SeaLion

Release 4.4.2

Anish Lakkapragada

May 08, 2022

CONTENTS:

The first machine learning framework that encourages learning ML concepts instead of memorizing class functions.

CHAPTER

ONE

INSTALLATION

SeaLion is provided on PyPI, and can be installed with Pip:

```
pip install sealion
```

On the first import, SeaLion will compile Cython files. This will take a while (around 30 seconds).

CHAPTER

TWO

REGRESSION

This module contains all of the regression modules you will need.

`class sealion.regression.LinearRegression`

While KNNs (K-Nearest-Neighbors) may be the simplest ML algorithms out there, Linear Regression is probably the one you heard first. You may have used it on a TI-84 before - all it does is it fit a line to the data. It does this through the gradient descent algorithm or a neat normal equation. We will use the normal equation as usually it works just the same and is faster, but for much larger datasets you should explore neural networks or dimensionality reduction (check the algos there.) The equation in school taught is `y = mx + b`, but we'll denote it as `y_hat = m1x1 + m2x2 ... mNxN + b`. The hat is meant to resemble the predictions, and the reason we do it from `m1...mN` is because our data can be in N dimensions, not necessarily one.

Some other things to know is that your data for `x_train` should always be 2D. 2D means that it is `[[]]`. This doesn't mean the data is necessarily 2D (this could look like `[[1, 2], [2, 3]]`) - but just means that its lists inside lists. `y_train` is your labels, or the "correct answers" so it should be in a 1D list, which is just a list. This library assumes just a bit of knowledge about this - and it isn't too difficult - so feel free to search this up.

Another thing to note here is that for our library you can enter in numpy arrays of python lists, but you will always get numpy arrays back (standard practice with other libs, too.)

A lot of the methods here are consistent and same to a lot of the other classes of the library, so reading this will make it a lot easier down the line.

The goal of this module is it for it to be a useful algorithm, but I also hope this is inspiring to your journey of machine learning. It isn't nearly as hard as it seems.

`evaluate`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns `r^2` score

`fit`(*x_train*, *y_train*)

Parameters

- **`x_train`** – 2D training data
- **`y_train`** – 1D training labels

Returns None

`predict`(*x_test*)

Parameters **`x_test`** – 2D prediction data

Returns predictions in a 1D numpy array

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions from the predict() method
- `y_test` – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

class sealion.regression.`LogisticRegression`(*accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=1000*, *show_acc=True*)

Logistic Regression is in the sweet spot of being easy to understand and useful. Despite having "regression" in the name, what it does is binary classification. Say you had a dataset of people's heights and weights, along with other attributes (we call them features) and you wanted to predict whether they were healthy (0) or unhealthy (1). You may choose to use logistic regression as this task is binary classification (classifying 2 categories.) Make sure your labels are 0 and 1.

Logistic Regression doesn't have a closed form solution, so we'll have to use the gradient descent algorithm. It may take longer but we've provided a progress bar for you to see how it's going.

You may want to look into is the sigmoid function, it's what really is at the core of distinguishing logistic and linear regression. It'll make more sense after you look at the differences in their output equations.

With that in mind, we can get started!

`__init__`(*accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=1000*, *show_acc=True*)

Parameters

- `accuracy_desired` – the accuracy at which the fit() method can stop
- `learning_rate` – how fast gradient descent should run
- `max_iters` – how many iterations of gradient descent are allowed
- `show_acc` – whether or not to show the accuracy in the fit() method

`evaluate`(*x_test*, *y_test*)

Parameters

- `x_test` – testing data (2D)
- `y_test` – testing labels (1D)

Returns what % of its predictions on x_test were correct

`fit`(*x_train*, *y_train*)

Parameters

- `x_train` – training data (2D)
- `y_train` – training labels (1D)

Returns

`predict`(*x_test*)

Parameters `x_test` – prediction data (2D)

Returns predictions in a 1D vector/list `reset_params()`

Run this after the fit() method has been run please. :return: Nothing, just redoes the weight init for all parameters.

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- **`y_pred`** – predictions from the predict() method
- **`y_test`** – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

`class sealion.regression.SoftmaxRegression`(*num_classes*, *accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=1000*, *show_acc=True*)

Once you know logistic regression, softmax regression is a breeze. Logistic regression is really a specific type of softmax regression, where the number of classes is equal to 2. Whereas logistic regression only can predict for 2 classes (0 and 1), softmax regression can predict for N number of classes. This can be 5, 3, or even a thousand! To define the number of classes in this model, insert the `num_classes` parameter in the init. ALL parameters in this class are the same as in logistic regression except for that argument.

Another note, if you use softmax regression with 2 classes - you just end up using logistic regression. In general you should use logistic regression if there are only 2 classes as it is faster and optimized as such.

If you're interested in the theory, the primary change is from the sigmoid function to the softmax function. Also look into the log loss and crossentropy loss, both of them are at the heart of softmax and logistic regression. Maybe interesting to read up on that.

`__init__`(*num_classes*, *accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=1000*, *show_acc=True*)

:param num_classes : number of classes your dataset has :param accuracy_desired: the accuracy at which the fit() method can stop :param learning_rate: how fast gradient descent should run :param max_iters: how many iterations of gradient descent are allowed :param show_acc: whether or not to show the accuracy in the fit() method

`evaluate`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns what % of its predictions on x_test were correct

`fit`(*x_train*, *y_train*)

Parameters

- **`x_train`** – training data (2D)
- **`y_train`** – training labels (1D)

Returns

`predict`(*x_test*)

Parameters **`x_test`** – prediction data (2D)

Returns predictions in a 1D vector/list `reset_params()`

Run this after the fit() method has been run please. :return: Nothing, just redoes the weight init for all parameters.

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- **`y_pred`** – predictions from the predict() method
- **`y_test`** – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

class sealion.regression.**RidgeRegression**(*alpha=0.5*)

Imagine you have a dataset with 1000 features. Most of these features will usually be irrelevant to the task your solving; only a few of them will really matter. If you don't want to use Dimensionality Reduction (see the algorithms there), you may want to consider using this. What ridge regression does is try to keep the weights as small as possible, a.k.a. regularization. This is because if a weight of a feature is not needed you want it to be 0 - you don't want it to be 0.01 because of overfitting or the particular instances of the training data. Therefore it will work well with many features as it reduces the weights, hence making the model overfit less and generalize more (do we really need those 0.1s and 0.2s in the weights?) As StatQuest said, it "desensitizes" the training data (highly recommend you watch that video - regularization can be tough.)

You'll be glad to know that this uses a closed form solution (generally much faster than iterative gradient descent.) There are other algorithms for regularization in regression like Lasso and Elastic Net, a combination of Lasso and Ridge, that are available in this module.

There's only one parameter you need to worry about, which is alpha. It is simply how much to punish the model for its weights (especially unnecessary ones). It's typically set between 0 and 1.

`__init__`(*alpha=0.5*)

Set the alpha parameter for the model.

