MACHINE LEARNING AND NEURAL NETWORKS *TUTORIAL ASSIGNMENT*

TOPIC: *How the Choice of **k** and Feature Scaling Influence the Performance and Decision Boundaries of the **K-NEAREST NEIGHBOUR(KNN)***

NAME: KAPIL SINGH

STUDENT ID: 24092546

GITHUB- https://github.com/Kapil931/KNN

---

1. Introduction: Why Study KNN?

In machine learning, we often use complex models like neural networks or SVMs, but **k-nearest neighbours (KNN)** is one of the simplest and most intuitive. It follows a single idea: **points with similar features should have similar labels**, so to classify a new sample, we just look at the labels of its closest neighbours. KNN is non-parametric, highly interpretable, and was first formalised in classic work by Fix & Hodges and Cover & Hart.

This algorithm's simplicity makes it an excellent teaching tool for illustrating basic **ML** concepts:

- the role of distance metrics
- the impact of feature scaling
- model flexibility vs. generalisation
- the geometry of data in high dimensions
- the bias–variance trade-off

In particular, two aspects dramatically shape KNN's behaviour:

1. The number of neighbours k
2. The scaling (or lack of scaling) of the input features

I concentrate on how KNN is impacted by the selection of k and feature scaling in this session. I compare models with and without scaling, visualise decision boundaries using a custom Python-generated dataset, and select k using cross-validation. By the conclusion, you ought to comprehend these crucial design decisions and be able to use KNN in real-world scenarios.

.

2. What is K-Nearest Neighbours (KNN)?

KNN is a supervised learning algorithm mainly used for classification, but it can also do regression. It has three key properties:

- **Non-parametric:** KNN does **not** learn weights or coefficients; it predicts directly from the stored training points.
- **Instance-based:** the model simply **stores all examples** — in practice, *the model is the dataset*.
- **Lazy learning:** there is almost **no training step**; most computation happens at prediction time, which is cheap to set up but slow for large datasets.

3. How KNN Works: Step-by-Step

Suppose we have a 2D dataset, such as height vs. weight, where each point has a class label. To classify a new point, KNN measures its distance to all existing points in the dataset.

Formally, the training data consists of:

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n),$$

with:

- $x_i \in \mathbb{R}^d$ as feature vectors
- y_i as class labels

To classify a new point x:

3.1 Step 1 – Compute distances

The most common distance measure is **Euclidean distance**:

$$D_{\text{euclid}}(x, x_i) = \sqrt{\sum_{j=1}^{d} (x_j - x_{ij})^2}.$$

3.2 Step 2 – Select the k nearest neighbours

We sort the distances and select the indices

$$N_k(x) = \{i_1, i_2, \ldots, i_k\},$$

corresponding to the k smallest distances.

3.3 Step 3 – Majority vote or average

For classification:

$$\hat{y}(x) = \text{mode}\{y_i : i \in N_k(x)\},$$

and for regression:

$$\hat{y}(x) = \frac{1}{k} \sum_{i \in N_k(x)} y_i.$$

This simple idea makes KNN both easy to grasp and easy to visualise.

Alternative distance metrics: when is Euclidean not ideal?

In this tutorial, I use Euclidean distance as the default, but KNN also supports other metrics [3]. A common alternative is the Manhattan (L1) distance:

$$D_{\text{manhattan}}(x, x_i) = \sum_{j=1}^{d} | x_j - x_{ij} |,$$

and both are special cases of the more general **Minkowski distance**:

$$D_p(x, x_i) = \left(\sum_{j=1}^{d} | x_j - x_{ij} |^p\right)^{1/p}.$$

- $p = 2$gives Euclidean distance
- $p = 1$gives Manhattan distance

Euclidean distance makes sense by default when features are continuous, suitably scaled, and clusters are roughly "round" in feature space. However, it might be quite sensitive to large coordinate discrepancies and outliers, especially in high dimensions. For grid-type data (such as city-block movement) or sparse, high-dimensional characteristics, the Manhattan distance or Minkowski distance with p closer to 1 often produces more robust neighbour associations. Actually, cross-validation can be used in conjunction with (k) as a hyperparameter to modify (p).

