```
!pip install xgboost
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.10/dist-packages (2.0.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from xgboost) (1.25.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from xgboost) (1.11.4)
```

```
from sklearn import svm, datasets
iris = datasets.load_iris()
```

```
import pandas as pd
df = pd.DataFrame(iris.data,columns=iris.feature_names)
df['flower'] = iris.target
df['flower'] = df['flower'].apply(lambda x: iris.target_names[x])
df[47:150]
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	flower
47	4.6	3.2	1.4	0.2	setosa
48	5.3	3.7	1.5	0.2	setosa
49	5.0	3.3	1.4	0.2	setosa
50	7.0	3.2	4.7	1.4	versicolor
51	6.4	3.2	4.5	1.5	versicolor
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

103 rows × 5 columns

Approach 1: Use train_test_split and manually tune parameters by trial and error

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3)
```

```
model = svm.SVC(kernel='rbf',C=30,gamma='auto')
model.fit(X_train,y_train)
model.score(X_test, y_test)
```

```
0.9333333333333333
```

Approach 2: Use K Fold Cross validation

Manually try suppling models with different parameters to cross_val_score function with 5 fold cross validation

```
from sklearn.model_selection import cross_val_score
```

```
cross_val_score(svm.SVC(kernel='linear',C=10,gamma='auto'),iris.data, iris.target, cv=5)
```

```
array([1.        , 1.        , 0.9       , 0.96666667, 1.        ])
```

```
cross_val_score(svm.SVC(kernel='rbf',C=10,gamma='auto'),iris.data, iris.target, cv=5)
```

```
array([0.96666667, 1.        , 0.96666667, 0.96666667, 1.        ])
```

```
cross_val_score(svm.SVC(kernel='rbf',C=20,gamma='auto'),iris.data, iris.target, cv=5)
```

```
array([0.96666667, 1.        , 0.9       , 0.96666667, 1.        ])
```

Above approach is manual. We can use for loop as an alternative

```
import numpy as np
kernels = ['rbf', 'linear']
C = [1,10,20]
avg_scores = {}
for kval in kernels:
    for cval in C:
        cv_scores = cross_val_score(svm.SVC(kernel=kval,C=cval,gamma='auto'),iris.data, iris.target, cv=5)
        avg_scores[kval + '_' + str(cval)] = np.average(cv_scores)

avg_scores
```

```
{'rbf_1': 0.9800000000000001,
 'rbf_10': 0.9800000000000001,
 'rbf_20': 0.9666666666666668,
 'linear_1': 0.9800000000000001,
 'linear_10': 0.9733333333333334,
 'linear_20': 0.9666666666666666}
```

From above results we can say that rbf with C=1 or 10 or linear with C=1 will give best performance

Approach 3: Use GridSearchCV

GridSearchCV does exactly same thing as for loop above but in a single line of code

```
from sklearn.model_selection import GridSearchCV
clf = GridSearchCV(svm.SVC(gamma='auto'), {
    'C': [1,10,20],
    'kernel': ['rbf','linear']
}, cv=5, return_train_score=False)
clf.fit(iris.data, iris.target)
clf.cv_results_
```

```
{'mean_fit_time': array([0.00100484, 0.0006494 , 0.00073299, 0.00069718, 0.00074325,
        0.00073876]),
 'std_fit_time': array([2.58868811e-04, 2.39167943e-05, 3.50948909e-05, 5.25749242e-05,
        2.78207411e-05, 1.16441603e-04]),
 'mean_score_time': array([0.0006146 , 0.00044088, 0.00049305, 0.00048633, 0.00046725,
        0.0005022 ]),
 'std_score_time': array([1.49077203e-04, 9.42812095e-06, 3.47885306e-05, 3.83248948e-05,
        5.38087017e-06, 8.28924313e-05]),
 'param_C': masked_array(data=[1, 1, 10, 10, 20, 20],
              mask=[False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_kernel': masked_array(data=['rbf', 'linear', 'rbf', 'linear', 'rbf', 'linear'],
              mask=[False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'C': 1, 'kernel': 'rbf'},
  {'C': 1, 'kernel': 'linear'},
  {'C': 10, 'kernel': 'rbf'},
  {'C': 10, 'kernel': 'linear'},
  {'C': 20, 'kernel': 'rbf'},
  {'C': 20, 'kernel': 'linear'}],
 'split0_test_score': array([0.96666667, 0.96666667, 0.96666667, 1.        , 0.96666667,
        1.        ]),
 'split1_test_score': array([1., 1., 1., 1., 1., 1.]),
 'split2_test_score': array([0.96666667, 0.96666667, 0.96666667, 0.9       , 0.9       ,
        0.9       ]),
 'split3_test_score': array([0.96666667, 0.96666667, 0.96666667, 0.96666667, 0.96666667,
        0.93333333]),
 'split4_test_score': array([1., 1., 1., 1., 1., 1.]),
 'mean_test_score': array([0.98      , 0.98      , 0.98      , 0.97333333, 0.96666667,
        0.96666667]),
 'std_test_score': array([0.01632993, 0.01632993, 0.01632993, 0.03887301, 0.03651484,
        0.0421637 ]),
 'rank_test_score': array([1, 1, 1, 4, 5, 6], dtype=int32)}
```

```
df = pd.DataFrame(clf.cv_results_)
df
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_kernel
0	0.001005	0.000259	0.000615	0.000149	1	rbf
1	0.000649	0.000024	0.000441	0.000009	1	linear
2	0.000733	0.000035	0.000493	0.000035	10	rbf
3	0.000697	0.000053	0.000486	0.000038	10	linear
4	0.000743	0.000028	0.000467	0.000005	20	rbf
5	0.000739	0.000116	0.000502	0.000083	20	linear

