In [1]:

```
# 导入所需要的函数模块
from marvel.AntPrint import ant_read_data,ant_write_data # 读取数据模块
from marvel.AntPrint import ant_print_all
from marvel.AntPrint import result_printer
from marvel.AntPrint.model import PSM
# 导入建模所需要的包

from scipy import stats
import seaborn as sns
import numpy as np
import pandas as pd
from scipy.stats import norm
import statsmodels.api as sm
from scipy.stats.mstats import winsorize
from linearmodels.iv.absorbing import AbsorbingLS, Interaction
from linearmodels import PanelOLS
from marvel.AntPrint import *
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
from statsmodels.formula.api import poisson
import patsy
from itertools import product
from sklearn.metrics import pairwise_distances
from statsmodels.stats.sandwich_covariance import cov_cluster
import itertools
```

Last executed in 20 s, finished in 21:48:05 2025-06-30

In [2]:

```
df = ant_read_data('workdata', mid_table = True)
```

Last executed in 11 s, finished in 21:48:16 2025-06-30

In [3]:

```
df['报告期1'] = df['报告期'].astype('str')
df['报告期1'] = pd.to_datetime(df['报告期1'])
df['year_month'] = df['报告期1'].apply(lambda x: x.strftime("%Y-%m"))
df['use'] = (df['累计pv'] > 0).astype(int)
df = df.reset_index(drop=True)
df['app_start'] = df.groupby(['匿名化用户id_rank'])['year_month'].transform(
lambda x: x.loc[df['use'] == 1].min())
df['app_start'] = pd.to_datetime(df['app_start'])
df['year_month'] = pd.to_datetime(df['year_month'])
```

Last executed in 9 s, finished in 21:48:25 2025-06-30

In [4]:

```
# 固定效应
df['type'] = df['period_length'].astype('category')
df['year_month_cat'] = df['year_month'].astype('category')
df['citycode_cat'] = df['citycode'].astype('category')
df['time_city'] = df.apply(lambda row: '{}_{}'.format(row['year_month_cat'],
row['citycode_cat']), axis=1).astype('category')
df['time_type'] = df.apply(lambda row: '{}_{}'.format(row['year_month_cat'],
row['type']), axis=1).astype('category')
df['用户id_rank']  = df['匿名化用户id_rank'].astype('category')

# 聚类层级
df['cluster_id'] = df['匿名化用户id_rank'].astype('category')
```

Last executed in 6 s, finished in 21:48:31 2025-06-30

In [5]:

```
regdata = df
ant_print_all(regdata[['new_use','period_length','性别','是否有医保','是否有社保',
'年龄','n_module_consumpt','all_consumpt','online_consumpt',
'offline_consumpt','num_consumpt','是否基金交易','余额宝余额','基金余额']],
model_method="describe")
```

```
******************************
describe
            new_use  period_length            性别    是否有医保    是否有社保 \
count  60060.000000   60060.000000  60060.000000  60060.000000  60060.000000

mean       0.739011       6.640293      0.400733      0.433333      0.332967

std        0.439178       5.084558      0.490051      0.495540      0.471279

min        0.000000       3.000000      0.000000      0.000000      0.000000

25%        0.000000       3.000000      0.000000      0.000000      0.000000

50%        1.000000       4.000000      0.000000      0.000000      0.000000

75%        1.000000       8.000000      1.000000      1.000000      1.000000

max        1.000000      21.000000      1.000000      1.000000      1.000000


                年龄 n_module_consumpt  all_consumpt  online_consumpt  \
count  60060.000000       60060.000000  60060.000000     60060.000000
mean      37.105128           5.145022   3929.583921       868.356031
std       12.558085           3.652004   8664.305087      1852.267457
min       18.000000           0.000000      0.000000         0.000000
25%       28.000000           2.000000    176.922500        30.000000
50%       34.000000           5.000000   1010.210000       250.950000
75%       45.000000           8.000000   3338.412500       809.402500
max       87.000000          21.000000  58067.040000     12774.180000

       offline_consumpt  num_consumpt        是否基金交易     余额宝余额 \
count      60060.000000  60060.000000  60038.000000   60060.000000
mean        2911.814177     30.225108      0.234518    5013.346014
std         7609.723784     43.815044      0.423701   16153.969313
min            0.000000      0.000000      0.000000       0.000000
25%            0.000000      4.000000      0.000000       0.000000
50%          327.000000     16.000000      0.000000      16.010000
75%         1985.160000     41.000000      0.000000    1073.037500
max        52202.970000   2121.000000      1.000000  108925.310000

               基金余额
count  60060.000000
mean     960.688443
std     5751.150588
min        0.000000
25%        0.000000
50%        0.000000
75%        0.100000
max    48361.160000
******************************
```

Last executed in 0 s, finished in 21:48:31 2025-06-30

正文 ¶

图1 动态效应 ¶

基准 ¶

In [6]:

```
regdata['relative_months'] = (regdata['year_month'].dt.year - regdata[
'app_start'].dt.year) * 12 + (regdata['year_month'].dt.month - regdata[
'app_start'].dt.month)

# 使用 np.where 进行调整
regdata['relative_months'] = np.where(regdata['relative_months'] < -10, -10,
np.where(regdata['relative_months'] >10, 10, regdata['relative_months']))
# 输出调整后的 relative_months 变量描述
ant_print_all(regdata['relative_months'], model_method="describe")


# 创建时间段的虚拟变量，-1 期除外
periods = list(range(-6, 11))
periods.remove(-1)
for period in periods:
    regdata[f'period_{period}'] = (regdata['relative_months'] == period).
astype(int)

if envir()=='仿真环境':
    pass
else:
    cats = regdata[['用户id_rank','time_city','time_type']]
    cluster_id = regdata['cluster_id']
    x = regdata[[f'period_{period}' for period in periods]]
    y_list = ['lnn_module_consumpt']
    models = []
    results = []
    for i in y_list:
        y = regdata[i]
        model = AbsorbingLS(y, x, absorb=cats,drop_absorbed=True)
        result = model.fit(method="lsmr", cov_type="clustered", clusters=
cluster_id)
        models.append(model)
        results.append(result)

    # 结果输出
    ant_print_all(result_printer, model_method="summary_col",
                  results=results, model_names = y_list,
                  regressor_order = [f'period_{period}' for period in
periods], drop_omitted=True,
                  info_list=["rsquared"], info_dict={"count": lambda x: x.
nobs}, info_order=["rsquared", "count"],float_format='%.3f')
```

```
******************************
describe
count    60060.000000
mean         3.641642
std          6.171012
min        -10.000000
25%         -1.000000
50%          5.000000
75%         10.000000
max         10.000000
Name: relative_months, dtype: float64
******************************

******************************

==============================
          lnn_module_consumpt
------------------------------
period_-6 0.009
          (0.013)
period_-5 0.011
          (0.013)
period_-4 0.000
          (0.013)
period_-3 0.003
          (0.012)
period_-2 -0.009
          (0.011)
period_0  0.044***
          (0.011)
period_1  0.037***
          (0.013)
period_2  0.035**
          (0.014)
period_3  0.046***
          (0.015)
period_4  0.051***
          (0.016)
period_5  0.047***
          (0.018)
period_6  0.044**
          (0.019)
period_7  0.033
          (0.020)
period_8  0.027
          (0.022)
period_9  0.003
          (0.023)
period_10 0.023
          (0.026)
rsquared  0.753
count     60060.000
```

```
============================
Standard errors in
parentheses.
* p<.1, ** p<.05, ***p<.01
****************************


Last executed in 3 s, finished in 21:48:35 2025-06-30
```

多期**DID**稳健**Borusyak et al.(2024)** ¶

In [7]:

```
def clustered_se(model, cluster_var):
    return np.sqrt(np.diag(cov_cluster(model, cluster_var)))

def calculate_dynamic_effects(data, treatment_col, time_col, outcome_col,
unit_col, rel_time_col,
                              min_rel_time=-5, max_rel_time=5, fixed_effects
=None, controls=None):
    """
  计算动态处理效应

  参数:
    data: DataFrame, 包含所有必要变量的面板数据
    treatment_col: str, 处理变量的列名
    time_col: str, 时间变量的列名
    outcome_col: str, 结果变量的列名
    unit_col: str, 单位ID变量的列名
    rel_time_col: str, 相对时间变量的列名
    min_rel_time: int, 最小相对时间 (默认 -5)
    max_rel_time: int, 最大相对时间 (默认 5)
    fixed_effects: list, 固定效应的列名列表
    controls: list, 控制变量的列名列表

  返回:
    DataFrame, 包含每个相对时间点的处理效应估计和标准误
    """
    dynamic_effects = []

    # 数据预处理
    data = data[(data[rel_time_col] != -99) &
                (data[rel_time_col] >= min_rel_time) &
                (data[rel_time_col] <= max_rel_time)].copy()

    # 准备固定效应
    if fixed_effects:
        for fe in fixed_effects:
            data[fe] = data[fe].astype('category')

    # 构建公式
    formula = f"{outcome_col} ~ "
    if controls:
        formula += " + ".join(controls)
    if fixed_effects:
        if controls:
            formula += " + "
        formula += " + ".join([f"C({fe})" for fe in fixed_effects])
```

```
    for t in range(min_rel_time, max_rel_time + 1):
        current_data = data[data[rel_time_col] == t]
        control = data[data[treatment_col] == 0]

        # 估计控制组模型
        model = sm.formula.ols(formula, data=control)
        result = model.fit()

        # 预测处理组的反事实结果
        current_data['y_hat'] = result.predict(current_data)
        current_data['effect'] = current_data[outcome_col] - current_data[
'y_hat']

        att = current_data['effect'].mean()

        # 计算聚类标准误
        model_se = sm.OLS(current_data['effect'],
                          sm.add_constant(current_data[controls] if controls
else pd.DataFrame(index=current_data.index)))
        results_se = model_se.fit()
        se_att = clustered_se(results_se, current_data[unit_col])[0]

        dynamic_effects.append({'rel_time': t, 'coefficient': att,
'std_error': se_att})
    return pd.DataFrame(dynamic_effects)
```

