StreamPy.ML Tutorial

# Introduction

## Design

### Overview

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

The ML framework takes many parameters. These parameters are described in the documentation for the Stream\_Learn class. The ML framework takes different parameters depending on if continual learning or batch learning is being used.

We go through a few examples to illustrate the use of these parameters.

## 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.

INSERT CODE FOR IMPORTS

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.

INSERT CODE FOR 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.

At this point, we can begin defining the plug-and-play components of the ML framework. The first function we want to define is a function we call all\_func(). This component is designed to be a very general function on a window of unfiltered training data and is most commonly used for plotting data. We define this function below:

INSERT CODE FOR all\_func()

We describe the parameters that this function takes:

* x: A numpy array containing the x values for the window of training data. This array will have dimensions nx1 in our case since our input data only has 1 feature. Here, min\_window\_size < n < max\_window\_size.
* y: A numpy array containing the y values for the window of training data. This array wil have dimensions nx1 in our case. n is the same as for x.
* model: An instance of a training model. We define this later.
* window\_state: A tuple with the values (current\_window\_size, steady\_state, reset, step\_size, max\_window\_size). We will use this later.

See the docs for Stream\_Learn for more descriptions of these parameters. Any function designed to fit in the “all” component of the ML framework must have the same signature as all\_func().

Our function only plots the data. We do this by using the linear\_regression module. This module has a function called plot(), which plots data along with a prediction line.

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.

INSERT CODE FOR i, 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.

INSERT CODE for m

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

We initialize a plot using the linear\_regression module. We also initialize a Stream x to add input values to.

INSERT CODE FOR init\_plot(), x

Finally, we initialize an instance of the ML framework.

INSERT CODE FOR model

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.

INSERT CODE FOR run()

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.

INSERT CODE FOR while

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 1x1 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.

INSERT CODE FOR IMPORTS

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.

INSERT CODE FOR 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 a initial numpy array of k centroids randomly placed in [-5, 5] x [-5, 5].

INSERT CODE FOR i, x, centroids

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.

INSERT CODE FOR m

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

Finally, we initialize an instance of the ML framework.

INSERT CODE FOR model

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.

INSERT CODE FOR run()

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

INSERT CODE FOR while

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.

# Advanced

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:

INSERT CODE FOR training function

All training functions will have the same signature. This enables the function to define and maintain