StreamPy.ML Tutorial

# Introduction

## Design

The ML framework is designed as a plug-and-play model for machine learning on streaming data. This type of abstraction enables a wide variety of machine learning algorithms to be used on streaming data. There are two main applications for which the ML framework can be used: continual learning and batch learning.

### Continual learning

Continual learning refers to problems in which machine learning is continuously run over a stream of data. For example, an algorithm for determining credit card fraud may use the last 200 transactions to determine if the current transaction is an anomaly. Over time, such an algorithm will adapt to the customer’s recent purchasing patterns and will continually update itself.

### Batch learning

Batch learning refers to problems in which machine learning is run once on an offline dataset. This is usually the prevalent scenario for common machine learning problems. However, such problems usually deal with an offline dataset for training and an offline dataset for prediction. Batch learning as used with streaming data differs from this. A machine learning model is trained once for an offline dataset as in offline batch learning; however, prediction values are generated continuously for a stream of data. For example, in a scenario where the user wants to analyze the sentiment of Twitter tweets in real-time, a machine learning model may be trained once on a labeled dataset and applied in real-time to a stream of tweets.

### Goals

The ML framework aims to facilitate the use of machine learning algorithms on streaming data. In this perspective, the goal is to minimize the amount of time a novice programmer needs to implement such an application. We define novice programmer as a person proficient in Python yet new to streaming data architecture. As the ML framework is built on top of PSTREAMS, it is very easy for a user to write code that runs machine learning algorithms on streaming data without advanced knowledge of the streaming architecture.

## Machine Learning Models

Machine learning models are mathematical functions in a defined hypothesis space that are used to predict output values for input values. A machine learning algorithm has two components: training and predicting. Given a training dataset, the algorithm first trains a model. It then uses the model to generate predictions. To abstract this process, the ML framework operates by splitting machine learning algorithms into these two components. The plug-and-play functionality comes from the ML framework’s ability to accept user-defined functions that run training and prediction respectively. To use the framework effectively, the user needs to take the following steps:

1. Determine the type of function desired to learn.
2. Implement a training function that uses a training dataset to train the function.
3. Implement a prediction function that uses the trained hypothesis function to generate a prediction value for an input.

These steps are general to machine learning; it is important that the user be able to recognize and separate a machine learning algorithm into these specific components.

There are two main classes of machine learning: supervised learning and unsupervised learning. Supervised learning refers to scenarios where the training dataset has labeled outputs and the goal is to learn these outputs generally. For example, a user may want to predict the price of a house given its location, year of construction, amenities, etc. In this case, the training dataset would consist of data points for many houses with the price labeled. Unsupervised learning refers to scenarios where the training dataset does not have labeled outputs and the goal is to learn patterns in the data. For example, re-examining the example discussed earlier about credit card fraud detection, the training data does not have any labels regarding if a transaction was an anomaly or not. Instead, the machine learning algorithm attempts to learn the underlying patterns in the transaction history to determine if the current transaction is an anomaly.

The ML framework is extremely flexible, supporting both supervised and unsupervised learning. The user only has to ensure that the training and prediction functions provided are consistent with each other.

# Basics

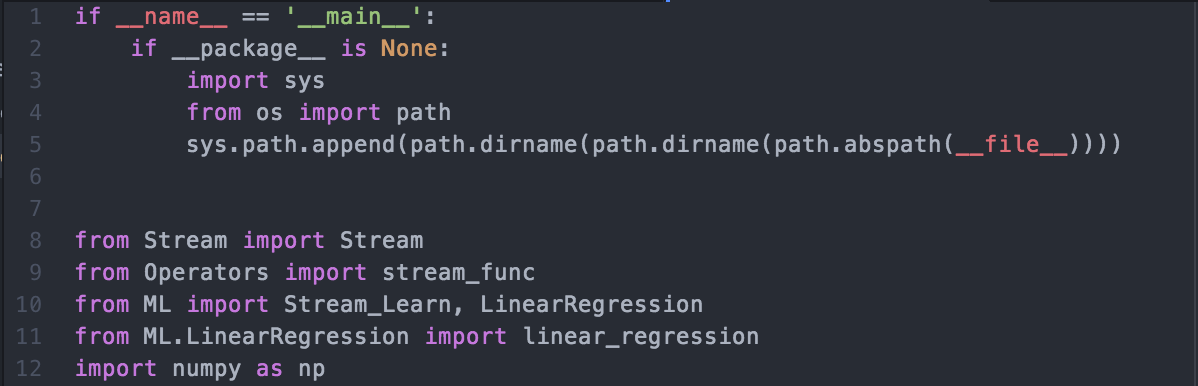
In order to use the ML framework, the user needs to choose a pair of training and prediction functions. The simple user can use predefined functions provided with the framework, or an advanced user can extend the framework by defining custom functions. Below, we go through a few examples to illustrate the use of predefined functions with the framework.

