Gretel Synthetics

Gretel.ai

Oct 14, 2021

CONTENTS:

CHAPTER

ONE

DOCUMENTATION

CHAPTER

TWO

TRY IT OUT NOW!

If you want to quickly discover gretel-synthetics, simply click the button below and follow the tutorials!

Check out additional examples here.

CHAPTER

THREE

GETTING STARTED

By default, we do not install Tensorflow via pip as many developers and cloud services such as Google Colab are running customized versions for their hardware.

```
pip install -U .
```

or

```
pip install gretel-synthetics
```

then...

```
$ pip install jupyter
$ jupyter notebook
```

When the UI launches in your browser, navigate to `examples/synthetic_records.ipynb` and get generating!

If you want to install `gretel-synthetics` locally and use a GPU (recommended):

1. Create a virtual environment (e.g. using `conda`)

```
$ conda create --name tf --python=3.8
```

1. Activate the virtual environment

```
$ conda activate tf
```

1. Run the setup script `./setup-utils/setup-gretel-synthetics-tensorflow24-with-gpu.sh`

The last step will install all the necessary software packages for GPU usage, `tensorflow=2.4` and `gretel-synthetics`. Note that this script works only for Ubuntu 18.04. You might need to modify it for other OS versions.

CHAPTER

FOUR

OVERVIEW

This package allows developers to quickly get immersed with synthetic data generation through the use of neural networks. The more complex pieces of working with libraries like Tensorflow and differential privacy are bundled into friendly Python classes and functions. There are two high level modes that can be utilized.

4.1 Simple Mode

The simple mode will train line-per-line on an input file of text. When generating data, the generator will yield a custom object that can be used a variety of different ways based on your use case. This notebook demonstrates this mode.

4.2 DataFrame Mode

This library supports CSV / DataFrames natively using the DataFrame "batch" mode. This module provided a wrapper around our simple mode that is geared for working with tabular data. Additionally, it is capabable of handling a high number of columns by breaking the input DataFrame up into "batches" of columns and training a model on each batch. This notebook shows an overview of using this library with DataFrames natively.

4.3 Components

There are four primary components to be aware of when using this library.

1) Configurations. Configurations are classes that are specific to an underlying ML engine used to train and generate data. An example would be using `TensorFlowConfig` to create all the necessary parameters to train a model based on TF. `LocalConfig` is aliased to `TensorFlowConfig` for backwards compatability with older versions of the library. A model is saved to a designated directory, which can optionally be archived and utilized later.

2) Tokenizers. Tokenizers convert input text into integer based IDs that are used by the underlying ML engine. These tokenizers can be created and sent to the training input. This is optional, and if no specific tokenizer is specified then a default one will be used. You can find an example here that uses a simple char-by-char tokenizer to build a model from an input CSV. When training in a non-differentially private mode, we suggest using the default `SentencePiece` tokenizer, an unsupervised tokenizer that learns subword units (e.g., **byte-pair-encoding (BPE)** [Sennrich et al.]) and **unigram language model** [Kudo.]) for faster training and increased accuracy of the synthetic model.

3) Training. Training a model combines the configuration and tokenizer and builds a model, which is stored in the designated directory, that can be used to generate new records.

4) Generation. Once a model is trained, any number of new lines or records can be generated. Optionally, a record validator can be provided to ensure that the generated data meets any constraints that are necessary. See our notebooks for examples on validators.

CHAPTER

FIVE

DIFFERENTIAL PRIVACY

Differential privacy support for our TensorFlow mode is built on the great work being done by the Google TF team and their TensorFlow Privacy library.

When utilizing DP, we currently recommend using the character tokenizer as it will only create a vocabulary of single tokens and removes the risk of sensitive data being memorized as actual tokens that can be replayed during generation.

There are also a few configuration options that are notable such as:

- `predict_batch_size` should be set to 1
- `dp` should be enabled
- `learning_rate`, `dp_noise_multiplier`, `dp_l2_norm_clip`, and `dp_microbatches` can be adjusted to achieve various epsilon values.
- `reset_states` should be disabled

Please see our example Notebook for training a DP model based on the Netflix Prize dataset.

CHAPTER

SIX

MODULES

6.1 Config

This module provides a set of dataclasses that can be used to hold all necessary confguration parameters for training a model and generating data.

For example usage please see our Jupyter Notebooks.

class gretel_synthetics.config.**BaseConfig**(*input_data_path: str = None*, *validation_split: bool = True*, *checkpoint_dir: str = None*, *training_data_path: str = None*, *field_delimiter: Optional[str] = None*, *field_delimiter_token: str = '<d>'*, *model_type: str = None*, *max_lines: int = 0*, *overwrite: bool = False*, *epoch_callback: Optional[Callable] = None*, *max_training_time_seconds: Optional[int] = None*, *vocab_size: int = 20000*, *character_coverage: float = 1.0*, *pretrain_sentence_count: int = 1000000*, *max_line_len: int = 2048*)

This class should not be used directly, engine specific classes should derived from this class.

as_dict()

