

了解你的数据结构了。我们明确一下你的数据结构：

SampleID	Stage	Equipment	Yield
1	stage_1	EQP_A	95.5
1	stage_2	EQP_B	95.5
...	...	...	...
2	stage_1	EQP_C	90.2
2	stage_2	EQP_D	90.2

即每个 SampleID 会对应多行数据，每个 Stage 有一个设备，每个 SampleID 有一个对应的良率（所有 stage 相同）。

---

## ► 完整的步骤及代码实现

以下是完整的流程，包括从原始数据到 Group Lasso 进行 Stage 选择：

### Step 1: 数据转换 (Pivot)

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from group_lasso import GroupLasso

# 假设 df 为原始数据
# 示例数据结构
# | SampleID | Stage | Equipment | Yield |

# Pivot 数据（每个 SampleID 为一行，每个 stage 是一列，值是 Equipment）
df_pivot = df.pivot_table(index=['SampleID', 'Yield'],
                           columns='Stage',
                           values='Equipment',
                           aggfunc='first').reset_index()

# 检查一下转换后的数据
print(df_pivot.head())
```

此时数据结构类似：

SampleID	Yield	stage_1	stage_2	... stage_n
1	95.5	EQP_A	EQP_B	... EQP_C
2	90.2	EQP_C	EQP_D	... EQP_A

---

## Step 2: 构造 Group Lasso 的输入数据

Group Lasso 要求输入的 X 是数值型矩阵，我们需要对每个 stage 下的 equipment 进行 One-hot 编码，并记录每个 stage 对应的 group index。

```
# 提取特征和目标
X_raw = df_pivot.drop(columns=['SampleID', 'Yield'])
y = df_pivot['Yield'].values

# 进行 One-hot encoding
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
X_encoded = encoder.fit_transform(X_raw)

# 记录每个 feature 对应的 stage (group) 信息
feature_names = encoder.get_feature_names_out(X_raw.columns)

# 构建 group 数组
group_list = []
for feature in feature_names:
    stage = feature.split('_')[0] # stage 名称为前缀
    group_list.append(stage)

# 将 stage 名称转换为整数 Group 索引
unique_stages = np.unique(group_list)
group_dict = {stage: idx for idx, stage in enumerate(unique_stages)}
groups = np.array([group_dict[stage] for stage in group_list])

# 检查结果
print('feature_names:', feature_names)
print('groups:', groups)
```

解释：

- 每个设备编码后的特征名类似：stage\_1\_EQP\_A
- 提取 stage\_1 作为 group 信息

---

## Step 3: 使用 Group Lasso 选择关键 Stage

```
# 定义 GroupLasso 模型
gl = GroupLasso(
    groups=groups,
```

```

        group_reg=0.05, # 正则化参数, 可调节
        l1_reg=0,
        frobenius_lipschitz=True,
        scale_reg="inverse_group_size",
        suppress_warning=True,
        n_iter=1000,
        tol=1e-3,
        fit_intercept=True,
        random_state=42
    )

# 拟合模型
gl.fit(X_encoded, y)

# 获取非零系数对应的 group 索引
selected_groups_indices = np.unique(groups[gl.sparsity_mask_])

# 获取被选中的 Stage 名称
selected_stages = unique_stages[selected_groups_indices]

print('Selected Important Stages:', selected_stages)

```

---

### ► 以上代码的完整流程说明：

- **Step 1** 将原始的长格式数据 Pivot 为宽格式（每个 stage 一列）。
- **Step 2** 对每个 stage 的 equipment 进行 one-hot 编码，并记录每个特征属于哪个 stage（group）。
- **Step 3** 利用 Group Lasso 方法找到影响良率的重要 stage。

这样，你就能准确高效地完成对 Stage 的筛选了。

下面逐个回答你的两个问题，并给出清晰的代码实现：

---

## ► 一、Group Lasso 参数调优方法

### 📌 重要参数说明：

- **group\_reg**（关键参数）：

- 影响特征选择稀疏性。
- 较大值 → 更少特征，更严格选择。
- 较小值 → 更多特征，更宽松选择。

- **l1\_reg**：

- 一般设为 0，不参与调参（只在需要额外 L1 惩罚时启用）。

### 📌 调参策略（推荐）：

- 使用交叉验证（Cross-validation），以验证预测性能与选择稳定性。
- Grid Search 或者手动设置多个不同的 **group\_reg**，观察模型的拟合效果。

### 📌 示例代码 (Grid Search CV):

```
from sklearn.model_selection import GridSearchCV, KFold
from group_lasso import GroupLasso
from sklearn.metrics import make_scorer, r2_score

