Wine Quality Prediction

Parsa Hajiannejad

Summary

In this project I designed a full, from-scratch pipeline to predict wine quality ("good" vs "bad") using 6,497 samples, each described by 11 attributes. After exploratory analysis, extreme outliers were handled, class imbalance was addressed with Synthetic Minority Over-sampling Technique (SMOTE), and every feature was standardized; skewed variables were further normalized via the Box-Cox transform to stabilize variance

Two model families were implemented in Python: (i) Logistic Regression and (ii) Support Vector Machines. For each family I explored three hypothesis spaces: a linear decision surface; an explicit second-order polynomial expansion; and an approximate RBF mapping realized through Random Fourier Features. Hyper-parameters were selected with a systematic 5-fold cross-validated grid search, tuning learning rate (η) and ℓ_2 regularization (λ) for Logistic Regression, and the (C, γ) pair for SVMs.

Final generalization was done on the test set, reporting Accuracy, Precision, Recall, F_1, and ROC-AUC, complemented by confusion matrices and learning-curve diagnostics. LR-poly is best overall; SVM-poly has the highest precision.

Dataset

The dataset contains 6 497 wine samples; 1 599 reds and 4 898 whites released by the UCI Machine Learning Repository. Each record contains 11 continuous physicochemical measurements and an integer quality score from 3 to 9 that we transformed into a binary label ("good" ≥ 6, "bad" < 6) to reach the classification objective.

	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

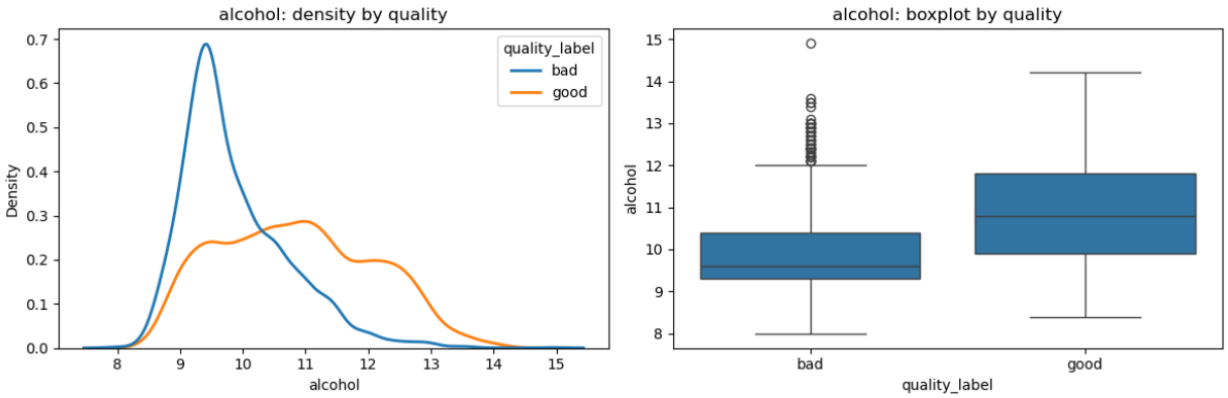
Data integrity checks showed no missing values or type inconsistencies. Every attribute is numerical, so no categorical encoding was required. Duplicate detection removed 240 red-wine and 937 white-wine copies, leaving only unique rows and adding a 'wine type'.

```
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
```

