Wine Quality Prediction

(Machine Learning Classification Based on SVM and Logistic Regression with Optional Kernelization)

Abdelrahman Mohamed Aly Sobh* 30061A

Statistical Methods for Machine Learning
A.Y. 2024/2025

Abstract

Implementation of two classic binary-classification algorithms from scratch: Support Vector Machine (SVM) and Logistic Regression (LR), with the option to apply non-linear kernelization.

Having the wine datasets [wine datasets]. We need to predict whether wines are
$$\begin{cases} \text{good} & \text{if quality} \geq 6 \\ \text{bad} & \text{if quality} < 6 \end{cases}$$
considering white and red wines simultaneously.

In order to obtain a satisfacting practical experiment, we have to follow statistical and machine learning practices for data analysis and preprocessing, model training, hyperparameter tuning, and evaluation.

*https://github.com/Abudo-S/WineQualityPrediction.

1. Index

2. Data Exploration and General Observations

By reading the information of both datasets, we get that:
the white-wine dataset has thes 12 columns (including the target label column "quality"): [fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide , density, pH, sulphates, alcohol, quality] with 1599 data examples. Meanwhile the red-wine dataset has the same columns with 4898 data examples.

So we can decduce that:

1. The red-wine dataset as well as the white-wine dataset have the same features labeled with the same target feature "quality".

2. All feature values are numeric with no NaN values.

3. Both datasets contain some duplicated instances.

We'd need to combine both datasets in order to develop a universal model for both of red and white wines, by introducing a new feature 'wine_type' valorized with 1: for red wine instance, 0: for white wine instance. In fact, introducing a new feature for the wine type the learning model would be to build relationships between the wine type and the other features. By combining both datasets, we're increasing the training-set volume which gives the possibility to the learning model to see further records, so it reduces the risk of overfitting.

So a combined sample of the labeled dataset becomes as the following:

Table 2.1: Wine Quality Data Sample

Fix.A.	Vol.A.	Citric.A.	Res.Sugar	Chlorides	Free.S.D	Total.S.D.	Density	pH	Sulphates	Alcohol	Quality	Wine Type
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	1
7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5	1
11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6	1
7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5	1

2.1. Observations on Data Duplication

Duplicated instances might lead to data leakage in which after dataset splitting into training and test sets, there might be identical records in both splitted sets. The model also becomes too specialized to the training data, including the duplicated instances, which will cause the problem of overfitting. Furthermore, there'd be additional elaboration cost of redundant data.

2.2. Observation on Dataset Dimenstionality

Since we have 5320 examples for 12 features (excluding the target label "quality"), which gives us the following ratio of samples to features (443:1). The ratio we have is considered a high ratio which gives us several examples per features avoiding the curse of dimensionality. The curse of dimensionality commonly verifies when we don't have enough examples that describe the dataset w.r.t. the number of dimensions. Having a high ratio of samples to features implies that we might not need to apply dimensionality reduction technique (ex. PCA).

2.3. Observations on data distribution within quality's values

As we can notice, the range of values for our target column "quality" is [3:9] inclusive. But we have a clear imbalance within quality's values "less data examples for extremely low or extremely high values" (like quality 3, 4, 8, or 9), meanwhile for middle values, we apparently have the majority of examples.

When we convert the continuous 'quality' score into a binary classification problem

$$\begin{cases} \text{good} & \text{if quality} \geq 6 \\ \text{bad} & \text{if quality} < 6 \end{cases}$$

We're essentially creating two classes for the label quality:

- Bad [-1] class: Quality scores of 3, 4, 5. (represent 37% of the whole dataset).
- Good [1] class: Quality scores of 6, 7, 8, 9. (represent 63% of the whole dataset).

We can clearly notice the imbalance within the data distribution over our two classes. Imbalanced data tends to be biased towards the majority class because it sees more examples of it. The model will struggle to correctly identify instances of the minority class.

In fig. 2.3 we can notice that the good datapoints are overwhelming the date distribution, especially within the white wine subset.

