MLDL EXPERIMENT 4

Aim: Implement K-Nearest Neighbors (KNN) and evaluate model performance.

Dataset Description:

This dataset, "car data.csv", consists of 301 entries representing used vehicles (primarily cars, with some motorcycles) available for sale. It includes 9 columns:

- **Car_Name (string):** The model name of the vehicle (e.g., "ritz", "fortuner", "Royal Enfield Classic 350"). There are 98 unique models, with "city" being the most frequent (26 entries).
- **Year (integer):** The manufacturing year, ranging from 2003 to 2018. The most common year is 2015 (61 entries).
- **Selling_Price (float):** The price at which the vehicle was sold, in lakhs (Indian rupees). This is likely the target variable for predictive modeling, ranging from 0.1 to 35 lakhs, with an average of ≈4.66 lakhs and median of 3.6 lakhs.
- **Present_Price (float):** The current ex-showroom price of the model, in lakhs, ranging from 0.32 to 92.6 lakhs (average ≈7.63 lakhs).
- **Kms_Driven (integer):** Total kilometers driven, ranging from 500 to 500,000 km (average ≈36,947 km).
- **Fuel_Type (categorical):** Type of fuel used – Petrol (239 entries), Diesel (60), CNG (2).
- **Seller_Type (categorical):** Seller category – Dealer (195), Individual (106).
- **Transmission (categorical):** Gear type – Manual (261), Automatic (40).
- **Owner (integer):** Number of previous owners – 0 (290 entries), 1 (10), 3 (1).

Dataset Source:https://www.kaggle.com/datasets/uciml/red-wine-quality-cortez-et-al-2009

Theory:

K-Nearest Neighbors (KNN) is a simple, non-parametric, instance-based supervised learning algorithm used for both classification and regression tasks.

- **Core Idea:** KNN operates on the principle of similarity: "birds of a feather flock together." For a new data point, it identifies the K most similar (nearest) points from the training data based on a distance metric, then makes a prediction.
    - **Classification:** The prediction is the majority class (vote) among the K neighbors.
    - **Regression:** The prediction is the average (or weighted average) of the target values among the K neighbors.
- **Distance Metrics: Common ones include:**
    - **Euclidean distance:** $\sqrt{[\Sigma(x_i - y_i)^2]}$ (straight-line distance, default for continuous features).
    - **Manhattan distance:** $\Sigma|x_i - y_i|$ (grid-like distance).

    - **Minkowski distance:** Generalization of the above (p=2 for Euclidean, p=1 for Manhattan).
- **Algorithm Steps:**
    - Store all training data points.
    - For a query point, compute distances to all training points.
    - Select the K smallest distances (nearest neighbors).
    - Aggregate: Vote for classification or average for regression.
- **Hyperparameters:**
    - **K:** Number of neighbors (small K risks overfitting/noise sensitivity; large K smooths but may underfit).
    - **Weights:** Uniform (equal vote) or distance-based (closer neighbors have more influence).
- **Theoretical Foundation:** KNN is a lazy learner (no explicit training phase; computation at prediction time). It assumes locally constant functions and relies on the "curse of dimensionality" being mitigated by feature scaling/normalization.
- **Complexity:** Training O(1), prediction O(n*d) where n is samples, d is features (can be optimized with KD-trees or Ball trees for low dimensions).

Limitations:

While KNN is intuitive and versatile, it has several drawbacks:

- **Computational Intensity:** Requires storing the entire dataset and computing distances for every prediction, making it slow and memory-heavy for large datasets (O(n) time per query).
- **Curse of Dimensionality:** Performance degrades in high-dimensional spaces as distances become less meaningful (all points seem "far" apart), leading to poor generalization.
- **Sensitivity to Noise and Irrelevant Features:** Outliers or noisy data can heavily influence predictions since there's no model to filter them; requires careful preprocessing.
- **Imbalanced Data:** In classification, majority classes can dominate votes unless weighted or balanced.
- **No Interpretability:** Doesn't provide feature importance or a learned model; it's a "black box" reliant on data instances.
- **Scalability Issues:** Not suitable for real-time applications with massive data; alternatives like approximate nearest neighbors (e.g., via Locality-Sensitive Hashing) are needed.
- **Hyperparameter Tuning:** Choosing optimal K and distance metric requires cross-validation, which adds computation.
- **Boundary Issues:** Poor at extrapolation beyond training data range.