```
df[['param_C','param_kernel','mean_test_score']]
```

	param_C	param_kernel	mean_test_score
0	1	rbf	0.980000
1	1	linear	0.980000
2	10	rbf	0.980000
3	10	linear	0.973333
4	20	rbf	0.966667
5	20	linear	0.966667

```
clf.best_params_
```

```
{'C': 1, 'kernel': 'rbf'}
```

```
clf.best_score_
```

```
0.9800000000000001
```

```
dir(clf)
```

```
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setstate__',
```

```
 '_validate_params',
 'best_estimator_',
 'best_index_',
 'best_params_',
 'best_score_',
 'classes_',
 'cv',
 'cv_results_',
 'decision_function',
 'error_score',
 'estimator',
 'fit',
 'get_params',
 'inverse_transform',
 'multimetric_',
 'n_features_in_',
 'n_jobs',
 'n_splits_',
 'param_grid',
 'pre_dispatch',
 'predict',
 'predict_log_proba',
 'predict_proba',
 'refit',
 'refit_time_',
 'return_train_score',
 'score',
 'score_samples',
 'scorer_',
 'scoring',
 'set_params',
 'transform',
 'verbose']
```

Use RandomizedSearchCV to reduce number of iterations and with random combination of parameters. This is useful when you have too many parameters to try and your training time is longer.

```
from sklearn.model_selection import RandomizedSearchCV
rs = RandomizedSearchCV(svm.SVC(gamma='auto'), {
        'C': [1,10,20],
        'kernel': ['rbf','linear']
    },
    cv=5,
    return_train_score=False,
    n_iter=2
)
rs.fit(iris.data, iris.target)
pd.DataFrame(rs.cv_results_)[['param_C','param_kernel','mean_test_score']]
```

	param_C	param_kernel	mean_test_score
0	1	linear	0.980000
1	10	linear	0.973333

Different models with different hyperparameters

```
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression

model_params = {
    'svm': {
        'model': svm.SVC(gamma='auto'),
        'params' : {
            'C': [1,10,20],
            'kernel': ['rbf','linear']
        }
    },
    'random_forest': {
        'model': RandomForestClassifier(),
        'params' : {
            'n_estimators': [1,5,10]
        }
    },
    'logistic_regression' : {
        'model': LogisticRegression(solver='liblinear',multi_class='auto'),
        'params': {
            'C': [1,5,10]
        }
    }
}


scores = []

for model_name, mp in model_params.items():
    clf =  GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False)
    clf.fit(iris.data, iris.target)
    scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_
    })

df = pd.DataFrame(scores,columns=['model','best_score','best_params'])
df
```

	model	best_score	best_params
0	svm	0.980000	{'C': 1, 'kernel': 'rbf'}
1	random_forest	0.960000	{'n_estimators': 5}
2	logistic_regression	0.966667	{'C': 5}

Next steps: Generate code with df View recommended plots

Based on above, I can conclude that SVM with C=1 and kernel='rbf' is the best model for solving my problem of iris flower classification

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, roc_curve
import matplotlib.pyplot as plt


X_train, X_test, y_train, y_test = train_test_split(X_test, y_test, test_size=0.2, random_state=42)


# Train a logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression()
```

```
# Make predictions on the test set
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]


from sklearn.metrics import precision_score, recall_score, f1_score

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

```
Accuracy: 0.7777777777777778
Precision: 0.8888888888888888
Recall: 0.7777777777777778
F1-score: 0.7925925925925926
```

Based on above, I can conclude that precision is the best evaluation metrics to assess model performance.

Start coding or generate with AI