Last executed in 0 s, finished in 21:48:35 2025-06-30

In [8]:

```
regdata['relative_months'] = (regdata['year_month'].dt.year - regdata[
'app_start'].dt.year) * 12 + (regdata['year_month'].dt.month - regdata[
'app_start'].dt.month)

if envir()=='仿真环境':
    pass
else:
    # 使用示例
    # 计算动态处理效应
    dynamic_effects = calculate_dynamic_effects(
        data=regdata,
        treatment_col='new_use',
        time_col='year_month',
        outcome_col='lnn_module_consumpt',
        unit_col='匿名化用户id_rank',
        rel_time_col='relative_months',
        min_rel_time=-6,
        max_rel_time=10,
        fixed_effects=['匿名化用户id_rank', 'year_month'],
    )


    ant_print_all(dynamic_effects[['rel_time','coefficient','std_error']],
model_method = 'value_counts')
```

```
******************************
rel_time  coefficient  std_error
-6          0.003720    0.010533     1
 3          0.066380    0.010488     1
 9          0.040912    0.012572     1
 8          0.059072    0.012339     1
 7          0.061619    0.011905     1
 6          0.068247    0.011353     1
 5          0.071843    0.011274     1
 4          0.070459    0.010779     1
 2          0.053761    0.010232     1
-5          0.000959    0.008581     1
 1          0.049094    0.009454     1
 0          0.052159    0.008956     1
-1          0.001987    0.005439     1
-2         -0.002043    0.006383     1
-3          0.001125    0.007357     1
-4         -0.005801    0.007974     1
 10         0.054834    0.012902     1
dtype: int64
******************************
```

Last executed in 894 s, finished in 22:03:28 2025-06-30

多期**DID**稳健**Sun and Abraham (2021)** ¶

In [9]:

```
def sun_abraham_estimator(df, y, tr, t, i, rel_time_var, cohorts,
never_treated=None,
                          cluster=None, min_rel_time=-5, max_rel_time=5,
fixed_effects=None):
    """
    Implement the Sun and Abraham (2021) interaction-weighted estimator with
customizable relative time range
    and fixed effects.
    """

    # 创建相对时间变量
    df['rel_time'] = (df[t].dt.year - df[rel_time_var].dt.year) * 12 + (df[t
].dt.month - df[rel_time_var].dt.month)

    # 确定所有可能的相对时间，限制在指定范围内
    all_event_times = sorted([time for time in df['rel_time'].unique()
                              if min_rel_time <= time <= max_rel_time])

    event_times = [t for t in all_event_times if t != -1]  # 排除基准期

    # 创建事件时间哑变量
    for lag in event_times:
        df[f'e{lag}'] = (df['rel_time'] == lag).astype(int)

    # 创建队列-事件时间交互项
    interactions = []
    for cohort in cohorts:
        for lag in event_times:
            col_name = f'e{lag}_c{cohort}'
            df[col_name] = df[f'e{lag}'] * (df[rel_time_var] == cohort).
astype(int)
            interactions.append(col_name)

    # 准备回归变量
    X = sm.add_constant(df[interactions])
    y_var = df[y]

    # 准备固定效应
    if fixed_effects:
        if 'unit' in fixed_effects:
            X= pd.concat([X,pd.get_dummies(df[i],prefix='id',drop_first =
True)],axis=1)
        if 'time' in fixed_effects:
            X= pd.concat([X,pd.get_dummies(df[t],prefix='time',drop_first =
True)],axis=1)
```

```
    # 运行回归
    model = sm.OLS(y_var, X)
    results = model.fit(cov_type='cluster', cov_kwds={'groups': df[cluster
]})

    # 计算队列权重
    cohort_sizes = df.groupby(rel_time_var).size()
    total_treated = cohort_sizes[cohorts].sum()
    weights = {cohort: cohort_sizes[cohort] / total_treated for cohort in
cohorts}

    dynamic_effects = []

    for lag in range(min_rel_time, max_rel_time + 1):
        if lag == -1:
            coef = 0
            se = 0
        else:
            # 计算加权平均处理效应
            cohort_effects = []
            cohort_weights = []
            for cohort in cohorts:
                if f'e{lag}_c{cohort}' in results.params.index:
                    cohort_effects.append(results.params[f'e{lag}_c{cohort}'
])
                    cohort_weights.append(weights[cohort])

            if cohort_effects:
                coef = np.average(cohort_effects, weights=cohort_weights)

                # 计算标准误

                var_coef = 0
                for i, cohort in enumerate(cohorts):
                    if f'e{lag}_c{cohort}' in results.params.index:
                        var_coef += (cohort_weights[i] ** 2) * results.
cov_params().loc[f'e{lag}_c{cohort}', f'e{lag}_c{cohort}']
                se = np.sqrt(var_coef)
            else:
                coef = np.nan
                se = np.nan

        sample_size = df[df['rel_time'] == lag].shape[0]

        dynamic_effects.append({
            'relative_time': lag,
            'coefficient': coef,
            'std_error': se,
            'sample_size': sample_size
        })
```

```
    # 将结果转换为DataFrame
    results_df = pd.DataFrame(dynamic_effects)

    return results_df
```

Last executed in 0 s, finished in 22:03:28 2025-06-30

In [10]:

```
regdata['app_start'] = regdata['app_start'].fillna(0)
if regdata.shape[0] > 20000:
    # 确定实际的处理组
    cohorts = sorted(regdata['app_start'].unique())
    cohorts = [c for c in cohorts if c != 0]  # 排除从未处理的组

    # 运行 Sun-Abraham 估计器，指定相对时间范围和固定效应
    results = sun_abraham_estimator(
        regdata, 'lnn_module_consumpt', 'new_use', 'year_month', '匿名化用户id_rank',
        'app_start', cohorts, cluster="用户id_rank",
        fixed_effects=['匿名化用户id_rank', 'year_month'],
        min_rel_time=-6, max_rel_time=10
    )

    # 打印结果
    ant_print_all(results[['relative_time','coefficient','std_error']], model_method="value_counts")
```

```
******************************
relative_time  coefficient  std_error
-6               0.017291    0.009495     1
 3               0.050225    0.013942     1
 9              -0.018858    0.013269     1
 8               0.023513    0.013397     1
 7               0.042330    0.013494     1
 6               0.056147    0.013490     1
 5               0.051022    0.013841     1
 4               0.050746    0.013952     1
 2               0.044862    0.014129     1
-5               0.016250    0.010176     1
 1               0.048845    0.014037     1
 0               0.044124    0.013811     1
-1               0.000000    0.000000     1
-2               0.011140    0.012874     1
-3               0.015263    0.011678     1
-4               0.018044    0.010819     1
 10             -0.026648    0.012888     1
dtype: int64
******************************


Last executed in 8 s, finished in 22:03:36 2025-06-30
```

图2 消费活跃程度与消费转移 ¶

系数提取出来后用stata画图

In [11]:

```
regdata = regdata.reset_index(drop=True)


if regdata.shape[0]>200:
    cats = regdata[['用户id_rank','time_city','time_type']]
    cluster_id = regdata['cluster_id']

    for col in ['avg_transactions', 'consumption_month']:
        models = []
        results = []
        qgroup = [f'{col}_q{i}_x_new_use' for i in range(1, 5)]
        x = regdata[qgroup]
        y = regdata['lnn_module_consumpt']
        model = AbsorbingLS(y, x, absorb=cats)
        result = model.fit(method="lsmr", cov_type="clustered", clusters=
cluster_id)
        models.append(model)
        results.append(result)

        # 结果输出
        ant_print_all(result_printer, model_method="summary_col",
                      results=results, model_names =['lnn_module_consumpt'],
                      regressor_order =  qgroup, drop_omitted=True,
                      info_list=["rsquared"], info_dict={"count": lambda x:
x.nobs}, info_order=["rsquared", "count"],float_format='%.3f')
```

```
****************************

================================================
                               lnn_module_consumpt
------------------------------------------------
avg_transactions_q1_x_new_use 0.073***
                              (0.022)
avg_transactions_q2_x_new_use 0.040*
                              (0.021)
avg_transactions_q3_x_new_use 0.039**
                              (0.018)
avg_transactions_q4_x_new_use 0.043***
                              (0.014)
rsquared                      0.753
count                         60060.000
================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
****************************

****************************

================================================
                                lnn_module_consumpt
------------------------------------------------
consumption_month_q1_x_new_use 0.041
                               (0.027)
consumption_month_q2_x_new_use 0.066***
                               (0.019)
consumption_month_q3_x_new_use 0.047***
                               (0.014)
consumption_month_q4_x_new_use 0.038***
                               (0.014)
rsquared                       0.753
count                          60060.000
================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
****************************



Last executed in 2 s, finished in 22:03:38 2025-06-30
```