Parameters **`alpha`** – default 0.5, ranges from 0 - 1

`evaluate`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns r^2 score

`fit`(*x_train*, *y_train*)

Parameters

- **`x_train`** – 2D training data
- **`y_train`** – 1D training labels

Returns

`predict`(*x_test*)

Parameters **`x_test`** – 2D prediction data

Returns predictions in a 1D numpy array `visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions from the predict() method
- `y_test` – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

class `sealion.regression.LassoRegression`(*alpha=0.5*, *accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=10000*, *show_acc=True*)

Regularizer for regression models just like Ridge Regression. A few notable differences, but for the most part will do the same thing. Lasso regression tries to minimize the weights just like ridge regression, but one of its big differences is its tendency to make the weights of the regression model 0. This greatly decreases overfitting by making sure unnecessary features aren't considered in the model.

Another difference is the use of gradient descent instead of a closed form solution like Ridge Regression. It shares the same alpha parameter to determine how much you want to "punish" (i.e. reduce) the weights, especially those not needed.

`__init__`(*alpha=0.5*, *accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=10000*, *show_acc=True*)

Parameters `alpha` – penalty for the weights, applied to both lasso and ridge components

Check above documentation for all other parameters.

`evaluate`(*x_test*, *y_test*)

Parameters

- `x_test` – testing data (2D)
- `y_test` – testing labels (1D)

Returns r^2 score for the predictions of x_test and y_test

`fit`(*x_train*, *y_train*)

Parameters

- `x_train` – training data (must be 2D)
- `y_train` – training labels (must be 1D)

Returns None, just the model has the weights and biases stored

`predict`(*x_test*)

Parameters `x_test` – testing data (must be 2D)

Returns predictions (1D vector/list)

`reset_params`()

Resets all weights and biases of the model. :return: none

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions from the predict() method
- `y_test` – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

`class sealion.regression.ElasticNet`(*l1_r=0.5*, *alpha=0.5*, *accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=1000*, *show_acc=True*)

Elastic Net is a combination of Ridge and Lasso Regression. It implements both penalties, and you just decide how much weight each should have. The parameter `l1_r` in the `__init__` of this class is the amount of "importance" lasso regression has (specifically the regularization term), on a scale from 0 - 1. If lasso regression gets an "importance" of 0.7, then we give the ridge regression part of this model an "importance" 0.3. Uses gradient descent, as there is no closed form solution.

`__init__`(*l1_r=0.5*, *alpha=0.5*, *accuracy_desired=0.95*, *learning_rate=0.01*, *max_iters=1000*, *show_acc=True*)

Parameters

- `l1_r` – The weight that lasso regression gets in this model. Default 0.5, but setting it higher tips the scale. Setting it to 0 or 1 makes it just ridge or lasso regression.
- `alpha` – penalty for the weights, applied to both lasso and ridge components

Check above documentation for all other parameters.

`evaluate`(*x_test*, *y_test*)

Parameters

- `x_test` – testing data (2D)
- `y_test` – testing labels (1D)

Returns r^2 score for the predictions of x_test and y_test

`fit`(*x_train*, *y_train*)

Parameters

- `x_train` – training data (must be 2D)
- `y_train` – training labels (must be 1D)

Returns None, just the model has the weights and biases stored

`predict`(*x_test*)

Parameters `x_test` – testing data (must be 2D)

Returns predictions (1D vector/list)

`reset_params`()

Resets all weights and biases of the model. :return: none

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions from the predict() method
- `y_test` – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

class sealion.regression.**ExponentialRegression**

Say you've got some curved data. How can a line possibly fit that data? Glad you ask - that's why this class exists. Exponential Regression does something very simple. All it does is just take the log transformation of exponential data, so it becomes a line and model that. Because now we can find `y_transformed_log = mx + b`, we can turn that `y_transformed_log` back into `y` by raising it to power `e`.

evaluate(*x_test*, *y_test*)

Parameters

- **x_test** – testing data (2D)
- **y_test** – testing labels (1D)

Returns r^2 score

fit(*x_train*, *y_train*)

Parameters

- **x_train** – training data (2D)
- **y_train** – training labels (1D)

Returns

predict(*x_test*)

Parameters **x_test** – prediction data (2D)

Returns predictions in a 1D vector/list

visualize_evaluation(*y_pred*, *y_test*)

Parameters

- **y_pred** – predictions from the predict() method
- **y_test** – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

class sealion.regression.**PolynomialRegression**

Polynomial Regression is like an extended version of Linear Regression (sort of like Exponential Regression.) All it does is turn the equation from y = m1x1 + m2x2 … mNxN + bias to y = m1x1^1 + m2x2^2 … mNxN^N + bias. It just adds those power combinations to make the regression module model the data better. Neural networks can also model those functions similarily.

Please normalize your data with this module with the (X - mu) / sigma or just by dividing by the maximum value. This will help with faster convergence.

Not as famous as some other regression algorithms, so may you need a bit of experimentation.

evaluate(*x_test*, *y_test*)

Parameters

- **x_test** – data to be evaluated on
- **y_test** – labels

Returns r^2 score

`fit`(*x_train*, *y_train*)

Parameters

- **`x_train`** – training data (2D)
- **`y_train`** – training labels (1D)

Returns

`predict`(*x_test*)

Parameters **`x_test`** – points to be predicted on. Has to be stored in 2D array, even if just one data point. Ex. `[[1, 1]]` or `[[1, 1], [2, 2], [3, 3]]`

Returns flattened numpy array of your data going through the forward pass (that's rounded as it's either 0 or 1)

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- **`y_pred`** – predictions from the predict() method
- **`y_test`** – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

CHAPTER

THREE

NEURAL NETWORKS

3.1 Models

class sealion.neural_networks.models.**NeuralNetwork**(*layers=None*)

This class is very rich and packed with methods, so I figured it would be best to have this tutorial guide you on a tutorial on the "hello world" of machine learning.

There are two ways to initialize this model with its layers, through the `__init__()` or through the `add()` methods.

The first way:

```
>>> from sealion import neural_networks as nn
>>> layers = [nn.layers.Flatten(), nn.layers.Dense(784, 64, activation=nn.layers.
↪ReLU()),
...     nn.layers.Dense(64, 32, activation=nn.layers.ReLU())]
>>> nn.layers.Dense(32, 10, activation=nn.layers.Softmax())]
>>> model = nn.models.NeuralNetwork(layers)
```

Or you can go through the second way:

```
>>> from sealion import neural_networks as nn
>>> model = nn.models.NeuralNetwork()
>>> model.add(nn.layers.Flatten())
>>> model.add(nn.layers.Dense(784, 64, activation=nn.layers.ReLU()))
>>> model.add(nn.layers.Dense(64, 32, activation=nn.layers.ReLU()))
>>> model.add(nn.layers.Dense(32, 10, activation=nn.layers.ReLU()))
```

Either way works just fine.

Next, you will want to perhaps see how complex the model is (having too many parameters means a model can easily overfit), so for that you can just do:

```
>>> num_params = model.num_parameters() # returns an integer
>>> assert num_params == 52650
```

Looks like our model will be pretty complex. Next up is finalizing and training.

```
>>> model.finalize(loss=nn.loss.CrossEntropy(), optimizer=nn.optimizers.Adam())
```

Here we use cross-entropy loss for this classification problem and the Adam optimizer.

