Experimental wine project report

Diego Alonso Leiva Cordova

Enrolment number: 58757A

Data Science for Economics 2024/2025

GitHub repository: https://github.com/Diego-LeivaC/wine_leiva.git

The objective of this project is to implement Support Vector Machine and Logistic Regression from scratch to predict whether wines are "good" (quality ≥ 6) or "bad" (quality < 6). Quality is the target variable of the dataset. The Python libraries that were used were Pandas for dataset manipulation, NumPy for mathematical operations and Matplotlib for data visualization. This report is divided in four parts:

1) Data exploration and preprocessing

The datasets are two, one for red wine, and one for white. The dependent feature variables are fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol, while the target variable is quality. Red dataset consists of 1599 rows and 12 columns of which 240 rows are duplicated; while white dataset consists of 4898 rows and 12 columns of which 937 are duplicated.

Before joining red and white datasets, it has created a new column "type" to differentiate the wine colour, for red colour the type is 0 and 1 for the white.

After joining the datasets, the duplicates were deleted, and the new shape of the joined dataset is 5320 rows and 13 columns. The data types are floating for the feature variables, except the new one "type" that is an integer; quality, the dependent variable is also an integer. Furthermore, the dataset does not contain null values.

The quality variable goes from 3 to 9, so for an efficient data manipulation this variable is converted to a binary one where zero is equal to bad wine (quality < 6), and one is equal to good wine (quality ≥ 6). After this conversion there are 3332 values for class one, and 1988 for class zero. Following the conversion, the data were split into training set (75%) `training_data` and test set (25%) `test_data`.

Firstly, it was created `X_train` that are the values of the training set except quality and `X_test` that are the values of the test set except quality. Secondly, it was created `y_test_true` that are the values of the test set of the quality variable. This last creation were compared with the wine class and the original value of quality variable, in other words, if the original quality is equal to 7, the wine class must be equal to 1 and the `y_test_true` must be equal to 1; and if the original quality is 5, the wine class must be 0 and the `y_test_true` must be 0. The comparison was fine according to a sample of 20.

2) Support Vector Machine and Logistic Regression implementation

2.1) Support Vector Machine (SVM)

The implementation of a simple linear SVM classifier from scratch consist of:

- Training a linear SVM using gradient descent
- Making predictions on the test set
- Evaluating model accuracy

Firstly, SVM uses labels of -1 and 1, so the labels 0 and 1 were preprocessing, this was applied to the new created training target label `y_train`:

```
def zeroOne(label):

    return np.where(label == 0, -1, 1)

y_train = zeroOne(training_data["quality"].values)
```

Defining the SVM training function: "`training_svm_vectorized`"

The inputs are:

- X: feature matrix (n_samples x n_features)
- y: labels (-1 or 1)
- lr: learning rate (0.0015 by assignation)
- C: regularization parameter (0.01 by assignation)
- epoch: number of training iterations (1000 by assignation)

The algorithm steps are:

1. Initialization:
   1.1. Weight vector "w" initialized to zeros (one weight per feature)
   1.2. Bias "b" initialized to zero
2. Training loop: For epoch iterations

2.1. Margin calculation: This computes the margin for each sample, which tells how far each point is from the decision boundary.

```
margins = y * (np.dot(X, w) - b)
```

2.2. Mask for Misclassified Points (where margin <1): These are the points that do not satisfy the SVM margin condition and contribute to the loss/gradient.

```
mask = margins < 1
```

2.3. Gradient computation:

For weights: This combines the regularization term 2*C*w and the gradient from misclassified points.

```
dw = 2 * C * w - np.dot(X[mask].T, y[mask])
```

For bias:

```
db = -np.sum(y[mask])
```

2.4. Parameter Update (gradient descent step)

```
w -= lr * dw

b -= lr * db
```

3. Output: A trained weight vector w, and a trained bias b.

Defining the SVM Prediction function: "`test_svm`"

- Given trained parameters w and b, predictions are made as follows:

```
pred = np.dot(X, w) - b

return np.where(pred > 0, 1, 0)
```

- So, if the result is positive classify as 1, if is negative classify as 0.

