**River: Online Machine Learning in Python**

**A Fast and Cheap Approach to Update an ML Model in Production**

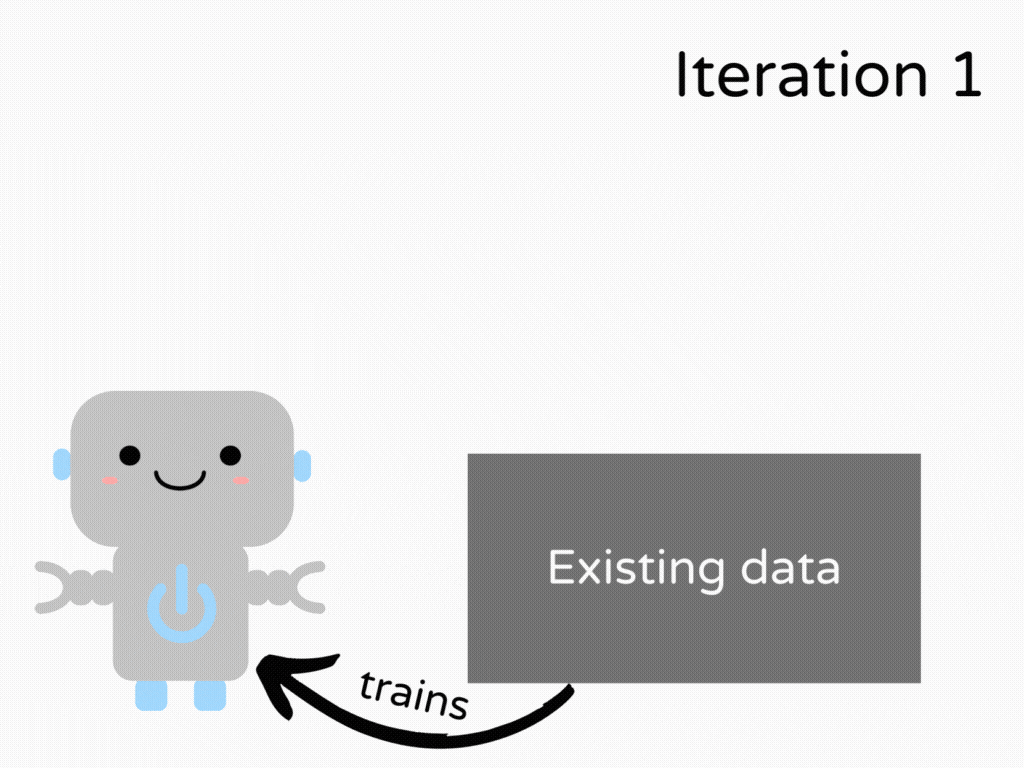
**Problem with Batch Learning**

It is common for data practitioners to use batch learning to learn from data. Batch learning is the training of ML models in batch. An ML pipeline with batch learning typically includes:

* Splitting the data into train and test sets
* Fitting a model to the train set
* Computing the performance of the model on the test…
* Pushing the model to production

However, in production, the pipeline doesn’t end here. To make sure the model is robust when the input data changes, data practitioners also need to periodically retrain the model on the combination of the **new dataset and the existing dataset**.

As the data grows, training the model takes more time and resources.