Serialize the config attrs to a dict

checkpoint_dir: str = None

Directory where model data will be stored, user provided.

epoch_callback: Optional[Callable] = None

Callback to be invoked at the end of each epoch. It will be invoked with an EpochState instance as its only parameter. NOTE that the callback is deleted when save_model_params is called, we do not attempt to serialize it to JSON.

field_delimiter: Optional[str] = None

If the input data is structured, you may specify a field delimiter which can be used to split the generated text into a list of strings. For more detail please see the `GenText` class in the `generate.py` module.

field_delimiter_token: str = '<d>'

Depending on the tokenizer used, a special token can be used to represent characters. For tokenizers, like SentencePiece that support this, we will replace the field delimiter char with this token to provide better learning and generation. If the tokenizer used does not support custom tokens, this value will be ignored

abstract get_generator_class() → None

This must be implemented by all specific configs. It should return the class that should be used as the Generator for creating records.

abstract get_training_callable() → Callable
This must be implemented by all specific configs. It should return a callable that should be used as the entrypoint for training a model.

gpu_check()
Optionally do a GPU check and warn if a GPU is not available, if not overridden, do nothing

input_data_path: str = None
Path to raw training data, user provided.

max_lines: int = 0
The maximum number of lines to utilize from the raw input data.

max_training_time_seconds: Optional[int] = None
If set, training will cease after the number of seconds specified elapses. This timeout will be evaluated after each epoch.

model_type: str = None
A string version of the model config class. This is used to keep track of what underlying engine was used when writing the config to a file. This will be automatically updated during construction.

overwrite: bool = False
Set to `True` to automatically overwrite previously saved model checkpoints. If `False`, the trainer will generate an error if checkpoints exist in the model directory. Default is `False`.

training_data_path: str = None
Where annotated and tokenized training data will be stored. This attr will be modified during construction.

validation_split: bool = True
Use a fraction of the training data as validation data. Use of a validation set is recommended as it helps prevent over-fitting and memorization. When enabled, 20% of data will be used for validation.

gretel_synthetics.config.**CONFIG_MAP = {'TensorFlowConfig': <class 'gretel_synthetics.conf**
A mapping of configuration subclass string names to their actual classes. This can be used to re-instantiate a config from a serialized state.

gretel_synthetics.config.**LocalConfig**
alias of *gretel_synthetics.config.TensorFlowConfig*

class gretel_synthetics.config.**TensorFlowConfig**(*input_data_path: str = None*, *validation_split: bool = True*, *checkpoint_dir: str = None*, *training_data_path: str = None*, *field_delimiter: Optional[str] = None*, *field_delimiter_token: str = '<d>'*, *model_type: str = None*, *max_lines: int = 0*, *overwrite: bool = False*, *epoch_callback: Optional[Callable] = None*, *max_training_time_seconds: Optional[int] = None*, *vocab_size: int = 20000*, *character_coverage: float = 1.0*, *pretrain_sentence_count: int = 1000000*, *max_line_len: int = 2048*, *epochs: int = 100*, *early_stopping: bool = True*, *early_stopping_patience: int = 5*, *best_model_metric: str = None*, *early_stopping_min_delta: float = 0.001*, *batch_size: int = 64*, *buffer_size: int = 10000*, *seq_length: int = 100*, *embedding_dim: int = 256*, *rnn_units: int = 256*, *learning_rate: float = 0.01*, *dropout_rate: float = 0.2*, *rnn_initializer: str = 'glorot_uniform'*, *dp: bool = False*, *dp_noise_multiplier: float = 0.1*, *dp_l2_norm_clip: float = 3.0*, *dp_microbatches: int = 64*, *gen_temp: float = 1.0*, *gen_chars: int = 0*, *gen_lines: int = 1000*, *predict_batch_size: int = 64*, *reset_states: bool = True*, *save_all_checkpoints: bool = False*, *save_best_model: bool = True*)

TensorFlow config that contains all of the main parameters for training a model and generating data.

Parameters

- **epochs** (*optional*) – Number of epochs to train the model. An epoch is an iteration over the entire training set provided. For production use cases, 15-50 epochs are recommended. The default is `100` and is intentionally set extra high. By default, `early_stopping` is also enabled and will stop training epochs once the model is no longer improving.
- **early_stopping** (*optional*) – deduce when the model is no longer improving and terminating training.
- **early_stopping_patience** (*optional*) – in the model. After this number of epochs, training will terminate.
- **best_model_metric** (*optional*) – The metric to use to track when a model is no longer improving. Alternative options are "val_acc" or "acc". A error will be raised if a valid value is not specified.
- **early_stopping_min_delta** (*optional*) – as an improvement, i.e. an absolute change of less than min_delta will count as no improvement.
- **batch_size** (*optional*) – Number of samples per gradient update. Using larger batch sizes can help make more efficient use of CPU/GPU parallelization, at the cost of memory. If unspecified, batch_size will default to `64`.

