本地数据文件系统构建与数据处理研究报告

曹芷慧

2025/1

摘要

本研究聚焦于构建本地数据文件系统，实现数据的增量更新、加载，对数据进行简单分析与可视化，并采用本地 Git 进行版本控制。阐述了研究背景、意义，详细介绍了系统构建、数据加载器设计、分析可视化过程及版本控制应用，最后总结成果并指出不足与展望。

1 引言

在当今数字化时代，数据已成为推动各行业发展的核心要素。随着数据量的迅猛增长以及业务需求的日益复杂，如何高效地管理和利用数据成为了关键问题。构建本地数据文件系统应运而生，它为数据的存储、管理和访问提供了一种可靠且灵活的解决方案。

本研究旨在构建一个功能完备的本地数据文件系统，实现数据的增量更新，并通过数据加载器按特定条件从本地文件系统导入数据。这不仅有助于提高数据管理的效率，减少数据处理的时间和资源消耗，还能为后续的数据分析和决策提供有力支持。

2 本地数据文件系统构建

2.1 目录结构设计

在构建本地数据文件系统时，设计了包含 data 和.cache 的文件夹结构。data 文件夹的主要作用是存放更新合并后的数据，这些数据是经过处理和整合的最终数据，对于后续的数据加载和分析具有重要意义。

而.cache 文件夹则用于存放缓存更新文件，缓存更新文件是在数据更新过程中产生的临时文件，用于存储尚未正式合并到 data 文件夹中的新数据或更新数据。这些文件在更新完成后会被清除，以释放存储空间并确保数据的一致性和整洁性。

2.2 增量数据更新实现

2.2.1 数据更新原理

在本研究中，数据更新的判断依据主要基于文件的时间戳。时间戳是记录文件创建或修改时间的数字标记，通过对比新获取数据文件的时间戳与 data 文件夹中已有数据文件的时间戳，可以判断数据是否为新数据或更新数据。如果新数据文件的时间戳晚于已有数据文件的时间戳，则认为该数据是更新数据，需要进行增量更新操作。此外，在一些情况下，也可以通过版本号来判断数据更新，例如在数据文件中添加版本号字段，每次数据更新时更新版本号，通过对比版本号来确定数据是否需要更新。

2.3 具体实现步骤

在提供的代码中，通过 process_files 函数实现数据的处理和分类，将与股票相关的数据和其他数据分别存储。然后，通过 save_json 函数将处理后的数据保存到相应的文件中。

在实现增量数据更新时,首先获取新的数据文件路径,例如 initial_file_paths_2 ,然后调用 process_files 函数对新数据文件进行处理，得到股票相关数据 stock_data_2 和其他数据 other_data_2 。接着，将 stock_data_2 保存到 `data` 文件夹中，文件名为 2024 – 12 – 10.json ，将 other_data_2 保存到 `.cache` 文件夹中，文件名为 2024 – 12 – 10.cache.json 。通过这种方式，实现了新数据的增量更新，将新数据与已有数据进行区分和存储，为后续的数据加载和分析提供了便利。

3 数据加载器构建

3.1 DataLoader 类设计

3.1.1 类的初始化

在DataLoader类的初始化方法___init___中，接收一个参数data_directory，并将其赋值给实例变量self .data_directory。这一步骤的作用是确定数据加载的来源目录，即指定从哪个本地目录中获取数据文件。通过初始化数据目录，DataLoader类能够明确数据的存储位置，为后续的数据加载方法提供必要的路径信息。例如，在金融数据处理场景中，如果data_directory被设置为存放股票数据的目录，那么DataLoader类就可以从该目录中加载股票数据文件，为按时间戳或股票名称导入数据提供基础。

3.1.2 数据加载方法

数据加载方法load_data用于从指定的数据目录中加载数据。该方法接收一个参数filename，表示要加载的数据文件名。在方法内部，首先通过os.path.join函数将数据目录self .data_directory和文件名filename拼接成完整的文件路径file_path，即：

$$\text{file\_path} = \text{os.path.join}(\text{self .data\_directory}, \text{filename})$$

然后，使用os.path.exists函数检查该文件路径是否存在，可表示为：

$$\text{if os.path.exists}(\text{file\_path}):$$

如果文件存在，则打开文件并使用json.load函数将文件内容解析为 JSON 格式的数据：

$$\text{with open(file\_path, 'r') as f: data = json.load(f)}$$

接着，检查解析后的数据是否为列表格式，如果是，则将其转换为 pandas 的 DataFrame 格式并返回；如果不是列表格式，则抛出ValueError异常，提示 JSON 数据格式不正确：

$$\text{if isinstance(data, list): import pandas as pd return pd.DataFrame(data) else: raise ValueError(''}$$