Onto training. Assuming our data is `60k * 28 * 28` (sounds a lot like MNIST) in the variable X_train and we have y_train is a one-hot encoded matrix size `60k * 10` (10 classes) we can do :

```
>>> model.train(X_train, y_train, epochs=20) # train for 20 epochs
```

which will work fine. Here we have a batch_size of 32 (default), and the way we make this run fast is by making batch_size datasets, and training them in parallel via multithreading.

If you want to change batch_size to 18 you could do:

```
>>> model.train(X_train, y_train, epochs=20, batch_size=18) # train for 20 epochs,␣
↪batch_size 18
```

If you want the gradients to be calculated over the entire dataset (this will be much longer) you can do:

```
>>> model.full_batch_train(X_train, y_train, epochs=20)
```

Lastly there's also something known as mini-batch gradient descent, which just does gradient descent but randomly chooses a percent of the dataset for gradients to be calculated upon. This cannot be parallelized, but still runs fast:

```
>>> model.mini_batch_train(X_train, y_train, N=0.1) # here we take 10% of X_train␣
↪or 6000 data-points
...                                                 # randomly selected for␣
↪calculating gradients at a time
```

All of these methods have a show_loop parameter on whether you want to see the tqdm loop, which is set true by default.

Now that we have trained our model, time to test and use it. To evaluate it, given X_test (shape `10k * 28 * 28`) and y_test (shape `10k * 10`), we can feed this into our `evaluate()` function:

```
>>> model.evaluate(X_test, y_test)
```

This gives us the loss, which can be not so interpretable - so we have some other options.

If you are doing a classification problem as we are here you may do instead:

```
>>> model.categorical_evaluate(X_test, y_test)
```

which just gives what percent of X_test was classified correctly.

For regression just do:

```
>>> model.regression_evaluate(X_test, y_test)
```

which gives the `r^2` value of the predictions on X_test. If you are doing this make sure that y_test is not one-hot encoded.

To predict on data we can just do:

```
>>> predictions = model.predict(X_test)
```

and if we want this in reverted one-hot-encoded form all we need to do is this:

```
>>> from utils import revert_softmax
>>> predictions = revert_softmax(predictions)
```

Storing around 53,000 parameters in matrices isn't much, but for good practice let's store it. Using the give_parameters() method we can save our weights in a list:

```
>>> parameters = model.give_parameters()
```

Then we may want to pickle this using the given method, into a file called "MNIST_pickled_params":

```
>>> file_name = "MNIST_pickled_params"
>>> model.pickle_params(FILE_NAME=file_name)
```

Now this is the beauty - let's say a few weeks from now we come back and realize we probably should train for 20 epochs. BUT - we don't want to have to restart training (imagine this was a really big network.)

So we can just load the weights using the pickle module, build the same model architecture, insert the params, and hit train()!!

```
>>> import pickle
>>> with open('MNIST_pickled_params.pickle', 'rb') as f :
>>>  params = pickle.load(f)
```

Now we can create the model structure (must be the EXACT same):

```
>>> model = nn.models.NeuralNetwork()
>>> model.add(nn.layers.Flatten())
>>> model.add(nn.layers.Dense(784, 64, activation = nn.layers.ReLU()))
>>> model.add(nn.layers.Dense(64, 32, activation = nn.layers.ReLU()))
>>> model.add(nn.layers.Dense(32, 10, activation = nn.layers.ReLU()))
```

Obviously we want to finalize our model for good practice:

```
>>> model.finalize(loss = nn.loss.CrossEntropy(), optimizer = nn.optimizers.Adam())
```

and we can enter in the weights & biases using the enter_parameters():

```
>>> model.enter_parameters(params) #must be params given from the give_parameters()␣
↪method
```

and train!

```
>>> model.train(X_train, y_train, epochs = 100) #let's try 100 epochs this time
```

The reason this is such a big deal is because we don't have to start training from scratch all over again. The model will not have to start from 0% accuracy, but may start with 80% given the params belonging to our partially-trained model we loaded from the pickle file. This is of course not a big deal for something like MNIST but will be for bigger model architectures, or datasets.

Of course there's a lot more things we could've changed, but I think that's pretty good for now!

`__init__`(*layers=None*)

`finalize`(*loss, optimizer*)

Both of these have to be the classes for the loss and optimizations

3.2 Layers

class `sealion.neural_networks.layers.`**`Layer`**

Base Layer class. All layers inherit from this.

`backward`(*grad*)

This method takes in a gradient (e.x `1/Z2`) and returns its grad (e.x. `dL/dA1`)

`forward`(*inputs*)

This method saves the inputs, and returns its outputs

class `sealion.neural_networks.layers.`**`Flatten`**

This would be better explained as Image Data Flattener. Let's say your dealing with MNIST (check out the examples on github) - you will have data that is 60000 by 28 by 28. Neural networks can't work with data like that - it has to be a matrix. What you could do is take that `60000 * 28 * 28` data and apply this layer to make the data `60000 * 784`. 784 because `28 * 28 = 784` and we just "squished" this matrix to one layer. If you have data that has colors, e.g. `60000 * 28 * 28 * 3`, applying this layer would turn it into a `60000 * 2352` matrix, `2352 = 28 * 28 * 3`.

An example for how this would work with something like MNIST (the grayscale `60k * 28 * 28` dataset) is shown below.

```
>>> from sealion import neural_networks as nn
>>> from sealion.neural_networks.models import NeuralNetwork
>>> model = NeuralNetwork()
>>> model.add(nn.layers.Flatten()) # always, always add that on the first layer
>>> model.add(nn.layers.Dense(784, whatever_output_size, more_args)) # now put that␣
↪data to ``28*28 = 784``.
>>> # Do more cool stuff...
```

class `sealion.neural_networks.layers.`**`Dropout`**(*dropout_rate*)

Dropout is one of the most well-known regularization techniques there is. Imagine you were working on a coding project with about 200 people. If we just relied on one person to know how to compile, another person to know how to debug, then what happens when those special people leave for a day, or worse leave forever?

Now I know that seems to have no connection to dropout, but here's how it does. In dropout this sort of scenario is prevented. "Dropping out", or setting to 0, some of the outputs of some of the neurons means that the model will have to learn that it can't just depend on one neuron to learn the most important features. This means has each neuron learn some features, some other features, etc. and can't just depend on one node. The model will become more robust and generalize better with Dropout as every neuron now has a better set of weights. Normally due to dropout, it will be applied in training, but then "reverse-applied" in testing. Dropout will make the training accuracy go down a bit, but remember in the end it's testing on real-world data that matters.

There's a parameter dropout_rate on what percent (from 0 to 1 here) you want each neuron in the layer you are at to become 0. This is essentially the chance of dropping out any given neuron, or usually what percent of neurons will be dropped out. Typical values range from 0.1 to 0.5. Example below.