- **buffer_size** (*optional*) – Buffer size which is used to shuffle elements during training. Default size is `10000`.
- **seq_length** (*optional*) – The maximum length sentence we want for a single training input in characters. Note that this setting is different than max_line_length, as seq_length simply affects the length of the training examples passed to the neural network to predict the next token. Default size is `100`.
- **embedding_dim** (*optional*) – Vector size for the lookup table used in the neural network Embedding layer that maps the numbers of each character. Default size is `256`.
- **rnn_units** (*optional*) – Positive integer, dimensionality of the output space for LSTM layers. Default size is `256`.
- **dropout_rate** (*optional*) – Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Using a dropout can help to prevent overfitting by ignoring randomly selected neurons during training. 0.2 (20%) is often used as a good compromise between retaining model accuracy and preventing overfitting. Default is 0.2.
- **rnn_initializer** (*optional*) – Initializer for the kernal weights matrix, used for the linear transformation of the inputs. Default is `glorot_transform`.
- **dp** (*optional*) – If `True`, train model with differential privacy enabled. This setting provides assurances that the models will encode general patterns in data rather than facts about specific training examples. These additional guarantees can usefully strengthen the protections offered for sensitive data and content, at a small loss in model accuracy and synthetic data quality. The differential privacy epsilon and delta values will be printed when training completes. Default is `False`.
- **learning_rate** (*optional*) – The higher the learning rate, the more that each update during training matters. Note: When training with differential privacy enabled, if the updates are noisy (such as when the additive noise is large compared to the clipping threshold), a low learning rate may help with training. Default is `0.01`.
- **dp_noise_multiplier** (*optional*) – The amount of noise sampled and added to gradients during training. Generally, more noise results in better privacy, at the expense of model accuracy. Default is `0.1`.
- **dp_l2_norm_clip** (*optional*) – The maximum Euclidean (L2) norm of each gradient is applied to update model parameters. This hyperparameter bounds the optimizer's sensitivity to individual training points. Default is `3.0`.
- **dp_microbatches** (*optional*) – Each batch of data is split into smaller units called micro-batches. Computational overhead can be reduced by increasing the size of micro-batches to include more than one training example. The number of micro-batches should divide evenly into the overall `batch_size`. Default is `64`.
- **gen_temp** (*optional*) – Controls the randomness of predictions by scaling the logits before applying softmax. Low temperatures result in more predictable text. Higher temperatures result in more surprising text. Experiment to find the best setting. Default is `1.0`.
- **gen_chars** (*optional*) – Maximum number of characters to generate per line. Default is `0` (no limit).
- **gen_lines** (*optional*) – Maximum number of text lines to generate. This function is used by `generate_text` and the optional `line_validator` to make sure that all lines created by the model pass validation. Default is `1000`.
- **predict_batch_size** (*optional*) – How many words to generate in parallel. Higher values may result in increased throughput. The default of `64` should provide reasonable performance for most users.

- **reset_states** (*optional*) – Reset RNN model states between each record created guarantees more consistent record creation over time, at the expense of model accuracy. Default is `True`.
- **save_all_checkpoints** (*optional*) – which can be useful for optimal model selection. Set to `False` to save only the latest checkpoint. Default is `True`.
- **save_best_model** (optional). Defaults to `True`. Track the best version of the model (checkpoint) – If `save_all_checkpoints` is disabled, then the saved model will be overwritten by newer ones only if they are better.

get_generator_class()
This must be implemented by all specific configs. It should return the class that should be used as the Generator for creating records.

get_training_callable()
This must be implemented by all specific configs. It should return a callable that should be used as the entrypoint for training a model.

gpu_check()
Optionally do a GPU check and warn if a GPU is not available, if not overridden, do nothing

gretel_synthetics.config.**config_from_model_dir**(*model_dir: str*) → *gretel_synthetics.config.BaseConfig*
Factory that will take a known directory of a model and return a class instance for that config. We automatically try and detect the correct BaseConfig sub-class to use based on the saved model params.

If there is no `model_type` param in the saved config, we assume that the model was saved using an earlier version of the package and will instantiate a TensorFlowConfig

6.2 Tokenizers

Interface definitions for tokenizers. The classes in the module are segmented into two abstract types: Trainers and Tokenizers. They are kept separate because the parameters used to train a tokenizer are not necessarily loaded back in and utilized by a trained tokenizer. While its more explicit to utilize two types of classes, it also removes any ambiguity in which methods are able to be used based on training or tokenizing.

Trainers require a specific configuration to be provided. Based on the configuration received, the tokenizer trainers will create the actual training data file that will be used by the downstream training process. In this respect, utilizing at least one of these tokenizers is required for training since it is the tokenizers responsbility to create the final training data to be used.

The general process that is followed when using these tokenizers is:

Create a trainer instance, with desired parameters, including providing the config as a required param.

Call the `annotate_data` for your tokenizer trainer. What is important to note here is that this method actually iterates the input data line by line, and does any special processing, then writes a new data file that will be used for actual training. This new data file is written to the model directory.

Call the `train` method, which will create your tokenization model and save it to the model directory.

