subwabbit Documentation

Release 2.0.0

Matej Jakimov

Jun 14, 2019

Contents

For Kaggle playing use official vowpalwabbit package, for production use subwabbit.

subwabbit is Python wrapper around great Vowpal Wabbit tool that aims to be as fast as Vowpal itself. It is ideal for real time use, when many lines need to be scored in just few milliseconds or when high throughput is required.

Advantages:

- more then 4x faster then official Python wrapper
- good latency guarantees - give 10ms for prediction and it will end in 10ms
- explainability - API for explaining prediction value
- use just `vw` CLI - no compiling
- proven by reliably running in production at Seznam.cz where it makes hundreds of thousands of predictions per second per machine

CHAPTER 1

Documentation

Full documentation can be found on Read the docs.

CHAPTER 2

Requirements

- Python 3.4+
- Vowpal Wabbit

You can install Vowpal Wabbit by running:

```
sudo apt-get install vowpal-wabbit
```

on Debian-based systems or by using Homebrew:

```
brew install vowpal-wabbit
```

You can also build Vowpal Wabbit from source, see instructions.

subwabbit will probably work on other Pythons than 3.4+ but it is not tested (contribution welcomed).

CHAPTER 3

Installation

```
pip install subwabbit
```

CHAPTER 4

Example use

```
from subwabbit import VowpalWabbitProcess, VowpalWabbitDummyFormatter

vw = VowpalWabbitProcess(VowpalWabbitDummyFormatter(), ['-q', 'ab'])

common_features = '|a common_feature1:1.5 common_feature2:-0.3'
items_features = [
    '|b item123',
    '|b item456',
    '|b item789'
]

for prediction in vw.predict(common_features, items_features, timeout=0.001):
    print(prediction)
0.4
0.5
0.6
```

This is the simplest use of *subwabbit* library. You have some common features that describe context - it can be location of user or daytime for example. Then there is collection of items to score, each item has its specific features. Use of *timeout* argument means "compute as many predictions as you can in 1ms", then stop.

4.1 More advanced use

With simple implementation above you will not use key feature of *subwabbit*: **you can format your vw lines while Vowpal is busy with computing predictions**. By using this trick, you can get great speedup and VW lines formatting abstraction as a bonus.

Suppose we have features as dicts:

```
common_features = {
    'common_feature1': 1.5,
    'common_feature2': -0.3
```

(continues on next page)

(continued from previous page)

```
}

items_features = [
    {'id': 'item123'},
    {'id': 'item456'},
    {'id': 'item789'}
]
```

Then implementation with use of formatter can look like this:

```
from subwabbit import VowpalWabbitBaseFormatter, VowpalWabbitProcess

class MyVowpalWabbitFormatter(VowpalWabbitBaseFormatter):

    def format_common_features(self, common_features, debug_info=None):
        return '|a ccommon_feature1:{:.2f} common_feature2:{:.2f}'.format(
            common_features['common_feature1'],
            common_features['common_feature2']
        )

    def format_item_features(self, common_features, item_features, debug_info=None):
        return '|b {}'.format(item_features['id'])

vw = VowpalWabbitProcess(MyVowpalWabbitFormatter(), ['-q', 'ab'])

for prediction in vw.predict(common_features, items_features, timeout=0.001):
    print(prediction)
0.4
0.5
0.6
```

CHAPTER 5

Benchmarks

Benchmarks were made on logistic regression model with L2 regularization and with many quadratic combinations to mimic real-world use case. Real dataset containing 1000 contexts and 3000 items was used. Model was pretrained on this dataset with random labels generated. You can see used features at:

- *tests/benchmarks/requests.json*
- *tests/benchmarks/items.json*

```
# Prepare environment
pip install pandas vowpalwabbit
cd tests/benchmarks
# benchmarks depends a lot whether Vowpal is trained or just initialized
python pretrain_model.py

# Benchmark official Python client
python benchmark_pyvw.py

# Benchmark blocking implementation
python benchmark_blocking_implementation.py

# Benchmark nonblocking implementation
python benchmark_blocking_implementation.py
```

5.1 Benchmark results

Results on Dell Latitude E7470 with Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz.

Table shows how many lines implementation can predict in 10ms:

	pyvw	subwabbit
mean	239.461000	1033.70000
min	83.000000	100.00000
25%	192.750000	650.00000
50%	240.000000	1000.00000
75%	288.000000	1350.00000
90%	316.000000	1600.00000
99%	349.000000	1900.00000
max	362.000000	2050.00000

subwabbit is in average more then **4x** faster than official Python wrapper.

