Supplementary Materials - *Source Code* for NeurIPS 2021 Submission, Paper #8392

Adversarial Robustness of Streaming Algorithms through Importance Sampling

1 Streaming k-means

```
from sklearn.datasets import make_blobs
import numpy as np
from clusopt_core.cluster import Streamkm
from pyspark.ml.linalg import Vectors
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.sql import *
from pyspark.sql import SQLContext
from pyspark.mllib.clustering import StreamingKMeans


# Create Dataset : regular batches around [-1,1] x [-1, 1]
#   and last batch around [9,10] x [9, 10]
NUM_REGULAR_BATCHES = 9
NUM_BATCHES = NUM_REGULAR_BATCHES + 1
NUM_REGULAR_BATCH_SAMPLES = 30
NUM_LAST_BATCH_SAMPLES = 15
NUM_SAMPLES = NUM_REGULAR_BATCHES*NUM_REGULAR_BATCH_SAMPLES + NUM_LAST_BATCH_SAMPLES
CLUSTER_STD_DEV = 0.3

batches = []
data = np.ndarray([0,2])
for i in range(NUM_BATCHES):
  cb = [-1, 1] if i < NUM_REGULAR_BATCHES else [9, 10]
  ns = NUM_REGULAR_BATCH_SAMPLES if i < NUM_REGULAR_BATCHES else NUM_LAST_BATCH_SAMPLES
  dataset, _ = make_blobs(n_samples=ns, centers=1, center_box=cb, random_state=42, cluster_s
  batches.append(dataset)
  data = np.concatenate([data, dataset])
```

```
# StreamKM++
CORESET_SIZE = 6

modelkm = Streamkm(
    coresetsize=CORESET_SIZE,
    length=NUM_SAMPLES,
    seed=42,
)

for i in range(NUM_BATCHES):
  modelkm.partial_fit(batches[i])
skm_centers, _ = modelkm.get_final_clusters(2, seed=42)
skm_coresets = modelkm.get_partial_cluster_centers()

# Spark Streaming KMeans
try:
  sc = SparkContext("local[*]", "RobuStreaming")
  ssc = StreamingContext(sc, 0.1)
  sqlCtx = SQLContext(sc)
  print("spark context initialized")
except:
  print("stopping spark context")
  ssc.stop()

trainingData = []
trainingQueue = []
for i in range(NUM_BATCHES):
  bdata = [Vectors.dense(s) for s in batches[i]]
  trainingData.append(bdata)
  dataf = sqlCtx.createDataFrame([((d),) for d in bdata], ["features"])
  trainingQueue.append(dataf.rdd.map(tuple))

eps = 0.1
trainingStream = ssc.queueStream(trainingQueue)
spark_model = StreamingKMeans(k=2, decayFactor=1.0).
  setInitialCenters([[0,0],[eps, eps]], [1.0, 1.0])
spark_model.trainOn(trainingStream)
ssc.start()
ssc.awaitTerminationOrTimeout(3)
spark_centers = spark_model.latestModel().centers
ssc.stop()
```

2 Streaming linear regression

```
from sklearn.datasets import make_blobs
import numpy as np
import pandas as pd
import random
import math
from sklearn.linear_model import LinearRegression
from river import evaluate
from river import linear_model
from river import metrics
from river import preprocessing

WITH_LAST_BATCH = True

NUM_REGULAR_BATCHES = 120
BATCH_SIZE_REGION = 5
BATCH_SIZE = BATCH_SIZE_REGION * 4
L = 6*np.ceil(np.sqrt(NUM_REGULAR_BATCHES))

if WITH_LAST_BATCH:
  NUM_BATCHES = NUM_REGULAR_BATCHES + 1
  NUM_REGULAR_SAMPLES = NUM_REGULAR_BATCHES * BATCH_SIZE_REGION * 4
  NUM_SAMPLES = NUM_REGULAR_SAMPLES + BATCH_SIZE
else:
  NUM_BATCHES = NUM_REGULAR_BATCHES
  NUM_REGULAR_SAMPLES = NUM_REGULAR_BATCHES * BATCH_SIZE_REGION * 4
  NUM_SAMPLES = NUM_REGULAR_SAMPLES


def create_dataset(n_reg_batches, region_batch_size, eps=0.3, l=1, s=0.01, r=L):