Now you will use the `load()` class method from an actual tokenizer class to load that trained model in and now you can use it on input data.

class gretel_synthetics.tokenizers.**Base**
High level base class for shared class attrs and validation. Should not be used directly.

class gretel_synthetics.tokenizers.**BaseTokenizer**(*model_data: Any*, *model_dir: str*)
Base class for loading a tokenizer from disk. Should not be used directly.

decode_from_ids(*ids: List[int]*) → str
Given a list of token IDs, convert it to a single string that would be the original string it was.

Note: We automatically call a method that can optionally restore any special reserved tokens back to their original values (such as field delimiter values, etc)

---

encode_to_ids(*data: str*) → List[int]
Given an input string, convert it to a list of token IDs

abstract classmethod load(*model_dir: str*)
Given a directory to a model, load the specific tokenizer model into an instance. Subclasses should implement this logic specific to how they need to load a model back in

abstract property total_vocab_size
Return the total count of unique tokens in the vocab, specific to the underlying tokenizer to be used.

class gretel_synthetics.tokenizers.**BaseTokenizerTrainer**(*, *config: None*, *vocab_size: Optional[int] = None*)
Base class for training tokenizers. Should not be used directly.

annotate_data() → Iterator[str]
This should be called _before_ training as it is required to have the annotated training data created in the model directory.

Read in the configurations raw input data path, and create a file I/O pipeline where each line of the input data path can optionally route through an annotation function and then we will write each raw line out into a training data file as specified by the config.

config: BaseConfig = None
A subclass instace of `BaseConfig`. This will be used to find the input data for tokenization

data_iterator() → Iterator[str]
Create a generator that will iterate each line of the training data that was created during the annotation step. Synthetic model trainers will most likely need to iterate this to process each line of the annotated training data.

num_lines: int = 0
The number of lines that were processed after `create_annotated_training_data` is called

train()
Train a tokenizer and save the tokenizer settings to a file located in the model directory specified by the `config` object

vocab_size: int = None
The max size of the vocab (tokens) to be extracted from the input dataset.

class gretel_synthetics.tokenizers.**CharTokenizer**(*model_data: Any*, *model_dir: str*)
Load a simple character tokenizer from disk to conduct encoding an decoding operations

classmethod load(*model_dir: str*)
Create an instance of this tokenizer.

Parameters **model_dir** – The path to the model directory

property total_vocab_size
Get the number of unique characters (tokens)

class gretel_synthetics.tokenizers.**CharTokenizerTrainer**(*, *config: None*, *vocab_size: Optional[int] = None*)

Train a simple tokenizer that maps every single character to a unique ID. If `vocab_size` is not specified, the learned vocab size will be the number of unique characters in the training dataset.

Parameters **vocab_size** – Max number of tokens (chars) to map to tokens.

class gretel_synthetics.tokenizers.**SentencePieceTokenizer**(*model_data: Any*, *model_dir: str*)

Load a SentencePiece tokenizer from disk so encoding / decoding can be done

classmethod load(*model_dir: str*)

Load a SentencePiece tokenizer from a model directory.

Parameters **model_dir** – The model directory.

property total_vocab_size

The number of unique tokens in the model

class gretel_synthetics.tokenizers.**SentencePieceTokenizerTrainer**(*, *character_coverage: float = 1.0*, *pretrain_sentence_count: int = 1000000*, *max_line_len: int = 2048*, ***kwargs*)

Train a tokenizer using Google SentencePiece.

character_coverage: float = None

The amount of characters covered by the model. Unknown characters will be replaced with the <unk> tag. Good defaults are `0.995` for languages with rich character sets like Japanese or Chinese, and 1.0 for other languages or machine data. Default is `1.0`.

max_line_line: int = None

Maximum line length for input training data. Any lines longer than this length will be ignored. Default is `2048`.

pretrain_sentence_count: int = None

The number of lines spm_train first loads. Remaining lines are simply discarded. Since spm_train loads entire corpus into memory, this size will depend on the memory size of the machine. It also affects training time. Default is `1000000`.

vocab_size: int = None

Pre-determined maximum vocabulary size prior to neural model training, based on subword units including byte-pair-encoding (BPE) and unigram language model, with the extension of direct training from raw sentences. We generally recommend using a large vocabulary size of 20,000 to 50,000. Default is `20000`.

exception gretel_synthetics.tokenizers.**TokenizerError**

gretel_synthetics.tokenizers.**tokenizer_from_model_dir**(*model_dir: str*) → *gretel_synthetics.tokenizers.BaseTokenizer*

A factory function that will return a tokenizer instance that can be used for encoding / decoding data. It will try to automatically infer what type of class to use based on the stored tokenizer params in the provided model directory.

If no specific tokenizer type is found, we assume that we are restoring a SentencePiece tokenizer because the model is from a version <= 0.14.x

Parameters **model_dir** – A directory that holds synthetic model data.