Workflow:

Data Loading and Exploration Load the CSV file into a pandas DataFrame, check for missing values using isnull().sum(), display basic information with info() and describe(), examine the distribution of the quality scores with value_counts() or a histogram, compute and visualize the correlation matrix (e.g. using seaborn heatmap) to understand relationships between features and quality, and look for obvious outliers or unusual patterns in boxplots or pairplots.

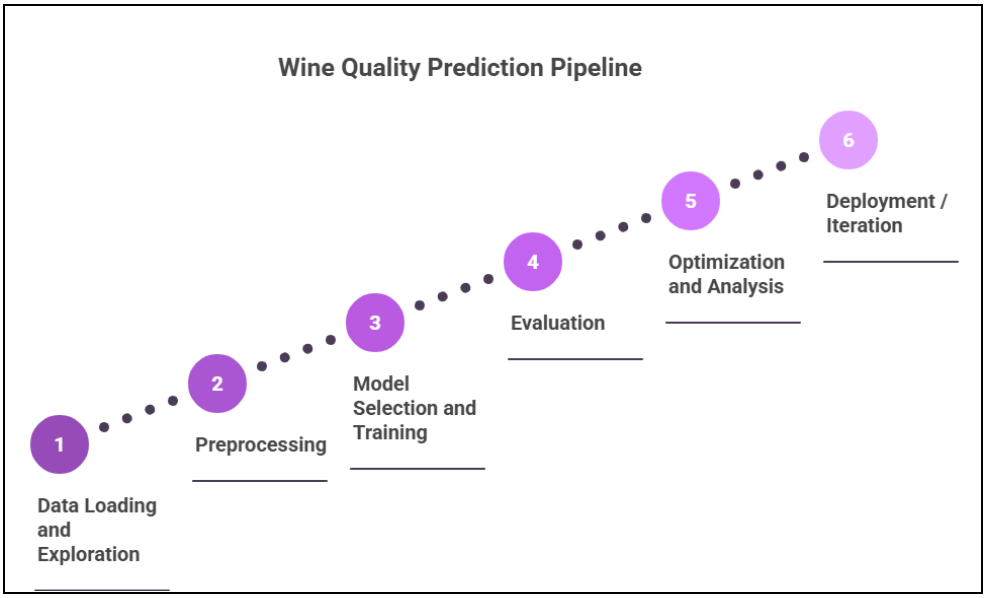
Preprocessing Separate the features (X) by dropping the quality column and keep quality as the target (y), optionally apply oversampling techniques such as SMOTE to handle the imbalance in quality classes (especially the very low and very high scores), scale all features using StandardScaler (strongly recommended for KNN because it is distance-based), then split the scaled data into training and test sets using an 80/20 ratio with a fixed random_state for reproducibility.

Model Selection and Training Use KNeighborsClassifier from scikit-learn since we are predicting discrete quality classes, perform hyperparameter tuning mainly on the number of neighbors (K) using GridSearchCV with 5-fold cross-validation and accuracy as the scoring metric (also consider testing weights='uniform' vs 'distance'), fit the model on the training data — note that KNN is a lazy learner so “training” mainly consists of storing the data points.

Evaluation Use the best model found to make predictions on the test set, calculate overall accuracy, generate a full classification report showing precision, recall, and F1-score for each quality class, display the confusion matrix to understand misclassification patterns (especially between neighboring scores like 5 vs 6), and perform cross-validation on the entire dataset (or on training data) to get a more reliable estimate of generalization performance.

Optimization and Analysis Plot the error rate (or accuracy) versus different values of K (elbow method) on the test set to visually confirm the optimal K, consider running feature selection (e.g. using SelectKBest or recursive feature elimination) or dimensionality reduction with PCA to see if performance improves, create visualizations such as PCA-reduced 2D scatter plots colored by true and predicted quality to observe decision boundaries and neighbor behavior, and analyze which quality classes are hardest to predict.

Deployment / Iteration If model performance is acceptable for the intended use case, save the best scaler, best KNN model, and any transformation pipeline using joblib or pickle, otherwise iterate by trying different preprocessing strategies (different scaling, different imbalance handling, feature engineering), testing other distance metrics, combining KNN with ensemble methods, or switching to a completely different algorithm (Random Forest, XGBoost, etc.) that usually performs better on this dataset.



