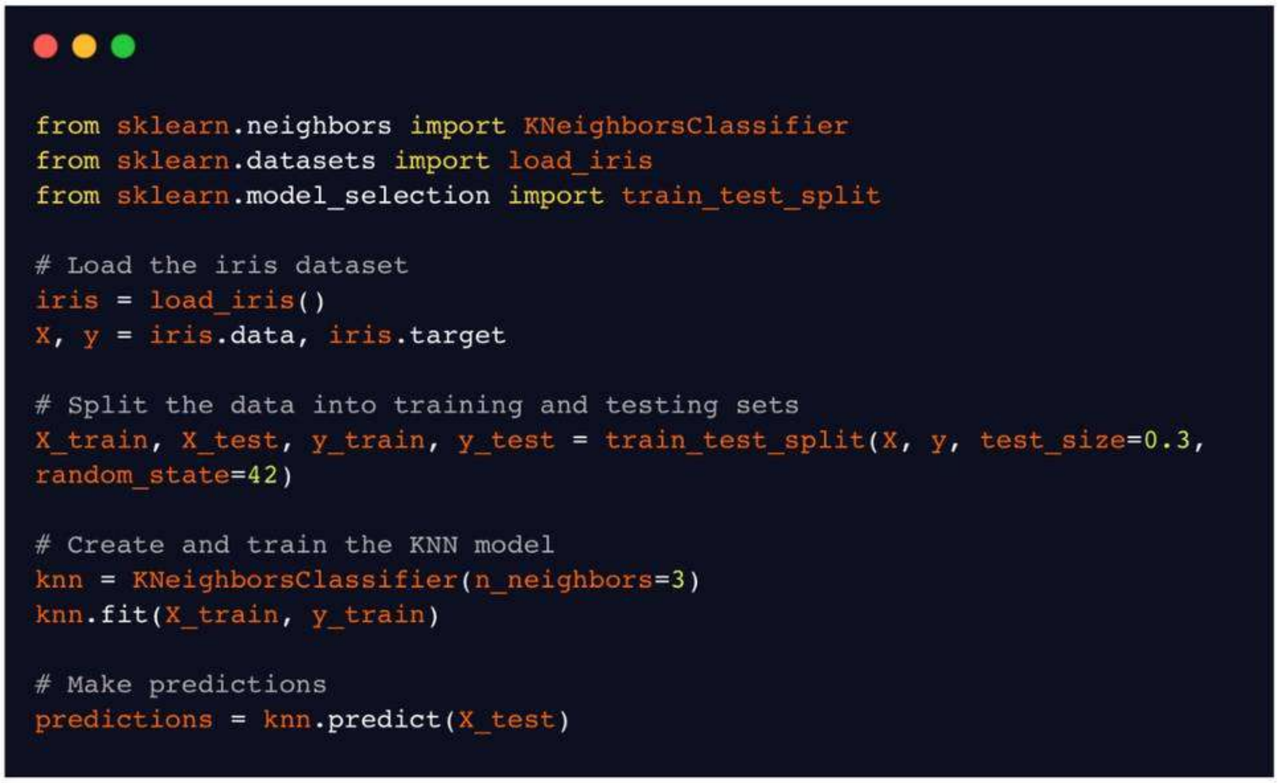
Mastering K-Nearest Neighbors Hyperparameters in Python

With Code Examples

save for later

Introduction to K-Nearest Neighbors (KNN)

K-Nearest Neighbors is a simple yet powerful machine learning algorithm used for classification and regression tasks. It works by finding the K closest data points to a new instance and making predictions based on their labels or values.