6.3 Train

Train models for creating synthetic data. This module is the primary entrypoint for creating a model. It depends on having created a engine specifc configuration and optionally a tokenizer to be used.

class gretel_synthetics.train.**EpochState**(*epoch: int*, *accuracy: Optional[float] = None*, *loss: Optional[float] = None*, *val_accuracy: Optional[float] = None*, *val_loss: Optional[float] = None*, *batch: Optional[int] = None*, *epsilon: Optional[float] = None*, *delta: Optional[float] = None*)

Training state passed to the epoch callback on BaseConfig at the end of each epoch.

class gretel_synthetics.train.**TrainingParams**(*tokenizer_trainer: None*, *tokenizer: None*, *config: None*)

A structure that is created and passed into the engine-specific training entrypoint. All engine-specific training entrypoints should expect to receive this object and process accordingly.

gretel_synthetics.train.**train**(*store: None*, *tokenizer_trainer: None = None*)

Train a Synthetic Model. This is a facade entrypoint that implements the engine specific training operation based on the provided configuration.

Parameters

- **store** – A subclass instance of `BaseConfig`. This config is reponsible for providing the actual training entrypoint for a specific training routine.
- **tokenizer_trainer** – An optional subclass instance of a `BaseTokenizerTrainer`. If provided this tokenizer will be used to pre-process and create an annotated dataset for training. If not provided a default tokenizer will be used.

gretel_synthetics.train.**train_rnn**(*store: None*)

Facade to support backwards compatibility for <= 0.14.x versions.

6.4 Generate

Abstract module for generating data. The `generate_text` function is the primary entrypoint for creating text.

class gretel_synthetics.generate.**BaseGenerator**

Do not use directly.

Specific generation modules should have a subclass of this ABC that implements the core logic for generating data

class gretel_synthetics.generate.**GenText**(*valid: bool = None*, *text: str = None*, *explain: str = None*, *delimiter: str = None*)

gretel_synthetics.generate.**PredString**

alias of `gretel_synthetics.generate.pred_string`

class gretel_synthetics.generate.**SeedingGenerator**(*config: None*, ***, *seed_list: List[str]*, *line_validator: Optional[Callable] = None*, *max_invalid: int = 1000*)

A single threaded line / text generator that is specifically for using with a list of seeds. This also exposes the `Settings` class back to the caller so the actual seed list can be directly accessed, which controls the underlying progression of the main text generator.

This is useful when you need to manipulate the actual seed list as data is being generated.

class gretel_synthetics.generate.**Settings**(*config: None*, *start_string: Union[str, List[str], None] = None*, *multi_seed: bool = False*, *line_validator: Optional[Callable] = None*, *max_invalid: int = 1000*, *tokenizer:* gretel_synthetics.tokenizers.BaseTokenizer *= None*, *generator:* gretel_synthetics.generate.BaseGenerator *= None*)

Do not use directly.

Arguments for a generator generating lines of text.

This class contains basic settings for a generation process. It is separated from the Generator class for ensuring reliable serializability without an excess amount of code tied to it.

This class also will take a provided start string and validate that it can be utilized for text generation. If the `start_string` is something other than the default, we have to do a couple things:

1) If the config utilizes a field delimiter, the `start_string` MUST end with that delimiter
2) Convert the user-facing delim char into the special delim token specified in the config

class gretel_synthetics.generate.**gen_text**(*valid: bool = None*, *text: str = None*, *explain: str = None*, *delimiter: str = None*)

A record that is yielded from the `Generator.generate_next` generator.

valid
: True, False, or None. If the line passed a validation function, then this will be `True`. If the validation function raised an exception then this will be automatically set to `False`. If no validation function is used, then this value will be `None`.

text
: The actual record as a string

explain
: A string that describes why a record failed validation. This is the string representation of the `Exception` that is thrown in a validation function. This will only be set if validation fails, otherwise will be `None`.

delimiter
: If the generated text are column/field based records. This will hold the delimiter used to separate the fields from each other.

as_dict() → dict
: Serialize the generated record to a dictionary

values_as_list() → Optional[List[str]]
: Attempt to split the generated text on the provided delimiter

  Returns A list of values that are separated by the object's delimiter or None is there is no delimiter in the text

gretel_synthetics.generate.**generate_text**(*config: None*, *start_string: Union[str, List[str], None] = None*, *line_validator: Optional[Callable] = None*, *max_invalid: int = 1000*, *num_lines: Optional[int] = None*, *parallelism: int = 0*) → Iterator[*gretel_synthetics.generate.GenText*]

A generator that will load a model and start creating records.

Parameters

- **config** – A configuration object, which you must have created previously

- **start_string** – A prefix string that is used to seed the record generation. By default we use a newline, but you may substitue any initial value here which will influence how the generator predicts what to generate. If you are working with a field delimiter, and you want to seed more than one column value, then you MUST utilize the field delimiter specified in your config. An example would be "foo,bar,baz,". Also, if using a field delimiter, the string MUST end with the delimiter value.

  Note: This param may also be a list of prefixes. If this is provided, then the generator will attempt to create exactly 1 record for each seed in the list. The `num_lines` param will be implicity set to the size of the list and this number of records will be created at a 1:1 ratio between prefix strings and valid records.