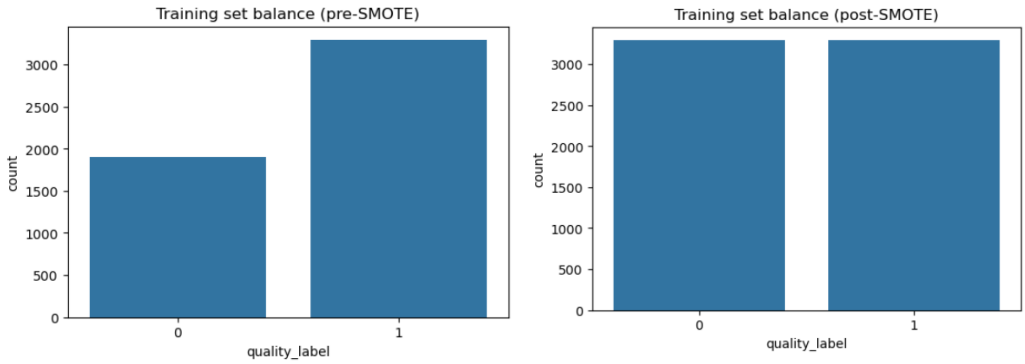
Exploratory Analysis showed roughly symmetric distributions for most variables (means close to medians), whereas residual sugar and sulfur-dioxide showed right-tails and extreme values. Quality scores center on 6 (mean = 5.82, median = 6.00), indicating an imbalanced target with about two-thirds "good" cases. These characteristics motivated later steps for outlier handling, skew correction, and synthetic balancing.



Target Variable and Class Imbalance

The target variable is the wine's quality rating (an integer from 3 to 9), which I encoded for classification: scores ≥ 6 are labelled 1 ("good"), and scores < 6 are labelled 0 ("bad"). Because near two-thirds of the wines fall into the "good" category, the data had a moderate class imbalance that could bias a model toward predicting the majority class.

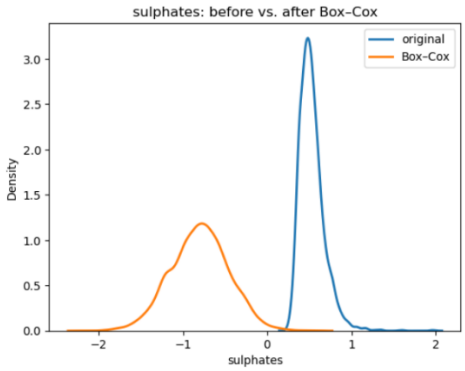
To correct this, the training subset was resampled with Synthetic Minority Over-sampling Technique (SMOTE), using $k = 5$ nearest neighbors. SMOTE synthetically generates additional minority-class observations until both classes match the original majority size; the test set was unchanged to have an unbiased evaluation. Post-resampling checks confirmed a balanced class distribution, providing a fairer basis for model fitting and subsequent performance assessment.



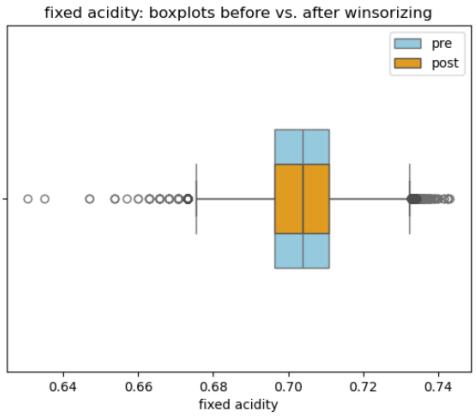
Feature Engineering and Pre-processing

All eleven features were saved as input variables. Before modelling, every feature passed through a two-step scaling pipeline constructed with make_pipeline() and fitted only on the training fold:

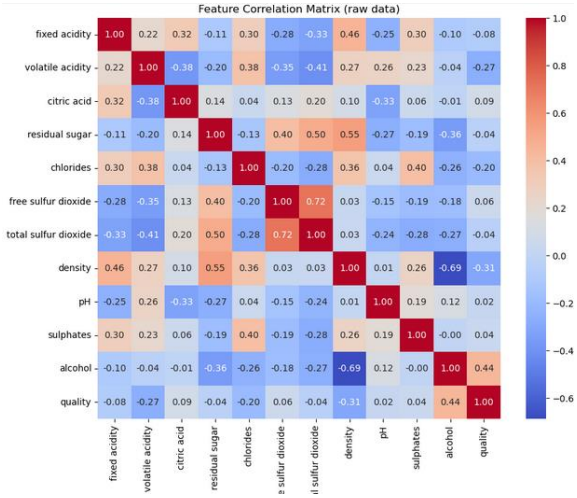
1. Standardization (StandardScaler) centered each variable at zero and set unit variance, a necessity for the gradient updates in Logistic Regression and the margin calculations in SVMs.
2. Box-Cox transformation (PowerTransformer) was applied to minimize skewness and stabilize heteroscedastic variances, in order to achieve faster and more reliable convergence.