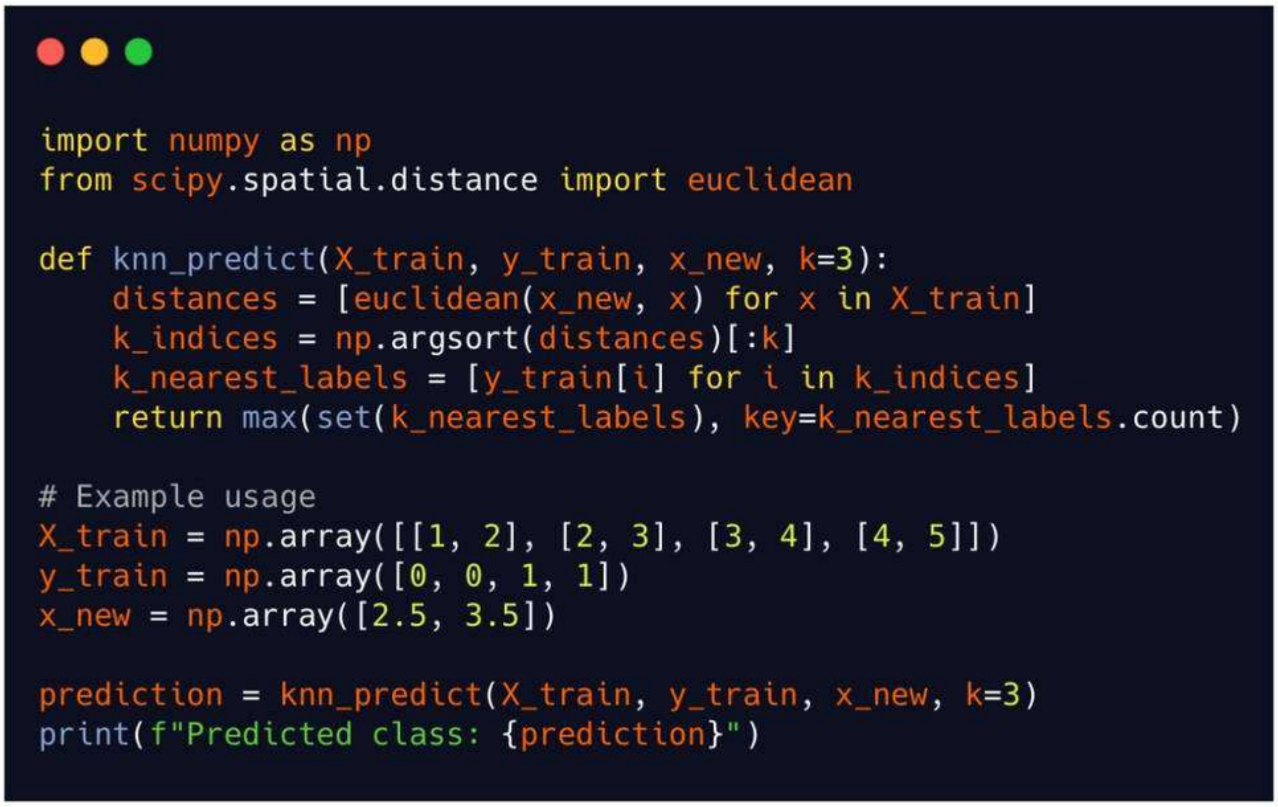


```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load the iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Create and train the KNN model
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Make predictions
predictions = knn.predict(X_test)
```

Swipe next ⟶

How KNN Works

KNN operates by calculating the distance between a new data point and all existing points in the dataset. It then selects the K nearest neighbors and uses their labels to make a prediction, either through majority voting (for classification) or averaging (for regression).



```python
import numpy as np
from scipy.spatial.distance import euclidean

def knn_predict(X_train, y_train, x_new, k=3):
    distances = [euclidean(x_new, x) for x in X_train]
    k_indices = np.argsort(distances)[:k]
    k_nearest_labels = [y_train[i] for i in k_indices]
    return max(set(k_nearest_labels), key=k_nearest_labels.count)

# Example usage
X_train = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y_train = np.array([0, 0, 1, 1])
x_new = np.array([2.5, 3.5])

prediction = knn_predict(X_train, y_train, x_new, k=3)
print(f"Predicted class: {prediction}")
```

Swipe next ⟶

save for later

Choosing the Right K Value

The value of K is a crucial hyperparameter in KNN. A small K can lead to overfitting, while a large K can result in underfitting. The optimal K value often depends on the specific dataset and problem at hand.

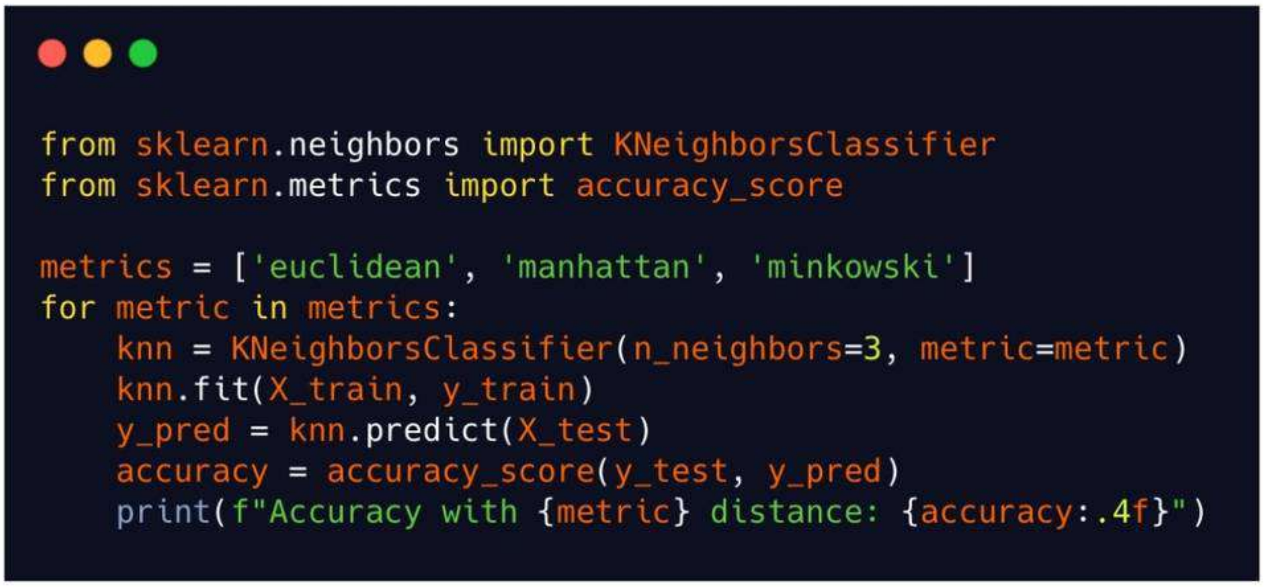
```
from sklearn.model_selection import cross_val_score

k_values = range(1, 31)
cv_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=5)
    cv_scores.append(scores.mean())

optimal_k = k_values[cv_scores.index(max(cv_scores))]
print(f"Optimal K value: {optimal_k}")
```

