use std::collections::BTreeSet;

use std::fmt::{self, Debug, Display};

use std::fs;

use crate::all\_config;

use crate::all\_config::AllConfig;

use anyhow::{bail, Context};

use bpr\_thrift::data::DataRecord;

use bpr\_thrift::prediction\_service::BatchPredictionRequest;

use bpr\_thrift::tensor::GeneralTensor;

use log::debug;

use ndarray::Array2;

use once\_cell::sync::OnceCell;

use ort::tensor::InputTensor;

use prometheus::{HistogramOpts, HistogramVec};

use segdense::mapper::{FeatureMapper, MapReader};

use segdense::segdense\_transform\_spec\_home\_recap\_2022::{DensificationTransformSpec, Root};

use segdense::util;

use thrift::protocol::{TBinaryInputProtocol, TSerializable};

use thrift::transport::TBufferChannel;

pub fn log\_feature\_match(

dr: &DataRecord,

seg\_dense\_config: &DensificationTransformSpec,

dr\_type: String,

) {

// Note the following algorithm matches features from config using linear search.

// Also the record source is MinDataRecord. This includes only binary and continous features for now.

for (feature\_id, feature\_value) in dr.continuous\_features.as\_ref().unwrap() {

debug!(

"{} - Continous Datarecord => Feature ID: {}, Feature value: {}",

dr\_type, feature\_id, feature\_value

);

for input\_feature in &seg\_dense\_config.cont.input\_features {

if input\_feature.feature\_id == \*feature\_id {

debug!("Matching input feature: {:?}", input\_feature)

}

}

}

for feature\_id in dr.binary\_features.as\_ref().unwrap() {

debug!(

"{} - Binary Datarecord => Feature ID: {}",

dr\_type, feature\_id

);

for input\_feature in &seg\_dense\_config.binary.input\_features {

if input\_feature.feature\_id == \*feature\_id {

debug!("Found input feature: {:?}", input\_feature)

}

}

}

}

pub fn log\_feature\_matches(drs: &Vec<DataRecord>, seg\_dense\_config: &DensificationTransformSpec) {

for dr in drs {

log\_feature\_match(dr, seg\_dense\_config, String::from("individual"));

}

}

pub trait Converter: Send + Sync + Debug + 'static + Display {

fn convert(&self, input: Vec<Vec<u8>>) -> (Vec<InputTensor>, Vec<usize>);

}

#[derive(Debug)]

#[allow(dead\_code)]

pub struct BatchPredictionRequestToTorchTensorConverter {

all\_config: AllConfig,

seg\_dense\_config: Root,

all\_config\_path: String,

seg\_dense\_config\_path: String,

feature\_mapper: FeatureMapper,

user\_embedding\_feature\_id: i64,

user\_eng\_embedding\_feature\_id: i64,

author\_embedding\_feature\_id: i64,

discrete\_features\_to\_report: BTreeSet<i64>,

continuous\_features\_to\_report: BTreeSet<i64>,

discrete\_feature\_metrics: &'static HistogramVec,

continuous\_feature\_metrics: &'static HistogramVec,

}

impl Display for BatchPredictionRequestToTorchTensorConverter {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(

f,

"all\_config\_path: {}, seg\_dense\_config\_path:{}",

self.all\_config\_path, self.seg\_dense\_config\_path

)

}

}

impl BatchPredictionRequestToTorchTensorConverter {

pub fn new(

model\_dir: &str,

model\_version: &str,

reporting\_feature\_ids: Vec<(i64, &str)>,

register\_metric\_fn: Option<impl Fn(&HistogramVec)>,

) -> anyhow::Result<BatchPredictionRequestToTorchTensorConverter> {

let all\_config\_path = format!("{}/{}/all\_config.json", model\_dir, model\_version);

let seg\_dense\_config\_path = format!(

"{}/{}/segdense\_transform\_spec\_home\_recap\_2022.json",

model\_dir, model\_version

);

let seg\_dense\_config = util::load\_config(&seg\_dense\_config\_path)?;

let all\_config = all\_config::parse(

&fs::read\_to\_string(&all\_config\_path)