Let's say you've gotten your models up so far:

```
>>> model.add(nn.layers.Flatten())
>>> model.add(nn.layers.Dense(128, 64, activation = nn.layers.ReLU()))
```

And now you want to add dropout. Well just add that layer like such:

```
>>> dropout_probability = 0.2
>>> model.add(nn.layers.Dropout(dropout_probability))
```

This will just mean that about 20% of the neurons coming from this first input layer will be dropped out. A higher dropout rate may not always lead to better generalization, but usually will decrease training accuracy.

In dropout remember 2 things, not just one matter. The probability, and the layers its applied at. **Experimentation is key**.

`class sealion.neural_networks.layers.`**`Dense`**(*input_size: int*, *output_size: int*, *activation=None*, *weight_init='xavier'*)

This is the class that you will depend on the most. This class is for creating the fully-connected layers that make up Deep Neural Networks - where each neuron in a previous layer is connected to each layer in the next. Feel free to watch a couple of youtube tutorials from 3Blue1Brown (if you like calculus :) or others to get a better understanding of these parameters, or just look at the examples on my github.

The main method of course is the init. You will have to define the input_size, the output_size, and the activation and weight initialization are optional parameters.

In a neural network the number of nodes starting in a layer are known as the input_size here (fan_in in papers), and the number of nodes in the output of a layer is the output_size here (fan_out in papers.) The activation parameter is for the activation/non-linearity function you want your layers to go through. If you don't understand what I just meant, sorry - but another way to think about it is that its a way to change the outputs of your network a bit for it to fit you dataset a little better (looking at a graph will help.) The default is no activation (some people call that Linear), so you'll need to add that yourself. I'll get to weight init in a bit. Examples below.

To add a layer, here's how it's done:

```
>>> from sealion import neural_networks as nn
>>> model = nn.models.NeuralNetwork()
>>> model.add(nn.layers.Dense(128, 64)) # input_size = 128, output_size = 64
```

This sets up a neural network with 128 incoming nodes (that means we have 128 numeric features), and then 64 output nodes.

Let's say we wanted an activation function, like a relu or sigmoid (there are a bunch to choose from this API.) You could add that layer here like such:

```
>>> model = nn.models.NeuralNetwork() # clear all existing layers
>>> model.add(nn.layers.Dense(128, 64, activation=nn.layers.Sigmoid())) # all␣
↪outputs go through the sigmoid function
```

Onto weight initalization! The jargon starts .... now. What weight init does is make the weights come from a special distribution (typically Gaussian) with a given standard deviation based on the input and output size. The reason this is done is because you don't want the weights to be initalized too big or else the gradients in backpropagation may cause the model to go way off and become NaNs or infinity (this is known as exploding gradients.) For neural networks that are small that solve datasets like XOR or even breast cancer this isn't a problem, but for deep neural networks for problems like MNIST this is a huge concern. The weight_init you will want to use is also dependent on what activation you are using. Most activation functions will do well with Xavier Glorot, so that is set as the default. You can choose to also use He for ReLU, LeakyReLU, ELU, or other variants of the ReLU activation. For SELU, you may choose to use LeCun weight initalization.

The possible choices are:

```
>>> "xavier" # no activation, logistic/sigmoid, softmax, and tanh
>>> "he"     # relu + similar variants
>>> "lecun"  # selu
>>> "none"   # if you want to do this
```

To set this you can just do:

```
>>> model = nn.models.NeuralNetwork() # clear all existing layers
>>> model.add(nn.layers.Dense(128, 64, activation=nn.layers.ReLU(), weight_init="he
↪"))
```

Sorry for so much documentation, but this really is the class you will call the most.

class sealion.neural_networks.layers.**BatchNormalization**(*input_size: int*, *momentum=0.9*, *epsilon=0.001*, *lr=0.01*)

Batch Normalization is a frequently used technique in today's models (especially in CNNs).

Often times you are told to normalize your data (make it bell-curved shaped) before you feed it to your model - that's because normalization makes it easier for the model to learn because the loss curve is smoother (thus getting to the minima is easier and faster.) So, why don't we apply normalization to the inputs of all of the other layers (aside from the input layer)? That's what batch normalization does.

I got this explanation from CodeEmporium

In order for a given hidden layer to normalize its inputs it needs to know the mean and variance (just standard deviation squared) of the inputs it usually gets. Note that this does change (because the parameters of prior layers change), so the mean and variance is not necessarily the same throughout training. To do this we need to use a moving average to approximate the mean and variance of the given inputs fed to a B.N. layer we would like to normalize.

While the B.N. first uses this mean and variance to normalize its inputs to a standard normal distribution, where the mean is 0 and the standard deviation is 1, it is not guaranteed that this distribution is optimal as inputs for the succeeding layer. So the B.N. layer also learns a mean and variance, called beta and gamma respectively, to adjust the standard normal distribution (it got by normalizing its inputs) before feeding it to the next layer.

Onto the parameters:

input_size: number of features the B.N. layer needs to normalize (just output_size of the Dense layer above)

momentum: how slowly to change the mean and variance that the B.N. layer uses for normalization. A higher momentum means that the mean and the variance the B.N. layer uses will change very slowly, whereas a lower momentum means that the mean and the variance the B.N. layer uses will change very quickly in response to changes in the B.N.'s inputs (and their mean and variance). Default 0.9.

epsilon: tiny value needed in normalization if the variance ever becomes 0 to protect against 0 division errors. Default 0.001.

lr: learning rate for the B.N. layer's learning of the normalized mean and variance. Default 0.01.

To add Batch Normalization to your model simply do:

```
>>> import sealion as sl
>>> model = sl.neural_networks.models.NeuralNetwork()
>>> model.add(...) # add whatever Dense Layers
>>> model.add(sl.neural_networks.layers.BatchNormalization(input_size = 5, momentum␣
↪= 0.9, lr = 0.01))
```

Note you may want to experiment on whether to place a Batch Normalization layer after an activation or before, etc. I don't know too much about this, but handsonml said this so I might as well too.

3.3 Activations

class `sealion.neural_networks.layers.`**`Tanh`**

Uses the tanh activation, which squishes values from -1 to 1.

class `sealion.neural_networks.layers.`**`Sigmoid`**

Uses the sigmoid activation, which squishes values from 0 to 1.

class `sealion.neural_networks.layers.`**`Swish`**

Uses the swish activation, which is sort of like ReLU and sigmoid combined. It's really just `f(x) = x * sigmoid(x)`. Not as used as other activation functions, but give it a try!

class `sealion.neural_networks.layers.`**`ReLU`**

The most well known activation function and the pseudo-default almost. All it does is turn negative values to 0 and keep the rest. Basically `f(x) = max(x, 0)`

class `sealion.neural_networks.layers.`**`LeakyReLU`**(*leak=0.01*)

Variant of the ReLU activation, just allows negatives values to be something more like 0.01 instead of 0, which means the neuron is "dead" as `0 * anything` is 0.

The leak is the slope of how low negative values you are willing to tolerate. Usually set from 0.001 to 0.2, but the default of 0.01 works usually quite well.

class `sealion.neural_networks.layers.`**`ELU`**(*alpha=1*)

Solves the similar dying activation problem. The default of 1 for alpha works quite well in practice, so you won't need to change it much.

class `sealion.neural_networks.layers.`**`SELU`**

Special type of activation function, that will "self-normalize" (have a mean of 0, and a standard deviation of 1) its outputs. This self-normalization typically leads to faster convergence.

