Разбор кода извлечения данных в БД

Наша задача была наполнить данными БД. Данные мы брали из двух источников:

1. Семантика имени файла

Структура наименования файла была рассмотрена в отдельном файле. Если кратко пройтись, то имя файла содержит в себе те данные, которые программа не записывает в качестве метаданных и само имя файла всегда имеет следующий вид:

<technique>_<instrument>_<sample_block>_<mode_block>_<replicate>.<ext>,

где sample_block всегда одинаковой внутренней структуры:

matrix-analyte-concValue concUnit-solvent-physicalState-container-treatment.

2. Метаданные файла

Структура метаданных отличается от оборудования и версии программного обеспечения, что было использовано для измерения, а потому для каждого «типа» такого файла необходимо использовать другой подход извлечения этих самых данных.

Типы файлов, с которыми мы будем работать:

1. Raman - один спектр -> один набор метаданных.
2. UVVis - несколько спектров -> несколько наборов метаданных (мультиспектральный файл).
3. FTIR - несколько спектров -> несколько наборов метаданных (мультиспектральный файл). [пока что мы его не рассматриваем в силу того, что при конвертации файла в читаемый без особого ПО файл, теряется весь блок метаданных]

Извлечение данных из имени файла

Для разбора данных из имени файла мы создаем файл parser_filename.py. Единственная задача этого файла - извлечение данных из имени файла и формирование единой структуры, которая будет вмещать в себя все эти данные. Примером такой структуры будет словарь словарей в python.

Дело в том, что данные полученные с имени файла мы будем записывать в разные поля таблиц БД. Поэтому нам надо распределить данные следующим структурированным образом:

Имя таблицы -> {поле таблицы: значение}

В файле представлена функция parse_filename(). Она парсит имя спектрального файла и извлекает семантические данные. Возвращает словарь, готовый к последующей записи в БД.

Принимает на вход три аргумента: filename, strict_blocks, strict_sample_block.

1. filename принимает путь к файлу
2. strict_blocks - флаг для строгости проверки блоков имени файла
3. strict_sample_block - флаг для строгости проверки элементов блока <sample> имени файла

Пройдемся по некоторым «тонким» местам кода:

```
if len(parts) != 5:
        raise ValueError(f"Некорректное имя файла: {filename}")
```

parts - список блоков, на которые имя было разобрано. В данном условии идет обязательная проверка на соответствие количества блоков той структуре, которую мы задали имени файлу. Нюанс в том, что если файл измерение образца будет таким же, как и предыдущее, то в одной папке держать два файла с одинаковыми названиями нельзя (пока что мы рассматриваем все спектральные файлы для обработки в одной папке). Но по сути же можно добавить приписку, но это будет уже 6-й блок в структуре. Поэтому этот код является строгой проверкой на соответствие. Если мы будем сортировать, к примеру по папкам, спектральные файлы, то можно оставить как есть, если же нет, то стоит заменить с «!=» на «>=». Код написан так, что только первые 5 блоков извлекаются как значимые данные, а все остальные - никуда не уходят.

Рассмотрим вполне возможную ситуацию с нарушением структуры имени файла. Предположим, что в имени файла один из блоков оказался пустым, например:

Raman__Oil-BHT-0.8wt-NA-liquid-Q2-fresh_NA_R02.csv

То есть у нас пустой блок <instrument> и если мы не сделаем с этим что-то, то функция будет возвращать пустую строку в качестве значения поля. Это неправильно, поэтому надо обработать заранее, благодаря следующему коду:

```
if strict_blocks:
    for i, part in enumerate(parts):
        if part == "":
            raise ValueError(f"Пустой блок в имени файла (позиция {i+1}):
'{filename}'")
```

Если мы выставили флаг strict_blocks, то будет происходить строгая проверка на наличие пустых блоков в имени файла. Если пустой блок в имени файла есть, то поднимется исключение и выполнение программы прервется.

