2021年度数据分析报告 | 杰然不同之GR

本报告由公众号【杰然不同之GR】编写，整理并发布，仅作个人学习使用，请勿用于任何商业用途，转载及其他形式合作请与我联系

简易目录

2.4.4) 每天做的事项数目的统计

2.4.5) 每天做事项先后编号并统计

2.4.6) 各事项关于日期求和的透视表

2.4.7) 每年, 月, 星期的事项关键词词云图

2.4.8) 每天事项的对应先后以及月度关联的网络图

2.5 一键式EDA

3.数据分析

3.1 方差分析每月或是每年的事项是否有差异

3.1.1) 原始事项时长数据的方差分析

3.1.2) 衍生数据的方差分析

3.2 LSTM预测前后事项

3.2.1) 建立字符索引

3.2.2) 建立LSTM模型

3.2.3) 模型训练

3.3 时长数据的简单聚类

3.4 关于日期的缺失值处理

3.5 关于日期的傅里叶分解求周期

3.6 关于日期的传统时间序列分析

3.6.1) ARIMA分析

3.6.2) GARCH分析

3.6.3) VAR分析

3.7 关于日期的NeuralProphet 时间序列

4.数据挖掘

4.1 通过生成聚合特征做日度时长数据的特征工程

4.2 类别变量的one-hot encoding

4.3 利用boruta筛选特征

4.4 利用PCA降维

4.5 利用optuna优化参数

4.6 stacking 融合

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import jieba
import statsmodels.tsa.stattools as st
import warnings
```

```
import lightgbm as lgb
import xgboost as xgb
import networkx as nx
import pycatflow as pcf
import statsmodels.api as sm
import statsmodels
import optuna
import pylab as pl
import matplotlib.cm as cm
import torch
import random
from scipy.cluster.hierarchy import linkage, dendrogram
from scipy.stats import kstest, shapiro
from optuna.samplers import TPESampler
from gensim.corpora import Dictionary
from gensim.models import TfidfModel
from wordcloud import WordCloud
from scipy import stats, signal, interpolate
from arch import arch_model
from boruta import BorutaPy
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa import stattools
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima_model import ARMA
from statsmodels.stats.diagnostic import acorr_ljungbox
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor,
GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
from sklearn.decomposition import PCA
from xgboost.sklearn import XGBRegressor,XGBRegressor
from autoviz.AutoViz_Class import AutoViz_Class
from tqdm import tqdm
from neuralprophet import NeuralProphet
from PIL import Image, ImageDraw, ImageFont
from matplotlib import rcParams
from torch import nn
from mpl_toolkits.mplot3d import Axes3D

config = {
    "font.family":'serif',
    "font.size": 40,
    'font.weight':'bold',
    "mathtext.fontset":'stix',
    'figure.titleweight': 'bold',
    'axes.titleweight': 'bold',
    'axes.labelweight': 'bold',
    "font.serif": ['STSong'],
    'axes.unicode_minus':False
}
rcParams.update(config)
path = 'D:/打卡/2021/年度总结'
warnings.filterwarnings("ignore")
%matplotlib inline
```

1.数据预处理

1.1 数据合并和一些纠错

```
df = pd.read_excel('2021年1月打卡记录.xlsx')
dfphone = pd.read_excel('1月.xlsx', sheet_name=1).rename(
    columns={'手机时间':'phone'}
)
df = pd.merge(
    df, dfphone[['date', 'phone']], on='date', how='outer'
)

# 一月的date略作处理
df['date'] = df['date'].astype(str).apply(lambda x:x.split()[0])

# 循环读入每天的数据
for month in [f'0{n}' for n in range(2, 10)] + ['10', '11', '12']:
    for day in [f'0{n}' for n in range(1, 10)] + [str(n) for n in range(10,
32)]:
        file = f'2021-{month}-{day}打卡记录.xlsx'
        phone = f'2021-{month}-{day}打卡总结.xlsx'
        try:
            temp = pd.read_excel(file).iloc[:, 1:]
            temp['phone'] = pd.read_excel(phone).loc[0, '手机时间']
            df = pd.concat([df, temp], axis=0)
        except:
            continue
```

```
# 一些纠错
corrects = {
    'quant':'量化',
    'quant2':'量化',
    '人文书记':'人文书籍',
    '数据竞赛':'数据分析',
    '正则表达式':'正则'
}

df['event'] = df['event'].apply(lambda x: corrects.get(x) if corrects.get(x) is
not None else x)
df.to_excel('2021汇总.xlsx', index=False)
```

1.2 新特征生成

```
df = pd.concat(
    [
        pd.read_excel(f'{n}汇总.xlsx')['date  event   start   end duration
attr    phone'.split()]
        for n in [2019, 2020, 2021]
    ], axis=0
)
```

1.2.1 根据date生成月日还有星期

```
# 1.2.1 根据date生成月日还有星期
df['event'] = df['event'].str.strip()
df['year'] = df['date'].astype(str).apply(lambda x:str(x.split('-')[0]))
df['month'] = df['date'].astype(str).apply(lambda x:str(x.split('-')[1]))
df['day'] = df['date'].astype(str).apply(lambda x:str(x.split('-')[2]))
df['weekday'] = pd.to_datetime(df['date']).dt.weekday + 1
```

1.2.2 根据start time和end time生成时和分

```
df['start_hour'] = df['start'].astype(str).apply(lambda x:str(x.split(':')[0]))
df['end_hour'] = df['end'].astype(str).apply(lambda x:str(x.split(':')[0]))
```

1.2.3 根据start和end生成中间时间

```
# 1.2.3 根据start和end生成中间时间
def get_mid(start, duration):
    times = str(start).split(':')
    minutes = int(times[0]) * 60 + int(times[1])
    res = minutes + duration // 2
    hour = res // 60
    minute = res - hour * 60
    return f'{int(hour)}:{int(minute)}' if minute >= 10 else
f'{int(hour)}:0{int(minute)}'


df['mid_time'] = df.apply(lambda x: get_mid(x['start'], x['duration']), axis=1)
```

1.2.4 根据事项的中间时间确定事项做的时候处于一天的时段

```
'''
一天的时间段划分为:
8-12点: morning
12-13点: noon
13-19点: afternoon
19点到24点: evening
0-8点: night
'''
def get_period(hour):
    hour = int(hour)
    if hour in range(8, 12):
        return 'morning'
    elif hour == 12:
        return 'noon'
    elif hour in range(13, 19):
        return 'afternoon'
    elif hour in range(19, 24):
        return 'evening'
    return 'night'


# 1.2.4 根据事项的中间时间确定事项做的时候处于一天的时段
df['mid_hour'] = df['mid_time'].astype(str).apply(lambda x:str(x.split(':')[0]))
```

```
df['day_period'] = df['mid_hour'].map(get_period)
df['week_order'] = df['date'].map(lambda x: pd.Period(x, freq='W'))
```

1.2.5 根据时长大小分为长中短

```
# 1.2.5 根据时长大小分为长中短
'''
30分钟以下为short
30-60分钟为middle
60分钟以上为long
'''

def get_duration(duration):
    if int(duration) < 30:
        return 'short'
    elif int(duration) <= 60 and int(duration) >= 30:
        return 'middle'
    return 'long'

df['duration_attr'] = df['duration'].map(get_duration)
df.head()
```

1.2.6 修改一些前后差别不大的事件

```
# 1.2.6 修改一些前后差别不大的事件
def correct_events(e):
    alist = {
        '代码':'改代码',
        '拍照':'摄影',
        '简历':'简历',
        '总结':'写报告',
        '报告':'写报告',
        '面试': '面试',
        '博弈论':'博弈论',
        '磨蹭':'磨蹭',
        'PPT':'PPT',
        '快递':'快递',
        'TPO':'托福听力',
        '算法':'算法课程',
        '推文':'写推文',
        '演唱会':'演唱会',
        '人文课':'人文课',
        'AQF':'AQF',
    }
    for k in alist:
        if k in e:
            e = alist[k]
    return e

df['event'] = df['event'].map(correct_events)
```

1.2.7 事项与属性的对应

```
# 1.2.7 事项与属性的对应
# 各别事项前后不统一 需要统一修正 修正成该事项出现最多的属性
events = df['event'].unique()
event2attr = {}
for e in events:
    attr = df[df['event'] == e]['attr'].value_counts().index[0]
    event2attr[e] = attr

df['attr'] = df['event'].map(event2attr)
df.to_excel('3年汇总.xlsx', index=False)
```

```
# 检查是否还存在事项的属性不唯一的情况
df[['event', 'attr']].drop_duplicates().to_excel('事项性质对应表.xlsx',
index=False)
df[['event', 'attr']].drop_duplicates()['event'].value_counts()
```

2.数据可视化

2.1 变量分类

```
date_vars = ['date']
years = range(2019, 2022)

numeric_vars = [
    'duration', 'phone'
]

cate_vars = [
    'event', 'year', 'month', 'day',
    'weekday', 'mid_hour', 'day_period', 'week_order', 'duration_attr', 'attr'
]
```

2.2 单变量分析

2.2.1) 数值变量的直方图

```
# 2.2.1) 数值变量的直方图
part = '2.2.1'
def numeric_hist(col, year):
    temp = df[df['year'] == str(year)].copy()
    if col == 'phone':
        temp = temp[[col, 'date']].drop_duplicates()
    plt.figure(figsize=(20, 10), dpi=100)
    plt.hist(temp[col], bins=20)
    plt.xlabel(col, fontsize=20)
    plt.ylabel('Frequency', fontsize=20)
    title = f'{part} Histgram of {col} in year {year}'
    plt.title(title, fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

for col in numeric_vars:
    for year in years:
        numeric_hist(col, year)
```

2.2.2) 类别变量的饼图

```
# 2.2.2) 类别变量的饼图
# 类别变量饼图
part = '2.2.2'
def plot_pie(col, data, n, year=None):
    fig = plt.figure(figsize=(20, 10), dpi=100)
    fig.tight_layout()
    ax = fig.add_subplot(111)
    df = data.copy()
    if year is not None:
        try:
```

```
            temp = df[df['year'] == str(year)].copy()
        except:
            temp = df.copy()
    else:
        temp = df.copy()
    data = temp[col].value_counts().sort_values(ascending=False).values[:n]
    labels = temp[col].value_counts().sort_values(ascending=False).index[:n]
    explodes = [0] * len(data)
    explodes[0] = 0.015
    ax.pie(
        data,
        labels=labels,
        radius=0.8,
        explode=explodes,
        autopct='%1.1f%%',
        pctdistance = 0.5,
        labeldistance=0.7,
        textprops={'fontsize': 25, 'color': 'black'}
    )
    plt.axis('equal')
    title = f'{part} Pie plot of {col} in year {year}' if year is not None else
\
        f'{part} Pie plot of {col}'
    plt.title(title, fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)


for col in cate_vars:
    if col == 'year':
        continue
    for year in years:
        plot_pie(col, df, 15, year)
```

2.2.3) 专注力分析

```
# 2.2.3) 专注力分析
part = '2.2.3'
def concentrate(year):
    temp = df[
        (df['year'] == str(year)) & (df['attr'] == '有效时间')
    ].copy()