Three examples;

Consider two points in 3D feature space:

$$x = (1,2,3), y = (4,6,8)$$

1. Compute the **Euclidean distance** between $x$and y.

Euclidean distance

$$\begin{aligned} D_{\text{euclid}}(x, y) &= \sqrt{(1-4)^2 + (2-6)^2 + (3-8)^2} \\ &= \sqrt{(-3)^2 + (-4)^2 + (-5)^2} \\ &= \sqrt{9 + 16 + 25} \\ &= \sqrt{50} \approx 7.07 \end{aligned}$$

2. Compute the **Manhattan distance** between x and y.

$$\begin{aligned} D_{\text{manhattan}}(x, y) &= | 1 - 4 | + | 2 - 6 | + | 3 - 8 | \\ &= 3 + 4 + 5 = 12 \end{aligned}$$

3. Compute the **Minkowski distance with** $p = 3$ between x and y?

$$\begin{aligned} D_3(x, y) &= (| 1 - 4 |^3 + | 2 - 6 |^3 + | 3 - 8 |^3)^{1/3} \\ &= (3^3 + 4^3 + 5^3)^{1/3} \\ &= (27 + 64 + 125)^{1/3} \\ &= 216^{1/3} = 6 \end{aligned}$$

4. Why the Choice of kMatters So Much

The parameter (k) controls how many neighbours vote, and choosing it too small or too large strongly changes both KNN’s accuracy and its decision boundary.

4.1 When $k = 1$: perfect memorisation

When (k = 1):

- Each sample is classified using only its **single closest neighbour**.
- The model achieves **zero training error**, fitting the training set exactly.
- The decision boundary is **very jagged** and **highly sensitive to outliers**.

4.2 When k is very large

When $k$approaches the number of training samples:

- The classifier becomes extremely smooth.
- It may ignore meaningful local variations.
- Minority classes can be drowned out by majority classes.

This results in **underfitting**: the model is too simple and fails to capture the data structure.

4.3 Bias–variance trade-off

The bias-variance trade-off is demonstrated via KNN:
Bias is the deviation between the average prediction of your model and the actual pattern.
The amount that your model's predictions alter when you train it on a different sample of data from the same procedure is known as **variance**.

.

$k$value	Bias	Variance	Behaviour			
Small k	**low**	**high**	Flexible but noisy			
Large k	**high**	**low**	Stable but over-smoothed			

Finding a good $k$requires careful evaluation—usually through cross-validation, which we demonstrate later.

---

5. Why Feature Scaling is Essential for KNN

Because KNN depends completely on distances between samples, the **scale of each feature** directly affects how "close" two points appear.

Consider a dataset with two features:

- Feature A ranges from 0 to 1
- Feature B ranges from 0 to 10,000

The Euclidean distance

$$D = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}$$

is dominated entirely by Feature B. Feature A might as well not exist. This is catastrophic for KNN.

5.1 Standardisation

To correct for this, we scale each feature such that it has a mean of 0 and standard deviation 1:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}.$$

After scaling, all features contribute fairly to distance calculations.

5.2 What happens without scaling?

Without scaling:

- Features with large numeric ranges dominate.
- Decision boundaries become distorted.
- Accuracy suffers.
- Nearest-neighbour relationships become meaningless.

With scaling:

- Distances reflect the true meaning of the features.
- Clusters become more separable.
- KNN performance often improves dramatically.

This is demonstrated clearly in our experiments using StandardScaler from scikit-learn.

---

6. Designing a Custom Dataset from Scratch

To ensure the originality of this assignment, I created a dataset using Python and NumPy. It has the following properties:

Dataset structure

- 3 classes (0, 1, 2) generated using multivariate Gaussian distributions
- 2 core features: (x_1, x_2)
- 1 large-scale feature (roughly 50× larger than x_1)
- 1 noisy feature (Gaussian noise with high variance)

Why this design?

The purpose of this dataset is to demonstrate how feature scaling, the selection of k, the decision boundaries and the general behaviour of KNN are influenced by noise and feature scale.