Swipe next ⟶

save for later

Distance Metrics in KNN

KNN relies on distance calculations to determine nearest neighbors. Common distance metrics include Euclidean, Manhattan, and Minkowski distances. The choice of metric can significantly impact the algorithm's performance.



```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

metrics = ['euclidean', 'manhattan', 'minkowski']
for metric in metrics:
    knn = KNeighborsClassifier(n_neighbors=3, metric=metric)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy with {metric} distance: {accuracy:.4f}")
```

Swipe next ⟶

follow for more

Weighted KNN

Weighted KNN assigns more importance to closer neighbors, potentially improving prediction accuracy. Weights can be uniform (equal importance) or distance-based (closer neighbors have higher weights).

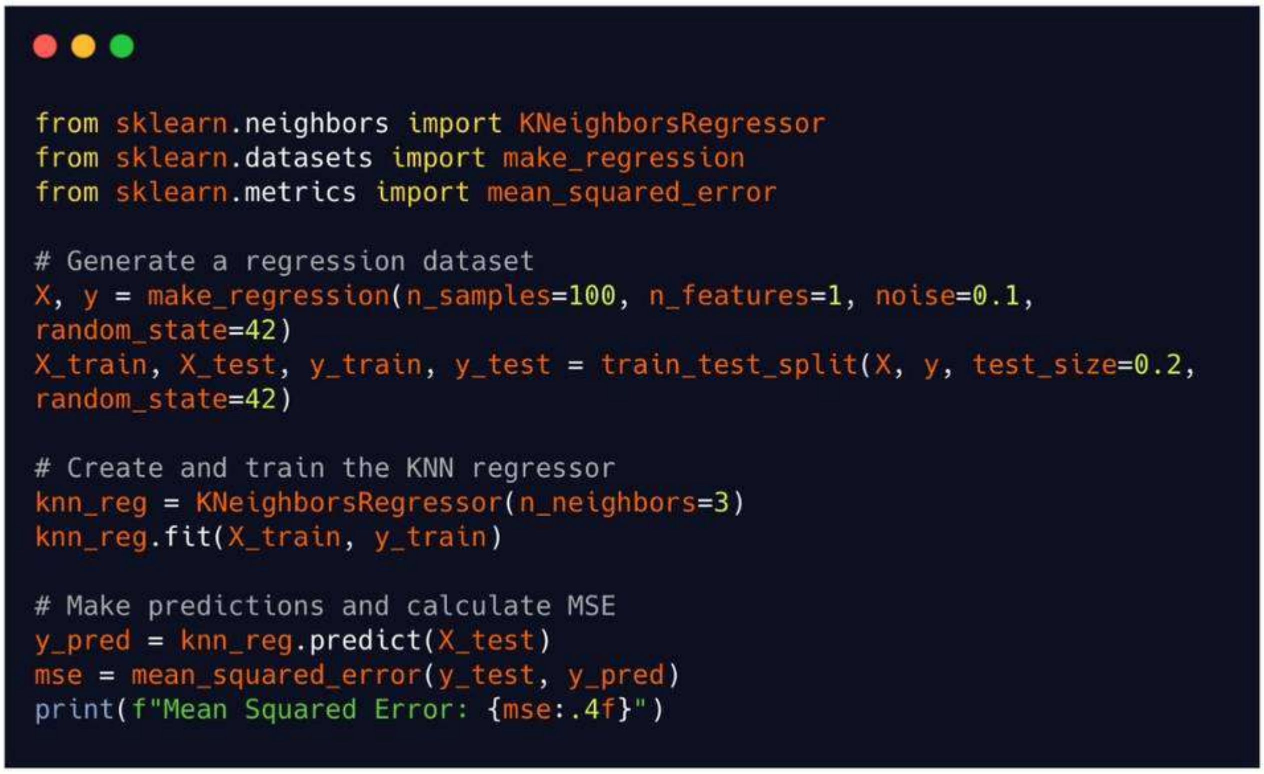
```python
from sklearn.neighbors import KNeighborsClassifier

# Uniform weights
knn_uniform = KNeighborsClassifier(n_neighbors=5, weights='uniform')
knn_uniform.fit(X_train, y_train)
accuracy_uniform = knn_uniform.score(X_test, y_test)

# Distance-based weights
knn_distance = KNeighborsClassifier(n_neighbors=5, weights='distance')
knn_distance.fit(X_train, y_train)
accuracy_distance = knn_distance.score(X_test, y_test)

print(f"Accuracy (uniform weights): {accuracy_uniform:.4f}")
print(f"Accuracy (distance weights): {accuracy_distance:.4f}")
```

Swipe next ⟶

save for later

KNN for Regression

KNN can be used for regression tasks by averaging the target values of the K nearest neighbors. This approach is useful for predicting continuous values based on similar instances in the dataset.



