下面我给你一个具体的、可执行的 Python 代码框架和详细注释，以实现**GPSA（Golden Path Search Algorithm）**在你所描述的数据场景中的应用：

🌟 数据准备要求：

假设你的 DataFrame 结构如下：

• Group: 组别

• Stage: 阶段名

• Equipment: 设备名

• Yield: 良率 (0~100 之间浮点数)

每个样本代表一个生产实例，列出每个阶段具体使用的设备以及对应的最终良率。

🚩 完整实现步骤与 Python 代码：

Step 1: 数据预处理

```
import pandas as pd
import numpy as np

# 假设 df 为原始数据
# df 示例结构:
# | Sample_ID | Group | Stage | Equipment | Yield |
# |-----------|--------|--------|-----------|-------|

# Pivot 数据，形成 (Sample_ID 为索引，每个阶段设备为列的形式)
df_pivot = df.pivot_table(index='Sample_ID', columns='Stage',
values='Equipment', aggfunc='first')
yield_df = df[['Sample_ID',
'Yield']].drop_duplicates().set_index('Sample_ID')

# 合并成一张完整数据表
data = df_pivot.join(yield_df)
data.dropna(inplace=True)
```

Step 2: 阶段筛选 (Group OGA, GOGA)

```python
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import r2_score

# 编码设备名为数字（One-hot encoding）
encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
X_encoded = encoder.fit_transform(data.iloc[:, :-1])
y = data['Yield'].values

# GOGA 阶段筛选实现 (Group Orthogonal Greedy Algorithm)
selected_stages = []
remaining_stages = list(data.columns[:-1])

threshold = 0.01  # 可根据需要调整的阈值
current_score = 0

while remaining_stages:
    scores = []
    for stage in remaining_stages:
        trial_stages = selected_stages + [stage]
        X_trial = encoder.fit_transform(data[trial_stages])
        model = LinearRegression().fit(X_trial, y)
        y_pred = model.predict(X_trial)
        score = r2_score(y, y_pred)
        scores.append((stage, score))

    # 选取提升最明显的阶段
    scores.sort(key=lambda x: x[1], reverse=True)
    best_stage, best_score = scores[0]

    if best_score - current_score > threshold:
        selected_stages.append(best_stage)
        remaining_stages.remove(best_stage)
        current_score = best_score
    else:
        break

print("关键阶段:", selected_stages)
```

Step 3: 生成关键阶段所有路径组合

```python
from itertools import product

# 获取每个关键阶段的唯一设备列表
equipments_per_stage = {stage: data[stage].unique() for stage in
selected_stages}
```

```
# 所有可能路径的组合 (Cartesian Product)
all_paths = list(product(*equipments_per_stage.values()))
print(f"共生成 {len(all_paths)} 条路径组合")
```

Step 4: 为每条路径构建良率预测模型

```
def predict_yield(path, data, stages, encoder):
    X_binary = pd.DataFrame(0, index=data.index, columns=stages)
    for stage, equip in zip(stages, path):
        X_binary[stage] = (data[stage] == equip).astype(int)

    X_encoded = encoder.fit_transform(X_binary)
    model = LinearRegression().fit(X_encoded, data['Yield'])
    predicted_yield = model.predict(X_encoded).mean()  # 平均预测值代表路径良率
    return predicted_yield

# 为所有路径预测良率
path_yields = []
for path in all_paths:
    yield_pred = predict_yield(path, data, selected_stages, encoder)
    path_yields.append((path, yield_pred))

# 根据良率排序
sorted_paths = sorted(path_yields, key=lambda x: x[1], reverse=True)
```

Step 5: 推荐黄金路径（前 10）

```
top_10_paths = sorted_paths[:10]

print("推荐黄金路径 (TOP 10):")
for i, (path, pred_yield) in enumerate(top_10_paths, 1):
    print(f"路径 {i}: {dict(zip(selected_stages, path))}, 预测良率: {pred_yield:.2f}%")
```