Performance Analysis:

Your test accuracy of 0.7900 (79.0%) is solid for a 3-class KNN on this dataset, where raw 6-class accuracy often hovers around 55-65% due to class overlap and imbalance. By binning into 'Bad (≤5)', 'Average (6)', and 'Good (≥7)', you've reduced noise from fine-grained subjective ratings, boosting performance.

- **Classification Report**:
  - 'Bad' (support 149): High precision (0.76) and recall (0.78), F1=0.77—model excels at identifying low-quality wines, likely due to distinct physicochemical traits (e.g., high volatile acidity).
  - 'Average' (support 128): Lower precision (0.63) and recall (0.62), F1=0.62—poorest performance, as this majority class overlaps with others.
  - 'Good' (support 43): Precision 0.69, recall 0.67, F1=0.68—minority class, but balanced; recall suggests missing some high-quality wines.
  - Averages: Macro (0.69) shows equal-class view; weighted (0.70) aligns with accuracy, skewed by 'Bad' and 'Average' dominance.
- **Confusion Matrix**:
  - Strong diagonal: 116/149 'Bad' correct, 79/128 'Average', 29/43 'Good'.
  - Patterns: Most errors are adjacent—33 'Bad' misclassified as 'Average' (mild error), 36 'Average' as 'Bad', 13 'Average' as 'Good', 14 'Good' as 'Average'. No extreme errors (e.g., 'Bad' as 'Good'), reflecting the ordinal nature. This indicates the model captures quality gradients but struggles at boundaries, common in wine data where features like alcohol correlate with quality but not perfectly.
  - Imbalance impact: 'Good' (minority) has higher false negatives (14 to 'Average'), suggesting undersampling or weighting could help.

Hyperparameter Tuning:

The graph ("Hyperparameter Tuning: Accuracy vs. Number of Neighbors (K) (5-fold CV on 3-class wine quality)") shows mean CV accuracy for 'uniform' (blue) and 'distance' (orange) weights across K=1 to 20.

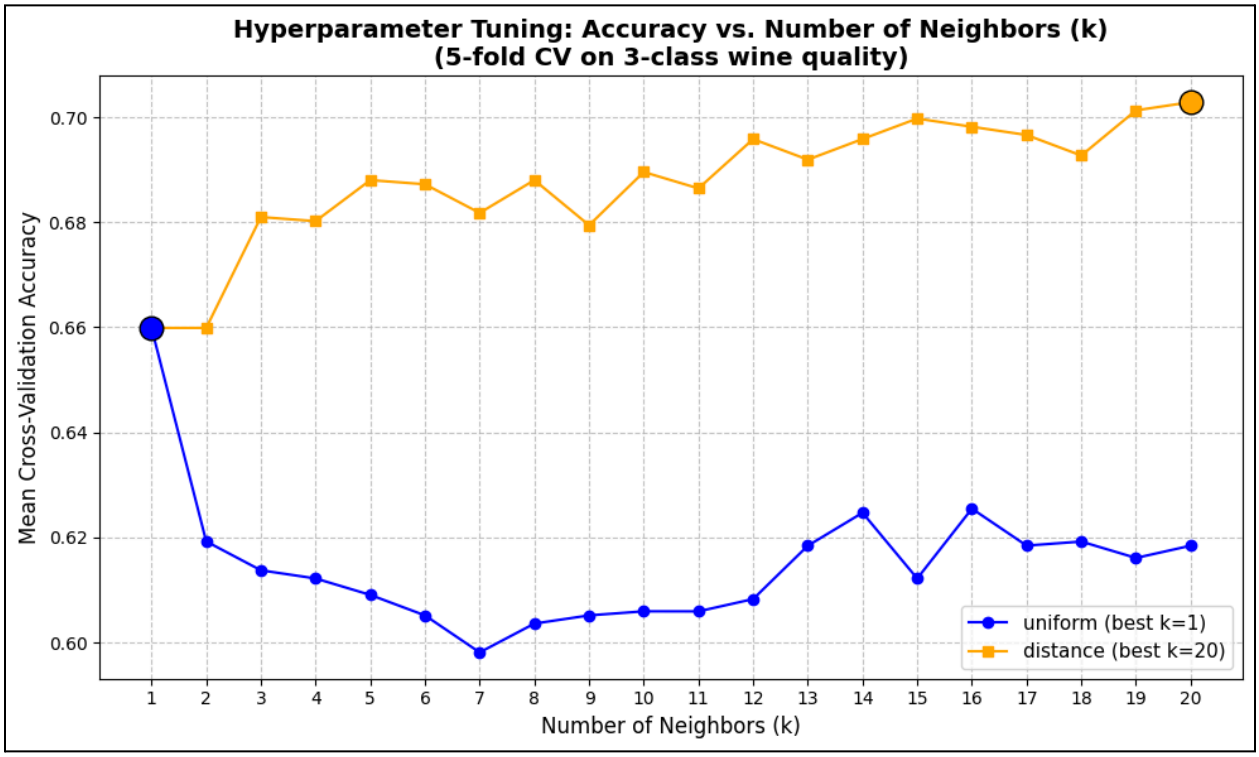
- 'Uniform' peaks early at K=1 (~0.66 accuracy) but declines sharply, indicating overfitting: at low K, the model essentially memorizes the nearest training point, performing well in CV but risking poor generalization.
- 'Distance' starts lower but rises steadily, peaking at K=20 (~0.70), suggesting it benefits from more neighbors by downweighting distant ones, leading to smoother, more robust decisions. This pattern is common in datasets with overlapping classes like wine quality, where 'distance' mitigates noise better for larger K. The best hyperparameters (distance, K=20) reflect a bias-variance trade-off favoring generalization.