- **line_validator** – An optional callback validator function that will take the raw string value from the generator as a single argument. This validator can executue arbitrary code with the raw string value. The validator function may return a bool to indicate line validity. This boolean value will be set on the yielded `gen_text` object. Additionally, if the validator throws an exception, the `gen_text` object will be set with a failed validation. If the validator returns None, we will assume successful validation.
- **max_invalid** – If using a `line_validator`, this is the maximum number of invalid lines to generate. If the number of invalid lines exceeds this value a `RunTimeError` will be raised.
- **num_lines** – If not `None`, this will override the `gen_lines` value that is provided in the `config`. .. note:

  ```
  If ``start_string`` is a list, this value will be set to the␣
  ↪length of that list and any other
  values for the param are ignored.
  ```

- **parallelism** – The number of concurrent workers to use. `1` (the default) disables parallelization, while a non-positive value means "number of CPUs + x" (i.e., use `0` for using as many workers as there are CPUs). A floating-point value is interpreted as a fraction of the available CPUs, rounded down.

Simple validator example:

```
def my_validator(raw_line: str):
    parts = raw_line.split(',')
    if len(parts) != 5:
        raise Exception('record does not have 5 fields')
```

Note: `gen_lines` from the `config` is important for this function. If a line validator is not provided, each line will count towards the number of total generated lines. When the total lines generated is >= `gen_lines` we stop. If a line validator is provided, only *valid* lines will count towards the total number of lines generated. When the total number of valid lines generated is >= `gen_lines`, we stop.

Note: `gen_chars`, controls the possible maximum number of characters a single generated line can have. If a newline character has not been generated before reaching this number, then the line will be returned. For example if `gen_chars` is 180 and a newline has not been generated, once 180 chars have been created, the line will be returned no matter what. As a note, if this value is 0, then each line will generate until a newline is observed.

Yields A `GenText` object for each record that is generated. The generator will stop after the max number of lines is reached (based on your config).

Raises **A RunTimeError if the max_invalid number of lines is generated** –

6.5 Batch

This module allows automatic splitting of a DataFrame into smaller DataFrames (by clusters of columns) and doing model training and text generation on each sub-DF independently.

Then we can concat each sub-DF back into one final synthetic dataset.

For example usage, please see our Jupyter Notebook.

class gretel_synthetics.batch.**Batch**(*checkpoint_dir: str*, *input_data_path: str*, *headers: List[str]*, *config:* gretel_synthetics.config.TensorFlowConfig, *gen_data_count: int = 0*)

A representation of a synthetic data workflow. It should not be used directly. This object is created automatically by the primary batch handler, such as `DataFrameBatch`. This class holds all of the necessary information for training, data generation and DataFrame re-assembly.

add_valid_data(*data:* gretel_synthetics.generate.GenText)

Take a `gen_text` object and add the generated line to the generated data stream

get_validator()

If a custom validator is set, we return that. Otherwise, we return the built-in validator, which simply checks if a generated line has the right number of values based on the number of headers for this batch.

This at least makes sure the resulting DataFrame will be the right shape

load_validator_from_file()

Load a saved validation object if it exists

reset_gen_data()

Reset all objects that accumulate or track synthetic data generation

set_validator(*fn: Callable*, *save=True*)

Assign a validation callable to this batch. Optionally pickling and saving the validator for loading later

property synthetic_df

Get a DataFrame constructed from the generated lines

class gretel_synthetics.batch.**DataFrameBatch**(*, *df: pandas.core.frame.DataFrame = None*, *batch_size: int = 15*, *batch_headers: List[List[str]] = None*, *config: Union[dict,* gretel_synthetics.config.BaseConfig*] = None*, *tokenizer:* gretel_synthetics.tokenizers.BaseTokenizerTrainer *= None*, *mode: str = 'write'*, *checkpoint_dir: str = None*)

Create a multi-batch trainer / generator. When created, the directory structure to store models and training data will automatically be created. The directory structure will be created under the "checkpoint_dir" location provided in the `config` template. There will be one directory per batch, where each directory will be called "batch_N" where N is the batch number, starting from 0.

Training and generating can happen per-batch or we can loop over all batches to do both train / generation functions.

Example

When creating this object, you must explicitly create the training data from the input DataFrame before training models:

```
my_batch = DataFrameBatch(df=my_df, config=my_config)
my_batch.create_training_data()
my_batch.train_all_batches()
```

Parameters

- **df** – The input, source DataFrame
- **batch_size** – If `batch_headers` is not provided we automatically break up the number of columns in the source DataFrame into batches of N columns.
- **batch_headers** – A list of lists of strings can be provided which will control the number of batches. The number of inner lists is the number of batches, and each inner list represents the columns that belong to that batch
- **config** – A template training config to use, this will be used as kwargs for each Batch's synthetic configuration. This may also be a sucblass of `BaseConfig`. If this is used, you can set the `input_data_path` param to the constant `PATH_HOLDER` as it does not really matter
- **tokenizer_class** – An optional `BaseTokenizerTrainer` subclass. If not provided the default tokenizer will be used for the underlying ML engine.

Note: When providing a config, the source of training data is not necessary, only the `checkpoint_dir` is needed. Each batch will control its input training data path after it creates the training dataset.