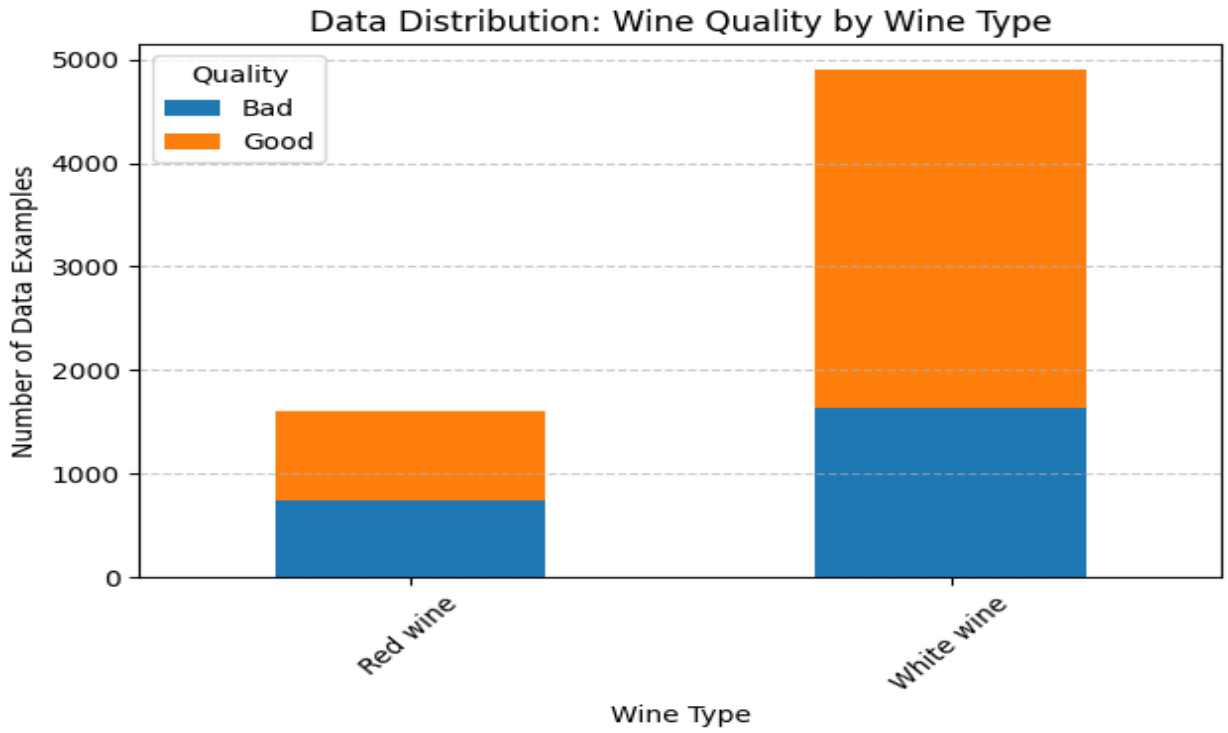

Figure 2.1: The distribution of labeled data examples per wine type

In case of SVM:

In an imbalanced dataset, the majority class has many more data points. Consequently, the support vectors that define the decision boundary are dominated by instances from the majority class. Since the SVM algorithm's primary objective is to maximize the margin. The decision boundary (hyperplane) gets pushed towards the minority class; therefore, many instances of the minority class could be incorrectly classified as the majority class.

In case of Logistic Regression:

The input space corresponding to these extremely low or extremely high quality scores will have very few data points, creating sparse regions. The model will learn to predict higher probabilities for the more frequent classes, assigning heigher weights to them. The decision boundary will be positioned to minimize overall prediction errors. Since errors on the majority class contribute more to the total loss, the model will prioritize correctly classifying the majority class instances, potentially pushing the boundary away from the minority class.

Possible solutions:

- Data augumentation: Oversampling of the minority class through creating new relevant synthetic (not duplicated) samples, in order to enhance the presence of the minor class in the whole dataset.
- Class weighting: Adding higher weight multiplier ($ex.[2:10]$) to the minor class and ($ex.0$) to the major class. The weight multiplier is used to give significant bias/weight vector updates in case of the minor class datapoints, so it would mitigate the overwhelming of the major class updates. A common weight formula is:

$$w_j = \frac{\text{total number of samples}}{\text{number of classes} \times \text{number of samples in class } j}$$

3. Data Preprocessing

3.1. Observations on feature scaling

After observing the values of the feature space, having:

Table 3.1: Minimum, Mean and Maximum Values of the Feature Space

Feature	Minimum	Mean	Maximum
fixed acidity	3.8	7.2153070648	15.9
volatile acidity	0.08	0.3396659997	1.58
citric acid	0.0	0.3186332153	1.66
residual sugar	0.6	5.4432353394	65.8
chlorides	0.009	0.0560338618	0.611
free sulfur dioxide	1.0	30.5253193782	289.0
total sulfur dioxide	6.0	115.7445744190	440.0
density	0.98711	0.9946966338	1.03898
pH	2.72	3.2185008465	4.01
sulphates	0.22	0.5312682777	2.0
alcohol	8.0	10.4918008312	14.9
wine_type	-1.0	-0.5077728182	1.0

Some features (ex. residual sugar, free sulfur dioxide, total sulfur dioxide, ...) which have high difference between the minumum and maximum values, will need to be standardized on the same scale. Especially in case of models that aim to find an optimal hyperplane (ex. SVM) that maximizes the margin between different classes. The calculation of this margin and the positioning of the hyperplane rely on the distances between data points in the feature space.