Step 6: 计算 Reliance Index (可信度)

```
def reliance_index(path, data, stages, encoder):
    correlations = []
    for stage, equip in zip(stages, path):
```

```
        binary_var = (data[stage] == equip).astype(int)
        corr = np.abs(np.corrcoef(binary_var, data['Yield'])[0,1])
        correlations.append(corr)

    avg_corr = np.mean(correlations)
    max_yield = data['Yield'].max()
    min_yield = data['Yield'].min()
    pred_yield = predict_yield(path, data, stages, encoder)

    rigp = 1 - (np.abs(pred_yield - avg_corr)) / (max_yield - min_yield)
    return rigp

# 计算TOP 10路径的Reliance Index
print("\nReliance Index (可信度):")
for i, (path, _) in enumerate(top_10_paths, 1):
    rigp = reliance_index(path, data, selected_stages, encoder)
    print(f"路径 {i}: 可信度: {rigp:.4f}")
```

🚀 完整执行流程总结：

- 数据预处理（整合数据）
- 阶段筛选（GOGA 算法）
- 枚举路径（笛卡尔积）
- 良率预测与路径排序
- 黄金路径推荐
- 计算 Reliance Index 提高推荐可信度

💻 实际执行提示：

- 上述代码适用于数据量在中小规模的数据。
- 如果设备数或路径数过于庞大，需考虑更高效的特征选择或路径采样方法。
- 具体数据的特征编码、模型类型或阈值参数应根据实际业务需求与数据特征进行调整。

以上代码完整体现了 GPSA 的核心思想与执行逻辑，帮助你实现一个基于多阶段制造流程的黄金路径搜索算法。

好的，下面我针对你的问题逐个修正并给出**更清晰完整的代码实现**：

一、数据预处理阶段（排除通过率较低的设备）

假设通过率阈值为 5%，即设备使用频率低于 5%的设备从数据中排除：

```
def filter_low_pass_equipment(df, threshold=0.05):
    equipment_counts = df['Equipment'].value_counts(normalize=True)
    equipment_to_keep = equipment_counts[equipment_counts >=
threshold].index
    df_filtered = df[df['Equipment'].isin(equipment_to_keep)].copy()
    return df_filtered

# 使用方法:
df_filtered = filter_low_pass_equipment(df, threshold=0.05)
```

二、GOGA 的 Group 筛选完整代码（含 group 选择）

GOGA 的核心思想是每次选择一个 **Group（即阶段）**，而非单个设备，因此需对每个 Group 所有设备进行 One-hot 编码作为整体。

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

def GOGA_group_selection(data, groups, yield_col='Yield', threshold=0.01):
    selected_groups = []
    remaining_groups = groups.copy()
    current_score = 0

    while remaining_groups:
        scores = []
        for group in remaining_groups:
            trial_groups = selected_groups + [group]
            X_trial = pd.get_dummies(data[trial_groups])
            model = LinearRegression().fit(X_trial, data[yield_col])
            score = model.score(X_trial, data[yield_col])  # 使用R^2衡量拟合效
果
            scores.append((group, score))
```

```
        # 选择效果最明显的Group
        best_group, best_score = max(scores, key=lambda x: x[1])

        if best_score - current_score > threshold:
            selected_groups.append(best_group)
            remaining_groups.remove(best_group)
            current_score = best_score
        else:
            break

    return selected_groups

# 调用方法示例:
all_groups = data.columns[:-1].tolist()   # 所有阶段
key_stages = GOGA_group_selection(data, all_groups, yield_col='Yield')
print("筛选出的关键阶段为: ", key_stages)
```

三、建立预测模型（含高阶交互项）

建立带有二阶或更高阶交互效应的回归模型。这里我们采用 PolynomialFeatures：

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

def predict_yield_with_interaction(path, data, stages,
interaction_order=2):
    # 创建二进制特征
    X_binary = pd.DataFrame(0, index=data.index, columns=stages)
    for stage, equip in zip(stages, path):
        X_binary[stage] = (data[stage] == equip).astype(int)

    # 交互项建模
    poly = PolynomialFeatures(degree=interaction_order,
interaction_only=True, include_bias=False)
    model = make_pipeline(poly, LinearRegression())
    model.fit(X_binary, data['Yield'])

    # 预测路径良率（取均值作为代表）
    predicted_yield = model.predict(X_binary).mean()
    return predicted_yield, model

# 为所有路径预测良率（示例：二阶交互项）
path_yields = []
models = {}
for path in all_paths:
    yield_pred, fitted_model = predict_yield_with_interaction(path, data,
key_stages, interaction_order=2)
    path_yields.append((path, yield_pred))
```

```
    models[path] = fitted_model  # 保存每条路径的模型（可选）

# 根据预测良率排序
sorted_paths = sorted(path_yields, key=lambda x: x[1], reverse=True)

# 显示 TOP10 路径
top_10_paths = sorted_paths[:10]
print("TOP 10 黄金路径及预测良率:")
for idx, (path, yield_pred) in enumerate(top_10_paths, 1):
    print(f"路径 {idx}: {dict(zip(key_stages, path))}, 预测良率:
{yield_pred:.2f}%")
```