    # Figure 1
    plt.figure(dpi=100)
    df[df['year'] == str(year)].groupby(['attr', 'duration_attr'])['date'].\
    count().unstack().plot(kind='bar', figsize=(20, 10))
    title = f'{part} Duration attribution of each time in year {year}'
    plt.title(title, fontsize=20)
    plt.xlabel('Effective time duration', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.legend(fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

    # Figure 2
```

```
    plt.figure(dpi=100)
    temp['duration'].plot(kind='hist', figsize=(20, 10))
    title = f'{part} Duration of effective time in year {year}'
    plt.title(title, fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.ylabel('Frequency', fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

    # Figure 3
    plt.figure(dpi=100)
    temp['duration_attr'].value_counts().plot(kind='bar', figsize=(20, 10))
    title = f'{part} Bar plot of duration attribution of effective time in year
{year}'
    plt.title(title, fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)
```

```
title = '有效时间在不同年和时长段的柱形图'
temp = df[df['attr'] == '有效时间'].groupby(['year', 'duration_attr'])['date'].\
    count().unstack()
temp = temp / temp.sum(axis=0)  #计算频率
temp.plot(
        kind='bar',
        figsize=(20, 10),
    )
plt.title(title, fontsize=20)
plt.xlabel('Year', fontsize=20)
plt.ylabel('Frequency', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(fontsize=20)
file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
concentrate(2021)
```

```
concentrate(2020)
```

```
concentrate(2019)
```

2.3 双变量分析

2.3.1) PyCatFlow图

```
# 2.3.1) PyCatFlow图
column_order = {
    date: i + 1 for i, date in enumerate(df['date'].unique())
}
df['column_order'] = df['date'].map(column_order)
translate = pd.read_csv('event.csv', encoding='gbk')
chi = translate['Chinese'].values
eng = translate['English'].dropna().values
translates = {
    c:e for c, e in zip(chi, eng)
}

translates
```

```
df['eng_event'] = df['event'].map(translates)

df['eng_attr'] = df['attr'].map({
    '有效时间':'effective',
    '浪费时间':'wasteful',
    '必要时间':'necessary',
})

temp = df[df['year'] == '2021']
temp[['eng_event', 'eng_attr', 'date', 'column_order']].\
to_csv('2021.csv', encoding='gbk', index=False)
```

```
# Loading and parsing data:
data = pcf.read_file(
    "2021.csv",
    columns="date",
    nodes="eng_event",
    categories="eng_attr",
    column_order="column_order"
)

# Generating the visualization
viz = pcf.visualize(
    data,
    spacing=100,
    width=88000,
    maxValue=20,
    minValue=2
)
viz
```

2.3.2) 单个类别变量 vs 数值

```
# 2.3.2) 单个类别变量 vs 数值
# 类别变量 VS 数值变量
part = '2.3.2'
def plot_num_vs_cate(num, cate, n=30, year=2021):
    if cate != 'year':
        stats = ['max', 'min', 'mean', 'std', 'median', 'sum']
        for sts in stats:
            temp = df[df['year'] == str(year)].groupby(cate)[num].\
```

```
            agg(sts).sort_values(ascending=False)
            plt.figure(dpi=100)
            title = f"{part} bar plot of {sts} of {num} in different {cate}
groups in year {year}"
            temp[:n].plot(
                kind='bar',
                figsize=(20, 10),
            )
            plt.title(title, fontsize=20)
            plt.xlabel(cate, fontsize=20)
            plt.ylabel(f'{sts} of {num}', fontsize=20)
            plt.xticks(fontsize=20)
            plt.yticks(fontsize=20)
            file = os.path.join(path, f'{title}.png')
            plt.savefig(file, dpi=600)

for num in numeric_vars:
    for cate in cate_vars:
        plot_num_vs_cate(num, cate)
```

```
def plot_box(num, cate, n=30, year=2021):
    temp = df[df['year'] == str(year)].groupby(cate)[num].agg('median').\
        sort_values(ascending=False)[:n]
    plt.figure(figsize=(20, 10), dpi=100)
    sns.boxplot(
        x=cate,
        y=num,
        data=df[df[cate].isin(temp.index)],
        order=temp.index
    )
    title = f'{part} Boxplot of {num} wrt {cate} in year {year}'
    plt.title(title, fontsize=20)
    plt.xlabel(cate, fontsize=20)
    plt.ylabel(num, fontsize=20)
    plt.xticks(rotation=90, fontsize=20)
    plt.yticks(fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

for num in numeric_vars:
    for cate in cate_vars:
        plot_box(num, cate)
```

2.3.3) 两个类别变量 VS 数值

```
# 2.3.3) 两个类别变量 VS 数值
part = '2.3.3'
def plot_count_plot(c1, c2, n=15, year=2021):
    temp = df[df['year'] == str(year)].groupby(c1)[c2].\
    count().sort_values(ascending=False)[:n]
    plt.figure(figsize=(20, 10), dpi=100)
    sns.countplot(
        x=c1,
        hue=c2,
        data=df[df[c1].isin(temp.index)],
        order=temp.index
```

```
    )
    plt.legend() if c2 != 'week_order' else plt.legend([])
    title = f'{part} Countplot of {c1} in different {c2} group'
    plt.title(title, fontsize=20)
    plt.xlabel(c1, fontsize=20)
    plt.ylabel('Count', fontsize=20)
    plt.xticks(rotation=90, fontsize=20)
    plt.yticks(fontsize=20)
    if c2 not in ['day', 'week_order']:
        plt.legend(fontsize=20, loc=1)
    else:
        plt.legend([])
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

for idx, v in enumerate(cate_vars):
    if idx == len(cate_vars) - 1:
        break
    for k in cate_vars[idx + 1:]:
        plot_count_plot(v, k)
```

2.4 衍生数据可视化

2.4.1) 日度汇总各类时间总时长

```
# 2.4.1) 日度汇总各类时间总时长
part = '2.4.1'
daily_attr = df.groupby(['date', 'attr'])['duration'].\
    agg('sum').unstack().reset_index()

daily_attr = pd.merge(
    daily_attr,
    df[['date', 'phone']].drop_duplicates(),
    on='date'
).rename(columns={'phone':'手机时间'})

daily_attr
```

```
daily_attr.describe()
```

```
daily_attr[daily_attr['date'] >= '2021-01-01'].describe()
```

```
title = f'{part} 日度汇总各类时间总时长'
daily_attr.set_index('date').plot(
    kind='line', figsize=(20, 10)
)

plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Sum of daily duration', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(loc=0, fontsize=20)
```

```
file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
# 生成新数据
daily_attr['weekday'] = pd.to_datetime(daily_attr['date']).dt.weekday + 1
daily_attr['month'] = daily_attr['date'].apply(lambda x:str(x).split('-')[1])
daily_attr['day'] = daily_attr['date'].apply(lambda x:str(x).split('-')[2])
daily_attr['week_order'] = daily_attr['date'].map(lambda x: pd.Period(x,
freq='W'))

daily_attr_copy = daily_attr.copy()
daily_attr_copy = daily_attr_copy.rename(
    columns={
        '必要时间': 'necessary time',
        '有效时间': 'effective time',
        '浪费时间': 'waste time',
        '手机时间': 'phone time',
    }
)
```

```
# 插值
daily_attr_copy = daily_attr_copy.dropna()
daily_attr_copy = daily_attr_copy[daily_attr_copy['date'] >= '2021-01-01']
def daily_interpolate(x, y, z, n=300, step=None):
    xx, yy, zz = x, y, z
    x, y, z = daily_attr_copy[x].values, daily_attr_copy[y].values,
daily_attr_copy[z].values
    upper, lower = 75, 25
    xmin, xmax = np.percentile(x, lower), np.percentile(x, upper)
    ymin, ymax = np.percentile(y, lower), np.percentile(y, upper) # 不直接用最大最
小避免异常值
    newfunc = interpolate.interp2d(x, y, z, kind='cubic')
    if step is not None:
        X, Y = np.arange(xmin, xmax + step, step), np.arange(ymin, ymax + step,
step)
    else:
        X, Y = np.linspace(xmin, xmax, n), np.linspace(ymin, ymax, n),
    Z = newfunc(X, Y)
    plt.figure(figsize=(20, 10))
    im2 = pl.imshow(
        Z,
        extent=[xmin, xmax, ymin, ymax],
        cmap=cm.jet_r,
        interpolation='bilinear',
        origin='lower',
        aspect='auto'
    )
    title = f'{part} Relationship between {zz} VS {xx} and {yy} after projection
to a plane'
    pl.xlabel(xx, fontsize=20)
    pl.ylabel(yy, fontsize=20)
    pl.xticks(fontsize=20)
    pl.yticks(fontsize=20)
    pl.title(title, fontsize=20)
    cb = pl.colorbar(im2)
    cb.ax.tick_params(labelsize=20)
```

```
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

    X, Y = np.meshgrid(X, Y)
    fig = plt.figure(figsize=(20, 10))
    ax = Axes3D(fig)
    surf2 = ax.plot_surface(
        X,
        Y,
        Z,
        rstride=1,
        cstride=1,
        cmap=cm.jet_r,
        linewidth=0.01,
        antialiased=True
    )
    ax.set_xlabel(xx, fontsize=20)
    ax.set_ylabel(yy, fontsize=20)
    ax.set_zlabel(zz, fontsize=20)
    pl.xticks(fontsize=15)
    pl.yticks(fontsize=15)
    ax.tick_params(axis='z',labelsize=15)
    title = f'{part} Relationship between {zz} VS {xx} and {yy}'
    ax.set_title(title, fontsize=20)
    cb = plt.colorbar(surf2, shrink=0.5, aspect=5)
    cb.ax.tick_params(labelsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)
```

```
x, y, z = 'waste time', 'necessary time', 'effective time'
daily_interpolate(x, y, z, n=400)
```

```
x, y, z = 'phone time', 'necessary time', 'effective time'
daily_interpolate(x, y, z)
```

2.4.2) 每天的第一件和最后一件事

```
# 2.4.2) 每天的第一件和最后一件事
part = '2.4.2'
first_last_event = pd.DataFrame()

# 每天做的第一件事
first_last_event['first_event'] = df[['date', 'event']].\
    groupby(['date'])['event'].first()

# 每天做的最后一件事
first_last_event['last_event'] = df[['date', 'event']].\
    groupby(['date'])['event'].last()


first_last_event = first_last_event.reset_index()
first_last_event
```

```
# 整体的饼图
plot_pie('first_event', first_last_event, 15)
```

```
# 2019饼图
plot_pie(
    'first_event',
    first_last_event[first_last_event['date'] < '2020-01-01'],
    10,
    year=2019
)
plot_pie(
    'last_event',
    first_last_event[first_last_event['date'] < '2020-01-01'],
    10,
    year=2019
)
```

```
# 2020饼图
temp = first_last_event[
    (first_last_event['date'] < '2021-01-01') & (first_last_event['date'] >=
'2020-01-01')
]
plot_pie('first_event', temp, 15, year=2020)
plot_pie('last_event', temp, 5, year=2020)
```

```
# 2021饼图
temp = first_last_event[
    (first_last_event['date'] < '2022-01-01') & (first_last_event['date'] >=
'2021-01-01')
]
plot_pie('first_event', temp, 15, year=2021)
plot_pie('last_event', temp, 5, year=2021)
```

2.4.3) 每天的第一件和最后一件事的时间

