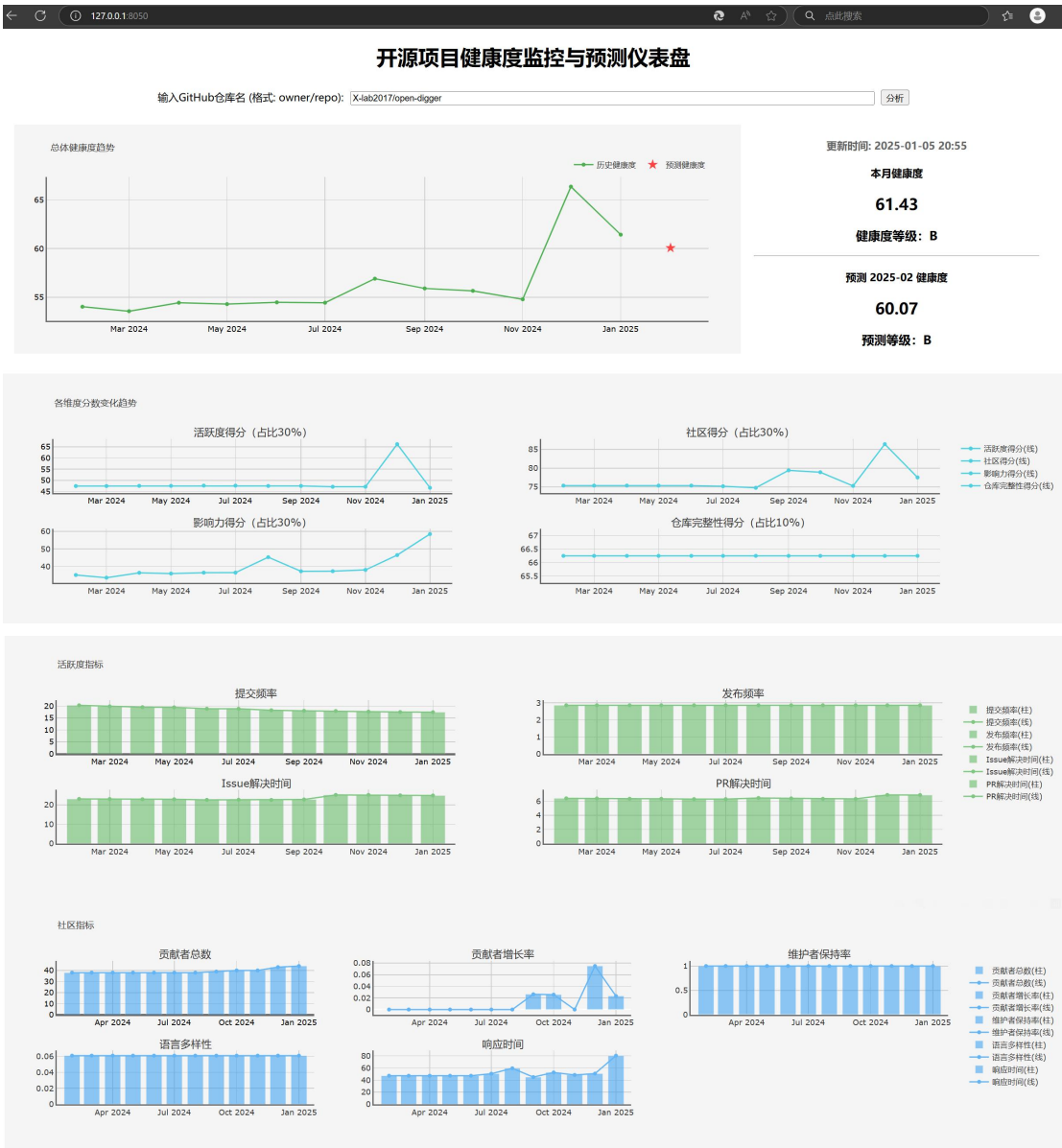


项目健康度实时监控和预测仪表盘设计

10235501435 张凯诚

一.项目简介








本项目是 OpenRank 大赛的参赛作品，主题为“开源项目健康度监控与指标设计”。在本项目中，我首先设计了一套用于计算项目健康度的评分系统，再利用 GitHub API 实时收集评分所需要的各项指标数据，并将指标数据和对应的评分存储到时序数据库 IoTDB 中，然后实时使用到现在为止 12 个月内的各项指标数据建立模型，预测该项目下一个月的健康度，最后用 Dash 库搭建了一个网站，建立了一个可交互的项目健康度实时监控和预测仪表盘。使用示例如下：