.with\_context(|| "error loading all\_config.json - ")?,

)?;

let feature\_mapper = util::load\_from\_parsed\_config(seg\_dense\_config.clone())?;

let user\_embedding\_feature\_id = Self::get\_feature\_id(

&all\_config

.train\_data

.seg\_dense\_schema

.renamed\_features

.user\_embedding,

&seg\_dense\_config,

);

let user\_eng\_embedding\_feature\_id = Self::get\_feature\_id(

&all\_config

.train\_data

.seg\_dense\_schema

.renamed\_features

.user\_eng\_embedding,

&seg\_dense\_config,

);

let author\_embedding\_feature\_id = Self::get\_feature\_id(

&all\_config

.train\_data

.seg\_dense\_schema

.renamed\_features

.author\_embedding,

&seg\_dense\_config,

);

static METRICS: OnceCell<(HistogramVec, HistogramVec)> = OnceCell::new();

let (discrete\_feature\_metrics, continuous\_feature\_metrics) = METRICS.get\_or\_init(|| {

let discrete = HistogramVec::new(

HistogramOpts::new(":navi:feature\_id:discrete", "Discrete Feature ID values")

.buckets(Vec::from(&[

0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0, 110.0,

120.0, 130.0, 140.0, 150.0, 160.0, 170.0, 180.0, 190.0, 200.0, 250.0,

300.0, 500.0, 1000.0, 10000.0, 100000.0,

] as &'static [f64])),

&["feature\_id"],

)

.expect("metric cannot be created");

let continuous = HistogramVec::new(

HistogramOpts::new(

":navi:feature\_id:continuous",

"continuous Feature ID values",

)

.buckets(Vec::from(&[

0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0, 110.0, 120.0,

130.0, 140.0, 150.0, 160.0, 170.0, 180.0, 190.0, 200.0, 250.0, 300.0, 500.0,

1000.0, 10000.0, 100000.0,

] as &'static [f64])),

&["feature\_id"],

)

.expect("metric cannot be created");

register\_metric\_fn.map(|r| {

r(&discrete);

r(&continuous);

});

(discrete, continuous)

});

let mut discrete\_features\_to\_report = BTreeSet::new();

let mut continuous\_features\_to\_report = BTreeSet::new();

for (feature\_id, feature\_type) in reporting\_feature\_ids.iter() {

match \*feature\_type {

"discrete" => discrete\_features\_to\_report.insert(feature\_id.clone()),

"continuous" => continuous\_features\_to\_report.insert(feature\_id.clone()),

\_ => bail!(

"Invalid feature type {} for reporting metrics!",

feature\_type

),

};

}

Ok(BatchPredictionRequestToTorchTensorConverter {

all\_config,

seg\_dense\_config,

all\_config\_path,

seg\_dense\_config\_path,

feature\_mapper,

user\_embedding\_feature\_id,

user\_eng\_embedding\_feature\_id,

author\_embedding\_feature\_id,

discrete\_features\_to\_report,

continuous\_features\_to\_report,

discrete\_feature\_metrics,

continuous\_feature\_metrics,

})

}

fn get\_feature\_id(feature\_name: &str, seg\_dense\_config: &Root) -> i64 {

// given a feature name, we get the complex feature type id

for feature in &seg\_dense\_config.complex\_feature\_type\_transform\_spec {

if feature.full\_feature\_name == feature\_name {

return feature.feature\_id;

}

}

-1

}

fn parse\_batch\_prediction\_request(bytes: Vec<u8>) -> BatchPredictionRequest {

// parse batch prediction request into a struct from byte array repr.

let mut bc = TBufferChannel::with\_capacity(bytes.len(), 0);

bc.set\_readable\_bytes(&bytes);

let mut protocol = TBinaryInputProtocol::new(bc, true);

BatchPredictionRequest::read\_from\_in\_protocol(&mut protocol).unwrap()

}

fn get\_embedding\_tensors(

&self,

bprs: &[BatchPredictionRequest],

feature\_id: i64,

batch\_size: &[usize],

) -> Array2<f32> {

// given an embedding feature id, extract the float tensor array into tensors.

let cols: usize = 200;

let rows: usize = batch\_size[batch\_size.len() - 1];

let total\_size = rows \* cols;

let mut working\_set = vec![0 as f32; total\_size];

let mut bpr\_start = 0;

for (bpr, &bpr\_end) in bprs.iter().zip(batch\_size) {

if bpr.common\_features.is\_some() {

if bpr.common\_features.as\_ref().unwrap().tensors.is\_some() {

if bpr

.common\_features

.as\_ref()