## Continual learning

### Linear Regression

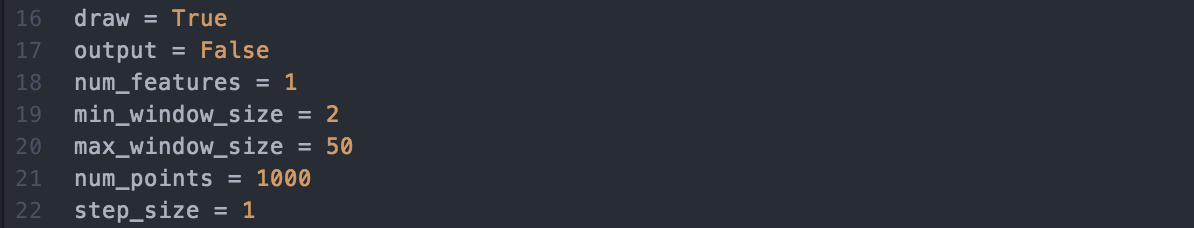
This example uses linear regression to predict trends for changing data. It is assumed that the user has basic knowledge of NumPy.

We first begin with importing the necessary modules to run the example.



The ML framework contains a class called Stream\_Learn, which runs the machine learning algorithms on the streaming data. The LinearRegression class contains training and prediction functions for using linear regression on streaming data. These functions are already prewritten and the user does not need to write new functions. The linear\_regression module contains helper functions for plotting the data as well as additional functions to run linear regression. We only need to use this module for plotting the data.

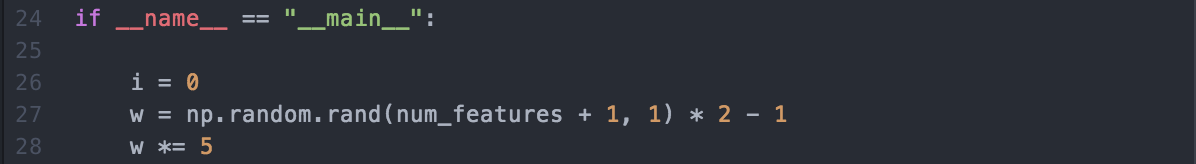
Next, we define some parameters.



These parameters have the following behaviors:

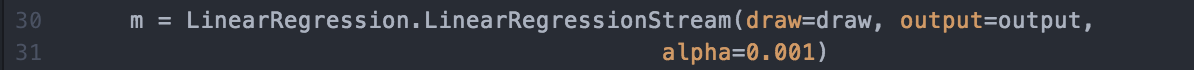
* draw: Signifies whether we want to plot the data or not.
* output: Signifies whether we want to print debug data (see docs for LinearRegression.LinearRegressionStream)
* num\_features: Describes the number of features our data has (since we want to plot it in this example, we will use data with 1 feature)
* min\_window\_size: Describes the minimum number of points needed to begin training. We set this to 2 since we only need 2 points to begin fitting a line.
* max\_window\_size: Describes the maximum number of points we use at any given time to train. Once this is reached, we begin advancing the window forward. See docs for Stream\_Learn.
* num\_points: Describes the number of points we add to the input stream.
* step\_size: Describes the number of points to advance the window forwards by. See docs for Stream\_Learn.

We have finished setting up the required pieces for the framework. At this point, we can begin using them to produce the final application. We first define a variable to count the number of points added and an initial value for a weight vector w.



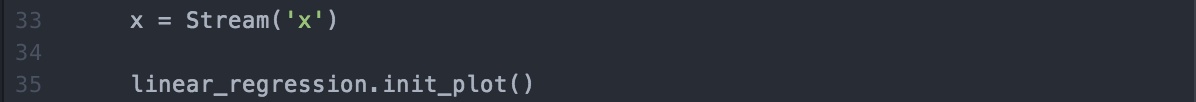
We initialize the weight vector to be a random 2-dimensional vector in [-5,5] x [-5,5].

Next, we initialize an instance of the LinearRegression class.

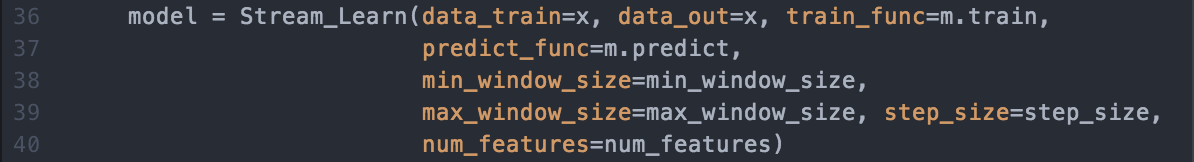


This instance contains functions for training and prediction using linear regression.

We initialize a Stream x to add input values to.



Finally, we initialize an instance of the ML framework.