---

batch_size: int = None
The max number of columns allowed for a single DF batch

batch_to_df(*batch_idx: int*) → pandas.core.frame.DataFrame
Extract a synthetic data DataFrame from a single batch.

Parameters **batch_idx** – The batch number

Returns A DataFrame with synthetic data

batches: Dict[int, Batch] = None
A mapping of `Batch` objects to a batch number. The batch number (key) increments from 0..N where N is the number of batches being used.

batches_to_df() → pandas.core.frame.DataFrame
Convert all batches to a single synthetic data DataFrame.

Returns A single DataFrame that is the concatenation of all the batch DataFrames.

config: Union[dict, BaseConfig] = None
The template config that will be used for all batches. If a dict is provided we default to a TensorFlowConfig.

create_training_data()
Split the original DataFrame into N smaller DataFrames. Each smaller DataFrame will have the same number of rows, but a subset of the columns from the original DataFrame.

This method iterates over each `Batch` object and assigns a smaller training DataFrame to the `training_df` attribute of the object.

Finally, a training CSV is written to disk in the specific batch directory

generate_all_batch_lines(*max_invalid=1000*, *raise_on_failed_batch: bool = False*, *num_lines: int = None*, *seed_fields: Union[dict, List[dict]] = None*, *parallelism: int = 0*) → Dict[int, *gretel_synthetics.batch.GenerationSummary*]

Generate synthetic lines for all batches. Lines for each batch are added to the individual `Batch` objects. Once generateion is done, you may re-assemble the dataset into a DataFrame.

Example:

```
my_batch.generate_all_batch_lines()
# Wait for all generation to complete
synthetic_df = my_batch.batches_to_df()
```

Parameters

- **max_invalid** – The number of invalid lines, per batch. If this number is exceeded for any batch, generation will stop.
- **raise_on_failed_batch** – If True, then an exception will be raised if any single batch fails to generate the requested number of lines. If False, then the failed batch will be set to `False` in the result dictionary from this method.
- **num_lines** – The number of lines to create from each batch. If `None` then the value from the config template will be used.

  Note: Will be overridden / ignored if `seed_fields` is a list. Will be set to the len of the list.

- **seed_fields** – A dictionary that maps field/column names to initial seed values for those columns. This seed will only apply to the first batch that gets trained and generated. Additionally, the fields provided in the mapping MUST exist at the front of the first batch.

  Note: This param may also be a list of dicts. If this is the case, then `num_lines` will automatically be set to the list length downstream, and a 1:1 ratio will be used for generating valid lines for each prefix.

- **parallelism** – The number of concurrent workers to use. `1` (the default) disables parallelization, while a non-positive value means "number of CPUs + x" (i.e., use `0` for using as many workers as there are CPUs). A floating-point value is interpreted as a fraction of the available CPUs, rounded down.

Returns

A dictionary of batch number to a dictionary that reports the number of valid, invalid lines and bool value that shows if each batch was able to generate the full number of requested lines:

```
{
    0: GenerationSummary(valid_lines=1000, invalid_lines=10, is_
↪valid=True),
    1: GenerationSummary(valid_lines=500, invalid_lines=5, is_
↪valid=True)
}
```

generate_batch_lines(*batch_idx: int*, *max_invalid=1000*, *raise_on_exceed_invalid: bool = False*, *num_lines: int = None*, *seed_fields: Union[dict, List[dict]] = None*, *parallelism: int = 0*) → *gretel_synthetics.batch.GenerationSummary*

Generate lines for a single batch. Lines generated are added to the underlying `Batch` object for each batch. The lines can be accessed after generation and re-assembled into a DataFrame.

Parameters

- **batch_idx** – The batch number
- **max_invalid** – The max number of invalid lines that can be generated, if this is exceeded, generation will stop
- **raise_on_exceed_invalid** – If true and if the number of lines generated exceeds the `max_invalid` amount, we will re-raise the error thrown by the generation module which will interrupt the running process. Otherwise, we will not raise the caught exception and just return `False` indicating that the batch failed to generate all lines.
- **num_lines** – The number of lines to generate, if `None`, then we use the number from the batch's config
- **seed_fields** – A dictionary that maps field/column names to initial seed values for those columns. This seed will only apply to the first batch that gets trained and generated. Additionally, the fields provided in the mapping MUST exist at the front of the first batch.

  Note: This param may also be a list of dicts. If this is the case, then `num_lines` will automatically be set to the list length downstream, and a 1:1 ratio will be used for generating valid lines for each prefix.

- **parallelism** – The number of concurrent workers to use. `1` (the default) disables parallelization, while a non-positive value means "number of CPUs + x" (i.e., use `0` for using as many workers as there are CPUs). A floating-point value is interpreted as a fraction of the available CPUs, rounded down.

master_header_list: List[str] = None

During training, this is the original column order. When reading from disk, we concatenate all headers from all batches together. This list is not guaranteed to preserve the original header order.

original_headers: List[str] = None

Stores the original header list / order from the original training data that was used. This is written out to the model directory during training and loaded back in when using read-only mode.

set_batch_validator(*batch_idx: int*, *validator: Callable*)

Set a validator for a specific batch. If a validator is configured for a batch, each generated record from that batch will be sent to the validator.