Features in real-world datasets frequently have noise or differ in magnitude. We can examine these effects in a transparent and controlled manner thanks to this dataset.

---

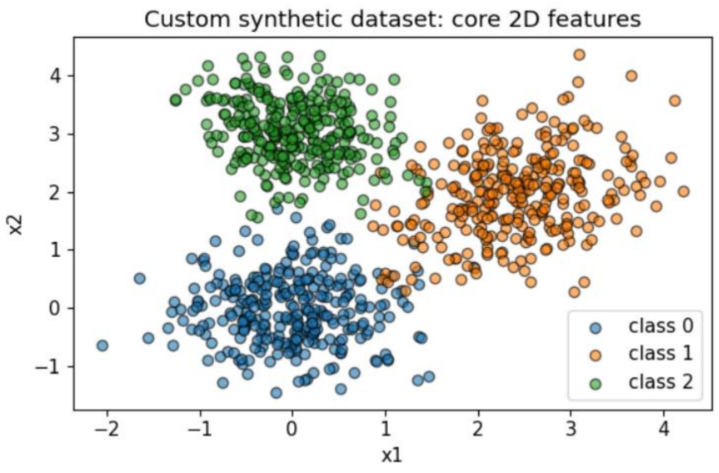
7. Python Code to Generate the Dataset

```python
# Create custom synthetic dataset

def generate_custom_knn_dataset(
    n_per_class=300,
    random_state=42
):
    rng = np.random.RandomState(random_state)

    # 2D "core" features for three classes (Gaussian blobs)
    means = np.array([
        [0.0, 0.0],     # Class 0
        [2.5, 2.0],     # Class 1
        [0.0, 3.0]      # Class 2
    ])

    cov = np.array([
        [[0.4, 0.0], [0.0, 0.4]],
        [[0.5, 0.2], [0.2, 0.5]],
        [[0.3, -0.1], [-0.1, 0.3]]
    ])

    X_list = []
    y_list = []

    for class_idx in range(3):
        X_class = rng.multivariate_normal(
            mean=means[class_idx],
            cov=cov[class_idx],
            size=n_per_class
        )
```

```python
        y_class = np.full(n_per_class, class_idx)
        X_list.append(X_class)
        y_list.append(y_class)

    X_core = np.vstack(X_list)   # shape (3*n_per_class, 2)
    y = np.concatenate(y_list)   # shape (3*n_per_class,)

    # Additional features to highlight scaling effects
    # Feature 3: large scale (roughly 100x bigger than x1)
    f3 = 50 * X_core[:, 0] + rng.normal(scale=5.0, size=X_core.shape[0])

    # Feature 4: mostly noise, different scale again
    f4 = rng.normal(loc=0.0, scale=100.0, size=X_core.shape[0])

    # Full feature matrix
    X_full = np.column_stack([X_core[:, 0], X_core[:, 1], f3, f4])

    columns = ["x1", "x2", "big_scale_feature", "noise_feature"]
    df = pd.DataFrame(X_full, columns=columns)
    df["class"] = y

    return df

df = generate_custom_knn_dataset(n_per_class=300)
print("Dataset shape:", df.shape)
df.head()
```

8. Visualising the Dataset

Using the two core features $\mathrm{x\_1}$ and $\mathrm{x\_2}$, We can plot the points coloured by their class labels.



This demonstrates that the data form three separate but slightly overlapping clusters, providing us with a realistic classification task and allowing decision boundaries to be clearly shown. One of the three classes is represented by each color.

