Goal

The goal of the project is to create a unified library of continuous glucose monitor (CGM) data sets that allows for rapid experimentation and testing of various time-series models.

At high level, we want to design an algorithm that given the data and the pre-defined configurations is going to do the pre-processing and produce either a table (Pandas DataFrame) or expose an iterator (PyTorch DataLoader) object. An example desired use case would be:

```
config = load_yaml('path_to_config')
data_formatter = DataFormatter(config)

table = data_formatter.get_table()
iterator = data_formatter.get_iter()
```

Data formatter

Each data set usually comes as `.csv` file. The `DataFormatter` class allows us to create a processed data object that can be plugged into any Python ML package either as an iterator or as a table. Additionally, `DataFormatter` keeps track of basic information about the data such as the input and data type for each of the columns.

At a glance, whenever created, `DataFormatter` object does the following:

1. Sets parameters, including data and input types, from the config file;
2. Reads the data and call private methods to do:
    1. (NaN) Clean NaN's
    2. (Typization) Ensure data types of the data conform to the ones in the config file
    3. (Drop) Drop columns or entries as specified in the config file
    4. (Encode) Encode categorical / date columns
    5. (Interpolate) Inerpolate curve segments to ensure equally-spaced time grid
    6. (Split) Split data into train, validation, and test
    7. (Scale) Scale the data

Importantly, while doing the above processing steps (4-7), we ensure that the data / input types get updated accordingly.

An implementation of the `DataFormatter` class looks as:

```
class DataFormatter():
  # Defines and formats data for the IGLU dataset.

  def __init__(self, cnf):
    """Initialises formatter."""
    self.params = cnf
    self.__process_column_definition()
    self.__check_column_definition()
    self.data = read_csv(self.params['data_csv_path'])

    self.__check_nan()
    self.__set_data_types()
    self.__drop()
    self.__encode()
    self.__interpolate()
    self.__split_data()
    self.__scale()

  def __process_column_definition(self):
    self._column_definition = []
    for col in self.params['column_definition']:
      self._column_definition.append((col[name], col[data_typ], col[input_type]))

  def __check_column_definition(self):
    # check that there is unique ID column
    assert len([col for col in self._column_definition if
        col[2] == InputTypes.ID]) == 1
    # check that there is unique time column
    assert len([col for col in self._column_definition if
        col[2] == InputTypes.TIME]) == 1
    # check that there is at least one target column
    assert len([col for col in self._column_definition if
        col[2] == InputTypes.TARGET]) >= 1

  def __set_data_types(self):
    for col in self._column_definition:
      if col[1] == DataTypes.DATE:
        self.data[col[0]] = self.data[col[0]].astype('date')
      if col[1] == DataTypes.CATEGORICAL:
        self.data[col[0]] = self.data[col[0]].astype('category')
      if col[1] == DataTypes.REAL_VALUED:
```

```
        self.data[col[0]] = self.data[col[0]].astype('float')

    def __check_nan(self):
      # delete rows where target, time, or id are na
      for row in self.data:
          if row[id_column] == NA and row[time_column] == NA and row[target_column] == NA:
              drop row

    def __drop(self):
      # drop columns that are not in the column definition
      self.data = self.data[[col[0] for col in self._column_definition]]
      # drop rows based on conditions set in the formatter
      for col in self.params['drop'].keys():
          for row in self.params['drop'][col]:
              delete self.data[row, col]

    def __interpolate(self):
      self.data, self._column_definition = utils.interpolate(self.data,
                                                self._column_definition,
                                                **self.params['interpolation_params'])

    def __split_data(self):
      self.train_data, self.val_data, self.test_data  = utils.split(self.data,
                                                                    self._column_definition,
                                                                    **self.params['split_params'])

    def __encode(self):
      self.data, self._column_definition, self.encoders = utils.encode(self.data,
                                                                    self._column_definition,
                                                                    **self.params['encoding_params'])

    def __scale(self):
      self.train_data, self.val_data, self.test_data = utils.scale(self.data,
                                                                    self._column_definition,
                                                                    self.train_idx,
                                                                    self.val_idx,
                                                                    self.test_idx,
                                                                    **self.params['scaling_params'])
```

While we provide implementation of some of the private class methods (`nan`, `typization`, `drop`) inside the class, for longer scripts (`interpolate`, `split`, `encode`, `scale`) we outsource the code to the `utils` module. This is done for the purposes of improved code readability.

Types

The input and data types are implemented as enumerations:

```
class DataTypes(enum.IntEnum):
  """Defines numerical types of each column."""
  REAL_VALUED = 0
  CATEGORICAL = 1
  DATE = 2

class InputTypes(enum.IntEnum):
  """Defines input types of each column."""
  TARGET = 0
  OBSERVED_INPUT = 1
  KNOWN_INPUT = 2
  STATIC_INPUT = 3
  ID = 4  # Single column used as an entity identifier
  SID = 5  # Single column used as a segment identifier
  TIME = 6
```

The data types are defined to be `real_valued`, `categorical`, `date`. We define a column to be real-valued if it takes values in $\mathbb{R}$ and has unbounded support. The real-valued columns are converted to the `float` in-built type. Note we do not distinguish between `int` and `float` because majority of the models are agnostic to this distinction. We define a column to be categorical if it has finite support. The categorical columns are either converted to `string` or `categorical` in-built types. We define a column to be a date column if it contains a time stamp. The date column is cnverted to an in-built `date` type.

Helper functions

Interpolation

Splitting

Scaling

Encoding