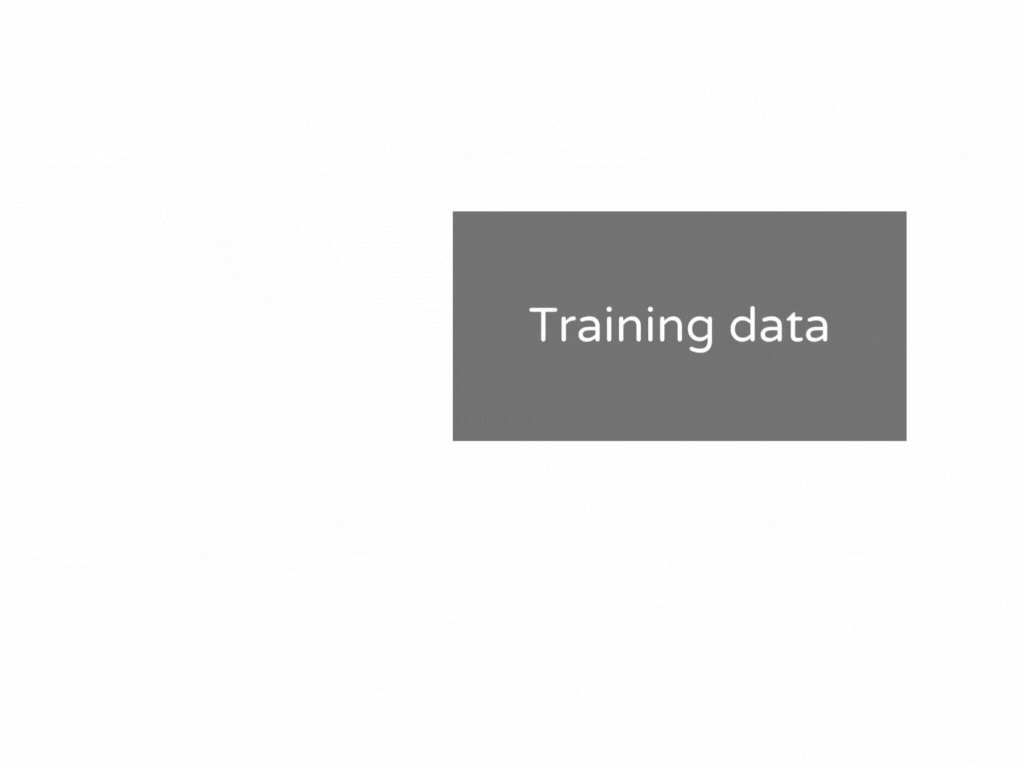
Demo of batch learning (by author)

Thus, batch learning is not ideal when:

* An application requires frequent model updates.
* Companies can’t afford computing resources to store and train big data.

**Introduction to Online Learning**

In online learning, the model is trained incrementally on a data stream instead of an entire dataset. In other words, the model only learns from an observation or a small group of observations at a time.



Demo of online learning (by author)

Thus, each learning step is fast and cheap, which makes it ideal for applications that change rapidly and for companies with limited computing resources.

In this article, you will learn how to use River to do machine learning on streaming data.

Feel free to play and fork the source code of this article here:

**[Data-science/streaming.ipynb at master · khuyentran1401/Data-science](https://github.com/khuyentran1401/Data-science/blob/master/machine-learning/river_streaming/streaming.ipynb" \t "_blank)**

**[You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…](https://github.com/khuyentran1401/Data-science/blob/master/machine-learning/river_streaming/streaming.ipynb" \t "_blank)**

[github.com](https://github.com/khuyentran1401/Data-science/blob/master/machine-learning/river_streaming/streaming.ipynb" \t "_blank)

**What is River?**

[River](https://github.com/online-ml/river) is a Python library for online machine learning. To install River, type:

pip install river

In the next few sections, we will compare using scikit-learn for batch learning and using River for online learning.

**Prepare the Data**

Before doing anything fancy, we will start with preparing our data.

Import the penguins dataset from seaborn:

import seaborn as sns  
  
df = sns.load\_dataset("penguins")

View the information of the data:

df.info()

Data columns (total 7 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 species 344 non-null object   
 1 island 344 non-null object   
 2 bill\_length\_mm 342 non-null float64  
 3 bill\_depth\_mm 342 non-null float64  
 4 flipper\_length\_mm 342 non-null float64  
 5 body\_mass\_g 342 non-null float64  
 6 sex 333 non-null object

Create the feature data (X) and the label data (y):

target = 'species'  
y = df[target]  
X = df.drop(target, axis=1)

Now we are ready to create an ML pipeline with scikit-learn and River!