二.项目构成

本项目由以下代码文件构成：

 calculator.py	2025/1/5 20:27	Python 源文件
 collector.py	2025/1/5 20:54	Python 源文件
 dashboard.py	2025/1/5 20:55	Python 源文件
 main.py	2025/1/5 21:08	Python 源文件
 models.py	2025/1/3 13:42	Python 源文件
 predictor.py	2025/1/4 10:45	Python 源文件
 storage.py	2025/1/4 21:33	Python 源文件

各代码文件的功能如下：

main.py: 主函数，用于整合各功能。

collector.py: 用于调用 GitHub API，从待分析的仓库中收集需要的各维度的指标数据。

models.py: 储存着一些后续会用到的指标名。

calculator.py: 用于计算项目健康度评分，里面是一套我设计的各相关指标的评分机制。

predictor.py: 存放着用于建模和预测的相关函数。

dashboard.py: 用于构建仪表盘，同时实现了搜索框的功能，可以在搜索框中输入待分析的仓库名

storage.py: 用于实现与数据库存储有关的操作

三.具体实现

1. 数据收集与处理

collector.py 通过调用 GitHub API 分四个维度收集数据，主要函数如下：

(1) 活跃度维度：

_calculate_commit_pattern: 计算 commit 规律性

_get_release_frequency: 计算指定日期前的 release 频率

_get_issue_resolution_time: 计算 issue 平均解决时间

_get_pr_resolution_time: 计算 pr 平均处理时间
collect_activity_metrics: 集成以上四个函数, 收集指定时间点的活跃度指标数据

(2) 社区维度:

_get_total_contributors: 获取指定日期前的贡献者数量
_calculate_contributor_growth: 计算贡献者增长率
_get_maintainer_retention: 计算维护者留存率
_get_language_diversity: 计算语言多样性
_get_average_response_time: 计算平均响应时间
_collect_community_metrics: 集成以上五个函数, 收集指定时间点的社区指标数据

(3) 影响力维度

_get_stars_count: 获取指定日期前的 stars 数量
_calculate_stars_growth: 计算 stars 增长率
_get_forks_count: 获取指定日期前的 forks 数量
_calculate_forks_growth: 计算 forks 增长率
_collect_impact_metrics: 集成以上四个函数, 收集指定时间点的影响力指标数据

(4) 仓库完整性维度

_get_test_coverage: 评估测试覆盖率
_calculate_documentation_score: 评估文档完整性
_evaluate_cicd: 评估 CI/CD 完整性
_collect_code_quality_metrics: 集成以上三个函数, 收集指定时间点的代码质量指标数据

*功能比较直观的函数的逻辑在这里不详细介绍了, 只介绍其中几个不直观的函数:

<1> _calculate_commit_pattern 计算 commit 规律性

目的: 计算代码提交模式的规律性得分(0-1 分), 主要考虑两个维度:

1. 工作日/周末提交的平衡性 (40%权重)
2. 提交时间间隔的稳定性 (60%权重)

逻辑步骤:

1. 数据预处理
2. 过滤目标日期之前的提交, 如果没有有效提交返回 0 分
3. 计算工作日平衡得分 (40%权重): 计算工作日提交占比, 与理想比例(5/7)对比, 差距越小分数越高, $\text{得分} = 0.4 * (1 - |\text{实际工作日比例} - 5/7|)$
4. 计算时间间隔规律性得分 (60%权重): 计算相邻提交之间的天数间隔, 计算间隔的标准差使用 $\exp(-\text{标准差}/30)$ 将结果归一化, $\text{得分} = 0.6 * \exp(-\text{标准差}/30)$
5. 计算最终得分 = 工作日平衡得分 + 时间间隔规律性得分
6. 这个算法倾向于奖励: 工作日/周末提交比例接近 5:2 且提交间隔稳定的项目

<2> _get_test_coverage : 评估测试覆盖率

目的: 评估项目的测试覆盖情况, 返回 0-1 之间的分数

逻辑步骤:

1. 检查仓库是否有测试目录(tests/test 等), 有则基础分 0.5 分

- 2.检查是否有 CI 测试配置文件(.travis.yml 等),每有一个加 0.1667 分
- 3.最终分数为基础分+CI 配置分,上限为 1 分

<3>_calculate_documentation_score :评估文档完整性

目的:评估项目文档的完整性,返回 0-1 之间的分数

逻辑步骤:

- 1.检查 4 个关键文档位置:README.md、CONTRIBUTING.md、docs/目录、wiki
- 2.每存在一个文档位置加 0.25 分
- 3.最终分数为所有存在文档的得分之和

<4>_evaluate_cicd 函数:评估 CI/CD 完整性

目的:评估项目 CI/CD 的完整性,返回 0-1 之间的分数

逻辑步骤:

- 1.获取项目的 GitHub Actions 工作流配置
- 2.检查工作流名称中是否包含 build/test/deploy/release 这些关键字
- 3.最终分数 = 找到的关键字数量 / 总期望关键字数量(4 个)

<5>_get_language_diversity:计算语言多样性

目的:度量仓库使用编程语言的丰富程度和均衡性。

逻辑:通过 GitHub API 获取仓库所有编程语言的代码量统计,计算每种语言的占比,用熵计算多样性指标,将结果归一化到 0-1 范围内,0 表示单一语言,1 表示多语言均匀分布

最后,用 collect_monthly_data 函数并发收集到运行时间为止的 12 个月内每个月 1 号 00:00:00 的各指标数据:

```
# 预获取仓库信息
repo_info = await self._cached_request(f"https://api.github.com/repos/{repo}")
if not repo_info:
    raise Exception("Failed to get repo info")

dates = [
    (now.replace(day=1) - relativedelta(months=i))
    .replace(hour=0, minute=0, second=0)
    for i in range(12)
]

# 并发收集所有指标数据
tasks = []
for date in dates:
    tasks.extend([
        self.collect_activity_metrics(repo, date),
        self.collect_community_metrics(repo, date),
        self.collect_impact_metrics(repo, date),
        self.collect_code_quality_metrics(repo, date),
    ])

all_results = await asyncio.gather(*tasks)

# 处理结果 - 每四个结果为一组
for i in range(0, len(all_results), 4):
    activity_data = all_results[i]
    community_data = all_results[i+1]
    impact_data = all_results[i+2]
    quality_data = all_results[i+3]

    month_index = i // 4

    data.append({
        'repo': repo,
        'date': dates[month_index].strftime('%Y-%m-%d'),
        **activity_data,      # 展开活跃度数据
        **community_data,    # 展开社区指标数据
        **impact_data,       # 展开影响力指标数据
        **quality_data,      # 展开仓库完整性数据
    })
```

2. 健康度分数评分机制设计

calculator.py 中存放着我设计的健康度分数评分机制，总体健康度分数和各维度的分数满分都是 100 分，不同分数对应的等级如下：

```
def get_health_grade(score: float) -> str:
    """根据得分返回健康度等级"""
    if score >= 90:
        return 'A+'
    elif score >= 80:
        return 'A'
    elif score >= 70:
        return 'B+'
    elif score >= 60:
        return 'B'
    elif score >= 40:
        return 'C'
    else:
        return 'D'
```

各维度分数在总健康度分数中的权重如下：

```
WEIGHTS = {
    'community': 0.30,    # 社区基础
    'activity': 0.30,     # 活跃程度
    'impact': 0.30,      # 技术影响力
    'code_quality': 0.10, # 仓库完整性得分
}
```

在每个维度中，不同指标的分数又有不同的权重：

(1) 在活跃度维度中：

```
weights = {
    'commit_frequency': 0.25,
    'release_frequency': 0.25,
    'issue_resolution_time': 0.25,
    'pr_resolution_time': 0.25,
}
```

各指标分数的计算逻辑如下：

```
scores = {
    'commit_frequency': min(metrics['commit_frequency'] * (100/15), 100), # 15次/月满
    'release_frequency': min(metrics['release_frequency'] * (100/20), 100), # 20次/月满
    'issue_resolution_time': 100 * (1 / (1 + metrics['issue_resolution_time']/7)), # 7天内处理最佳
    'pr_resolution_time': 100 * (1 / (1 + metrics['pr_resolution_time']/7)), # 7天内处理最佳
}
```

