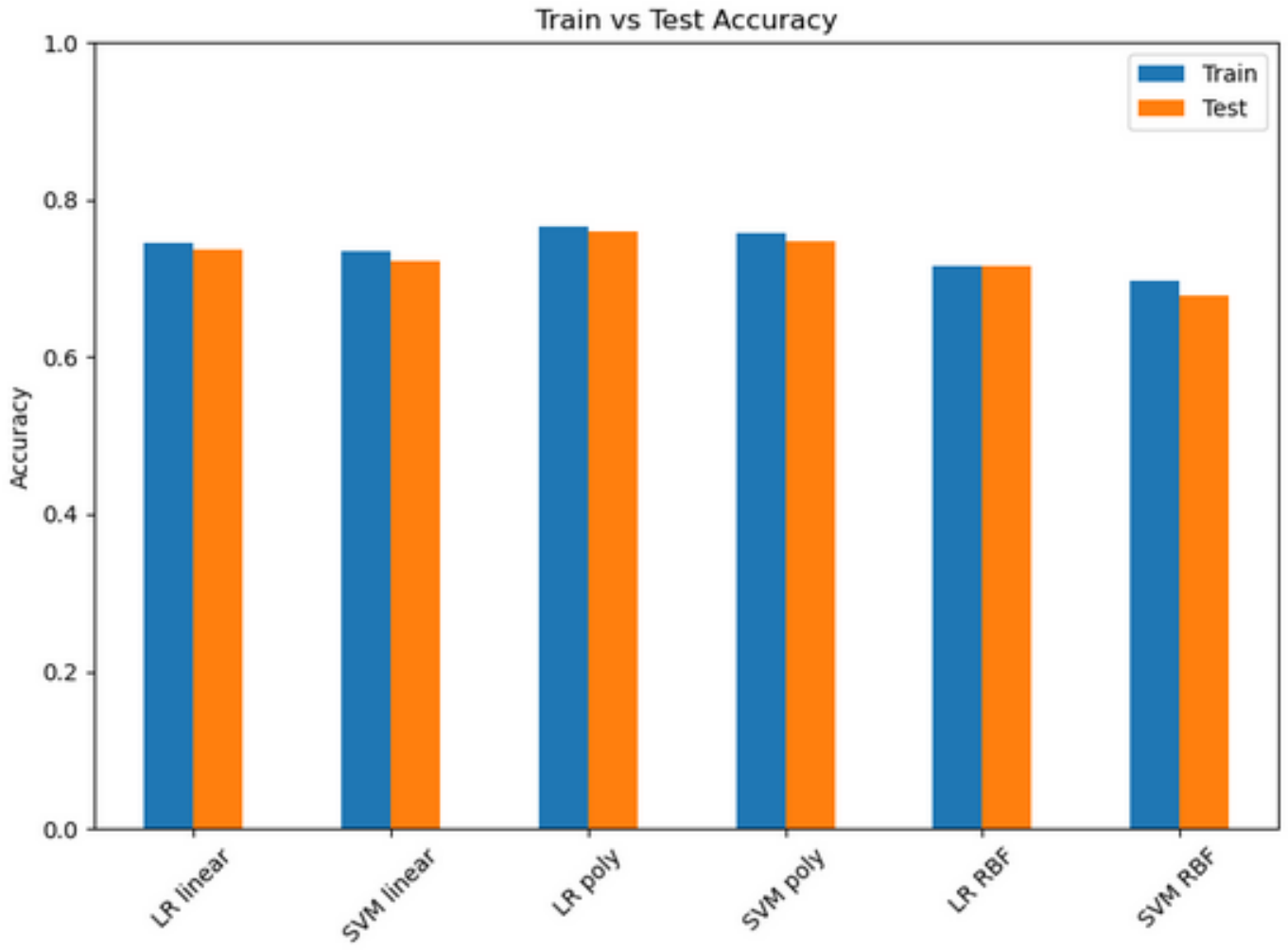


Support Vector Machine Loss

- Poly SVM again achieves the lowest hinge-loss (~ 6.1) and fastest convergence
- Linear SVM stabilizes around loss = 6.4
- RBF SVM begins very high (≈ 10) but drops to ≈ 6.3 but is still above poly
- Polynomial kernel gives better margin separability here
- the RBF approximation did not outperform explicit degree-2 mapping