CHAPTER 6

License

CHAPTER 7

Contents

7.1 Throughput vs. latency

There are two implementations of *subwabbit.base.VowpalWabbitBaseModel*. Both implementations run `vw` subprocess and communicates with subprocess through pipes, but implementations differ in whether pipe is blocking or nonblocking.

7.1.1 Blocking

subwabbit.blocking.VowpalWabbitProcess

Blocking implementation use buffered binary IO. When *predict()* method is called, there is loop that:

- creates batch of VW lines
- sends this batch to Vowpal and flush Python-side buffer into system pipe buffer
- waits for predictions from last but one batch (writing is one batch ahead, so Vowpal should always be busy with processing lines)

There is also *train()* method that looks very similar, but usually you run training on instance with *write_only=True* so there is no need to wait for predictions.

7.1.2 Nonblocking

- *subwabbit.nonblocking.VowpalWabbitNonBlockingProcess*

Warning: Nonblocking implementation is only available for Linux based systems.

Warning: Training is not implemented for nonblocking variant.

Blocking implementation has great throughput, depends on features you have and arguments of vw process, it can be even optimal, so Vowpal itself is a bottleneck. However, due to blocking system calls, it can miss *timeout*. That is unacceptable if there is SLO with low-latency requirements.

Nonblocking implementation works similar to blocking, but it does not block for system calls when there are no predictions to read or system level buffer for VW lines is full, which helps to keep latencies very stable.

There is comparison of running time of `predict()` method with *timeout* set to 10ms:

	pyvw	blocking	nonblocking
mean	0.010039	0.010929	0.009473
min	0.010012	0.010054	0.009049
25%	0.010025	0.010130	0.009142
50%	0.010036	0.010312	0.009355
75%	0.010048	0.010630	0.009804
90%	0.010063	0.010950	0.010024
99%	0.010091	0.013289	0.010140
max	0.010138	0.468903	0.010999

Nonblocking implementation reduced latency peaks significantly, from almost 460ms to just 1ms.

Nonblocking implementation makes more system calls with smaller batches then blocking implementation and it comes with price of slightly lower throughput.

Predicted lines per request:

	pyvw	blocking	nonblocking
mean	239.461000	1033.70000	911.890000
min	83.000000	100.00000	0.000000
25%	192.750000	650.00000	552.000000
50%	240.000000	1000.00000	841.500000
75%	288.000000	1350.00000	1271.750000
90%	316.000000	1600.00000	1574.000000
99%	349.000000	1900.00000	1900.130000
max	362.000000	2050.00000	2022.000000

Note: Nonblocking implementation may have even zero predictions per call. It can happen due to previous call not having enough time to clean buffers before timeout, thus next call has to clean buffers and that can take all of it's time. See `predict()` *metrics* argument for details how to monitor this behavior.

7.2 Monitoring and debugging

This section gives overview of *subwabbit* monitoring and debugging capabilities.

7.2.1 Monitoring

It is good practice to monitor your system's behavior and fire an alert when system behavior changes.

Both blocking and nonblocking implementations of `predict()` can collect some metrics that can be helpful. There are two kinds of metrics:

- `metrics` - one numeric measurment per one call of *predict()* method. They are relatively cheap to collect and should be monitored in production.
- `detailed_metrics` - more measurements per one call of *predict()*. Each metric value is a list containing tuple `(time, numeric value)`. Their collection brings some overhead (e.g. reallocation of memory for growing lists of measurements). They are useful for profiling and can answer questions like "What is the bottleneck, formatting Vowpal lines or Vowpal itself?" or "Can change in some parameter bring some additional performance?".

See API documentation for more details about collected metrics for specific implementation.

See example of visualizing `detailed_metrics`:

```
pip install jupyter pandas matplotlib
jupyter notebook examples/Detailed-metrics.ipynb
```

7.2.2 Debugging

Sometimes it is useful to save some internal state like final formatted VW line. For these cases you can use `debug_info` parameter, which can be passed both to `predict()` and `train()` methods and which is passed to all following `subwabbit.base.VowpalWabbitBaseFormatter` calls and to private method calls. You can pass dict for example and fill it by some useful information.