```
# 2.4.3) 每天的第一件和最后一件事的时间
part = '2.4.3'
first_last_time = pd.merge(
    df.groupby(['date'])['start', 'start_hour'].first(),
    df.groupby(['date'])['end', 'end_hour'].last(),
    on='date'
).reset_index().rename(columns={
    'start':'first_start_time',
    'start_hour':'first_start_hour',
    'end':'last_end_time',
    'end_hour':'last_end_hour'
})

first_last_time
```

```
# 2019饼图
temp = first_last_time[first_last_time['date'] < '2020-01-01']
plot_pie('first_start_hour', temp, 15, year=2019)
plot_pie('last_end_hour', temp, 5, year=2019)
```

```
# 2020饼图
temp = first_last_time[
    (first_last_time['date'] < '2021-01-01') & (first_last_time['date'] >=
'2020-01-01')
]
plot_pie('first_start_hour', temp, 15, year=2020)
plot_pie('last_end_hour', temp, 5, year=2020)
```

```
# 2021饼图
temp = first_last_time[
    (first_last_time['date'] < '2022-01-01') & (first_last_time['date'] >=
'2021-01-01')
]
plot_pie('first_start_hour', temp, 15, year=2021)
plot_pie('last_end_hour', temp, 5, year=2021)
```

2.4.4) 每天做的事项数目的统计

```
# 2.4.4) 每天做的事项数目的统计
part = '2.4.4'
df.groupby('date')['event'].count().\
    reset_index().\
    sort_values(by='event', ascending=False)
```

```
title = f'{part} 每天做的事项数目的统计'
df.groupby('date')['event'].count().plot(
    figsize=(20, 10)
)
plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Count', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
title = f'{part} 每天做的事项数目的统计直方图'
df.groupby('date')['event'].count().plot(
    kind='hist', figsize=(20, 10)
)

plt.title(title, fontsize=20)
plt.xlabel('Count', fontsize=20)
plt.ylabel('Frequency', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

2.4.5) 每天做事项先后编号并统计

```
# 2.4.5) 每天做事项先后编号并统计
part = '2.4.5'
def get_order(alist):
    return range(1, 1 + len(alist))

daily_event_order = df.groupby('date')['event'].agg(get_order)
daily_event_order = pd.concat(
    [daily_event_order.explode().reset_index().rename(columns=
{'event':'order'}),
    df['event'].reset_index()], axis=1
)[['date', 'event', 'order']]
daily_event_order['order'] = daily_event_order['order'].astype(int)
daily_event_order
```

```
# 每个事项平均处于每天的第几件事
event_mean_order = daily_event_order[
    (daily_event_order['date'] < '2022-01-01') &
    (daily_event_order['date'] >= '2021-01-01')
].groupby('event')['order'].mean()

title = f'{part} 各事项平均顺序的统计柱形图'
event_mean_order.sort_values().plot(
    kind='bar', figsize=(20, 10)
)
plt.title(title, fontsize=20)
plt.xlabel('Event', fontsize=20)
plt.ylabel('Mean Order', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
# 每个次序做得最多的事
order_most_event = daily_event_order[
    (daily_event_order['date'] < '2022-01-01') &
    (daily_event_order['date'] >= '2021-01-01')
].groupby(['order', 'event'])['date'].agg('count').reset_index()

order_most_event = order_most_event.rename(
    columns={'date': 'count'}
)
```

```
# 每个顺序做的最多的k件事
k = 1
order_mostk_event = pd.concat(
    [order_most_event[order_most_event['order'] == m].\
    sort_values(by='count', ascending=False)[:k]
    for m in order_most_event['order'].unique()], axis=0
)
order_mostk_event
```

2.4.6) 各事项关于日期的透视表

```
# 2.4.6) 各事项关于日期的透视表
part = '2.4.6'
def get_pivot(df, item, layer, attr, ops):
    tmp = df.groupby([item, layer])
[attr].agg(ops).to_frame('values').reset_index()
    my_pivot = pd.pivot_table(
        data=tmp,
        index=item,
        columns=layer,
        values='values',
        fill_value=0
    )
    my_pivot.columns = [
        my_pivot.columns.names[0] + f'_{attr}_pivot_{ops}_' + str(col)
        for col in my_pivot.columns
    ]
    return my_pivot
```

```
# 日度时长总和透视
sum_pivot = get_pivot(df, 'date', 'event', 'duration', 'sum')
sum_pivot
```

```
# 接单工作每天总时长分析
print(
    'The mean time for 接单工作 in year 2021 every day is',
sum_pivot['event_duration_pivot_sum_接单工作'][
                        sum_pivot.index >= '2021-01-01'
].mean()
)

sum_pivot['event_duration_pivot_sum_接单工作'][
    sum_pivot['event_duration_pivot_sum_接单工作'] > 0
]
```

```
title = f'{part} 每日接单工作时长折线图'

sum_pivot['event_duration_pivot_sum_接单工作'][
    sum_pivot['event_duration_pivot_sum_接单工作'] > 0
].plot(figsize=(20, 10), title=title)

plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Sum of duration', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
title = f'{part} 每日接单工作时长直方图'

sum_pivot['event_duration_pivot_sum_接单工作'][
    sum_pivot['event_duration_pivot_sum_接单工作'] > 0
].plot(
    kind='hist', figsize=(20, 10), title=title
)

plt.title(title, fontsize=20)
plt.xlabel('Sum of duration', fontsize=20)
plt.ylabel('Frequency', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
# 磨蹭每天总时长分析
print(
    'The mean time for 磨蹭 in year 2021 every day is',
sum_pivot['event_duration_pivot_sum_磨蹭'][
                    sum_pivot.index >= '2021-01-01'
].mean()
)

sum_pivot['event_duration_pivot_sum_磨蹭'][
    sum_pivot['event_duration_pivot_sum_磨蹭'] > 0
]
```

```
title = f'{part} 每日磨蹭时长折线图'

sum_pivot['event_duration_pivot_sum_磨蹭'][
    sum_pivot['event_duration_pivot_sum_磨蹭'] > 0
].plot(figsize=(20, 10), title=title)


plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Sum of duration', fontsize=20)
plt.xticks(fontsize=20)
```

```
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
title = f'{part} 每日磨蹭时长直方图'

sum_pivot['event_duration_pivot_sum_磨蹭'][
    sum_pivot['event_duration_pivot_sum_磨蹭'] > 0
].plot(
    kind='hist', figsize=(20, 10), title=title
)

plt.title(title, fontsize=20)
plt.xlabel('Count', fontsize=20)
plt.ylabel('Frequency', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
# 次数透视
count_pivot = get_pivot(df, 'date', 'event', 'duration', 'count')
count_pivot
```

```
# 接单工作次数分析
title = f'{part} 每日接单工作次数折线图'

count_pivot['event_duration_pivot_count_接单工作'][
    count_pivot['event_duration_pivot_count_接单工作'] > 0
].plot(figsize=(20, 10), title=title)

plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Count', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
title = f'{part} 每日接单工作次数直方图'

count_pivot['event_duration_pivot_count_接单工作'][
    count_pivot['event_duration_pivot_count_接单工作'] > 0
].plot(kind='hist', figsize=(20, 10), title=title)

plt.title(title, fontsize=20)
plt.xlabel('Count', fontsize=20)
plt.ylabel('Frequency', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
```

```
plt.savefig(file, dpi=600)
```

```
# 磨蹭次数分析
print(
    'The mean count for 磨蹭 in year 2021 every day is',
count_pivot['event_duration_pivot_count_磨蹭'][
                    count_pivot.index >= '2021-01-01'
].mean()
)

count_pivot['event_duration_pivot_count_磨蹭'][
    count_pivot['event_duration_pivot_count_磨蹭'] > 0
]
```

```
title = f'{part} 每日磨蹭次数折线图'

count_pivot['event_duration_pivot_count_磨蹭'][
    count_pivot['event_duration_pivot_count_磨蹭'] > 0
].plot(figsize=(20, 10), title=title)

plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Count', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
title = f'{part} 每日磨蹭次数直方图'

count_pivot['event_duration_pivot_count_磨蹭'][
    count_pivot['event_duration_pivot_count_磨蹭'] > 0
].plot(kind='hist', figsize=(20, 10), title=title)

plt.title(title, fontsize=20)
plt.xlabel('Count', fontsize=20)
plt.ylabel('Frequency', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

2.4.7) 每年, 月, 星期的事项关键词词云图

```
part = '2.4.7'
def get_keys(dic, threshold=.5):
    _sum = sum(list(dic.values()))
    res, key_sum = [], 0
    for k, v in sorted(dic.items(), key=lambda x:x[1], reverse=True):
        if key_sum / _sum <= threshold:
            res.append(k)
            key_sum += v
```

```
    return ' '.join(res)

def get_key_words(info, col, threshold=1):
    info = info.groupby(col)['event'].agg(list).reset_index()
    dct = Dictionary(info['event'].values)
    corpus = [dct.doc2bow(line) for line in info['event'].values]

    # 用TfidfModel筛选关键词
    model = TfidfModel(corpus)
    movie_profile = {}
    for i, mid in enumerate(info[col]):
        vector = model[corpus[i]]
        movie_tag = sorted(vector, key=lambda x:x[1], reverse=True)
        movie_profile[mid] = dict(map(lambda x:(dct[x[0]], x[1]), movie_tag))

    info['key_words'] = [get_keys(dic, threshold) for _, dic in
movie_profile.items()]
    return info

# 定义生成背景字样的函数
def WordPic(text, path):
    lens = 2000
    image = Image.new("RGB",(lens,lens),"white")
    draw_table = ImageDraw.Draw(im=image)
    draw_table.text(
        xy=(0, lens // 3),
        text=str(text),
        fill='#000000',
        font=ImageFont.truetype('C:\\windows\\Fonts\\STHUPO.ttf', 450)
    )

    # image.show()  # 直接显示图片
    image.save(os.path.join(path, 'back.png'), 'PNG')  # 保存在当前路径下，格式为PNG
    image.close()

# 筛选关键词中的event做词云图
def plot_key_word_cloud(info, col, path='', mask=None, store=True):
    info['plot_words'] = info.apply(
        lambda x: [e for e in x['event'] if e in x['key_words']],
        axis=1
    )
    for c, words in tqdm(info[[col, 'key_words']].itertuples(index=False)):
        try:
            dpi = 500
            if mask is not None:
                dpi = 1500
                WordPic(c, path)
                mask = np.array(Image.open(os.path.join(path, 'back.png')))

            cut_text = ' '.join(jieba.cut(''.join(words)))
            wordcloud = WordCloud(
                font_path='C:\\windows\\Fonts\\STHUPO.ttf',
                background_color='white', mask=mask,
                scale=2
            ).generate(cut_text)
            plt.figure(dpi=200)
            plt.imshow(wordcloud, interpolation='bilinear')
            plt.axis('off')
```

```
            if store:
                plt.savefig(
                    os.path.join(path, f'{part} 按{col}分类后{c}的关键词.png'),
                    dpi=dpi
                )
        except:
            continue
```

```
# 年度关键词词云图
col = 'year'
year_key_words = get_key_words(df, col)
plot_key_word_cloud(year_key_words, col, path=path, mask=1)
year_key_words
```