**Batch Learning with scikit-learn**

Import useful libraries:

from sklearn.model\_selection import train\_test\_split  
from sklearn.pipeline import make\_pipeline  
from sklearn.impute import SimpleImputer  
from sklearn.preprocessing import OneHotEncoder  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.compose import ColumnTransformer  
from sklearn.metrics import confusion\_matrix, f1\_score

To train a model in batch learning, we often start with splitting the dataset into train and test sets:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=1)

Create a pipeline with scikit-learn’s transformers and classifier:

# Get numerical and categorical features  
numeric\_features = X\_train.select\_dtypes(exclude=object).columns  
categorical\_features = X\_train.select\_dtypes(include=object).columns  
  
# Specify transformers for each type of features  
numeric\_transformer = SimpleImputer()  
categorical\_transformer = make\_pipeline(  
 SimpleImputer(strategy="most\_frequent"), OneHotEncoder()  
)  
preprocessor = ColumnTransformer(  
 transformers=[  
 ("num", numeric\_transformer, numeric\_features),  
 ("cat", categorical\_transformer, categorical\_features),  
 ]  
)  
  
# Create a pipeline with transformers and classifier  
sklearn\_clf = make\_pipeline(preprocessor, DecisionTreeClassifier())

An overview of the pipeline:

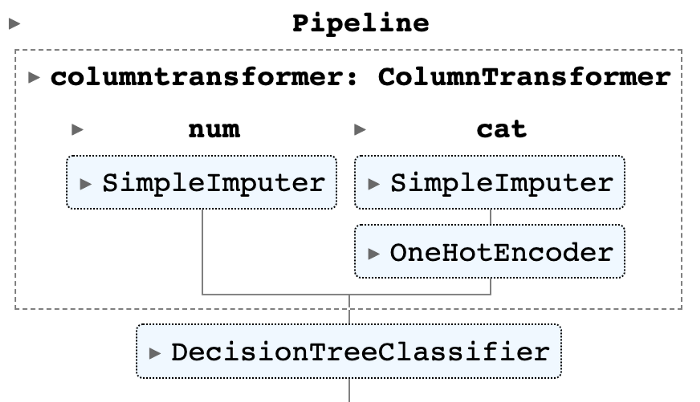


Image by Author

Train the model and predict the test data:

# Train the model  
sklearn\_clf.fit(X\_train, y\_train)  
  
# Get prediction  
y\_pred = sklearn\_clf.predict(X\_test)

These steps are pretty standard for data practitioners. Let’s turn this code into online learning and compare the difference between the two approaches.

**Online Learning with River**

**Stream through the dataset**

Import useful libraries:

from river import (  
 stream,  
 compose,  
 preprocessing,  
 evaluate,  
 metrics,  
 tree,  
 imblearn,  
 stats,  
)  
import numbers import numbers

In online learning, observations are provided one at a time. We will mimic this behavior by iterating through each row of two pandas DataFrames (X and y) with stream.iter\_pandas :

for xi, yi in stream.iter\_pandas(X, y):  
 pass

Let’s see how the last xi and yi look like:

>>> xi  
{'island': 'Biscoe',  
 'bill\_length\_mm': 49.9,  
 'bill\_depth\_mm': 16.1,  
 'flipper\_length\_mm': 213.0,  
 'body\_mass\_g': 5400.0,  
 'sex': 'MALE'}  
  
>>> yi  
'Gentoo'

**Compute running statistics**

Since there are some missing values in the dataset, we will impute the missing values with the mean of the dataset.

To find the mean of the dataset, we add up N non-null values in the dataset and divide the result by N.