【整合以上步骤：完整流程】

整体完整代码流程：

```
# === Step 1: 数据预处理（筛选低通过率设备）===
df_filtered = filter_low_pass_equipment(df, threshold=0.05)

# 数据 Pivot 为可用结构（每个样本一行，每个 stage 一列）
df_pivot = df_filtered.pivot_table(index='Sample_ID', columns='Stage',
values='Equipment', aggfunc='first')
yield_df = df_filtered[['Sample_ID',
'Yield']].drop_duplicates().set_index('Sample_ID')
data = df_pivot.join(yield_df).dropna()

# === Step 2: 使用 GOGA 进行关键阶段选择 ===
all_groups = data.columns[:-1].tolist()
key_stages = GOGA_group_selection(data, all_groups, yield_col='Yield',
threshold=0.01)
print("筛选出的关键阶段为：", key_stages)

# === Step 3: 生成关键阶段所有可能路径 ===
from itertools import product
equipments_per_stage = {stage: data[stage].unique() for stage in
key_stages}
all_paths = list(product(*equipments_per_stage.values()))
print(f"关键阶段生成路径数：{len(all_paths)}")

# === Step 4: 为每个路径构建交互效应预测模型并预测良率 ===
path_yields = []
models = {}
for path in all_paths:
    yield_pred, fitted_model = predict_yield_with_interaction(path, data,
key_stages, interaction_order=2)
    path_yields.append((path, yield_pred))
```

```
    models[path] = fitted_model

sorted_paths = sorted(path_yields, key=lambda x: x[1], reverse=True)
top_10_paths = sorted_paths[:10]

print("推荐的 TOP10 黄金路径：")
for idx, (path, yield_pred) in enumerate(top_10_paths, 1):
    print(f"路径 {idx}: {dict(zip(key_stages, path))}, 预测良率:
{yield_pred:.2f}%")

# === Step 5: Reliance Index 计算 ===
def reliance_index(path, data, stages):
    corr_sum = 0
    for stage, equip in zip(stages, path):
        binary_var = (data[stage] == equip).astype(int)
        corr_sum += np.abs(np.corrcoef(binary_var, data['Yield'])[0,1])
    avg_corr = corr_sum / len(stages)
    max_yield = data['Yield'].max()
    min_yield = data['Yield'].min()
    pred_yield, _ = predict_yield_with_interaction(path, data, stages,
interaction_order=2)
    rigp = 1 - (np.abs(pred_yield - avg_corr)) / (max_yield - min_yield)
    return rigp

print("\nReliance Index 可信度:")
for idx, (path, _) in enumerate(top_10_paths, 1):
    rigp = reliance_index(path, data, key_stages)
    print(f"路径 {idx}: 可信度: {rigp:.4f}")
```

🚀 总结说明：

- 已补充设备通过率筛选逻辑。
- 提供了完整 GOGA 的 group 选择代码。
- 构建并应用了二阶交互项预测模型。
- 提供了 Reliance Index 计算代码以验证推荐路径的可信性。

以上代码适合数据规模适中情形，如果路径组合极大，建议进一步优化效率。

下面我逐一清晰地解答你提出的两个问题：

🚩 一、group.so 完整代码（作为 GOGA 的替代方案）

Group LASSO (GLASSO) 是一种有效的分组特征选择方法，可以作为 GOGA 的备选方案。你可以使用 group-lasso Python 库实现。具体方法如下：