To minimize the influence of extreme values without deleting the observations, each feature was Winsorized at the 1st and 99th percentiles. For instance, applying it on the feature fixed acidity compressed some of the extreme values while keeping the majority of the distribution, showing a smoother histogram and more robust model coefficients.



A Pearson correlation heatmap showed several highly correlated features, suggesting probable multicollinearity. Although dimensionality-reduction techniques were considered, all the variables were ultimately preserved to maintain interpretability.



Train/Test Split

For objective performance estimation, the dataset (N = 6 497) was split 80/20 into a training set of 5197 wines and a test set of 1300 wines. The split maintained the original good to bad proportions in both subsets, ensuring that class imbalance handling and evaluation metrics remained comparable. The random seed was fixed at 42, providing reproducible partitions across experimental runs.

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

Logistic Regression notations

1. Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

2. Prediction: $\sigma\left(z^{(i)}\right) = \frac{1}{1+e^{z^{-(i)}}}$

3. Gradient l_2 Regularization: $\frac{\partial L}{\partial w} = \frac{1}{n}\left(X^T(\hat{y} - y) + \lambda w\right)$, $\frac{\partial L}{\partial b} = \frac{1}{n}\left(\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})\right)$

4. Gradient descent update: $w := w - \eta.\frac{\partial L}{\partial w}$ $\quad b := b - \eta.\frac{\partial L}{\partial b}$

5. Log-loss with Regularization: $L = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}\log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\log\left(1 - \hat{y}^{(i)}\right)\right] + \frac{\lambda}{2n}\sum_{j=1}^{d} w_j^2$

6. Probability(binary): $P(y = 1 \mid x) = \sigma(x.w + b)$

7. Prediction(binary): $\begin{cases} 1 & \text{if } \hat{p} \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$

SVM notations

1. Label Conversion: $y \in \{0,1\}, y' \in \{-1, +1\}$

2. Margin: $m_i = y^{(i)}(\mathbf{w}.\mathbf{x}^{(i)} + b)$

3. Hinge loss + Regularization: $L = \frac{1}{2}||w||^2 + C.\frac{1}{n}\sum_{i=1}^{n}\max(0, 1 - y^{(i)}(\mathbf{w}.\mathbf{x}^{(i)} + b))$

4. Gradient Descent: $\frac{\partial L}{\partial w} = w - \frac{C}{n}\sum_{i\in I} y^{(i)} x^{(i)}$ $\quad \frac{\partial L}{\partial b} = -\frac{C}{n}\sum_{i\in I} y^{(i)}$