附录一 ¶

表**I 2 Goodman Bacon**分解结果+图**I 1 Goodman Bacon** 分解权重图 ¶

In [12]:

```
def bacon(formula, data, id_var, time_var, fixed_effects, cluster):
    outcome_var, treated_var, control_vars = unpack_variable_names(formula)
    data = rename_vars(data, id_var, time_var, outcome_var, treated_var)

    treatment_group_calc = create_treatment_groups(data, control_vars)
    two_by_twos, data = treatment_group_calc['two_by_twos'],
treatment_group_calc['data']

    if control_vars:
        control_formula = f"treated ~ " + " + ".join(control_vars)
        data = run_fwl(data, control_formula)

    if not control_vars:
        for i in range(len(two_by_twos)):
            treated_group, untreated_group = two_by_twos.iloc[i][['treated',
'untreated']]
            data1 = subset_data(data, treated_group, untreated_group)
            weight = calculate_weights(data1, treated_group, untreated_group
)
            data1 = data1.set_index(['id', 'time'])
            X = sm.add_constant(data1['treated'])
            y_var = data1['outcome']

            # 使用 PanelOLS 来处理固定效应
            if fixed_effects:
                model = PanelOLS(y_var, X, entity_effects = 'id' in
fixed_effects,time_effects = 'time' in fixed_effects)

            else:
                model = sm.OLS(y_var,X)

            results = model.fit(cov_type='clustered', cluster_entity=True)

            estimate = results.params['treated']
            two_by_twos.loc[i, 'estimate'] = estimate
            two_by_twos.loc[i, 'weight'] = weight

        two_by_twos = scale_weights(two_by_twos)

        return two_by_twos

    else:
        Omega = calculate_Omega(data)
        beta_hat_w = calculate_beta_hat_w(data)
        r_collapse_x_p = collapse_x_p(data, control_vars)
        data, g_control_formula = r_collapse_x_p['data'], r_collapse_x_p[
'g_control_formula']

        for i in range(len(two_by_twos)):
```

```
            treated_group, untreated_group = two_by_twos.iloc[i][['treated',
'untreated']]
            data1 = data[data['treat_time'].isin([treated_group,
untreated_group])]
            weight_est = calc_controlled_beta_weights(data1,
g_control_formula)
            two_by_twos.loc[i, 'weight'] = weight_est['s_kl']
            two_by_twos.loc[i, 'estimate'] = weight_est['beta_hat_d_bkl']
        two_by_twos = scale_weights(two_by_twos)

        # Convert Omega to a string representation if it's not already a
simple type
        if not isinstance(Omega, (int, float, str)):
            Omega = str(Omega)

        result_df = pd.DataFrame({
            'beta_hat_w': [beta_hat_w],
            'Omega': [Omega],
            "two_by_twos": [two_by_twos]
        })

        # Add the two_by_twos DataFrame as additional columns
        result_df = pd.concat([result_df, two_by_twos.reset_index()], axis=1
)

        return result_df
```

```
Last executed in 0 s, finished in 22:03:38 2025-06-30
```

In [13]:

```
def unpack_variable_names(formula):
    lhs, rhs = formula.split('~')
    outcome_var = lhs.strip()
    rhs_vars = rhs.split('+')
    treated_var = rhs_vars[0].strip()
    control_vars = [var.strip() for var in rhs_vars[1:]]
    return outcome_var, treated_var, control_vars

def rename_vars(data, id_var, time_var, outcome_var, treated_var):
    data = data.rename(columns={id_var: 'id', time_var: 'time', outcome_var:
'outcome', treated_var: 'treated'})
    return data


def create_treatment_groups(data, control_vars):
    df_treat = data[data['treated'] == 1].groupby('id')['time'].min().
reset_index().rename(columns={'time': 'treat_time'})
    # 合并处理时间到原始数据
    data = pd.merge(data, df_treat, on='id', how='left')

    # 为未处理的情况创建一个特殊的期间（比如最大日期后一个月）
    max_period = data['time'].max()
    untreated_period = max_period + pd.offsets.MonthEnd(1)
    min_period = data['time'].min()

    # 填充未处理的情况
    data['treat_time'] = data['treat_time'].fillna(untreated_period)

    # 创建 two_by_twos
    if not control_vars:
        unique_periods = sorted(data['treat_time'].unique())
        two_by_twos = []
        for treated, untreated in product(unique_periods, repeat=2):
            if treated != untreated and treated != untreated_period and
treated != min_period:
                two_by_twos.append({
                    'treated': treated,
                    'untreated': untreated,
                    'estimate': np.nan,
                    'weight': np.nan,
                    'type': (
                        "Treated vs Untreated" if untreated ==
untreated_period else
                        "Later vs Always Treated" if treated == min_period
or untreated == min_period else
                        "Earlier vs Later Treated" if treated < untreated
else
                        "Later vs Earlier Treated"
```

```
                    )
                })
        two_by_twos = pd.DataFrame(two_by_twos)
    else:
        two_by_twos = []
        for treated in data['treat_time'].unique():
            if treated != untreated_period:
                for untreated in data['treat_time'].unique():
                    if untreated > treated:
                        two_by_twos.append({
                            'treated': treated,
                            'untreated': untreated,
                            'estimate': np.nan,
                            'weight': np.nan,
                            'type': (
                                "Treated vs Untreated" if untreated ==
untreated_period else
                                "Later vs Always Treated" if treated ==
min_period or untreated == min_period else
                                "Both Treated"
                            )
                        })
        two_by_twos = pd.DataFrame(two_by_twos)

    return {"two_by_twos": two_by_twos, "data": data}


def subset_data(data, treated_group, untreated_group):
    data = data[data['treat_time'].isin([treated_group, untreated_group])]
    if treated_group < untreated_group:
        data = data[data['time'] < untreated_group]
    else:
        data = data[data['time'] >= untreated_group]
    return data

def calculate_weights(data, treated_group, untreated_group):
    # 计算 D_tilde
    N = len(data['id'].unique())
    T = len(data['time'].unique())
    D_bar = data['treated'].mean()

    data['D_it'] = data['treated']
    data['D_i'] = data.groupby('id')['D_it'].transform('mean')
    data['D_t'] = data.groupby('time')['D_it'].transform('mean')
    data['D_tilde_it'] = (data['D_it'] - data['D_i']) - (data['D_t'] - D_bar
)

    # 计算总方差 V^D
    D_tilde_sum = np.sum(data['D_tilde_it'] ** 2)
    total_variance = (1 / (N * T)) * D_tilde_sum
```

```
    untreated_period = data['time'].max() + pd.DateOffset(months=1)

    if untreated_group == untreated_period:
        n_u = (data['treat_time'] == untreated_group).sum()
        n_j = (data['treat_time'] == treated_group).sum()
        n_jU = n_j / (n_j + n_u)
        D_j = data.loc[data['treat_time'] == treated_group, 'treated'].mean
()
        V_jU = n_jU * (1 - n_jU) * D_j * (1 - D_j)
        weight = ((n_j + n_u) ** 2 * V_jU) / total_variance

    elif treated_group < untreated_group:
        n_e = (data['treat_time'] == treated_group).sum()
        n_l = (data['treat_time'] == untreated_group).sum()
        n_el = n_e / (n_e + n_l)
        D_e = data.loc[data['treat_time'] == treated_group, 'treated'].mean
()
        D_l = data.loc[data['treat_time'] == untreated_group, 'treated'].
mean()
        V_el = n_el * (1 - n_el) * (D_e - D_l) / (1 - D_l) * (1 - D_e) / (1
- D_l)
        weight = (((n_e + n_l) * (1 - D_l)) ** 2 * V_el) / total_variance

    else:
        n_e = (data['treat_time'] == untreated_group).sum()
        n_l = (data['treat_time'] == treated_group).sum()
        n_el = n_e / (n_e + n_l)
        D_e = data.loc[data['treat_time'] == untreated_group, 'treated'].
mean()
        D_l = data.loc[data['treat_time'] == treated_group, 'treated'].mean
()
        V_le = n_el * (1 - n_el) * (D_l / D_e) * (D_e - D_l) / D_e
        weight = (((n_e + n_l) * D_e) ** 2 * V_le) / total_variance

    return weight
```

