SkData: Data Sets and Algorithm Evaluation Protocols in Python

James Bergstra[‡][*], Nicolas Pinto[§], David D. Cox[¶]

http://www.youtube.com/watch?v=u5amehIiImo

Abstract—Machine learning benchmark data sets come in all shapes and sizes, whereas classification algorithms assume sanitized input, such as (x, y) pairs with vector-valued input x and integer class label y. Researchers and practitioners know all too well how tedious it can be to get from the URL of a new data set to a NumPy ndarray suitable for e.g. pandas or sklearn. The SkData library handles that work for a growing number of benchmark data sets (small and large) so that one-off in-house scripts for downloading and parsing data sets can be replaced with library code that is reliable, community-tested, and documented. The SkData library also introduces an open-ended formalization of training and testing protocols that facilitates direct comparison with published research. This paper describes the usage and architecture of the SkData library.

Index Terms—machine learning, cross validation, reproducibility

Introduction

There is nothing standard about data sets for machine learning. The nature of data sets varies widely, from physical measurements of flower petals ([Iris]), to pixel values of tiny public domain images ([CIFAR-10]), to the movie watching habits of NetFlix users ([Netflix]). Some data sets are tiny and others are vast databases that push the limits of storage technology. Different data sets test different algorithms' abilities to make different kinds of statistical inference. Often a single data set may be used in several ways to evaluate multiple kinds of algorithm. This flexibility and undefined-ness makes it challenging to design software abstractions for data sets.

In contrast to the great variety of data sets though, researchers have condensed the variety of data sets to a much smaller set of machine learning problems. For example, a great deal of machine learning research addresses the *classification problem* of assigning an integer-valued *label* (y) to some vector of binary- or real-valued *features* (X). Many classification algorithms have been developed, such as Support Vector Machines, Decision Trees, and Nearest Neighbors. The reason that they are all called classification algorithms is that they provide a common mathematical interface.

While the neatness of these mathematical abstractions is reflected in the organization of machine learning libraries such as [sklearn], we believe there is a gap in Python's machine learning stack between raw data sets and such neat, abstract interfaces. Data, even when it is provided specifically to test classification algorithms, is seldom provided as (feature, label) pairs. Guidelines regarding standard experiment protocols (e.g. which data to use for training) are expressed informally in web page text if at all. The SkData library consolidates myriad little details of idiosyncratic data processing required to run experiments on standard data sets, and packages them as a library of reusable code. It serves as both a gateway to access a growing list of standard public data sets, and as a framework for expressing precise evaluation protocols that correspond to standard ways of using those data sets.

This paper introduces the SkData library ([SkData]) for accessing data sets in Python. SkData provides two levels of interface:

1) It provides *low-level* idiosyncratic logic for acquiring, unpacking, and parsing standard data sets so that they can be loaded into sensible Python data structures.
2) It provides *high-level* logic for evaluating machine learning algorithms using strictly controlled experiment protocols, so that it is easy to make direct, valid model comparisons.

These interfaces are provided on a data-set-by-data-set basis. All data sets supported by SkData provide a low-level interface. For a data set called `foo` the low-level interface would normally be provided a submodule called `foo.dataset`. SkData provides a high-level interface for some, but not all supported data sets. This high-level interface would normally be provided by submodule `foo.view`. The high-level modules provide one or more views of the low-level data which make the underlying data fit the form required by machine learning algorithms.