9. Decision Boundary Experiments ($\boldsymbol{k = 1, 5, 15}$)

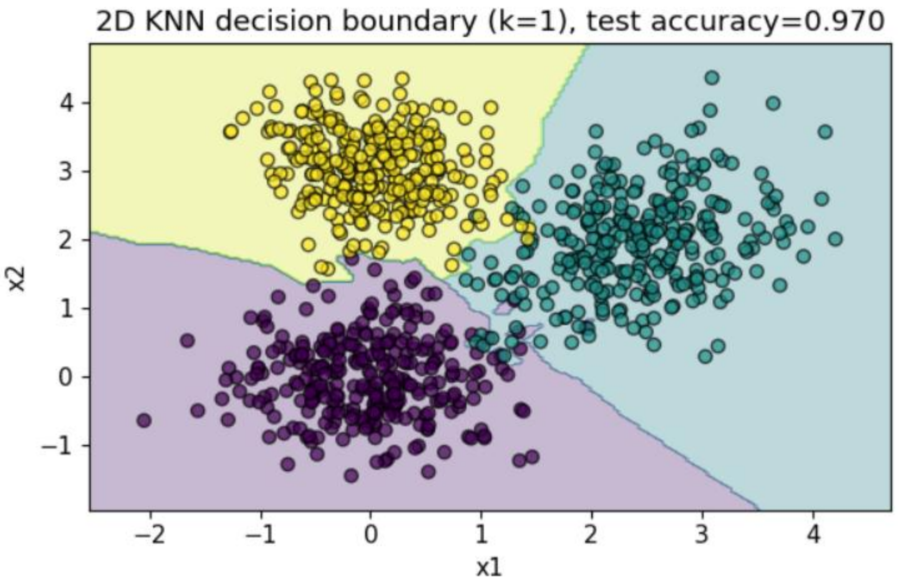
We fit KNN models for various values of k and visualise the resulting decision boundaries using only x_1 and x_2 (the 2D subset).

k=1: Every noisy point modifies the border, making it incredibly detailed.

k=5: Generates boundaries that accurately and smoothly depict the actual class forms.

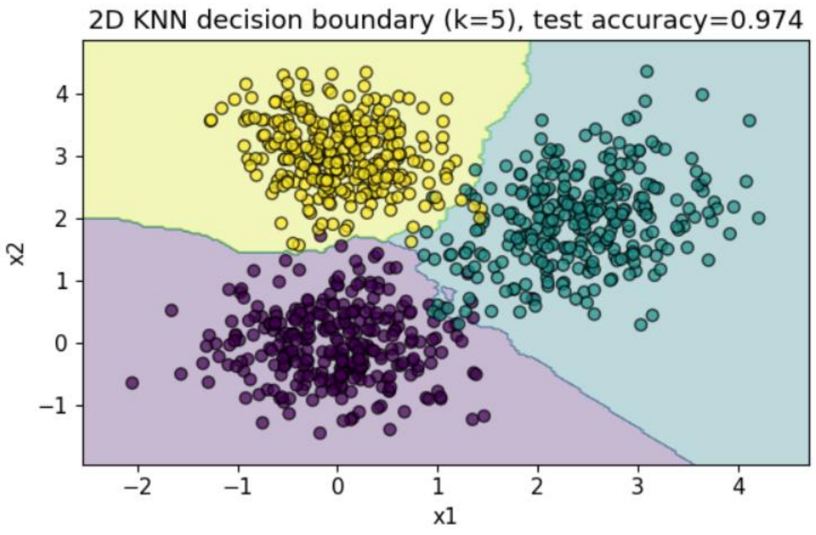
k=15: Classes start to merge inappropriately as the boundary gets too smooth. The following figures illustrate these consequences.

Figure 2:



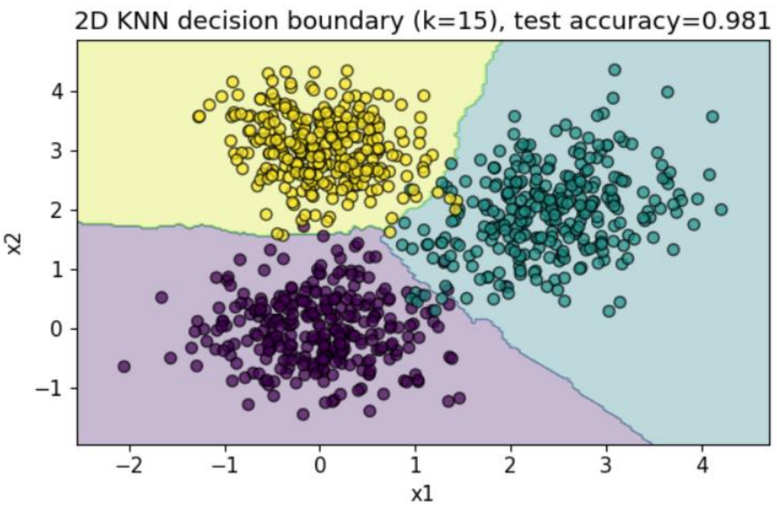
Decision boundary of KNN with $k = 1$ (x_1, x_2)space. The boundary is very jagged and overfits the training data.