Но что если внутри блока <sample> будет меньшее количество элементов или какой-то элемент будет пустым? Это тоже надо отслеживать, поэтому введем две аналогичные проверки:

```
if len(sample_parts) != 7:
    raise ValueError(f"Некорректный sample-блок: {sample_block}")
```

и по аналогии со строгой проверкой блоков, но для элементов sample_block:

```
if strict_sample_block:
    for i, part in enumerate(sample_parts):
        if part == "":
            raise ValueError(
                f"Пустой элемент в sample_block (позиция {i+1}): '{sample_block}'"
            )
```

sample_parts - список элементов блока <sample>

Теперь следующий момент: концентрация вещества в имени файла записана слитно на третьей позиции в блоке <sample>. Возможна проблема, что может быть указано только одно значение и тут важно, какое именно:

1. Только концентрация -> оставляем как есть
2. Только единицы -> не записываем, так как к чему они, если нет величины концентрации?
3. Ничего нет -> ничего не записываем.

Для реализации такой логики написана функция parse_concentration(), которая на вход принимает элемент третьей позиции с блока <sample> и возвращает кортеж из двух элементов: концентрация и единицы.

Предположим, что мы допускаем пустые блоки в имени файла и в самих блоках, тогда в базу данных вместо пустых строк надо записывать NULL, для этого надо все пустые строки преобразовать в None, что бы значения корректно записались в БД. Для этого есть вспомогательная функция normalize_block(), которая преобразовывает пустые строки в None, а непустые не трогает.

Далее просто формируем итоговую структуру, которая будет возвращать все извлеченные данные.

Запись извлеченных данных в БД

Нам необходимо просто по нужным таблицам и полям в них разложить полученные данные с имени. Для этого напишем функцию insert_from_parsed_filename(). Она будет вставлять данные в таблицы sample, measurement и spectrum_file и возвращать (sample_id, measurement_id, spectrum_id). Связи создаются автоматически.

Во-первых, рассмотрим момент с тем, что в имени файла могут находиться блоки со значением NA. NA - это не пустой блок, а обозначение типа None/NULL, но в виде строки, поэтому мы должны пройтись по всем значениям полей в таблицах структуры и заменить NA на None для последующей корректной записи в БД. Для этого мы просто заново сформируем словари с преобразованными значениями:

```
sample_data = {
            k: normalize_value(v)
            for k, v in parsed_data["sample"].items()
        } # генератор словаря
measurement_data = {
            k: normalize_value(v)
            for k, v in parsed_data["measurement"].items()
        }
spectrum_data = {
            k: normalize_value(v)
            for k, v in parsed_data["spectrum_file"].items()
```

```
        }
```

parsed_data - структура, которую мы получаем при извлечении данных с имени файла.
normalize_value() - функция преобразования пустой строки или NA в None.
Также перед формированием мы должны проверить, что таблицы совпадают по названиям и типам полей с реальной БД (можно считать это немного излишним, а потому для парсеров метаданных я такого уже не делал), для этого есть функция validate_parsed_data() [там достаточно комментариев в коде]
Далее просто записываем в БД готовые, обработанные данные.

Все функции парсинга метаданных возвращают такую же по логике структуру, что и парсер имени файла. Если файл мультиспектральный - список из таких структур.
Логика записи структур в БД такая:
Так как парсер имени файла уже записывает какие-то данные в поля таблиц, создавая при этом саму запись таблицы, мы должна «обогатить» эту запись метаданными и тогда она будет полной. Если таких записей должно быть несколько, в силу того, что файл мультиспектральный, мы прокопируем с изначальной, первой обогащенной записи те поля, которые идут с имени файла (они ведь везде повторяются, поэтому незачем постоянно их парсить, достаточно прокопировать) и формирование последующих записей будет состоять из «копируемые поля + новый данные записи».
Также в db_insert.py в функциях мы обновляем некоторые данные таблиц БД. С этими частями кода стоит быть внимательным, так как вполне возможно, что человек сам внесет всю необходимую информацию в БД и обновлять ее НЕ НАДО будет. Для этого можно внести проверку.
Также стоит дополнить код следующими функциями проверки:

1. Проверка на повторную запись. (незачем один и тот же файл постоянно извлекать в БД. Его стоит просто игнорировать)
2. Проверка на обновление данных полей таблиц БД. (если поля уже наполнены вручную, то их нельзя обновлять) [это вообще можно не вводить, если мы полностью отказываемся от обновления определенных полей и оставляем только автоматическое наполнение]
3. …

Текущая версия реализует базовый функционал наполнения БД. Реализация тонких проверок намеренно отложена. Сначала отладим алгоритмы извлечения данных из сложных форматов (Raman, UVVis, FTIR [в будущем]), не загромождая код обработкой исключений, специфичных для логики управления БД.