
superintendent

Release 0.6.0

Jan Freyberg

Sep 11, 2022

CONTENTS:


1	Installation	3
1.1	Development installation	3
2	Introduction	5
2.1	When do you need data labelling?	5
2.2	Quickstart	5
2.3	Necessary components to label data	6
2.4	Elements of a Superintendent Widget	6
3	Active learning with superintendent	9
3.1	What is active learning?	9
3.2	Active learning in superintendent	10
3.3	Active learning strategies	12
3.4	Active learning for multi-output widgets	13
3.5	Preprocessing data before passing it to the model	13
3.6	Which model to choose	13
4	Scaling labelling and active learning	15
4.1	Distributing the labelling of images across people	16
4.2	Retrieving data from the distributed widget	17
4.3	Doing active learning during distributed labelling	18
5	API Reference	19
5.1	Acquisition functions	20
6	Examples	23
6.1	Labelling text	23
6.2	Labelling images with superintendent	24
6.3	Custom pre-processing steps as part of superintendent	25
6.4	Using docker-compose to serve a superintendent interface to many users	28
7	Indices and tables	35
	Index	37

Practical data labelling and active learning in Jupyter notebooks.

superintendent is a set of **ipywidget**-based interactive labelling tools for your data. It allows you to flexibly label all kinds of data.

It also allows you to combine your data-labelling task with a statistical or machine learning model to enable quick and practical active learning.

For example:

```
Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0, ,  
↪description='Progress:', max=1...
```

Note: Throughout the documentation, all widgets will look like they do in a jupyter notebook. However, clicking on any of the buttons will not trigger anything. This is because there is no running backend. To really get a feel for how it all works, you should install **superintendent** and try it out in a notebook.

INSTALLATION

The simplest way to install superintendent is to use pip:

```
pip install superintendent
```

You will also want to install libraries that provide the UI for labelling. One that was specifically designed to work with superintendent is [ipyannotations](#).

If you also want to run the examples, you need some dependencies not installed by default. You can get them by installing the additional dependencies example `pip install superintendent[examples]`.

1.1 Development installation

If you want to contribute to superintendent, you can get an editable installation of the library by using `flit`, the package used for developing superintendent:

```
git clone https://github.com/janfreyberg/superintendent.git
cd superintendent
pip install flit
flit install --symlink --deps all
```


INTRODUCTION

One of the most important activities in a machine learning problem can be the data labelling stage. If you start out with completely unlabelled data, you can either use unsupervised learning techniques, or you can label data, and either use semi-supervised or supervised machine learning techniques.

superintendent is a library that helps you label data in Jupyter notebooks. It manages the “annotation loop”: handling a queue of data to be labelled, processing incoming data, and even handling multiple labellers.

superintendent also lets you use a machine learning to speed up the labelling process.

The library is designed to work with the large and growing data annotation ecosystem that already exists in Jupyter notebooks. It effectively wraps other UI elements, managing the data input and annotation output.

For example, you can use [ipyannotations](#) (superintendent’s sister library) with it.

2.1 When do you need data labelling?

Many machine learning and data science projects start with data, and you want to make some predictions about the data. This is hard, because you likely don’t have labels at the beginning. For example, say you work at manufacturing plant and want to build a classifier for defective components. You may have a set of pictures of these components, but you maybe don’t have labels for which are defective.

A project like this can be sped up immensely by quickly labelling a few of pictures. And with the right infrastructure, you can label several hundred in just a few hours. The rest of the project will then become much easier.

2.2 Quickstart

Let’s start by taking a classification widget, which lets you annotate data points as belonging to one of a set of classes. First I will show you how quickly you can get going, and then I will discuss in a bit more detail what each component means.

```
from superintendent import Superintendent
from ipyannotations.images import ClassLabeller
from sklearn.datasets import load_digits

input_data = load_digits().data.reshape(-1, 8, 8)
input_widget = ClassLabeller(
    options=list(range(1, 10)) + [0], image_size=(100, 100))
data_labeller = Superintendent(
    features=input_data,
    labelling_widget=input_widget,
```

(continues on next page)

(continued from previous page)

```
)  
data_labeller
```

```
Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,   
↪description='Progress:', max=1...
```

The above interface takes the images from the dataset and displays them. It offers a set of buttons for each class the image could possibly belong to. When one of the buttons is clicked, the label is stored, and the next image is loaded automatically. The progress bar at the top will show how much of the dataset has been labelled already.

Note: Throughout the documentation, all widgets will look like they do in a jupyter notebook. However, clicking on any of the buttons will not trigger anything. This is because there is no running backend. To really get a feel for how it all works, you should install `superintendent` and try it out in a notebook.

2.3 Necessary components to label data

For labelling your data, you need:

1. The data points to annotate.
2. A way of showing those datapoints to the user.
3. A way of capturing input.
4. Some method to store the captured annotation.

`Superintendent` handles the first and last aspect: taking your data, organising it in a way that lets people label it, and then storing the annotations in the same format.

Steps 2 and 3 are so highly dependent on the data that it will likely differ for many use cases. Users can therefore define their own labelling procedures. Common use cases are implemented by the `ipyannotations` library, which is written to accompany `superintendent`.

Note: If your data is large - such as big images - it may be useful to store it as individual files on disk, use the file paths as the features supplied to `superintendent`. You can then handle loading the file when it needs to be displayed.

2.4 Elements of a Superintendent Widget

In the example above, the “UI” which provides the input (and the image display) is provided by a wholly separate library: `ipyannotations`. This is a deliberate choice to uncouple any data labelling tools from the core of `superintendent`, which is handling the queue of data points to label.

To show which parts of the widgets are provided by third party libraries, and which parts are from `superintendent`, I am going to highlight them:

```
from superintendent import Superintendent  
from ipyannotations.images import ClassLabeller  
from sklearn.datasets import load_digits  
import ipywidgets
```

(continues on next page)

(continued from previous page)

```

input_widget = ClassLabeller(options=list(range(1, 10)) + [0], image_size=(100, 100))
input_data = load_digits().data.reshape(-1, 8, 8)
data_labeller = Superintendent(
    features=input_data,
    labelling_widget=input_widget,
)

data_labeller.children[0].layout = ipywidgets.Layout(
    border='solid 2px orange',
)
data_labeller.children[1].layout = ipywidgets.Layout(
    border='solid 2px green',
)

data_labeller

```

```

Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↪description='Progress:', max=1...