Meanwhile the values of the feature space for good/bad wine qualities are:

Table 3.2: Good Quality Wine: Minimum, Mean and Maximum Values of the Feature Space

Feature	Minimum	Mean	Maximum
fixed acidity	3.8	7.1488329686	15.6
volatile acidity	0.08	0.3061962071	1.04
citric acid	0.0	0.3270119134	1.66
residual sugar	0.7	5.3255774374	65.8
chlorides	0.012	0.0511823487	0.415
free sulfur dioxide	1.0	31.1309263311	112.0
total sulfur dioxide	6.0	113.6970581084	294.0
density	0.98711	0.9940828410	1.03898
pH	2.72	3.2208071967	4.01
sulphates	0.22	0.5353245806	1.95
alcohol	8.4	10.8501580355	14.2
wine_type	-1.0	-0.5842450766	1.0

Table 3.3: Bad Quality Wine: Minimum, Mean and Maximum Values of the Feature Space

Feature	Minimum	Mean	Maximum
fixed acidity	4.2	7.3299916107	15.9
volatile acidity	0.1	0.3974098154	1.58
citric acid	0.0	0.3041778523	1.0
residual sugar	0.6	5.6462248322	23.5
chlorides	0.009	0.0644039430	0.611
free sulfur dioxide	2.0	29.4804949664	289.0
total sulfur dioxide	6.0	119.2770553691	440.0
density	0.98722	0.9957555810	1.00315
pH	2.74	3.2145218121	3.9
sulphates	0.25	0.5242701342	2.0
alcohol	8.0	9.8735444631	14.9
wine_type	-1.0	-0.3758389262	1.0

3.2. Standard Scaling

standard_scaler.fit_transform() is used to compute the mean (μ) and standard deviation (σ) for each feature on the training data and then to rescale the training data. The testing data are directly rescaled through standard_scaler.transform() using the parameters computed on the training data. Generally, We should avoid scaling categorical features, they have discrete values and by scaling makes them, their values become continuous, which can mislead the learning models by lossing the interpretability and the nature of the target feature. Note that dimension scaling is always applied after dataset splitting, in order to avoid data leakage from training set to test set.

3.3. Should We Apply PCA?

To extract only important features and reduce the dimensionality, therefore reducing the traing cost. It could be needed, if we notice overfitting caused by irrelevant features. Generally, in our case since we've already aforementioned [ref.2.2] that in our case the samples-to-features ratio is high, we don't need to reduce the number of dimensions. But just for an experiment we can try it. (it might be useful in case of further noticed overfitting!)

4. Support Vector Machine (SVM)

Support vector machine aims to find an optimal hyperplane that best separates different classes in a dataset. The optimal hyperplane is the one that has the maximizied margin to the nearest training data points (support vectors) of any class. For linearly separable data, there might be many possible hyperplanes that can separate the classes. And it'll be quite feasibile to find the optimal hyperplane. For non-linearly separable date, we can introduce a generic approach which works also in case of linearity by exploiting the unconstrainted weights/bias update which uses:

- Regularization parameter λ: A higher value of λ increases the cost of misclassifications, forcing the algorithm to try harder to classify all training points correctly, even if it means a smaller margin. This can lead to a smaller margin and potentially overfitting if λ is too large.

  A lower value of λ decreases the cost of misclassifications, allowing the algorithm to have a larger margin, even if it misclassifies some training points. This can lead to underfitting if λ is too small.

- The learning rate α aims to adjust the weights and bias (determines how big of a step the algorithm takes in the direction opposite to the gradient) trying to reach the approximated convergence to the the optimal decision boundary.

The gradient descent strategy can be stocastic or batch: In our case we use the batch strategy in order to avoid the noisy updates in case of SGD.

The experimenal results after model training and testing are:

Table 4.1: SVM Classification Report

Class	Precision	Recall	F1-Score	Support
bad [-1]	0.66	0.52	0.58	398
good [1]	0.74	0.84	0.79	666
Accuracy		0.72		1064