Train vs Test Accuracy



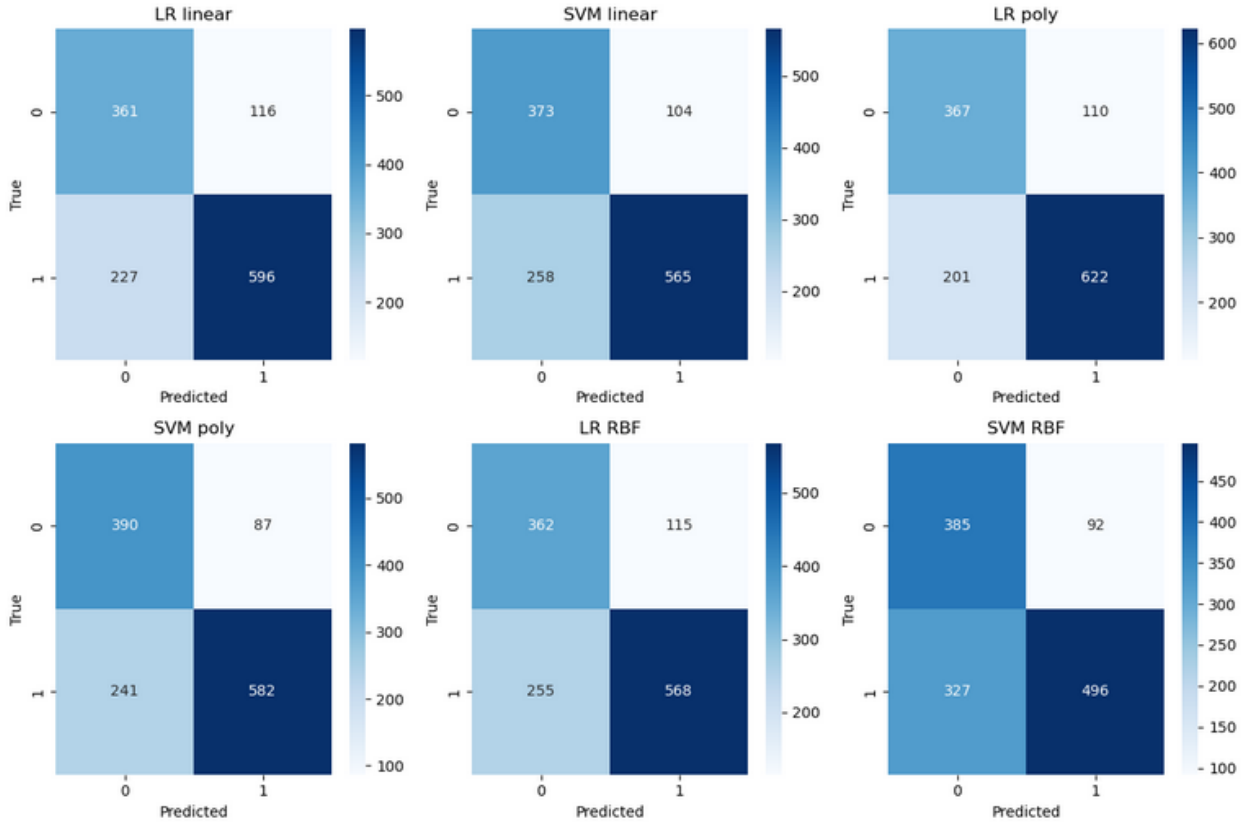
Overfitting or Underfitting

- Small gaps between train and test accuracies ($\Delta \lesssim 0.02$) for all models, indicating minimal over-fitting.
- Best generalization: LR poly (train ≈ 0.77, test ≈ 0.76).
- Lowest both train and test: SVM RBF (train ≈ 0.70, test ≈ 0.68).
- Even the most flexible models (poly features) generalize well
- no drastic over or underfitting observed

Model	Train	Test
LR linear	0.74	0.73
SVM linear	0.73	0.72
LR poly	0.76	0.76
SVM poly	0.75	0.74
LR RBF	0.71	0.71
SVM RBF	0.69	0.67