(2) 在社区维度中：

```
weights = {
    'maintainer_retention': 0.35,
    'contributor_growth': 0.15,
    'total_contributors': 0.35,
    'response_time': 0.10,
    'language_diversity': 0.05
}
```

各指标分数的计算逻辑如下：

```
scores = {
    'maintainer_retention': metrics['maintainer_retention'] * 100,
    'contributor_growth': min(metrics['contributor_growth'] * 1000, 100), # 10%增长率满分
    'total_contributors': min(math.log(max(metrics['total_contributors'], 1), 2) * 20, 100), # 64人达到满分，32人80分，16人60分。
    'response_time': 100 * (1 / (1 + metrics['response_time']/48)), # 48小时内响应最佳
    'language_diversity': metrics['language_diversity'] * 100
}
```

(3) 在仓库完整性维度中:

```
weights = {
    'documentation_score': 0.35,
    'test_coverage': 0.35,
    'ci_cd_score': 0.30
}
```

各指标分数的计算逻辑如下:

```
scores = {
    'documentation_score': metrics['documentation_score'] * 100,
    'test_coverage': metrics['test_coverage'] * 100,
    'ci_cd_score': metrics['ci_cd_score'] * 100
}
```

(4) 在影响力维度中:

```
weights = {
    'stars_growth': 0.20,
    'forks_growth': 0.20,
    'stars_count': 0.30,
    'forks_count': 0.30
}
```

各指标分数的计算逻辑如下:

```
scores = {
    # 增长率评分
    'stars_growth': min(metrics['stars_growth'] * 1000, 100), # 10%增长率满分
    'forks_growth': min(metrics['forks_growth'] * 1000, 100), # 10%增长率满分
    # 数量评分基准
    'stars_count': min(math.log(max(metrics['stars_count'], 1), 10) * 25, 100), # 得分 = min(log10(指标数量) * 25, 100)
    'forks_count': min(math.log(max(metrics['forks_count'], 1), 10) * 25, 100)
}
```

3. 数据库的链接与数据存储

在本项目中, 数据存储使用的是时序数据库 IoTDB, 在 storage.py 实现了如下操作:

1. 设计了 query_metrics 函数, 用于在 IoTDB 数据库中查询时间范围内每个月的 1 号 00:00:00 的指标数据

```
def query_metrics(self, repo: str, start_time: datetime, end_time: datetime) -> Dict:
    """查询指标数据"""
```

2. 设计了 view_all_data 函数, 用于在 IoTDB 数据库中查询特定时间范围内的所有数据, 这个函数可以用来测试数据库是否正确存储数据

```
def view_all_data(self, repo: str, start_time: datetime, end_time: datetime):
    """查看指定时间范围内的所有数据"""
```

3. 设计了 store_metrics 函数, 这是本代码文件中最核心的操作, 用于将时间范围内每个月 1 号 00:00:00 的各项指标数据存储到 IoTDB 数据库中

```
def store_metrics(self, repo: str, metrics: Dict[str, Any], timestamp: datetime):
    try:
```

4. 设计了 clear_all_data 函数, 这个函数用于错误处理, 当数据库发生某种存储错误时, 可以在主函数中调用这个函数来清空 IoTDB 数据库中的所有数据

```
def clear_all_data(self):
    """清空数据库中的所有数据"""
```


4. 建模与预测

predictor.py 中是建模和预测部分的代码，核心思路是：使用本月及本月之前的 11 个月（共 12 个月）的各指标数据分别建立模型，预测出下一个月的各项指标的大小，再根据这些指标的大小使用 calculator.py 中的逻辑计算出预测的下个月的健康度分数。