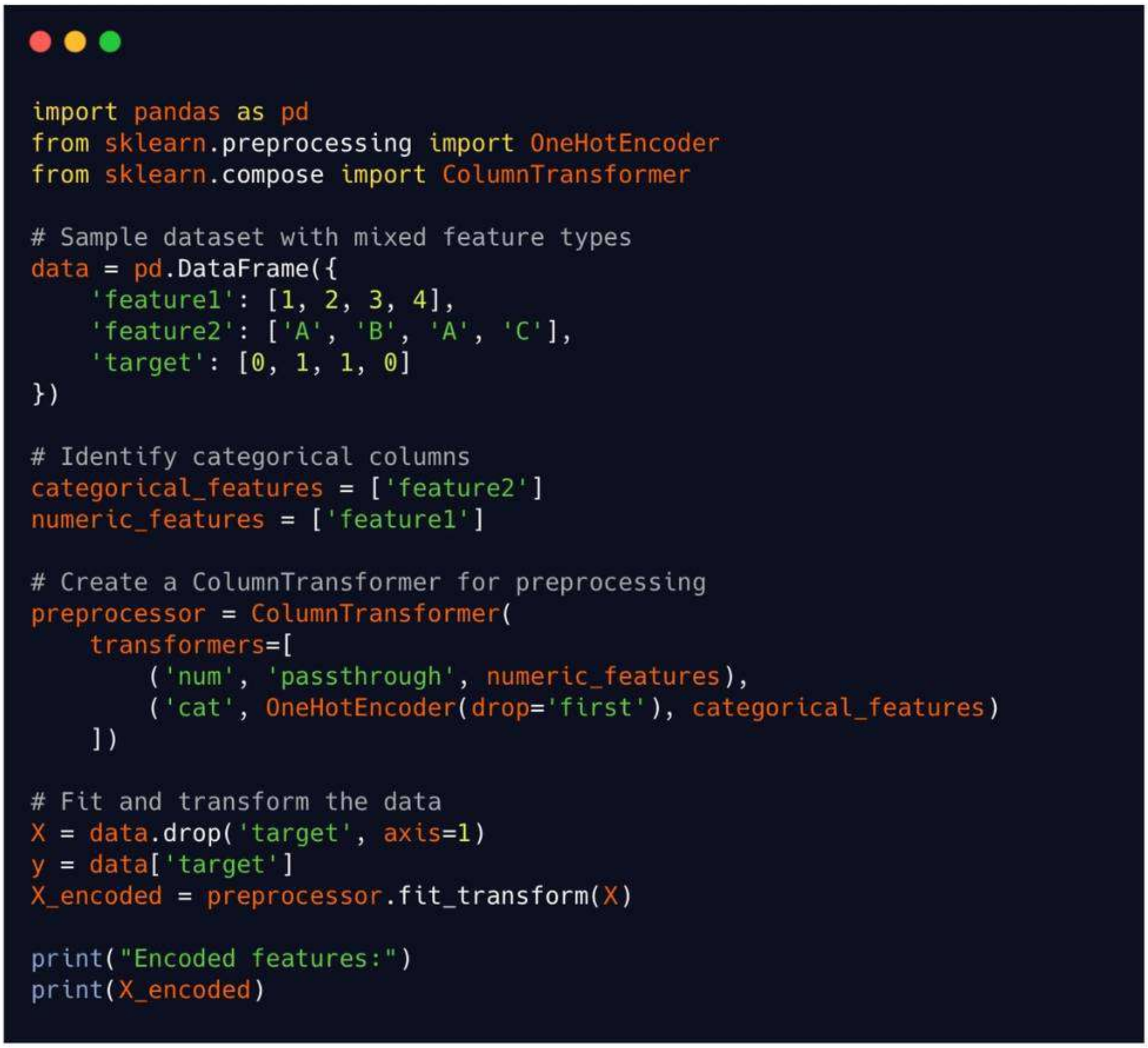
```python
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error

# Generate a regression dataset
X, y = make_regression(n_samples=100, n_features=1, noise=0.1,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create and train the KNN regressor
knn_reg = KNeighborsRegressor(n_neighbors=3)
knn_reg.fit(X_train, y_train)

# Make predictions and calculate MSE
y_pred = knn_reg.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.4f}")
```

Swipe next ⟶

save for later

Handling Categorical Features

KNN works best with numerical features. For categorical data, one-hot encoding or label encoding can be used to convert them into a suitable format for KNN.