Relative to language-agnostic repositories (such as the [UCI] database of machine learning data sets), SkData provides Python code for downloading and loading diverse data representations into more standardized in-memory formats. Anyone using these data sets in a Python program would have to use something like the low-level routines in SkData anyway to simply load the data. Relative to standardized repositories such as [MLData], SkData provides convenient downloading and loading logic, as well as formal protocols (in Python) for model selection and evaluation. Relative to the [Pandas] Python library, SkData provides data set-specific logic for downloading, parsing, and model evaluation; Pandas provides useful data structures and statistical routines. It would make sense to use SkData and Pandas together, and future data set modules in SkData may use Pandas internally. The

* *Corresponding author: james.bergstra@uwaterloo.ca*
‡ *University of Waterloo*
§ *Massachusetts Institute of Technology*
¶ *Harvard University*

[PyTables] library provides a high-performance HDF5 wrapper. It would make sense to use SkData and PyTables together, such as for example for low-level SkData routines to store and manipulate downloaded data.

This paper is organized into the following sections:

1) Data set access (low-level)
2) Intro to experiment protocols (high-level)
3) Protocol case study: simple cross-validation
4) The experiment protocol
5) Command-line interface
6) Current list of data sets

Data Set Access (Low-level Interface)

There is nothing standard about data sets, and SkData's *low-level interface* correspondingly comprises many modules that are not meant to be formally interchangeable. Still, there are *informal* sorts of similarities in some aspects of what users want to do with data, at least in the context of doing machine learning. SkData's low-level modules provide logic for several common activities for most of the data sets supported by the library:

- downloading,
- verifying archive integrity,
- decompressing,
- loading into Python, and
- deleting cached data.

These common activities are typically implemented by methods on singleton classes within SkData's low-level modules. The data set class for the Labeled Faces in the Wild ([LFW]) data set provides a representative example of what low-level data set objects look like. What follows is an abridged version of what appears in `skdata.lfw.dataset`.

```
"""
<Description of data set>

<Citations to key publications>
"""

published_scores = {'PC11': .881, ...}

url_to_data_file = ...
sha1_of_data_file = ...

class LFW(object):

    @property
    def home(self):
        """Return cache folder for this data set"""
        return os.path.join(
            skdata.data_home.get_data_home(),
            'lfw')

    def fetch(self, download_if_missing=True):
        """Return iff required data is in cache."""
        ...

    def clean_up(self):
        """Remove cached and downloaded files"""
        ...

    @property
    def meta(self):
        """Return meta-data as list of dicts"""
        ...
```

The next few sub-sections describe what the methods of this class (as a representative low-level data set classes) and other elements of the module are supposed to do. There is a convention that this low-level logic for each data (e.g. *foo*) should be written in a Python file called `skdata.foo.dataset`. Other projects may implement data set classes in whatever files are convenient. Technically, there is no requirement that the low-level routines adhere to any standard interface, because SkData includes no functions meant to work on *any* data set.

Context and Documentation

First, notice that the `dataset.py` file includes a significant docstring describing the data set and providing some history regarding its usage. This docstring should provide links to key publications that either introduced or used this data set.

If the data set has a home page, that should be documented here too. Many data sets' home pages maintain a table of benchmarks and pointers to influential model evaluation papers. It is appropriate to reproduce such tables in this `dataset.py` file either in the docstring, or, more helpfully, as a module-level Python dictionary (e.g. the `published_scores` module-level dictionary in our example). Such a dictionaries makes it easier to produce figures and tables showing performance relative to models from the literature.

Downloading and Deleting

Often the first order of business when dealing with a data set is to download it. Data sets come from a range of sources, but it is worth distinguishing those that can be downloaded freely (we will call these *public*) from the rest (*private*). The SkData library is suitable and useful for both public and private data, but it is more useful for public data sets because the original download from a canonical internet source can be automated. Whether a data set is private or public, the `dataset.py` file should include checksums for verifying the correctness of important data files when it makes sense to do so.

Most `dataset` modules use SkData's `get_data_home()` function to identify a local location for storing large files. This location defaults to `.skdata/` but it can be set via a `$SKDATA_ROOT` environment variable. In our code example, `LFW.home()` uses this mechanism to identify a location where it can store downloaded and decompressed data. The convention is that a dataset called `foo` would use `path.join(get_data_home(), 'foo')` as a persistent cache location.