Last executed in 0 s, finished in 22:03:38 2025-06-30

In [14]:

```
def scale_weights(two_by_twos):
    sum_weight = two_by_twos['weight'].sum()
    two_by_twos['weight'] /= sum_weight
    return two_by_twos


def calculate_beta_hat_w(data):
    Vd_w = np.var(data['d_ikt_til'], ddof=0)
    if Vd_w == 0:
        return np.nan
    C = np.cov(data['outcome'], data['d_ikt_til'], ddof=0)[0, 1]
    beta_hat_w = C / Vd_w
    return beta_hat_w

def calculate_Omega(data):
    Vd_w = np.var(data['d_ikt_til'], ddof=0)
    V_d = np.var(data['d_it_til'], ddof=0)
    if V_d == 0:
        return np.nan
    Omega = Vd_w / V_d
    return Omega

def collapse_x_p(data, control_vars):
    control_data = data[['treat_time', 'time'] + control_vars]
    control_data = control_data.apply(pd.to_numeric, errors='coerce')
    control_data['time'] = control_data['time'].astype(int)
    control_data['treat_time'] = control_data['treat_time'].astype(int)

    # 去除重复的列名
    control_data = control_data.loc[:, ~control_data.columns.duplicated()]

    # 确保只对控制变量进行分组转换
    group_control_data = control_data.groupby(['treat_time', 'time'])[
control_vars].transform('mean')
    group_control_data.columns = ['g_' + col for col in control_vars]

    # 将转换后的控制变量与原数据合并
    data = pd.concat([data.reset_index(drop=True), group_control_data.
reset_index(drop=True)], axis=1)
    data['g_p'] = data.groupby(['treat_time', 'time'])['p'].transform('mean'
)

    g_control_formula = " + ".join(['g_' + col for col in control_vars])
    return {'data': data, 'g_control_formula': g_control_formula}

def calc_VD(data):
    model = sm.OLS(data['treated'], data[['id', 'time']]).fit()
```

```
    data = data.copy()  # 创建数据副本，避免SettingWithCopyWarning
    data.loc[:, 'Dtilde'] = model.resid
    VD = np.var(data['Dtilde'], ddof=0)
    return {'data': data, 'VD': VD}

def partial_group_x(data, g_control_formula):
    g_vars = g_control_formula.split(" + ")
    data = data.copy()  # 创建数据副本，避免SettingWithCopyWarning
    for v in g_vars:
        model = sm.OLS(data[v], data[['id', 'time']]).fit()
        data.loc[:, 'p_' + v] = model.resid
    return data

def calc_pgjtile(data, g_control_formula):
    g_vars = g_control_formula.split(" + ")
    p_g_vars = ['p_' + v for v in g_vars]
    formula = "Dtilde ~ " + " + ".join(p_g_vars)
    model = sm.OLS.from_formula(formula, data).fit()
    data = data.copy()  # 创建数据副本，避免SettingWithCopyWarning
    data.loc[:, 'pgjtilde'] = model.predict(data)
    Rsq = model.rsquared
    return {'data': data, 'Rsq': Rsq}

def calc_Vdp(data):
    model = sm.OLS(data['g_p'], data[['id', 'time']]).fit()
    data = data.copy()  # 创建数据副本，避免SettingWithCopyWarning
    data.loc[:, 'ptilde'] = model.resid
    data.loc[:, 'dp'] = data['pgjtilde'] - data['ptilde']
    Vdp = np.var(data['dp'], ddof=0)
    return {'data': data, 'Vdp': Vdp}

def calc_BD(data, g_control_formula):
    formula = "outcome ~ treated + " + g_control_formula + " + id + time"
    model = sm.OLS.from_formula(formula, data).fit()
    BD = model.params['treated']
    return BD

def calc_Bb(data):
    model = sm.OLS(data['outcome'], data[['dp', 'time', 'id']]).fit()
    Bb = model.params['dp']
    return Bb

def calculate_beta_hat_d_bkl(Rsq, VD, BD, Vdp, Bb):
    beta_hat_d_bkl = ((1 - Rsq) * VD * BD + Vdp * Bb) / ((1 - Rsq) * VD +
Vdp)
    return beta_hat_d_bkl

def calculate_s_kl(N, Rsq, VD, Vdp):
    s_kl = N ** 2 * ((1 - Rsq) * VD + Vdp)
```

```
    return s_kl

def calc_controlled_beta_weights(data, g_control_formula):
    r_calc_VD = calc_VD(data)
    VD = r_calc_VD['VD']
    data = r_calc_VD['data']

    data = partial_group_x(data, g_control_formula)

    r_calc_pgjtilde = calc_pgjtile(data, g_control_formula)
    Rsq = r_calc_pgjtilde['Rsq']
    data = r_calc_pgjtilde['data']

    r_calc_Vdp = calc_Vdp(data)
    Vdp = r_calc_Vdp['Vdp']
    data = r_calc_Vdp['data']

    BD = calc_BD(data, g_control_formula)
    Bb = calc_Bb(data)

    N = len(data)

    s_kl = calculate_s_kl(N, Rsq, VD, Vdp)
    beta_hat_d_bkl = calculate_beta_hat_d_bkl(Rsq, VD, BD, Vdp, Bb)

    return {'s_kl': s_kl, 'beta_hat_d_bkl': beta_hat_d_bkl}

def run_fwl(data, control_formula):
    model = sm.OLS.from_formula(control_formula, data).fit()

    data['p'] = model.predict(data)
    data['d'] = model.resid
    data['d_it'] = data['d']
    data['d_i_bar'] = data.groupby('id')['d'].transform('mean')
    data['d_t_bar'] = data.groupby('time')['d_it'].transform('mean')
    data['d_bar_bar'] = data['d_it'].mean()
    data['d_it_til'] = data['d_it'] - data['d_i_bar'] - data['d_t_bar'] +
data['d_bar_bar']

    data['d_kt_bar'] = data.groupby(['treat_time', 'time'])['d_it'].
transform('mean')
    data['d_k_bar'] = data.groupby('treat_time')['d_it'].transform('mean')

    data['d_ikt_til_step1'] = data['d_it'] - data['d_i_bar']
    data['d_ikt_til_step2'] = data['d_kt_bar'] - data['d_k_bar']
    data['d_ikt_til_step2_corrected'] = data.groupby('treat_time')['d_it'].
transform('mean') - data['d_k_bar']
    data['d_ikt_til'] = data['d_ikt_til_step1'] - data[
'd_ikt_til_step2_corrected']
    data['d_kt_til'] = (data['d_kt_bar'] - data['d_k_bar']) - (data[
'd_t_bar'] - data['d_bar_bar'])
```

```
    return data
```

Last executed in 0 s, finished in 22:03:38 2025-06-30

In [15]:

```
def print_summary(two_by_twos):
    summary = two_by_twos.groupby('type').agg({'weight': 'sum', 'estimate':
lambda x: np.average(x, weights=two_by_twos.loc[x.index, 'weight'])})
    ant_print_all(summary[['type','weight','estimate']],model_method =
'value_counts')
```

Last executed in 0 s, finished in 22:03:38 2025-06-30

表I 2 Goodman Bacon分解结果 ¶

In [16]:

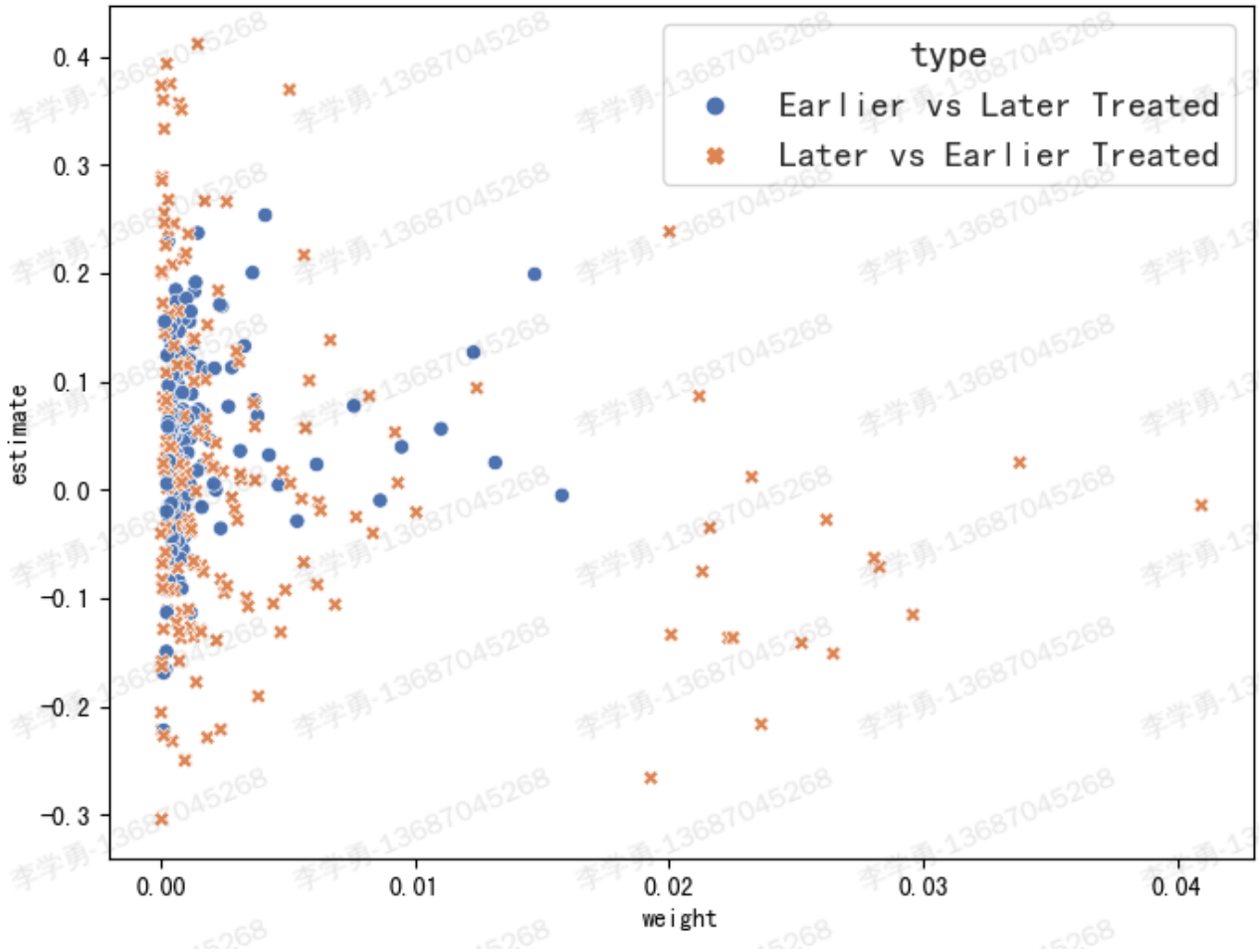
```
ret_bacon = bacon('lnn_module_consumpt ~ new_use', regdata.reset_index(),
id_var='匿名化用户id_rank', time_var='year_month',fixed_effects=['匿名化用户
id_rank','year_month'],cluster="用户id_rank")

# 计算每个类型的加权平均估计值
weighted_estimates = ret_bacon.groupby('type').apply(
    lambda x: np.sum(x['estimate']*x['weight'])
).reset_index(name='weighted_estimate')

# 计算每个类型的总权重
total_weights = ret_bacon.groupby('type')['weight'].sum().reset_index(name=
'total_weight')

# 合并结果
results = pd.merge(weighted_estimates, total_weights, on='type')
results['weighted_estimate'] = results['weighted_estimate']/results[
'total_weight']

ant_print_all(weighted_estimates,model_method = 'value_counts')
ant_print_all(results,model_method = 'describe',des=['dtypes'])
ant_print_all(results[['weighted_estimate','total_weight','type']],
model_method = 'value_counts')
```

```
******************************
type                        weighted_estimate
Earlier vs Later Treated     0.015835            1
Later vs Earlier Treated    -0.024844            1
dtype: int64
******************************

******************************
dtypes
type                    object
weighted_estimate      float64
total_weight           float64
dtype: object
******************************

******************************
weighted_estimate  total_weight  type
-0.033267          0.746804      Later vs Earlier Treated    1
 0.062542          0.253196      Earlier vs Later Treated    1
dtype: int64
******************************


Last executed in 66 s, finished in 22:04:44 2025-06-30
```