The model is trained using the training data:

```
w_trained, b_trained = training_svm_vectorized(X_train, y_train)
```

The prediction on the test set is:

```
y_pred = test_svm(X_test, w_trained, b_trained)
```

The accuracy that computes the proportion of correctly classified test examples is:

```
accuracy = np.mean(y_pred == y_test_true)
```

Finally, there is a sample output comparison to visualize the first 150 test predictions and compared it with true labels.

```
for i in range(150):

    print(f"Example {i+1}: Prediction = {y_pred[i]}, Real = 
{y_test_true[i]}")
```

The SVM accuracy is 0.6376, but there many class 1 predictions. It was proving different data split from 60% to 80% of training set and the best is 78% of training set, because it has the best accuracy 0.6541 and the predictions are more stable (more class 0).

2.2) Logistic Regression (LR)

The LR uses labels 0 and 1 which are already present in the target column:

```
y_train = training_data["quality"].values
```

The core of LR is the sigmoid function that maps a real number into the range (0, 1) that represents the probability that a sample belongs to class 1.

```
def sigmoid(z):

    return 1 / (1 + np.exp(-z))
```

Defining the LR training function: `"training_lr"`

The inputs are:

- X: Feature matrix (n_samples x n_features)
- y: Labels (0 or 1)
- lr: Learning rate (0.00015 by assignation)
- epochs: Number of training iterations (1000 by assignation)

The algorithm steps are:

1. Initialization:
   1.1. Weight vector w initialized to zeros
   1.2. Bias b initialized to zero.
2. Training loop:
   2.1. Linear model (logits): `linear_model = np.dot(X, w) + b`
   2.2. Sigmoid activation (predicted probabilities):

   ```
   predictions = sigmoid(linear_model)
   ```

2.3. Error computation: `error = predictions - y`

2.4. Gradient calculation:

For weights: `dw = np.dot(X.T, error) / n_samples`

For bias: `db = np.sum(error) / n_samples`

2.5. Parameter update (Gradient Descent step):

```
w -= lr * dw

b -= lr * db
```

3. Output: trained weights w and a trained bias b.

Defining the LR Prediction function: "`test_lr`"

```
def test_lr(X, w, b):

    probabilities = sigmoid(np.dot(X, w) + b)

    return (probabilities >= 0.5).astype(int)
```

So, if the probability is equal or higher to 0.5 it predicts class 1, else predicts class 0.

The model is trained on the training data.

```
w_lr, b_lr = training_lr(X_train, y_train)
```

Prediction on Test Set:

```
y_pred_lr = test_lr(X_test, w_lr, b_lr)
```

Accuracy:

```
accuracy_lr = np.mean(y_pred_lr == y_test_true)
```

Output:

```
print("Logistic Regression accuracy:", accuracy_lr)
```

The accuracy is 0.6371, lower than linear SVM, so to increase it, hyperparameter tuning and accuracy estimation via 5-fold cross validation was applied to LR.

Defining cross-validation function: "`k_fold_cross_validation_lr`"

1. Randomly shuffles dataset indices for proper splitting.
2. Splits data into 5 equal folds (k=5).

3. Hyperparameter Grid Search: Iterates over all combinations of learning rates (`learning_rates`) and epoch numbers (`epoch_numbers`).
4. Cross-Validation Loop:

- For each fold:

   a. Use 1-fold as validation set and remaining folds as training set.
   b. Train Logistic Regression model with `training_lr()`.
   c. Predict on validation fold using `test_lr()`.
   d. Compute fold accuracy.

- Averages the 5-fold accuracies.

5. Selection of the best hyperparameter**:** Keeps track of the hyperparameters that achieve the highest average cross-validation accuracy.

Output: Prints results for each hyperparameter combination and returns the best learning rate and epochs.

This cross-validation uses the full dataset (no train/test split):

```
X_full = data.drop("quality", axis=1).values

y_full = data["quality"].values
```

Tries 4 different learning rates and 4 different epoch numbers.

```
learning_rates = [0.0001, 0.00015, 0.001, 0.01]

epoch_numbers = [100, 500, 1000, 1500]
```

Performs 5-fold cross-validation over all combinations.

