Lab 2: Classification Using KNN and RNN Algorithms

Nitish Dhinaharan

University of the Cumberlands

MSCS-634-B01 Advanced Big Data and Data Mining

Dr. Satish Penmatsa

November 09, 2025

Lab 2: Classification Using KNN and RNN Algorithms

Github Link

https://github.com/ndhinaharan36295/MSCS-634_Lab-2

Step 1: Load and Prepare the Dataset

```
# Step 1: Load and Prepare the Dataset

# 1) load wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# 2) quick exploration
df = pd.DataFrame(X, columns=wine.feature_names)
df['target'] = y

print("Feature names:", wine.feature_names)
print("Target names:", wine.target_names)
print("\nClass distribution:")
print(df['target'].value_counts())

# 3) split dataset into train-test (80/20)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

```
Feature names: ['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins', 'color_intensity', 'hue',
Target names: ['class_0' 'class_1' 'class_2']

Class distribution:
target
1    71
0    59
2    48
Name: count, dtype: int64
```

Step 2: Implement K-Nearest Neighbors (KNN)

```
# Step 2: Implement K-Nearest Neighbors (KNN)

k_values = [1, 5, 11, 15, 21]
knn_results = {}

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    knn_results[k] = acc

knn_results
```

```
{1: 0.7777777777777778,
 5: 0.8055555555555556,
 11: 0.8055555555555556,
 15: 0.8055555555555556,
 21: 0.8055555555555556}
```

Step 3: Implement Radius Neighbors (RNN)

```
# Step 3: Implement Radius Neighbors (RNN) with scaling

# scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

radius_values = [350, 400, 450, 500, 550, 600]
rnn_results = {}

for r in radius_values:
    rnn = RadiusNeighborsClassifier(radius=r, outlier_label=None)
    rnn.fit(X_train_scaled, y_train)
    y_pred = rnn.predict(X_test_scaled)
    acc = accuracy_score(y_test, y_pred)
    rnn_results[r] = acc

rnn_results
```

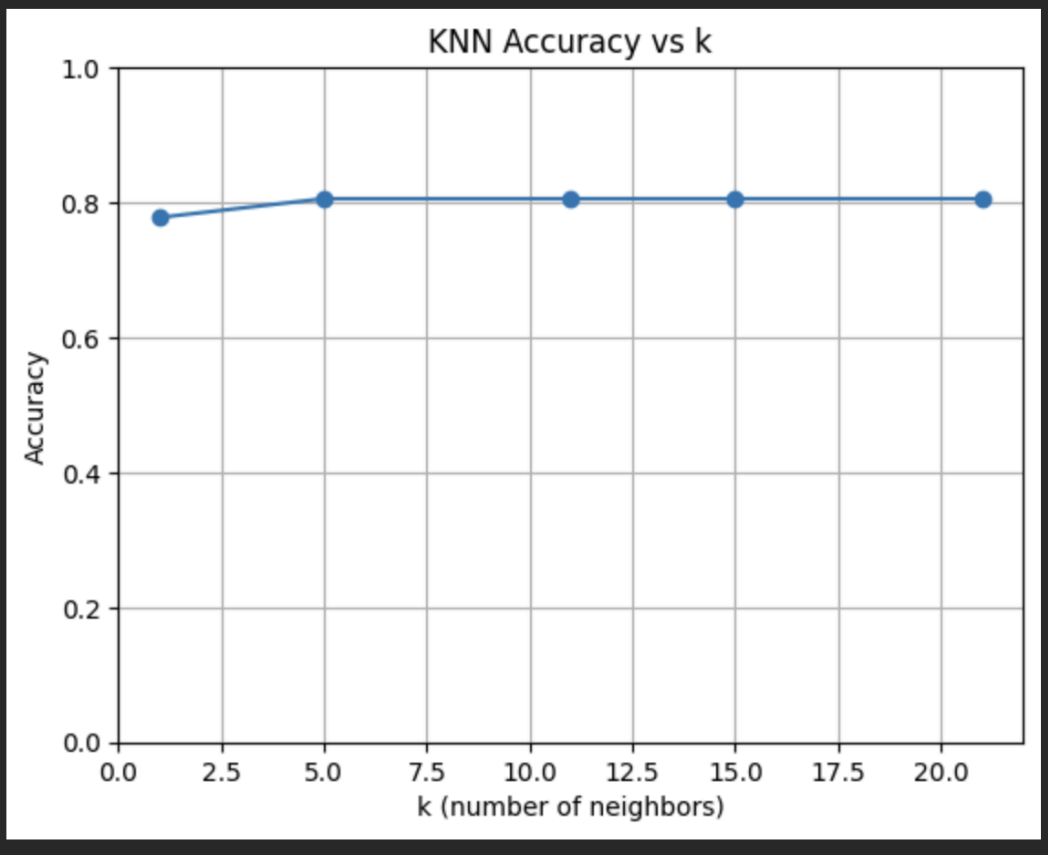
```
{350: 0.3888888888888889,
 400: 0.3888888888888889,
 450: 0.3888888888888889,
 500: 0.3888888888888889,
 550: 0.3888888888888889,
 600: 0.3888888888888889}
```

Step 4: Visualize and Compare Results

KNN plot:

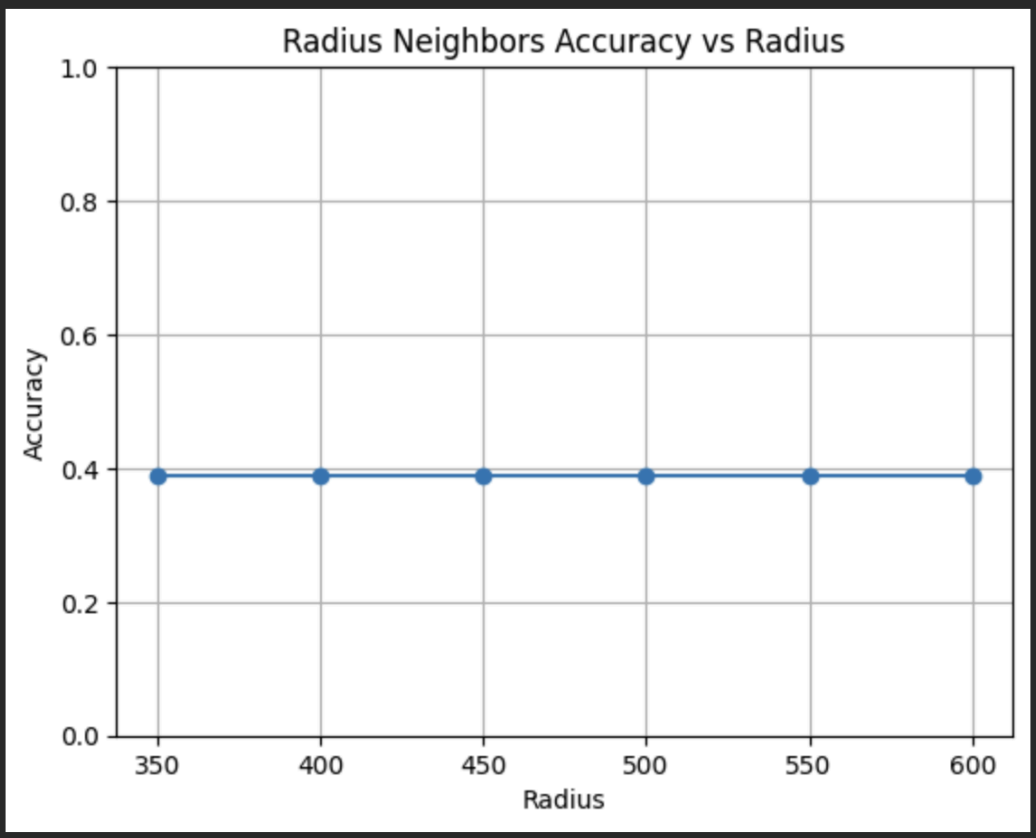
```
# Step 4: Visualize and Compare Results