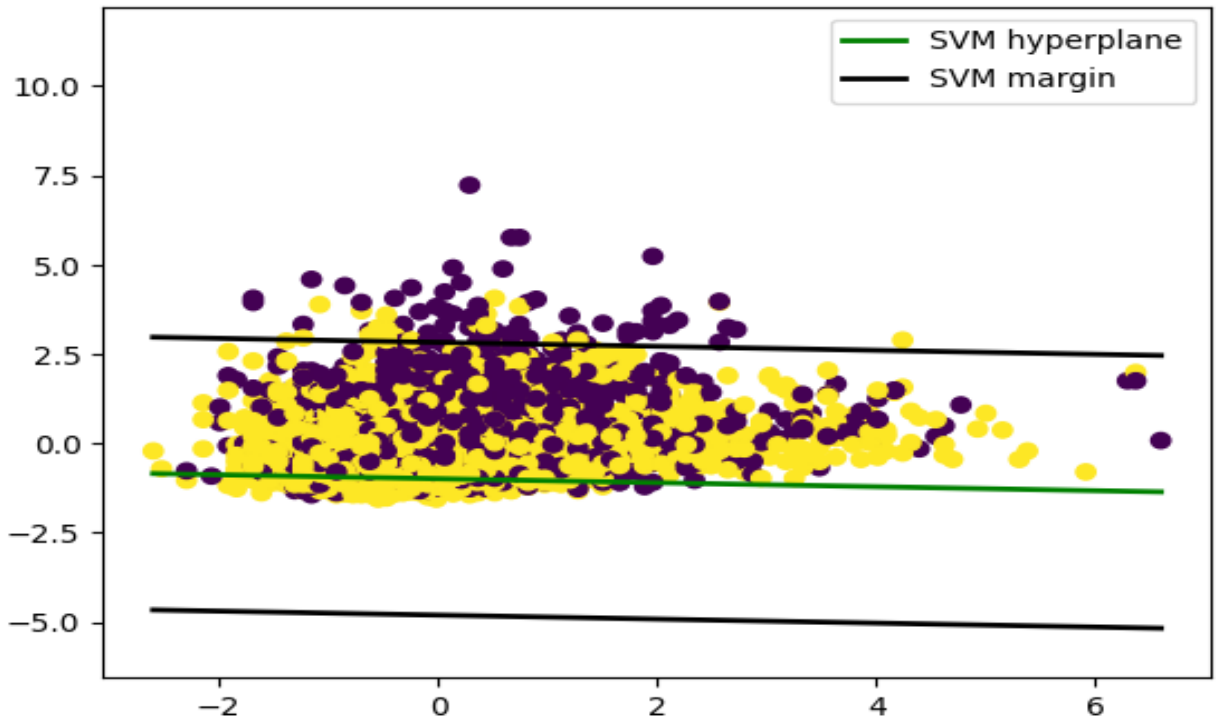

Figure 4.1: SVM hyperplane & margins

5. Logistic Regression (LR)

It seeks to find a logaritmic equation(sigmoid) that best describes how one or more independent variables (features) relate to a dependent variable (target label). The model aims to find the "best-fit" approximated hyperplane that minimizes the gradient(not exact value predictor "based on a predefined threshold"). It squashes any real-valued number into a value between 0 and 1, which can be interpreted as a probability: $\sigma(z) = \frac{1}{1+e^{-z}}$

The learning rate σ aims to adjust the weights and bias (determines how big of a step the algorithm takes in the direction opposite to the gradient) trying to reach the approximated convergence to the optimal decision boundary.

The experimenal results after model training and testing are:

Table 5.1: LR Classification Report

Class	Precision	Recall	F1-Score	Support
bad [-1]	0.53	0.84	0.65	398
good [1]	0.86	0.55	0.67	666
Accuracy		0.66		1064

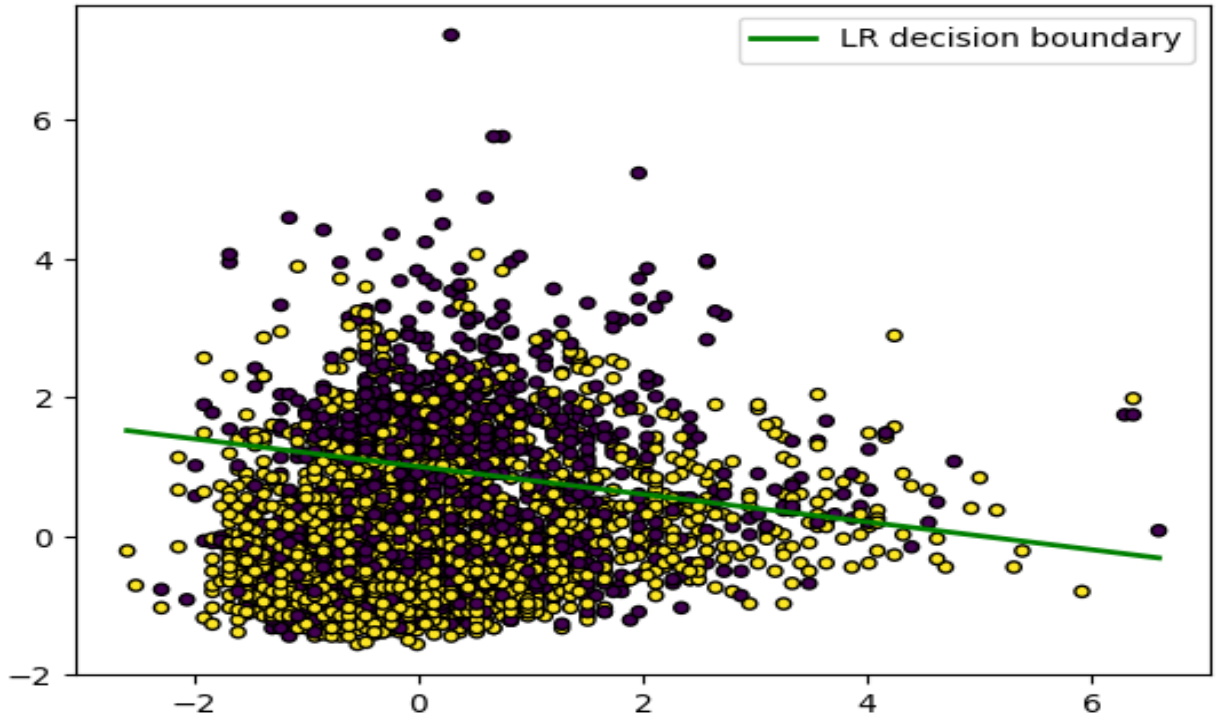

Figure 5.1: LR hyperplane

6. Hyperparameter Tuning

We'll try to apply k-fold cross validation on the logistic regression since it has a lower accuracy compared to SVM w.r.t. default hyperparameters per each predictor. Let's break down the hyperparameters of logistic regression:

- Learning rate α: the most common values of α are [0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0]. If we choose define too high α, we'll see a huge loss because of extremely large steps taken in weight updates. On the other hand, if we choose define too low α, we'll have tiny steps taken in weight updates, leading to extremely slow convergence. Therefore, in our hyperparameter-tuning experience, we might need to tune only [0.001, 0.01, 0.1] since the other values are used for wider search.

- Number of iterations (epochs): the most common values of n.epochs are [50, 100, 200, 500, 1000, 2000]. It's known that the number of epochs is the iterations taken to update weights till reaching an approximated optimal boundary decision. High number of epochs might need lower number of learning rate that need to be proportionally adequate to the the step size * number of steps to take, in order to update the weights. So in our hyper-parameter tuning experience, we might need to tune only [100, 200, 500] w.r.t. the an inversely related α, given our mid-sized training set. So if our chosen learning rate is high, each step taken by the predictor is large, we might reach the vicinity of the boundary faster, potentially needing fewer epoch. And viceversa, if our chosen learning rate is low, each step is small. The predictor will approach the boundry more stably and slowly, but it will require a much higher number of epochs to reach convergence.

- Threshold: this number determines the separation between the probabilities of targert classes, the default values (0.50) indicates the total equality while defining the finale predication. Infact, it's conditioned to [(y_predicted >= threshold) then 1, -1 otherwise]. We can notice that the precision ratio of the minor class ["bad" -1] in our previous prediction is pretty low w.r.t. SVM's precision ratio in case of the same class. Which indicates lower correct classification of positive examples. Meanwhile the recall of the same class is noticeably higher, which indicates that we have fewer examples of false negative predictions. Subsequently, since we're in case of imbalanced data distribution among classes, we'd need to balance the threshold in order to predict more correct positives in case of the minor class ["bad" -1]. The threshold will need to be tuned with lower values than the default one w.r.t. the minor class presence in the dataset.

- Lambda/regularization parameter λ initially for simplicity we would set it to 0. Then if we notice any type of overfitting caused by irregular weights, we'll start tuning it w.r.t. α, threshold and n_iterations.

So as an initial grid to use in LR hyperparameter tuning, we can adopt:

```
lr_params =
      [{'learning_rate':0.001, 'lambda_param': 0.000, 'n_iterations':500, 'threshold':0.37},
       {'learning_rate':0.01, 'lambda_param': 0.000, 'n_iterations':200, 'threshold':0.37},
       {'learning_rate':0.1, 'lambda_param': 0.000, 'n_iterations':100, 'threshold':0.37},
       {'learning_rate':0.001, 'lambda_param': 0.000, 'n_iterations':500, 'threshold':0.31},
       {'learning_rate':0.01, 'lambda_param': 0.000, 'n_iterations':200, 'threshold':0.31},
       {'learning_rate':0.1, 'lambda_param': 0.000, 'n_iterations':100, 'threshold':0.31},
       {'learning_rate':0.001, 'lambda_param': 0.000, 'n_iterations':500, 'threshold':0.37},
       {'learning_rate':0.01, 'lambda_param': 0.000, 'n_iterations':200, 'threshold':0.34},
       {'learning_rate':0.1, 'lambda_param': 0.000, 'n_iterations':100, 'threshold':0.31}]
```

And the maximized retrieved hyperparameters (w.r.t. accuracy metric and k = 5) are:

```
{'learning_rate': 0.01, 'lambda_param': 0.0, 'n_iterations': 200, 'threshold': 0.31}
# acc. 0.7154543398266607
```

Which obviously shall give us higher performance scores with a good balance between precision a recall:

Table 6.1: Tuned LR Classification Report

Class	Precision	Recall	F1-Score	Support
bad [-1]	0.64	0.63	0.63	398
good [1]	0.78	0.79	0.78	666
Accuracy		0.73		1064

6.1. Analysis of misclassified examples

Since we have tuned only the standard logistic regression. So we shall focalize on its performance. Even after hyperparameter tuning, the "bad" class (minor class with recall 0.63, precision 0.64 and f1-score with almost the same value before tuning [0.63 => 0.65]) is harder to predict than the "good" class (major class with recall 0.79, precision 0.78 and f1-score with higher value [0.67 => 0.78]). In other words, the hyperparameter tuning is baised towards the major class. For these possibile reasons:

- Class imbalance (most probable): since the ratio of support between "bad" and "good" classes is (1 : 1.67) which reflects the lower number of training examples for the "bad" class. Logistic Regression and general standard ML models are designed to optimize the overall accuracy or minimize a loss function by updating weights and bais values treating all errors equally; so in the scenario of imbalance classes, the major class performes more updates w.r.t. the minor one.

