技术栈

使用sqlalchemy作为orm框架处理sqlite, pydantic作为数据模型

pydantic1

sqlalchemy2

streamlit data_editor

streamlit_sqlite

实现思路

整体思路是: 保证DataFrame的数据和数据库的数据一致, 避免数据的重复加载
增删查改的操作都是对数据库的操作, 操作数据库后需要保证DataFrame的数据同步更新

1. 初始化变量, 使用st.session_state保存变量

```
84
85  st.session_state.setdefault('data_table', [])
86  st.session_state.setdefault('username', '')
87  st.session_state.setdefault('evolve_r', 0.02)
88  st.session_state.setdefault('n_trail', 10)
89  st.session_state.setdefault('n_epoch', 1)
90  st.session_state.setdefault('ckpt_path', list_ckpt_paths(BASE_CKPT_DIR)[0])
91  st.session_state.setdefault('mode', 2)
92  st.session_state.setdefault('configs', {
93      k: st.session_state[k] for k in ('evolve_r', 'n_trail', 'n_epoch', 'ckpt_path', 'mode')
94  })
95  if not st.session_state.data_table:
96      st.session_state.data_table = get_data_from_db()
97
```

2. 数据加载部分, 执行sql语句, 生成数据, 一般页面刷新时执行

```
    leo
79  def get_data_from_db():
80      logger.debug("init")
81      db_objs = session.query(BatchData).all()
82      return [BatchDataRead.from_orm(db_obj).dict() for db_obj in db_objs]
```

3. 数据编辑部分, 实际上是对数据库的操作, 操作数据库后需要保证DataFrame的数据同步更新, 这里把操作放到回调函数上, 避免数据频繁更新导致页面闪烁

```python
leo
def table_update_handler():
    logger.debug(f"{st.session_state.edited_info=}")
    edited_rows = st.session_state.edited_info.get('edited_rows')
    added_rows = st.session_state.edited_info.get('added_rows')
    deleted_rows = st.session_state.edited_info.get('deleted_rows')
    # 更改列
    for id_, update_data in edited_rows.items():
        row_id = int(edited_df.loc[id_].id)
        row_db = session.query(BatchData).where(BatchData.id == row_id).first()
        logger.info(f"{row_id=}, {update_data=}")
        for field in update_data:
            setattr(row_db, field, update_data[field])
        session.commit()

    # 增加列
    for new_row_data in added_rows:
        logger.debug(f"{new_row_data=}")

    for deleted_row in deleted_rows:
        row_id = int(edited_df.loc[deleted_row].id)
        logger.debug(f"{row_id=}")
        row_db = session.query(BatchData).where(BatchData.id == row_id).first()
        session.delete(row_db)
        session.commit()
        logger.debug(f"{st.session_state.data_table[-1].get('id')=}")

        deleted_row_data = next(filter(lambda x: x.get('id') == row_id, st.session_state.data_table[::-1], ))
        logger.debug(f"{deleted_row_data=}")
        list.remove(st.session_state.data_table, deleted_row_data)
        logger.debug(f"{st.session_state.data_table[-1].get('id')=}")
```

4. 最终数据处理部分, 实际上是按照不同条件生成特定的sql语句, 然后执行sql语句, 生成数据

```python
leo
def make_train_dicts(with_entities: tuple, conditions: tuple):
    return [
        DataRow(path=_.ann_file,
                img_prefix=_.img_prefix).dict()
        for _ in session.query(BatchData)
        .with_entities(*with_entities)
        .where(and_(*conditions)).all()
    ]


leo
def prepare_train_data(mode_id) -> dict[str, list]:
    # 训练集/测试集 实体类定义, 查询条件定义
    train_we, train_cd = tuple(), tuple()
    val_we, val_cd = tuple(), tuple()
    if mode_id == 1:
        train_we = (BatchData.img_prefix, BatchData.ann_file_lbs.label('ann_file'))
        train_cd = (BatchData.is_train == 1, BatchData.ann_file_lbs.is_not(None))
        val_we = (BatchData.img_prefix, BatchData.ann_file_lbs.label('ann_file'))
        val_cd = (BatchData.is_validation == 1, BatchData.ann_file_lbs.is_not(None))
    elif mode_id == 2:
        train_we = (BatchData.img_prefix,
                    case( *whens: (BatchData.ann_file_lbs.is_not(None), BatchData.ann_file_lbs),
                         else_=BatchData.ann_file).label('ann_file'))
        train_cd = (BatchData.is_train == 1,)
        val_we = (BatchData.img_prefix,
                  case( *whens: (BatchData.ann_file_lbs.is_not(None), BatchData.ann_file_lbs),
                       else_=BatchData.ann_file).label('ann_file'))
        val_cd = (BatchData.is_validation == 1,)
    elif mode_id == 3:
        train_we = (BatchData.img_prefix, BatchData.ann_file.label('ann_file'))
        train_cd = (BatchData.is_train == 1, BatchData.ann_file.is_not(None))
        val_we = (BatchData.img_prefix, BatchData.ann_file.label('ann_file'))
        val_cd = (BatchData.is_validation == 1, BatchData.ann_file.is_not(None))
    return {
        "train_data": make_train_dicts(train_we, train_cd),
        "val_data": make_train_dicts(val_we, val_cd),
    }
```

5. 数据库模型定义部分, 将数据表定义成类, 以使用sqlalchemy操作sqlite

```python
#!/usr/bin/env python3
# -*- coding:utf-8 -*-

from sqlalchemy import Column, Integer, String, create_engine, Date, TIMESTAMP
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()
# 创建 SQLite 数据库引擎
engine = create_engine('sqlite:///data.db', echo=True)


leo
class BatchData(Base):
    __tablename__ = 'batch_data'
    id = Column(Integer, primary_key=True, autoincrement=True)
    year = Column(Integer)
    census_batch = Column(String)  # 普查批次
    id_code = Column(String)
    precision = Column(String)  # 精度
    is_train = Column(Integer)
    is_validation = Column(Integer)
    ann_file = Column(String)
    ann_file_lbs = Column(String)
    img_prefix = Column(String)
    filter_empty_gt = Column(Integer)
    update_cache = Column(Integer)
    create_at = Column(TIMESTAMP)


Base.metadata.create_all(engine)

Session = sessionmaker(bind=engine, )
session = Session()

```

6. 数据模型定义部分, 主要用户数据创建, 将读取到的数据行转为数据模型类, 也可以快速转为字典

```python
leo
class BatchDataRead(BatchDataBase): ...


leo
class BatchDataCreate(BatchDataBase):
    year: int | None = 2024
    census_batch: str | None = ''  # 普查批次
    id_code: str | None = ''  # 编号
    precision: str | None = ''  # 精度
    is_train: bool | None = True
    is_validation: bool | None = False

    ann_file: str | None = ''
    ann_file_lbs: str | None = ''
    img_prefix: str | None = ''
    filter_empty_gt: bool | None = False
    update_cache: bool | None = False
    create_at: datetime | None = datetime.now()
```