# KNN plot
plt.figure()
plt.plot(list(knn_results.keys()), list(knn_results.values()), marker='o')
plt.title("KNN Accuracy vs k")
plt.xlabel("k (number of neighbors)")
plt.ylabel("Accuracy")
plt.ylim(0, 1)
plt.grid(True)
plt.show()
```



RNN plot:

```
# RNN plot
plt.figure()
plt.plot(list(rnn_results.keys()), list(rnn_results.values()), marker='o')
plt.title("Radius Neighbors Accuracy vs Radius")
plt.xlabel("Radius")
plt.ylabel("Accuracy")
plt.ylim(0, 1)
plt.grid(True)
plt.show()
```



From the results shown in the plots, K-Nearest Neighbors (KNN) consistently achieved higher and more stable accuracy (around 0.80–0.82) across all tested values of k. This indicates that the KNN classifier performs well on the Wine dataset and is relatively insensitive to moderate changes in the number of neighbors. The slight improvement from k = 1 to k = 5

suggests that incorporating a few additional neighbors helps smooth out noise while maintaining strong predictive performance.

In contrast, the Radius Neighbors (RNN) classifier performed significantly worse, with accuracy plateauing near 0.39 for all tested radius values (350–600). This flat and low accuracy trend indicates that the chosen radius values were too large relative to the feature scales, causing the model to include almost all training points within each query's neighborhood. As a result, RNN lost its locality sensitivity and effectively produced majority-class predictions regardless of input features. This behavior demonstrates RNN's high sensitivity to both feature scaling and the chosen radius parameter.

Therefore, KNN is preferable when the dataset has a moderate number of samples, balanced classes, and uniform feature scaling. It provides more predictable results, requires only one intuitive hyperparameter (k), and performs robustly without heavy preprocessing. RNN may be preferable only when the data distribution is non-uniform or exhibits variable density. For example, in spatial or sensor datasets, and after proper feature scaling. In those cases, carefully tuning the radius can adapt the model to local density variations. KNN's stable accuracy and RNN's flat low performance clearly show that KNN is the more reliable choice for the Wine dataset without additional feature normalization or advanced hyperparameter tuning.