图I 1 Goodman Bacon 分解权重图 ¶

In [17]:

```
# 初始化画布
ant_plot.re_init()
# 添加一个1x1的子图
ant_plot.add_subplot()
# 设置图片大小
ant_plot.figure_set(method="set_size_inches", w=8, h=6)
# 设置字体大小
ant_plot.set_theme(font_scale=1.3)
# 绘制散点图
ant_plot.plot(df=ret_bacon, method="ant_plot_df", kind="scatter",
              x="weight", y="estimate", hue="type", style="type")
# 输出图片
ant_plot.show()
```



Last executed in 2 s, finished in 22:04:46 2025-06-30

图**I 3** 消费板块数量分位数回归（得出系数后用**stata**画图） ¶

In [18]:

```
y_list = ['n_module_consumpt']

regdata['n_module_consumpt_w'] = regdata['n_module_consumpt'] - regdata.
groupby('匿名化用户id_rank')['n_module_consumpt'].transform('mean')

cluster_id = regdata['cluster_id']
quantile = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
models = []
results = []
if regdata.shape[0]>200:
    for i in quantile:
        mod = smf.quantreg('n_module_consumpt_w ~ new_use', regdata)
        res = mod.fit(q= i, cov_type="cluster", cov_kwds={'groups':
cluster_id})
        models.append(mod)
        results.append(res)
    ant_print_all(result_printer, model_method="summary_col",
                      results=results, model_names =['10%','20%','30%',
'40%', '50%', '60%', '70%', '80%', '90%'],
                      regressor_order =  ["new_use"], drop_omitted=True,
                      info_list=["rsquared"], info_dict={"count": lambda x:
x.nobs}, info_order=["rsquared", "count"],float_format='%.6f')
```

```
******************************

==============================================================================================

             10%           20%           30%           40%           50%
60%           70%           80%           90%
------------------------------------------------------------------------------
-----------------------------------------------
new_use  0.272728***  0.181817***  0.090910***  0.136363***  0.090909***
0.090915***  0.090907***  0.000001     0.000000
         (0.031865)    (0.024896)    (0.020067)    (0.019022)    (0.015503)
(0.024073)    (0.024237)    (0.029848)    (0.038032)
rsquared -            -            -            -            -            -
-            -            -
count    60060.000000 60060.000000 60060.000000 60060.000000 60060.000000
60060.000000 60060.000000 60060.000000 60060.000000
==============================================================================================

Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
******************************
```

Last executed in 4 s, finished in 22:04:50 2025-06-30

附录III ¶

图A3 板块平均消费金额占比分布 ¶

系数提取出来后用本地python画图

In [19]:

```
# 定义各消费板块变量的顺序（根据需求调整）
consumption_categories = [
    '日用百货', '餐饮美食', '充值缴费', '服饰鞋包', '文化休闲', '其他', '交通出行',
    '数码电器', '家居家装', '美容美发', '医疗健康', '生活服务', '公共服务', '爱车养车',
    '母婴亲子', '运动户外', '商业服务', '酒店旅游', '教育培训', '宠物', '住房物业'
]

# 对各消费板块数据直接计算均值
df_avg = regdata[consumption_categories].mean().reindex(
consumption_categories).reset_index()
df_avg.columns = ['消费类别', '平均消费金额']

# 使用 ant_print_all 导出结果
ant_print_all(df_avg[['消费类别', '平均消费金额']], model_method='value_counts')
```

```
******************************
消费类别 平均消费金额
交通出行 49.937527    1
文化休闲 205.741080   1
酒店旅游 22.505324    1
运动户外 6.829015     1
美容美发 63.031580    1
生活服务 99.620507    1
爱车养车 47.134126    1
母婴亲子 24.959840    1
服饰鞋包 222.734811   1
日用百货 491.102112   1
数码电器 124.782835   1
住房物业 0.930835     1
教育培训 3.419887     1
家居家装 125.017761   1
宠物   2.196340     1
商业服务 19.844968    1
医疗健康 198.220419   1
其他   145.676269   1
公共服务 2.192778     1
充值缴费 106.707611   1
餐饮美食 609.159194   1
dtype: int64
******************************


Last executed in 0 s, finished in 22:04:50 2025-06-30
```

附录图I 2 样本选择偏差匹配的平衡性检验 ¶

In [20]:

```
df = ant_read_data('psm_data',mid_table = True)

## 时间变量
df['报告期1'] = df['报告期'].astype('str')
df['报告期1'] = pd.to_datetime(df['报告期1'])
df['year_month'] = df['报告期1'].apply(lambda x: x.strftime("%Y-%m"))
df['type'] = df['period_length'].astype('category')

# 固定效应相关变量
df['year_month_cat'] = df['year_month'].astype('category')
df['citycode_cat'] = df['citycode'].astype('category')
df['time_city'] = df.apply(lambda row: '{}_{}'.format(row['year_month_cat'],
row['citycode_cat']), axis=1).astype('category')
df['time_type'] = df.apply(lambda row: '{}_{}'.format(row['year_month_cat'],
row['type']), axis=1).astype('category')
df['用户id_rank']  = df['匿名化用户id_rank'].astype('category')

# 聚类
df['cluster_id'] = df['匿名化用户id_rank'].astype('category')
```

Last executed in 1316 s, finished in 22:26:46 2025-06-30

In [21]:

```
df_t= df[df['是否视障用户'] == 1]
df_c = df[df['是否视障用户'] == 0]

#保留视障开始记录当月没有使用的个体
df_t['tag'] = 0
df_t.loc[((df_t['报告期']== 20200331) & (df_t['累计pv'] == 0)),'tag'] = 1
df_temp = df_t[df_t['tag']== 1 ]
df_temp = df_temp['匿名化用户id_rank'].unique()
df_t = df_t[df_t['匿名化用户id_rank'].isin(df_temp)]

df_t = df_t[(df_t['period_length'] >= 3)]
df = pd.concat([df_t,df_c],ignore_index=True)

# 过渡 便于之后再用
df_original = df
```