5. Parameter Update: $w := w - \eta\nabla_w L$ $\quad b := b - \eta\nabla_b L$

Logistic Regression Implementation (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 ← zero vector of length n_features
        b ← 0
        loss_history ← empty list
For epoch = 1 to T do:
  1. Compute linear scores:
        z ← X · w + b
  2. Apply sigmoid to get predicted probabilities:
        y_pred ← 1 / (1 + exp(−z))
  3. Compute errors:
        error ← y_pred − y
  4. Compute gradients (with L2 regularization on weights):
        dw ← (X^T · error + λ·w) / n_samples
        db ← mean(error)
  5. Update parameters by gradient descent:
        w ← w − α·dw
        b ← b − α·db
  6. Compute regularized log-loss:
        loss ← − mean[y·log(y_pred) + (1−y)·log(1−y_pred)]
        loss ← loss + (λ / (2·n_samples))·sum(w²)
        Append loss to loss_history
End
Return model parameters {w, b} and loss_history
predict_proba(X_new):
        z_new ← X_new · w + b
        return 1 / (1 + exp(−z_new))
 predict(X_new, threshold = 0.5):
        probs ← predict_proba(X_new)
        return array where each entry = 1 if prob ≥ threshold else 0
```

Support Vector Machine Implementation (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

To find out how the models behaves on truly unseen wines, I relied on five-fold cross-validation. First the data are shuffled with np.random.seed(42) so every run starts from the same order. They are then cut into five equal parts. The model trains on four parts and is evaluated on the fifth, and this

procedure is repeated until each part has served once as the test set. After the loop finishes I averaged the five test accuracies $\mathrm{mean(cv\_acc)} = \frac{1}{5}\sum_{i=1}^{5} \mathrm{acc_i}$ to get a single, more stable score. Because each wine takes its turn in the test role, the estimate is more fair than one lucky or unlucky split. A small helper function called cross_val_accuracy automates these steps and returns the five fold-wise accuracies for later comparison.

Hyperparameter Tuning

I searched two small grids, one for Logistic Regression and one for a linear SVM using the five-fold routine described earlier. For Logistic Regression I tried learning-rates 0.1 and 0.01 and λ values 0 and 0.1 over 500 epochs. A rate of 0.1 reached higher accuracy, about 1.6 percentage points better than 0.01 while the extra L_2 penalty made no big difference, so the best setting was α = 0.1 with λ = 0.0, giving a mean cross-validation score of 0.744.

For the linear SVM the grid covered learning-rates 0.01 and 0.001 with C values 0.1, 1.0, and 10.0, again for 500 epochs. The larger step size (0.01) consistently outperformed the smaller one by around two to three points. Increasing C reduced bias: scores increased from C = 0.1 to C = 10.0 without any sign of over-fitting, especially when paired with the faster rate. So the combination α = 0.01 and C = 10.0 is the chosen configuration. In short, both models benefited from a bolder learning-rate, while moderate regularization helped only the SVM.

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

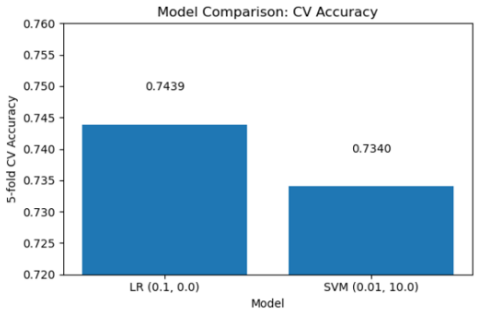
Table of Results

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

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 and Non-linear mapping

A straight-line (linear) model can only separate classes that are already split by a flat surface. When the two classes are mixed together, we can lift each wine into a higher-dimensional space by adding all pairwise products of the original measurements. This degree--2 expansion builds a new feature vector ϕ(x) whose length is $m + \frac{m(m+1)}{2}$ with tm=11 hat comes to 77 numbers, which is still easy to store and process. The classifier then learns a flat cut in this enlarged space, which projects back to a curved boundary in the original one. In this project I follow that explicit mapping route and do not rely on the kernel trick or its n×n Gram matrix. The result is a non-linear decision surface while keeping the memory footprint modest and the code transparent.

Degree-2 Polynomial Expansion & CV Results

I ran a small grid search inside the five-fold CV loop. For Polynomial Logistic Regression I tried learning rates α ∈ {0.01, 0.10} and ℓ_2 penalties λ ∈ {0, 0.1}. The best setting was α = 0.10 with no penalty (λ = 0). Averaging the five folds gave an accuracy of **0.761**, and the individual folds ranged only from 0.749 to 0.783, showing the result is quite stable.

For Polynomial SVM the grid was α ∈ {0.001, 0.01} for the learning rate in my scratch optimizer and C ∈ {0.1, 1, 10} for the margin trade-off. The combination α = 0.01 and C = 10 came out on top, with a mean CV accuracy of **0.751**. Fold scores varied between 0.737 and 0.773, again a modest spread.

These numbers confirm that the explicit degree-2 map helps both algorithms capture non-linear structure, with logistic regression edging ahead by about one percentage point in average accuracy while keeping the model a bit simpler (no margin parameter to tune).