```

- The orange part is provided by Superintendent. It contains a progress bar, but if you are doing active learning, it would also contain a button and performance indicator.
- The green part is provided by the specific labelling widget I have chosen: `ipyannotations.images.ClassLabeller`.

For many examples of annotation widgets, please see [ipyannotations](#). To see how to write your own, please read [custom-input](#).

ACTIVE LEARNING WITH SUPERINTENDENT

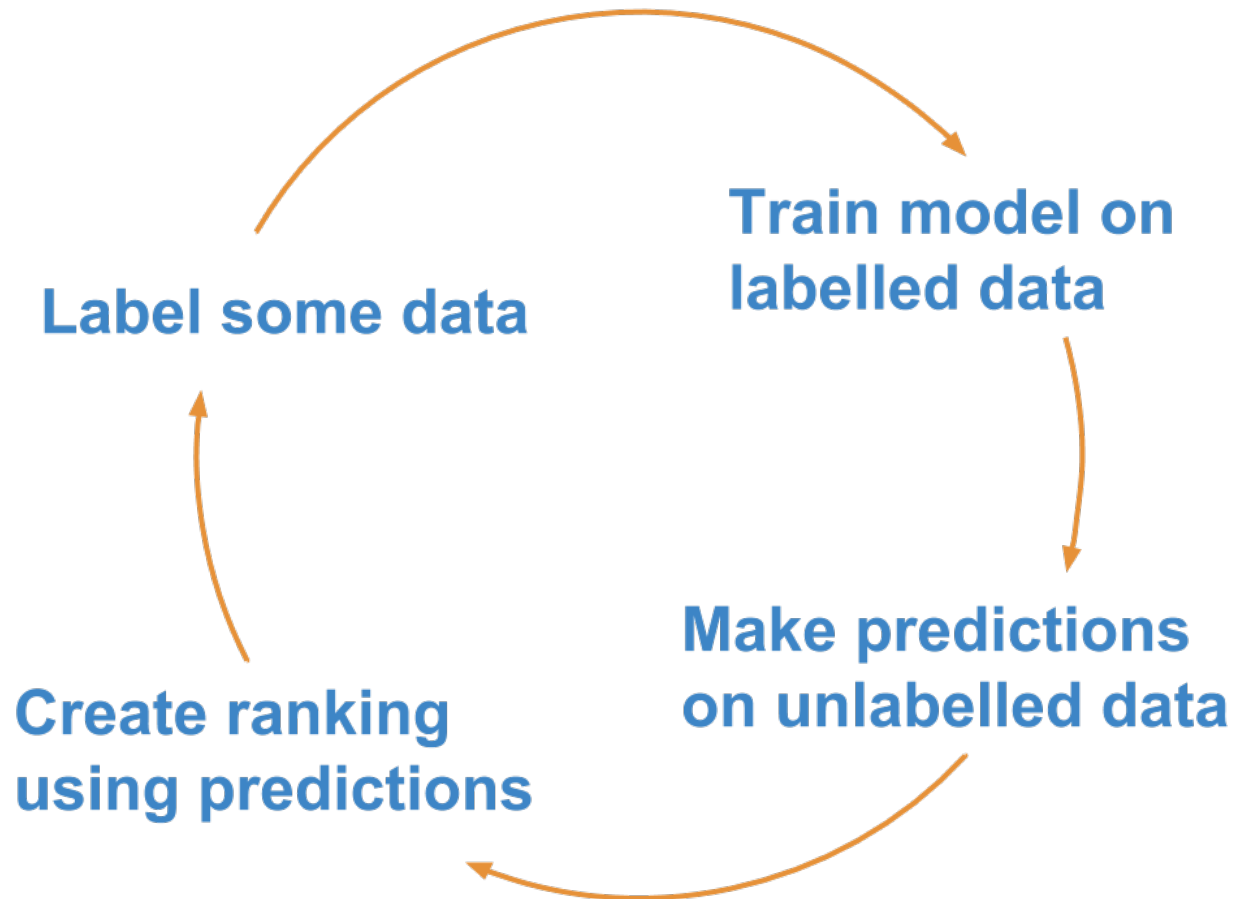
3.1 What is active learning?

Active learning is a semi-supervised machine learning approach that involves labelling data to optimally train a machine learning model.

This means a human and a machine learning algorithm interact, with the human labelling cases the machine learning algorithm is most unsure about.

A common implementation is what is known as “pool-based” active learning: You label a few cases, train your model, make predictions about the unlabelled data, and then label the points for which your model is not (yet) producing high-probability predictions.

The rough steps are:



This approach is generally more efficient than labelling data points at random, and it allows you to reach a better model performance faster.

3.2 Active learning in superintendent

The active learning process in `superintendent` is easy. `superintendent` is designed to work with any machine learning model that outputs continuous probabilities and follows the `scikit-learn` interface (i.e. the model implements a `fit` and `predict_proba` method).

You then simply pass the model, as well as a method of re-ordering the data, to a `superintendent` widget. This then gives you a button that allows you to re-train a model.

To demonstrate this, we'll create manual labels for the MNIST dataset, which we can download using `scikit-learn`'s `datasets` module. For simplicity, we will only use the first 500 images.

```
from sklearn.datasets import load_digits
digits = load_digits().data[:500, :]

print(digits.shape)
```

```
(500, 64)
```

These are 8x8 pixel images, but the 64 pixels have been “flattened” into the second array dimension, which we can undo:

```
digits = digits.reshape(-1, 8, 8)
print(digits.shape)
```

```
(5000, 8, 8)
```

To label this data, we need a “data annotation” widget. Superintendent does not ship the functionality to annotate data itself. Instead, it is designed to work with separate, modular data annotation widgets. In particular, the `ipyannotations` library, which is maintained by the same people, works well:

```
from ipyannotations.images import ClassLabeller

annotation_widget = ClassLabeller(
    options=[f"{i}" for i in range(10)],
    image_size=(256, 256),
    allow_freetext=False)

annotation_widget.display(digits[0])
annotation_widget
```

```
ClassLabeller(children=(Box(children=(Output(layout=Layout(margin='auto', min_height=
↪ '50px'))),), layout=Layout...
```

For `superintendent`, we will use this annotation widget to actually collect labels. However, first we need to think about how we are going to use machine learning to make this easiest.

Now, in most applications these days, you would likely classify images using a convolutional neural network. But for now, we can take a stab at it using a simple logistic regression model, which isn’t great, but fairly good.