Last executed in 8 s, finished in 22:26:53 2025-06-30

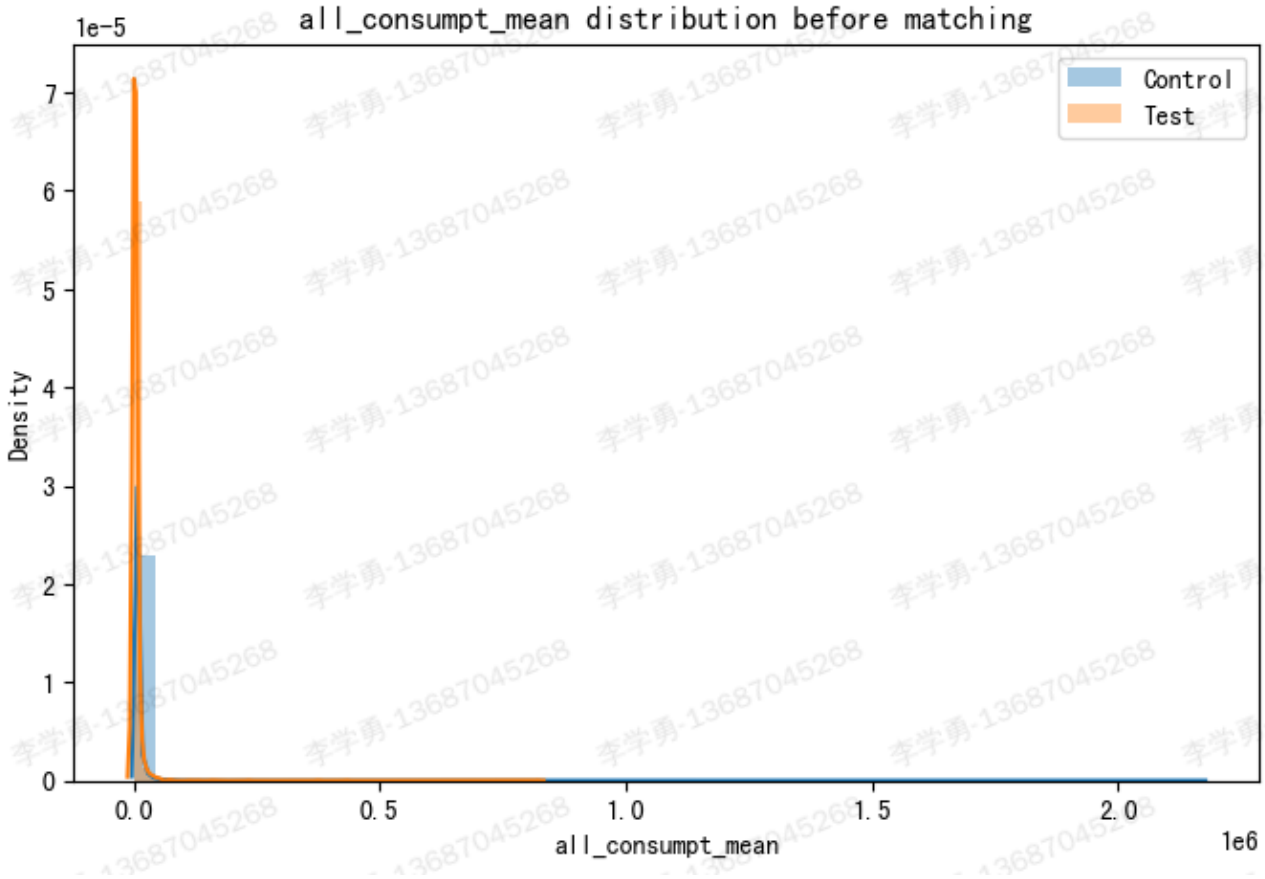
In [22]:

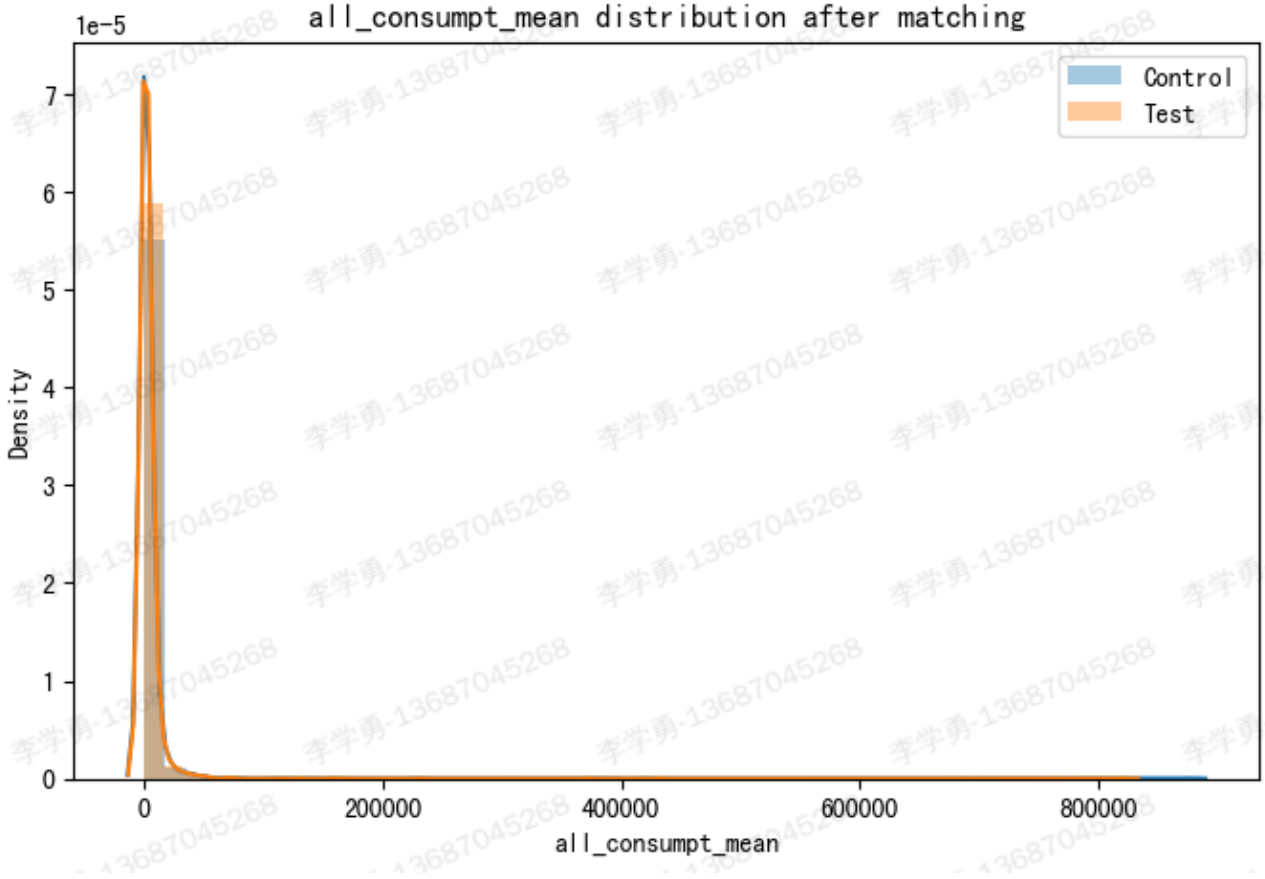
```
# 根据事前的消费数据选出特征相似的个体
df = df[df['报告期'] < 20200331]
## 计算事前的消费特征，用均值匹配
df['all_consumpt_mean'] = df.groupby('匿名化用户id_rank')['all_consumpt'].
transform('mean')
df['num_consumpt_mean'] = df.groupby('匿名化用户id_rank')['num_consumpt'].
transform('mean')
df['n_module_consumpt_mean'] = df.groupby('匿名化用户id_rank')[
'n_module_consumpt'].transform('mean')


xvar = ["all_consumpt_mean", "num_consumpt_mean", "n_module_consumpt_mean"]
test = df[df['是否视障用户'] == 1]
test = test[["匿名化用户id_rank","all_consumpt_mean", "num_consumpt_mean",
"n_module_consumpt_mean","是否视障用户"]]
test = test.drop_duplicates(subset="匿名化用户id_rank")

control = df[df['是否视障用户'] == 0]
control = control[["匿名化用户id_rank","all_consumpt_mean",
"num_consumpt_mean", "n_module_consumpt_mean","是否视障用户"]]
control = control.drop_duplicates(subset="匿名化用户id_rank")


# 初始化模型并计算倾向分
model = PSM(test=test, control=control, xvar=xvar, yvar="是否视障用户")
# 支持probit/logit
model.calculate_pscore(covariates=xvar, method="probit")
# 打印probit的模型拟合结果
ant_print_all(model.pscore_model, model_method="summary")

model.match(nmatches=1, replacement=False, radius=None, measure='covariates'
, scale=True, seed=0)
```

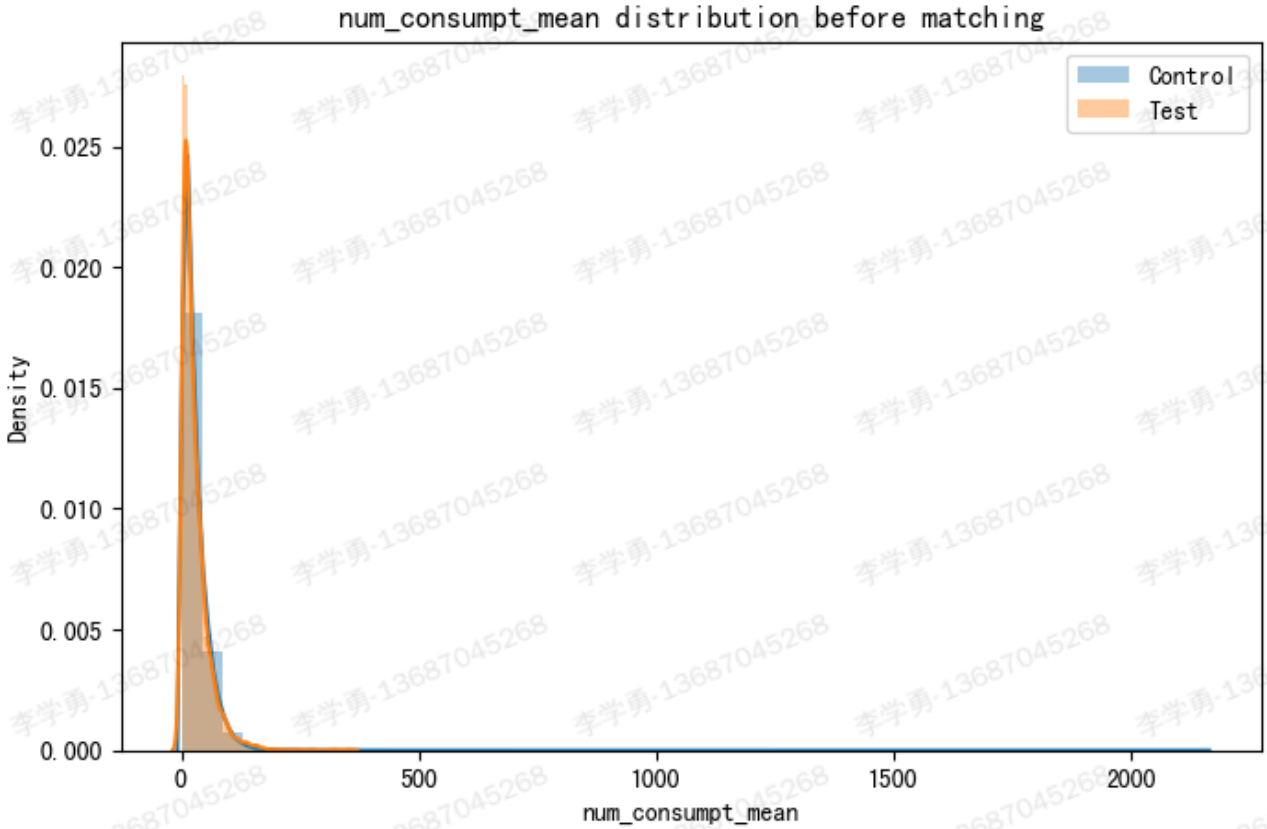
```
******************************
                          Probit Regression Results

==============================================================================

Dep. Variable:                      y   No. Observations:
148545
Model:                         Probit   Df Residuals:
148541
Method:                           MLE   Df Model:
3
Date:                Mon, 30 Jun 2025   Pseudo R-squ.:
0.01041
Time:                        14:26:59   Log-Likelihood:                -
13474.
converged:                       True   LL-Null:                       -
13615.
Covariance Type:            nonrobust   LLR p-value:                 3.762e-
61
==============================================================================================

                               coef    std err          z      P>|z|
[0.025      0.975]
----------------------------------------------------------------------------------
--------------
all_consumpt_mean        1.781e-07   3.69e-07      0.482      0.630   -5.46e-
07    9.02e-07
num_consumpt_mean           0.0019      0.000      8.250      0.000
0.001       0.002
n_module_consumpt_mean     -0.0611      0.004    -16.954      0.000      -
0.068      -0.054
const                      -1.8360      0.017   -106.576      0.000      -
1.870      -1.802
==============================================================================================

******************************


Last executed in 15 s, finished in 22:27:09 2025-06-30
```