Parameters

- **batch_idx** – The batch number .
- **validator** – A callable that should take exactly one argument, which will be the raw line generated from the `generate_text` function.

train_all_batches()

Train a model for each batch.

train_batch(*batch_idx: int*)

Train a model for a single batch. All model information will be written into that batch's directory.

Parameters **batch_idx** – The index of the batch, from the `batches` dictionary

class gretel_synthetics.batch.**GenerationProgress**(*current_valid_count: int = 0, current_invalid_count: int = 0, new_valid_count: int = 0, new_invalid_count: int = 0, completion_percent: float = 0.0, timestamp: float = <factory>*)

This class should not have to be used directly.

It is used to communicate the current progress of record generation.

When a callback function is passed to the `RecordFactory.generate_all()` method, each time the callback is called an instance of this class will be passed as the single argument:

```
def my_callback(data: GenerationProgress):
    ...

factory: RecordFactory
df = factory.generate_all(output="df", callback=my_callback)
```

This class is used to periodically communicate progress of generation to the user, through a callback that can be passed to `RecordFactory.generate_all()` method.

completion_percent: float = 0.0
The percentage of valid lines/records that have been generated.

current_invalid_count: int = 0
The number of invalid lines/records that were generated so far.

current_valid_count: int = 0
The number of valid lines/records that were generated so far.

new_invalid_count: int = 0
The number of new valid lines/records that were generated since the last progress callback.

new_valid_count: int = 0
The number of new valid lines/records that were generated since the last progress callback.

timestamp: float = None
The timestamp from when the information in this object has been captured.

class gretel_synthetics.batch.**GenerationSummary**(*valid_lines: int = 0, invalid_lines: int = 0, is_valid: bool = False*)

A class to capture the summary data after synthetic data is generated.

class gretel_synthetics.batch.**RecordFactory**(**, num_lines: int, batches: dict, header_list: list, delimiter: str, seed_fields: Union[dict, list] = None, max_invalid=1000, validator: Optional[Callable] = None, parallelism: int = 4, invalid_cache_size: int = 100*)

A stateful factory that can be used to generate and validate entire records, regardless of the number of underlying header clusters that were used to build multiple sub-models.

Instances of this class should be created by calling the appropiate method of the `DataFrameBatch` instance. This class should not have to be used directly. You should be able to create an instance like so:

```
factory = batcher.create_record_factory(num_lines=50)
```

The class is init'd with default capacity and limits as specified by the `num_lines` and `max_invalid` attributes. At any time, you can inspect the state of the instance by doing:

```
factory.summary
```

The factory instance can be used one of two ways: buffered or unbuffered.

For unbuffered mode, the entire instance can be used as an iterator to create synthetic records. Each record will be a dictionary.

Note: All values in the generated dictionaries will be strings.

The `valid_count` and `invalid_count` counters will update as records are generated.

When creating the record factory, you may also provide an entire record validator:

```
def validator(rec: dict):
    ...

factory = batcher.create_record_factory(num_lines=50, validator=validator)
```

Each generated record dict will be passed to the validator. This validator may either return False or raise an exception to mark a record as invalid.

At any point, you may reset the state of the factory by calling:

```
factory.reset()
```

This will reset all counters and allow you to keep generating records.

Finally, you can generate records in buffered mode, where generated records will be buffered in memory and returned as one collection. By default, a list of dicts will be returned:

```
factory.generate_all()
```

You may request the records to be returned as a DataFrame. The dtypes will be inferred as if you were reading the data from a CSV:

```
factory.generate_all(output="df")
```

Note: When using `generate_all`, the factory states will be reset automatically.

---

generate_all(*output: Optional[str] = None*, *callback: Optional[callable] = None*, *callback_interval: int = 30*, *callback_threading: bool = False*)

Attempt to generate the full number of records that was set when creating the `RecordFactory`. This method will create a buffer that holds all records and then returns the the buffer once generation is complete.

Parameters

- **output** – How the records should be returned. If `None`, which is the default, then a list of record dicts will be returned. Other options that are supported are: 'df' for a DataFrame.
- **callback** – An optional callable that will periodically be called with a `GenerationProgress` instance as the single argument while records are being generated.
- **callback_interval** – If using a callback, the minimum number of seconds that should occur between callbacks.

- **callback_threading** – If enabled, a watchdog thread will be used to execute the callback. This will ensure that the callback is called regardless of invalid or valid counts. If callback threading is disabled, the callback will only be called after valid records are generated. If the callback raises and exception, then a threading event will be set which will trigger the stopping of generation.

Returns Generated records in an object that is dependent on the `output` param. By default this will be a list of dicts.

validator: Callable = None

An optional callable that will receive a fully constructed record for one final validation before returning or yielding a single record. Records that do not pass this validation will also increment the `invalid_count`.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

F

G

I

L

M

N

O