If you are using this activation function make sure weight_init is set = "lecun" in whichever layer applied. It also need its inputs (in the beginning and all throughout) to be standardized (mu = 0, sigma = 1) for it to work, so make sure to get that taken care of. You can do that by standardizing your inputs and then always using SELU activation function (remember the "lecun" part!).

class `sealion.neural_networks.layers.`**`PReLU`**(*lr=0.0001*, *momentum=0.9*)

The PReLU activation function is essentially the same thing as the LeakyReLU activation, except that the "leak" parameter is learnt during training. To learn this parameter, gradient descent is used - so you have a learning rate and momentum parameter. Both are on a scale of 0 to 1. We initialize this "leak" parameter to 0.25 at the first iteration.

class `sealion.neural_networks.layers.`**`Softmax`**

Softmax activation function, used for multi-class (2+) classification problems. Make sure to use crossentropy with softmax, and it is only meant for the last layer!

3.4 Losses

class sealion.neural_networks.loss.**Loss**

Base loss class.

class sealion.neural_networks.loss.**MSE**

MSE stands for mean-squared error, and its the loss you'll want to use for regression. To set it in the model.finalize() method just do:

```
>>> from sealion import neural_networks as nn
>>> model = nn.models.NeuralNetwork(layers_list)
>>> model.finalize(loss=nn.loss.MSE(), optimizer=...)
```

and you're all set!

class sealion.neural_networks.loss.**CrossEntropy**

This loss function is for classification problems. I know there's a binary log loss and then a multi-category cross entropy loss function for classification, but they're essentially the same thing so I thought using one class would make it easier. Remember to use one-hot encoded data for this to work (check out utils).

If you are using this loss function, make sure your last layer is Softmax and vice versa. Otherwise, annoying error messages will occur.

To set this in the `model.finalize()` method do:

```
>>> from sealion import neural_networks as nn
>>> model = nn.models.NeuralNetwork()
>>> # ... add the layers ...
>>> model.add(nn.layers.Softmax()) # last layer has to be softmax
>>> model.finalize(loss=nn.loss.CrossEntropy(), optimizer=...)
```

and that's all there is to it.

3.5 Optimizers

class sealion.neural_networks.optimizers.**Optimizer**

Base optimizer class. All optimizers extend from this.

class sealion.neural_networks.optimizers.**GD**(*lr=0.001*, *clip_threshold=inf*)

The simplest optimizer - you will quickly outgrow it. All you need to understand here is that the learning rate is just how fast you want the model to learn (default 0.001) set typically from 1e-6 to 0.1. A higher learning rate may mean the model will struggle to learn, but a slower learning rate may mean the model will but will also have to take more time. It is probably the most important hyperparameter in all of today's machine learning.

The clip_threshold is simply a value that states if the gradients is higher than this value, set it to this. This is to prevent too big gradients, which makes training harder. The default for all these optimizers is infinity, which just means no clipping - but feel free to change that. You'll have to experiment quite a bit to find a good value. This method is known as gradient clipping.

In the `model.finalize()` method:

```
>>> model.finalize(loss=..., optimizer=nn.optimizers.GD(lr=0.5, clip_threshold=5))
↪# here the learning
>>> # learning rate is 0.5 and the threshold for clipping is 5.
```

class sealion.neural_networks.optimizers.**Momentum**(*lr=0.001*, *momentum=0.9*, *nesterov=False*, *clip_threshold=inf*)

If you are unfamiliar with gradient descent, please read the docs on that in GD class for this to hopefully make more sense.

Momentum optimization is the exact same thing except with a little changes. All it does is accumulate the past gradients, and go in that direction. This means that as it makes updates it gains momentum and the gradient updates become bigger and bigger (hopefully in the right direction.) Of course this will be uncontrolled on its own, so a momentum parameter (default 0.9) exists so the previous gradients sum don't become too big. There is also a nesterov parameter (default false, but set that true!) which sees how the loss landscape will be in the future, and makes its decisions based off of that.

An example:

```
>>> momentum = nn.optimizers.Momentum(lr=0.02, momentum=0.3, nesterov=True) #␣
↪learning rate is 0.2, momentum at 0.3, and we have nesterov!
>>> model.finalize(loss=..., optimizer=momentum)
```

There's also a clip_threshold argument which you implements gradient clipping, an explanation you can find in the GD() class's documentation.

Usually though this works really good with SGD…

class sealion.neural_networks.optimizers.**SGD**(*lr=0.001*, *momentum=0.0*, *nesterov=False*, *clip_threshold=inf*)

SGD stands for Stochastic gradient descent, which means that it calculates its gradients on random (stochastic) picked samples and their predictions. The reason it does this is because calculating the gradients on the whole dataset can take a really long time. However ML is a tradeoff and the one here is that calculating gradients on just a few samples means that if those samples are all outliers it can respond poorly, so SGD will train faster but not get as high an accuracy as Gradient Descent on its own.

Fortunately though, there are work arounds. Implementing momentum and nesterov with SGD means you get faster training and also the convergence is great as now the model can go in the right direction and generalize instead of overreact to hyperspecific training outliers. By default nesterov is set to False and there is no momentum (set to 0.0), so please change that as you please.

To use this optimizer, just do:

```
>>> model.finalize(loss=..., optimizer=nn.optimizers.SGD(lr=0.2, momentum=0.5,␣
↪nesterov=True, clip_threshold=50))
```

Here we implemented SGD optimization with a learning rate of 0.2, a momentum of 0.5 with nesterov's accelerated gradient, and also gradient clipping at 50.

class sealion.neural_networks.optimizers.**AdaGrad**(*lr=0.001*, *nesterov=False*, *clip_threshold=inf*, *e=1e-10*)

Slightly more advanced optimizer, an understanding of momentum will be invaluable here. AdaGrad and a whole plethora of optimizers use adaptive gradients, or an adaptive learning rate. This just means that it will assess the landscape of the cost function, and if it is steep it will slow it down, and if it is flatter it will accelerate. This is a huge deal for avoiding gradient descent from just going into a steep slope that leads to a local minima and being stuck, or gradient descent being stuck on a saddle point.

The only new parameter is e, or this incredibly small value that is meant to prevent division by zero. It's set to 1e-10 by default, and you probably won't ever need to think about it.

As an example:

```
>>> model.finalize(loss=..., optimizer=nn.optimizers.AdaGrad(lr=0.5, nesterov=True,␣
↪clip_threshold=5))
```

AdaGrad is not used in practice much as often times it stops before reaching the global minima due to the gradients being too small to make a difference, but we have it anyways for your enjoyment. Better optimizers await!

`class sealion.neural_networks.optimizers.`**`RMSProp`**(*lr=0.001*, *beta=0.9*, *nesterov=False*, *clip_threshold=inf*, *e=1e-10*)

RMSprop is a widely known and used algorithm for deep neural network. All it does is solve the problem AdaGrad has of stopping too early by not scaling down the gradients so much. It does through a beta parameter, which is set to 0.9 (does quite well in practice.) A higher beta means that past gradients are more important, and a lower one means current gradients are to be valued more.