# 定义 GridSearch 参数空间
param_grid = {'group_reg': [0.001, 0.01, 0.05, 0.1, 0.5, 1.0]}

# 自定义模型 Wrapper 以兼容 GridSearchCV
class GLassoWrapper(GroupLasso):
    def set_params(self, **params):
        for param, value in params.items():
            setattr(self, param, value)
        return self

gl_wrapper = GLassoWrapper(
    groups=groups,
    l1_reg=0,
    frobenius_lipschitz=True,
    scale_reg="inverse_group_size",
    suppress_warning=True,
    n_iter=1000,
```

```

        tol=1e-3,
        fit_intercept=True,
        random_state=42
    )

cv = KFold(n_splits=5, shuffle=True, random_state=42)
grid = GridSearchCV(gl_wrapper, param_grid, scoring=make_scorer(r2_score),
cv=cv)
grid.fit(X_encoded, y)

print("Best group_reg:", grid.best_params_)


```

通过这种方式，你就能科学有效地确定最佳参数。

## 二、选中 Stage 后进行 Third-Order（3 阶）交互效应路径预测

这部分实现与原论文一致：

- 每个 Stage 从选中的设备集合中组合成不同路径。
- 针对每条路径构建带有三阶交互效应的回归模型。
- 用 AIC 选出每条路径的最优模型。
- 排序并推荐良率最高的路径（黄金路径）。

 完整实现代码：

### Step 1: 提取关键 Stage 的数据

假设前一步你获得了关键 Stage 列表：selected\_stages

```

from itertools import product
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import warnings
warnings.filterwarnings("ignore")

X_stages = df_pivot[selected_stages]
y = df_pivot['Yield'].values

# 获得每个 stage 的设备列表

```

```
equipments_per_stage = {stage: X_stages[stage].unique() for stage in
selected_stages}
```

```
# 生成路径组合
```

```
all_paths = list(product(*equipments_per_stage.values()))
```

```
print(f"共生成路径数量: {len(all_paths)}")
```

---

## Step 2: 为每条路径建立最高 3 阶交互模型，并使用 AIC 进行模型选择

定义路径良率预测函数:

```
def aic_score(n, mse, num_params):
    return n * np.log(mse) + 2 * num_params

best_path_results = []

for path in all_paths:
    # 创建路径的 01 特征
    X_binary = pd.DataFrame()
    for stage, equip in zip(selected_stages, path):
        X_binary[f"{stage}={equip}"] = (X_stages[stage] ==
equip).astype(int)

    best_aic = np.inf
    best_pred_yield = None

    # 尝试 1, 2, 3 阶交互模型
    for degree in [1, 2, 3]:
        poly = PolynomialFeatures(degree=degree, interaction_only=True,
include_bias=False)
        X_poly = poly.fit_transform(X_binary)

        model = LinearRegression().fit(X_poly, y)
        y_pred = model.predict(X_poly)
        mse = mean_squared_error(y, y_pred)

        # 计算 AIC
        aic = aic_score(len(y), mse, X_poly.shape[1])

        if aic < best_aic:
            best_aic = aic
            best_pred_yield = y_pred.mean() # 路径良率预测用均值表示

    best_path_results.append((path, best_pred_yield, best_aic))

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

```
# 推荐前 10 路径
print("推荐黄金路径 Top 10 (路径、预测良率、AIC):")
for idx, (path, yield_pred, aic) in enumerate(sorted_paths[:10], 1):
    print(f"{idx}. 路径: {dict(zip(selected_stages, path))}, 预测良率: {yield_pred:.2f}, AIC: {aic:.2f}")
```

---

### Step 3: Reliance Index 验证路径可靠性 (与原文一致)

```
def reliance_index(path, X_stages, y, stages):
    corr_sum = 0
    for stage, equip in zip(stages, path):
        binary_var = (X_stages[stage] == equip).astype(int)
        corr = np.abs(np.corrcoef(binary_var, y)[0,1])
        corr_sum += corr

    avg_corr = corr_sum / len(stages)
    max_yield = y.max()
    min_yield = y.min()

    # 用之前计算的 best_pred_yield
    poly = PolynomialFeatures(degree=3, interaction_only=True,
include_bias=False)
    X_binary = pd.DataFrame({f"{stage}={equip}": (X_stages[stage] ==
equip).astype(int) for stage, equip in zip(stages, path)})
    model = LinearRegression().fit(poly.fit_transform(X_binary), y)
    pred_yield = model.predict(poly.fit_transform(X_binary)).mean()

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

print("\nReliance Index (路径可靠性):")
for idx, (path, _, _) in enumerate(sorted_paths[:10], 1):
    rigp = reliance_index(path, X_stages, y, selected_stages)
    print(f"{idx}. 路径: {dict(zip(selected_stages, path))}, Reliance Index: {rigp:.4f}")
```

---

### 🎯 完整流程总结:

1. 使用交叉验证优化 Group Lasso 参数。
2. 利用选中的关键 stage 构建三阶交互效应预测模型。
3. 每个路径用 AIC 选择最优交互阶数。