预测策略相关代码如下：

```
def _predict_trend(self, data: pd.Series, metric_name: str) -> float:
    """根据指标类型使用不同的预测策略"""
    try:
        if len(data) < 2:
            return float(data.iloc[-1]) if len(data) > 0 else 0.0

        # 数据类型转换和清理
        data = pd.to_numeric(data, errors='coerce')
        data = data.dropna()

        if len(data) < 2:
            return float(data.iloc[-1]) if len(data) > 0 else 0.0

        # 累积型指标列表
        cumulative_metrics = ['total_contributors', 'stars_count', 'forks_count']

        if metric_name in cumulative_metrics:
            # 对于累积型指标，确保预测值不小于最后一个实际值
            last_value = float(data.iloc[-1])
            # 使用最近3个月的平均增长量
            recent_growth = data.diff().tail(3).mean()
            return max(last_value + recent_growth, last_value)
        else:
            # 其他指标使用线性回归
            X = np.arange(len(data), dtype=np.float64).reshape(-1, 1)
            y = np.array(data.values, dtype=np.float64)

            try:
                A = np.vstack([X.ravel(), np.ones(len(X))]).T
                slope, intercept = np.linalg.lstsq(A, y, rcond=None)[0]
                next_value = slope * len(data) + intercept
                return max(0.0, float(next_value))
            except np.linalg.LinAlgError:
                return float(data.iloc[-1])

    except Exception as e:
        logger.error(f"趋势预测失败: {str(e)}")
        return float(data.iloc[-1]) if len(data) > 0 else 0.0
```

可以看到，对于不同类型的指标数据，我采用了不同的预测方法：

1. 对于累积型指标，如总贡献者数量、stars 数量、forks 数量，要确保预测值不小于最后一个实际值：

预测方式如下：预测值 = max(最后一个月的值 + 近 3 个月的平均增长量, 最后一个月的值)

2. 对于非累积型指标，如贡献者增长率、forks 增长率：

预测方式如下：

```
# 其他指标使用线性回归
X = np.arange(len(data), dtype=np.float64).reshape(-1, 1)
y = np.array(data.values, dtype=np.float64)
```

使用线性回归进行预测，其中：

X: 时间序列索引

y: 实际值

使用最小二乘法计算斜率和截距，从而预测下一个时间点的值

接着在 predict_next_month 函数中使用 12 个月的数据来预测下个月的各项指标的大小：

```
# 获取当前时间和时间范围
now = datetime.now(timezone.utc)
end_time = now.replace(day=1, hour=0, minute=0, second=0, microsecond=0)
start_time = end_time - relativedelta(months=12)

# 查询历史数据
metrics = self.storage.query_metrics(repo, start_time, end_time)

if not metrics:
    raise Exception("没有足够的历史数据用于预测")

# 转换为DataFrame并清理列名
df = pd.DataFrame.from_dict(metrics, orient='index')
df.index = pd.to_datetime(df.index, unit='ms')
df.columns = df.columns.str.split('.').str[-1]

# 预测下个月的指标
next_month = end_time + relativedelta(months=1)
predicted_metrics = {}

# 整数类型的指标
integer_metrics = ['total_contributors', 'stars_count', 'forks_count']

for column in df.columns:
    if column != 'repo_name':
        predicted_value = self._predict_trend(df[column], column)
        # 对整数类型指标取整
        if column in integer_metrics:
            predicted_value = int(round(predicted_value))
        predicted_metrics[column] = predicted_value
```

并将各项指标的大小及分数、总健康度及分数等数据存到数据库中：

```
# 计算健康度分数
scores = self.calculator.calculate_total_score(health_metrics)

# 合并预测指标和分数
complete_metrics = {
    **activity_metrics,
    **community_metrics,
    **impact_metrics,
    **code_quality_metrics,
    'repo_name': repo
}

# 存储预测指标
self.storage.store_metrics(repo=repo, metrics=complete_metrics, timestamp=next_month)

# 存储预测分数
score_data = {
    'total_score': scores['total_score'],
    'health_grade': scores['health_grade'],
    **{f"{dim}_score": details['score'] for dim, details in scores['dimensions'].items()},
    **{f"{dim}_grade": details['grade'] for dim, details in scores['dimensions'].items()}
}

self.storage.store_metrics(
    repo=f"{repo}_scores",
    metrics=score_data,
    timestamp=next_month
)

# 保存完整预测结果
self.predicted_metrics = {
    'timestamp': next_month,
    'repo_name': repo,
    **complete_metrics,
    **score_data
}
```


5. 设计可交互仪表盘

为了将各项指标大小及各维度分数的变化趋势可视化，且为了满足可交互性的需求，我用python的dash库设计了一个简单的网站，实现了“搜索指定仓库”、“实时分析仓库”和“创建可交互图表”的功能。