The `fetch` method downloads, verifies the correctness-of, and decompresses the various files that make up the data set. It stores downloaded files within the folder returned by `LFW.home()`. If `download_if_missing` is False, then `fetch` raises an exception if the data is not present. When `fetch()` returns, it means that the data can be loaded (see below).

If a data set module downloads or creates files, then it should also provide a mechanism for deleting them. In our LFW example, the `clean_up` method recursively deletes the entire `LFW.home()` folder, erasing the downloaded data and all derived files. Other data sets may wish to provide a more fine-grained approach to clean-up that perhaps erase derived files, but not any archive files that cannot easily be replaced.

Decompressing, Parsing, and Loading

Experienced machine learning practitioners are well aware that in terms of files and formats, a data set may be just about anything.

Some of the more popular data sets in machine learning and computer vision include one or more of:

- Comma Separated Value (CSV) text files,
- XML documents (with idiosyncratic internal structure),
- Text files with ad-hoc formatting,
- Collections of image, movies, audio files,
- Matlab workspaces,
- Pickled NumPy `ndarray` objects, and
- HDF5 databases.

Correctly interpreting meta-data can be tricky and writing code to simply load media collections that include files with non-homogeneous formats, encoding types, sampling frequencies, color spaces, and so on can be tedious.

One of the main reasons for developing and releasing SkData was to save scientists the trouble of re-writing scripts that make sense of data set files. A low-level data set module should include the logic for reading, walking, parsing, etc. any and all raw archive files. This logic should turn those raw archive files into appropriate Python data structures such as lists, dictionaries, NumPy arrays, Panda data frames, and/or PyTables `Table` objects.

For example, the low-level LFW data set class's `meta` attribute is computed by parsing a few text files and walking the directory structure within `LFW.home()`. The `meta` property is a list of dictionaries enumerating what images are present, how large they are, what color space they use, and the name of the individual in each image. It does not include all the pixel data because, in our judgement, the pixel data required a lot of memory and could be provided instead by a *lazy array* (see [Dealing with Large Data] below). The LFW low-level module contains an additional method called `parse_pairs_file` which parses some additional archived text files describing the train/test splits that the LFW authors recommend using for the development and evaluation of algorithms. This may seem ad-hoc, and indeed it is. Low-level modules are meant to be particular to individual data sets, and not standardized.

There isn't a lot more to say about low-level dataset modules in general. Section [Current List of Data Sets] below enumerates the data sets currently in SkData that have some degree of low-level support, and that list continues to grow.

Intro to Experiment Protocols (High-level Interface)

Users who simply want a head start in getting Python access to downloaded data are well-served by the low-level modules, but users who want a framework to help them reproduce previous machine learning results by following specific experiment protocols will be more interested in using SkData's higher-level `view` interface. The next few sections describe the high-level protocol abstractions provided by SkData's various data set-specific `view` modules.

Background: Classification and Cross-Validation

Before we get into `view` module abstractions for experiment protocols, this section will introduce the machine learning methodology that these abstractions will ultimately provide.

SkData's high-level modules currently provide structure for classification problems. A classification problem, in machine learning terms, is a scenario in which *labels* (without loss of generality: integers) are to be predicted from *features*. If we wish to predict the name of an individual in a photograph, or categorize email as spam or not-spam, it is natural to look at these as classification problems.