```
# 2021年每月关键词
col = 'month'
month_key_words = get_key_words(df[df['year'] == '2021'], col)
plot_key_word_cloud(month_key_words, col, path=path)
month_key_words
```

```
# 2021年 每day_period关键词
col = 'day_period'
day_period_key_words = get_key_words(df[df['year'] == '2021'], col)
plot_key_word_cloud(day_period_key_words, col, path=path)
day_period_key_words
```

```
# 2021年 duration_attr关键词
col = 'duration_attr'
duration_attr_key_words = get_key_words(df[df['year'] == '2021'], col)
plot_key_word_cloud(duration_attr_key_words, col, path=path)
duration_attr_key_words
```

```
# 2021年 attr关键词
col = 'attr'
attr_key_words = get_key_words(df[df['year'] == '2021'], col, threshold=.8)
plot_key_word_cloud(attr_key_words, col, path=path)
attr_key_words
```

```
# 2021年 weekday关键词
col = 'weekday'
weekday_key_words = get_key_words(df[df['year'] == '2021'], col, threshold=1)
plot_key_word_cloud(weekday_key_words, col, path=path)
weekday_key_words
```

```
# 2021年 mid_hour关键词
col = 'mid_hour'
midhour_key_words = get_key_words(df[df['year'] == '2021'], col, threshold=1)
plot_key_word_cloud(midhour_key_words, col, path=path)
midhour_key_words
```

2.4.8) 每天事项的对应先后以及月度关联的网络图

```
# 2.4.8) 每天事项的对应先后以及月度关联的网络图
node_color = {
    '有效时间':'r',
    '浪费时间':'b',
    '必要时间':'g'
}

part = '2.4.8'

def sort_dic(dic:dict, reverse=True):
    temp = sorted(dic.items(), key=lambda x:x[1], reverse=reverse)
    res = '{'
    for k, v in temp:
        res += f'{k}:{v},'
    res += '}'
    return res

def net(df, col_from, col_to, year, weight=None, dis=None, directed=False):
    if weight is None:
        df['weight'] = 1
        df = df.groupby([col_from, col_to])['weight'].sum().reset_index()
        df = df.rename(columns={
            col_from:'from',
            col_to:'to'
        })
    else:
        df = df.rename(columns={
            col_from:'from',
            col_to:'to',
            weight:'weight'
        })

    if directed:
        GA = nx.from_pandas_edgelist(
            df,
            source="from",
            target="to",
            edge_attr='weight',
            create_using=nx.DiGraph()
        )
    else:
        GA = nx.from_pandas_edgelist(
            df,
            source="from",
            target="to",
            edge_attr='weight',
        )
    print(nx.info(GA))
    author_lst = df['from'].to_list() + df['to'].to_list()

    dic = {i: author_lst.count(i) for i in author_lst if author_lst.count(i) >
0}
    nodelst = []
    node_colors = []
    sizes = []
```

```
    for m, k in dic.items():
        if node_color.get(event2attr.get(m)) is not None:
            node_colors.append(node_color.get(event2attr.get(m)))
        else:
            node_colors.append('m')
        nodelst.append(m)
        sizes.append(k * 3000)
    print('closeness_centrality:', sort_dic(nx.betweenness_centrality(GA)))
    plt.figure(figsize=(50, 50), dpi=50)
    pos = nx.spring_layout(GA, k=dis, iterations=50)

    labels = {}
    for m,k in dic.items():
        if k >= 1:
            #set the node name as the key and the label as its value
            labels[m] = m
    #set the argument 'with labels' to False so you have unlabeled graph

    edges = GA.edges()
    weights = [GA[u][v]['weight'] for u,v in edges]

    nx.draw(
        GA,
        pos,
        with_labels=False,
        alpha=0.5,
        node_color=node_colors,
        nodelist=nodelst,
        node_size=sizes,
        font_family='SimHei',
    )

    #Now only add labels to the nodes you require (the hubs in my case)
    nx.draw_networkx_labels(
        GA,
        pos,
        labels,
        font_family='SimHei',
        font_size=60,
        font_color='black'
    )

    nx.draw_networkx_edges(
        GA,
        pos,
        width=weights,
        edge_color='c',
        alpha=0.5,
        arrowsize=250,
        arrows=True,
    )
    title = f'{part} {col_from}与{col_to}在{year}的网络图'
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=50)

# 月份与事项的关系图
def month2event(n):
    assert n in range(2019, 2022)
```

```
    n = str(n)
    temp = df[df['year'] == n].copy()
    month_event = temp.groupby(['month', 'event'])
['duration'].count().reset_index()
    net(month_event, 'month', 'event', year=n, weight='duration',
directed=False, dis=3)

# 前后事项的网络图
def PostEvent(n):
    assert n in range(2019, 2022)
    n = str(n)
    temp = df[df['year'] == n].copy()
    pro_post_ass = temp[['date', 'event']]
    pro_post_ass['previous_event'] = temp.groupby(['date'])['event'].shift()
    pro_post_ass['post_event'] = temp.groupby(['date'])['event'].shift(-1)
    pro_post_ass = pro_post_ass.dropna()
    net(pro_post_ass, 'event', 'post_event', year=n, directed=True, dis=5)
```

```
month2event(2019)
```

```
month2event(2020)
```

```
month2event(2021)
```

```
PostEvent(2019)
```

```
PostEvent(2020)
```

```
PostEvent(2021)
```

2.5 一键式EDA

2.5.1 原始数据

```
AV = AutoViz_Class()
dft = AV.AutoViz(
    filename=None,
    sep=",",
    depVar="duration",
    dfte=df,
    header=0,
    verbose=0,
    lowess=True,
    chart_format="svg",
    max_rows_analyzed=150000,
    max_cols_analyzed=30,
)
```

2.5.2 衍生数据

```
AV = AutoViz_Class()
dft = AV.AutoViz(
    filename=None,
    sep=",",
    depVar="effective time",
    dfte=daily_attr_copy,
    header=0,
    verbose=0,
    lowess=True,
    chart_format="svg",
    max_rows_analyzed=150000,
    max_cols_analyzed=30,
)
```

3.数据分析

3.1 方差分析每月或是每年的事项是否有差异

3.1.1 原始事项时长数据的方差分析

```
# 原始事项数据的方差分析
part = '3.1.1'

def vr_event(data, num, cate):
    formula = f'{num}~C({cate})'
    anova_re_2019 = anova_lm(ols(formula, data=data[[num, cate]]).fit())
    res = anova_re_2019.loc[f'C({cate})', 'PR(>F)']
    return res
```

```
# 事项时长数据
vr_duration_table = pd.DataFrame()
for event in tqdm(df['event'].unique()):
    for cate in cate_vars:
        if cate not in ['event', 'attr']:
            temp = df[df['event'] == event].copy()
            temp[cate] = temp[cate].astype(str)
            res = vr_event(temp, 'duration', cate)
            vr_duration_table.loc[event, cate] = res

# 该表格是各个事项的时长数据关于各分类变量的方差分析结果
vr_duration_table
```

```
# 选取那些在不同分类情况下时长有显著差异的数据
vr_duration_table.to_excel('事项时长数据的方差分析结果.xlsx', index=True)
significant_event = vr_duration_table[vr_duration_table > .05].dropna(axis=0,
how='all')
significant_event
```

```
# 对于有显著差异的事项 做出在不同分类中均值的bar图
mins = 30  # 剔除掉出现次数少的事项
for e in tqdm(significant_event.index):
    if len(df[df['event'] == e]) >= mins:
        for c in significant_event.columns:
            if significant_event.loc[e, c] >= .05:
                plt.figure(dpi=100)
                df[df['event'] == e].groupby(c)['duration'].agg('mean').plot(
                    kind='bar', figsize=(20, 10)
                )
                title = f'{part} mean value of {e} duration in different {c}
groups'
                plt.title(title, fontsize=20)
                plt.xlabel(e, fontsize=20)
                plt.ylabel('Mean value', fontsize=20)
                plt.xticks(fontsize=20)
                plt.yticks(fontsize=20)
                file = os.path.join(path, f'{title}.png')
                plt.savefig(file, dpi=600)
```

```
# 只分析2021年
vr_duration_table2021 = pd.DataFrame()
for event in tqdm(df['event'].unique()):
    for cate in cate_vars:
        if cate not in ['event', 'attr', 'year']:
            try:
                temp = df[
                    (df['event'] == event) & (df['year'] == '2021')
                ].copy()
                temp[cate] = temp[cate].astype(str)
                res = vr_event(temp, 'duration', cate)
                vr_duration_table2021.loc[event, cate] = res
            except:
                continue

# 该表格是各个事项的时长数据关于各分类变量的方差分析结果
vr_duration_table2021
```

```
significant_event2021 = vr_duration_table2021[vr_duration_table2021 > .05].\
    dropna(axis=0, how='all')

mins = 20

for e in tqdm(significant_event2021.index):
    if len(df[df['event'] == e]) >= mins:
        for c in significant_event2021.columns:
            if significant_event2021.loc[e, c] >= .05:
                plt.figure(dpi=100)
                df[
                    (df['event'] == e) & (df['year'] == '2021')
                ].groupby(c)['duration'].agg('mean').plot(
                    kind='bar', figsize=(20, 10)
                )
                title = f'{part} mean value of {e} duration in different {c}
groups in year 2021'
                plt.title(title, fontsize=20)
                plt.xlabel(e, fontsize=20)
                plt.ylabel('Mean value', fontsize=20)
                plt.xticks(fontsize=20)
                plt.yticks(fontsize=20)
                file = os.path.join(path, f'{title}.png')
                plt.savefig(file, dpi=600)
```

3.1.2 衍生数据的方差分析

```
part = '3.1.2'

daily_attr['weekday'] = pd.to_datetime(daily_attr['date']).dt.weekday + 1
daily_attr['year'] = daily_attr['date'].apply(lambda x:str(x).split('-')[0])
daily_attr['month'] = daily_attr['date'].apply(lambda x:str(x).split('-')[1])
daily_attr['day'] = daily_attr['date'].apply(lambda x:str(x).split('-')[2])
daily_attr['week_order'] = daily_attr['date'].map(lambda x: pd.Period(x,
freq='W'))
daily_attr
```

```
# 变量类型划分
num_vars = [
    '有效时间', '浪费时间', '必要时间', '手机时间'
]

cates_vars = [
    'year', 'month', 'day', 'weekday', 'week_order'
]

# 逐一进行方差分析
daily_sum_table = pd.DataFrame()
for num in tqdm(num_vars):
    for cate in cates_vars:
        temp = daily_attr[[num, cate]].copy()
        temp[cate] = temp[cate].astype(str)
        res = vr_event(temp, num, cate)
        daily_sum_table.loc[num, cate] = res