如果文件不存在，则抛出FileNotFoundError异常，提示文件未找到：

$$\text{else: raise FileNotFoundError(f '' \{file\_path\} '')}$$

通过这样的流程，确保了数据加载的准确性和稳定性，能够处理文件不存在、数据格式错误等常见问题。

3.2 导入数据实现

3.2.1 按时间戳筛选逻辑

在代码中，按时间戳导入数据的筛选逻辑主要通过 filter_by_publish_time 方法实现。该方法接收一个 DataFrame 对象 df 和一个时间戳字符串 publish_time 作为参数。在方法内部，使用

df['publishTime'].str.contains(publish_time) 语句来筛选出 publishTime 列中包含指定时间戳的行。这里利用了 pandas 的字符串操作方法 str.contains，它会在指定列的每个元素中查找是否包含给定的字符串。如果包含，则该行数据会被保留，最终返回一个只包含符合时间戳条件的 DataFrame 对象。通过这种方式，能够快速准确地从大量数据中筛选出特定时间戳的数据，满足按时间戳导入数据的需求。

3.2.2 按股票名称匹配逻辑

按股票名称导入数据的名称匹配逻辑主要通过 filter_by_match 方法实现。该方法接收一个 DataFrame 对象 df 和一个股票名称字符串 match_name 作为参数。在方法内部，使用

df['match'].str.contains(match_name) 语句来筛选出 match 列中包含指定股票名称的行。与按时间戳筛选类似，这里同样利用了 pandas 的 str.contains 方法，在 match 列的每个元素中查找是否包含给定的股票名称字符串。如果包含，则该行数据会被保留，最终返回一个只包含符合股票名称条件的 DataFrame 对象。通过这种方式，实现了根据股票名称从数据中筛选出相关数据的功能，满足按股票名称导入数据的需求。

4 数据简单分析与可视化

4.1 数据简单分析

4.1.1 分析思路与方法

本研究对数据进行分析的主要目的是从股票相关数据中提取有价值的信息，以便更好地理解股票市场动态和相关趋势，为投资者或相关研究提供数据支持。在分析过程中，采用了以下方法：

- 筛选与分类：通过 is_stock_related 函数检查文本中是否包含与股票相关的关键词，如“上涨”“下跌”“股”“%”等，将数据分为股票相关数据和其他数据。这种分类有助于专注于股票领域的数据，排除无关信息的干扰，提高分析效率。
- 数据整合：利用 process_files 函数处理多个文件路径下的文件，将不同文件中的数据整合到一起，并根据股票相关性进行分类存储。通过这种方式，将分散的数据集中起来，为后续的统一分析提供基础。
- 条件筛选：在 DataLoader 类中，实现了按时间戳和股票名称筛选数据的方法。通过 filter_by_publish_time 方法，根据 publishTime 列筛选出特定时间戳的数据；通过 filter_by_match 方法，根据 match 列筛选出包含指定股票名称的数据。这种条件筛选能够从大量数据中提取出符合特定条件的数据子集，满足不同的分析需求。

4.1.2 分析结果展示

以具体数据展示分析结果，在处理完 initial_file_paths_1 和 initial_file_paths_2 中的文件后，得到了不同日期的股票相关数据文件，如 2024 – 12 – 04.json 和 2024 – 12 – 10.json。通过 DataLoader 类加载这些数据文件，并进行合并和筛选操作。例如，按时间戳筛选出 2024 – 12 – 04 的数据，或者按股票名称筛选出特定股票的数据，得到相应的 DataFrame。从这些筛选后的数据中，可以观察到以下特征：

- 时间趋势：通过分析不同时间戳的数据，可以观察到股票相关信息的发布时间分布情况。例如，某些时间段内股票相关信息的发布量可能较高，这可能与市场的活跃程度、重大事件的发生等因素有关。
- 股票名称匹配：按股票名称筛选数据后，可以针对特定股票进行深入分析。了解该股票在不同时间的相关信息，如价格波动、市场评论等，有助于把握该股票的市场表现和投资者关注焦点。