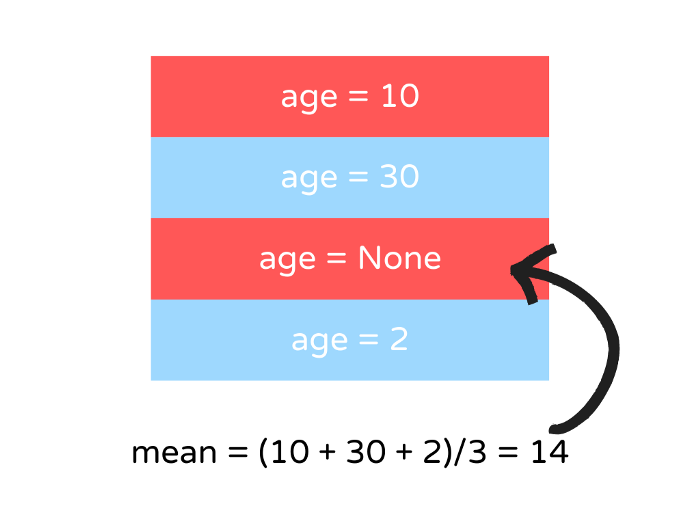


Image by Author

In online learning, we can’t apply the same procedure to compute the mean since we don’t know the values of the entire dataset. Thus, we will use *running statistics* to estimate the mean instead.

To compute the running mean, update the mean whenever there is a new value. Then use that running mean to update the missing value.

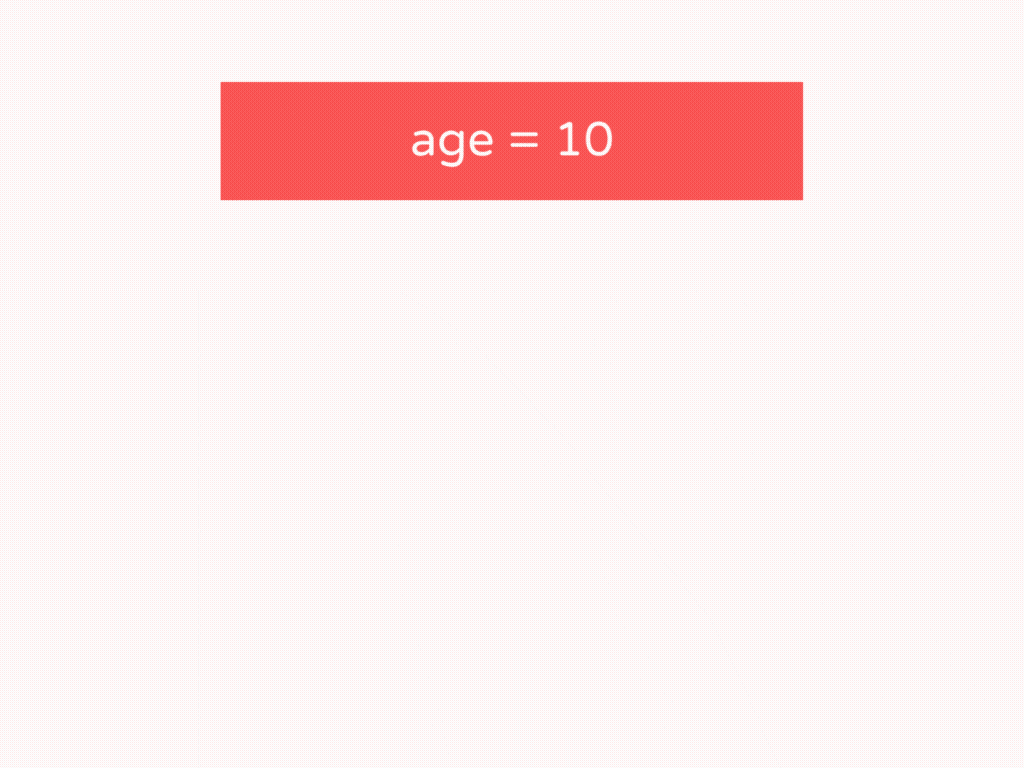


Image by Author

In River, we can use preprocessing.StatImputer to replace missing values with a running statistic.

