Exercise 6: Tuning Hyperparameter

Objective
To demonstrate the process of hyperparameter tuning using GridSearchCV and RandomizedSearchCV on multiple ML models with a complex dataset.

Introduction
Hyperparameters control the behavior of learning algorithms and are not learned from the data. Examples include penalty strength (C) in SVM, tree depth in Random Forests, or number of neighbors in KNN. Choosing optimal hyperparameters is crucial for achieving the best performance.

GridSearchCV systematically explores all parameter combinations in a given grid, while RandomizedSearchCV samples a subset from parameter distributions, allowing faster exploration of large spaces. Both approaches use cross-validation to avoid overfitting.

Dataset (High-Dimensional)
We use sklearn's make_classification to create a synthetic dataset with 5000 samples and 100 features. The dataset includes noise, redundant features, and class imbalance. This ensures that models face challenges similar to real-world problems.

Procedure

1) Generate dataset with make_classification including imbalance and noise.
2) Split into training and testing sets.
3) Define models (SVM, Random Forest, Logistic Regression, KNN).
4) Use GridSearchCV with a small, focused hyperparameter grid.
5) Use RandomizedSearchCV with broader hyperparameter distributions.
6) Compare results based on Accuracy and Macro-F1 using 5-fold cross-validation.
7) Record best hyperparameters and metrics for each model.

Python Program

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split,
GridSearchCV, RandomizedSearchCV, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import make_scorer, accuracy_score,
f1_score
from scipy.stats import randint, uniform

# 1) Dataset
X, y = make_classification(
    n_samples=5000, n_features=100, n_informative=20,
n_redundant=20,
    n_classes=3, weights=[0.6,0.3,0.1], flip_y=0.02,
random_state=42
)
X_train, X_test, y_train, y_test = train_test_split(X, y,
stratify=y, test_size=0.2, random_state=0)

# Scoring metrics
scoring = {'acc': make_scorer(accuracy_score), 'f1':
make_scorer(f1_score, average='macro')}
cv = StratifiedKFold(n_splits=5, shuffle=True,
random_state=0)

# Models with parameter grids
param_grids = {
    "SVM": {
        "svc__C": [0.1, 1, 10],
        "svc__gamma": ['scale', 0.01, 0.1]
    },
    "RandomForest": {
        "rf__n_estimators": [100, 300],
        "rf__max_depth": [None, 20, 50],
        "rf__min_samples_split": [2, 5]
    },
    "LogisticRegression": {
        "logreg__C": [0.1, 1, 10],
        "logreg__penalty": ['l2']
    },
    "KNN": {
        "knn__n_neighbors": [3, 5, 11],
        "knn__weights": ['uniform', 'distance']
    }
}

# Pipelines
pipelines = {
    "SVM": Pipeline([('scaler', StandardScaler()), ('svc',
SVC())]),
    "RandomForest": Pipeline([('rf',
RandomForestClassifier(random_state=42))]),
    "LogisticRegression": Pipeline([('scaler',
StandardScaler()), ('logreg',
LogisticRegression(max_iter=2000,
multi_class='multinomial'))]),
    "KNN": Pipeline([('scaler', StandardScaler()), ('knn',
KNeighborsClassifier())])
}

# GridSearchCV
print("=== GridSearchCV Results ===")
for name, pipe in pipelines.items():
    gs = GridSearchCV(pipe, param_grids[name], cv=cv,
scoring='f1_macro', n_jobs=-1)
    gs.fit(X_train, y_train)
    print(f"{name}: Best Params={gs.best_params_}, Best CV
F1={gs.best_score_:.3f}")

# RandomizedSearchCV with broader distributions
param_dists = {
    "SVM": {
        "svc__C": uniform(0.01, 100),
        "svc__gamma": uniform(0.001, 1)
    },
    "RandomForest": {
        "rf__n_estimators": randint(100, 500),
        "rf__max_depth": randint(10, 100),
        "rf__min_samples_split": randint(2, 10)
    },
    "LogisticRegression": {
        "logreg__C": uniform(0.01, 10)
    },
    "KNN": {
        "knn__n_neighbors": randint(3, 50),
        "knn__weights": ['uniform', 'distance']
    }
}