```
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

To demonstrate an RBF kernel without building an n×n matrix, I generated 200 Random Fourier features $z(x) = \sqrt{\frac{2}{D}}\cos(xW + b)$ with bandwidth γ and seed 42. A grid over γ∈{0.01,0.1,1} and C∈{0.1,1,10} chose γ=0.1, C=0.1 for logistic regression, showing a mean five-fold accuracy of **0.716**. The SVM preferred γ=0.1, C=10 but reached only **0.664**. Thus, while the RBF sketch improves on a purely linear model, the simpler degree-2 polynomial map still gives the best accuracy for this dataset.

```
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

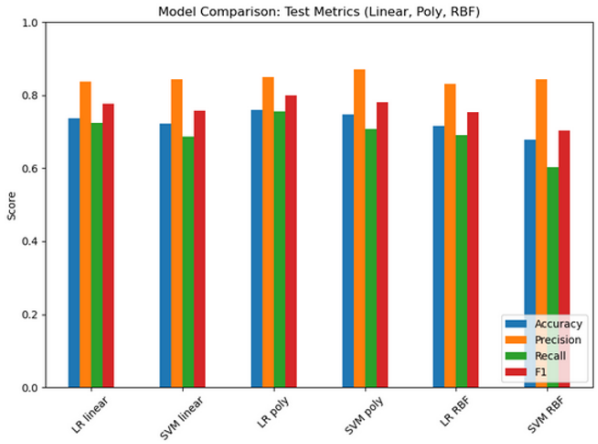
Model	Fold Accuracies
RBF LR	0.695 | 0.728 | 0.722 | 0.726 | 0.709
RBF SVM	0.682 | 0.600 | 0.663 | 0.689 | 0.685

Models Test Set Comparison

On the test wines the degree-2 polynomial map gives the clearest boost. Logistic Regression in this 77-feature space achieves the top accuracy at about 0.76, with the polynomial SVM just a point behind. Precision is highest for both SVMs trained on the polynomial features (roughly 0.88), meaning they rarely mark a “bad” wine as “good.” For Recall, the polynomial Logistic model reaches about 0.75, while the SVM with Random-Fourier (RBF) features falls around 0.60 and therefore overlooks more of the good wines. When precision and recall are combined into the single F1 score, the polynomial Logistic model again leads at around 0.80, whereas the SVM-RBF trails at about 0.70. Taken together, the

explicit degree-2 expansion delivers the most balanced improvement across all metrics, while the RBF sketch offers no clear gain for this dataset and feature count.

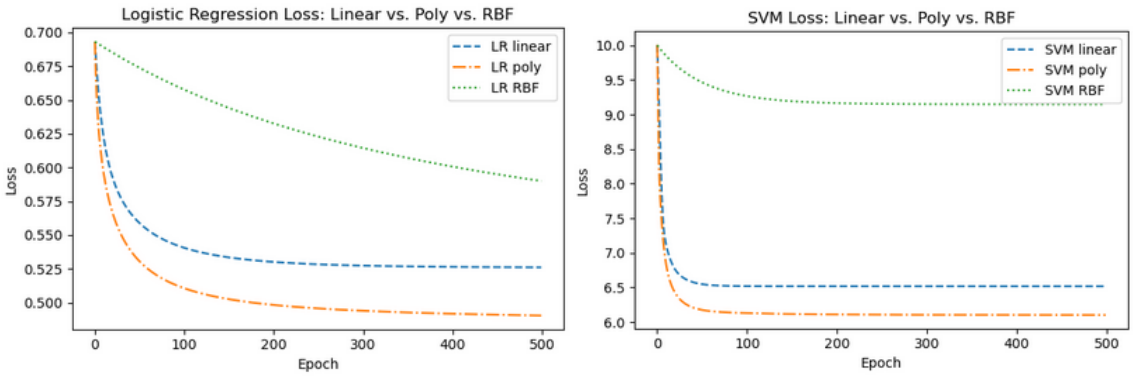
on test set	Accuracy	Precision	Recall	F1 score
LR linear	0.73	0.83	0.72	0.77