7.3 Explaining predictions

It is practical to understand your model. There are various ways how to gain some insights about your model behavior, see for example excellent Dan Becker's tutorial on Kaggle: https://www.kaggle.com/learn/machine-learning-explainability .

Vowpal Wabbit offers various options how to inspect learned weights, **subwabbit** helps with use of *audit mode*. It allows to easily compute which features contributes the most for particular line's prediction.

7.3.1 How to explain prediction

At first, you need to turn on *audit_mode* by passing `audit_mode=True` argument to `subwabbit.base.VowpalWabbitBaseModel` constructor.

Warning: When audit mode is turned on, it is not possible to call `predict()` and `train()` methods.

Then use `explain_vw_line()` to retrieve explanation string. It will look like this: `c^c8*f^f102371218195482689 36:23365229:1:0.0220863@0 a^a342653813893595809 1*e^e115:1296634:0.2:0.0987504@0`

Features used for prediction are separated by *tab* and for each feature, there is string in format: `namespace^feature:hashindex:value:weight[@ssgrad]`

Then we can use `get_human_readable_explanation()` function to transform explanation string into more interpretable structure:

subwabbit.base.VowpalWabbitBaseFormatter.**get_human_readable_explanation**(*self*, *explanation_string: str*, *feature_translator: Any = None*) → List[Dict[KT, VT]]

Transform explanation string into more readable form. Every feature used for prediction is translated into this structure:

```
{
    # For each feature used in higher interaction there is a 2-tuple
    'names': [('Human readable namespace name 1', 'Human readable feature name 1'), ...],
    'original_feature_name': 'c^c8*f^f102'  # feature name how vowpal sees it,
    'hashindex': 123,  # Vowpal's internal hash of feature name
    'value': 0.123, # value for feature in input line
    'weight': -0.534, # weight learned by VW for this feature
    'potential': value * weight,
    'relative_potential': abs(potential) / sum_of_abs_potentials_for_all_features
}
```

Parameters

- **explanation_string** – Explanation string from `explain_vw_line()`
- **feature_translator** – Any object that can help you with translation of feature names into human readable form, for example some database connection. See `parse_element()`

Returns List of dicts, sorted by contribution to final score

You may also want to overwrite *parse_element()* method on your formatter to translate Vowpal feature names into human readable form, for example translate IDs to their names, potentialy using some mapping in database.

7.3.2 Example

Feature importances can also be visualized in *Jupyter notebook*, see complete example of how to use *subwabbit* for explaining predictions:

```
pip install jupyter
jupyter notebook examples/Explaining-prediction.ipynb
```

7.3.3 Notes

Note: This explanation is valid if you use sparse features, since expected value of every feature is close to zero. When you use dense features, you should normalize your features. If you do not normalize to zero mean, explaining features by their absolute contribution is not informative because you also need to consider how feature value differs from some expected value of that feature. In this case, you should use SHAP values for better interpretability, see https://www.kaggle.com/learn/machine-learning-explainability for more details. You still may find *subwabbit* explaining functionality useful, but interpreting results results won't be straightforward.

Note: In case you have correlated features, it is better to sum their potentials and consider them as single feature, otherwise you may underestimate influence of these features.

7.4 API

7.4.1 Base classes

class `subwabbit.base.`**`VowpalWabbitBaseFormatter`**

Formatter translates structured information about context and items to Vowpal Wabbit's input format: https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Input-format

It also can implement reverse translation, from Vowpal Wabbits feature names into human readable feature names.

`format_common_features`(*common_features: Any*, *debug_info: Any = None*) → str

Return part of VW line with features that are common for one call of predict/train. This method will run just once per one call of `subwabbit.base.VowpalWabbitBaseModel`'s *predict()* or *train()* method.

Parameters

- **`common_features`** – Features common for all items
- **`debug_info`** – Optional dict that can be filled by information useful for debugging

Returns Part of line that is common for each item in one call. Returned string has to start with '|' symbol.

`format_item_features`(*common_features: Any*, *item_features: Any*, *debug_info: Any = None*) → str

Return part of VW line with features specific to each item. This method will run for each item per one call of `subwabbit.base.VowpalWabbitBaseModel`'s *predict()* or *train()* method.

Note: It is a good idea to cache results of this method.