It is useful to set this up formally. If $\mathscr{Y}$ is our set of possible labels, and $\mathscr{X}$ is the set of possible feature vectors, then a *classifier* is a mapping (or *model*) $m : \mathscr{X} \rightarrow \mathscr{Y}$. A *classification algorithm* is a procedure for selecting a particular model from a set $\mathscr{M}$ of possible models. Generally this selection is made on the basis of data that represent the sorts of features and labels that we believe will arise. If we write this belief as a joint density $P(x,y)$ over $\mathscr{X} \times \mathscr{Y}$ then we can write down one of the most important selection criteria for classification models:

$$\ell(m) = \mathbb{E}\left[\mathbb{I}_{\{y \neq m(x)\}}\right] \tag{1}$$

$$m^{(*)} = \operatorname{argmin}_{m \in \mathscr{M}} \ell(m) \tag{2}$$

Any function like the ℓ here that assigns a real-valued score to a model can be called a *loss* function. This particular loss function is called the Zero-One loss because it is the expected value of a random variable that is either Zero (when our classifier is wrong) or One (when our classifier predicts the label). In terms of end-of-the-day accuracy, $m^{(*)}$ is, by definition, the best model we could possibly choose. Classification algorithms represent various ways of minimizing various loss functions over various sets of models.

In practice, we cannot expect a mathematical expression for $P(x,y)$. Instead, we must content ourselves with a sample D of $< x,y >$ pairs. An enumeration of the various ways of using the examples in D to select and evaluate models from $\mathscr{M}$ is beyond the scope of this paper. (For more information, see e.g. [HTF09]). SkData is designed to support the full variety of such protocols, but in the interest of keeping this paper focused, we will only use what is called *simple cross-validation* to illustrate how SkData's high-level `view` modules make it easy to evaluate classification algorithms on a range of classification tasks.

Protocol Case Study: Simple Cross-Validation

Simple cross-validation is a technique for evaluating a learning algorithm (e.g. a classification algorithm), on the basis of a representative sample of independent, identically drawn (*iid*) $< x,y >$ pairs. It is helpful to think of a learning algorithm as encapsulating the selection criterion and optimization algorithm corresponding to Eqns 1 and 2, and as providing a mapping $A : \mathscr{D} \rightarrow \mathscr{M}$ from a data set to a model. Evaluating a classification algorithm means estimating how accurate it is likely to be on data it has never seen before. Simple cross-validation makes this estimate by partitioning all available data D into two disjoint subsets. The first subset D_{train} is called a *training* set; it is used to choose a model m from $\mathscr{M}$. The second subset D_{test} is called a *test* set; since this data was not used during training, it represents a sample of all data that the learning algorithm has never seen. Mathematically, simple cross-validation means evaluating an algorithm A as follows:

$$m = A(D_{\text{train}}) \tag{3}$$

$$\ell(A) = \frac{1}{|D_{\text{test}}|} \sum_{<x,y> \in D_{\text{test}}} \mathbb{I}_{\{y \neq m(x)\}} \tag{4}$$

The abstractions provided by SkData make it as easy to evaluate an algorithm on a data set as Eqns 3 and 4 suggest. Conveniently, the [sklearn] library provides learning algorithms such as `LinearSVC` that implement a methods `fit` and `predict` that correspond exactly to the requirements of Eqns. 3 and 4 respectively. As a convenience and debugging utility, SkData provides a simple wrapper called `SklearnClassifier` that makes it easy to apply any `sklearn` classifier to any SkData classification view. Using this wrapper, evaluating an SVM on the [Iris] data set for example, looks like this:

```
1 from sklearn.svm import LinearSVC
2 from skdata.base import SklearnClassifier
3 from skdata.iris.view import SimpleCrossValidation
4
5 # Create an evaluation protocol
6 iris_view = SimpleCrossValidation()
7
8 # Choose a learning algorithm
9 estimator = LinearSVC
10 algo = SklearnClassifier(estimator)
11
12 # Run the evaluation protocol
13 test_error = iris_view.protocol(algo)
14
15 # See what happened:
16 for report in algo.results['best_model']:
17     print report['train_name'], report['model']
18
19 for report in algo.results['loss']:
20     print report['task_name'], report['err_rate']
21
22 print "TL;DR: average test error:", test_error
```

The next few Subsections explain what these functions do, and suggest how Tasks and Protocols can be used to encode more elaborate types of evaluation.