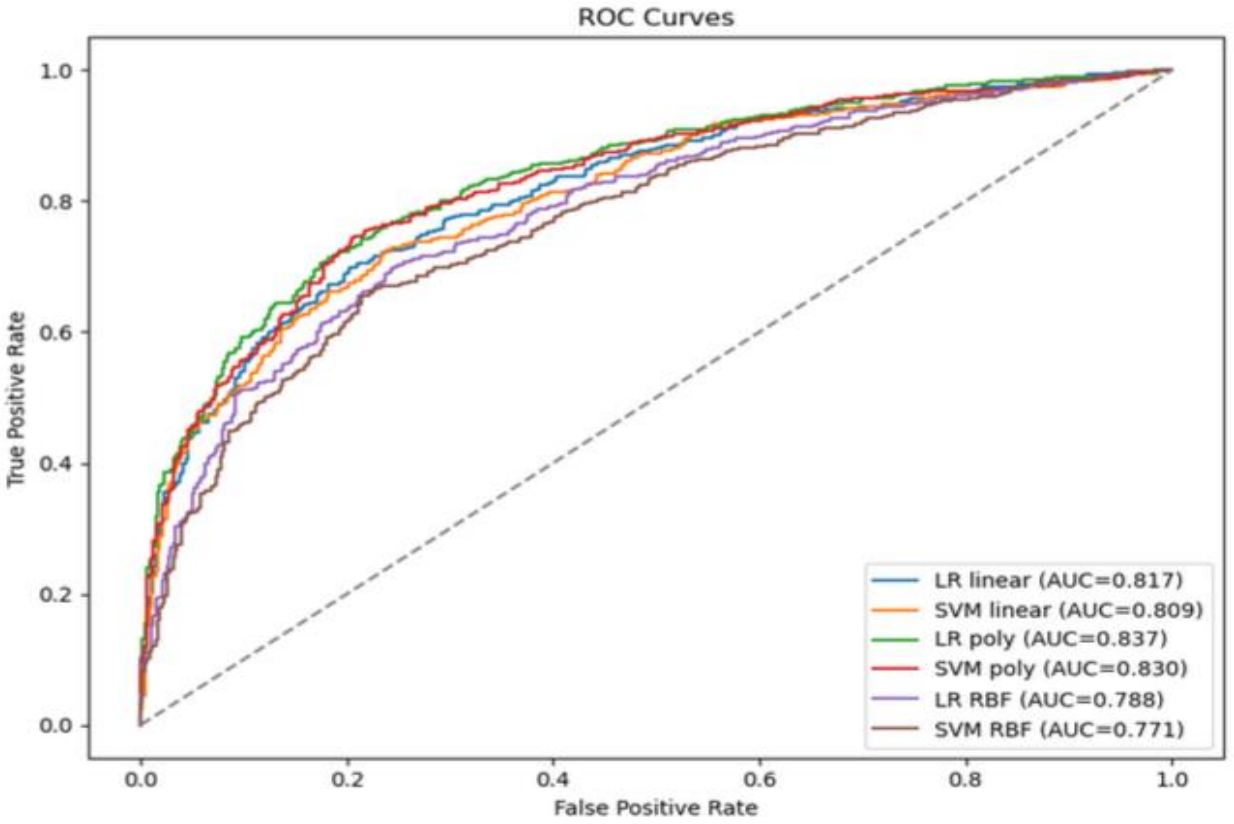
Confusion Matrices

- **Linear vs. Polynomial vs. RBF:**
  - LR linear makes 116 false-positives and 227 false-negatives
  - SVM linear slightly reduces false-positives (104) but increases false-negatives (258)
- **Polynomial Expansion yields the best balance:**
  - LR poly cuts false-negatives to 201 while keeping false-positives at 110
  - SVM poly further reduces false-positives to 87 with only 241 false-negatives
- **RBF Approximation under-performs here:**
  - LR RBF has 115 false-positives and 255 false-negatives
  - SVM RBF is most conservative (FP=92) but misses 327 positives (FN), costing recall
- **Degree-2 polynomial** features most effectively reduce both types ofclassification errors in this binary task



ROC Curves

- **AUC ranking** (best→worst):
  1. Logistic Regression (poly): AUC ≈ 0.837
  2. SVM (poly): AUC ≈ 0.830
  3. Logistic Regression (linear): AUC ≈ 0.817
  4. SVM (linear): AUC ≈ 0.809
  5. Logistic Regression (RBF): AUC ≈ 0.788
  6. SVM (RBF): AUC ≈ 0.771
- Polynomial kernels consistently dominate their linear counterparts at almost all false-positive rates.
- RBF approximation curves lie below both linear and poly, showing weaker separability.
- Explicit degree-2 mapping delivers the strongest trade-off between TPR and FPR; the Random Fourier RBF approximation fails to surpass even the simple linear model in ROC performance.



Conclusion

- Explicit degree-2 polynomial features had the strongest performance across metrics and ROC/AUC
- Approximate RBF kernels (Random Fourier) did not outperform linear or polynomial variants
- Systematic 5-fold CV hyperparameter tuning was critical to find optimal α, λ, C, and γ
- Comprehensive evaluation (confusion matrices, ROC/AUC, learning curves) confirmed minimal overfitting and robust generalization
- Final reslut: using Logistic Regression with polynomial features gives the most interpretable, highly accurate wine quality classification

Declaration

- *I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.*