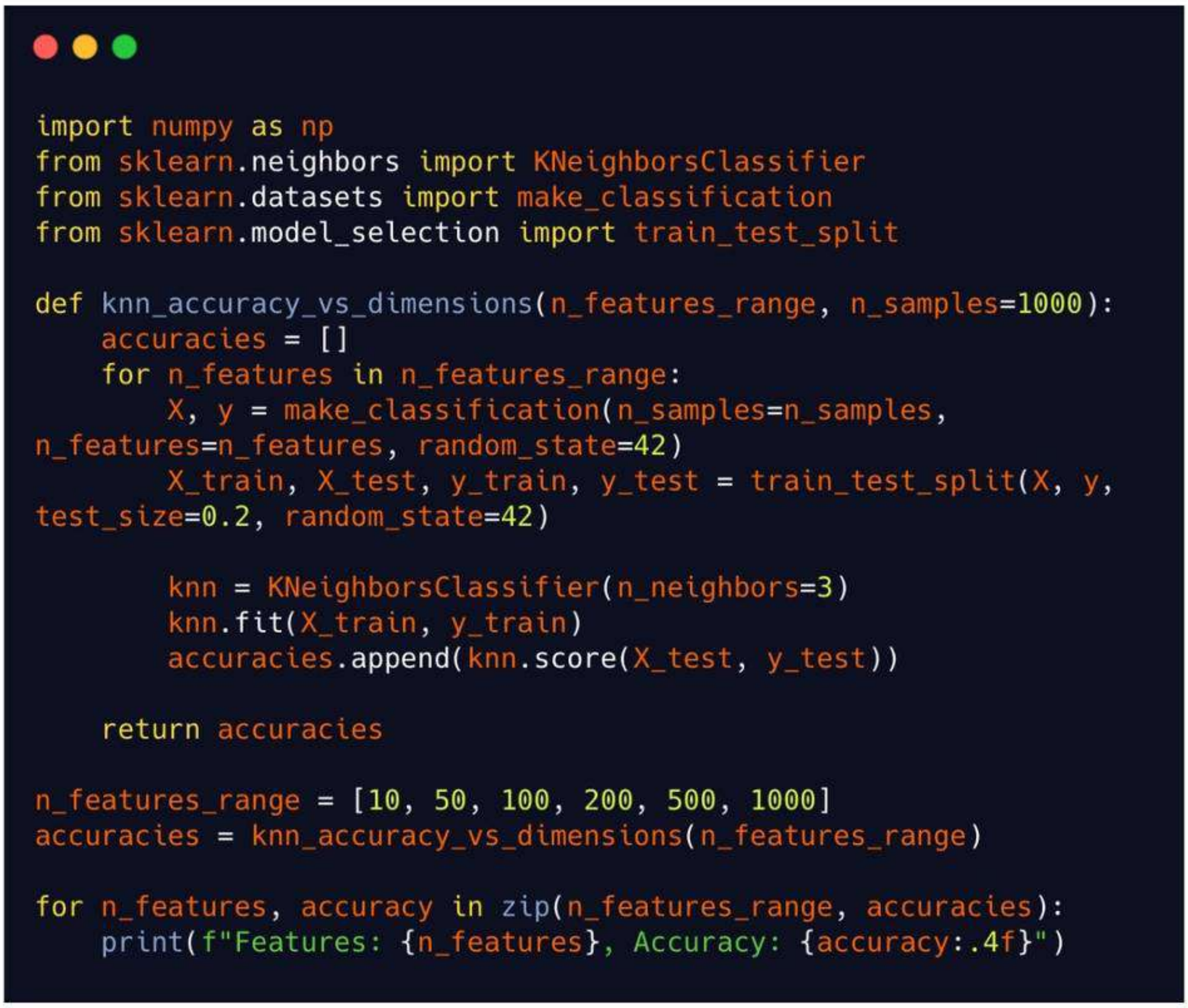
```python
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# Sample dataset with mixed feature types
data = pd.DataFrame({
    'feature1': [1, 2, 3, 4],
    'feature2': ['A', 'B', 'A', 'C'],
    'target': [0, 1, 1, 0]
})

# Identify categorical columns
categorical_features = ['feature2']
numeric_features = ['feature1']

# Create a ColumnTransformer for preprocessing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', 'passthrough', numeric_features),
        ('cat', OneHotEncoder(drop='first'), categorical_features)
    ])

# Fit and transform the data
X = data.drop('target', axis=1)
y = data['target']
X_encoded = preprocessor.fit_transform(X)

print("Encoded features:")
print(X_encoded)
```

Swipe next →

save for later

Curse of Dimensionality in KNN

As the number of features increases, KNN's performance can degrade due to the curse of dimensionality. This phenomenon occurs because the concept of "nearest" becomes less meaningful in high-dimensional spaces.