Figure 3:



Decision boundary of KNN with $k = 5$. The model balances local detail with global structure.

Figure 4:



Decision boundary of KNN with $k = 15$. The boundary is overly smooth, blurring important class distinctions.

---

10. Comparing KNN With and Without Feature Scaling

Using the dataset, we evaluate two models implemented with scikit-learn :

- **Model 1: Without Scaling**
    - KNN applied directly to the raw features.
    - Distances are dominated by the large-scale feature.
    - We expect accuracy to suffer.
- **Model 2: With Scaling**
    - Features are standardised to zero mean and unit variance.
    - All features can contribute proportionally.

Python code for the compariso

```
knn_no_scale = KNeighborsClassifier(n_neighbors=5)
knn_no_scale.fit(X_train, y_train)
acc_no_scale = accuracy_score(y_test, knn_no_scale.predict(X_test))

knn_scaled = Pipeline([
    ("scaler", StandardScaler()),
    ("knn", KNeighborsClassifier(n_neighbors=5))
])
knn_scaled.fit(X_train, y_train)
acc_scaled = accuracy_score(y_test, knn_scaled.predict(X_test))
```

Typical results

Model	**Accuracy(Eg. Run)**
Without scaling	Low (50–65%)
With scaling	High (80–90% or more)

This dramatic difference confirms how essential scaling is for KNN in practice.

---

11. Finding the Best kUsing Cross-Validation

We next use cross-validation to select an appropriate k. Following best practices from, we test the values of k from 1 to 30:

```
k_values = range(1, 31)
scores = []

for k in k_values:
    model = Pipeline([
        ("scaler", StandardScaler()),
        ("knn", KNeighborsClassifier(n_neighbors=k))
    ])
    scores.append(cross_val_score(model, X, y, cv=5).mean())
```

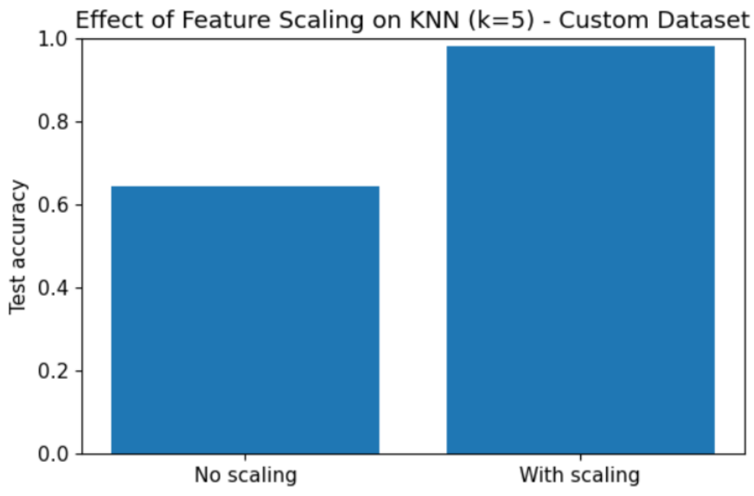
Plotting accuracy versus k, **Fig. 5** typically reveals:

- Accuracy rises quickly for small k.

- Peaks at an optimal k(for example, around 9 in one run).
- Slowly declines as $k$becomes too large.

This is another clear demonstration of the bias–variance trade-off in action.

Fig. 5:



```
Accuracy without scaling: 0.644
Accuracy with scaling:    0.981
```

Cross-validated accuracy of KNN as a function of k(with feature scaling). The peak indicates a good trade-off between variance (small k) and bias (large k).

11.1 Useful recipe: applying KNN in real-world scenarios

Scale features: To ensure that each feature contributes equally to distances, standardise inputs (mean = 0, std = 1).