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(
    solver="lbfgs",
    multi_class="multinomial",
    max_iter=5000
)
```

In addition, all scikit-learn models expect data to be “rectangular”. This means we need to preprocess the data that goes into our model. We can pass an arbitrary pre-processing function to `superintendent` - as long as it accepts the features and labels, and returns the transformed features and labels:

```
def preprocess_mnist(x, y):
    return x.reshape(-1, 64), y
```

Now that we have a dataset, an interface to label this dataset, and a supervised machine learning model we want to train on our dataset, we can pass both to `superintendent`’s `ClassLabeller`. This will create an interface for us to label data, retrain our model, *and* benefit from active learning.

Since we are using images, we can use the `from_images` class constructor that sets the correct display function for us.

```
from superintendent import Superintendent

data_labeller = Superintendent(
```

(continues on next page)

(continued from previous page)

```

    features=digits,
    model=model,
    labelling_widget=annotation_widget,
    acquisition_function='entropy',
    model_preprocess=preprocess_mnist
)

data_labeller

```

```

Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↪description='Progress:', max=1...

```

Whenever you re-train a model, if you have also specified the `acquisition_function` keyword argument, the data will be automatically re-ordered in a way prioritise data points the model is uncertain about.

Additionally, the widget will display your accuracy on the data you have already labelled. This is evaluated as the mean model score across three folds of cross-validated evaluation.

3.3 Active learning strategies

<code>:py:obj:entropy <superintendent.acquisition_functions.entropy>\ (probabilities)</code>	Sort by the entropy of the probabilities (high to low).
<code>:py:obj:margin <superintendent.acquisition_functions.margin>\ (probabilities)</code>	Sort by the margin between the top two predictions (low to high).
<code>:py:obj:certainty <superintendent.acquisition_functions.certainty>\ (probabilities)</code>	Sort by the certainty of the maximum prediction.

You can implement your own strategy: the functions should simply take in a numpy array (shape $n_samples, n_classes$) of probabilities of each class for each sample, and return a ranking of the rows of that array.

For example, if sorting by margin, an input of:

0	1	2
0.1	0.8	0.1
0.3	0.3	0.4
0.33	0.33	0.34
0.01	0.01	0.98

Should produce an output of:

```
[2, 1, 0, 3]
```

because the third entry has the lowest margin, then the second entry, then the first, and then the last.

3.4 Active learning for multi-output widgets

When you pass a model into a multi-labelling widget, `superintendent` will wrap your model in a `MultiOutputClassifier` wrapper class.

The active learning strategy will average the metric used for prioritisation (e.g. certainty, margin) across the different classes.

3.5 Preprocessing data before passing it to the model

In general, you will often want to pass different parts of your data to your display function and your model. In general, `superintendent` does not provide “pre-model” hooks. Instead, any pre-processing that is specific to your model or your display function, can be specified in the `display_func`, or in a `scikit-learn Pipeline` object.

You can find an example of this [here](#)

3.6 Which model to choose

The choice of model is ultimately driven by the same factors that should drive your model choice if you had a complete set of labelled data and wanted to build a supervised machine learning model.

SCALING LABELLING AND ACTIVE LEARNING

One of the main challenges about labelling data is that it can take a lot of time.

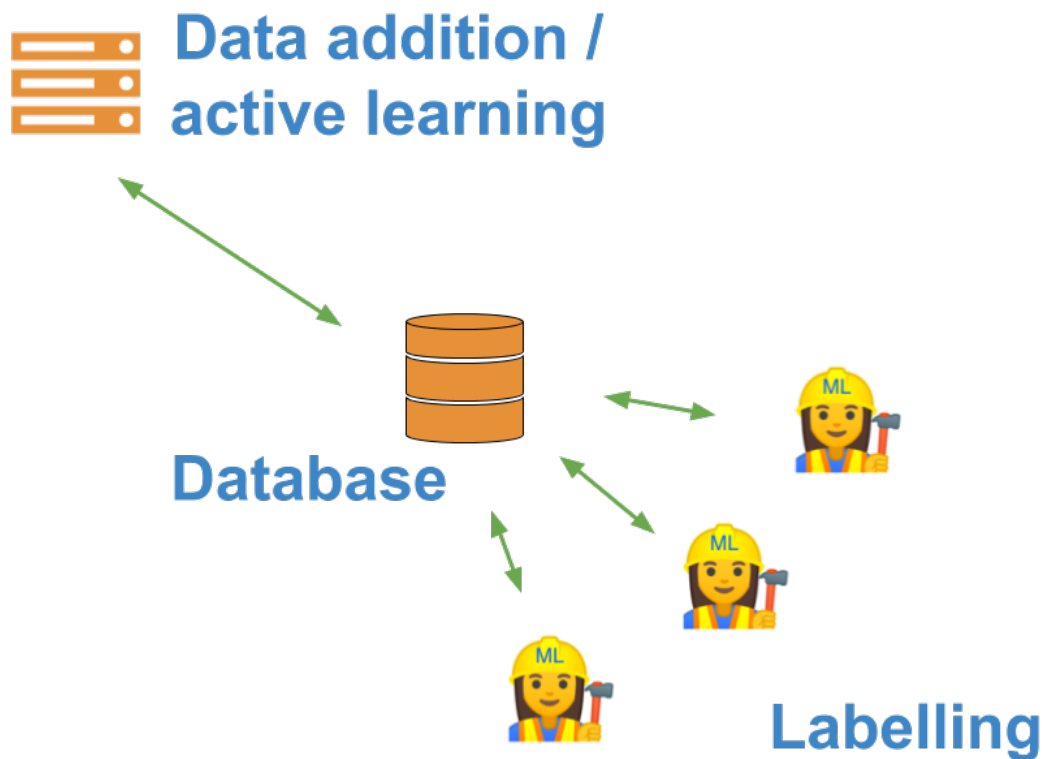
To get around this, many people want to distribute the task across multiple people - potentially even outsourcing it to a crowd platform - and this is challenging using a standard in-memory python object.

In `superintendent`, you can get around this by using the `superintendent.distributed` submodule. The labelling widgets effectively replicate the widgets in the main `superintendent` module, but do so using a database to store the “queue” of objects, as well as the results of the labelling.

The distributed submodule stores and retrieves data from a SQL database, serialising / deserialising it along the way. You simply pass your data in the same way you do with `superintendent` widgets, and can retrieve the labels in the same way. In theory, other than having to set up the database, everything else should be the same.

<p>Warning: For demonstration purposes, this example uses an SQLite file as a database. However, this is unsuitable for real distribution of labelling, as if it is on a shared file-system, it will break. In production, a database server is recommended. In the past, <code>superintendent</code> has been used successfully with PostgreSQL, but any database that works with SQLAlchemy should work.</p>

The component diagram looks a bit like this:



This allows you to ask your colleagues to label data for you. By removing the labelling process from the active learning process, it also allows you to scale the compute that does the active learning, e.g. use a server with GPUs to train complex models, while the labelling user can just use a laptop.

Ultimately, the database architecture also means that you have more persistent storage, and are more robust to crashes.

4.1 Distributing the labelling of images across people

`superintendent` uses `SQLAlchemy` to communicate with the database, and all you need to provide is a “connection url”.

First, we make sure that we are using a completely fresh database:

```
import os
if os.path.isfile("demo.db"):
    os.remove("demo.db")
```

Then, we can load the MNIST data using scikit-learn. To make sure we display them correctly, we’ll also define a pre-processing function that reshapes the data:

```
from sklearn.datasets import load_digits
import numpy as np
digits = load_digits().data.reshape(-1, 8, 8)
```

To create a “distributed” superintendent widget, all we have to do is import it from the distributed submodule, and pass a database connection string:

```
from superintendent import Superintendent
from ipyannotations.images import ClassLabeller

annotation_widget = ClassLabeller(
    options=[f"{i}" for i in range(10)],
    image_size=(256, 256),
    allow_freetext=False)

widget = Superintendent(
    connection_string="sqlite:///demo.db",
    labelling_widget=annotation_widget,
)
```

We can then add data to the database. Because every widget connects to the DB, we should only run this code once:

```
widget.add_features(digits[:1000, :])
```

We can then start labelling data by displaying the widget:

```
widget
```

```
Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↪description='Progress:', max=1...
```

You can inspect by using the `widget.queue` attribute, which encapsulates the database connection and the methods for retrieving and submitting data.

```
#from pprint import pprint

#with widget.queue.session() as session:
#    pprint(session.query(widget.queue.data).first().__dict__)
```

As you can see, `superintendent` added our entries into the database. The format of this row is not necessarily important, as you can retrieve the data needed using `superintendent` itself - this will convert it back to python objects.

For your information, data is serialised as JSON, with custom JSON encoders for numeric data such as numpy arrays or pandas objects.

4.2 Retrieving data from the distributed widget

Any superintendent connected to the database can retrieve the labels using `widget.new_labels`:

```
pprint(widget.new_labels[:30])
```

```
Pretty printing has been turned OFF
```

4.3 Doing active learning during distributed labelling

One of the great benefits of using the distributed submodule is that you can perform active learning, where the labelling of data and the training of the active learning model are split across different machines. You can achieve this by creating a widget object that you don't intend to use for labelling - only for orchestration of labelling by others. First, you run create the widget as normal:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(
    multi_class='auto',
    solver='lbfgs',
    max_iter=5000
)

widget = Superintendent(
    connection_string="sqlite:///demo.db",
    features=digits[:1000, :],
    labelling_widget=annotation_widget,
    model=model,
    model_preprocess=lambda x, y: (x.reshape(-1, 64), y)
)
```

Then, you call orchestrate on it:

```
widget.orchestrate(interval_seconds=None)
```

```
Score: 0.970
```

Note: By default, the orchestration runs forever, in regular intervals. The interval can be specified as either seconds (`interval_seconds`), or number of labels created (`interval_n_labels`). You might not want repeated orchestration - for example, you may instead want to run orchestration using cron scheduling. You can do that the way I am doing it above: by passing `None` as the `interval_seconds` keyword argument.

In either case, the orchestration is best run from a python script from the command line, rather than from a jupyter notebook.

API REFERENCE

There really only is one class you should use as your entry point to superintendent:

class `superintendent.Superintendent(**kwargs: Any)`

Data point labelling.

This is a base class for data point labelling.

Make a class that allows you to label data points.

Parameters

- **features** (`np.ndarray`, `pd.DataFrame`, `sequence`) – This should be either a numpy array, a pandas dataframe, or any other sequence object (e.g. a list). You can also add data later.
- **labels** (`np.array`, `pd.Series`, `sequence`) – The labels for your data, if you have some already.
- **queue** (`BaseLabellingQueue`) – A queue object. The interface needs to follow the abstract class `superintendent.queueing.BaseLabellingQueue`. By default, `SimpleLabellingQueue` (an in-memory queue using python's deque)
- **labelling_widget** (`Optional[widgets.Widget]`) – An input widget. This needs to follow the interface of the class `superintendent.controls.base.SubmissionWidgetMixin`
- **model** (`sklearn.base.BaseEstimator`) – An sklearn-interface compliant model (that implements `fit`, `predict`, `predict_proba` and `score`).
- **eval_method** (`callable`) – A function that accepts three arguments - model, x, and y - and returns the score of the model. If None, `sklearn.model_selection.cross_val_score` is used.
- **acquisition_function** (`callable`) – A function that re-orders data points during active learning. This can be a function that accepts a numpy array (class probabilities) or a string referring to a function from `superintendent.acquisition_functions`.
- **shuffle_prop** (`float`) – The proportion of data points that is shuffled when re-ordering during active learning. This is to avoid biasing too much towards the model predictions.
- **model_preprocess** (`callable`) – A function that accepts x and y data and returns x and y data. y can be None (in which it should return x, None) as this function is used on the un-labelled data too.
- **worker_id** (`bool` / `str`) – If True, will check for the worker's ID first - this can be helpful when working in a distributed fashion. If a string, this is used as the worker ID. If False, a UUID is generated for this widget.

add_features(*features*, *labels=None*)

Add data to the widget.

This adds the data provided to the queue of data to be labelled. You Can optionally provide labels for each data point.

Parameters

- **features** (*Any*) – The data you’d like to add to the labelling widget.
- **labels** (*Any*, *optional*) – The labels for the data you’re adding; if you have labels.

orchestrate(*interval_seconds: Optional[float] = None*, *interval_n_labels: int = 0*, *shuffle_prop: float = 0.1*, *max_runs: float = inf*)

Orchestrate the active learning process.