Prints the best hyperparameters found:

```
best_lr, best_epochs = k_fold_cross_validation_lr(X=X_full, y=y_full,

    learning_rates=learning_rates, epoch_numbers=epoch_numbers, k=5)
```

Trains final LR model on training set using best lr (0.0001) and epochs (1000):

```
w_final, b_final = training_lr(X_train, y_train, lr=best_lr,
epochs=best_epochs)
```

Makes predictions on the test set:

```
y_pred_final = test_lr(X_test, w_final, b_final)
```

Computes final accuracy on the test set:

```
final_accuracy = np.mean(y_pred_final == y_test_true)
```

The best accuracy for the Logistic regression is 0.6396.

3) Kernel methods

Before passing the Gaussian Kernel, the next code performs the normalization of the data: it first computes the mean and standard deviation of each feature from the training set, then scales the training and test data to have zero mean and unit variance. This standardization improves numerical stability and model convergence for kernel methods.

```
# Prepare the data

X_train_raw = training_data.drop("quality", axis=1).values

X_test_raw = test_data.drop("quality", axis=1).values

# Compute the mean and standard deviation on training data

mean = np.mean(X_train_raw, axis=0)

std = np.std(X_train_raw, axis=0)

# Normalize the training data

X_train_norm = (X_train_raw - mean) / std

# Normalize the test data using training mean and std

X_test_norm = (X_test_raw - mean) / std
```

3.1) Kernel SVM

This part implements Kernel SVM from scratch using a Radial Basis Function (RBF).

Firstly, converts labels to {-1, 1} for SVM formulation.

Then, computes the Gaussian kernel (similarity) between two datasets using the formula:

$K(xi,xj)=\exp(-1/2\gamma\cdot||xi-xj||^2)$

Defining SVM Training: "`training_svm_kernel_vectorized`"

Workflow:

1. Initialize dual variables alpha and bias.
2. Compute Kernel Matrix K (pairwise similarities).

3. Iterative optimization loop:
    3.1. For epochs number of iterations:
    3.2. Compute margin for all training samples (vectorized).
    3.3. Identify violations (samples with margin < 1).
    3.4. Update:
        3.4.1. Increase alpha only for violations.
        3.4.2. Update bias based on violating samples.
    3.5. Apply regularization (shrink alpha with penalty C).
4. Returns learned alpha and bias

The dual form of SVM is optimized in terms of alpha (dual variables), allowing the kernel trick to work. The vectorized loop speeds up the process by handling all samples simultaneously.

Defining SVM Prediction: **`"predict_svm_kernel"`**

Workflow:

1. Compute the kernel matrix between the test and training data.
2. Calculate decision values using learned alpha and bias.
3. Convert decision values to binary predictions (0/1).

For SVM `y_train` is different to LR, so it needs to convert:

```
y_train = zeroOne(training_data["quality"].values)
```

Trains SVM with vectorized training loop using RBF kernel:

```
alpha_trained, bias_trained =
training_svm_kernel_vectorized(X_train_norm, y_train)
```

Predicts on test set and prints accuracy:

```
y_pred_kernel = predict_svm_kernel(X_test_norm, X_train_norm, y_train,
alpha_trained, bias_trained)

accuracy_kernel = np.mean(y_pred_kernel == y_test_true)

print("SVM Kernel (RBF) accuracy with vectorized inner loop:",
accuracy_kernel)
```

So, the Kernel SVM accuracy is 0.7233, higher than linear SVM (0.6541)

3.2) Kernel Logistic Regression

The sigmoid function defines the LR activation needed to map linear outputs to probabilities between 0 and 1. The `rbf_kernel` function is the same that the SVM.

The `training_lr_kernel` function does a train of the model in kernel space by updating weights w and bias b over 1000 epochs using gradient descent. The kernel matrix K is computed for all training samples. For each epoch, the linear model is computed in kernel space, followed by calculating the sigmoid predictions. The gradient of the loss function with respect to the weights and bias is computed and used to update w and b.

The `predict_lr_kernel` function generates predictions on test data by computing the RBF kernel between test and training samples and applying the trained weights and sigmoid function. The class predictions are obtained by thresholding probabilities at 0.5.

Finally, the script trains the model using the normalized training data and evaluates its performance on normalized test data and prints the accuracy: 0.6823.

4) Evaluation and analysis

4.1) Confusion matrix and metrics

The `confusion_matrix_manual` function calculates the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), returning them in a 2x2 confusion matrix. From this matrix, the metrics are `accuracy_manual` computes the overall proportion of correct predictions, `precision_manual` calculates the proportion of positive predictions that are correct, `recall_manual` measures the proportion of actual positives that are correctly identified, and `f1_score_manual` combines precision and recall into a single harmonic mean score.