- Insufficient class complexity: the minor class doesn't have significant (or unique) features that can reflect a significant variance to weight/bias updates w.r.t. the major one. Infact as seen above (ref. 3.1) the minor class (bad quality) shares almost all features' values with the major class (good quality) with exception to higher volatile acidity, citric acid and notable higher values of sulfur dioxides. This shows us how the datapoints of our two class are overlapping.

- Logistic regression linearity: Since LR is a linear model, it suffers to find a linear decision boundary (especially in our case of interleaving datapoints of different classes, since there's no linear separability for the minor class features).

- Possibile noisy datapoints that make the minor class interleave with the major one, resulting a baised prediction towards the major class.

7. Kernelized Support Vector Machine (KSVM)

Non-linear kernels are generally used when we notice that that datapoints of different classes are not linearly serparable, such that there exists an extreme overlapping between classes' data points w.r.t. the original dimensions. Interleaved patterns between classes (non-linear separability) makes it difficult for linear models to find the appropriate decision boundry. The kernel trick introduces the mapping from the original feature space ϕ of datapoint x to a higher dimensional space exploiting a kernel function $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$, aiming to

find an appropriate decision boundary in the higher dimensional space. The kernel function to choose depends the prior knowlege of data distribution; in case of no prior knowlege, the gaussian kernel is the most common kernel. In case of KSVM

- Training:
  1. Calculate the kernel matrix on the training set through K(X, X) using a predefined kernel function.
  2. Resolve the QP problem to determine alpha vector such that 0 <= α_i <= regularization term w.r.t. the kernel matrix.
  3. Determine labeled support vectors SV w.r.t. the resolved alphas. To avoid SV with $\alpha = 0$, a common lower threshold should be applied (ex. 0.00001). The maximized margin SV shouldn't exceed the regularization term.
  4. Calculate the bias term as bias = $y_k - \sum_{i \in S_V} \alpha_i y_i K(x_i, x_k)$ for such a support vector (x_k, y_k) w.r.t. any support vectors x_i.
- Prediction: To classify a new data point x_{test}, the decision function is calculated through: $f(x_{\text{test}}) = \sum_{i \in S_V} \alpha_i y_i K(x_i, x_{\text{test}}) + \text{bias}$ where SV refers to the support vectors determined in the training phase, α_i is the alpha multiplier associated to a support vector data point x_i. The kernel function $f(x_{\text{test}})$ is used to measure similarity score between the test point and each support vector. Then we use the sign function (as used in the standard SVM) to predict -1 or 1.

And here're the performance scores after applying the polynomial kernel on SVM, without any specific hyperparameter tuning:

Table 7.1: Polynomial KSVM Classification Report

Class	Precision	Recall	F1-Score	Support
bad [-1]	0.69	0.62	0.65	398
good [1]	0.79	0.83	0.81	666
Accuracy		0.75		1064

8. Kernelized Logistic Regression (KLR)

- Training:
  1. Calculate the kernel matrix on the training set through K(X, X) using a predefined kernel function.
  2. For a specific number of epochs: Find the sigmoid predition scores based on $\sum_{i \in N} K(x_i, x_i) \cdot \alpha_i + \text{bias}$.
  3. For a specific number of epochs: Continue updating alphas using the gradient descent based on K.

4. For a specific number of epochs: Continue updating bias using the gradient descent based on training error w.r.t. the real scores.

- Prediction: To classify a new data point x_{test}, the decision function is calculated through: $f(x_{\text{test}}) = \sum_{i \in N} K(x_i, x_t est) \cdot \alpha_i + \text{bias}$, where N refers to the number of trained examples, α_i is the alpha multiplier associated to a trained example x_i. The kernel function $f(x_{\text{test}})$ is used to measure similarity score between the test point and each trained data point. Then we use the threholded sigmoid function (as used in the standard LR) to predict -1 or 1.

  And here're the performance scores after applying the polynomial kernel on LR, without any specific hyperparameter tuning:

Table 8.1: Polynomial KLR Classification Report

Class	Precision	Recall	F1-Score	Support
bad [-1]	0.46	0.71	0.56	398
good [1]	0.79	0.58	0.67	666
Accuracy		0.62		1064

9. Kernelization's Consequences on Prediction and Performance

- Prediction:
  - The prediction phase becomes more complex and time consuming since it needs to calculate the test kernel matrix:
    In case of KSVM: K(X_{test}, SV) based on the predefined support vectors.
    In case of KLR: K(X_{test},X_{training}) based on the pre-stored training set.
  - It also uses the alpha vector and the bias term pre-calculated in the training phase, in order to give a prediction.
  - Since the non-linear boundaries can be very complex. This increases the model's capacity to fit more the training data. This also increases the risk of overfitting if the kernel-specific hyperparameters in additional to normal model's hyperparameters are not tuned correctly.
- Performance:
  - Increased Time Complexity:
    - The Computation of Kernel Matrix: The most significant bottleneck during training is the computation of the kernel matrix to be defined in the

training phase, which is an $N\ddot{O}N$ matrix where N is the number of training samples. Each entry K_{ij} requires a kernel evaluation, which leads to an $O(N^2 * D)$ complexity for constructing the matrix. For large datasets, N^2 grows very quickly, making kernelized models much slower than linear models.