This method can either re-train the classifier and re-order the data once, or it can run a never-ending loop to re-train the model at regular intervals, both in time and in the size of labelled data.

Parameters

- **interval_seconds** (*int*, *optional*) – How often the retraining should occur, in seconds. If this is None, the retraining only happens once, then returns (this is suitable) if you want the retraining schedule to be maintained e.g. by a cron job). The default is 60 seconds.
- **interval_n_labels** (*int*, *optional*) – How many new data points need to have been labelled in between runs in order for the re-training to occur.
- **shuffle_prop** (*float*) – What proportion of the data should be randomly sampled on each re- training run.
- **max_runs** (*float*, *int*) – How many orchestration runs to do at most. By default infinite.

Return type

None

retrain(*button=None*)

Re-train the classifier you passed when creating this widget.

This calls the fit method of your class with the data that you’ve labelled. It will also score the classifier and display the performance.

Parameters

button (*widget.Widget*, *optional*) – Optional & ignored; this is passed when invoked by a button.

5.1 Acquisition functions

During active learning, acquisition functions rank unlabelled data points based on model predictions.

In superintendent, the functions accept a 2- or 3-dimensional array of shape `n_samples`, `n_classes`, (`n_outputs`).

The third dimension only applies in a multi-output classification setting, and in general superintendent calculates the score for each data point and then averages in this case.

`superintendent.acquisition_functions.entropy(probabilities: ndarray) → ndarray`

Sort by the entropy of the probabilities (high to low).

Parameters

- **probabilities** (*np.ndarray*) – An array of probabilities, with the shape `n_samples, n_classes`
- **shuffle_prop** (*float* (`default=0.1`)) – The proportion of data points that should be randomly shuffled. This means the sorting retains some randomness, to avoid biasing your new labels and catching any minority classes the algorithm currently classifies as a different label.

`superintendent.acquisition_functions.margin(probabilities: ndarray) → ndarray`

Sort by the margin between the top two predictions (low to high).

Parameters

- **probabilities** (*np.ndarray*) – An array of probabilities, with the shape `n_samples, n_classes`
- **shuffle_prop** (*float*) – The proportion of data points that should be randomly shuffled. This means the sorting retains some randomness, to avoid biasing your new labels and catching any minority classes the algorithm currently classifies as a different label.

`superintendent.acquisition_functions.certainty(probabilities: ndarray)`

Sort by the certainty of the maximum prediction.

Parameters

- **probabilities** (*np.ndarray*) – An array of probabilities, with the shape `n_samples, n_classes`
- **shuffle_prop** (*float*) – The proportion of data points that should be randomly shuffled. This means the sorting retains some randomness, to avoid biasing your new labels and catching any minority classes the algorithm currently classifies as a different label.

EXAMPLES

These examples showcase how superintendent can work for you.

6.1 Labelling text

Let's assume we have a text dataset that contains some labelled sentences and some unlabelled sentences. For example, we could get the headlines for a bunch of UK news websites (the code for this comes from the great github project [compare-headlines](#) by [isobelweinberg](#)):

```
import requests
from bs4 import BeautifulSoup
import datetime

headlines = []
labels = []

r = requests.get('https://www.theguardian.com/uk').text #get html
soup = BeautifulSoup(r, 'html5lib') #run html through beautiful soup
headlines += [headline.text for headline in
               soup.find_all('span', class_='js-headline-text')][:10]
labels += ['guardian'] * (len(headlines) - len(labels))

soup = BeautifulSoup(requests.get('http://www.dailymail.co.uk/home/index.html').text,
                      'html5lib')
headlines += [headline.text.replace('\n', '').replace('\xa0', '').strip()
               for headline in soup.find_all(class_='linkro-darkred')][:10]
labels += ['daily mail'] * (len(headlines) - len(labels))
```

```
from superintendent import Superintendent
from ipyannotations.text import ClassLabeller

input_widget = ClassLabeller(options=['professional', 'not professional'])
input_data = headlines
data_labeller = Superintendent(
    features=input_data,
    labelling_widget=input_widget,
)

data_labeller
```

```

Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↪description='Progress:', max=1...

```

For further options of labelling text, including tagging parts of text, check the [text widgets](#) in ipyannotations.

6.2 Labelling images with superintendent

6.2.1 Labelling images randomly

Since labelling images is a frequent use case, there is a special factory method for labelling images that are stored in numpy arrays:

```

from superintendent import Superintendent
from ipyannotations.images import ClassLabeller
from sklearn.datasets import load_digits
import ipywidgets

input_widget = ClassLabeller(options=list(range(1, 10)) + [0], image_size=(100, 100))
input_data = load_digits().data.reshape(-1, 8, 8)
data_labeller = Superintendent(
    features=input_data,
    labelling_widget=input_widget,
)

data_labeller

```

```

Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↪description='Progress:', max=1...

```

For further options of labelling images, including localising objects, check the [image widgets](#) in ipyannotations.

6.2.2 Labelling images with active learning

Often, we have a rough idea of an algorithm that might do well on a given task, even if we don't have any labels at all. For example, I know that for a simple image set like MNIST, logistic regression actually does surprisingly well.

In this case, we want to do two things:

1. We want to keep track of our algorithm's performance
2. We want to leverage our algorithm's predictions to decide what data point to label.

Both of these things can be done with superintendent. For point one, all we need to do is pass an object that conforms to the fit / predict syntax of sklearn as the `model` keyword argument.

For the second point, we can choose any function that takes in probabilities of labels (in shape `n_samples, n_classes`), sorts them, and returns the sorted integer index from most in need of labelling to least in need of labelling. Superintendent provides some functions, described in the `superintendent.acquisition_functions` submodule, that can achieve this. One of these is the `entropy` function, which calculates the entropy of predicted probabilities and prioritises high-entropy samples.

As an example:

```

from superintendent import Superintendent
from ipyannotations.images import ClassLabeller
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_digits
import ipywidgets

input_widget = ClassLabeller(options=list(range(1, 10)) + [0], image_size=(100, 100))
input_data = load_digits().data.reshape(-1, 8, 8)
data_labeller = Superintendent(
    features=input_data,
    labelling_widget=input_widget,
    model=LogisticRegression(
        solver="lbfgs", multi_class="multinomial", max_iter=5000),
    acquisition_function='entropy',
    model_preprocess=lambda x, y: (x.reshape(-1, 64), y)
)

data_labeller

```

```

Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↳description='Progress:', max=1...