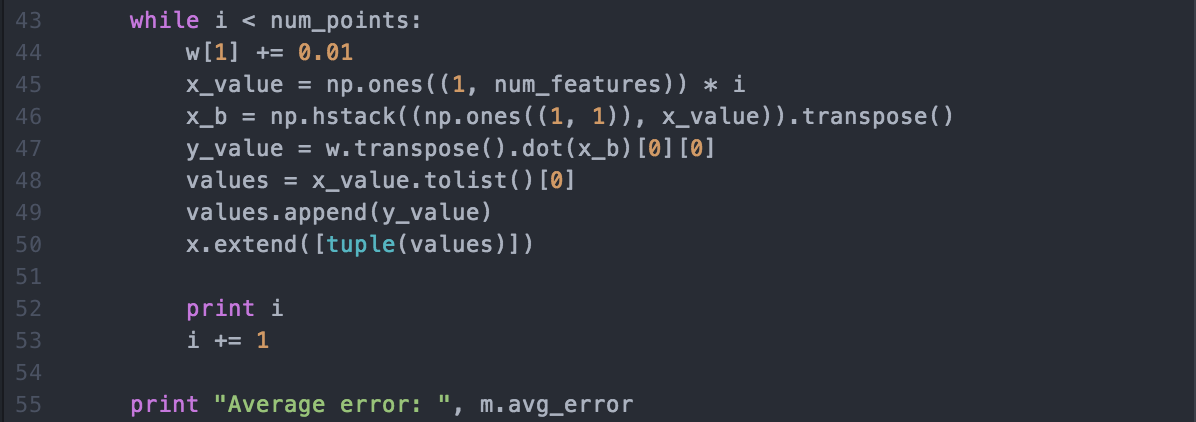


We see that the ML framework uses all the pieces we have set up together. The train and prediction functions are already defined for us.

To begin running the application, we call the run() method on the model.



We notice that after calling run(), nothing happens. This is because the input data stream x does not have any data yet. To begin training the model, we add data points to x.



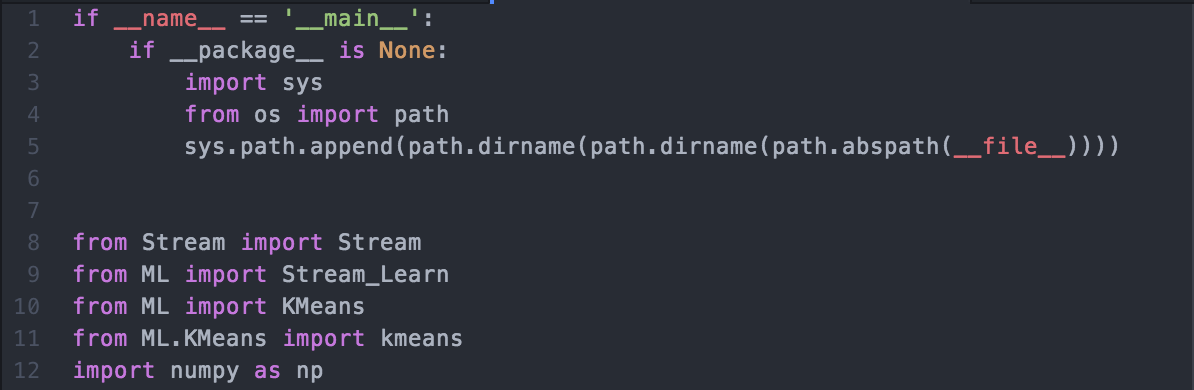
We generate new points by using the current iteration count i as the x value and the prediction y value generated by using the weight vector w as y. We first create a numpy array x\_value with dimensions 1xnum\_features with the value of i. We then prepend 1 to this array to account for the bias term and transpose the resulting array to create a vector with dimensions 2x1. We can then generate the y value by taking the dot product of w with the x vector. Finally, we create a list of the x value and the y value and extend this as a tuple into the x Stream. This list has 2 values in it. The ML framework expects the input stream to be a sequence of tuples, each with at least num\_features values. See the docs for Stream\_Learn for more information regarding the format of data.

After ending this while loop, the application will begin to run.

### K-Means

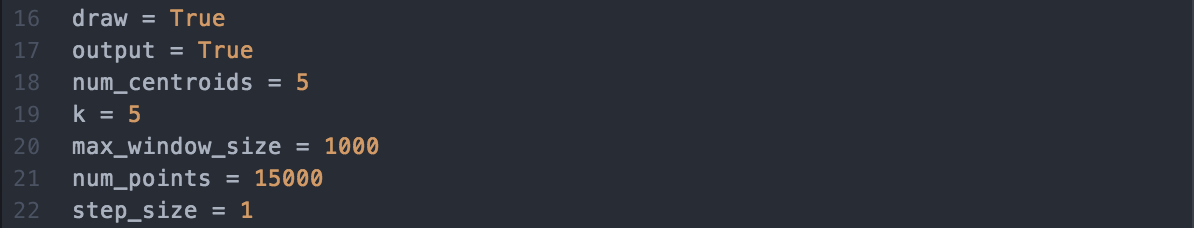
This example uses k-means clustering to predict centers for changing data. It is assumed that the user has basic knowledge of NumPy.