- Alpha multipliers (α) calculation:
  In case of KSVM: Quadratic Programming (QP) Solver: Solving the underlying QP problem to find the optimal alphas also typically scales between $O(N^2)$ and $O(N^3)$ which depends heavily on the size N of the training set.
  In case of KLR: Iteratively re-weighted the gradient descent to update α which essentially involves operations on the N×N kernel matrix, leading to training complexities $O(N^2$ * n_epochs).
- Additional Hyperparameter Tuning: Kernelized models introduce more hyperparameters to be tuned (ex. gamma, degree, ...) that need extensive tuning w.r.t. the chosen kernel function. Each combination requires retraining the entire model, multiplying the training time significantly for each time we need to tune a single combination of hyperparameters.

* Increased Space Complexity:
  - Storing the N×N kernel matrix during training requires $O(N^2)$ memory, which can quickly become a challenge for large datasets.
  - In case of KSVM: The trained model needs to store all support vectors SV(and their associated α and labels) for prediction, leading to O(Nsv * D) memory consumption for the model itself, for the dimension space D.
  - In case of KLR: The whole training K matrix needs to be preserved also for the prediction phase, occupying the memory always with additional $O(N^2)$.

9.1. Comparison between Standard Models and Kernelized Models

– The created non-linear decision boundary in the original input space, allows predictor to model complex relationships in data that would be impossible with a simple linear model.

– Unlike a linear model, where the weight vector directly tells us the importance and direction of each feature, a kernerlized decision boundary is defined by a weighted sum of kernel evaluations with training K matrix (or support vectors in case of SVM). Which makes it much harder to interpret the direct relationship between input features and the decision boundary.

– Standard model may underfit if the data truly requires a non-linear boundary, since it won't handle the underlying relationship between features and labels.

Meanwhile a kernelized model may overfit if the hyperparameters aren't well-tuned given the high dimensionality mapping.

– The phase of hyperparameter tuning in kernelized models is relatively high w.r.t. the standard ones. It needs extensive tuning w.r.t. the chosen kernel function retraining the entire model over the high-dimensional mapped datapoints. Which introduces a real challange for large-sized datasets.

10. Model Evaluation and Plots

– SVM:

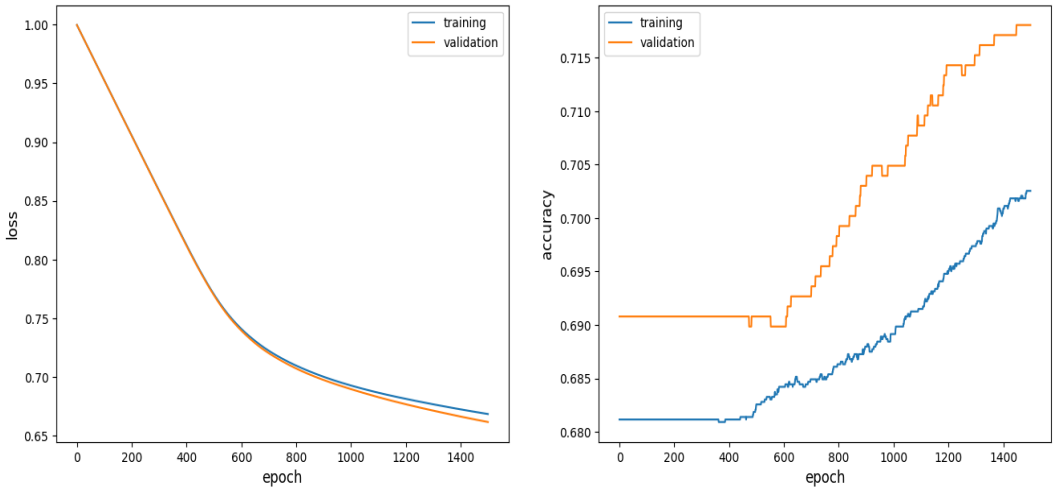

Figure 10.1: SVM Performace Evaluation

* In the loss plot: the loss curves are in a decreasing similar order, till approximated convergence w.r.t. the end of training phase (the validation curve even gets lower). This scenario indicates healthy learning phase w.r.t. the loss plot.
* In the accuracy plot: the validation curve is higher than the training one, and both are in an increasing order. This scenario indicates healthy learning phase which reflects correct predictions w.r.t. the accuracy plot.