```

6.3 Custom pre-processing steps as part of superintendent

Data often needs to be passed to the display widget and the model in different formats. For displaying, since the display widget is custom and supplied by the user, we recommend designing your own - and an example of this is shown here.

For modelling, you have multiple options. A common option which is very transferable is to write a [scikit-learn Pipeline](#), in which you re-format your data so that the model you build can process it.

However, to make life slightly easier, superintendent also provides an optional `model_preprocess` argument. This function needs to accept two arguments - `x` and `y` - which are the features and annotations. It needs to return two values.

To demonstrate all components of pre-processing, we will write a tool to annotate emails, and classify them.

6.3.1 Data: Emails with metadata

First, let's create some dummy data:

```

import pandas as pd

n_rows = 50

example_emails = [
    "Hi John,\nthis is just to say nice work yesterday.\nBest,\nJim",
    "Hi Mike,\nthis is just to say terrible work yesterday.\nBest,\nJim",
]

example_recipients = ["John", "Mike"]

example_timestamps = ["2018-02-01 15:00", "2018-02-01 15:03"]

```

(continues on next page)

(continued from previous page)

```
example_df = pd.DataFrame({
    'email': example_emails,
    'recipient': example_recipients,
    'timestamp': example_timestamps
})

display(example_df)
```

```

                                email recipient \
0  Hi John,\nthis is just to say nice work yester...    John
1  Hi Mike,\nthis is just to say terrible work ye...    Mike

                                timestamp
0  2018-02-01 15:00
1  2018-02-01 15:03
```

As you can see, the dataframe contains several columns, and it would be nice to display this data to the user for annotation. However, we also likely only want to pass one column (the email text) to the model for classification.

6.3.2 Display function and annotation widget

First, let's write a widget that displays the email in a way that's natural to the user. We first define a function that accepts a single data point (a row of our dataframe) and displays it:

```
from IPython.display import display, Markdown

def display_email(row):
    """
    The display function gets passed your data - in the
    case of a dataframe, it gets passed a row - and then
    has to "display" your data in whatever way you want.

    It doesn't need to return anything
    """
    display(Markdown("**To:** " + row["recipient"]))
    display(Markdown("**At:** " + row["timestamp"]))

    display(Markdown(row["email"].replace("\n", "\n\n")))
```

With a display function like this, we can then define an annotation widget with the `ipyannotations.generic.ClassLabeller`:

```
import ipyannotations.generic

annotation_widget = ipyannotations.generic.ClassLabeller(
    options=['positive', 'negative'],
    display_function=display_email,
)
annotation_widget.display(example_df.iloc[0])
annotation_widget
```

```
ClassLabeller(children=(Box(children=(Output(layout=Layout(margin='auto', min_height=
↪ '50px'))),), layout=Layout...
```

6.3.3 Model Pipeline

We only want to pass the E-Mail text to our model, and to achieve this we can write a small pre-processing function that is applied to **both** the features and labels whenever a model is fit.

We then can write a model that uses scikit-learn's feature-vectorizer and applies a logistic regression.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.base import TransformerMixin
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

def preprocessor(x, y):
    # only take Email column, leave everything else
    return x["email"], y

model = Pipeline([
    ('tfidf_vectorizer', TfidfVectorizer()),
    ('logistic_regression', LogisticRegression())
])
```

6.3.4 The widget

Now that we have assembled the necessary components, we can create our widget:

```
from superintendent import Superintendent

widget = Superintendent(
    features=example_df,
    model=model,
    model_preprocess=preprocessor,
    labelling_widget=annotation_widget,
    acquisition_function='margin',
)

widget
```

```
Superintendent(children=(HBox(children=(HBox(children=(FloatProgress(value=0.0,
↪ description='Progress:', max=1...
```

6.4 Using docker-compose to serve a superintendent interface to many users

The documentation already shows how distributed labelling using a SQL database to distribute the data can be used pretty effectively (see `:doc:../.. /distributing-labelling`).

However, setting up a database can be difficult; and although even for hundreds of labellers even a small database instance would suffice, there will be some costs associated with it in the cloud.

However, more and more organisations have `docker` running on a server; and even if your organisation does not have their own hardware or runs `docker` in the cloud, all of the popular cloud providers offer `docker` (and, in particular, `docker-compose`) as a service.

This means it becomes relatively easy for you to manage a database as a back-end, a jupyter server as a front-end, and a model-training server to support active learning.

6.4.1 docker-compose

Docker-compose allows you to specify “multi-container” (i.e. multi-machine) applications that you can then all start and stop at the same time.

You should make sure you have `docker` and `docker-compose` installed before continuing.

Here, we are going to start four machines, and the configuration file will look like this:

```
version: '3.1'

services:

  db:
    image: postgres
    restart: always
    environment: &environment
      POSTGRES_USER: superintendent
      POSTGRES_PASSWORD: superintendent
      POSTGRES_DB: labelling
      PGDATA: /data/postgres
    volumes:
      - "postgres-data:/data/postgres"
    ports:
      - 5432:5432

  adminer:
    image: adminer
    restart: always
    ports:
      - 8080:8080

  orchestrator:
    build:
      context: .
      dockerfile: tensorflow.Dockerfile
    restart: always
    depends_on:
```

(continues on next page)

(continued from previous page)

```

- "db"
environment: *environment
entrypoint: python /app/orchestrate.py
volumes:
  - ./orchestrate.py:/app/orchestrate.py

notebook:
  build:
    context: .
    dockerfile: voila.Dockerfile
  restart: always
  depends_on:
    - "db"
  environment: *environment
  volumes:
    - ./voila-interface.py:/home/anaconda/app/app.py
  ports:
    - 8866:8866

volumes:
  postgres-data:

```

Let's go through each item.

- db

The database server. This will use an official (PostgreSQL)[<https://www.postgresql.org/>] docker image. You can see that we are providing a “volume”, meaning all the data inside the database is stored in the directory `./postgres-data`.

Note: The username / password here are just as examples; and you should use some randomly generated strings for safety.

- adminer

this is purely to be able to have a graphical interface to the database.

- notebook:

This is the server that will actually server our notebook as a website. It uses an image called voila - which actually doesn't exist yet; we will create that soon.

Note that we're placing a notebook into the home folder; this means the container will know what to serve

Note also that we're giving this server the **same** environment variables as the databse server (which we captures using `&environment`)

- orchestrator

This server will run an orchestration script (which we are mounting as a volume) that will re-train and re-order the data in the database.

6.4.2 The notebook (our webapp)

To make superintendent read from the database and display the images (we'll be using MNIST again...), we need one file with the following content:

./voila-interface.ipynb