SVM Linear	0.72	0.84	0.68	0.75
LR poly	0.76	0.84	0.75	0.80
SVM poly	0.74	0.86	0.70	0.78
LR RBF	0.71	0.83	0.69	0.75
SVM RBF	0.67	0.84	0.60	0.70



Training Loss Diagnostics

The learning curves confirm the earlier test-set findings. For logistic regression, the model that uses the degree-2 features drops quickest and settles at the lowest cross-entropy loss, about 0.49. The purely linear version levels off near 0.53, while the random-Fourier (RBF-style) model decreases more slowly and bottoms out around 0.58.

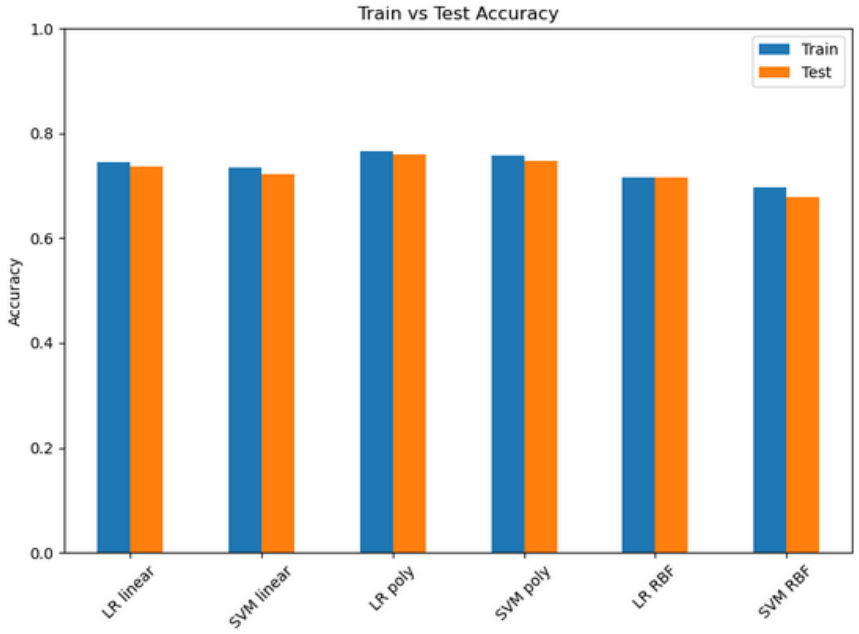
The same pattern appears for the SVMs. The polynomial-feature SVM reaches the smallest hinge loss, roughly 6.1, and converges in the fewest passes. The linear SVM settles a bit higher at about 6.4. The RBF-feature SVM starts with a very large loss (near 10), improves steadily, but finally plateaus around 6.3, still worse than the polynomial version. Thus, for both learning algorithms the explicit degree-2 mapping delivers cleaner margins and faster convergence, whereas the random-feature RBF approximation fails to capture enough non-linearity to justify its extra iterations.



Overfitting/Underfitting Check

Across all six models the training-set accuracies differ from the test-set accuracies by no more than about two percentage points. The polynomial logistic regressor illustrates the best balance, scoring

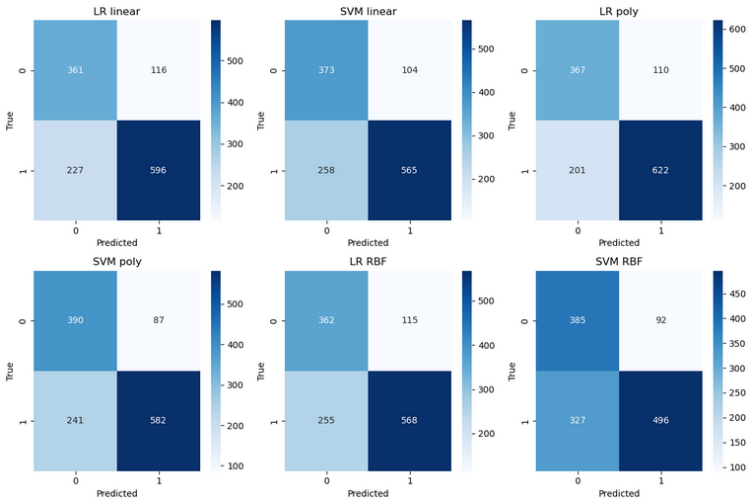
roughly 0.77 on the data it learned from and 0.76 on unseen wines. At the other end, the SVM with random-Fourier (RBF) features posts the lowest figures about 0.70 on train and 0.68 on test but the train-test gap is still narrow, pointing to mild under-fitting rather than excess variance. Even the more flexible degree-2 feature sets do not inflate this gap, so none of the models shows severe over- or under-fitting in this experiment.