We first begin with importing the necessary modules to run the example.



Once again, we import the Stream\_Learn class from the ML framework. We also import modules for running k-means. The KMeans class provides training and prediction functions for using k-means on streaming data. Like the functions in the LinearRegression class, these functions are already prewritten and the user does not have to write new functions. The kmeans class contains helper functions for plotting the data as well as additional functions to run linear regression.

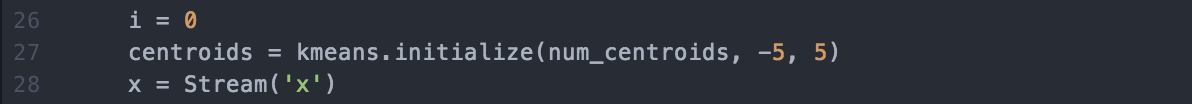
Next, we define some parameters.



These parameters have the following behaviors:

* draw: Signifies whether we want to plot the data or not.
* output: Signifies whether we want to print debug data (see docs for KMeans.KMeansStream)
* k: The number of centers we want to learn.
* max\_window\_size: Describes the maximum number of points we use at any given time to train. Once this is reached, we begin advancing the window forward. See docs for Stream\_Learn.
* num\_points: Describes the number of points we add to the input stream.
* step\_size: Describes the number of points to advance the window forwards by. See docs for Stream\_Learn.

We define a variable i to count the number of points added and the input data Stream x. We also define an initial numpy array of k centroids randomly placed in [-5, 5] x [-5, 5].



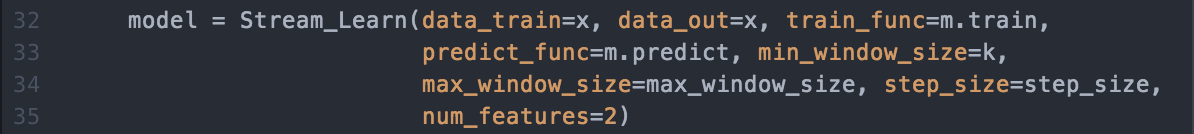
The function kmeans.initialize initializes num\_centroids in a 2 dimensional box with lower and upper bounds.

We now initialize the components of the framework.



Like the LinearRegression class, this instance contains functions for training and prediction using k-means.

Finally, we initialize an instance of the ML framework.

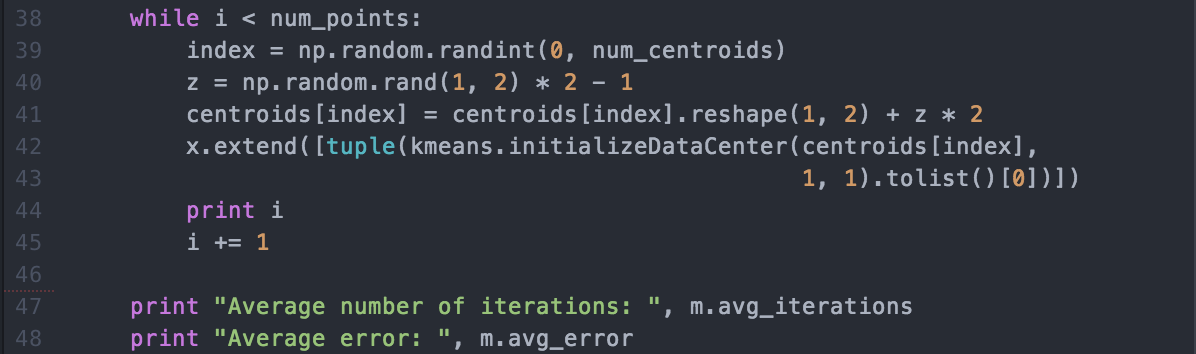


We see that the ML framework uses all the pieces we have set up together. We set the min\_window\_size to k points since we cannot predict k centers for less than k points.

To begin running the application, we call the run() method on the model.



Once again, nothing happens after calling run(). We need to add data to the input data stream x.



To add new data to x, we first choose a random centroid out of the centroids we have initialized. We then generate a random 1x2 numpy array z; this is a random velocity for this centroid. We move this centroid by adding this velocity to it, then add a random point generated around this centroid to x. We move the centroids to model changing data; it is natural to have data with drifting centroids over time.

We generate a random point around a centroid by using the function kmeans.initializeDataCenter(). See docs for KMean.kmeans.

We should now be able to see points being generated as well as their predicted center, indicated by color.