To learn from one instance and transform that instance, we will use the learn\_one method and the transform\_one method.

X\_sample = [{"age": 10}, {"age": 30}, {"age": None}, {"age": 2}]  
mean = stats.Mean()  
imputer = preprocessing.StatImputer(("age", mean))  
for xi\_sample in X\_sample:  
 imputer.learn\_one(xi\_sample)  
 print(imputer.transform\_one(xi\_sample))

{'age': 10}  
{'age': 30}  
{'age': 20.0}  
{'age': 2}

**Create an ML pipeline**

river.compose provides several methods that are similar to the ones in sklearn.compose to build a machine learning pipeline.

Let’s use these methods to create a pipeline that transforms categorical and numerical features:

cat = (  
 compose.SelectType(object)  
 | preprocessing.StatImputer()  
 | preprocessing.OneHotEncoder(sparse=True)  
)  
num = compose.SelectType(numbers.Number) | preprocessing.StatImputer()  
preprocessor = num + cat

Use the pipeline to learn and transform an observation:

preprocessor.learn\_one(xi)  
preprocessor.transform\_one(xi)

{'island\_Biscoe': 1,  
 'bill\_length\_mm\_49.9': 1,  
 'bill\_depth\_mm\_16.1': 1,  
 'flipper\_length\_mm\_213.0': 1,  
 'body\_mass\_g\_5400.0': 1,  
 'sex\_MALE': 1,  
 'bill\_length\_mm': 49.9,  
 'bill\_depth\_mm': 16.1,  
 'flipper\_length\_mm': 213.0,  
 'body\_mass\_g': 5400.0}

Finally, we will use a decision tree algorithm to learn from the data.

Traditional batch decision trees can’t cope with online learning requirements since they need to be retrained with the entire dataset when there are new observations.

Thus, we will use [Hoeffding Tree](https://riverml.xyz/0.14.0/api/tree/HoeffdingTreeClassifier/) (HT) classifier for online learning instead. HT is the most popular family of incremental decision trees to date.

classifier = tree.HoeffdingTreeClassifier()

Learn more about [online machine learning with decision trees](https://maxhalford.github.io/slides/online-decision-trees.pdf).

Next, combine transformers and the HT classifier into a pipeline that transforms the data and learns from it:

def get\_pipeline():  
 cat = (  
 compose.SelectType(object)  
 | preprocessing.StatImputer()  
 | preprocessing.OneHotEncoder(sparse=True)  
 )  
 num = compose.SelectType(numbers.Number) | preprocessing.StatImputer()  
 classifier = tree.HoeffdingTreeClassifier()  
  
 return (num + cat) | classifier  
  
pipeline = get\_pipeline()

Visualize the pipeline:

pipeline

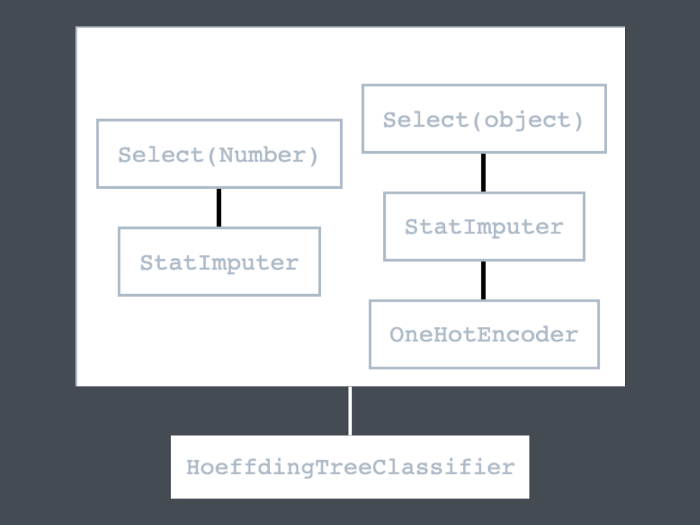


Image by Author

**Train the model and predict new observations**

Let’s use this pipeline to create a function that makes predictions and trains a model on a stream of data:

def train(X, y):  
 pipeline = get\_pipeline()  
  