Case Study Step 1: Creating a View

The first statement of our cross-validation code sample creates a *view* of the Iris data set.

```
6 iris_view = SimpleCrossValidation()
```

The `SimpleCrossValidation` class uses Iris data set's low-level interface to load features into a numpy `ndarray`, and generally prepare it for usage by sklearn. In general, a View may be configurable (e.g. how to partition D into training and testing sets) but this simple demonstration protocol does not require any parameters.

Case Study Step 2: Creating a Learning Algorithm

The next two statements of our cross-validation code sample create a *learning algorithm*, as a SkData class.

```
10 estimator = LinearSVC
11 algo = SklearnClassifier(estimator)
```

The argument to `SklearnClassifier` is a parameter-free function that constructs a `sklearn.Estimator` instance, ready to be fit to data. The `algo` object keeps track of the interactions between the `iris_view` protocol object and the `estimator` classifier object. When wrapping around sklearn's `Estimators` it is admittedly confusing to call `algo` the learning algorithm when `estimator` is also deserving of that name. The reason we call `algo` the learning algorithm here (rather than `estimator`) is that SkData's high-level modules expect a particular interface of learning algorithms. That high-level interface is defined by `skdata.base.LearningAlgo`.

The `SklearnClassifer` acts as an adapter that implements the `skdata.base.LearningAlgo` interface in terms of `sklearn.Estimator`. The class serves two roles: (1) it provides a reference implementation for how handle commands from a protocol object; (2) it supports unit tests for protocol classes in Skdata. Researchers are encouraged to implement their own `LearningAlgo` classes following the example of the `SklearnClassifier` class. Custom LearningAlgo classes can compute and save algorithm-specific statistics, and implement performance-enhancing hacks such as custom data iterators and pre-processing caches. The practice of appending a summary dictionary to the lists in self.results has proved useful in our own work, but it likely not the best technique for all scenarios. A `LearningAlgo` subclass should somehow record the results of model training and testing, but SkData's high-level `view` modules does not require that those results be stored in any particular way. We will see more about how a protocol object drives training and testing later in [The Evaluation Protocol].

Case Study Step 3: Evaluating the Learning Algorithm

The heavy lifting of the evaluation process is carried out by the `protocol()` call on line 14.

```
14 test_error = iris_view.protocol(algo)
15
16 # See what happened:
17 for report in algo.results['best_model']:
18     print report['train_name'], report['model']
19
20 for report in algo.results['loss']:
21     print report['task_name'], report['err_rate']
```

The `protocol` method encapsulates a sort of dialog between the `iris_view` object as a driver, and the `algo` object as a handler of commands from the driver. The protocol in question (`iris.view.SimpleCrossValidation`) happens to use just two kinds of command:

1) Learn the best model for training data
2) Evaluate a model on testing data

The first kind of command produces an entry in the `algo.results['best_model']` list. The second kind of command produces an entry in the `algo.results['loss']` list.

After the `protocol` method has returned, we can loop over these lists (as in lines 17-21) to obtain a summary of what happened during our evaluation protocol.

The Experiment Protocol

Now that we have seen the sort of code that SkData's high-level evaluation protocol is meant to support, the next few sections dig a little further into how it works.

The Protocol Container: `Task`

The main data type supporting SkData's experiment protocol is what we have called the `Task`. The `skdata.base` file defines the `Task` class, and it used in all aspects of the protocol layer. A `Task` instance represents a semantically labeled subsample of a data set. It is simply a dictionary container with access to elements by object attribute (it is a namespace), but it has two required attributes: `name` and `semantics`. The `name` attribute is a string that uniquely identifies this Task among all tasks involved in a Protocol. The `semantics` attribute is a string that identifies what *kind* of Task this is.