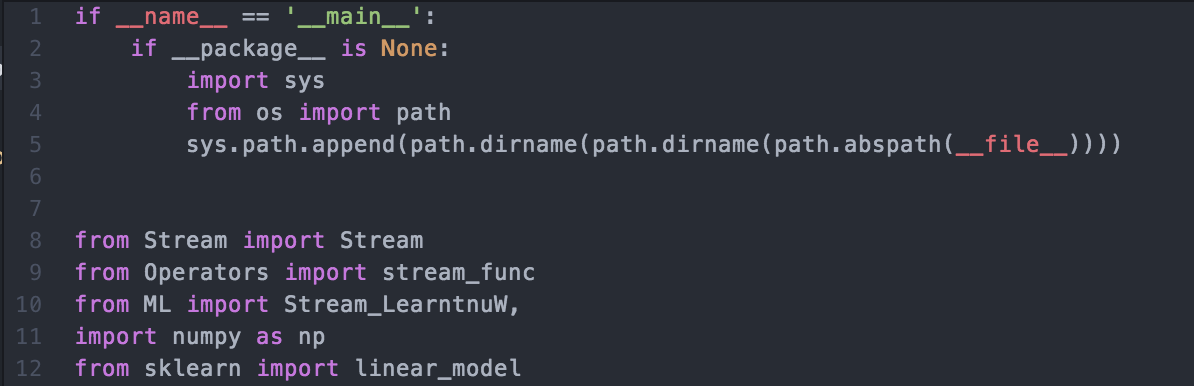
## Batch learning

We can also use the framework for applications that use batch learning. In this class of machine learning, we train on a batch training data set once and use the model to predict in real-time for a stream of prediction data.

### Linear Regression

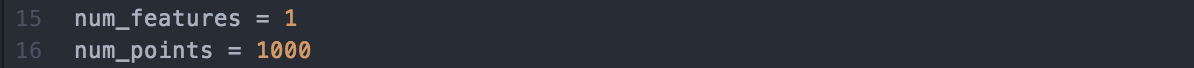
This example uses linear regression on a batch data set with points on the line . We generate a stream of random points and predict the y values. It is assumed that the user has basic knowledge of NumPy.

We first begin with importing the necessary modules to run the example.



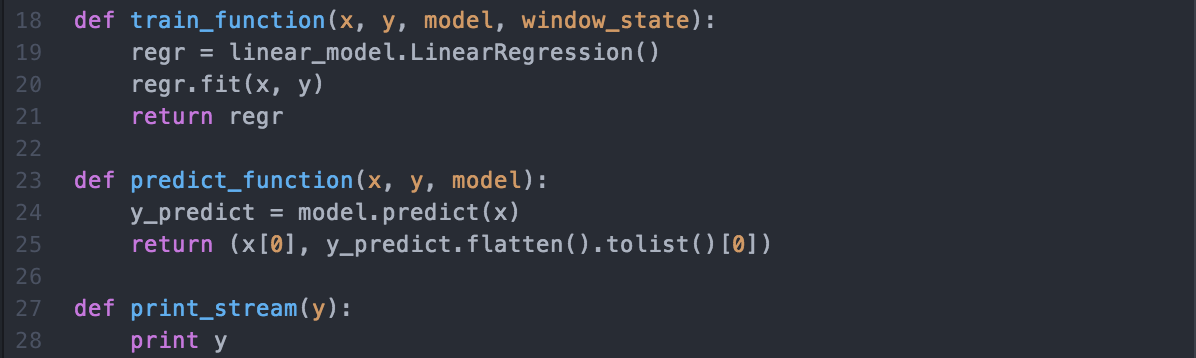
As for the continual learning examples, we import the Stream\_Learn class from the ML framework. We also import the linear\_model class from scikit-learn. We use this to run linear regression on the data.

Next, we define some parameters.

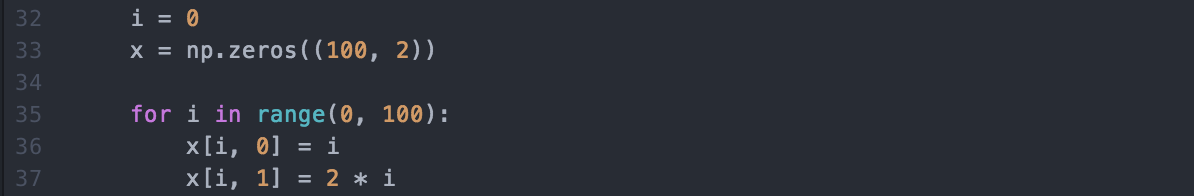


The num\_features variable describes the number of features in the input data. We set this to 1 for this example. The num\_points variable describes the number of points we want to add to the prediction stream to predict y values for.

