VeridicalFlow: a Python package for building trustworthy data science pipelines with PCS

James Duncan[*1], **Rush Kapoor**[†2], **Abhineet Agarwal**[‡3], **Chandan Singh**[§2], **and Bin Yu**[1, 2]

1 Statistics Department, University of California, Berkeley **2** EECS Department, University of California, Berkeley **3** Physics Department, University of California, Berkeley

Summary

`VeridicalFlow` is a Python package that simplifies building reproducible and trustworthy data science pipelines using the PCS (predictability-computability-stability) framework (Yu & Kumbier, 2020). It provides users with a simple interface for stability analysis, i.e. checking the robustness of results from a data science pipeline to various judgement calls made during modeling. This ensures that arbitrary judgement calls made by data practitioners (e.g. specifying a default imputation strategy) do not dramatically alter the final conclusions made in a modeling pipeline. In addition to wrappers facilitating stability analysis, `VeridicalFlow` also automates many cumbersome coding aspects of Python pipelines, including experiment tracking and saving, parallelization, and caching, all through integrations with existing Python packages. Overall, the package helps to code using the PCS framework, by screening models for predictive performance, helping automate computation, and facilitating stability analysis.

Statement of need

Predictability, computability, and stability are central concerns in modern statistical/machine learning practice, as they are required to help vet that findings reflect reality, can be reasonably computed, and are robust as the many judgment calls during the data science life cycle which often go unchecked (Yu & Kumbier, 2020).

The package focuses on stability but also provides wrappers to help support and improve predictability and computability. Stability is a common-sense principle related to notions of scientific reproducibility (Ivie & Thain, 2018), sample variability, robust statistics, sensitivity analysis (Saltelli, 2002), and stability in numerical analysis and control theory. Moreover, stability serves as a prerequisite for understanding which parts of a model will generalize and can be interpreted (Murdoch et al., 2019).

Importantly, current software packages offer very little support to facilitate stability analyses. `VeridicalFlow` helps fill this gap by making stability analysis simple, reproducible, and computationally efficient. This enables a practitioner to represent a pipeline with many different perturbations in a simple-to-code way while using prediction analysis as a reality check to screen out poor models.

---

[*]Equal contribution
[†]Equal contribution
[‡]Equal contribution
[§]Equal contribution

Features

Using `VeridicalFlows`'s simple wrappers easily enables many best practices for data science and makes writing powerful pipelines straightforward.

Stability	Computability	Reproducibility
Replace a single function (e.g. preprocessing) with a set of functions representing different judgment calls and easily assess the stability of downstream results	Automatic parallelization and caching throughout the pipeline	Automatic experiment tracking and saving

The main features of `VeridicalFlow` center around stability analysis, a method for evaluating the constancy of some target quantity relative to a set of reasonable or realistic perturbations. The central concept is the `Vset`, short for "veridical set," which replaces a given static pipeline step with a set of functions subject to different pipeline perturbations that are documented and argued for via PCS documentation (Yu & Kumbier, 2020). Then, a set of useful analysis functions and computations enable simple assessment of the pipeline's stability to these perturbations on top of predictive screening for reality checks to filter unstable pipeline paths from further analysis.

A stability analysis example

1. Define stability target

In the example below, we will probe the stability of the permutation feature importance metric for random forest relative to data resampling, data preprocessing, and model hyperparameter perturbations. Below, we create a `Vset` which applies three custom data preprocessing functions and another that calculates the permutation importance metric via the function `sklearn.inspection.permutation_importance`.

```python
from vflow import Vset, build_vset
from sklearn.impute import KNNImputer, SimpleImputer
from sklearn.inspection import permutation_importance

preproc_list = [SimpleImputer(strategy='mean'),
                SimpleImputer(strategy='median'),
                KNNImputer()]

# create a Vset which varies over preproc_list
# we use output_matching=True to ensure that preprocessing strategies
# match throughout the pipeline
preproc_set = Vset('preproc', preproc_list, ['mean', 'med', 'knn'],
                   output_matching=True)

# create the feature importance Vset using helper build_vset
feat_imp_set = build_vset('feat_imp', permutation_importance,
                          n_repeats=4)
```