A task's semantics identifies (to the learning algorithm) which other attributes are present in the task object, and how they should be interpreted. For example, if a task object has `'vector_classification'` semantics, then it is expected to have (a) an `ndarray` attribute called `x` whose rows are examples and columns are features, and (b) an `ndarray` vector attribute `y` whose elements label the rows of `x`. If a task object instead has `'indexed_image_classification'` semantics, then it is expected to have (a) a sequence of RGBA image ndarrays in attribute `.all_images`, (b) a corresponding sequence of labels `.all_labels`, and (c) a sequence of integers `.idxs` that picks out the relevant items from `all_images` and `all_labels` as defined by NumPy's `take` function.

The set of semantics is meant to be open. In the future, SkData may have a data set for which none of these semantics applies. For example SkData may, in the future, provide access to aligned multi-lingual databases of text. At that point it may well be a good idea to define a `'phrase_translation'` task whose inputs and outputs are sequences of words. The new semantics string would cause existing learning algorithms to fail, but failing is reasonable because phrase translation is not obviously reducible to existing semantics.

The semantics identifiers employed so far in SkData include:

- `'vector_classification'`
- `'indexed_vector_classification'`
- `'indexed_image_classification'`
- `'image_match_indexed'`

Vector classification was explained above, it corresponds quite directly to the sort of `X` and `y` arguments expected by e.g. sklearn's `LinearSVC.fit`. The *indexed* semantics allow learning algorithms to cache example-wise pre-processing in certain protocols, such as K-fold cross-validation. The general idea is that Tasks with e.g. `'indexed_vector_classification'` semantics share the *same* `X` and `y` arrays, but use different index lists to denote different selections from `X` and `y`. Whenever different indexed tasks refer to the same rows of `X` and `y`, the learning algorithm can re-use cached pre-processing. The `'image_match_indexed'` semantics was introduced to accommodate the LFW data set in which image pairs are labeled according to whether they feature the same person or different people. Future data sets featuring labeled image pairs may leverage learning algorithms written for LFW by reusing the `'image_match_indexed'` semantics. Future data sets with new kinds of data may wish to use new semantics strings.

Protocol Commands (LearningAlgo Interface)

Now that we have established what Tasks are, we can describe the methods that a `LearningAlgo` must support in order to participate in the most basic protocols:

`best_model(task, valid=None)`
: Instruct a learning algorithm to find the best possible model for the given task, and return that model to the protocol driver. If a `valid` (validation) task is provided, then use it to detect overfitting on `train`.

`loss(model, task)`
: Instruct a learning algorithm to evaluate the given model for the given task. The returned value should be a floating point scalar, but the semantics of that scalar are defined by the semantics of the task.

`forget_task(task)`
: Instruct the learning algorithm to free any possible memory that has been used to cache computations related to this task, because the task will not be used again by the protocol.

These functions are meant to have side effects, in the sense that the `LearningAlgo` instance is expected to record statistics and summaries etc., but the `LearningAlgo` instance is expected *not* to cheat! For example, the `best_model` method should use *only* the examples in the `task` argument as training data. The interface is not designed to make this sort of cheating difficult to do, it is only designed to make cheating easy to avoid.

A `LearningAlgo` can also include additional methods for use by protocols. For example, one data set in SkData features a protocol that distinguishes between the selection of features and the selection of a classifier of those features. That protocol calls an additional method that is not widely used:

`retrain_classifier(model, task)`
: Instruct the learning algorithm, to retrain only the classifier, and not repeat any internal feature selection that has taken place.

When new protocols require new commands for learning algorithms, our policy is to add them. As evidenced by the short list of commands above, we have only had to do this once to date.

The SemanticsDelegator LearningAlgo

Authors of new `LearningAlgo` base classes may wish to inherit from `base.SemanticsDelegator` instead. The `SemanticsDelegator` class handles calls to e.g. `best_model` by appending the semantics string to the call name, and calling that more specialized function, e.g. `best_model_indexed_vector_classification`. While the number of protocol commands may be small, a new `LearningAlgo` subclass might implement some protocol commands quite differently for different semantics strings, with little code overlap. The `SemanticsDelegator` base class makes writing such `LearningAlgo` classes a little easier.