An example:

```
>>> model.finalize(loss=..., optimizer=nn.optimizers.RMSprop(nesterov=True, beta=0.
↪9))
```

Of course there is the nesterov, clipping threshold, and `e` parameter all for you to tune.

`class sealion.neural_networks.optimizers.`**`Adam`**(*lr=0.001*, *beta1=0.9*, *beta2=0.999*, *nesterov=False*, *clip_threshold=inf*, *e=1e-10*)

Most popularly used optimizer, typically considered the default just like ReLU for activation functions. Combines the ideas of RMSprop and momentum together, meaning that it will adapt to different landscapes but move in a faster direction. The beta1 parameter (default 0.9) controls the momentum optimization, and the beta2 parameter (default 0.999) controls the adaptive learning rate parameter here. Once again higher betas mean the past gradients are more important.

Often times you won't know what works best - so hyperparameter tune.

As an example:

```
>>> model.finalize(loss=..., optimizer=nn.optimizers.Adam(lr=0.1, beta1=0.5,␣
↪beta2=0.5))
```

Adaptive Gradients may not always work as good as SGD or Nesterov + Momentum optimization. For MNIST, I have tried both and there's barely a difference. If you are using Adam optimization and it isn't working maybe try using nesterov with momentum instead.

`class sealion.neural_networks.optimizers.`**`Nadam`**(*lr=0.001*, *beta1=0.9*, *beta2=0.999*, *clip_threshold=inf*, *e=1e-10*)

Nadam optimization is the same thing as Adam, except there's nesterov updating. Basically this class is the same as the Adam class, except there is no nesterov parameter (default true.)

As an example:

```
>>> model.finalize(loss=..., optimizer=nn.optimizers.Nadam(lr=0.1, beta1=0.5,␣
↪beta2=0.5))
```

`class sealion.neural_networks.optimizers.`**`AdaBelief`**(*lr=0.001*, *beta1=0.9*, *beta2=0.999*, *nesterov=False*, *clip_threshold=inf*, *e=1e-10*)

AdaBelief is a recently popular optimizer that I thought to implement after checking in with the original paper. The way it works is by establishing the general "belief" (prediction) in where the gradient will go. If the belief and the actual given gradient are very similar, there will be a large change in the weights (larger gradients), whereas if they are very different, there will be a small change in the weights. This has worked well in practice, and its results are comparable to Adam.

For those interested in the original paper, go here : https://arxiv.org/pdf/2010.07468.pdf

CHAPTER FOUR

DECISION TREES

`class sealion.decision_trees.DecisionTree`(*max_branches=inf*, *min_samples=1*)

Decision Trees are powerful algorithms that create a tree by looking at the data and finding what questions are best to ask? For example if you are trying to predict whether somebody has cancer it may ask, "Do they have cells with a `size >= 5`?" or "Is there any history of smoking or drinking in the family?" These algorithms can easily fit most datasets, and are very well capable of overfitting (luckily we offer some parameters to help with that.)

If you are going to be using categorical features with this Decision Tree, make sure to one hot encode it. You can do that with the `one_hot()` function in the utils module.

This Decision Tree class doesn't support regression, or any labels that have continuous values. It is only for classification tasks. The reason this is such is because most curves and lines can be formed or modeled better with regression algorithms (Linear, Polynomial, Ridge, etc. are all available in the regression module) or neural networks. Decision Trees will typically overfit on such tasks.

`__init__`(*max_branches=inf*, *min_samples=1*)

Parameters

- `max_branches` – maximum number of branches for the decision tree
- `min_samples` – minimum number of samples in a branch for the branch to split

`average_branches()`

Returns an estimate of how many branches the decision tree has right now with its current parameters.

`evaluate`(*x_test*, *y_test*)

Parameters

- `x_test` – 2D testing data.
- `y_test` – 1D testing labels.

Returns accuracy score.

`fit`(*x_train*, *y_train*)

Parameters

- `x_train` – training data - 2D (make sure to one_hot categorical features)
- `y_train` – training labels

Returns

`give_tree`(*tree*)

Parameters `tree` – a tree made by you or given by give_best_tree in the RandomForest module

Returns None, just that the tree you give is now the tree used by the decision tree.

`predict`(*x_test*)

Parameters `x_test` – 2D prediction data.

Returns predictions in 1D vector/list.

`return_tree`()

Returns the tree inside (if you want to look at it)

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions given by model
- `y_test` – actual labels

Returns an image of the predictions and the labels.

CHAPTER

FIVE

ENSEMBLE LEARNING

class sealion.ensemble_learning.**RandomForest**(*num_classifiers=20*, *max_branches=inf*, *min_samples=1*, *replacement=True*, *min_data=None*)

Random Forests may seem intimidating but they are super simple. They are just a bunch of Decision Trees that are trained on different sets of the data. You give us the data, and we will create those different sets. You may choose for us to sample data with replacement or without, either way that's up to you. Keep in mind that because this is a bunch of Decision Trees, classification is only supported (avoid using decision trees for regression - it's range of predictions is limited.) The random forest will have each of its decision trees predict on data and just choose the most common prediction (not the average.)

Enjoy this module - it's one of our best.

__init__(*num_classifiers=20*, *max_branches=inf*, *min_samples=1*, *replacement=True*, *min_data=None*)

Parameters

- **num_classifiers** – Number of decision trees you want created.
- **max_branches** – Maximum number of branches each Decision Tree can have.
- **min_samples** – Minimum number of samples for a branch in any decision tree (in the forest) to split.
- **replacement** – Whether or not any of the data points in different chunks/sets of data can overlap.
- **min_data** – Minimum number of data there can be in any given data chunk. Each classifier is trained on a chunk of data, and if you want to make sure each chunk has 3 points for example you can set min_data = 3. It's default is 50% of the amount of data, the None is just a placeholder.

evaluate(*x_test*, *y_test*)

Parameters

- **x_test** – testing data (2D)
- **y_test** – testing labels (1D)

Returns accuracy score

fit(*x_train*, *y_train*)

Parameters

- **x_train** – 2D training data
- **y_train** – 1D training labels

Returns

give_best_tree(*x_test*, *y_test*)

You give it the data and the labels, and it will find the tree in the forest that does the best. Then it will return that tree. You can then take that tree and put it into the DecisionTree class using the give_tree method.

Parameters

- **x_test** – testing data (2D)
- **y_test** – testing labels (1D)

Returns tree that performs the best (dictionary data type)

predict(*x_test*)

Parameters **x_test** – testing data (2D)

Returns Predictions in 1D vector/list.

visualize_evaluation(*y_pred*, *y_test*)

Parameters

- **y_pred** – predictions from the predict() method
- **y_test** – labels for the data

Returns a matplotlib image of the predictions and the labels ("correct answers") for you to see how well the model did.

class sealion.ensemble_learning.**EnsembleClassifier**(*predictors*, *classification=True*)

Aside from random forests, voting/ensemble classifiers are also another popular way of ensemble learning. How it works is by training multiple different classifiers (you choose!) and predicting the most common class (or the average for regression - more on that later.) Pretty simple actually, and works quite effectively. This module also can tell you the best classifier in a group with its get_best_predictor(), so that could be useful. Similar to `give_best_tree()` in the random forest module, what it does is give the class of the algorithm that did the best on the data you gave it. This can also be used for rapid hypertuning on the exact same module (giving the same class but with different parameters in the init.)