# 该表格是各个事项的时长日度求和数据数据关于各分类变量的方差分析结果
daily_sum_table
```

```
# 对于有显著差异的事项 做出在不同分类中均值的bar图
for e in tqdm(daily_sum_table.index):
    for c in daily_sum_table.columns:
        if daily_sum_table.loc[e, c] >= .05:
            plt.figure(dpi=100)
            daily_attr.groupby(c)[e].agg('mean').plot(
                kind='bar', figsize=(20, 10)
            )
            title = f'{part} mean value of {e} duration in different {c} groups'
            plt.title(title, fontsize=20)
            plt.xlabel(e, fontsize=20)
            plt.ylabel('Mean value', fontsize=20)
            plt.xticks(fontsize=20)
            plt.yticks(fontsize=20)
            file = os.path.join(path, f'{title}.png')
            plt.savefig(file, dpi=600)
```

3.2 LSTM预测前后事项

3.2.1 建立字符索引

```
corpus_chars = df['event'].values
idx_to_char = list(set(corpus_chars)) # 去重，得到索引到字符的映射
char_to_idx = {char: i for i, char in enumerate(idx_to_char)} # 字符到索引的映射
vocab_size = len(char_to_idx)
print(vocab_size)
print('len(corpus_chars) = ', len(corpus_chars))

corpus_indices = [char_to_idx[char] for char in corpus_chars]  # 将每个字符转化为索引，得到一个索引的序列
sample = corpus_indices[: 20]
print('chars:', ' '.join([idx_to_char[idx] for idx in sample]))
print('indices:', sample)
```

```
def load_data():
    idx_to_char = list(set(corpus_chars))
    char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
    vocab_size = len(char_to_idx)
    corpus_indices = [char_to_idx[char] for char in corpus_chars]
    return corpus_indices, char_to_idx, idx_to_char, vocab_size
```

```
# 随机采样
def data_iter_random(corpus_indices, batch_size, num_steps, device=None):
    # 减1是因为对于长度为n的序列，X最多只有包含其中的前n - 1个字符
    num_examples = (len(corpus_indices) - 1) // num_steps  # 下取整，得到不重叠情况下的样本个数
    example_indices = [i * num_steps for i in range(num_examples)]  # 每个样本的第一个字符在corpus_indices中的下标
    random.shuffle(example_indices)

    def _data(i):
        # 返回从i开始的长为num_steps的序列
        return corpus_indices[i: i + num_steps]
    if device is None:
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    for i in range(0, num_examples, batch_size):
        # 每次选出batch_size个随机样本
        batch_indices = example_indices[i: i + batch_size]  # 当前batch的各个样本的首字符的下标
        X = [_data(j) for j in batch_indices]
        Y = [_data(j + 1) for j in batch_indices]
        yield torch.tensor(X, device=device), torch.tensor(Y, device=device)
```

3.2.2 建立LSTM模型

```
corpus_indices, char_to_idx, idx_to_char, vocab_size = load_data()
num_inputs, num_hiddens, num_outputs = vocab_size, 128, vocab_size
pred_period, pred_len, prefixes = 50, 50, ['磨蹭', '运动']
rnn_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens)
num_steps, batch_size = 35, 2
X = torch.rand(num_steps, batch_size, vocab_size)
state = None
Y, state_new = rnn_layer(X, state)
print(Y, state_new)
```

```
def one_hot(x, n_class, dtype=torch.float32):
    result = torch.zeros(len(x), n_class, dtype=dtype, device=x.device)  # shape: (n, n_class)
    result.scatter_(1, x.long().view(-1, 1), 1)  # result[i, x[i, 0]] = 1
    return result

def to_onehot(X, n_class):
    return [one_hot(X[:, i], n_class) for i in range(X.shape[1])]

class RNNModel(nn.Module):
    def __init__(self, rnn_layer, vocab_size):
        super(RNNModel, self).__init__()
        self.rnn = rnn_layer
```

```
        self.hidden_size = rnn_layer.hidden_size * (2 if rnn_layer.bidirectional
else 1)
        self.vocab_size = vocab_size
        self.dense = nn.Linear(self.hidden_size, vocab_size)

    def forward(self, inputs, state):
        # inputs.shape: (batch_size, num_steps)
        X = to_onehot(inputs, vocab_size)
        X = torch.stack(X)  # X.shape: (num_steps, batch_size, vocab_size)
        hiddens, state = self.rnn(X, state)
        hiddens = hiddens.view(-1, hiddens.shape[-1])  # hiddens.shape:
(num_steps * batch_size, hidden_size)
        output = self.dense(hiddens)
        return output, state
```

```
# 实现一个预测函数，与前面的区别在于前向计算和初始化隐藏状态
def predict_rnn_pytorch(prefix, num_chars, model, vocab_size, device,
idx_to_char,
                        char_to_idx):
    state = None
    output = [char_to_idx[prefix[0]]]  # output记录prefix加上预测的num_chars个字符
    for t in range(num_chars + len(prefix) - 1):
        X = torch.tensor([output[-1]], device=device).view(1, 1)
        (Y, state) = model(X, state)  # 前向计算不需要传入模型参数
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            output.append(Y.argmax(dim=1).item())
    return '=>'.join([idx_to_char[i] for i in output])
```

```
device = 'cpu'
model = RNNModel(rnn_layer, vocab_size).to(device)
predict_rnn_pytorch(['运动'], 10, model, vocab_size, device, idx_to_char,
char_to_idx)
```

3.2.3 模型训练

```
def grad_clipping(params, theta, device):
    norm = torch.tensor([0.0], device=device)
    for param in params:
        norm += (param.grad.data ** 2).sum()
    norm = norm.sqrt().item()
    if norm > theta:
        for param in params:
            param.grad.data *= (theta / norm)

def data_iter_consecutive(corpus_indices, batch_size, num_steps, device=None):
    if device is None:
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    corpus_len = len(corpus_indices) // batch_size * batch_size  # 保留下来的序列的
长度
    corpus_indices = corpus_indices[: corpus_len]  # 仅保留前corpus_len个字符
    indices = torch.tensor(corpus_indices, device=device)
    indices = indices.view(batch_size, -1)  # resize成(batch_size, )
    batch_num = (indices.shape[1] - 1) // num_steps
```

```
    for i in range(batch_num):
        i = i * num_steps
        X = indices[:, i: i + num_steps]
        Y = indices[:, i + 1: i + num_steps + 1]
        yield X, Y
```

```
def train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
                                  corpus_indices, idx_to_char, char_to_idx,
                                  num_epochs, num_steps, lr, clipping_theta,
                                  batch_size, pred_period, pred_len, prefixes):
    loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    model.to(device)
    for epoch in tqdm(range(num_epochs)):
        l_sum, n = 0.0, 0
        data_iter = data_iter_consecutive(corpus_indices, batch_size, num_steps,
device) # 相邻采样
        state = None
        for X, Y in data_iter:
            loss_list = []
            if state is not None:
                # 使用detach函数从计算图分离隐藏状态
                if isinstance (state, tuple): # LSTM, state:(h, c)
                    state[0].detach_()
                    state[1].detach_()
                else:
                    state.detach_()
            (output, state) = model(X, state) # output.shape: (num_steps *
batch_size, vocab_size)
            y = torch.flatten(Y.T)
            l = loss(output, y.long())
            loss_list.append(l)
            optimizer.zero_grad()
            l.backward()
            grad_clipping(model.parameters(), clipping_theta, device)
            optimizer.step()
            l_sum += l.item() * y.shape[0]
            n += y.shape[0]


        if (epoch + 1) % pred_period == 0:
            for prefix in prefixes:
                print(' -', predict_rnn_pytorch(
                    prefix, pred_len, model, vocab_size, device, idx_to_char,
                    char_to_idx))
```

```
rnn_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens)
model = RNNModel(rnn_layer, vocab_size).to(device)
num_epochs, batch_size, lr, clipping_theta = 2000, 64, 1e-2, 1e-2
print_num = 10
pred_period = num_epochs / print_num
pred_len, prefixes = 15, [
    ['华尔街日报'], ['晚饭'], ['摄影'], ['接单工作']
]
train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
    corpus_indices, idx_to_char, char_to_idx,
    num_epochs, num_steps, lr, clipping_theta,
    batch_size, pred_period, pred_len, prefixes
)
```

```
rnn_layer = nn.GRU(input_size=vocab_size, hidden_size=num_hiddens)
model = RNNModel(rnn_layer, vocab_size).to(device)
num_epochs, batch_size, lr, clipping_theta = 2000, 64, 1e-2, 1e-2
print_num = 10
pred_period = num_epochs / print_num
pred_len, prefixes = 15, [
    ['华尔街日报'], ['晚饭'], ['接单工作'], ['托福听力'], ['磨蹭']
]
train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
    corpus_indices, idx_to_char, char_to_idx,
    num_epochs, num_steps, lr, clipping_theta,
    batch_size, pred_period, pred_len, prefixes
)
```

3.3 时长数据的简单聚类

```
part = '3.3'

dates = pd.DataFrame(
    pd.date_range(
    start='2021-01-01',
    end='2021-12-31'
).astype(str), columns=['date']
)

data = pd.merge(
    dates, daily_attr[num_vars + ['date']], on='date', how='left'
)
```

```
mergings = linkage(data.drop('date', axis=1).dropna())
plt.figure(figsize=(20, 10))
dendrogram(
    mergings,
    leaf_rotation=90,
    leaf_font_size=10,
)
plt.yticks(fontsize=20)
title = f'{part} 2021时长数据层次聚类图'
plt.title(title, fontsize=10)
plt.savefig(os.path.join(path, f'{title}.png'), dpi=600)
```

3.4 关于日期的缺失值处理

```
# 对日度数据做处理 只分析2021年
part = '3.4'

# 生成新特征
data['weekday'] = pd.to_datetime(data['date']).dt.weekday + 1
data['month'] = data['date'].apply(lambda x:str(x).split('-')[1])
data['day'] = data['date'].apply(lambda x:str(x).split('-')[2])
data['week_order'] = data['date'].map(lambda x: pd.Period(x, freq='W'))

data.set_index('date')[num_vars].plot(
    figsize=(20, 10)
)

title = '各类时长每日总和的折线图'
plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Sum of duration', fontsize=20)
plt.yticks(fontsize=20)
plt.xticks(fontsize=20)
plt.legend(loc=1, fontsize=20)

print('缺失的行数有', data['有效时间'].isnull().sum())
```

```
# 采样三次样条处理缺失
for col in data.columns:
    if '时间' in col:
        upper, lower = data[col].max(), data[col].min()
        data[col] = data[col].interpolate(method='spline', order=3)
        # 避免插值结果太超过想象
        data[col] = np.where(abs(data[col]) > upper, upper, data[col])
        data[col] = np.where(data[col] < lower, lower, data[col])
data = data.fillna(axis=0, method='bfill')


title = f'{part} 处理缺失后的每日时长总和图'
data.set_index('date')[num_vars].plot(
    figsize=(20, 10), title=title
)

plt.title(title, fontsize=20)
plt.xlabel('Date', fontsize=20)
plt.ylabel('Sum of duration', fontsize=20)
plt.yticks(fontsize=20)
plt.xticks(fontsize=20)
plt.legend(loc=1, fontsize=20)

file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)

print('缺失的行数有', data['有效时间'].isnull().sum())
```

3.5 关于日期的傅里叶分解求周期

```
part = '3.5'