The `SklearnClassifier` uses the `SemanticsDelegator` in a different way, to facilitate a cascade of fallbacks from specialized semantics to more general ones. The indexed image tasks are converted first to indexed vector tasks, and then to non-indexed vector tasks before finally being handled by the `sklearn` classifier. This pattern of using machine learning reductions to solve a range of tasks with a smaller set of core learning routines is a powerful one, and a `LearningAlgo` subclass presents a natural place to implement this pattern.

Protocol Objects

Having looked at the `Task` and `LearningAlgo` classes, we are finally ready to look at that last piece of SkData's protocol layer: the Protocol objects themselves. Protocol objects (such as `iris.view.SimpleCrossValidation`) walk a learning algorithm through the process of running an experiment. To do so, they must provide a *view* of the data set they represent (e.g. Iris) that corresponds to one of the Task semantics. They must create Task objects from subsets of that view in order to call the methods of a `LearningAlgo`.

In the case study we looked at earlier, the call to `iris_view.protocol(algo)` constructed two Task objects corresponding to a training set (`train`) and a test set (`test`) of the Iris data and then did the following:

```
model = algo.best_model(train)
err = algo.loss(model, test)
return err
```

More elaborate protocols construct more task objects, and train and test more models, but typically the `protocol` methods are quite short. Doubly-nested K-fold cross-validation is probably the most complicated evaluation protocol, but it still consists essentially of two nested for loops calling `best_model` and `loss` using a single K-way data partition. It can be useful to implement longer protocols as iterators rather than methods so that they can be aborted early.

Dealing with Large Data

Generally, each data set module is free to deal with large data in a manner befitting its data set, although particular Task semantics constrain the data representations that can be used at the protocol layer. Two complementary techniques are used within the SkData library to keep memory and CPU usage under control when dealing with potentially enormous data sets. The first technique is to use the indexed Task semantics. Recall that when using indexed semantics, a Task includes an indexable data structure (e.g. `ndarray`, `DataFrame`, or `Table`) containing the whole of the data set D, and a vector of positions within that data structure indicating a subset of examples. Many indexed Task instances can be allocated at once because each indexed Task shares a pointer to a common data set. Only a vector of positions must be allocated for each Task, which is relatively small.

The second technique is to use the *lazy array* in `skdata.larray` as the indexable data structure for indexed Tasks. The `larray` can delay many transformations of an `ndarray` until elements are accessed by `__getitem__`. For example, if a protocol only requires the first 100 examples of a huge data set, then only those examples will be loaded and processed. The `larray` supports transformations such as re-indexing, elementwise functions, a lazy `zip`, and cacheing. Lazy evaluation together with cacheing makes it possible for protocol objects to pass very large data sets to learning algorithms, and for learning algorithms to treat very large data sets in sensible ways. The lazy array does not make batch learning algorithms into online ones, but it provides a mechanism for designing iterators so that online algorithms can traverse large numbers of examples in a cache-efficient way.

Command-line Interface

Some data sets also provide a `main.py` file that provides a command-line interface for operations such as downloading, visualizing, and deleting data. The LFW data set for example, has a simple main.py script that supports one command that downloads (if necessary) and visualizes a particular variant of the data using [glumpy].

```
python -c skdata/lfw/main.py show funneled
```

Several other data sets also have `main.py` scripts, which support various commands. These scripts are meant to follow the convention that running them with no arguments prints a usage description, but they may not all conform. In most cases, the scripts are very short and easy to read so go ahead and look at the source if the help message is lacking.

Current List of Data Sets

The SkData library currently provides some level of support for about 40 data sets (some data sets are parametrically related, not clearly distinct). The data sets marked with (*) provide the full set of low-level, high-level, and script interfaces described above. Details and references for each one can be found in the SkData project web page, wiki, and source code. Many of the synthetic data sets are inherited from the `sklearn` project; the authors have contributed most of the image data sets.