 # Initialize metrics   
 f1\_score = metrics.MicroF1()  
 cm = metrics.ConfusionMatrix()  
  
 f1\_scores = []  
  
 # Iterate over the dataset  
 for xi, yi in stream.iter\_pandas(X, y, shuffle=True, seed=1):  
 # Predict the new sample  
 yi\_pred = pipeline.predict\_one(xi)  
  
 # Get the score  
 if yi\_pred is not None:  
 f1\_score.update(yi, yi\_pred)  
 f1\_scores.append(f1\_score.get() \* 100)  
 cm.update(yi, yi\_pred)  
  
 # Train the model with the new sample  
 pipeline.learn\_one(xi, yi)  
  
 return f1\_scores, cm, pipeline  
  
  
f1\_scores, cm, pipeline = train(X, y)

In the code above, we iterate through each sample. For each sample, we:

* Predict the new sample with the existing model
* Compute the micro-average F1 score and the confusion matrix for the new prediction then update the existing scores
* Save the new F1 score to a list of F1 scores for further analysis
* Train the model with the new sample and update the model



Image by Author

Let’s inspect the parameters of the tree classifier:

pipeline.steps['HoeffdingTreeClassifier'].summary

{'n\_nodes': 1,  
 'n\_branches': 0,  
 'n\_leaves': 1,  
 'n\_active\_leaves': 1,  
 'n\_inactive\_leaves': 0,  
 'height': 1,  
 'total\_observed\_weight': 344.0}

Visualize the change in the score as the iteration increases with a line plot.

import matplotlib.pyplot as plt   
  
def plot(scores: list):  
 iters = range(len(scores))  
 ax = sns.lineplot(x=iters, y=scores)  
 ax.set(xlabel='num\_iters', ylabel='score')  
 plt.show()  
   
plot(f1\_scores)

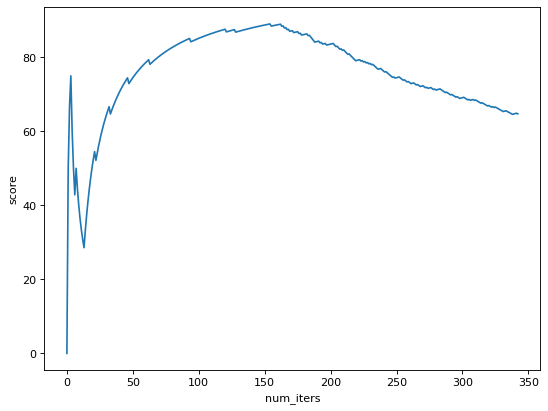


Image by Author

We can see that the micro F1 score reaches the highest value around the 150th iteration and then decreases afterward.

We can also evaluate the model’s performance on a streaming dataset with the evaluate.progressive\_val\_score method:

metric = metrics.MicroF1()  
  
evaluate.progressive\_val\_score(  
 dataset=stream.iter\_pandas(X, y, shuffle=True, seed=1),  
 model=pipeline,  
 metric=metric,  
 print\_every=50, # print every 50 iterations  
)

[50] MicroF1: 73.47%  
[100] MicroF1: 84.85%  
[150] MicroF1: 88.59%  
[200] MicroF1: 83.42%  
[250] MicroF1: 74.70%  
[300] MicroF1: 68.90%  
MicroF1: 64.72%

The final score is pretty low. Let’s dive deeper into the model performance by viewing the confusion matrix:

cm

Adelie Chinstrap Gentoo   
 Adelie 143 8 0   
Chinstrap 44 22 2   
 Gentoo 66 1 57

We can see that the majority of false predictions are classified as the Adelie species. This could be because the data is imbalanced. We can confirm this by looking at the count for each value in y .

y.value\_counts()

Adelie 152  
Gentoo 124  
Chinstrap 68

When inspecting the values of y , we can see that there are more Adelie labels than other labels.