- **Logistic Regression Performance:**

Confusion Matrix:

	Predicted 0	Predicted 1
Actual 0	23	407
Actual 1	**15**	**726**

Accuracy: 0.6396 | Precision: 0.6408 | **Recall: 0.9798** | F1-score: 0.7748

- **Kernel Logistic Regression Performance:**

Confusion Matrix:

	Predicted 0	Predicted 1
Actual 0	93	337
Actual 1	35	706

Accuracy: 0.6823 | Precision: 0.6769 | Recall: 0.9528 | **F1-score: 0.7915**

- **SVM Performance:**

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	91	339
Actual 1	66	675

Accuracy: 0.6541 | Precision: 0.6657 | Recall: 0.9109 | F1-score: 0.7692

- **Kernel SVM Performance:**

Confusion Matrix:

	Predicted 0	Predicted 1
Actual 0	**363**	**67**
Actual 1	257	484

Accuracy: 0.7233 | Precision: 0.8784 | Recall: 0.6532 | F1-score: 0.7492

4.2) Loss and accuracy plot

4.2.1) Loss and accuracy for Kernel Logistic Regression

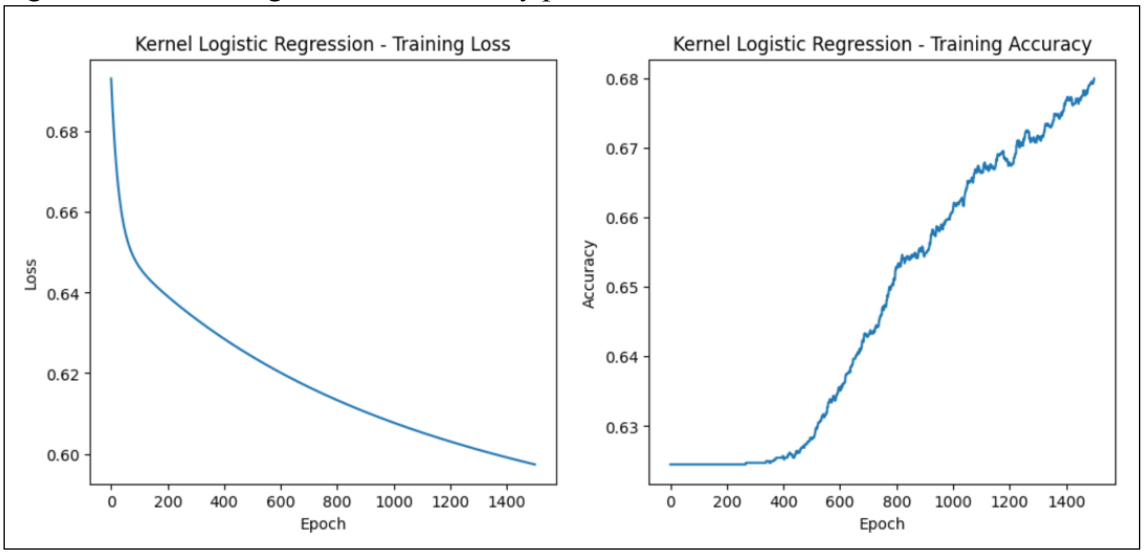
The training process is designed to monitoring the evolution of the model's performance (accuracy) and the optimization objective (loss) over the course of 1500 epochs.

The KLR model applies logistic regression in kernel space, allowing capture non-linear relationships in the input features. It's computed a kernel matrix K that measures the similarity between each pair of training examples based on the RBF kernel with a fixed gamma value of 0.1; then learns a weight vector w in this kernel-induced space.

The optimization is performed using gradient descent with a learning rate 0.0001. During training, the binary cross-entropy loss is calculated at each epoch to measure how well the model fits the data. Furthermore, it's computed the training accuracy by thresholding the predicted probabilities at 0.5 and comparing them to the true binary labels (quality).

For monitoring, the loss values and accuracy scores are stored during training and can be visualized as curves to detect overfitting or wrong training.