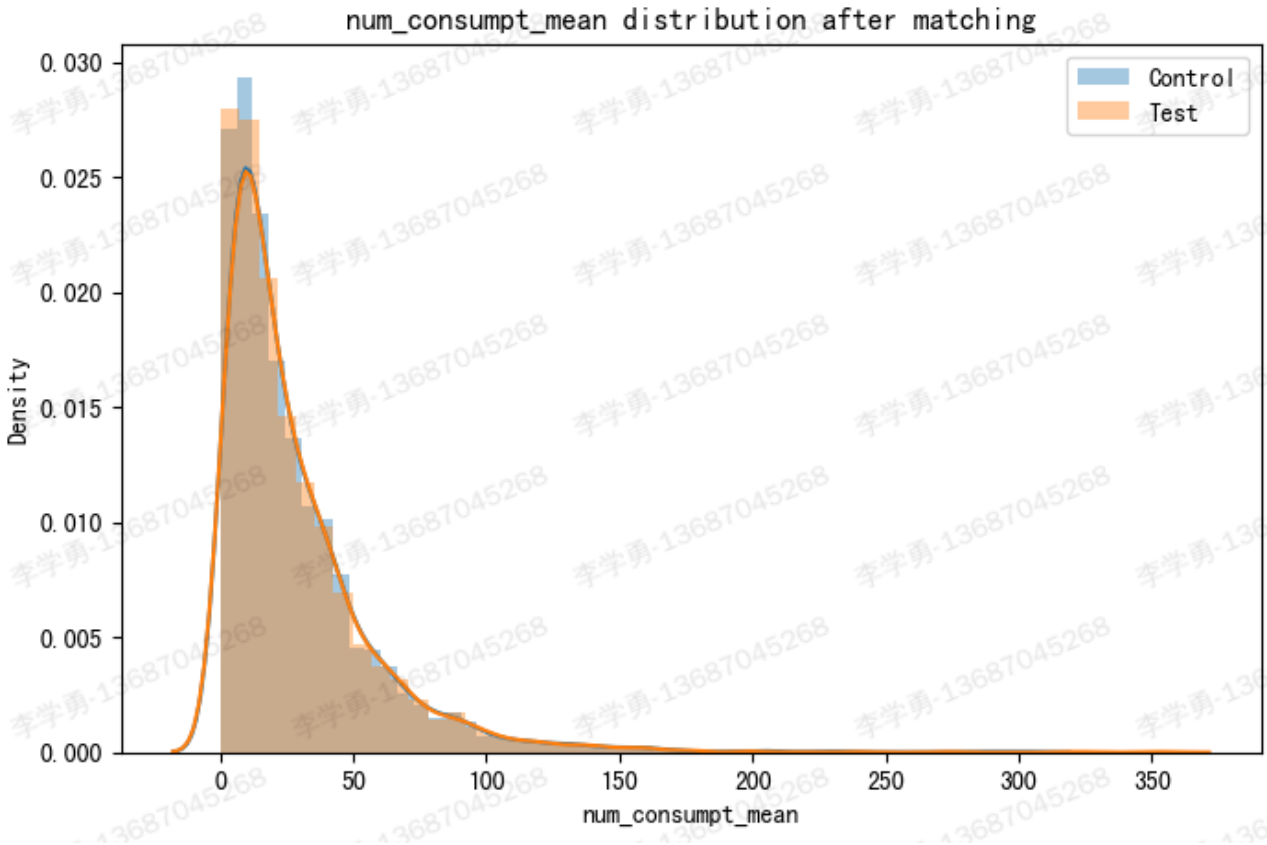
In [23]:

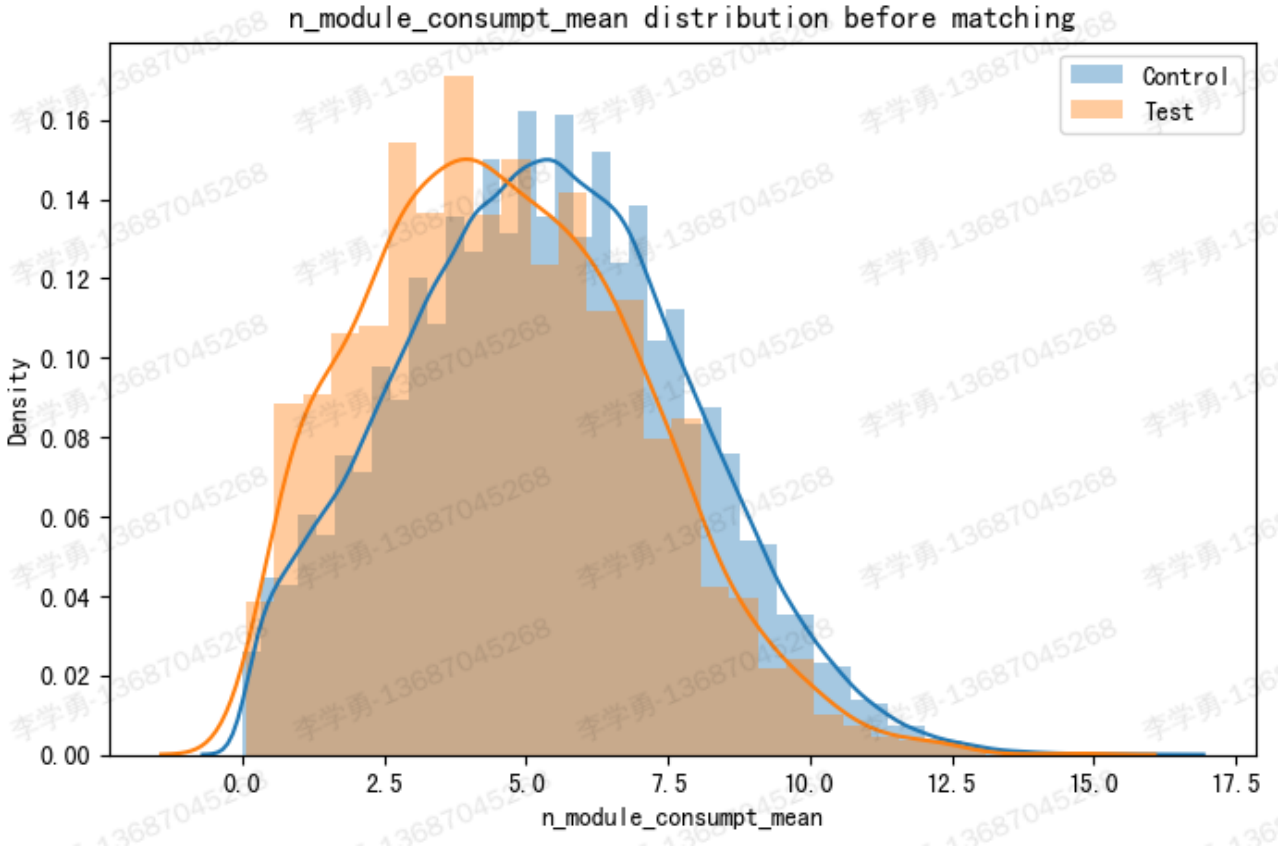
```
for x in xvar:
    model.plot_balance(covariates=x, stage="before")
    model.plot_balance(covariates=x, stage="after")
    # 匹配前后数据做t假设检验(也支持输入“rank”做秩和检验)
model.match_quality(covariates=xvar, test_method="t", stage="before")
ant_print_all(model, model_attr="quality")
model.match_quality(covariates=xvar, test_method="t", stage="after")
ant_print_all(model, model_attr="quality")
```