2. Define model hyperparameter perturbations

We can also specify modeling perturbations, both within a single class of models (hyperparameter perturbations) and across different classes. Here we'll use the helper `build_vset` to create hyperparameter perturbations for random forest.

```python
from sklearn.ensemble import RandomForestRegressor as RF

# hyperparameters to try
RF_params = {
    'n_estimators': [100, 300],
    'min_samples_split': [2, 10]
}

# we could instead pass a list of distinct models
# and corresponding param dicts
RF_set = build_vset('RF', RF, param_dict=RF_params)
```

3. Define data perturbations

For stability analysis, it is often useful to add data perturbations such as the bootstrap in order to assess stability over resampling variability in the data.

```python
from sklearn.utils import resample

# create a Vset for bootstrapping from data 100 times
# we use lazy=True so that the data will not be resampled until needed
boot_set = build_vset('boot', resample, reps=100, lazy=True)
```

4. Fit all models for all combinations of resampling and preprocessing

Now we can load in our data and fit each of the four random forest models to the 300 combinations of resampled training data and preprocessing functions.

```python
from vflow import init_args

# read in some data
X_train, y_train, X_val, y_val = ...

# wrap data for use with vflow
X_train, y_train, X_val, y_val = \
    init_args([X_train, y_train, X_val, y_val])

# bootstrap from training data by calling boot_fun
X_trains, y_trains = boot_set(X_train, y_train)

# apply three preprocessing methods to each bootstrap sample
X_trains = preproc_set.fit_transform(X_trains)

# fit the 4 RF models to each of the boot/preproc combos
RF_set.fit(X_trains, y_trains)
```

We can examine the pipeline graph to see what happened so far using the utility function build_graph, which results in Figure 1.

```python
from vflow import build_graph
build_graph(RF_set)
```
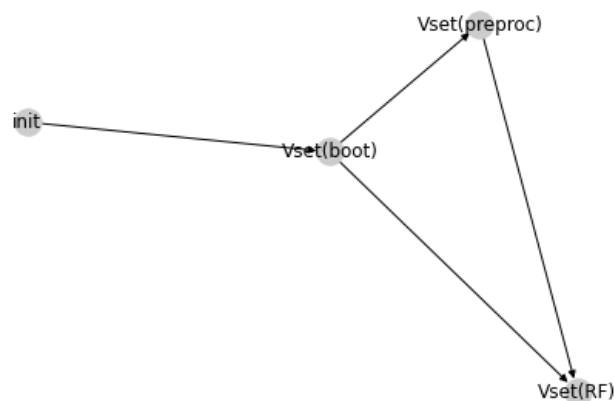
Figure 1: The pipeline graph that results from fitting `RF_set`.

5. Calculate feature importances and perturbation statistics

Finally, we calculate the importance metric and examine its mean and standard deviation across bootstrap perturbations for each combination of data preprocessing and modeling hyperparameters. This allows us to assess the stability of the feature importances conditioned on different pipeline paths:

```python
from vflow import dict_to_df, perturbation_stats

# calculate importances
importances = feat_imp_set(RF_set.out,
                           preproc_set.fit_transform(X_val), y_val)

# the helper dict_to_df converts the output to a pandas.DataFrame and
# using param_key='out' separates the importance dict into multiple cols
importances_df = dict_to_df(importances, param_key='out')

# get count, mean, and std of the permutation importances
perturbation_stats(importances_df, 'preproc', 'RF',
                   wrt='out-importances_mean',
                   prefix='X', split=True)
```

preproc	RF	X-count	X0-mean	X0-std	X1-mean	X1-std	X2-mean	X2-std
knn	(n_estimators=100, min_samples_split=10)	10	-0.004982	0.008000	0.008937	0.017888	1.501685	0.083409
knn	(n_estimators=100, min_samples_split=2)	10	-0.000434	0.012686	0.006632	0.014363	1.508886	0.054652
knn	(n_estimators=300, min_samples_split=10)	10	-0.002849	0.010865	0.010854	0.010772	1.506707	0.071501
knn	(n_estimators=300, min_samples_split=2)	10	-0.005079	0.009201	0.006587	0.012941	1.534967	0.081649
...	...	...	...	...	...	...	...	...
med	(n_estimators=100, min_samples_split=10)	10	-0.014884	0.008509	-0.002013	0.015589	1.635742	0.088294
med	(n_estimators=100, min_samples_split=2)	10	-0.009400	0.013442	-0.000890	0.017286	1.631176	0.069840
med	(n_estimators=300, min_samples_split=10)	10	-0.008485	0.010060	-0.002412	0.018975	1.583543	0.081383
med	(n_estimators=300, min_samples_split=2)	10	-0.010600	0.011587	-0.005605	0.016469	1.627945	0.048281