.unwrap()

.tensors

.as\_ref()

.unwrap()

.contains\_key(&feature\_id)

{

let source\_tensor = bpr

.common\_features

.as\_ref()

.unwrap()

.tensors

.as\_ref()

.unwrap()

.get(&feature\_id)

.unwrap();

let tensor = match source\_tensor {

GeneralTensor::FloatTensor(float\_tensor) =>

//Tensor::of\_slice(

{

float\_tensor

.floats

.iter()

.map(|x| x.into\_inner() as f32)

.collect::<Vec<\_>>()

}

\_ => vec![0 as f32; cols],

};

// since the tensor is found in common feature, add it in all batches

for row in bpr\_start..bpr\_end {

for col in 0..cols {

working\_set[row \* cols + col] = tensor[col];

}

}

}

}

}

// find the feature in individual feature list and add to corresponding batch.

for (index, datarecord) in bpr.individual\_features\_list.iter().enumerate() {

if datarecord.tensors.is\_some()

&& datarecord

.tensors

.as\_ref()

.unwrap()

.contains\_key(&feature\_id)

{

let source\_tensor = datarecord

.tensors

.as\_ref()

.unwrap()

.get(&feature\_id)

.unwrap();

let tensor = match source\_tensor {

GeneralTensor::FloatTensor(float\_tensor) => float\_tensor

.floats

.iter()

.map(|x| x.into\_inner() as f32)

.collect::<Vec<\_>>(),

\_ => vec![0 as f32; cols],

};

for col in 0..cols {

working\_set[(bpr\_start + index) \* cols + col] = tensor[col];

}

}

}

bpr\_start = bpr\_end;

}

Array2::<f32>::from\_shape\_vec([rows, cols], working\_set).unwrap()

}

// Todo : Refactor, create a generic version with different type and field accessors

// Example paramterize and then instiantiate the following

// (FLOAT --> FLOAT, DataRecord.continuous\_feature)

// (BOOL --> INT64, DataRecord.binary\_feature)

// (INT64 --> INT64, DataRecord.discrete\_feature)

fn get\_continuous(&self, bprs: &[BatchPredictionRequest], batch\_ends: &[usize]) -> InputTensor {

// These need to be part of model schema

let rows: usize = batch\_ends[batch\_ends.len() - 1];

let cols: usize = 5293;

let full\_size: usize = rows \* cols;

let default\_val = f32::NAN;

let mut tensor = vec![default\_val; full\_size];

let mut bpr\_start = 0;

for (bpr, &bpr\_end) in bprs.iter().zip(batch\_ends) {

// Common features

if bpr.common\_features.is\_some()

&& bpr

.common\_features

.as\_ref()

.unwrap()

.continuous\_features

.is\_some()

{

let common\_features = bpr

.common\_features

.as\_ref()

.unwrap()

.continuous\_features

.as\_ref()

.unwrap();

for feature in common\_features {

match self.feature\_mapper.get(feature.0) {

Some(f\_info) => {

let idx = f\_info.index\_within\_tensor as usize;

if idx < cols {

// Set value in each row

for r in bpr\_start..bpr\_end {

let flat\_index: usize = r \* cols + idx;

tensor[flat\_index] = feature.1.into\_inner() as f32;

}

}

}

None => (),

}

if self.continuous\_features\_to\_report.contains(feature.0) {

self.continuous\_feature\_metrics

.with\_label\_values(&[feature.0.to\_string().as\_str()])

.observe(feature.1.into\_inner())

} else if self.discrete\_features\_to\_report.contains(feature.0) {

self.discrete\_feature\_metrics

.with\_label\_values(&[feature.0.to\_string().as\_str()])

.observe(feature.1.into\_inner())

}

}

}

// Process the batch of datarecords

for r in bpr\_start..bpr\_end {

let dr: &DataRecord =

&bpr.individual\_features\_list[usize::try\_from(r - bpr\_start).unwrap()];

if dr.continuous\_features.is\_some() {

for feature in dr.continuous\_features.as\_ref().unwrap() {

match self.feature\_mapper.get(&feature.0) {

Some(f\_info) => {

let idx = f\_info.index\_within\_tensor as usize;

let flat\_index: usize = r \* cols + idx;

if flat\_index < tensor.len() && idx < cols {

tensor[flat\_index] = feature.1.into\_inner() as f32;