Example:

```
>>> from sealion.regression import SoftmaxRegression
>>> from sealion.naive_bayes import GaussianNaiveBayes
>>> from sealion.nearest_neighbors import KNearestNeighbors
>>> ec = EnsembleClassifier({'algo1': SoftmaxRegression(num_classes=3), 'algo2':␣
↪GaussianNaiveBayes(), 'algo3': KNearestNeighbors()},
...     classification=True)
>>> ec.fit(X_train, y_train)
>>> y_pred = ec.predict(X_test) # predict
>>> ec.evaluate_all_predictors(X_test, y_test)
algo1 : 95%
algo2 : 90%
algo3 : 75%
>>> best_predictor = ec.get_best_predictor(X_test, y_test) # get the best predictor
>>> print(best_predictor) # is it Softmax Regression, Gaussian Naive Bayes, or␣
↪KNearestNeighbors that did the best?
<regression.SoftmaxRegression object at 0xsomethingsomething>
>>> y_pred = best_predictor.predict(X_test) # looks like softmax regression, let's␣
↪use it
```

Here we first important all the algorithms we are going to be using from their respective modules. Then we create an ensemble classifier by passing in a dictionary where each key stores the name, and each value stores the algorithm. `Classification = True` by default, so we didn't need to put that (if you want regression put it to False. A good way to remember `classification = True` is the default is that this is an EnsembleCLASSIFIER.)

We then fitted that and got it's predictions. We saw how well each predictor did (that's where the names come in) through the `evaluate_all_predictors()` method. We could then get the best predictor and use that class. Note that this class will ONLY use algorithms other than neural networks, which should be plenty. This is because neural networks have a different `evaluate()` method and typically will be more random in performance than other algorithms.

I hope that example cleared anything up. The `fit()` method trains in parallel (thanks joblib!) so it's pretty fast. As usual, enjoy this algorithm!

`__init__`(*predictors*, *classification=True*)

Parameters

- **`predictors`** – dict of `{name (string):  algorithm (class)}`. See example above.
- **`classification`** – is it a classification or regression task? default classification - if regression set this to False.

`evaluate`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns accuracy score

`evaluate_all_predictors`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns None, just prints out the name of each algorithm in the predictors dict fed to the __init__ and its score on the data given.

`fit`(*x_train*, *y_train*)

Parameters

- **`x_train`** – 2D training data
- **`y_train`** – 1D training labels

Returns

`get_best_predictor`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns the class of the algorithm that did best on the given data. look at the above example if this doesn't make sense.

`predict`(*x_test*)

Parameters `x_test` – testing data (2D)

Returns Predictions in 1D vector/list.

`visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions from the predict() method
- `y_test` – labels for the data

Returns a matplotlib image of the predictions and the labels (“correct answers”) for you to see how well the model did.

CHAPTER SIX

MIXTURES

class sealion.mixtures.**GaussianMixture**(*n_clusters=5*, *retries=3*, *max_iters=200*, *kmeans_init=True*)

Gaussian Mixture Models are really just a fancier extension of KMeans. Make sure you know KMeans before you read this. If you are unfamiliar with KMeans, feel free to look at the examples on GitHub for unsupervised clustering or look at the unsupervised clustering documentation (which contains the KMeans docs.) You may also want to know what a gaussian (normal) distribution is.

In KMeans, you make the assumption that your data is in spherical clusters, all of which are the same shape. What if your data is instead in such circles, but also maybe ovals that are thin, tall, etc. Basically what if your clusters are spherical-esque but of different shapes and sizes.

This difference can be measured by the standard deviation, or variance, of each of the clusters. You can think of each cluster as a gaussian distribution, each with a different mean (similiar to the centroids in KMeans) and standard deviation (which effects how skinny/wide it is.) This model is called a mixture, because it essentially is just a mixture of gaussian functions. Some of the clusters maybe bigger than others (aside from shape), and this is also taken into account with a mixture weight - a coefficient that is multiplied to each gaussian distribution.

With this gaussian distribution, you can then take any points and assign them to the cluster they have the highest change of being in. Because this is probabilities, you can say that a given data point has a 70% chance of being in this class, 20% this class, 5% this class, and 5% this other class - which is something you cannot do with KMeans. The parameters of the gaussian distribution are learnt using an algorithm known as Expectation Maximization, which you may want to learn about (super cool)!

You can also do anomaly detection with Gaussian Mixture Models! You do this by looking at the probability each data point belonged to the cluster it had the highest chance of being in. We can call this list of a probability that a data point belonged to the cluster it was chosen to for all data points "confidences". If a given data points confidence is in the lowest _blank_ (you set this) percent of confidences, it's marked as an anomaly. This _blank_ is from 1 to 100, and is essentially what percent of your data you think are outliers.

To find the best value for n_clusters, we also have a visualize_elbow_curve method to do that for you. Check the docs below if you're interested!

That's a lot, so let's take a look at its methods!

NOTE: X SHOULD BE 2D FOR ALL METHODS

__init__(*n_clusters=5*, *retries=3*, *max_iters=200*, *kmeans_init=True*)

Parameters

- **n_clusters** – number of clusters assumed in the data
- **retries** – number of times the algorithm will be tried, and then the best solution picked

Max_iters maximum number of iterations that the algorithm will be ran for each retry

Kmeans_init whether the centroids found by using KMeans should be used to initialize the means in this Gaussian Mixture. This will only work if your data is in more than one dimension. If you are using a lot of retries, this may not be a good option as it may lead to the same solution over and over again.

`aic()`

Returns the AIC metric (lower is better.)

`anomaly_detect`(*X*, *threshold*)

Parameters

- `X` – prediction data
- `threshold` – what percent of the data you believe is an outlier

Returns whether each data point in X is an anomaly based on whether its confidence in the cluster it was assigned to is in the lowest **threshold** percent of all of the confidences.

`bic()`

Returns the BIC metric (lower is better.)

`confidence_samples`(*X*)

This method essentially gives the highest probability in the rows of probabilities in the matrix that is the output of the `soft_predict()` method.

Translation:

It's telling you the probability each data point had in the cluster it had the largest probability in (and ultimately got assigned to), which is really telling you how confident it is that a given data point belongs to the data.

Parameters `X` – prediction data

`fit`(*X*)

this method finds the parameters needed for Gaussian Mixtures to function

:param X : training data

`predict`(*X*)

this method returns the cluster each data point in X belongs to

:param X : prediction data

`soft_predict`(*X*)

this method returns the probability each data point belonged to each cluster. This is stored in a matrix with the length of X rows and the width of the amount of clusters.