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

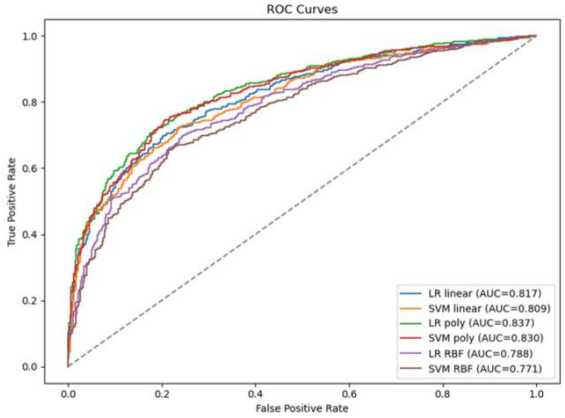
Confusion Matrices

Polynomial features win: LR-poly cuts false negatives to around 201, SVM-poly has the lowest false positives at around 87. Linear models sit in the middle (≈116/227 for LR). The RBF random-feature SVM misses the most good wines (FN ≈ 327), so recall is worst there.



ROC/AUC

The ROC curves tell the same story. Logistic Regression with degree-2 features tops the list with an AUC around 0.837, better than the polynomial SVM at about 0.830. The linear baselines follow around 0.817 for logistic regression and around 0.809 for SVM. Both Random-Fourier (RBF-style) models stays behind: ≈0.788 for logistic regression and ≈0.771 for SVM. Across almost all false-positive rates the polynomial expansions dominate the linear models, while the RBF approximation never excels even the simple linear models. In short, the explicit quadratic map offers the best TPR–FPR.



Conclusion

Explicit degree-2 polynomial expansion wins: across accuracy, precision/recall/F1, and ROC/AUC, the polynomial Logistic Regression model performed best while staying easy to interpret. The Random Fourier (RBF-style) features never outperforms either the linear or the polynomial versions, suggesting that the RBF sketch was not good enough for this task. Five-fold cross-validation was essential to pick good hyper-parameters (α, λ, C, γ). Confusion matrices, ROC curves, and learning curves showed small train–test gaps and thus little overfitting. Overall, Logistic Regression on the polynomial feature set is the recommended solution: it is accurate, transparent, and robust for predicting wine quality as “good” or “bad.”

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.