  # creates data samples as follows:
  #   a series of regular batches with ponts samples around a constelation of
  #    [-l+eps, l], [l-eps, -l], [-l, l-eps], [l, -l+eps], and a last batch
  #    around [l*r, l*r]. The stddev of sampling is s.
  # returns:
  #   batches - array of dataframes, batch_size long each
  #   dataset - array of ndarrays, each of shape [batch_size, 2]

  n_region_samples = n_reg_batches * region_batch_size
  cU = [[-l+eps-s, l-s], [-l+eps+s, l+s]]
  cD = [[l-eps-s,-l-s], [l-eps+s, -l+s]]
  cL = [[-l-s, l-eps-s], [-l+s,l-eps+s]]
  cR = [[l-s, -l+eps-s], [l+s, -l+eps+s]]
  cX = [[l*r-s, l*r-s], [l*r+s, l*r+s]]
  datasetU, _ = make_blobs(n_samples=n_region_samples, centers=1, center_box=cU,
```

```
                            random_state=42, cluster_std=s)
  datasetD, _ = make_blobs(n_samples=n_region_samples, centers=1, center_box=cD,
                            random_state=42, cluster_std=s)
  datasetL, _ = make_blobs(n_samples=n_region_samples, centers=1, center_box=cL,
                            random_state=42, cluster_std=s)
  datasetR, _ = make_blobs(n_samples=n_region_samples, centers=1, center_box=cR,
                            random_state=42, cluster_std=s)
  datasetX, _ = make_blobs(n_samples=region_batch_size * 4, centers=1,
                            center_box=cX, random_state=42, cluster_std=s)

  dataset = np.concatenate([datasetU, datasetD, datasetL, datasetR])
  np.random.shuffle(dataset)
  if WITH_LAST_BATCH:
    dataset = np.concatenate([dataset, datasetX])

  batches = []
  for i in range(n_reg_batches):
    batch = np.concatenate([datasetU[i*region_batch_size: (i+1)*region_batch_size, :],
                            datasetD[i*region_batch_size: (i+1)*region_batch_size, :],
                            datasetL[i*region_batch_size: (i+1)*region_batch_size, :],
                            datasetR[i*region_batch_size: (i+1)*region_batch_size, :]])
    np.random.shuffle(batch)
    batches.append(pd.DataFrame(data=batch, columns=["X", "Y"]))

  if WITH_LAST_BATCH:
    batches.append(pd.DataFrame(data=datasetX, columns=["X", "Y"]))
  return batches, dataset

batches, all_dataset =  create_dataset(NUM_REGULAR_BATCHES, BATCH_SIZE_REGION)
dataset = np.split(all_dataset, NUM_BATCHES)

class LeverageSampler:
  def __init__(self, scale=10):
    self.first = True
    self.scale = scale

  def Sample(self, batch):
    added = 0
    for b in batch:
      recompute_iv = False

      if self.first:
        self.first = False
        self.M = b
        added += 1
        self.M = np.expand_dims(self.M, axis=0)
```

```
        recompute_iv = True
      else:
        score = np.matmul(b,np.matmul(self.MTM_IV, b.T))
        assert(score > 0)
        probAdd = min(1.0, score*self.scale)
        randomNum = random.uniform(0,1)
        if randomNum <= probAdd:
          added +=1
          self.M = np.vstack([self.M, b/math.sqrt(probAdd)])
          recompute_iv = True
      if recompute_iv:
        self.MTM_IV = np.linalg.pinv(np.matmul(self.M.T, self.M))
    return added

  def Reg(self):
    A = self.M[:,:-1]
    b = self.M[:,-1:]
    return LinearRegression().fit(A,b)

  def MeanSquaredError(self, dbatches, n):
    reg = self.Reg()
    loss = 0
    for i, batch in enumerate(dbatches):
      if i <=n:
        features = batch[:,0]
        actual = batch[:,1]
        preds = reg.predict(batch[:,:-1])
        for i in range(len(preds)):
          loss += (preds[i] - actual[i])**2
    return loss/((n+1)*BATCH_SIZE)


# Leveraged Sampler - Train and Loss:
smp = LeverageSampler()
loss_arr = []

for i, batch in enumerate(dataset):
  added = smp.Sample(batch)
  loss = smp.MeanSquaredError(dataset, i)
  loss_arr.append(loss)
  reg = smp.Reg()
  a_smp = reg.coef_
  b_smp = reg.intercept_

# River Stream  - Train and Loss:
def RiverMeanSquaredError(model, dbatches, n):
```

```
  # Compute loss over dataset first n batches

  a = model['LinearRegression'].weights['X']
  b = model['LinearRegression'].intercept
  loss = 0
  for i, batch in enumerate(dbatches):
    if i <= n:
      features = batch[:,0]
      actual = batch[:,1]
      preds = a*features + b
      for i in range(len(preds)):
        loss += (preds[i] - actual[i])**2
  return loss/((n+1)*BATCH_SIZE)

model = (
    preprocessing.StandardScaler() |
    linear_model.LinearRegression(intercept_lr=.12)
)

river_loss_arr = []
if WITH_LAST_BATCH:
  total_batches = NUM_REGULAR_BATCHES + 1
else:
  total_batches = NUM_REGULAR_BATCHES

for i in range(total_batches):
  batch = batches[i]
  model.learn_many(pd.DataFrame(batch, columns=["X"]), batch["Y"].squeeze())
  loss = RiverMeanSquaredError(model, dataset, i)
  river_loss_arr.append(loss)

a_river = model['LinearRegression'].weights['X']
b_river = model['LinearRegression'].intercept
```

3 Sampling vs. Sketching

```
import numpy as np
import random
import math
from scipy.linalg import null_space
from sklearn.linear_model import LinearRegression

NUMERICAL_SCALING = 1000