:param X : prediction data

`visualize_clustering`(*color_dict*)

This method will not work for data that has only 1 dimension (univariate.) It will plot the data you just gave in the fit() method.

Parameters `color_dict` – parameter of the label a cluster was assigned to to its color (must be matplotlib compatible) The color dict could be `{0 :  "green", 1 :  "blue", 2 : "red"}` for example.

`visualize_elbow_curve`(*min_n_clusters=2*, *max_n_clusters=5*)

This method tries different values for n_cluster, from min_n_cluster to max_n_cluster, and then plots their AIC and BIC metrics. Finding the n_cluster that leads to the "elbow" is probably the optimal n_cluster value.

Parameters

- `min_n_clusters` – the minimum value of n_clusters to be tried
- `max_n_clusters` – the max value of n_clusters to be tried

CHAPTER

SEVEN

NEAREST NEIGHBORS

class `sealion.nearest_neighbors.`**`KNearestNeighbors`**(*k=5*, *regression=False*)

Arguably the easiest machine learning algorithm to make and understand. Simply looks at the k closest points (you give these points) for a data point you want to predict on, and if the majority of the closest k points are class X, it will predict back class X. The k number is decided by you, and should be odd (what happens if there's a tie?). If used for regression (we support that), it will just take the average of all of their values. If you are going to use regression, make sure to set regression = True - otherwise you will get a very low score in the evaluate() method as it will assume it's for classification (KNNs typically are.)

A great introduction into ML - maybe consider using this and then seeing if you can beat it with your own version from scratch.If you can - please send it on GitHub!

Other than that, enjoy!

`__init__`(*k=5*, *regression=False*)

Parameters

- **`k`** – number of points for a prediction used in the algorithm
- **`regression`** – are you using regression or classification (if classification - do nothing)

`evaluate`(*x_test*, *y_test*)

Parameters

- **`x_test`** – testing data (2D)
- **`y_test`** – testing labels (1D)

Returns accuracy score (r^2 score if regression = True)

`fit`(*x_train*, *y_train*)

Parameters

- **`x_train`** – 2D training data
- **`y_train`** – 1D training labels

Returns

`predict`(*x_test*)

Parameters **`x_test`** – 2D prediction data

Returns predictions in 1D vector/list `visualize_evaluation`(*y_pred*, *y_test*)

Parameters

- `y_pred` – predictions given by model, 1D vector/list
- `y_test` – actual labels, 1D vector/list

Visualize the predictions and labels to see where the model is doing well and struggling.

CHAPTER

EIGHT

UTILITIES

sealion.utils.**confusion_matrix**(*y_pred*, *y_test*, *plot=True*)

Confusion matrices are an often used tool for seeing how well your model did and where it can improve.

A confusion matrix is just a matrix with the number of a certain class classified as another. So if your classifier predicted `[0, 1]` and the correct answers were `[1, 0]` - then you would get 1 zero predicted as a 1, 1 one predicted as a 0, and no 0s predicted as 0s and 1s predicted as 1s. By default this method will plot the confusion matrix, but if you don't want it just set it to False.

Some warnings - make sure that y_pred and y_test are both 1D and start at 0. Also make sure that all predictions in y_pred exist in y_test. This will probably not be a big deal with traditional datasets for machine learning.

To really understand this function try it out - it'll become clear with the visualization.

Parameters

- **y_pred** – predictions (1D)
- **y_test** – labels (1D)
- **plot** – whether or not to plot, default True

Returns the matrix, and show a visualization of the confusion matrix (if plot = True)

sealion.utils.**one_hot**(*indices*, *depth*)

So you've got a feature in a data where a certain value represents a certain category. For example it could be that 1 represents sunny, 2 represents rainy, and 0 represents cloudy. Well what's the problem? If you feed this to your model - it's going to think that 2 and 1 are similar, because they are just 1 apart - despite the fact that they are really just categories. To fix that you can feed in your features - say it's a list like `[2, 2, 1, 0, 1, 0]` and it will be one hot encoded to whatever depth you please.

Here's an example (it'll make it very clear):

```
>>> features_weather = [1, 2, 2, 2, 1, 0]
>>> one_hot_features = one_hot(features_weather, depth  = 3) #depth is three here␣
↪because you have 3 categories - rainy, sunny, cloudy
>>> one_hot_features
[[0, 1, 0], #1 is at first index
 [0, 0, 1], #2 is at second index
 [0, 0, 1], #2 is at second index
 [0, 0, 1], #2 is at second index
 [0, 1, 0], #1 is at first index
 [1, 0, 0]] #0 is at 0 index
```

That looks like our data features, one hot encoded at whatever index. Make sure to set the depth param correctly.

For these such things, play around - it will help.

Parameters

- **indices** – features_weather or something similar as shown above
- **depth** – How many categories you have

Returns one-hotted features

sealion.utils.**revert_one_hot**(*one_hot_data*)

Say from the `one_hot()` data you've gotten something like this :

```
[[0, 0, 1],
[1, 0, 0],
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]]
```

and you want to change it back to its original form. Use this function - it will turn that one-hotted data above to `[2, 0, 1, 2, 0]` - which is just the index of the one in each data point of the list. Hence it is reverting the one_hot transformation.

Parameters **one_hot_data** – data in one_hot form. Must be a numpy array (2D)!

Returns index of the one for each of the data points in a 1D numpy array

sealion.utils.**revert_softmax**(*softmax_data*)

Say from the softmax function (in neural networks) you've gotten something like this :

```
[[0.2, 0.3, 0.5],
[0.8, 0.1, 0.1],
[0.15, 0.8, 0.05],
[0.3, 0.3, 0.4],
[0.7, 0.15, 0.15]]
```

and you want to change it back to its pre-softmax form. Use this function - it will turn that softmax-ed data above to `[2, 0, 1, 2, 0]` - which is just the index of the one in each data point of the list. Hence it is reverting the softmax transformation.

Parameters **softmax_data** – data in one_hot form. Must be a numpy array (2D)!

Returns index of the one for each of the data points in a 1D numpy array

sealion.utils.**precision**(*tp*, *fp*, *tn*, *fn*)

Precision is simply a measure of how much of the data we said are positive are actually positive. Used mostly for binary classification.

Parameters

- **tp** – number of true positives
- **fp** – number of false positives
- **tn** – number of true negatives
- **fn** – number of false negatives

Returns precision metric

sealion.utils.**recall**(*tp*, *fp*, *tn*, *fn*)

Recall is a measure of how much of the data that is actually positive was classified to be positive. There's a tradeoff between precision and recall, because as you decrease precision by predicting less positives, you increase recall by predicting more negatives (that are really positive - this is known as a false negative.)

Parameters

- `tp` – number of true positives
- `fp` – number of false positives
- `tn` – number of true negatives
- `fn` – number of false negatives

Returns recall metric

`sealion.utils.`**`f1_score`**(*precision*, *recall*)

The f1_score is harmonic mean between the precision and recall scores. It is used to assemble them into one score. The harmonic mean is used often times to combine information about different measures, even if they are in different units.

:param precision : precision score :param recall : recall score :return: f1_score

INDEX