Figure 2: Perturbation statistics of permutation feature importances.

As seen in Figure 2, we can filter over the data preprocessing and modeling perturbations via the helper `filter_vset_by_metric` to select the top combinations in terms of stability (or another metric of interest) and continue our analysis on a held-out test set.

Computation and tracking

The package also helps users to improve the efficiency of their computational pipelines. Computation is (optionally) handled through Ray (Moritz et al., 2018), which easily facilitates parallelization across different machines and along different perturbations of the pipeline. Caching is handled via joblib, so that individual parts of the pipeline do not need to be rerun. Moreover, `vflow` supports lazy evaluation of `Vsets`, as shown in the example above. Thus, computation and data can be deferred to when it is needed, saving on memory and allowing the pipeline graph to be built and examined before beginning computation.

Experiment-tracking and saving are (optionally) handled via integration with MLFlow (Zaharia et al., 2018), which enables automatic experiment tracking and saving.

Related packages

The code here heavily derives from the wonderful work of previous projects. It hinges on the data science infrastructure of Python, including packages such as pandas (McKinney & others, 2011), NumPy (Van Der Walt et al., 2011), and scikit-learn (Pedregosa et al., 2011) as well as newer projects such as imodels (Singh et al., 2021) and NetworkX (Hagberg & Conway, n.d.).

The functionality provided by `VeridicalFlow` is related to the `sklearn.pipeline.Pipe line` class but allows for more general pipeline steps (e.g. steps need not use the `fit` or `transform` methods) and for reuse of those steps in the same or other pipelines. Moreover, pipeline graphs in `VeridicalFlow` are generated dynamically by interactions between `Vsets`. This added flexibility of pipelines in `VeridicalFlow` is akin to the dynamic computational graphs in TensorFlow (via `tensorflow.keras.layers` and `tensorflow.function`) and Ray (via `@ray.remote`). Indeed, when a `Vset` is created with `is_async=True`, `VeridicalFlow`'s pipeline graph is backed by Ray's task graph.

Acknowledgements

References

Hagberg, A., & Conway, D. (n.d.). *NetworkX: Network analysis with python*.

Ivie, P., & Thain, D. (2018). Reproducibility in scientific computing. *ACM Computing Surveys (CSUR), 51*(3), 1–36. https://doi.org/10.1145/3186266

McKinney, W., & others. (2011). Pandas: A foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing, 14*(9), 1–9. https://doi.org/10.5281/zenodo.3509134

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., & others. (2018). Ray: A distributed framework for emerging AI applications. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 561–577.

Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R., & Yu, B. (2019). Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences*, *116*(44), 22071–22080. https://doi.org/10.1073/pnas.1900654116

Pedregosa, F., Varoquaux, G. ë. l., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., & others. (2011). Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, *12*, 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

Saltelli, A. (2002). Sensitivity analysis for importance assessment. *Risk Analysis*, *22*(3), 579–590. https://doi.org/10.1111/0272-4332.00040

Singh, C., Nasseri, K., Tan, Y. S., Tang, T., & Yu, B. (2021). Imodels: A python package for fitting interpretable models. *Journal of Open Source Software*, *6*(61), 3192. https://doi.org/10.21105/joss.03192

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22–30. https://doi.org/10.1109/mcse.2011.37

Yu, B., & Kumbier, K. (2020). Veridical data science. *Proceedings of the National Academy of Sciences*, *117*(8), 3920–3929. https://doi.org/10.1145/3336191.3372191

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., & others. (2018). Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, *41*(4), 39–45. https://doi.org/10.1145/3399579.3399867