- Blobs
  - Synthetic: isotropic Gaussian blobs
- Boston
  - Real-estate features and prices
- Brodatz
  - Texture images
- CALTECH101
  - Med-res Images of 101 types of object
- CALTECH256
  - Med-res Images of 256 types of object
- CIFAR10 (*)
  - Low-res images of 10 types of object
- Convex
  - Small images of convex and non-convex shapes
- Digits
  - Small images of hand-written digigs
- Diabetes
  - Small non-synthetic temporal binary classification
- IICBU2008
  - Benchark suite for biological image analysis
- Iris (*)
  - Features and labels of iris specimens
- FourRegions
  - Synthetic
- Friedman{1, 2, 3}
  - Synthetic
- Labeled Faces in the Wild (*)
  - Face pair match verification
- Linnerud
  - Synthetic
- LowRankMatrix
  - Synthetic
- Madelon
  - Synthetic
- MNIST (*)
  - Small images of hand-written digigs
- MNIST Background Images
  - MNIST superimposed on natural images
- MNIST Background Random
  - MNIST superimposed on noise
- MNIST Basic
  - MNIST subset
- MNIST Rotated
  - MNIST digits rotated around
- MNIST Rotated Background Images
  - Rotated MNIST over natural images
- MNIST Noise {1,2,3,4,5,6}
  - MNIST with various amounts of noise
- Randlin
  - Synthetic
- Rectangles
  - Synthetic
- Rectangles Images
  - Synthetic
- PascalVOC {2007, 2008, 2009, 2010, 2011}
  - Labeled images from PascalVOC challenges
- PosnerKeele (*)

	Dot pattern classification task
PubFig83	Face identification
S Curve	Synthetic
SampleImages	Synthetic
SparseCodedSignal	Synthetic
SparseUncorrelated	Synthetic
SVHN (*)	Street View House Numbers
Swiss Roll	Synthetic dimensionality reduction test
Van Hateren Natural Images	High-res natural images

Conclusions

Standard practice for handling data in machine learning and related research applications involves a significant amount of manual work. The lack of formalization of data handling steps is a barrier to reproducible science in these domains. The SkData library provides both low-level data wrangling logic (downloading, decompressing, loading into Python) and high-level experiment protocols that make it easier for researchers to work on a wider variety of data sets, and easier to reproduce one another's work. Development to date has focused on classification tasks, and image labeling problems in particular, but the abstractions used in the library should apply to many other domains from natural language processing and audio information retrieval to financial forecasting. The protocol layer of the SkData library (especially using the `larray` module) supports large or infinite (virtual) data sets as naturally as small ones. The library currently provides some degree of support for about 40 data sets, and about a dozen of those feature full support of SkData's high-level, low-level, and `main.py` script APIs.

Acknowledgements

This work was funded by the Rowland Institute of Harvard, the National Science Foundation (IIS 0963668) in the United States, and the Banting Postdoctoral Fellowship program in Canada.

References

[CIFAR-10] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images.* Masters Thesis, University of Toronto, 2009.
[glumpy] https://code.google.com/p/glumpy/
[HTF09] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2009.
[Iris] http://archive.ics.uci.edu/ml/datasets/Iris
[LFW] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments.* University of Massachusetts, Amherst TR 07-49, 2007.
[Netflix] http://www.netflixprize.com/
[MLData] http://mldata.org
[Pandas] http://pandas.pydata.org
[PyTables] http://pytables.org
[SkData] http://jaberg.github.io/skdata/
[sklearn] Pedregosa et al. *Scikit-learn: Machine Learning in Python*, JMLR 12 pp. 2825--2830, 2011.
[UCI] http://archive.ics.uci.edu/ml/