4. 排序并推荐黄金路径。
5. 用 Reliance Index 检验路径的预测可靠性。

以上方法和代码准确、严格对应于你给出的论文实现方法和流程需求，能很好地解决你当前的问题。

原文中提到的正交方法，称为 **Group Orthogonal Greedy Algorithm (GOGA)**，是一种基于贪心正交逐步回归的算法。

它的主要思想：

- 每次迭代选择一个 **Stage (group)**，加入到模型中；
- 评估加入该 **Stage** 后模型的性能（一般用类似 AIC 的评价指标）；
- 通过正交化过程保证选取的 **group** 之间互相独立。

下面给你完整的 Python 代码实现：

---

### ► Step-by-step 完整实现 Group Orthogonal Greedy Algorithm (GOGA)

数据前处理（跟之前一样）：

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import warnings
warnings.filterwarnings("ignore")

# 假设 df_pivot 是之前处理好的数据
X_raw = df_pivot.drop(columns=['SampleID', 'Yield'])
y = df_pivot['Yield'].values

# One-hot encode 所有 Stage 的 Equipment
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
X_encoded = encoder.fit_transform(X_raw)
feature_names = encoder.get_feature_names_out(X_raw.columns)
```



```

# 获取每个特征所属的 group(stage)
group_names = [f.split('_')[0] for f in feature_names]
unique_groups = np.unique(group_names)
group_indices = {g: np.where(np.array(group_names)==g)[0] for g in
unique_groups}

# 定义 HDGIC 准则 (原文的评估标准)
def HDGIC(n, mse, num_params):
    return n * np.log(mse) + num_params * np.log(n) * (np.log(n) -
np.log(np.log(n)))

# 正交化函数
def orthogonalize(X_new, X_existing):
    if X_existing.size == 0:
        return X_new
    proj = X_existing @ np.linalg.pinv(X_existing) @ X_new
    return X_new - proj

# Group OGA 主函数
def Group_OGA(X, y, group_indices, max_groups=None):
    selected_groups = []
    remaining_groups = list(group_indices.keys())
    X_selected = np.empty((X.shape[0], 0))
    best_hdgic = np.inf
    best_model = None

    n_samples = len(y)

    while remaining_groups and (max_groups is None or len(selected_groups)
< max_groups):
        group_scores = []
        for group in remaining_groups:
            indices = group_indices[group]
            X_candidate = X[:, indices]

            # 正交化当前 group 相对于已选的 group
            X_ortho = orthogonalize(X_candidate, X_selected)

            X_temp = np.hstack([X_selected, X_ortho])
            model = LinearRegression().fit(X_temp, y)
            y_pred = model.predict(X_temp)
            mse = mean_squared_error(y, y_pred)
            num_params = X_temp.shape[1]
            hdgic_score = HDGIC(n_samples, mse, num_params)

            group_scores.append((group, hdgic_score, X_ortho, model))

        # 选择使 HDGIC 最小的 group
        group_scores.sort(key=lambda x: x[1])
        best_group, group_hdgic, best_X_ortho, best_group_model =
group_scores[0]

        if group_hdgic < best_hdgic:
            best_hdgic = group_hdgic
            selected_groups.append(best_group)
            remaining_groups.remove(best_group)
            X_selected = np.hstack([X_selected, best_X_ortho])

```

```
        best_model = best_group_model
        print(f"已选择 Stage: {best_group}, 当前 HDGIC:
{group_hdgic:.2f}")
    else:
        # 若 HDGIC 未改善, 则停止选择
        break

    return selected_groups, best_model

# 执行 Group OGA
selected_stages_goga, final_model = Group_OGA(X_encoded, y, group_indices)

print("最终选中的关键 Stage 为: ", selected_stages_goga)
```

---

### 📌 代码核心逻辑说明:

#### 1. One-hot encoding 特征:

每个 stage 下的设备分别编码, 获得 X 矩阵。

#### 2. 正交化步骤 (orthogonalize) :

每次新加入 group 对应的特征需要对已选特征做正交化处理, 以保证特征组之间的独立性。

#### 3. HDGIC 指标 (原文的评价指标) :

根据文章原定义的 HDGIC 标准来衡量加入 group 后的模型优劣。

#### 4. 逐步迭代选择:

每次迭代选择能够最大程度降低 HDGIC 的 group, 直到无法再显著降低为止。

---

### 🚩 与 Group Lasso 区别:

#### • GOGA:

- 逐步加入 group, 每次评估效果, 有明确停止条件;
- 选中的 group 特征之间是正交化的, 避免共线性。

#### • Group Lasso:

- 一次性优化得到稀疏解；
- 对共线性敏感，依赖正则化参数调节。

你可以将 GOGA 作为备选方案，跟 Group Lasso 进行对比分析。

---

这样，你就清楚地拥有了文章中所描述的正交方法（GOGA）的完整实现代码。