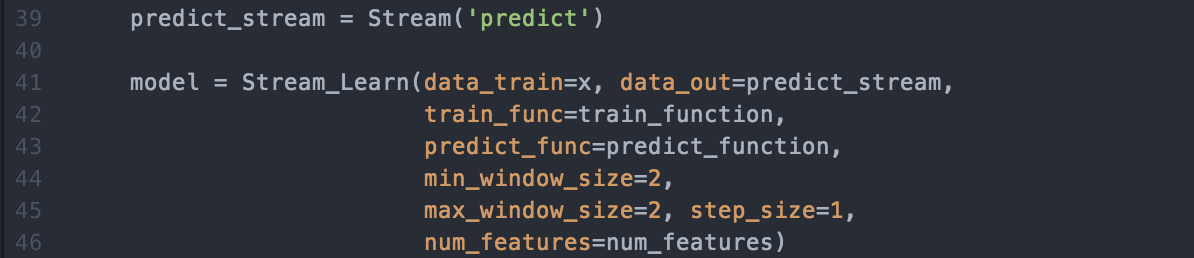
We now define training and prediction functions for linear regression. See the Advanced section for further information on how to write these functions. We also define a stream function to print the values in the output stream returned by the framework.



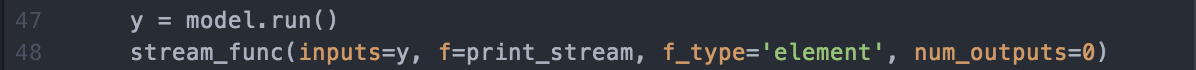
Next, we define a variable to count the number of points added to the prediction stream. We also define a numpy array x with dimensions 100x2. We add values to x on the line.



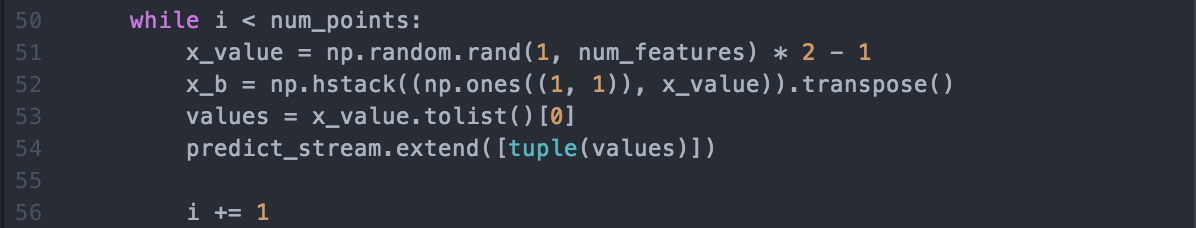
We initialize a Stream to add prediction values to. We can now initialize an instance of the ML framework.



We begin running the model by calling the run() method for model and assign the printing stream function we defined earlier to the output stream to print the values.



Finally, we add points to the prediction stream.



We generate a random x value, add the bias term to it, and add it to the prediction stream.

We see from the output of this example that the ML framework learns the prediction line and predicts the correct value for the prediction data. Furthermore, this model is trained only once. We note that the ML framework ignores the values for min\_window\_size, max\_window\_size, and step\_size. This example also shows how easy it is to plug an external machine learning algorithm from a library like scikit-learn into the ML framework.

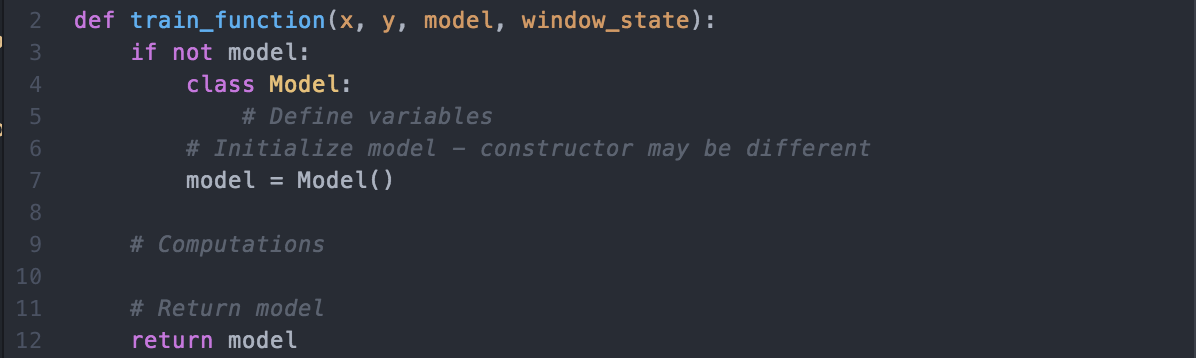
# Advanced

## Writing Training and Prediction Functions

So far, we have been using prewritten training and prediction functions. However, the library provided with the ML framework has a limited number of functions built-in and a user may want to write their own functions. We describe the general structure of these functions.