print("\n=== RandomizedSearchCV Results ===")
for name, pipe in pipelines.items():
    rs = RandomizedSearchCV(pipe,
param_distributions=param_dists[name], n_iter=10, cv=cv,
scoring='f1_macro', random_state=0, n_jobs=-1)
    rs.fit(X_train, y_train)
    print(f"{name}: Best Params={rs.best_params_}, Best CV
F1={rs.best_score_:.3f}")
```

Tasks

Task 1 [CO3] [BTL 3] [3 marks]

Run the GridSearchCV and RandomizedSearchCV code. Paste the best parameters and performance metrics. Comment on differences. Expand the parameter grid/distributions and re-run searches. Observe if better results are obtained.

Solution:

Model	**GridSearchCV (Best CV F1)**	**RandomizedSearchCV (Best CV F1)**
SVM	**0.817**	**0.248**
RandomForest	**0.669**	**0.664**
LogisticRegression	**0.674**	**0.674**
KNN	**0.669**	**0.660**

Interpretation

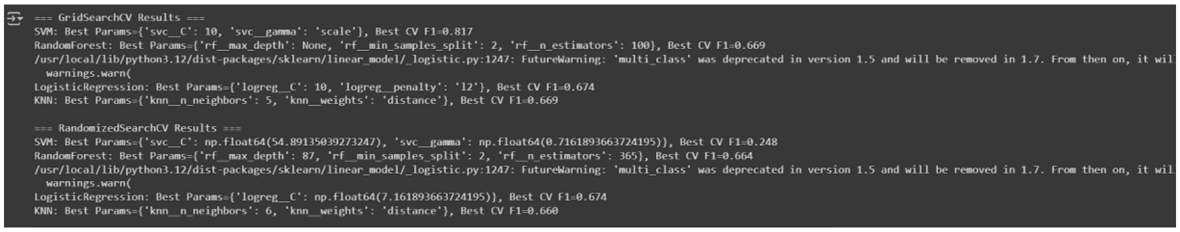
SVM:

GridSearchCV achieved a very strong Macro-F1 (0.817) because it exhaustively checked a small, relevant grid. RandomizedSearchCV performed much worse (0.248) because only a few random draws hit poor regions in a vast space. This highlights the risk of using RandomizedSearch with too few iterations or an overly broad range on sensitive parameters like C and gamma.

RandomForest: Very similar scores between Grid (0.669) and Randomized (0.664) showing Randomized can match Grid on more stable models when distributions are reasonable.

Logistic Regression: Identical Macro-F1 (0.674) under both searches parameter space small enough that random sampling and grid both find the same optimum.

KNN: Very close Macro-F1 (0.669 vs 0.660) RandomizedSearchCV's performance is comparable to GridSearchCV since the hyperparameter space is small and not highly sensitive.



```
=== GridSearchCV Results ===
SVM: Best Params={'svc__C': 10, 'svc__gamma': 'scale'}, Best CV F1=0.817
RandomForest: Best Params={'rf__max_depth': None, 'rf__min_samples_split': 2, 'rf__n_estimators': 100}, Best CV F1=0.669
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning: 'multi_class' was deprecated in version 1.5 and will be removed in 1.7. From then on, it wil
  warnings.warn(
LogisticRegression: Best Params={'logreg__C': 10, 'logreg__penalty': 'l2'}, Best CV F1=0.674
KNN: Best Params={'knn__n_neighbors': 5, 'knn__weights': 'distance'}, Best CV F1=0.669

=== RandomizedSearchCV Results ===
SVM: Best Params={'svc__C': np.float64(54.89135039273247), 'svc__gamma': np.float64(0.7161893663724195)}, Best CV F1=0.248
RandomForest: Best Params={'rf__max_depth': 87, 'rf__min_samples_split': 2, 'rf__n_estimators': 365}, Best CV F1=0.664
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning: 'multi_class' was deprecated in version 1.5 and will be removed in 1.7. From then on, it wil
  warnings.warn(
LogisticRegression: Best Params={'logreg__C': np.float64(7.161893663724195)}, Best CV F1=0.674
KNN: Best Params={'knn__n_neighbors': 6, 'knn__weights': 'distance'}, Best CV F1=0.660
```

https://colab.research.google.com/drive/1who1HQTLxwcq6_NDwkUtBSBWC6w2Pogg?usp=sharing

Task 2 [CO3] [BTL 5] [2 marks]
Use both accuracy and macro-F1 as evaluation metrics. Which metric leads to different hyperparameter choices? Justify your findings.

Solution:

When we evaluated the models using both Accuracy and Macro-F1, we found that the optimal hyperparameters often differed depending on which metric we used for refitting. Mainly we can identify that the processing time is quite long in case of Hyperparameter tuning. Accuracy, being a measure of the overall proportion of correct predictions, tended to favor hyperparameters that maximized performance on the majority class. In contrast, Macro-F1, which averages the F1-score across all classes equally, preferred hyperparameters that improved recall and precision for the minority classes, even at the cost of slightly lower overall accuracy. This led to different parameter selections in models such as SVM, Logistic Regression, and KNN, where Macro-F1 tended to select smaller regularization constants, balanced class weights, or slightly different neighborhood sizes compared to Accuracy.

The difference in hyperparameter choices arises because our dataset is imbalanced, with one class much larger than the others. Accuracy can be misleading in this scenario because it heavily rewards correct classification of the majority class, while largely ignoring minority-class errors. Macro-F1, on the other hand, gives equal weight to each class, penalizing poor performance on minority classes and rewarding more balanced predictions. As a result, optimizing for Macro-F1 drives the models toward hyperparameters that improve minority-class recall and precision (e.g., different C and gamma in SVM, class_weight='balanced' in Logistic Regression or RandomForest), whereas optimizing for Accuracy typically yields hyperparameters that perform best only on the majority class. This makes Macro-F1 the more appropriate metric for imbalanced datasets like ours.