Parameters

- **`common_features`** – Features common for all items
- **`item_features`** – Features for item
- **`debug_info`** – Optional dict that can be filled by information useful for debugging

Returns

Part of line that is specific for item. Depends on whether namespaces are used or not in `format_common_features` method:

- namespaces are used: returned string has to start with `'|NAMESPACE_NAME'` where *NAMESPACE_NAME* is the name of some namespace

- namespaces are not used: returned string should not contain '|' symbol

get_formatted_example(*common_line_part: str*, *item_line_part: str*, *label: Optional[float] = None*, *weight: Optional[float] = None*, *debug_info: Optional[Dict[Any, Any]] = None*)

Compose valid VW line from its common and item-dependent parts.

Parameters

- **common_line_part** – Part of line that is common for each item in one call.
- **item_line_part** – Part of line specific for each item
- **label** – Label of this row
- **weight** – Optional weight of row
- **debug_info** – Optional dict that can be filled by information useful for debugging

Returns One VW line in input format: https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Input-format

get_human_readable_explanation(*explanation_string: str*, *feature_translator: Any = None*) → List[Dict[KT, VT]]

Transform explanation string into more readable form. Every feature used for prediction is translated into this structure:

```
{
    # For each feature used in higher interaction there is a 2-tuple
    'names': [('Human readable namespace name 1', 'Human readable feature␣
↪name 1'), ...],
    'original_feature_name': 'c^c8*f^f102'  # feature name how vowpal sees it,
    'hashindex': 123,  # Vowpal's internal hash of feature name
    'value': 0.123, # value for feature in input line
    'weight': -0.534, # weight learned by VW for this feature
    'potential': value * weight,
    'relative_potential': abs(potential) / sum_of_abs_potentials_for_all_
↪features
}
```

Parameters

- **explanation_string** – Explanation string from `explain_vw_line()`
- **feature_translator** – Any object that can help you with translation of feature names into human readable form, for example some database connection. See `parse_element()`

Returns List of dicts, sorted by contribution to final score

get_human_readable_explanation_html(*explanation_string: str*, *feature_translator: Any = None*, *max_rows: Optional[int] = None*)

Visualize importance of features in Jupyter notebook.

Parameters

- **explanation_string** – Explanation string from `explain_vw_line()`
- **feature_translator** – Any object that can help you with translation, e.g. some database connection.
- **max_rows** – Maximum number of most important features. None return all used features.

Returns *IPython.core.display.HTML*

`parse_element`(*element: str*, *feature_translator: Any = None*) → Tuple[str, str]

This method is supposed to translate namespace name and feature name to human readable form.

For example, element can be "a_item_id^i123" and result can be ('Item ID', 'News of the day: ID of item is 123')

Parameters

- **`element`** – namespace name and feature name, e.g. a_item_id^i123
- **`feature_translator`** – Any object that can help you with translation, e.g. some database connection

Returns tuple(human understandable namespace name, human understandable feature name)

class `subwabbit.base.`**`VowpalWabbitDummyFormatter`**

Formatter that assumes that either common features and item features are already formatted VW input format strings.

class `subwabbit.base.`**`VowpalWabbitBaseModel`**(*formatter: subwabbit.base.VowpalWabbitBaseFormatter*)

Declaration of Vowpal Wabbit model interface.

`explain_vw_line`(*vw_line: str*, *link_function: bool = False*)

Uses VW audit mode to inspect weights used for prediction. Audit mode has to be turned on by passing `audit_mode=True` to constructor.

Parameters

- **`vw_line`** – String in VW line format
- **`link_function`** – If your model use link function, pass True

Returns (raw prediction without use of link function, explanation string)

`predict`(*common_features: Any, items_features: Iterable[Any], timeout: Optional[float] = None, debug_info: Any = None, metrics: Optional[Dict[KT, VT]] = None, detailed_metrics: Optional[Dict[KT, VT]] = None*) → Iterable[float]

Transforms iterable with item features to iterator of predictions.

Parameters

- **`common_features`** – Features common for all items
- **`items_features`** – Iterable with features for each item
- **`timeout`** – Optionally specify how much time in seconds is desired for computing predictions. In case timeout is passed, returned iterator can has less items that items features iterable.
- **`debug_info`** – Some object that can be filled by information useful for debugging.
- **`metrics`** – Optional dict that is populated with some metrics that are good to monitor.
- **`detailed_metrics`** – Optional dict with more detailed (and more time consuming) metrics that are good for debugging and profiling.

Returns Iterable with predictions for each item from `items_features`

`train`(*common_features: Any, items_features: Iterable[Any], labels: Iterable[float], weights: Iterable[Optional[float]], debug_info: Any = None*) → None

Transform features, label and weight into VW line format and send it to Vowpal.