Figure nro.1: Training Loss and Accuracy plot for KLR



The training loss decreases across epochs, indicating that the model is successfully learning to fit the data over time; the model adjusts the most during the initial learning phase. After 300 epochs, the curve continues to decline gradually without fluctuations or increases, this shows stable convergence, with no signs of overfitting in the loss curve.

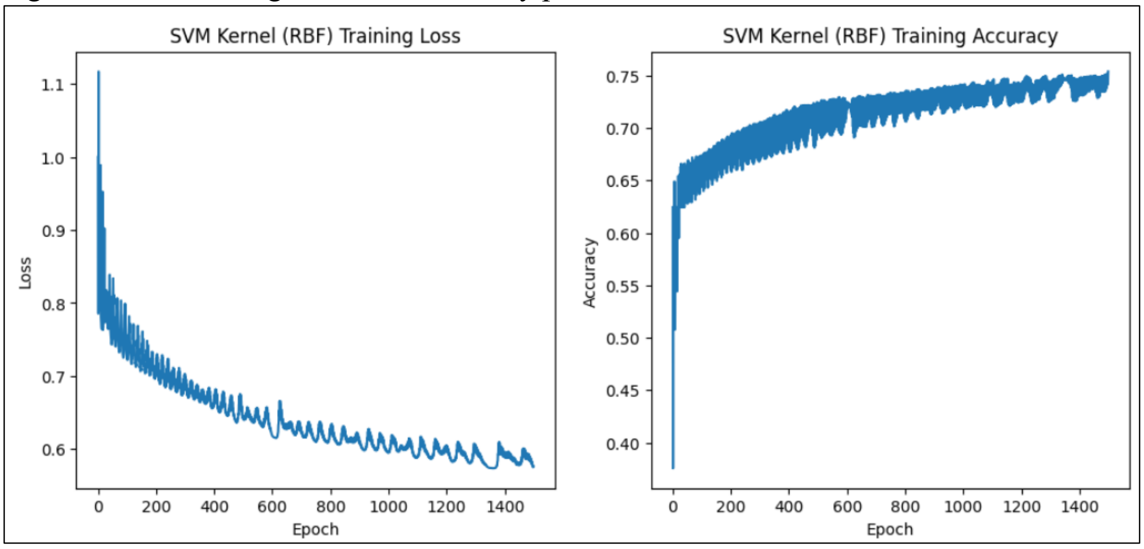
The training accuracy for the first 400 epochs remains relatively flat, this suggests that the model needed more epochs to start achieving a better classification. After 500 epochs, accuracy rises quickly and almost linearly, this phase represents the period when the model starts effectively learning useful patterns from the data. Even at the final epoch (1500), accuracy is still increasing (reaching 0.68), meaning the model hasn't reached the limit yet and could likely still improve with more training.

4.2.2) Loss and accuracy for Kernel Support Vector Machine

The training process was implemented in a vectorized form to improve efficiency and included an historical of hinge loss and training accuracy.

The KSVM training involved updating the dual variables and the model bias using gradient, focusing updates on samples violating the margin condition. The hinge loss function measured the model's error by determining how far the prediction of the classifier were from the target margin. Across the epochs, the loss values were recorded to monitor how well the model was learning to separate the classes in kernel space. The training accuracy was computed in each epoch based on the model’s predictions compared to the true labels.

Figure nro.2: Training Loss and Accuracy plot for KSVM



The hinge loss starts approximately at 1.1 at the beginning of training and shows a clear decreasing trend as epochs increases, indicating that the model is improving at finding a better separating margin between classes. The loss curve exhibits oscillations or noise and may be due the non-smooth nature of the hinge loss. By the end of training (around epoch 1500), the loss stabilizes near 0.6, suggesting that the model has largely converged.

The accuracy starts approximately at 0.4 but increases rapidly in the first 100 epoch, reflecting the model’s ability to quickly learn from the data when starting from zero initialization. After that, accuracy continues to improve steadily, approaching about 0.75 near epoch 1500. It shows slight oscillations, which is consistent with the hinge loss behaviour, typical when training with fixed learning rates and simple gradient methods.

On overall, Kernel SVM has the highest accuracy and precision than any other model and is better balanced according to the confusion matrix. (more true negatives and less false positive); according to the plots: Kernel SVM has a stronger training accuracy (~75%) than Kernel logistic and a similar loss (<0.6). Consider also that the hyperparameter tuning via 5-fold cross validation was applied only to LR model.