4.2 可视化工具的选择

Dash 是一个用于构建交互式数据仪表盘的 Python 框架，基于 Flask、Plotly.js 和 React.js 技术栈，具有数据可视化能力强、响应式布局、交互功能丰富、后端支持良好等特点。在本研究中，Dash 能够很好地满足数据可视化的要求。在数据加载和处理方面，通过 DataLoader 类从本地数据文件系统中按特定条件加载数据，得到的 DataFrame 数据可以直接与 Dash 进行集成。例如，将加载和筛选后的数据传递给 Dash 的图表组件，用于生成可视化图表。在交互性方面，Dash 的回调函数机制可以实现根据用户输入的股票名称或时间戳实时更新图表和数据展示。用户在输入框中输入股票名称或时间戳，点击提交按钮后，回调函数能够根据用户输入筛选数据，并更新可视化图表，展示符合条件的数据分布和趋势，为用户提供了灵活的数据探索和分析方式。

4.3 可视化实现

4.3.1 可视化布局设计

在 Dash 应用中，布局设计如下：

```
app.layout = html.Div(children=[
    html.H1(children='Stock News Analysis'),

    html.Div(children='''
            Input a stock name or a publish time to filter the data.
        '''),

    dcc.Input(id='stock-name-input', type='text', placeholder='Enter Stock Name'),
    dcc.Input(id='publish-time-input', type='text', placeholder='Enter Publish Time (e

    html.Button('Submit', id='submit-button', n_clicks=0),

    dcc.Graph(id='filtered-data-graph'),
    html.Div(id='output-message'),
```

```
    html.Div(id='details-list')
])
```

设计思路是提供一个简洁明了的界面，方便用户进行数据筛选和查看可视化结果。
各组件的功能如下：

- html.H1 组件显示标题 “Stock News Analysis”，突出应用的主题。
- html.Div 组件中的提示文本 “Input a stock name or a publish time to filter the data.” 告知用户可以输入股票名称或发布时间来筛选数据。
- dcc.Input 组件创建了两个输入框，分别用于输入股票名称和发布时间，id 属性为后续的回调函数提供了标识。
- html.Button 组件创建了提交按钮，用户点击该按钮后触发数据筛选和图表更新操作。
- dcc.Graph 组件用于显示可视化图表，展示筛选后的数据分布情况。
- html.Div 组件（id 为 output−message）用于显示筛选结果的提示信息，告知用户当前展示的数据条件。
- html.Div 组件（id 为 details−list）用于显示具体字段的详细信息，让用户能够查看筛选数据的具体内容。

4.3.2 回调函数实现

回调函数实现交互的核心代码如下：

```
@app.callback(
    [Output('filtered-data-graph', 'figure'),
     Output('output-message', 'children'),
     Output('details-list', 'children')],
    Input('submit-button', 'n_clicks'),
    Input('stock-name-input', 'value'),
    Input('publish-time-input', 'value')
)
def update_graph(n_clicks, stock_name, publish_time):
    if n_clicks > 0:
        filtered_df = combined_df

        if stock_name:
            filtered_df = data_loader.filter_by_match(filtered_df, stock_name)
        if publish_time:
            filtered_df = data_loader.filter_by_publish_time(filtered_df, publish_time

        if filtered_df.empty:
            return px.histogram(), "No results found for the given criteria.", "No det
```

```
        fig = px.histogram(filtered_df, x='publishTime', title='Filtered Publish Time
                           labels={'publishTime': 'Publish Time'}, color='match')

        unique_details = set()
        for _, row in filtered_df.iterrows():
            detail = (f"Website: {row['captureWebsite']}, "
                      f"Text: {row['text']}, "
                      f"Followers: {row['followers']}, "
                      f"Emotion: {row['emotion']}")
            unique_details.add(detail)

        details_display = html.Ul([html.Li(detail) for detail in unique_details])

        return fig, f"Showing results for stock name: {stock_name if stock_name else '

    return px.histogram(), "Please enter a stock name or publish time and click Submit
```

当用户点击提交按钮（submit−button）时，n_clicks 值增加，触发回调函数 update_graph。回调函数接收 n_clicks、stock_name（用户输入的股票名称）和 publish_time（用户输入的发布时间）作为输入参数。首先，根据用户输入对 combined_df 数据进行筛选。如果用户输入了股票名称，则调用 data_loader.filter_by_match 方法按股票名称筛选数据；如果用户输入了发布时间，则调用 data_loader.filter_by_publ 方法按时间戳筛选数据。如果筛选后的数据为空，则返回空的直方图、提示信息和无详细信息提示。否则，使用 plotly.express 的 px.histogram 函数创建直方图，展示筛选后数据的发布时间分布情况。同时，从筛选后的数据中提取关键字段信息，如网站、文本、关注者和情感等，去重后生成详细信息列表，并通过 html.Ul 和 html.Li 组件展示在页面上。最后，返回生成的图表、筛选结果提示信息和详细信息列表，实现数据筛选和图表更新的交互功能。