def FRT(raw, k):
    fft_size = 1024*4
    peak_num =3 #找前几个周期;
    figure_flag = 1 #傅里叶变换是否画图，0：不画图 1：画图
    #获取指数同比序列
    seq = raw[k]
    seq_log = np.array(seq)
    dseq_log = signal.detrend(seq_log) #去趋势项
    freq_index = [i for i in range(fft_size)]
    freq_index[:int(fft_size/2+1)] = [i / fft_size for i in range(0,
int(fft_size / 2 + 1))]
    # freq_index(fft_size/2+2:end) = [-fft_size/2+1:1:-1]/fft_size
    freq_index[int(fft_size/2+1):] = [i / fft_size for i in range(int(-fft_size
/ 2 + 1), 0)]
    # data_fft = abs(fftshift(fft(dseq_log, fft_size)));
    data_fft = abs(np.fft.fftshift(np.fft.fft(dseq_log, fft_size)))
    #freq_index = fftshift(freq_index);
    freq_index = np.fft.fftshift(freq_index)
    peak_num = 3

    loc_raw = list(signal.find_peaks(data_fft[int(fft_size/2):])[0])
    peak_raw = data_fft[int(fft_size/2):][loc_raw]
    data_dict = { key: value for key,value in zip(loc_raw,peak_raw)}
    temp = sorted(data_dict.items(),key=lambda x:x[1],reverse=True)
    loc_peaks = temp[:peak_num]

    plt.figure(figsize=(20,10))
    plt.plot(freq_index[int(fft_size/2+1):],data_fft[int(fft_size/2+1):])
    plt.grid(True)
    plt.xlabel('频率(Hz)', fontsize=20)
    plt.ylabel('幅度', fontsize=20)
    title = f'{part} {k}幅度频率图'
    plt.title(title, fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    for i in range(peak_num):
        plt.text(freq_index[loc_peaks[i][0]+int(fft_size/2-1)],
        loc_peaks[i][1],
        '[' + str(round(1 / freq_index[loc_peaks[i][0] + int(fft_size/2-1)],2))
+ \
            ',' + str(round(loc_peaks[i][1])) + ']',
        fontsize=20
        )
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)

    # dseq_log,fft_size,peak_num,figure_flag
    def period_mean_fft(data,nfft,peak_num,figure_flag):
        data_n = np.array(data)
        if len(data_n.shape)>1 and data_n.shape[-1]>data_n.shape[0]:
            data_n = data_n.T
        Y_fft = np.fft.fft(data_n,nfft)
        nfft = len(Y_fft)
        Y_fft = np.delete(Y_fft,0)
        #power = abs(Y(1:floor(nfft/2))).^2;  %求功率谱
        power = np.abs(np.square(Y_fft[:int(np.floor(nfft/2))]))  #求功率谱
```

```
        #Amplitude = abs(Y(1:floor(nfft/2)));  %求振幅
        amplitude = np.abs(Y_fft[:int(np.floor(nfft/2))])  #求振幅
        nyquist = 1/2
        #freq = (1:floor(nfft/2))/(floor(nfft/2))*nyquist;
        # %求频率，从低频到高频，最高的频率是nyquist,最低的频率是2/N
        freq = [nyquist*i/np.floor(nfft/2) for i in
range(1,int(np.floor(nfft/2)+1))]
        freq = np.array(freq)
        period = 1/freq
        #[peaks,posi] = findpeaks(power,'NPeaks',peak_num,'SortStr','descend');
        loc_raw = list(signal.find_peaks(power)[0])
        peak_raw = power[loc_raw]
        data_dict = { key: value for key,value in zip(loc_raw,peak_raw)}
        temp = sorted(data_dict.items(),key=lambda x:x[1],reverse=True)
        loc_peaks = temp[:peak_num]
        posi = [loc_peaks[i][0] for i in range(peak_num)]
        T_fft = period[posi]
        if figure_flag == 1:
            plt.figure(figsize=(20,10))
            plt.plot(period,power)
            plt.grid(True)
            plt.xlabel('周期', fontsize=20)
            plt.ylabel('功率', fontsize=20)
            title = f'{part} {k}的周期-振幅图'
            plt.title(title, fontsize=20)
            for i in range(peak_num):
                plt.text(period[loc_peaks[i][0]],
                loc_peaks[i][1],
                '[' + str(round(period[loc_peaks[i][0]], 2)) + \
                    ',' + str(round(loc_peaks[i][1])) + ']',
                fontsize=20
                )
            plt.xticks(fontsize=20)
            plt.yticks(fontsize=20)
            plt.xlim(12,300)
            file = os.path.join(path, f'{title}.png')
            plt.savefig(file, dpi=600)

        return T_fft

    return period_mean_fft(dseq_log,fft_size,peak_num,figure_flag)
```

```
# 有效时间
FRT(data, '有效时间')
```

```
# 浪费时间
FRT(data, '浪费时间')
```

```
# 必要时间
FRT(data, '必要时间')
```

```
# 手机时间
FRT(data, '手机时间')
```

3.6 关于日期的传统时间序列分析

3.6.1 ARIMA分析

```
part = '3.6.1'

def cum_sum(data, n=1):
    if n == 1:
        return data.cumsum()
    return cum_sum(data, n - 1).cumsum()

def diff(data, n):
    if n == 1:
        return data.diff()
    return diff(data, n - 1).diff()


#ARIMA
class ARIMA_auto:
    def __init__(self, data, col):
        self.col = col
        self.data = data
        self.result_ARMA = self.fit()
        self.predictions_ARMA = self.pre()
        self.arch_p = self.arch_diag()

    # Test seasonality and stationarity
    def diff_stat(self):
        nor_p = stats.shapiro(self.data)
        if nor_p[1] > 0.05:
            print(f'{self.col} can be assumed to be normally distributed')
        else:
            print(f'{self.col} cannot be assumed to be normally distributed')

        # ACF
        fig, ax = plt.subplots(figsize=(20, 10))
        fig = plot_acf(self.data, alpha=0.05, ax=ax, lags=10, unbiased=True)
        title = f'{part} {self.col}的ACF图'
        plt.xlabel('Order', fontsize=20)
        plt.ylabel('ACF', fontsize=20)
        plt.title(title, fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)

        # PACF
        fig, ax = plt.subplots(figsize=(20, 10))
        fig = plot_pacf(self.data, alpha=0.05, ax=ax, lags=10)
        title = f'{part} {self.col}的PACF图'
        plt.xlabel('Order', fontsize=20)
        plt.ylabel('PACF', fontsize=20)
        plt.title(title, fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)
```

```
        # 时间序列分解
        ts = seasonal_decompose(self.data, freq=12)
        plt.figure(figsize=(20, 10))
        ts.plot()
        title = f'{part} {self.col}的时间序列分解图'
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)

        p_value = adfuller(self.data)[1]
        if p_value < 0.05:
            print(f'{self.col} is stationary')
            return 0
        print(f'{self.col} is not stationary')
        i = 1
        p_value = adfuller(diff(self.data, i).dropna())[1]
        while p_value >= 0.05:
            i += 1
            p_value = adfuller(diff(self.data, i).dropna())[1]
        return i


    # find the order for ARMA
    @staticmethod
    def find_order(series):
        order = st.arma_order_select_ic(
                series,
                max_ar=3,
                max_ma=3,
                ic=['aic', 'bic', 'hqic']
            )
        order_min = order.aic_min_order
        p = order_min[0]
        q = order_min[1]
        return p, q

    @staticmethod
    def _fit(series, order):
        p, d, q = order
        if d == 0:
            model = ARMA(
                    series,
                    order=(p, q)
            )
        else:
            model = ARMA(
                    diff(series, d).dropna(),
                    order=(p, q)
            )
        return model.fit(disp=-1)

    def fit(self):
        # find the order for differentiation
        d = self.diff_stat()
        if d > 0:
            p, q = self.find_order(diff(self.data, d).dropna())
        else:
            p, q = self.find_order(self.data)
```

```
        self.arima_order = p, d, q
        # fit the ARMA
        try:
            return self._fit(self.data, self.arima_order)
        except:
            d += 1
            p, q = self.find_order(diff(self.data, d).dropna())
            self.arima_order = p, d, q
            return self._fit(self.data, self.arima_order)

    # predict
    def pre(self):
        predictions_ARMA_diff = pd.Series(
            self.result_ARMA.fittedvalues,
            copy=True,
            index=self.data.index
        )
        plt.figure(figsize=(20, 10))
        if self.arima_order[1] > 0:
            predictions_ARMA = cum_sum(predictions_ARMA_diff,
self.arima_order[1])
        else:
            predictions_ARMA = predictions_ARMA_diff

        temp = pd.concat(
            [self.data, predictions_ARMA], axis=1
        )
        temp.columns = ['original data', 'fitted data']
        colormaps = {
                'original data':'blue',
                'fitted data':'red'
            }
        for col in temp.columns:
            temp[col].plot(
                style=colormaps.get(col)
            )
        plt.legend(loc=1, fontsize=20)
        title = f'{part} {self.col} ARIMA({self.arima_order[0]},
{self.arima_order[1]}, {self.arima_order[2]})'
        plt.xlabel('Date', fontsize=20)
        plt.ylabel('Duration', fontsize=20)
        plt.title(title, fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)
        return predictions_ARMA

    # test ARCH effect
    def arch_diag(self):
        # ARIMA残差正态检验
        fig = plt.figure(figsize=(20, 10))
        ax1 = plt.subplot(121)
        resid = (self.data - self.predictions_ARMA).dropna()
        resid.hist(ax=ax1, bins=100)
        resid.plot(ax=ax1, kind='kde')
        plt.xlabel('Square of residuals', fontsize=20)
```

```
        plt.ylabel('Frequency', fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        ax2 = plt.subplot(122)
        stats.probplot(resid, dist='norm', plot=ax2, fit=True)  # test normality
        plt.xlabel('Theoritical quantitles', fontsize=20)
        plt.ylabel('Real quantitles', fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        plt.title('Probability Plot', fontsize=20)
        title = f'{part} {self.col}的ARIMA残差正态检验图'
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)


        #test ARCH effect
        resid2 = pd.DataFrame(resid.apply(lambda x : x ** 2))
        resid2.plot(figsize=(20, 10))
        title = f'{part} {self.col} Time Series of Residual squares'
        plt.xlabel('Date', fontsize=20)
        plt.ylabel('Residual squares', fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        plt.legend([])
        plt.title(title, fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)

        # LB test
        q, p = acorr_ljungbox(resid2)
        with plt.style.context('ggplot'):
            fig = plt.figure(figsize=(20, 10))
            axes = fig.subplots(1, 2)
            axes[0].plot(q, label='Q')
            axes[0].tick_params(axis='x',labelsize=20)
            axes[0].tick_params(axis='y',labelsize=20)
            axes[0].set_ylabel('Q', fontsize=20)
            axes[0].set_xlabel('Qrder', fontsize=20)
            axes[1].plot(p, label='p')
            axes[1].tick_params(axis='x',labelsize=20)
            axes[1].tick_params(axis='y',labelsize=20)
            axes[1].set_ylabel('P', fontsize=20)
            axes[1].set_xlabel('Qrder', fontsize=20)
            axes[0].legend(fontsize=20)
            axes[1].legend(fontsize=20)
            plt.tight_layout()
            title = f'{part} {self.col}的GARCH检验图'
            file = os.path.join(path, f'{title}.png')
            plt.savefig(file, dpi=600)

        if max(p) < .05:
            print(f'There exists strong ARCH effect for {self.col}')
        else:
            print(f'There does not exist strong ARCH effect for {self.col}')
        return max(p)
```