📌 安装 group-lasso 库：

```
pip install group-lasso
```

📌 完整的 Group LASSO 代码实现：

```
from group_lasso import GroupLasso
from sklearn.preprocessing import OneHotEncoder

def group_lasso_selection(data, groups, yield_col='Yield', alpha=0.01):
    X = data[groups]
    y = data[yield_col]

    # One-hot encode features
    encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
    X_encoded = encoder.fit_transform(X)

    # 设置 group 定义
    group_sizes = [len(data[col].unique()) for col in groups]
    group_array = []
    for idx, size in enumerate(group_sizes):
        group_array += [idx] * size
    group_array = np.array(group_array)

    # 建立 Group Lasso 模型
    gl = GroupLasso(
        groups=group_array,
        group_reg=alpha,
        l1_reg=0,
        frobenius_lipschitz=True,
        scale_reg="inverse_group_size",
        subsampling_scheme=None,
        supress_warning=True,
        n_iter=1000,
        tol=1e-3,
        warm_start=False,
        fit_intercept=True
    )

    gl.fit(X_encoded, y.values)

    # 获取选中的 group 索引
    selected_group_indices = np.unique(group_array[gl.sparsity_mask_])
    selected_groups = [groups[idx] for idx in selected_group_indices]
```

```
    return selected_groups

# 调用示例
selected_groups_glasso = group_lasso_selection(data, groups=all_groups, alpha=0.01)
print("Group LASSO 筛选的关键阶段:", selected_groups_glasso)
```

这样就得到了一个有效的备用方法。

🚩 **二、路径组合非常多时的效率优化建议：**

当路径组合数量非常庞大时（例如数百万或数十亿组合），枚举所有路径是不现实的。因此你需要以下的策略来优化效率：

🔑 **(1) 阶段分层或分段筛选：**

- 分步骤进行路径搜索，例如：
- 先从每个 group 内部最优路径选择，再组合跨 group 路径。
- 先找出影响最大的几个 group 或 stage，然后再进行局部组合分析。

🔑 **(2) 采用启发式或贪心算法：**

- 可以利用贪婪算法逐步加入路径：
- 先固定表现最好的阶段设备组合，再逐个阶段添加设备，每次选用当前阶段表现最好的少数设备组合。

示例（贪婪路径搜索伪代码）：

```
# 伪代码示例
optimal_path = []
for stage in selected_stages:
    best_equip = None
    best_yield = -np.inf
    for equip in data[stage].unique():
        current_path = optimal_path + [equip]
```

```
        predicted_yield, _ = predict_yield_with_interaction(current_path,
data, selected_stages[:len(current_path)], interaction_order=2)
        if predicted_yield > best_yield:
            best_yield = predicted_yield
            best_equip = equip
    optimal_path.append(best_equip)
```

这样能快速定位到较好的路径而无需全组合搜索。

🔑 **(3) 采用元启发式算法（Meta-heuristic algorithms）：**

• 例如遗传算法（Genetic Algorithm, GA）、粒子群优化（PSO）：

• 初始随机生成路径组合。

• 通过迭代优化（mutation、crossover 等操作）寻找高良率路径。

推荐使用库：PyGAD（遗传算法的 Python 实现）

```
pip install pygad
```

遗传算法伪代码框架：

```
import pygad

def fitness_func(solution, solution_idx):
    # solution 代表设备的选择方案
    selected_equipment = [equipments[i][solution[i]] for i in
range(len(selected_stages))]
    predicted_yield, _ = predict_yield_with_interaction(selected_equipment,
data, selected_stages, interaction_order=2)
    return predicted_yield  # 直接以良率作为适应度

# 初始化遗传算法（示例参数）
ga_instance = pygad.GA(num_generations=50,
                       num_parents_mating=10,
                       fitness_func=fitness_func,
                       sol_per_pop=20,
                       num_genes=len(selected_stages),
                       gene_space=[range(len(equipments[stage])) for stage
in selected_stages],
                       mutation_percent_genes=10)

ga_instance.run()
solution, solution_fitness, _ = ga_instance.best_solution()
```

🔑 **(4) 使用并行计算和 GPU 加速：**

• 如果你的计算机资源允许，可以使用并行处理库如 multiprocessing、joblib，甚至 GPU 计算来并行处理路径的预测任务。

并行示例：

```
from joblib import Parallel, delayed

results = Parallel(n_jobs=8)(delayed(predict_yield_with_interaction)(
    path, data, selected_stages, interaction_order=2) for path in
all_paths)
```

📌 **综上所述的推荐优化方案：**

• **首选贪心算法或阶段式组合搜索**：能快速获得较优路径。

• **如果想进一步优化**：推荐使用遗传算法等 Meta-heuristic 方法。

• **并行处理与 GPU 加速**：可与上述方法结合使用进一步提升效率。

🚩 **总结：**

• 已提供 group.so 的具体实现（即 GroupLasso 代码）。

• 针对路径组合数量过大的情况，提出了详细的优化建议与具体实现参考代码。

你可根据具体数据规模和实际算力选择合适的方法进行优化处理。