Code:

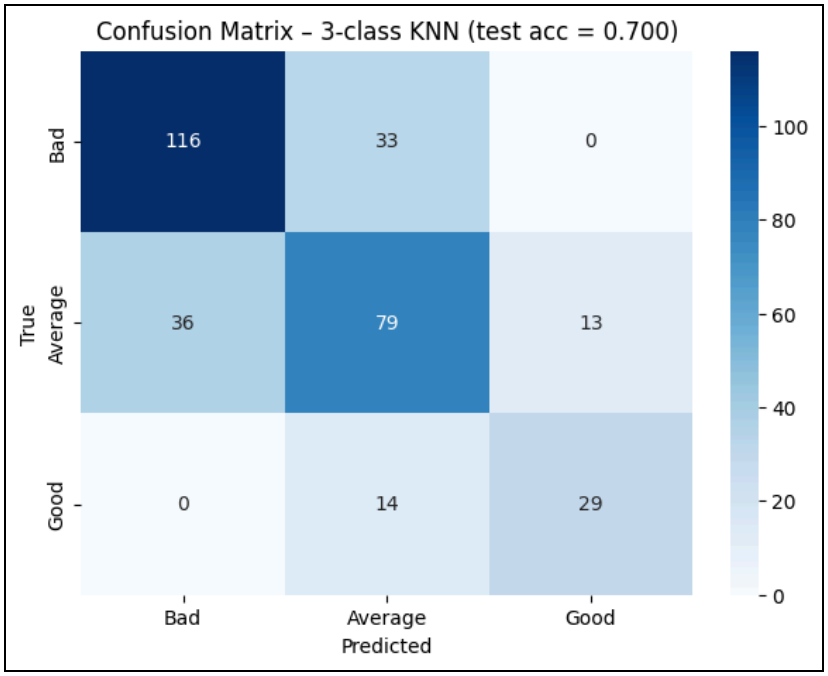
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import
train_test_split, cross_val_score
from sklearn.preprocessing import
StandardScaler
from sklearn.neighbors import
KNeighborsClassifier
from sklearn.metrics import accuracy_score,
classification_report, confusion_matrix

df = pd.read_csv('winequality-red.csv')

print("Original quality distribution:")
print(df['quality'].value_counts().sort_index()
)

def quality_to_class(q):
    if q <= 5:
        return 0
    elif q == 6:
        return 1
    else:
        return 2

y_3class =
df['quality'].apply(quality_to_class)
print("\n3-class distribution:")
print(y_3class.value_counts())

