prac4

October 18, 2024

Aim: Program for Classification using KNN algorithm

Theory: ### Classification Using K-Nearest Neighbors (KNN) Algorithm: Theory

The **K-Nearest Neighbors (KNN)** algorithm is a simple, yet powerful machine learning technique used for both classification and regression tasks. It is a type of **instance-based learning** where the algorithm does not learn an explicit model during the training phase but rather stores all the training instances and makes predictions based on the local neighborhood of a new data point.

Here's a breakdown of how KNN works for classification.

0.0.1 1. Basic Concept of KNN

KNN classifies a data point based on the majority class of its **k-nearest neighbors** in the feature space. It assumes that similar data points tend to be near each other. For a given data point, the algorithm looks at the (k) closest data points (neighbors) and assigns the majority class among them to the new point.

0.0.2 2. How KNN Works

The KNN algorithm for classification follows these steps:

1. **Choose a Value for K**: Decide how many neighbors (k) to consider for classifying a new data point.
2. **Calculate Distance**: For a new data point (to be classified), compute the distance between this point and every other point in the training dataset. Common distance measures include:
    - **Euclidean Distance**: The straight-line distance between two points.
    - **Manhattan Distance**: The distance along axes at right angles.
    - **Minkowski Distance**: A generalization of both Euclidean and Manhattan distances.
3. **Find Nearest Neighbors**: Identify the (k) points in the training data that are closest to the new data point based on the calculated distance.
4. **Vote for the Majority Class**: Among the (k) nearest neighbors, count the number of instances for each class. The class that has the most votes is assigned to the new data point.
5. **Class Assignment**: The new data point is classified into the majority class of the (k) nearest neighbors.

0.0.3 3. Choosing the Value of K

- **Small K**: When (k) is small, the classifier becomes more sensitive to the nearest points. This can lead to overfitting, where the model becomes sensitive to noise or outliers in the training data.
- **Large K**: When (k) is large, the classifier considers more neighbors and becomes more generalized. However, too large a (k) can cause underfitting, where the model oversmooths and cannot capture patterns well.

A common approach is to use **cross-validation** to select the optimal (k) by testing different values on a validation set.

0.0.4 4. Distance Metrics

The performance of the KNN algorithm largely depends on the distance metric used. Common metrics include:

- **Euclidean Distance**: Most commonly used for continuous variables, calculated as the straight-line distance between points.
- **Manhattan Distance**: Useful when the feature space is structured more like a grid.
- **Hamming Distance**: Often used when working with categorical variables.

The choice of distance metric should be informed by the nature of the problem and the type of features in the dataset.

0.0.5 5. Advantages of KNN

- **Simplicity**: KNN is easy to understand and implement. There's no training phase, making it a simple algorithm that can be deployed without significant computational overhead.
- **No Assumptions**: KNN makes no assumptions about the underlying data distribution. It is a non-parametric algorithm, meaning it works well even when the data is not linearly separable or follows any specific pattern.
- **Flexibility**: KNN can be used for both classification and regression, though it is more commonly applied to classification tasks.

0.0.6 6. Disadvantages of KNN

- **Computationally Expensive**: Since KNN stores all the training data and makes predictions by calculating distances to each point, the algorithm can be slow for large datasets.
- **Sensitive to the Scale of Data**: Features with larger ranges may dominate the distance calculation. Normalization or standardization of data is often required to prevent this.
- **Sensitive to Outliers**: KNN can be affected by noisy data or outliers, especially when (k) is small. Outliers can mislead the voting process.

0.0.7 7. Practical Considerations

- **Data Normalization**: Since KNN is distance-based, it's crucial to normalize or standardize the features so that all features contribute equally to the distance calculation.
- **Dimensionality**: In high-dimensional spaces, KNN can suffer from the "curse of dimensionality" because distances between points become less meaningful. Dimensionality reduction techniques like PCA can be applied before using KNN to mitigate this problem.

0.0.8 8. Use Cases of KNN

- **Image Recognition**: KNN is used in image classification tasks, where it classifies images based on pixel intensity or feature vectors.
- **Recommendation Systems**: KNN can be used to recommend items to users based on their similarity to other users (user-based) or to similar items (item-based).
- **Medical Diagnosis**: KNN can be applied to medical data to classify a patient into a disease category based on similarity to other patients' records.