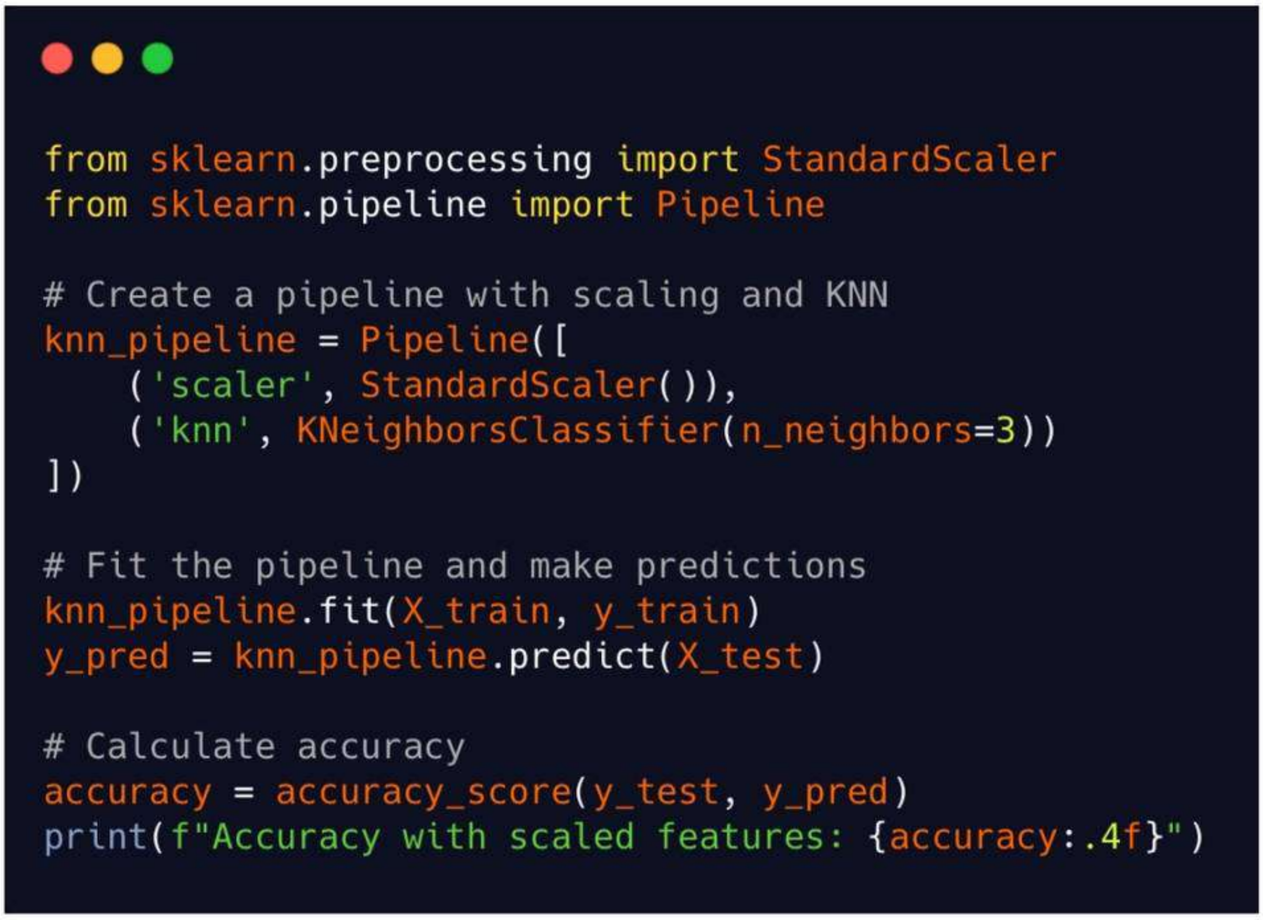


```python
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

def knn_accuracy_vs_dimensions(n_features_range, n_samples=1000):
    accuracies = []
    for n_features in n_features_range:
        X, y = make_classification(n_samples=n_samples,
n_features=n_features, random_state=42)
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

        knn = KNeighborsClassifier(n_neighbors=3)
        knn.fit(X_train, y_train)
        accuracies.append(knn.score(X_test, y_test))

    return accuracies

n_features_range = [10, 50, 100, 200, 500, 1000]
accuracies = knn_accuracy_vs_dimensions(n_features_range)

for n_features, accuracy in zip(n_features_range, accuracies):
    print(f"Features: {n_features}, Accuracy: {accuracy:.4f}")
```

Swipe next ⟶

Scaling Features for KNN

Feature scaling is crucial for KNN as it ensures all features contribute equally to distance calculations. Common scaling techniques include standardization and normalization.