# Sketching.

class Sketch:
  def __init__(self, numRows, numCols):
    self.S = []
    for r in range (numRows):
      dummyRow = []
      for c in range (numCols):
        dummyRow.append(0)
      self.S.append(dummyRow)
    for r in range(numRows):
      for c in range(numCols):
        randomReal = random.uniform(0,1)
        if randomReal < 0.5:
          self.S[r][c] = -1
        else:
          self.S[r][c] = 1
    self.S = np.array(self.S)

  def apply(self, A, i, t):
    return np.matmul(self.S[:i, :t], A[:t, :])

  def generateAdvInput(self, d):
    ns = np.array(null_space(self.S))
    #extract first column of ns
    kernelVec=ns[:,0]
    A = []
    colScale = []
    for j in range(d):
      colScale.append(NUMERICAL_SCALING*random.uniform(0,1))

    for i in range(len(kernelVec)):
      rowVec = []
      for j in range(d):
        rowVec.append(kernelVec[i]*colScale[j])
      A.append(rowVec)
```

```
    return np.array(A)

def outputReg(M):
  A = M[:,:-1]
  b = M[:,-1:]
  reg = LinearRegression().fit(A,b)
  return reg

def squaredLoss(reg, M):
  features = M[:,:-1]
  b = M[:,-1:]
  preds = reg.predict(features)
  return np.sum((preds - b)**2) / len(b)

# Leveraged Sampling: Code repeated to make section self contained.

class LeverageSampler:
  def __init__(self, scale=10):
    self.first = True
    self.scale = scale

  def Sample(self, batch):
    added = 0
    for b in batch:
      recompute_iv = False

      if self.first:
        self.first = False
        self.M = b
        added += 1
        self.M = np.expand_dims(self.M, axis=0)
        recompute_iv = True
      else:
        score = np.matmul(b,np.matmul(self.MTM_IV, b.T))
        assert(score > 0)
        probAdd = min(1.0, score*self.scale)
        randomNum = random.uniform(0,1)
        if randomNum <= probAdd:
          added += 1
          self.M = np.vstack([self.M, b/math.sqrt(probAdd)])
          recompute_iv = True
      if recompute_iv:
        self.MTM_IV = np.linalg.pinv(np.matmul(self.M.T, self.M))
    return added

  def Reg(self):
```

```
    A = self.M[:,:-1]
    b = self.M[:,-1:]
    return LinearRegression().fit(A,b)

# Compare methods over data:
TOTAL_SKETCHED_SAMPLES = 50
TOTAL_DATA_SAMPLES = 2000

def simulate_sketching_lr(total_sketched_samples=100,
                          total_data_samples=1000,
                          data_dim=10,
                          batch_size=25,
                          initial_batches=1):
  S = Sketch(total_sketched_samples, total_data_samples)
  A = S.generateAdvInput(data_dim)
  smp_data = np.split(A, int(total_data_samples / batch_size))
  effecive_data_samples = initial_batches*batch_size
  done = False
  sqerr_sketch = []
  sqerr_smp = []
  batch_num = 0
  smp_total = 0

  smp = LeverageSampler()

  while not done:
    if effecive_data_samples >= total_data_samples:
      done = True
      effecive_data_samples = total_data_samples
    effective_sketched_samples = int(effecive_data_samples * (total_sketched_samples / total
    SA_i = S.apply(A, effective_sketched_samples, effecive_data_samples)
    regSA_i = outputReg(SA_i)
    sqerr_sketch.append(squaredLoss(regSA_i, A[:effecive_data_samples, :]))

    smp_batch = smp_data[batch_num]
    smp_added = smp.Sample(smp_batch)
    smp_total += smp_added
    sqerr_smp.append(squaredLoss(smp.Reg(), A[:effecive_data_samples, :]))

    effecive_data_samples += batch_size
    batch_num += 1
  return sqerr_sketch, sqerr_smp, smp_total

errlst_sketch, errlst_smp, smp_total = simulate_sketching_lr(
    total_sketched_samples=TOTAL_SKETCHED_SAMPLES,
    total_data_samples=TOTAL_DATA_SAMPLES,
```

```
data_dim=6,
batch_size=10,
initial_batches=10)
```