– LR:

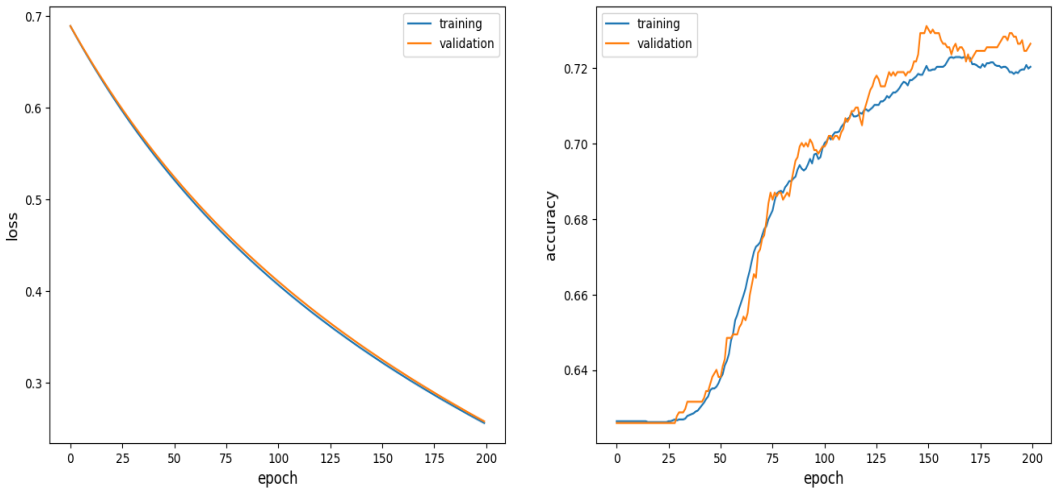

Figure 10.2: LR Performace Evaluation

* In the loss plot: the loss curves are in a decreasing similar order, till approximated convergence w.r.t. the end of training phase. This scenario indicates healthy learning phase w.r.t. the loss plot.
* In the accuracy plot: the training curve in several epochs get higher than the validation one (or in other words, the validation curve suffers from some oscillations within epochs before the 175th epoch), and both are in an increasing order which indicate and acceptable learning phase. But the oscillations in the validation curve indicate the suffering of the model to predict unseen datapoints (overfitting).

– Polynomial KLR:

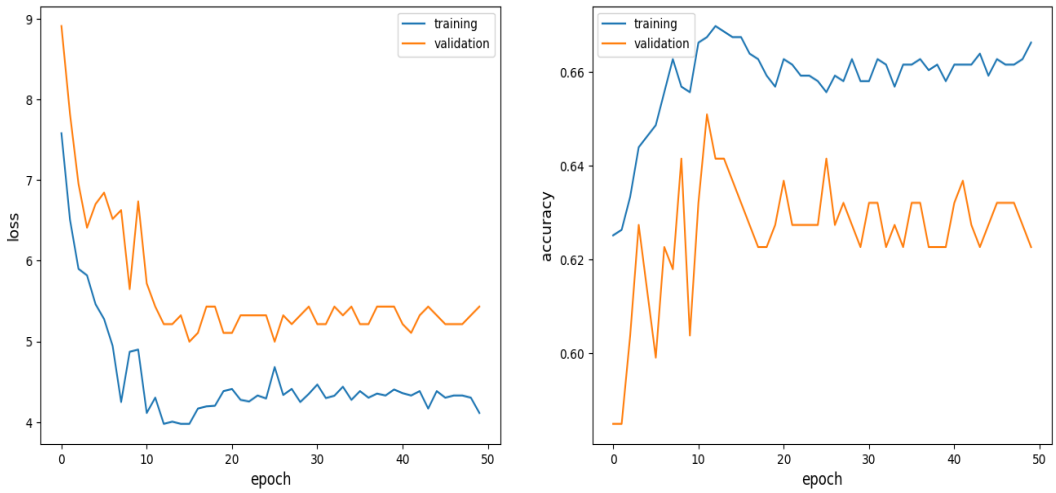

Figure 10.3: Polynomial KLR Performace Evaluation

The polynomial KLR has oscilliation/spikes (zig-zags) in the training and validation curves within loss and accuracy plots, indicating the instability of the training phase caused by non-coherent weight/bias update steps. An initial solution could be the hyperparameter tuning for polynomial KLR which will be very expensive since it's a kernelized version of LR.

Note that in case of kernelized SVM model, we can't plot the loss and accuracy curves per epochs since there's no convention for weights update to converge towards the optimality through epochs * learning rate steps. The kernel trick requires to solve a QP problem in order to determine optimal alphas without iterating the learning phase. Furthermore, it could be useful to plot the loss and accuracy curves for the kernelized models within the K-fold cross validation. Observing the change of training and validation metrices per fold's iteration, which would help to deduce the relationship between hyperparameters and overfitting/underfitting. But it'll have high time complexity since it requires the reconstruction of kernel matrices.

11. References

- Foundations of Machine Learning (2nd edition): cs.nyu.edu
- Introduction to Machine Learning in python (O'REILLY): www.nrigroupindia.com