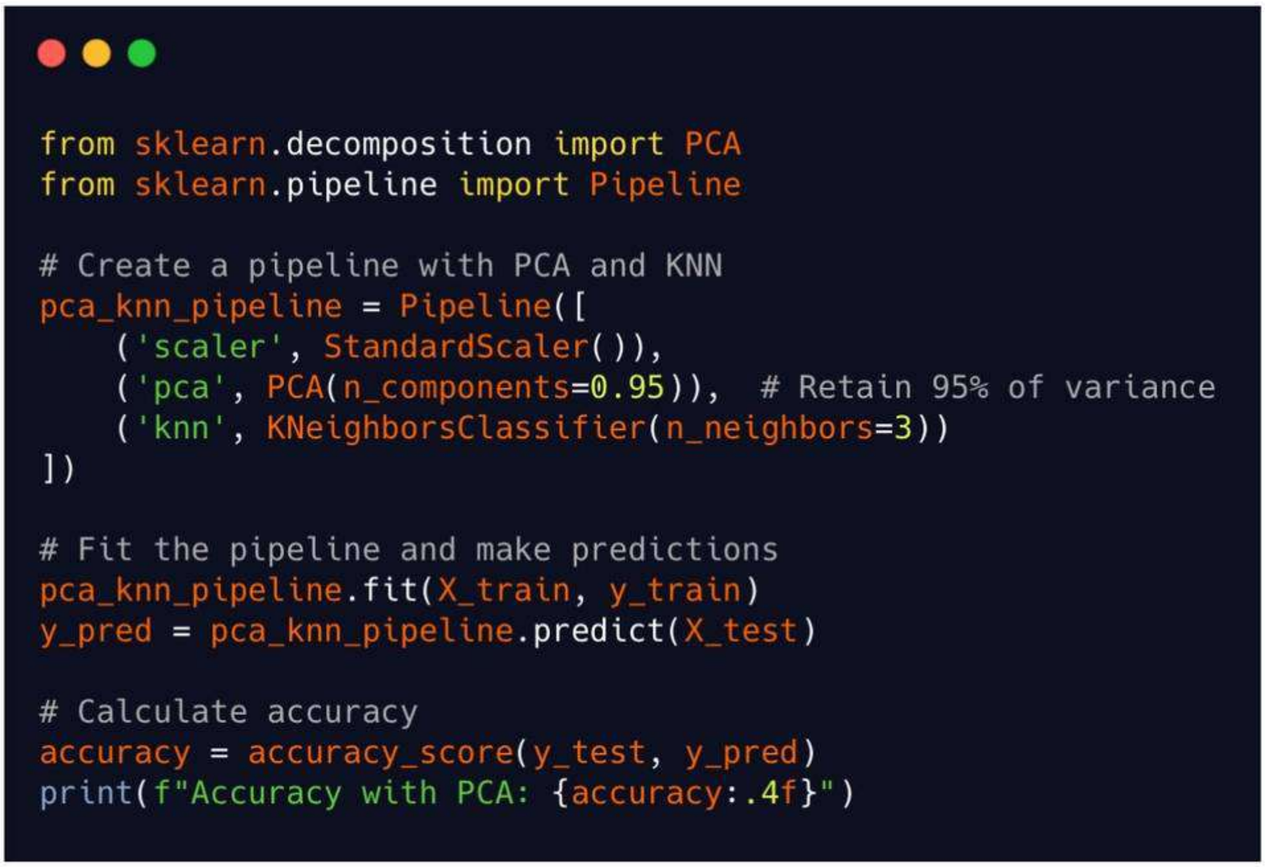
```python
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Create a pipeline with scaling and KNN
knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier(n_neighbors=3))
])

# Fit the pipeline and make predictions
knn_pipeline.fit(X_train, y_train)
y_pred = knn_pipeline.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy with scaled features: {accuracy:.4f}")
```

Swipe next ⟶

follow for more

KNN with PCA for Dimensionality Reduction

To mitigate the curse of dimensionality, we can use Principal Component Analysis (PCA) to reduce the number of features while retaining most of the variance in the data.



```python
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline

# Create a pipeline with PCA and KNN
pca_knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.95)),  # Retain 95% of variance
    ('knn', KNeighborsClassifier(n_neighbors=3))
])

# Fit the pipeline and make predictions
pca_knn_pipeline.fit(X_train, y_train)
y_pred = pca_knn_pipeline.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy with PCA: {accuracy:.4f}")
```

Swipe next ⟶

save for later

Cross-Validation for KNN Hyperparameter Tuning

Cross-validation helps in finding optimal hyperparameters for KNN, such as the number of neighbors and weight function, by evaluating the model's performance on different subsets of the data.

