

in train_test_split() method in Python, what's the use of stratify parameter?



In the `train_test_split()` function from `sklearn.model_selection`, the `stratify` parameter ensures that the training and test sets maintain the same proportion of class labels as in the original dataset. This is particularly useful for imbalanced datasets, where some classes are underrepresented.

Usage:

python



Kopyala



Düzenle

```
from sklearn.model_selection import train_test_split X = [...] # Feature data y = [...] # Target Labels X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

Key Benefits:

- **Prevents bias:** Ensures that both the training and test sets reflect the original distribution of classes.
- **Improves model performance:** Helps models generalize better by training on a balanced dataset.
- **Avoids data leakage issues:** Prevents one class from being over- or under-represented in the test set.

If `stratify=None` (default), the split is purely random, which may lead to class imbalances in small datasets.



iris dataset

The **Iris dataset** is a well-known dataset in machine learning and statistics. It contains **150 samples** of iris flowers, categorized into **three species**:

- **Setosa**
- **Versicolor**
- **Virginica**

Each sample has **four features** (all measured in cm):

1. **Sepal length**
2. **Sepal width**
3. **Petal length**
4. **Petal width**

Using train_test_split() with stratify

Since the **Iris dataset** has three balanced classes (50 samples each), stratifying ensures that all three species are proportionally represented in both the training and test sets.

Example:

python



Kopyala



Düzenle

```
from sklearn import datasets from sklearn.model_selection import train_test_split # Load the dataset iris = datasets.load_iris() X, y = iris.data, iris.target # Features and Labels # Split data with stratification X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42) # Check class distribution import numpy as np print("Training set class distribution:", np.bincount(y_train)) print("Test set class distribution:", np.bincount(y_test))
```

Output:

kotlin



Kopyala



Düzenle

```
Training set class distribution: [40 40 40]
Test set class distribution: [10 10 10]
```

Since the original dataset has 50 samples per class, the stratified split keeps the ratio the same (80% in train, 20% in test).

Without `stratify=y`, the split would be purely random, possibly leading to an imbalanced distribution in small datasets.



how will i implement knn classifier to iris dataset



To implement a K-Nearest Neighbors (KNN) classifier on the Iris dataset, follow these steps:

Step 1: Import Required Libraries

python



Kopyala



Düzenle

```
import numpy as np import matplotlib.pyplot as plt from sklearn import datasets from
sklearn.model_selection import train_test_split from sklearn.preprocessing import
StandardScaler from sklearn.neighbors import KNeighborsClassifier from sklearn.metrics
import accuracy_score, classification_report, confusion_matrix
```

Step 2: Load the Iris Dataset

python



Kopyala



Düzenle

```
# Load dataset iris = datasets.load_iris() X, y = iris.data, iris.target # Features and
Labels
```

Step 3: Split the Dataset into Training and Testing Sets

python



Kopyala



Düzenle

```
# Split data into 80% training and 20% testing with stratification X_train, X_test,
y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

Step 4: Feature Scaling (Standardization)

KNN is a distance-based algorithm, so it's recommended to scale the features.

python



Kopyala



Düzenle

```
scaler = StandardScaler() X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)
```

Step 5: Train the KNN Classifier

python



Kopyala



Düzenle

```
# Initialize KNN with k=5 (default) knn = KNeighborsClassifier(n_neighbors=5) # Train the
model knn.fit(X_train, y_train)
```

Step 6: Make Predictions

python



Kopyala



Düzenle

```
y_pred = knn.predict(X_test)
```

Step 7: Evaluate the Model

python



Kopyala



Düzenle

```
# Accuracy
accuracy = accuracy_score(y_test, y_pred) print(f"Accuracy: {accuracy:.2f}") #
Classification Report print("\nClassification Report:") print(classification_report(y_test,
y_pred)) # Confusion Matrix print("\nConfusion Matrix:") print(confusion_matrix(y_test,
y_pred))
```

Finding the Best k (Hyperparameter Tuning)

To find the best k value, test multiple values and choose the one with the highest accuracy.

python



Kopyala



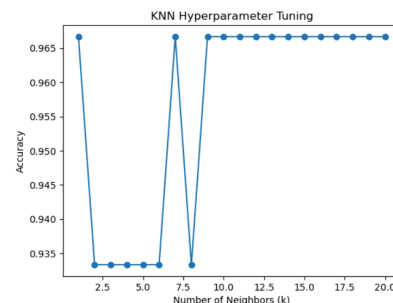
Düzenle

```
accuracies = [] k_values = range(1, 21) for k in k_values: knn =
KNeighborsClassifier(n_neighbors=k) knn.fit(X_train, y_train) y_pred = knn.predict(X_test)
accuracies.append(accuracy_score(y_test, y_pred)) # Plot accuracy vs. k plt.plot(k_values,
accuracies, marker='o') plt.xlabel('Number of Neighbors (k)') plt.ylabel('Accuracy')
plt.title('KNN Hyperparameter Tuning') plt.show()
```

Expected Output

- **Accuracy:** ~97-100% (since Iris is a simple dataset)
- **Confusion Matrix:** Should have most predictions on the diagonal
- **Best k :** Usually between 3-7

This is a complete implementation of **KNN on the Iris dataset**. Let me know if you have any questions!



According to your knn implementation on iris dataset; graph looked like this. How will I interpret? What are the results?



Your **KNN Hyperparameter Tuning** graph shows how the accuracy of the model changes with different values of k (number of neighbors). Here's how to interpret it:

Key Observations:

1. For $k = 1$:
 - The accuracy is very high (~96.7%).
 - This suggests that the model is closely fitting the training data.
 - However, $k = 1$ can lead to **overfitting** because the model simply memorizes the nearest neighbor.
2. For $k = 2$ to 6 :
 - The accuracy significantly drops to around 93.3%.

- This suggests that these values of k may not be optimal for generalizing well.
3. For k = 7:
- Accuracy suddenly increases back to **96.7%**.
 - This might indicate that k = 7 is a **better choice** to balance bias and variance.
4. For k = 9 onwards:
- The accuracy stabilizes at a high value (~96.7%).
 - This suggests that **higher k values (9-20) provide stable and reliable performance**.

Choosing the Best k:

- k = 1 achieves high accuracy but may overfit.
- k = 7 or higher (e.g., 9, 11, etc.) provides stable performance.
- If you want **less variance**, choosing a value like k = 9 or 11 would be a good option.

Conclusion:

- **Best k:** Values around 7-11 seem to be optimal.
- **Avoid k = 2 to 6**, as accuracy is lower.
- **Higher k values (9-20)** provide consistent accuracy, meaning the model is stable.

Would you like to test different distance metrics (e.g., Manhattan distance) to see if performance changes? 🚀