Parameters

- **common_features** – Features common for all items
- **items_features** – Iterable with features for each item
- **labels** – Iterable with same length as items features with label for each item
- **weights** – Iterable with same length as items features with optional weight for each item
- **debug_info** – Some object that can be filled by information useful for debugging

7.4.2 Blocking implementation

class subwabbit.blocking.**VowpalWabbitProcess**(*formatter: subwabbit.base.VowpalWabbitBaseFormatter, vw_args: List[T], batch_size: int = 20, write_only: bool = False, audit_mode: bool = False*)

Class representing Vowpal Wabbit model. It runs `vw` command through subprocess library and communicates through pipes.

__init__(*formatter: subwabbit.base.VowpalWabbitBaseFormatter, vw_args: List[T], batch_size: int = 20, write_only: bool = False, audit_mode: bool = False*)

Parameters

- **formatter** – Instance of `subwabbit.base.VowpalWabbitBaseFormatter`
- **vw_args** – List of command line arguments for vw command, eg. [‘-q’, ‘::’] This list MUST NOT specify *-p* argument for *vw* command
- **batch_size** – Number of lines communicated to Vowpal in one system call, has influence on performance. Smaller batches slightly reduces latencies and throughput.
- **write_only** – whether we expect to get predictions or we will just train This can greatly improve training performance but disables predicting.
- **audit_mode** – When set to True, VW is launched in audit mode with *-a* argument (overwrites *-t* argument). This allows to run *explain_vw_line* and *get_human_readable_explanation* methods.

> **Warning:** WARNING: When audit_mode is turned on, it is not possible to call other methods then *explain_vw_line*.

close()

Gracefully stop Vowpal Wabbit process

explain_vw_line(*vw_line: str*, *link_function=False*)

Uses VW audit mode to inspect weights used for prediction. Audit mode has to be turned on by passing `audit_mode=True` to constructor.

Parameters

- **vw_line** – String in VW line format
- **link_function** – If your model use link function, pass True

Returns (raw prediction without use of link function, explanation string)

predict(*common_features: Any, items_features: Iterable[Any], timeout: Optional[float] = None, debug_info: Any = None, metrics: Optional[Dict[KT, VT]] = None, detailed_metrics: Optional[Dict[KT, VT]] = None*) → Iterable[float]

Transforms iterable with item features to iterator of predictions.

Parameters

- **common_features** – Features common for all items
- **items_features** – Iterable with features for each item
- **timeout** – Optionally specify how much time in seconds is desired for computing predictions. In case timeout is passed, returned iterator can has less items that items features iterable.
- **debug_info** – Some object that can be filled by information useful for debugging.
- **metrics** – Optional dict populated with metrics that are good to monitor:
  - `prepare_time` - Time from call start to start of prediction loop, including `format_common_features` call
  - `total_time` - Total time spend in predict call
  - `num_lines` - Count of predictions performed
- **detailed_metrics** –

  Optional dict with more detailed (and more time consuming) metrics that are good for debugging and profiling:

  - `generating_lines_time` - time spent by generating VW line
  - `sending_lines_time` - time spent by sending VW lines to OS pipe buffer
  - `receiving_lines_time` - time spent by reading predictions from OS pipe buffer

  For each key, there will be list of tuples (time, metric value).

Returns Iterable with predictions for each item from `items_features`

train(*common_features: Any, items_features: Iterable[Any], labels: Iterable[float], weights: Iterable[Optional[float]], debug_info: Any = None*) → None

Transform features, label and weight into VW line format and send it to Vowpal.

Parameters

- **common_features** – Features common for all items
- **items_features** – Iterable with features for each item
- **labels** – Iterable with same length as items features with label for each item
- **weights** – Iterable with same length as items features with optional weight for each item
- **debug_info** – Some object that can be filled by information useful for debugging

7.4.3 Nonblocking implementation

class subwabbit.nonblocking.**VowpalWabbitNonBlockingProcess**(*formatter: subwabbit.base.VowpalWabbitBaseFormatter, vw_args: List[T], batch_size: int = 20, audit_mode: bool = False, max_pending_lines: int = 20, write_timeout_ms: float = 0.001, pipe_buffer_size_bytes: Optional[int] = None*)

Class representing Vowpal Wabbit model. It runs vw bash command through subprocess library and communicates through non-blocking pipes.