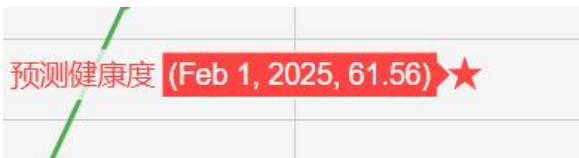
布局如下：

1. 最上方是一个搜索框，可以在这里输入你想要分析的仓库的名称，然后点击“分析”键，这里我以最近常见的一个项目：opendigger 为例。

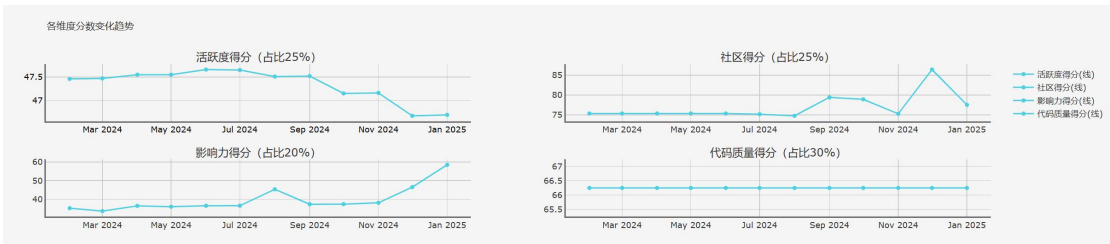
开源项目健康度分析与预测仪表盘



和下面的所有图表一样, 这个图表是可视化的, 将鼠标拖动到其中的点上可以查看详细信息, 比如：



在健康度图表下面的是各维度评分的变化趋势图表：

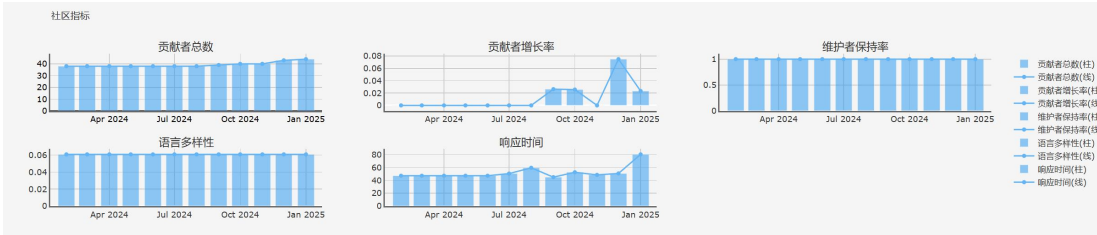


在各维度的分数图表后面的是四个维度各指标的变化趋势图标，依次是：

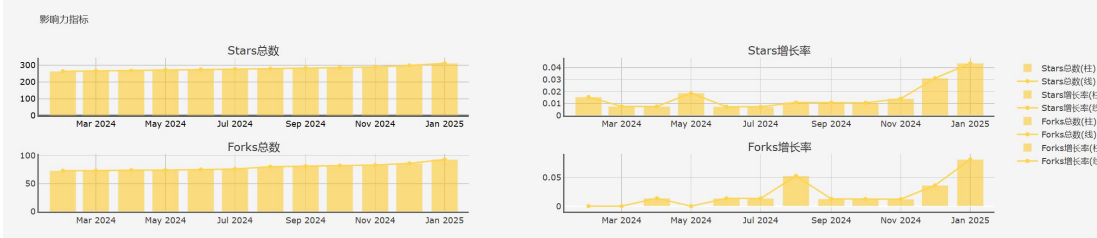
活跃度维度：



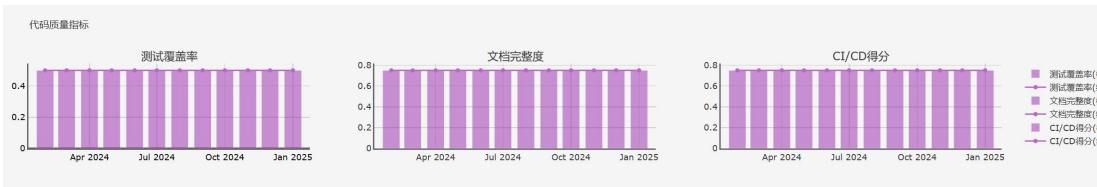
社区维度：



影响力维度：



代码质量维度：



6. 主函数

主函数除了集成以上各部分以外，在数据取用方面还实现了以下逻辑：当用户输入一个仓库名时，主函数会首先查看 IoTDB 数据库中是否已经存储了该仓库的数据，如果有，则会跳过收集的过程，直接将数据库中的相关信息传入 dashboard.py：

```
INFO:dashboard:开始处理仓库：X-lab2017/open-digger
INFO:storage:成功连接到IoTDB
INFO:main:跳过已存在数据：X-lab2017/open-digger at 2024-02
INFO:main:跳过已存在数据：X-lab2017/open-digger at 2024-03
INFO:main:跳过已存在数据：X-lab2017/open-digger at 2024-04
INFO:main:跳过已存在数据：X-lab2017/open-digger at 2024-05
INFO:main:跳过已存在数据：X-lab2017/open-digger at 2024-06
```