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

# Create the GridSearchCV object
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)

# Fit the grid search
grid_search.fit(X_train, y_train)

# Print the best parameters and score
print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score:", grid_search.best_score_)
```

Swipe next ⟶

follow for more

KNN for Anomaly Detection

KNN can be used for anomaly detection by identifying data points that are far from their nearest neighbors. This approach is particularly useful for detecting outliers in datasets.

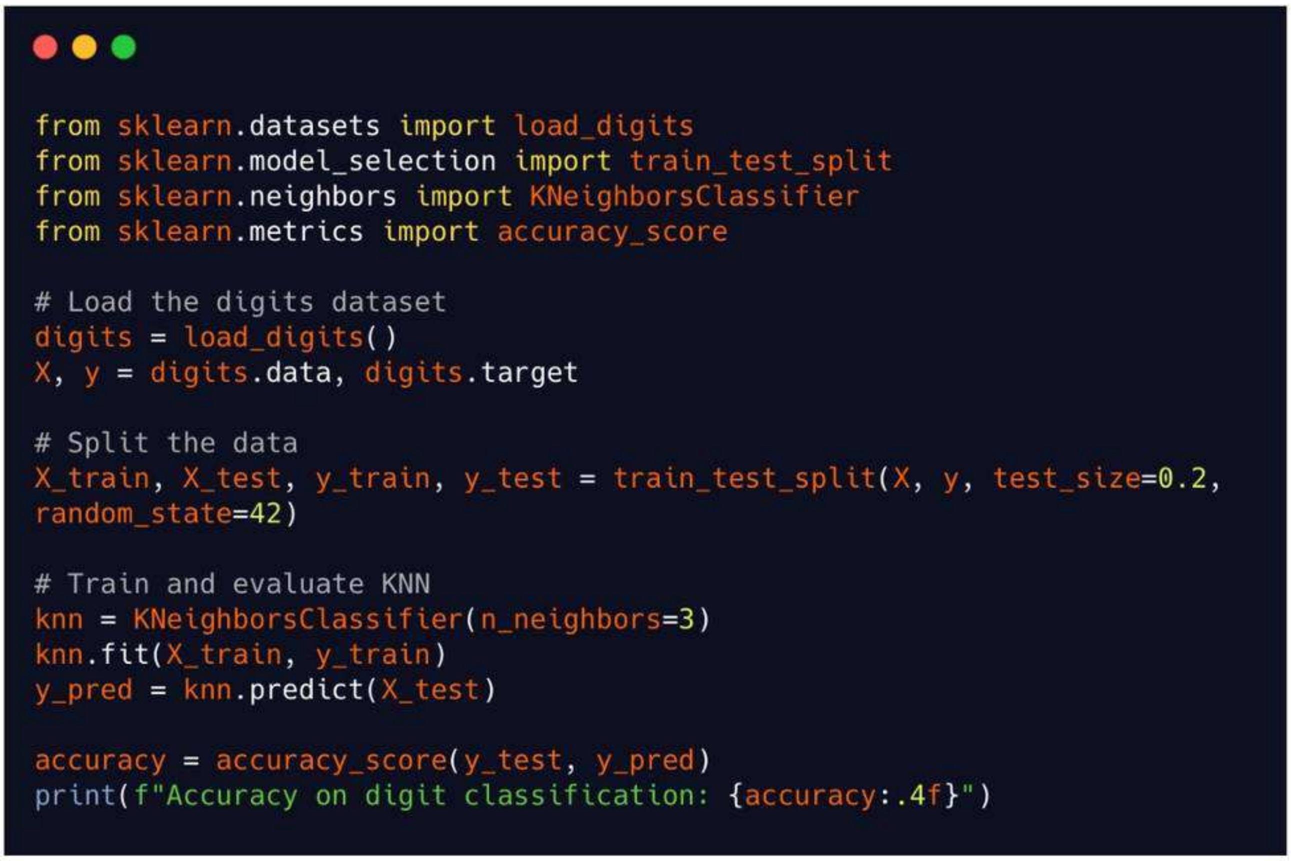
```python
import numpy as np
from sklearn.neighbors import NearestNeighbors

def knn_anomaly_detection(X, k=5, threshold=1.5):
    # Fit the nearest neighbors model
    nn = NearestNeighbors(n_neighbors=k)
    nn.fit(X)

    # Calculate the average distance to k-nearest neighbors
    distances, _ = nn.kneighbors(X)
    avg_distances = np.mean(distances, axis=1)

    # Identify anomalies
    threshold_value = np.mean(avg_distances) + threshold *
np.std(avg_distances)
    anomalies = X[avg_distances > threshold_value]

    return anomalies

# Generate sample data with outliers
X = np.random.randn(100, 2)
X = np.vstack([X, [[10, 10], [-10, -10]]])  # Add outliers

anomalies = knn_anomaly_detection(X)
print(f"Number of anomalies detected: {len(anomalies)}")
```

Swipe next ⟶

save for later

KNN for Image Classification

KNN can be applied to image classification tasks by treating each pixel as a feature. However, this approach may require dimensionality reduction techniques for larger images.



```python
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the digits dataset
digits = load_digits()
X, y = digits.data, digits.target

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train and evaluate KNN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy on digit classification: {accuracy:.4f}")
```

Swipe next ⟶

Additional Resources

- "A Comprehensive Study of k-Nearest Neighbor Machine Learning Algorithm" - ArXiv.org Reference: arXiv:2004.04523
- "Nearest Neighbor Methods in Learning and Vision: Theory and Practice" - MIT Press (Not an ArXiv source, but a comprehensive book on KNN)
- "An Improved KNN Algorithm for Imbalanced Data Classification" - ArXiv.org Reference: arXiv:1811.00839

Data scientist
&ML Engineer

Follow For More Data
Science Content