```
# 4类时间的ARIMA分析
arima_dict = {}
for col in num_vars:
    try:
        arima_dict[col] = ARIMA_auto(data.set_index('date')[col], col)
    except:
        continue
```

3.6.2 GARCH分析

```
part = '3.6.2'

class garch_auto:
    def __init__(self, data, col, order, dist='normal', n=20):
        self.data = data
        self.col = col
        self.arch = self.fit(order, dist=dist)
        self.n = n
        self.arch_pred = self.predict_garch(self.arch, n=self.n)

    def fit(self, order, dist):
        garch = arch_model(
            self.data,
            mean='AR',
            lags=order[0],
            vol='GARCH',
            p=1,
            q=1,
            dist=dist
        ).fit()

        self.condition_vol = pd.Series(
            garch.conditional_volatility,
            index=self.data.index
        )
        temp = pd.concat(
            [self.condition_vol, self.data], axis=1
        )
        temp.columns = ['conditional volatility', 'real values']
        print(garch.summary)

        # show the plot
        plt.figure(figsize=(20, 10))
        colormaps = {
                'conditional volatility':'blue',
                'real values':'red'
            }
        for col in temp.columns:
            temp[col].plot(
                style=colormaps.get(col)
            )
        plt.legend(loc=1, fontsize=20)
        title = f'{part} {self.col}波动率分解图'
        plt.xlabel('Date', fontsize=20)
        plt.ylabel('Volatility', fontsize=20)
        plt.xticks(fontsize=20)
```

```
        plt.yticks(fontsize=20)
        plt.title(title, fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)
        return garch

    #predict
    def predict_garch(self, garch, n=100, ar_p=1):
        params = garch.params.values
        conditional_volatility = garch.conditional_volatility
        garch_ret = []
        bef_ret = np.array(self.data[-(n + ar_p):])
        a = np.array(params[1:ar_p + 1])
        w = a[::-1]
        for i in range(n):
            fit = params[0] + w.dot(bef_ret[i:ar_p + i])
            garch_ret.append(fit)

        bef2 = [conditional_volatility.values[-1]]
        for i in range(n):
            alpha0 = params[ar_p + 1]
            alpha1 = params[ar_p + 2]
            beta1 = params[ar_p + 3]
            new = alpha0 + (alpha1 + beta1) * (bef2[-1])
            bef2 = np.append(bef2, new)
        garch_vol_pre = bef2[-n:]

        # summary the data into one df
        index = pd.date_range(
            start=pd.to_datetime(self.data.index[0]),
            periods=n + len(self.data)
        )

        # add NA so that the length matches
        condition_vol = self.condition_vol.to_list() + [np.nan] * n
        real_value = self.data.to_list() + [np.nan] * n
        predicted_volatility = [np.nan] * len(self.data) +
garch_vol_pre.tolist()

        # summarize the data
        predict = pd.concat(
            [
                pd.Series(condition_vol),
                pd.Series(real_value),
                pd.Series(predicted_volatility),
            ], axis=1
        )
        predict.index = index
        predict.columns = [
                'conditional volatility', 'real values', 'predicted_volatility'
            ]

        # plot the dataset
        plt.figure(figsize=(20, 10))
        colormaps = {
                'conditional volatility':'blue',
                'real values':'red',
                'predicted_volatility':'green'
```

```
            }
        for col in predict.columns:
            predict[col].plot(
                style=colormaps.get(col)
            )
        plt.legend(loc=1, fontsize=20)
        title = f'{part} {self.col}向前{n}步波动率预测图'
        plt.xlabel('Date', fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        plt.title(title, fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)
        return predict
```

```
# 检验是否存在GARCH效应
arch_p_dict = {
    k: v.arch_p for k, v in arima_dict.items()
}
print(arch_p_dict)
```

```
# 对存在GARCH效应的序列做GARCH建模
garch_dict = {}
garch_pre = {}
for name, p in arch_p_dict.items():
    if p < .05:
        garch_dict[name] = garch_auto(data.set_index('date')[name], name,
arima_dict[name].arima_order)
```

3.6.3 VAR分析

```
part = '3.6.3'

class VAR:
    def __init__(self, df, arima_dict):
        self.df = df.astype(float)
        self.cols = ','.join(list(self.df.columns))
        self.arima_dic = arima_dict
        self.model, self.model_res = self.fit()

    def fit(self):
        cols = self.df.columns
        varLagNum = [self.arima_dic.get(col).arima_order[0] for col in cols]
        orgMod = sm.tsa.VARMAX(self.df, order=varLagNum,trend='c',exog=None)
        fitMod = orgMod.fit(maxiter=200,disp=True)
        print(fitMod.summary())
        resid = fitMod.resid
        result = {'fitMod':fitMod,'resid':resid}
        return fitMod, result

    def Test_Cusum(self):
        result = statsmodels.stats.diagnostic.\
            breaks_cusumolsresid(self.model_res.get('resid'))
        if result[1] > .05:
            print('There is no drifting')
```

```
        else:
            print('There is drifting')
        return result

    def impulse_responses(self):
        title = f'{part} {self.cols} VAR模型的脉冲响应分析'
        ax = self.model.impulse_responses(20, orthogonalized=True).\
            plot(figsize=(20, 10))
        plt.xlabel('Prediction length', fontsize=20)
        plt.ylabel('Sum of duration', fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        plt.title(title, fontsize=20)
        plt.legend(loc=1, fontsize=20)
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)


    def variance_decompositon(self):
        md = sm.tsa.VAR(self.df.values)
        re = md.fit(2)
        fevd = re.fevd(10)
        print(fevd.summary())
        fevd.plot(figsize=(20, 10))
        plt.legend(list(self.df.columns), fontsize=20)
        title = f'{part} {self.cols} VAR模型的方差分解'
        file = os.path.join(path, f'{title}.png')
        plt.savefig(file, dpi=600)
```

```
price_vol_int_var = VAR(data[list(arima_dict.keys())], arima_dict)
price_vol_int_var.impulse_responses()
price_vol_int_var.variance_decompositon()
```

3.7 关于日期的NeuralProphet 时间序列

```
part = '3.7'

NP_dict = {}
for col in num_vars:
    nprophet_model = NeuralProphet()
    test_length = 30
    temp = data.rename(columns={'date':'ds', col: 'y'})
    metrics = nprophet_model.fit(temp[['ds', 'y']], freq="D")
    future_df = nprophet_model.make_future_dataframe(
        temp[['ds', 'y']],
        periods = test_length,
        n_historic_predictions=len(temp[['ds', 'y']])
    )
    preds_df_2 = nprophet_model.predict(future_df)
    title = f'{part} {col}的NeuralProphet时间序列预测图'
    nprophet_model.plot(
        preds_df_2,
        ylabel=col,
        figsize=(20, 10),
    );
    plt.xlabel('Date', fontsize=20)
```

```
    plt.ylabel(col, fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.title(title, fontsize=20)
    file = os.path.join(path, f'{title}.png')
    plt.savefig(file, dpi=600)
    NP_dict[col] = nprophet_model, preds_df_2
```

4.数据挖掘

4.1 通过生成聚合特征做日度时长数据的特征工程

```
nums = [
    '浪费时间', '必要时间', '手机时间'
]

cates = [
    'month', 'day', 'weekday'
]

y = '有效时间'
```

```
# 基于A/B的统计特征
for num_var in tqdm(nums):
    for cate_var in cates:
        for sts in ['mean', 'median', 'std', 'max', 'min']:
            data[f'{cate_var}_{num_var}_{sts}'] = data.groupby(cate_var)
[num_var].\
                transform(sts).values
        data[f'{cate_var}_{num_var}_cv'] = data[f'{cate_var}_{num_var}_std'] / \
            data[f'{cate_var}_{num_var}_mean']
```

```
# 流量平滑特征
# 基于A/B的统计特征
for num_var in tqdm(nums):
    for cate_var in cates:
        data[f'{num_var}_div_{cate_var}_{num_var}_mean'] = data[num_var] / \
            (data[f'{cate_var}_{num_var}_mean'] + 1e-5)
        data[f'{num_var}_div_{cate_var}_{num_var}_median'] = data[num_var] / \
            (data[f'{cate_var}_{num_var}_median'] + 1e-5)
        # 黄金组合特征
        data[f'{num_var}_minus_{cate_var}_{num_var}_mean'] = data[num_var] - \
            data[f'{cate_var}_{num_var}_mean']
```

```
# 相邻有序类别统计特征的变化
sts_list = ['max', 'min', 'mean', 'std', 'median']
for num_var in tqdm(nums):
    for cate_var in cates:
        for sts in sts_list:
            data[f'{cate_var}_{num_var}_{sts}_diff']  =
data[f'{cate_var}_{num_var}_{sts}'] - \
                data.groupby(cate_var)[f'{cate_var}_{num_var}_{sts}'].shift()

            data[f'{cate_var}_{num_var}_{sts}_ratio'] =
data[f'{cate_var}_{num_var}_{sts}'] / \
                data.groupby(cate_var)[f'{cate_var}_{num_var}_{sts}'].shift()
            data[f'{cate_var}_{num_var}_{sts}_diff'].fillna(method='bfill',
inplace=True)
            data[f'{cate_var}_{num_var}_{sts}_ratio'].fillna(method='bfill',
inplace=True)
```

4.2 类别变量的one-hot encoding

```
for cate_var in cates:
    onehot = pd.get_dummies(data[cate_var], drop_first=True, prefix=cate_var +
'_')
    data = pd.concat([data.drop(cate_var, axis=1), onehot], axis=1)

data
```

4.3 利用boruta筛选特征

```
# 划分数据集
train, test = train_test_split(data, test_size=.2)
X = train.drop(['date', 'week_order', y], axis=1)
Y = train[y]
```

```
clf = XGBRegressor()
boruta = BorutaPy(
   estimator = clf,
   n_estimators = 'auto',
   max_iter = 100 # number of trials to perform
)

# 模型训练
boruta.fit(np.array(X), np.array(Y))
# 输出结果
green_area = X.columns[boruta.support_].to_list()
blue_area  = X.columns[boruta.support_weak_].to_list()
print('features in the green area:', green_area)
print('features in the blue area:', blue_area)
cols = green_area + blue_area
train = train[cols]
```

4.4 利用PCA降维

```
dt = data[cols]
pca = PCA(n_components=0.9)
dt = pd.DataFrame(pca.fit_transform(StandardScaler().fit_transform(dt)))
dt
```

```
X_train = dt.loc[train.index, :]
X_test = dt.loc[test.index, :]
y_train = data.loc[train.index, y]
y_test = data.loc[test.index, y]
```

4.5 利用optuna优化参数