### Structure of a Training Function

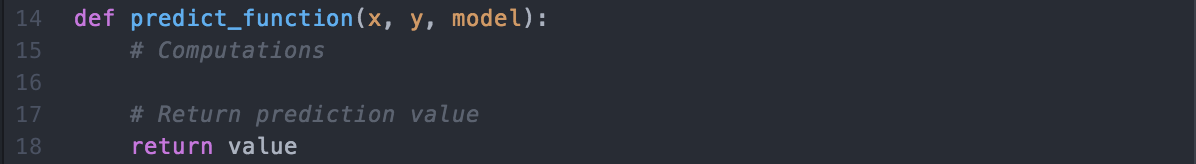
A training function has the following structure:



All training functions will have the same signature. This enables the function to define and maintain an internal state to save the machine learning model. We first check whether the provided state is already initialized. If it is not, then we initialize it by defining a class and setting it to be an instance of this class. This allows the state to hold any variables necessary for the machine learning model. At the end of this function, we return the updated state. This will be passed to the training function for the next window, allowing us to maintain the state between window transitions.

### Structure of a Prediction Function

A prediction function has the following structure:



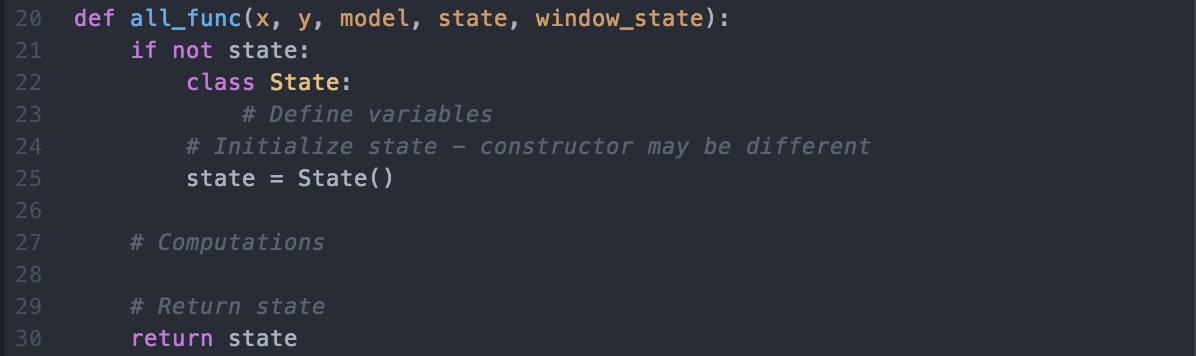
All prediction functions will have the same signature. The prediction function receives the current state containing the machine learning model as updated by the training function. We can use this to predict a value for a given input. This function returns the prediction value. This function should not modify the state – the only function that modifies the state is the training function.

## Additional Components

In addition to the training and prediction functions, the ML framework supports plug-and-play usage for filtering and general tasks. We describe how to write functions for these components.

### General functions

The ML framework supports the use of a general function on the window of training data. These function is referred to as all\_func. The general structure of this function is:



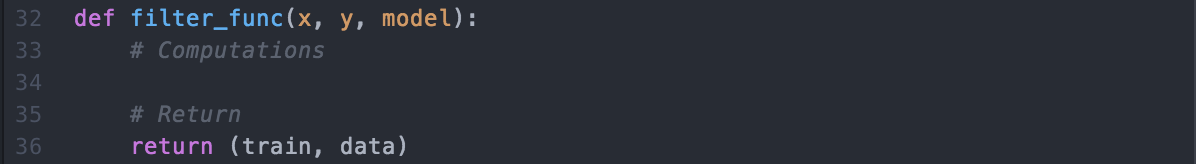
This function is similar to the training function in that it receives the window of training data and the current state containing the machine learning model. However, this function also receives a second state that it can use to store any information. Let us call the first state (containing the machine learning model) the model and this second state “state”. This second state is defined by this function and updated by this function only. We first check whether the provided state is already initialized. If it is not, then we initialize it by defining a class and setting it to be an instance of this class. We return this state at the end of the function.

This function can be used for many things. The most common use for this is plotting data. For example, the state defined by this function can be used to store an instance of a plot. This plot can be updated for each window.

Like the prediction function, this function should not modify the model state. This function will also be ignored if the training data is not a stream and batch learning is being used.

### Filtering

The ML framework supports the use of a filtering function for a stream of training data. This can be used to classify data as outliers and filter data from the training function. For example, if we are running linear regression and we receive a data point a significant distance away from the prediction line, we may choose to ignore it as noise. The filtering function has the following structure:



The filtering function takes a single data point in the form of tuples for the x and y values. It also takes the current model for machine learning. The return type is a list containing a Boolean value and a tuple.

The first value is a Boolean value that signifies whether the data is to be used for training (True) or if it is an outlier (False). The second value is a tuple containing the data in the correct format. This brings us to the second use of this function: transforming data. The input data stream for the ML framework must be in the form of a tuple containing values for a single data point. However, if a filtering function is used, this constraint can be ignored as long as the filtering function returns the data in the form of a tuple. For example, we may add JSON objects to the input data stream. The filtering function will then receive the JSON object along with the current model. We may choose to ignore some data points based on an attribute in the JSON data. We then return the Boolean value signifying whether we want to use this data for training, along with the actual data we want to use in the form of a tuple.