若数据库中无待查询仓库的信息，则会启动 collector.py，开始收集该仓库的各项数据：

```
INFO:storage:成功存储月度指标数据：X-lab2017/open-digger at 2024-09
INFO:storage:成功存储月度指标数据：X-lab2017/open-digger_scores at 2024-09
INFO:main:完成 X-lab2017/open-digger 在 2024-09 的数据处理
```

四.项目实例

实例：监控和预测 Easy-Graph 的健康度

- 1.查询 Easy-Graph 的开发者或机构，发现是 easy-graph，于是我们便得到了用于搜索的完整项目名:easy-graph/Easy-Graph
- 2.启动 lotdb 数据库，在终端输入环境变量 set GITHUB_TOKEN=你的 token,token 从 github 中创建，这是用于访问 github 仓库的秘钥
- 3.运行 main.py，这时终端会告诉你仪表盘所在的网址

Dash is running on <http://127.0.0.1:8050/>

```
INFO:__main__:仪表盘已启动, 访问 http://localhost:8050 查看
INFO:dash.dash:Dash is running on http://127.0.0.1:8050/
```

- 4.进入该网址，在搜索框中输入你想分析的项目名称(这里是 easy-graph/Easy-Graph)

开源项目健康度监控与预测仪表盘

输入GitHub仓库名 (格式: owner/repo):

分析

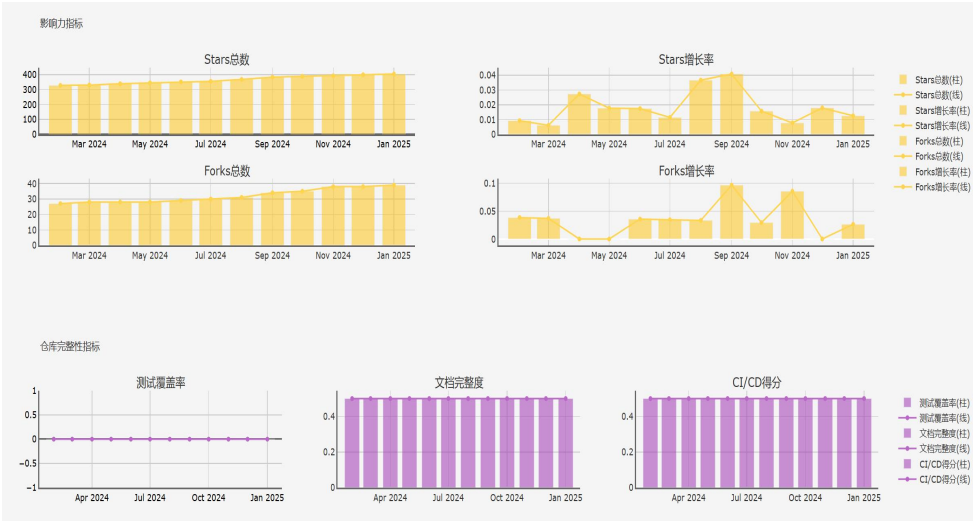
5. 点击“分析”，如果是第一次分析该项目，则加载时间会较长；若已经分析过该项目，则会立刻给出反应
6. 加载完成后，即可展示该项目的健康度信息

开源项目健康度监控与预测仪表盘

输入GitHub仓库名 (格式: owner/repo):

分析





五. 总结与思考

本项目为分析与预测开源项目的健康度提供了一些个人的想法, 包括设计了一套健康度评分系统、设计了简单的建模和预测下一个月数据的方法、构建了实时的可交互仪表盘等。但仍有很多可以进一步优化的部分, 比如可以设计更精确的预测方式、通过大量数据评估并设计更合理的分数计算方式、进一步优化代码来加快数据收集的速度等。