```
sampler = TPESampler(seed=10) # for reproducibility

def objective(trial):
    dtrain = lgb.Dataset(X_train, label=y_train)
```

```
    param = {
        'objective': 'regression_l2',
        'metric': 'mse',
        'verbosity': -1,
        'boosting_type': 'gbdt',
        'lambda_l1': trial.suggest_loguniform('lambda_l1', 1e-8, 10.0),
        'lambda_l2': trial.suggest_loguniform('lambda_l2', 1e-8, 10.0),
        'num_leaves': trial.suggest_int('num_leaves', 2, 512),
        'learning_rate': trial.suggest_loguniform('learning_rate', 1e-8, 1.0),
        'n_estimators': trial.suggest_int('n_estimators', 700, 3000),
        'feature_fraction': trial.suggest_uniform('feature_fraction', 0.4, 1.0),
        'bagging_fraction': trial.suggest_uniform('bagging_fraction', 0.4, 1.0),
        'bagging_freq': trial.suggest_int('bagging_freq', 1, 7),
        'min_child_samples': trial.suggest_int('min_child_samples', 5, 100),
    }

    gbm = lgb.train(param, dtrain)
    return mean_squared_error(y_test, gbm.predict(X_test))

study = optuna.create_study(direction='minimize', sampler=sampler)
study.optimize(objective, n_trials=100)
lgb_params = study.best_params
print(lgb_params)
```

```
def objective(trial):
    dtrain = xgb.DMatrix(X_train, label=y_train)

    param = {
        'objective': 'reg:squarederror',
        'metric': 'mse',
        'verbosity': 0,
        'boosting_type': 'gbdt',
        'lambda_l1': trial.suggest_loguniform('lambda_l1', 1e-8, 10.0),
        'lambda_l2': trial.suggest_loguniform('lambda_l2', 1e-8, 10.0),
        'num_leaves': trial.suggest_int('num_leaves', 2, 512),
        'learning_rate': trial.suggest_loguniform('learning_rate', 1e-8, 1.0),
        'n_estimators': trial.suggest_int('n_estimators', 700, 3000),
        'feature_fraction': trial.suggest_uniform('feature_fraction', 0.4, 1.0),
        'bagging_fraction': trial.suggest_uniform('bagging_fraction', 0.4, 1.0),
        'bagging_freq': trial.suggest_int('bagging_freq', 1, 7),
        'min_child_samples': trial.suggest_int('min_child_samples', 5, 100),
    }

    gbm = xgb.train(param, dtrain)
    return mean_squared_error(y_test, gbm.predict(xgb.DMatrix(X_test,
label=np.ones_like(y_test))))

study = optuna.create_study(direction='minimize', sampler=sampler)
study.optimize(objective, n_trials=100)
xgb_params = study.best_params
print(xgb_params)
```

4.6 stacking 融合

```
part = '4.6'
```

```python
# 1.基础代码
def stacking_reg(
                clf, train_x, train_y, test_x, clf_name, kf, label_split=None
            ):
    train = np.zeros((train_x.shape[0], 1))
    test = np.zeros((test_x.shape[0], 1))
    test_pre = np.empty((folds, test_x.shape[0], 1))
    cv_scores = []
    for i, (train_index, test_index) in enumerate(kf.split(train_x,
label_split)):
        tr_x = train_x[train_index]
        tr_y = train_y[train_index]
        te_x = train_x[test_index]
        te_y = train_y[test_index]
        if clf_name in ['rf', 'ada', 'gb', 'et', 'lr', 'lsvc', 'knn']:
            clf.fit(tr_x, tr_y)
            pre = clf.predict(te_x).reshape(-1, 1)
            train[test_index] = pre
            test_pre[i, :] = clf.predict(test_x).reshape(-1, 1)
            cv_scores.append(mean_squared_error(te_y, pre))

        elif clf_name in ['xgb']:
            train_mat = clf.DMatrix(tr_x, label=tr_y, missing=-1)
            test_mat = clf.DMatrix(te_x, label=te_y, missing=-1)
            z = clf.DMatrix(test_x, label=np.ones(test_x.shape[0]), missing=-1)
            num_round = 10000
            early_stopping_rounds = 100
            watch_list = [(train_mat, 'train'), (test_mat, 'eval')]
            if test_mat:
                model = clf.train(
                    xgb_params,
                    train_mat,
                    num_boost_round=num_round,
                    evals=watch_list,
                    early_stopping_rounds=early_stopping_rounds
                )
                pre = model.predict(
                    test_mat,
                    ntree_limit=model.best_ntree_limit
                ).reshape(-1, 1)
                train[test_index] = pre
                test_pre[i, :] = model.predict(
                    z,
                    ntree_limit=model.best_ntree_limit
                ).reshape(-1, 1)

                cv_scores.append(mean_squared_error(te_y, pre))

        elif clf_name in ['lgb']:
            train_mat = clf.Dataset(tr_x, label=tr_y)
            test_mat = clf.Dataset(te_x, label=te_y)
            num_round = 7000
            early_stopping_rounds = 100
            if test_mat:
                print(train_mat)
                model = clf.train(
                    lgb_params,
                    train_mat,
```

```
                    num_round,
                    valid_sets=test_mat,
                    # early_stopping_rounds=early_stopping_rounds
                )
                pre = model.predict(
                    te_x,
                    num_iteration=model.best_iteration
                ).reshape(-1, 1)
                train[test_index] = pre
                test_pre[i, :] = model.predict(
                    test_x, num_iteration=model.best_iteration
                ).reshape(-1, 1)
                cv_scores.append(mean_squared_error(te_y, pre))
        else:
            raise IOError('please add new clf.')
        print(f"{clf_name} now score is {cv_scores}")
    test[:] = test_pre.mean(axis=0)
    print(f"{clf_name}_score_list:{cv_scores}")
    print(f"{clf_name}_Score_mean:{np.mean(cv_scores)}")
    return train.reshape(-1, 1), test.reshape(-1, 1)
```

```
# 2 模型融合stacking基学习器
def rf_reg(x_train, y_train, x_valid, kf, label_split=None):
    rf = RandomForestRegressor(
        n_estimators=600,
        max_depth=20,
        n_jobs=-1,
        random_state=2021,
        max_features='auto',
        verbose=1
    )
    train, test = stacking_reg(
        rf, x_train, y_train, x_valid, 'rf', kf, label_split=label_split
    )
    return train, test, 'rf_reg'

def ada_reg(x_train, y_train, x_valid, kf, label_split=None):
    ad = AdaBoostRegressor(
        n_estimators=30,
        random_state=2021,
        learning_rate=.01
    )
    train, test = stacking_reg(
        ad, x_train, y_train, x_valid, 'ada', kf, label_split=label_split
    )
    return train, test, 'ada_reg'

def gb_reg(x_train, y_train, x_valid, kf, label_split=None):
    gbdt = GradientBoostingRegressor(
        learning_rate=.04,
        n_estimators=100,
        subsample=.8,
        random_state=2021,
        max_depth=5,
        verbose=1
    )
    train, test = stacking_reg(
```

```
        gbdt, x_train, y_train, x_valid, 'gb', kf, label_split=label_split
    )
    return train, test, 'gb'


def xgb_reg(x_train, y_train, x_valid, kf, label_split=None):
    train, test = stacking_reg(
        xgb, x_train, y_train, x_valid, 'xgb', kf, label_split=label_split
    )
    return train, test, 'xgb_reg'

def lgb_reg(x_train, y_train, x_valid, kf, label_split=None):
    train, test = stacking_reg(
        lgb, x_train, y_train, x_valid, 'lgb', kf, label_split=label_split
    )
    return train, test, 'lgb_reg'
```

```
# 3 模型融合stacking预测函数
def stacking_pred(
    x_train, y_train, x_valid, kf, clf_list, label_split=None, clf_fin='lgb',
if_concat_origin=True
):
    for k, clf_list in enumerate(clf_list):
        clf_list = [clf_list]
        col_list, train_data_list, test_data_list = [], [], []
        for clf in clf_list:
            train_data, test_data, clf_name = clf(
                x_train,
                y_train,
                x_valid,
                kf,
                label_split=label_split
            )
            train_data_list.append(train_data)
            test_data_list.append(test_data)
            col_list.append(f"clf_{clf_name}")
        train, test = np.concatenate(train_data_list, axis=1),
np.concatenate(test_data_list, axis=1)


        if if_concat_origin:
            train = np.concatenate([x_train, train], axis=1)
            test = np.concatenate([x_valid, test], axis=1)
        print('train:', train.shape)
        print('test:', test.shape)
        print(x_train.shape)
        print(train.shape)
        print(clf_name)
        print(clf_name in ['lgb'])
        if clf_fin in ['rf', 'ada', 'gb', 'et', 'lr', 'lsvc', 'knn']:
            if clf_fin in ['rf']:
                clf = RandomForestRegressor(
                    n_estimators=600,
                    max_depth=20,
                    n_jobs=-1,
                    random_state=2021,
                    max_features='auto',
```

```
                verbose=1
            )
        elif clf_fin in ['ada']:
            clf = AdaBoostRegressor(
                n_estimators=30,
                random_state=2021,
                learning_rate=.01,
            )
        elif clf_fin in ['gb']:
            clf = GradientBoostingRegressor(
                learning_rate=.04,
                n_estimators=100,
                subsample=.8,
                random_state=201,
                max_depth=5,
                verbose=1
            )
    elif clf_fin in ['xgb']:
        clf = xgb
        train_mat = clf.DMatrix(train, label=y_train, missing=-1)
        test_mat = clf.DMatrix(train, label=y_train, missing=-1)
        num_round = 10000
        early_stopping_rounds = 100
        watch_list = [(train_mat, 'train'), (test_mat, 'eval')]
        model = clf.train(
            xgb_params,
            train_mat,
            num_boost_round=num_round,
            evals=watch_list,
            early_stopping_rounds=early_stopping_rounds
        )
        pre = model.predict(
                clf.DMatrix(test),
                ntree_limit=model.best_ntree_limit
            ).reshape(-1, 1)
        return pre

    elif clf_fin in ['lgb']:
        print(clf_name)
        clf = lgb
        train_mat = clf.Dataset(train, label=y_train)
        test_mat = clf.Dataset(train, label=y_train)
        num_round = 10000
        early_stopping_rounds = 1000
        model = clf.train(
            lgb_params,
            train_mat,
            num_round,
            valid_sets=test_mat,
            # early_stopping_rounds=early_stopping_rounds
        )
        print('pred')
        pre = model.predict(
            test,
            num_iteration=model.best_iteration
        ).reshape(-1, 1)
        print(pre)
        return pre
```

```
folds = 5
seed = 1
kf = KFold(n_splits=5, shuffle=True, random_state=2021)
```

```
clf_list = [rf_reg, ada_reg, gb_reg, xgb_reg, lgb_reg]
train_y = y_train.values
pred = stacking_pred(
    X_train.values,
    train_y,
    X_test.values,
    kf,
    clf_list,
    label_split=None,
    clf_fin='lgb'
)
```

```
title = f'{part} Predicted value VS real value'
plt.figure(figsize=(20, 10))
plt.scatter(y_test, pred.reshape(1, -1)[0], s=600)
plt.xlabel('real value', fontsize=20)
plt.ylabel('predicted value', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.title(title, fontsize=20)
file = os.path.join(path, f'{title}.png')
plt.savefig(file, dpi=600)
```

```
preds = pd.concat(
    [y_test.reset_index(),
    pd.Series(pred.reshape(1, -1)[0], name='预测有效时间').reset_index()],
    axis=1
)[['有效时间', '预测有效时间']]

preds
```

更多学习笔记，请关注公众号【杰然不同之GR】