[ ]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
 ↪installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
 ↪docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
 ↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that␣
 ↪gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
 ↪outside of the current session
```

```
/kaggle/input/apple-quality/apple_quality.csv
```

[ ]:
```
df = pd.read_csv("/kaggle/input/apple-quality/apple_quality.csv")
```

[ ]:
```
df.describe()
```

[ ]:

	A_id	Size	Weight	Sweetness	Crunchiness	\
count	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	
mean	1999.500000	-0.503015	-0.989547	-0.470479	0.985478	
std	1154.844867	1.928059	1.602507	1.943441	1.402757	
min	0.000000	-7.151703	-7.149848	-6.894485	-6.055058	
25%	999.750000	-1.816765	-2.011770	-1.738425	0.062764	
50%	1999.500000	-0.513703	-0.984736	-0.504758	0.998249	
75%	2999.250000	0.805526	0.030976	0.801922	1.894234	
max	3999.000000	6.406367	5.790714	6.374916	7.619852	

```
          Juiciness     Ripeness
count  4000.000000  4000.000000
mean      0.512118     0.498277
std       1.930286     1.874427
min      -5.961897    -5.864599
25%      -0.801286    -0.771677
50%       0.534219     0.503445
75%       1.835976     1.766212
max       7.364403     7.237837
```

[ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4001 entries, 0 to 4000
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   A_id         4000 non-null   float64
 1   Size         4000 non-null   float64
 2   Weight       4000 non-null   float64
 3   Sweetness    4000 non-null   float64
 4   Crunchiness  4000 non-null   float64
 5   Juiciness    4000 non-null   float64
 6   Ripeness     4000 non-null   float64
 7   Acidity      4001 non-null   object
 8   Quality      4000 non-null   object
dtypes: float64(7), object(2)
memory usage: 281.4+ KB
```

[ ]:
```
df.head(5)
```

[ ]:
```
    A_id      Size    Weight  Sweetness  Crunchiness  Juiciness  Ripeness  \
0    0.0 -3.970049 -2.512336   5.346330    -1.012009   1.844900  0.329840
1    1.0 -1.195217 -2.839257   3.664059     1.588232   0.853286  0.867530
2    2.0 -0.292024 -1.351282  -1.738429    -0.342616   2.838636 -0.038033
3    3.0 -0.657196 -2.271627   1.324874    -0.097875   3.637970 -3.413761
4    4.0  1.364217 -1.296612  -0.384658    -0.553006   3.030874 -1.303849

       Acidity Quality
0  -0.491590483    good
1  -0.722809367    good
2   2.621636473     bad
3   0.790723217    good
4   0.501984036    good
```

[ ]:
```
df['Acidity'] = pd.to_numeric(df['Acidity'], errors='coerce')
```

[ ]:
```
df['Acidity'][:5]
```

[ ]:
```
0   -0.491590
1   -0.722809
2    2.621636
3    0.790723
4    0.501984
Name: Acidity, dtype: float64
```

[ ]:
```
df.isnull().sum()
```

[ ]:
```
A_id           1
Size           1
Weight         1
Sweetness      1
Crunchiness    1
Juiciness      1
Ripeness       1
Acidity        1
Quality        1
dtype: int64
```

[ ]:
```
df.dropna(inplace = True)
```

[ ]:
```
df.value_counts('Quality')
```

[ ]:
```
Quality
good    2004
bad     1996
Name: count, dtype: int64
```

[ ]:
```
from sklearn.preprocessing import OrdinalEncoder
encoder = OrdinalEncoder(categories=[['bad', 'good']])
df['Quality'] = encoder.fit_transform(df[['Quality']])
```

[ ]:
```
df['Quality'][:5]
```

[ ]:
```
0    1.0
1    1.0
2    0.0
3    1.0
4    1.0
Name: Quality, dtype: float64
```

[ ]:
```
X = df.drop('Quality', axis = 1)
y = df['Quality']
```

[ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[ ]:
```
X_train.shape, X_test.shape
```

[ ]:
```
((3200, 8), (800, 8))
```

[ ]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[ ]:
```
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
```

[ ]:
```
knn = KNeighborsClassifier()

param_grid = {
    'n_neighbors': range(1, 21),
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski']
}

grid_search = GridSearchCV(estimator=knn, param_grid=param_grid, cv=5,␣
 ↪n_jobs=-1, scoring='accuracy')

grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Cross-Validation Accuracy:", grid_search.best_score_)

best_knn = grid_search.best_estimator_
y_pred = best_knn.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred)
print("Test Set Accuracy:", test_accuracy)
```

```
Best Parameters: {'metric': 'euclidean', 'n_neighbors': 12, 'weights':
'distance'}
Best Cross-Validation Accuracy: 0.8834375
Test Set Accuracy: 0.8925
```

0.0.9 Conclusion

The K-Nearest Neighbors algorithm is a simple yet powerful method for classification tasks. By relying on the proximity of data points, it captures the idea that similar inputs should lead to similar

outputs. Despite its simplicity, KNN can perform surprisingly well in a variety of applications, but it requires careful consideration of the value of (k), the choice of distance metric, and the preprocessing of data.