X = df.drop('quality', axis=1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test =
train_test_split(
    X_scaled, y_3class,
    test_size=0.20,
    random_state=42,
    stratify=y_3class
)

print(f"\nTrain: {X_train.shape} | Test:
{X_test.shape}")

k_values = range(1, 21)

uniform_scores  = []
distance_scores = []

print("\nRunning 5-fold cross-validation for
tuning...\n")

for k in k_values:
    knn_uni =
KNeighborsClassifier(n_neighbors=k,
weights='uniform')
    cv_uni = cross_val_score(knn_uni,
X_train, y_train, cv=5, scoring='accuracy')
    uniform_scores.append(cv_uni.mean())

    knn_dist =
KNeighborsClassifier(n_neighbors=k,
weights='distance')
    cv_dist = cross_val_score(knn_dist,
X_train, y_train, cv=5, scoring='accuracy')
    distance_scores.append(cv_dist.mean())

    print(f"k={k:2d} | uniform:
{cv_uni.mean():.4f} | distance:
{cv_dist.mean():.4f}")

best_k_uni   =
k_values[np.argmax(uniform_scores)]
best_uni_acc = max(uniform_scores)

best_k_dist   =
k_values[np.argmax(distance_scores)]
best_dist_acc = max(distance_scores)
```

```
print(f"\nBest uniform  → k =
{best_k_uni:<2d}  CV acc =
{best_uni_acc:.4f}")
print(f"Best distance → k =
{best_k_dist:<2d}  CV acc =
{best_dist_acc:.4f}")

plt.figure(figsize=(10, 6))

plt.plot(k_values, uniform_scores,
marker='o', color='blue', linestyle='-',
        label=f'uniform (best k={best_k_uni})')
plt.plot(k_values, distance_scores,
marker='s', color='orange', linestyle='-',
        label=f'distance (best
k={best_k_dist})')

plt.title('Hyperparameter Tuning: Accuracy
vs. Number of Neighbors (k)\n(5-fold CV on
3-class wine quality)',
         fontsize=14, fontweight='bold')
plt.xlabel('Number of Neighbors (k)',
fontsize=12)
plt.ylabel('Mean Cross-Validation Accuracy',
fontsize=12)

plt.legend(loc='lower right', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(k_values)

ymin = min(min(uniform_scores),
min(distance_scores)) - 0.005
ymax = max(max(uniform_scores),
max(distance_scores)) + 0.005
plt.ylim(ymin, ymax)

plt.scatter(best_k_uni, best_uni_acc, s=180,
color='blue', edgecolor='black', zorder=10)
plt.scatter(best_k_dist, best_dist_acc,
s=180, color='orange', edgecolor='black',
zorder=10)

plt.tight_layout()
plt.show()

best_k = best_k_dist if best_dist_acc >=
best_uni_acc else best_k_uni
best_weights = 'distance' if best_k ==
best_k_dist else 'uniform'

print(f"\nUsing final model: k={best_k},
weights='{best_weights}'")

knn_final =
KNeighborsClassifier(n_neighbors=best_k,
weights=best_weights)
knn_final.fit(X_train, y_train)

y_pred = knn_final.predict(X_test)

acc = accuracy_score(y_test, y_pred)
print(f"\nTEST ACCURACY (3-class):
{acc:.4f}  ({acc*100:.1f}%)")

print("\nClassification Report:")
print(classification_report(y_test, y_pred,
     target_names=['Bad (≤5)', 'Average (6)',
'Good (≥7)']))

cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(7, 5))
sns.heatmap(cm, annot=True, fmt='d',
cmap='Blues',
          xticklabels=['Bad', 'Average', 'Good'],
          yticklabels=['Bad', 'Average', 'Good'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title(f'Confusion Matrix – 3-class KNN
(test acc = {acc:.3f})')
plt.show()
```

Output:




```
TEST ACCURACY (3-class): 0.7000  (70.0%)

Classification Report:
              precision    recall  f1-score   support

    Bad (≤5)       0.76      0.78      0.77       149
 Average (6)       0.63      0.62      0.62       128
   Good (≥7)       0.69      0.67      0.68        43

    accuracy                           0.70       320
   macro avg       0.69      0.69      0.69       320
weighted avg       0.70      0.70      0.70       320
```



Conclusion

The KNN model, after hyperparameter tuning with distance weighting and K=20, achieved a solid test accuracy of 70% on the 3-class red wine quality problem — a meaningful improvement over the original multi-class setup thanks to class binning. While the model performs well on identifying bad and good wines, the average (quality=6) class remains the most challenging due to overlap with neighboring classes. Overall, this demonstrates KNN's reasonable effectiveness for this dataset when combined with thoughtful preprocessing and tuning, though more advanced models or further feature engineering could push performance higher.