```
import os
from superintendent import Superintendent
from ipyannotations.images import ClassLabeller
from IPython import display

user = os.getenv('POSTGRES_USER', "superintendent")
pw = os.getenv('POSTGRES_PASSWORD', "superintendent")
db_name = os.getenv('POSTGRES_DB', "labelling")

db_string = f"postgresql+psycpg2://{user}:{pw}@localhost:5432/{db_name}"

input_widget = ClassLabeller(options=list(range(1, 10)) + [0], image_size=(100, 100))

widget = Superintendent(database_url=db_string, labelling_widget=input_widget)

display.display(widget)
```

6.4.3 The orchestration script (our machine learning model)

This script will look *very* similar to our notebook, but we will additionally create our machine learning model. This time, we will use a neural network, using keras.

```
import os
import time

import sqlalchemy
from sklearn.datasets import load_digits
from sklearn.model_selection import cross_val_score
from tensorflow import keras

from superintendent import Superintendent
from ipyannotations.images import ClassLabeller

def keras_model():
    model = keras.models.Sequential(
        [
            keras.layers.Conv2D(
                filters=8, kernel_size=3, activation="relu", input_shape=(8, 8, 1)
            ),
            keras.layers.MaxPool2D(2),
            keras.layers.Conv2D(filters=16, kernel_size=3, activation="relu"),
            keras.layers.GlobalMaxPooling2D(),
            keras.layers.Flatten(),
            keras.layers.Dense(10, activation="softmax"),
        ]
    )
```

(continues on next page)

(continued from previous page)

```

)
model.compile(keras.optimizers.Adam(), keras.losses.CategoricalCrossentropy())
return model

def evaluate_keras(model, x, y):
    return cross_val_score(model, x, y, scoring="accuracy", cv=3)

def wait_for_db(db_string):
    database_up = False
    connection = sqlalchemy.create_engine(db_string)
    while not database_up:
        time.sleep(2)
        try:
            print("attempting connection...")
            connection.connect()
            database_up = True
            print("connected!")
        except sqlalchemy.exc.OperationalError:
            continue

model = keras.wrappers.scikit_learn.KerasClassifier(keras_model, epochs=5)

user = os.getenv("POSTGRES_USER")
pw = os.getenv("POSTGRES_PASSWORD")
db_name = os.getenv("POSTGRES_DB")

db_string = f"postgresql+psycopg2://{user}:{pw}@db:5432/{db_name}"

# wait some time, so that the DB has time to start up
wait_for_db(db_string)

# create our superintendent class:
input_widget = ClassLabeller(options=list(range(1, 10)) + [0], image_size=(100, 100))

widget = Superintendent(
    database_url=db_string,
    labelling_widget=input_widget,
    model=model,
    eval_method=evaluate_keras,
    acquisition_function="entropy",
    shuffle_prop=0.1,
    model_preprocess=lambda x, y: (x.reshape(-1, 8, 8, 1), y),
)

# if we've never added any data to this db, load it and add it:
if len(widget.queue) == 0:
    digit_data = load_digits().data
    widget.add_features(digit_data)

```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    # run orchestration every 30 seconds
    widget.orchestrate(interval_seconds=30, interval_n_labels=10)
```

Note: In this case, we are adding the data for the images straight into the data-base. This means the `numpy` array is serialised using JSON. If your images are large, this can be too much for the database. Instead, it's recommended that you only place the *filepaths* of the image into the database.

6.4.4 Dockerfiles

Then, we need to actually build two docker images: one that will run the web application, and one that will run the orchestration:

Web application (voila) dockerfile

```
FROM continuumio/miniconda3:4.6.14-alpine

RUN /opt/conda/bin/pip install --upgrade pip

RUN mkdir /home/anaconda/app
WORKDIR /home/anaconda/app

# install some extra dependencies
RUN /opt/conda/bin/pip install voila>=0.1.2
RUN /opt/conda/bin/pip install ipyannotations
RUN /opt/conda/bin/pip install "superintendent>=0.6.0"

ENTRYPOINT ["/opt/conda/bin/voila", "--debug", "--VoilaConfiguration.extension_language_
↪mapping={'.py':'python'}"]
CMD ["app.ipynb"]
```

Model training dockerfile

```
FROM tensorflow/tensorflow:2.10.0

RUN pip install "ipyannotations>=0.5.1"
RUN pip install "superintendent>=0.6.0"

RUN mkdir /app
WORKDIR /app

ENTRYPOINT ["python"]
CMD ["app.py"]
```

6.4.5 Starting

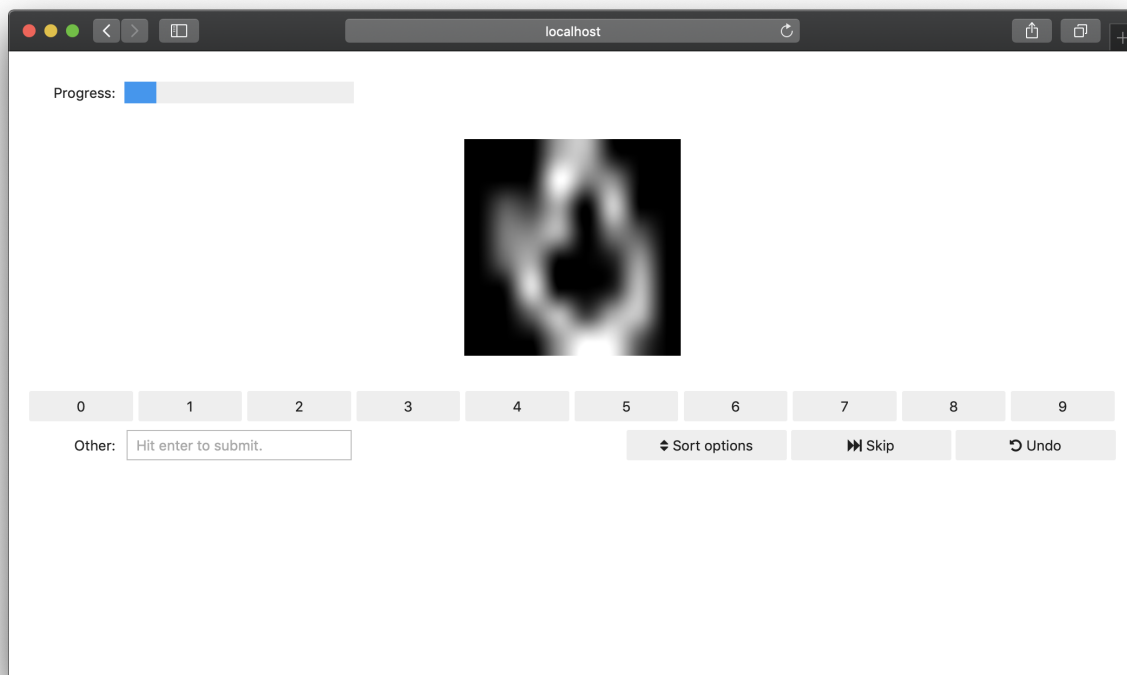
At this point, our folder structure should be:

```
.
├── docker-compose.yml
├── orchestrate.py
├── voila-interface.ipynb
├── tensorflow.Dockerfile
└── voila.Dockerfile
```