4.3.3 可视化结果展示

图 1: Dash 界面

最终实现的可视化图表能够清晰地展示不同条件下数据的分布情况。例如，当用户输入股票名称为“迦南智能”时，图表展示了“迦南智能”在不同时间戳下的数据分布，并且在“details-list”区域显示

了相关数据的详细信息，如网站、文本内容、关注者数量和情感倾向等。

5 本地 Git 版本控制

5.1 Git 应用

在本研究过程中，每次完成一项重要任务后，都会进行 Git 提交操作。在完成本地数据文件系统构建后，包括目录结构的创建和增量数据更新功能的实现，使用 git add. 命令将所有相关文件添加到暂存区，然后执行 git commit -m ” 完成本地数据文件系统构建，包括目录结构和增量更新功能” 命令进行提交。这样，在 Git 的版本历史中就记录了这一阶段的工作成果，方便后续回溯和查看。在构建数据加载器时，当完成 DataLoader 类的设计和按时间戳、股票名称导入数据功能的实现后，同样先使用 git add. 将代码文件添加到暂存区，再通过 git commit -m ” 完成数据加载器构建，实现按时间戳和股票名称导入数据功能” 进行提交。在进行数据简单分析与可视化工作时，完成数据筛选、分析以及 Dash 可视化界面的设计和回调函数的实现后，执行 git add. 和 git commit -m ” 完成数据简单分析与可视化，包括数据筛选、Dash 界面设计和回调函数实现” 的操作，将这一阶段的工作成果保存到版本控制系统中。

5.2 版本管理的优势

在代码管理方面，通过 Git 的版本控制，可以清晰地记录代码的每一次修改，包括修改的内容、时间和作者等信息。这使得在后续的开发过程中，如果发现代码存在问题，可以方便地查看历史版本，找到问题出现的源头，从而快速进行修复。同时，版本控制也方便了代码的回溯，当需要回退到之前的某个稳定版本时，可以通过 git checkout 命令轻松实现，确保代码的稳定性和可靠性。

从研究过程回溯的角度来看，版本控制为研究提供了详细的历史记录。在研究过程中，可能会对数据处理方法、分析思路或可视化方式进行多次尝试和调整，通过 Git 的提交记录，可以清晰地看到整个研究过程的演进，了解每个阶段所做的工作和决策。这对于研究的总结和反思非常有帮助，也便于向他人展示研究的过程和成果，提高研究的可重复性和可信度。

6 总结与展望

6.1 研究成果总结

在本研究中，成功构建了本地数据文件系统，通过合理设计目录结构，实现了数据的有效存储和管理。其中 data 文件夹用于存放更新合并后的数据，.cache 文件夹用于存放缓存更新文件（更新后清除），并通过代码实现了增量数据更新功能，确保了数据更新的高效性和准确性。

构建了 DataLoader 类作为数据加载器，实现了按时间戳或匹配股票名称以 DataFrame 格式从本地数据文件系统导入数据的功能。这为后续的数据处理和分析提供了灵活的数据获取方式，满足了不同场景下对数据的需求。

利用 Dash 工具对数据进行了简单分析并可视化，通过设计合理的布局和实现交互功能，将数据中的信息以直观的图表和详细信息列表的形式呈现给用户。用户可以通过输入股票名称或时间戳筛选数据，实时查看筛选后的数据分布和详细内容，便于对股票数据进行深入分析和理解。

在整个研究过程中，使用本地 Git 进行版本控制，每次完成一项重要任务后都进行提交操作。通过 git init 初始化仓库，使用 git add 和 git commit 命令对文件进行添加和提交，确保了代码的可追溯性

和稳定性，方便了研究过程的管理和回溯。

6.2 研究不足

在数据处理方面，当前的增量更新和数据加载逻辑虽然能够满足基本需求，但在处理大规模数据时，可能存在性能瓶颈。例如，在数据量较大时，按时间戳或股票名称筛选数据的速度可能会变慢，影响数据分析的效率。

在可视化方面，目前的可视化界面虽然实现了基本的交互功能，但在可视化效果和用户体验上还有提升空间。例如，图表的样式和颜色可以更加丰富和美观，以增强数据展示的吸引力；界面的布局可以进一步优化，使其在不同设备上的显示效果更加友好。