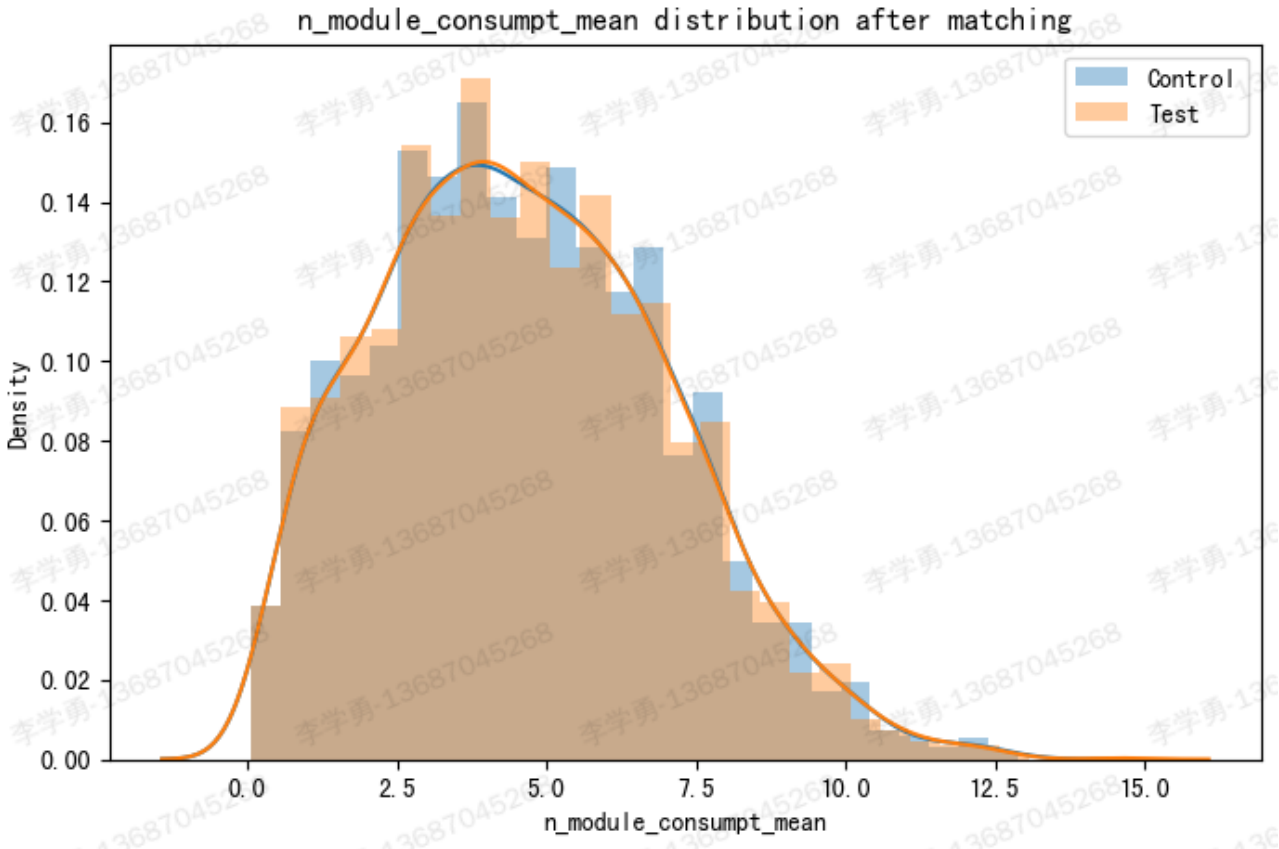

all_consumpt_mean distribution before matching
1e-5
Control
Test
Density
0
1
2
3
4
5
6
7
0.0
0.5
1.0
1.5
2.0
1e6
all_consumpt_mean


all_consumpt_mean distribution after matching
1e-5
Control
Test
Density
0
1
2
3
4
5
6
7
0
200000
400000
600000
800000
all_consumpt_mean


num_consumpt_mean distribution before matching
Control
Test
Density
0.025
0.020
0.015
0.010
0.005
0.000
0
500
1000
1500
2000
num_consumpt_mean


num_consumpt_mean distribution after matching
Control
Test
Density
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
50
100
150
200
250
300
350
num_consumpt_mean


n_module_consumpt_mean distribution before matching
Control
Test
Density
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
n_module_consumpt_mean


n_module_consumpt_mean distribution after matching
Control
Test
Density
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
n_module_consumpt_mean

```
******************************
                              p-value
all_consumpt_mean        2.739708e-01
num_consumpt_mean        5.349503e-02
n_module_consumpt_mean   7.337278e-49
******************************

******************************
                               p-value
all_consumpt_mean         0.980238
num_consumpt_mean         0.956009
n_module_consumpt_mean    0.989473
******************************
```

Last executed in 10 s, finished in 22:27:19 2025-06-30

图3 助盲功能的长期影响 ¶

In [24]:

```
regdata = ant_read_data('quarter_work',mid_table = True)

# 将 float64 日期转换为 datetime
regdata['日期'] = pd.to_datetime(regdata['日期'].astype('int').astype('str'))
regdata['year_quarter'] = regdata['日期'].dt.to_period('Q')
regdata = regdata.rename(columns={"lnn_modules": "LnN_modules"})
```

Last executed in 0 s, finished in 22:27:19 2025-06-30

In [25]:

```
def clustered_se(model, cluster_var):
    return np.sqrt(np.diag(cov_cluster(model, cluster_var)))

def calculate_dynamic_effects(data, treatment_col, time_col, outcome_col,
unit_col, rel_time_col,
                              min_rel_time=-5, max_rel_time=5, fixed_effects
=None, controls=None):
    """
  计算动态处理效应

  参数:
    data: DataFrame, 包含所有必要变量的面板数据
    treatment_col: str, 处理变量的列名
    time_col: str, 时间变量的列名
    outcome_col: str, 结果变量的列名
    unit_col: str, 单位ID变量的列名
    rel_time_col: str, 相对时间变量的列名
    min_rel_time: int, 最小相对时间 (默认 -5)
    max_rel_time: int, 最大相对时间 (默认 5)
    fixed_effects: list, 固定效应的列名列表

  返回:
    DataFrame, 包含每个相对时间点的处理效应估计和标准误
    """
    dynamic_effects = []

    # 数据预处理
    data = data[(data[rel_time_col] != -99) &
                (data[rel_time_col] >= min_rel_time) &
                (data[rel_time_col] <= max_rel_time)].copy()

    # 准备固定效应
    if fixed_effects:
        for fe in fixed_effects:
            data[fe] = data[fe].astype('category')

    # 构建公式
    formula = f"{outcome_col} ~ "
    if controls:
        formula += " + ".join(controls)
    if fixed_effects:
        if controls:
            formula += " + "
        formula += " + ".join([f"C({fe})" for fe in fixed_effects])

    for t in range(min_rel_time, max_rel_time + 1):
```

```
        current_data = data[data[rel_time_col] == t]
        control = data[data[treatment_col] == 0]

        # 估计控制组模型
        model = sm.formula.ols(formula, data=control)
        result = model.fit()

        # 预测处理组的反事实结果
        current_data['y_hat'] = result.predict(current_data)
        current_data['effect'] = current_data[outcome_col] - current_data[
'y_hat']

        att = current_data['effect'].mean()

        # 计算聚类标准误
        model_se = sm.OLS(current_data['effect'],
                          sm.add_constant(current_data[controls] if controls
else pd.DataFrame(index=current_data.index)))
        results_se = model_se.fit()
        se_att = clustered_se(results_se, current_data[unit_col])[0]

        dynamic_effects.append({'rel_time': t, 'coefficient': att,
'std_error': se_att})
    return pd.DataFrame(dynamic_effects)
```

```
Last executed in 0 s, finished in 22:27:19 2025-06-30
```

In [26]:

```
if regdata.shape[0] > 2000:
    # 计算动态处理效应
    dynamic_effects = calculate_dynamic_effects(
        data=regdata,
        treatment_col='new_use',
        time_col='year_quarter',
        outcome_col='LnN_modules',
        unit_col='匿名化用户id_rank',
        rel_time_col='relative_quarter',
        min_rel_time=-6,
        max_rel_time=8,
        fixed_effects=['匿名化用户id_rank', 'year_quarter']
    )


    ant_print_all(dynamic_effects[['rel_time','coefficient','std_error']],
model_method = 'value_counts')
```

```
******************************
rel_time  coefficient  std_error
-6          0.025194    0.040017     1
-5         -0.020789    0.020235     1
-4          0.017294    0.013726     1
-3         -0.009582    0.009700     1
-2         -0.001026    0.006947     1
-1          0.001583    0.004313     1
 0          0.052404    0.009446     1
 1          0.093078    0.010951     1
 2          0.101540    0.011835     1
 3          0.105956    0.012437     1
 4          0.199587    0.013238     1
 5          0.215482    0.013846     1
 6          0.451265    0.014013     1
 7          0.412638    0.014785     1
 8          0.401903    0.016185     1
dtype: int64
******************************
```

Last executed in 614 s, finished in 22:37:33 2025-06-30

In [ ]:

Last executed in 0 s, finished in 22:37:33 2025-06-30