In the next section, we will learn how to deal with an imbalanced dataset with River.

**Deal with Imbalanced Data**

To deal with imbalanced data, we will use the RandomSampler class to adjust the number of samples for each label, which allows us to attain the target distribution.

RandomSampler uses both [under-sampling and over-sampling](https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/) to fit the specified constraints.

For example, in the example below, desired\_dist={"Adelie": 0.1, "Gentoo": 0.4, "Chinstrap": 0.5} tells River to sample the data so that the classifier encounters 10% of Adelie , 40% of Gentoo , and 50% of Chinstrap .

classifier = tree.HoeffdingTreeClassifier()  
sampler = imblearn.RandomSampler(  
 classifier=classifier,  
 desired\_dist={"Adelie": 0.1, "Gentoo": 0.4, "Chinstrap": 0.5},  
 seed=2,  
)

Let’s incorporate the RandomSampler class to our pipeline and see if the performance improves:

def get\_pipeline():  
 # Specify the transfomers  
 cat = (  
 compose.SelectType(object)  
 | preprocessing.StatImputer()  
 | preprocessing.OneHotEncoder(sparse=True)  
 )  
 num = compose.SelectType(numbers.Number) | preprocessing.StatImputer()  
  
 # Specify classifiers  
 classifier = tree.HoeffdingTreeClassifier()  
 sampler = imblearn.RandomSampler(  
 classifier=classifier,  
 desired\_dist={"Adelie": 0.1, "Gentoo": 0.4, "Chinstrap": 0.5},  
 seed=2,  
 )   
 return (num + cat) | sampler  
  
f1\_scores, cm, pipeline = train(X, y)

Plot the micro-average F1 scores across iterations:

plot(f1\_scores)

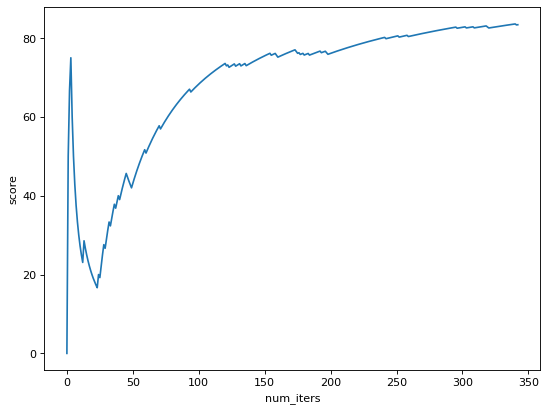


Image by Author

The micro-average F1 score initially decreases in the first few iterations and then increases in the later iterations.

The confusion matrix also shows that there are more correct predictions when using RandomSampler.

cm

Adelie Chinstrap Gentoo   
 Adelie 111 37 3   
Chinstrap 5 62 1   
 Gentoo 5 6 113

Let’s call the evaluate.progressive\_val\_score method to get the F1 score for each iteration:

pipeline = get\_pipeline()  
  
metric = metrics.MicroF1()  
  
evaluate.progressive\_val\_score(  
 dataset=stream.iter\_pandas(X, y, shuffle=True, seed=1),  
 model=pipeline,  
 metric=metric,  
 print\_every=50,  
)

[50] MicroF1: 42.86%  
[100] MicroF1: 67.68%  
[150] MicroF1: 75.17%  
[200] MicroF1: 75.88%  
[250] MicroF1: 80.32%  
[300] MicroF1: 82.61%  
MicroF1: 83.38%

Nice! The final micro-average F1 score is 83.38% when using RandomSampler .

**Conclusion**

Congratulations! You have just learned how to use River to do online machine learning. I hope this article will give you the knowledge needed to to create a production-ready machine learning model.