> **Warning:** Available on Linux only.

__init__(*formatter: subwabbit.base.VowpalWabbitBaseFormatter, vw_args: List[T], batch_size: int = 20, audit_mode: bool = False, max_pending_lines: int = 20, write_timeout_ms: float = 0.001, pipe_buffer_size_bytes: Optional[int] = None*)

Parameters

- **formatter** – Instance of `subwabbit.base.VowpalWabbitBaseFormatter`
- **vw_args** – List of command line arguments for vw command, eg. [‘-q’, ‘::’] This list MUST NOT specify *-p* argument for *vw* command
- **batch_size** – Maximum number of lines communicated to Vowpal in one system call. Smaller batches means less system calls overhead, but also higher risk of keeping mess for other calls.
- **audit_mode** – When turned on, VW is launched in audit mode with -*a* argument (overwrites -*t* argument). This allows to run *explain_vw_line* and *get_human_readable_explanation* methods.
- **max_pending_lines** – How many lines can wait for prediction in buffers. Recommended to set it to same value as *batch_size*, but it can be higher.
- **write_timeout_ms** – When *predict* is called with timeout, then *write_timeout_ms* before timeout sending lines to vowpal stops. It provides time to finish work without keeping mess that next call have to clean.
- **pipe_buffer_size_bytes** – Optionally set size of system buffer for sending lines to Vowpal. None means use default buffer size, for more details see http://man7.org/linux/man-pages/man7/pipe.7.html and `detailed_metrics` argument of `predict()` method

> **Warning:** WARNING: When audit_mode is turned on, it is not possible to call other methods then *explain_vw_line*.

cleanup(*deadline: Optional[float] = None*, *debug_info: Any = None*)

Cleans buffers after previous calls

Parameters **deadline** – Optional unix timestamp to end

explain_vw_line(*vw_line: str*, *link_function=False*)

Uses VW audit mode to inspect weights used for prediction. Audit mode has to be turned on by passing `audit_mode=True` to constructor.

Parameters

- **vw_line** – String in VW line format
- **link_function** – If your model use link function, pass True

Returns (raw prediction without use of link function, explanation string)

predict(*common_features: Any, items_features: Iterable[Any], timeout: Optional[float] = None, debug_info: Any = None, metrics: Optional[Dict[KT, VT]] = None, detailed_metrics: Optional[Dict[KT, VT]] = None*) → Iterable[float]

Transforms iterable with item features to iterator of predictions.

Parameters

- **common_features** – Features common for all items
- **items_features** – Iterable with features for each item
- **timeout** – Optionally specify how much time in seconds is desired for computing predictions. In case timeout is passed, returned iterator can has less items that items features iterable.
- **debug_info** – Some object that can be filled by information useful for debugging.
- **metrics** – Optional dict populated with metrics that are good to monitor:
  - `cleanup_time` - Time spent on cleaning buffers after last calls
  - `before_cleanup_pending_lines` - Count of lines pending in buffers before cleaning
  - `after_cleanup_pending_lines` - Count of lines pending in buffers after cleaning
  - `prepare_time` - Time from call start to start of prediction loop, including `format_common_features` call
  - `total_time` - Total time spend in predict call
  - `num_lines` - Count of predictions performed
- **detailed_metrics** –

  Optional dict with more detailed (and more time consuming) metrics that are good for debugging and profiling:

  - `sending_bytes` - number of bytes (VW lines) sent to OS pipe buffer
  - `receiving_bytes` - number of bytes (predictions) received from OS pipe buffer
  - `pending_lines` - number of pending lines sent to VW at the time
  - `generating_lines_time` - time spent by generating VW lines batch
  - `sending_lines_time` - time spent by sending lines to OS pipe buffer
  - `receiving_lines_time` - time spent by receiving predictions from OS pipe buffer

  For each key, there will be list of tuples (time, metric value).

Returns Iterable with predictions for each item from `items_features`

`train`(*common_features: Any, items_features: Iterable[Any], labels: Iterable[float], weights: Iterable[Optional[float]], debug_info: Any = None*) → None

Transform features, label and weight into VW line format and send it to Vowpal.

Parameters

- **`common_features`** – Features common for all items
- **`items_features`** – Iterable with features for each item
- **`labels`** – Iterable with same length as items features with label for each item
- **`weights`** – Iterable with same length as items features with optional weight for each item
- **`debug_info`** – Some object that can be filled by information useful for debugging

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index