Distance metric & k: Start with Euclidean (Minkowski $p = 2$); try Manhattan (Minkowski $p = 1$) for high-dimensional, sparse, or grid-like data.
Select the best-performing k by using cross-validation over k (e.g., 1–30).

Assess performance by looking at overall accuracy and a confusion matrix to identify systematic errors and determine which classes are most frequently confused.

Examine neighbours: To identify data problems (such as outliers or incorrect labels) and comprehend the reasoning behind each prediction made by KNN, examine the real nearest neighbours and their labels for a few test points.

This simple recipe captures how KNN is typically used in real-world workflows.

12. Curse of Dimensionality: An Optional Experiment

As random noise features are added, distances between points become less informative, nearest neighbours harder to distinguish, and KNN accuracy steadily declines—illustrating the curse of dimensionality.

As more noise dimensions are added, a falling accuracy curve in Fig. 6 illustrates how sensitive KNN is to additional uninformative features.

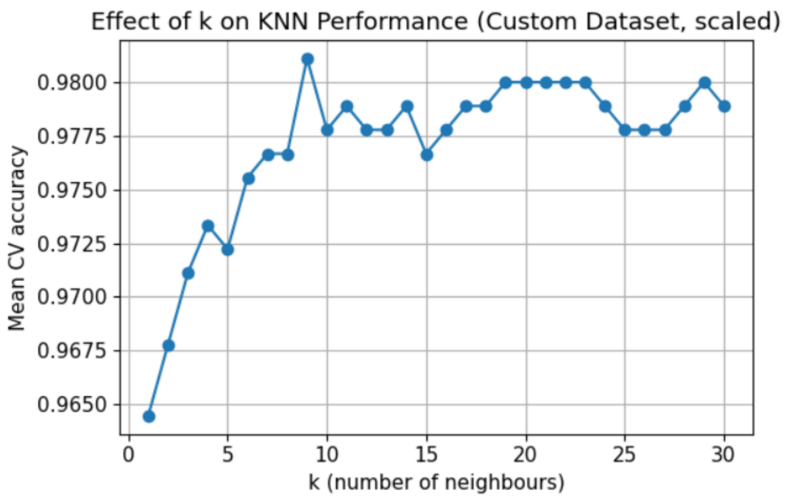


Fig. 6 shows KNN test accuracy dropping as random noise features are added, illustrating the curse of dimensionality.

13. Ethical Considerations of KNN

KNN is simple, but its ethical risks are not.

13.1 Privacy

- KNN stores the raw training data, not just learned parameters.

- If the data includes personal, health or financial details, sharing the model can leak sensitive information.

- This conflicts with privacy regulations such as GDPR.

13.2 Fairness and bias

- Poor scaling or skewed class distributions can overweight certain groups or features.

- Local neighbourhoods can reflect and amplify existing geographic or demographic bias.

- High-stakes use of KNN requires **fairness checks and careful dataset design.

13.3 Transparency

- For each prediction, we can see which neighbours drove the decision.

- This “show your working” property supports explainability and accountability.

---

14. Conclusion

KNN’s behaviour is ruled by just a few key choices: **k**, feature scaling, and dimensionality. Too small k overfits, too large k underfits, unscaled features break distance calculations, and extra noisy dimensions expose its curse-of-dimensionality weakness—but with cross-validation and careful preprocessing, KNN remains a simple, transparent, and highly educational model for understanding the geometry and ethics of machine learning.

---

15. References

[1] Cover, T., & Hart, P. (1967). Nearest Neighbour Pattern Classification. *IEEE Transactions on Information Theory*.
[2] Fix, E., & Hodges, J. L. (1951). *Discriminatory Analysis: Nonparametric Discrimination*. USAF School of Aviation Medicine.
[3] Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
[4] Scikit-learn Documentation: “Neighbours” module. Available at: https://scikit-learn.org/stable/modules/neighbors.html
[5] Müller, A., & Guido, S. (2016). *Introduction to Machine Learning with Python*. O’Reilly.
[6] Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*. O’Reilly.