Now, we can run `docker-compose up`, which will:

1. build the docker images specified in `docker-compose.yml`
2. start the four different docker images

Now, if you visit <http://localhost:8866>, you will be able to start labelling. And, of course, if you do this on a web-server, you'll be able to point other people to that address, so they can start labelling too.



As you and your colleagues proceed with the labelling, you can inspect the content of the database at <http://localhost:8080>, the “adminer” interface (a web interface to inspect databases). Make sure to set “System” to PostgreSQL when you log in.

localhost

Language: English PostgreSQL » db » labelling » public » Select: superintendent Logout

Adminer 4.7.1 4.7.5

DB: labelling Schema: public

SQL command Import Export Create table

select superintendent

Select: superintendent

Select data Show structure Alter table New item

Select Search Sort Limit 50 Text length 100 Action Select

SELECT * FROM "superintendent" LIMIT 50 (0.006 s) Edit

Modify	Id	Input	output	inserted_at	priority
<input type="checkbox"/> edit	1008	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 12.0, 4.0, 0.0, 0.0, 0.0, 0.0, 0.0, 6...	"6"	2020-01-15 11:12:10.255882	7
<input type="checkbox"/> edit	932	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 3.0, 15.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12...	"6"	2020-01-15 11:12:10.255882	8
<input type="checkbox"/> edit	1513	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 10.0, 6.0, 0.0, 0.0, 0.0, 0.0, 0.0, 4...	"4"	2020-01-15 11:12:10.255882	9
<input type="checkbox"/> edit	540	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 2.0, 11.0, 12.0, 1.0, 0.0, 0.0, 0.0, 2.0, 1...	"3"	2020-01-15 11:12:10.255882	10
<input type="checkbox"/> edit	376	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 5.0, 13.0, 16.0, 8.0, 0.0, 0.0, 0.0, 8...	"9"	2020-01-15 11:12:10.255882	11
<input type="checkbox"/> edit	1420	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 3.0, 16.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10...	"4"	2020-01-15 11:12:10.255882	12
<input type="checkbox"/> edit	751	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 6.0, 11.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0...	"6"	2020-01-15 11:12:10.255882	13
<input type="checkbox"/> edit	1662	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 11.0, 8.0, 0.0, 0.0, 0.0, 0.0, 0.0, 6...	"4"	2020-01-15 11:12:10.255882	14
<input type="checkbox"/> edit	852	{"__type__": "__np.ndarray__", "__content__": [0.0, 5.0, 16.0, 15.0, 5.0, 0.0, 0.0, 0.0, 0.0, 2.0, 1...	"2"	2020-01-15 11:12:10.255882	16
<input type="checkbox"/> edit	1682	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 6.0, 12.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2...	"4"	2020-01-15 11:12:10.255882	17
<input type="checkbox"/> edit	1537	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 13.0, 9.0, 0.0, 0.0, 0.0, 0.0, 0.0, 4...	"4"	2020-01-15 11:12:10.255882	18
<input type="checkbox"/> edit	1525	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 9.0, 9.0, 12.0, 12.0, 0.0, 0.0, 0.0, 0.0, 1...	"5"	2020-01-15 11:12:10.255882	20
<input type="checkbox"/> edit	712	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 2.0, 14.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 11...	"6"	2020-01-15 11:12:10.255882	21
<input type="checkbox"/> edit	391	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 2.0, 13.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0...	"4"	2020-01-15 11:12:10.255882	24
<input type="checkbox"/> edit	673	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 1.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 7...	"6"	2020-01-15 11:12:10.255882	25
<input type="checkbox"/> edit	1652	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 13.0, 8.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2...	"4"	2020-01-15 11:12:10.255882	26
<input type="checkbox"/> edit	106	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 9.0, 15.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 13...	NULL	2020-01-15 11:12:10.255882	1121
<input type="checkbox"/> edit	189	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 4.0, 14.0, 6.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10...	"6"	2020-01-15 11:12:10.255882	7
<input type="checkbox"/> edit	663	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 1.0, 12.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 6...	"6"	2020-01-15 11:12:10.255882	9
<input type="checkbox"/> edit	1064	{"__type__": "__np.ndarray__", "__content__": [0.0, 0.0, 0.0, 6.0, 13.0, 3.0, 0.0, 0.0, 0.0, 0.0, 1...	"6"	2020-01-15 11:12:10.255882	13
<input type="checkbox"/> edit	1745	{"__type__": "__np.ndarray__", "__content__": [0.0, 2.0, 11.0, 16.0, 13.0, 2.0, 0.0, 0.0, 0.0, 0.0, 11...	"2"	2020-01-15 11:12:10.255882	3

Page 1 2 3 4 5 ... 36 Whole result 1,797 rows Modify Save Selected (0) Edit Clone Delete Export (1,797)

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

INDEX

A

`add_features()` (*superintendent.Superintendent method*), 19

C

`certainty()` (*in module superintendent.acquisition_functions*), 21

E

`entropy()` (*in module superintendent.acquisition_functions*), 20

M

`margin()` (*in module superintendent.acquisition_functions*), 21

O

`orchestrate()` (*superintendent.Superintendent method*), 20

R

`retrain()` (*superintendent.Superintendent method*), 20

S

`Superintendent` (*class in superintendent*), 19