For further clarification, assume that the raw JSON data contains attributes for location and text. The filtering function may filter out data based on the location and return the text as a tuple with one value.

## Resetting

There may be cases when we would like to clear the training window of all data and reset it to the minimum window size. For example, consider a linear regression model using a window with a minimum size of 2 points and a maximum size of 100 points. Furthermore, let the window be full such that the window currently has 100 points. If the distribution of the training data changes, we would have to wait 100 points before the model predicts the new distribution with accuracy. Instead, if we are able to detect when the distribution of data changes, we can reset the window and begin retraining the model with 2 points. The following diagram illustrates the reset functionality:

Time

Signal

measurement

model

Model changed

To reset the window, we can call the reset() method of the Stream\_Learn. The window will be reset as soon as the window reaches the maximum window size. If the window is already full, the window will be reset as soon as reset() is called.

### Training and All\_Func

There is one parameter of the training function and the all\_func that we have not yet discussed: window\_state. This parameter is a tuple with the values (current\_window\_size, steady\_state, reset, step\_size, max\_window\_size). We describe these values below:

* current\_window\_size: The number of points in the current window
* steady\_state: Boolean, describes whether the window has reached the max\_window\_size
* reset: A Boolean value that describes whether the window should be reset. Set this to True to reset the window.
* step\_size: The number of points by which the window shifts.
* max\_window\_size: The maximum size of the window

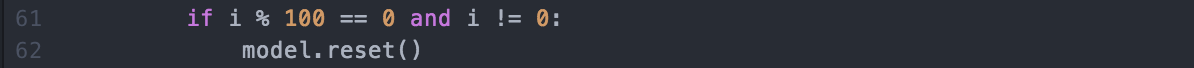
For the majority of users, these attributes are not useful. However, if we are writing a function that uses the incremental nature of the sliding window, then these attributes become very useful. For example, if we want to incremental calculate the sum of the current window, we need to know how many points are in the window, how many points will be removed when the window shifts, and how many points will be added when the window shifts. We can use the fact that the majority of the window stays the same to only add and remove sums for the new and old points. To see a real example of this, see the code for the incremental training function for linear regression.

## Improving the Linear Regression Example

We can use the advanced tools we have learned to improve the previous example of linear regression. We will add the following features:

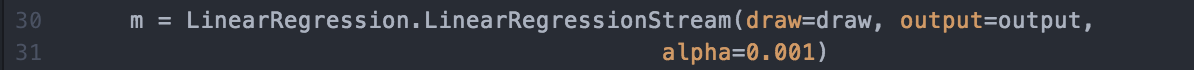
* Reset the window every 100 points
* Plot the data without resetting the plot

The first feature is quite easy to add. We simply add the following line in the while loop for adding data:

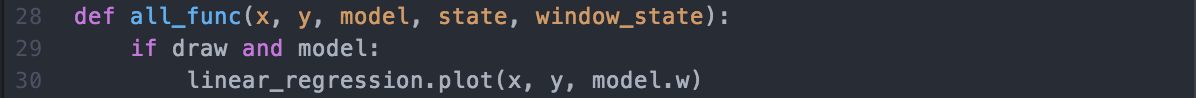


This code resets the window every 100 points and makes sure not to reset the window when no points have been added.

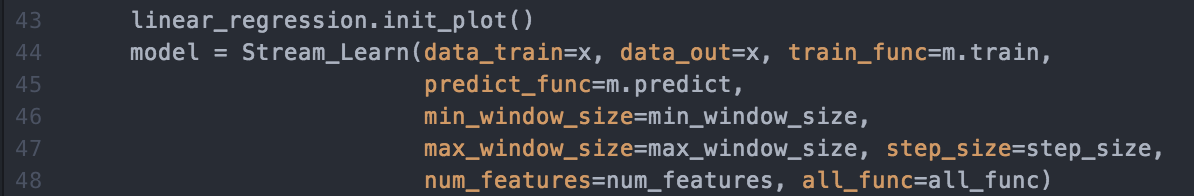
The second feature involves using the all\_func component of the ML framework. The reset functionality only resets the window for the training function, not the all\_func function. Currently, we have been using the training function to plot the data, as seen in the following code:



We want the plot to maintain the points even after a reset. To do this, we will use the all\_func function to plot the data.



We use the plot function in the linear\_regression model to plot the data. We do not need to save the plot in this case since we redraw the plot for every window. To enable this function in the ML framework, we pass it as a parameter for the initialization of model. We also initialize the plot.



Now, running this code will produce an example of linear regression where the line resets every 100 points. We can see the improvement in the prediction by resetting the window.