}

}

None => (),

}

if self.continuous\_features\_to\_report.contains(feature.0) {

self.continuous\_feature\_metrics

.with\_label\_values(&[feature.0.to\_string().as\_str()])

.observe(feature.1.into\_inner() as f64)

} else if self.discrete\_features\_to\_report.contains(feature.0) {

self.discrete\_feature\_metrics

.with\_label\_values(&[feature.0.to\_string().as\_str()])

.observe(feature.1.into\_inner() as f64)

}

}

}

}

bpr\_start = bpr\_end;

}

InputTensor::FloatTensor(

Array2::<f32>::from\_shape\_vec([rows, cols], tensor)

.unwrap()

.into\_dyn(),

)

}

fn get\_binary(&self, bprs: &[BatchPredictionRequest], batch\_ends: &[usize]) -> InputTensor {

// These need to be part of model schema

let rows: usize = batch\_ends[batch\_ends.len() - 1];

let cols: usize = 149;

let full\_size: usize = rows \* cols;

let default\_val: i64 = 0;

let mut v = vec![default\_val; full\_size];

let mut bpr\_start = 0;

for (bpr, &bpr\_end) in bprs.iter().zip(batch\_ends) {

// Common features

if bpr.common\_features.is\_some()

&& bpr

.common\_features

.as\_ref()

.unwrap()

.binary\_features

.is\_some()

{

let common\_features = bpr

.common\_features

.as\_ref()

.unwrap()

.binary\_features

.as\_ref()

.unwrap();

for feature in common\_features {

match self.feature\_mapper.get(feature) {

Some(f\_info) => {

let idx = f\_info.index\_within\_tensor as usize;

if idx < cols {

// Set value in each row

for r in bpr\_start..bpr\_end {

let flat\_index: usize = r \* cols + idx;

v[flat\_index] = 1;

}

}

}

None => (),

}

}

}

// Process the batch of datarecords

for r in bpr\_start..bpr\_end {

let dr: &DataRecord = &bpr.individual\_features\_list[r - bpr\_start];

if dr.binary\_features.is\_some() {

for feature in dr.binary\_features.as\_ref().unwrap() {

match self.feature\_mapper.get(&feature) {

Some(f\_info) => {

let idx = f\_info.index\_within\_tensor as usize;

let flat\_index: usize = r \* cols + idx;

v[flat\_index] = 1;

}

None => (),

}

}

}

}

bpr\_start = bpr\_end;

}

InputTensor::Int64Tensor(

Array2::<i64>::from\_shape\_vec([rows, cols], v)

.unwrap()

.into\_dyn(),

)

}

#[allow(dead\_code)]

fn get\_discrete(&self, bprs: &[BatchPredictionRequest], batch\_ends: &[usize]) -> InputTensor {

// These need to be part of model schema

let rows: usize = batch\_ends[batch\_ends.len() - 1];

let cols: usize = 320;

let full\_size: usize = rows \* cols;

let default\_val: i64 = 0;

let mut v = vec![default\_val; full\_size];

let mut bpr\_start = 0;

for (bpr, &bpr\_end) in bprs.iter().zip(batch\_ends) {

// Common features

if bpr.common\_features.is\_some()

&& bpr

.common\_features

.as\_ref()

.unwrap()

.discrete\_features

.is\_some()

{

let common\_features = bpr

.common\_features

.as\_ref()

.unwrap()

.discrete\_features

.as\_ref()

.unwrap();

for feature in common\_features {

match self.feature\_mapper.get(feature.0) {

Some(f\_info) => {

let idx = f\_info.index\_within\_tensor as usize;

if idx < cols {

// Set value in each row

for r in bpr\_start..bpr\_end {

let flat\_index: usize = r \* cols + idx;

v[flat\_index] = \*feature.1;

}

}

}

None => (),

}

if self.discrete\_features\_to\_report.contains(feature.0) {

self.discrete\_feature\_metrics

.with\_label\_values(&[feature.0.to\_string().as\_str()])

.observe(\*feature.1 as f64)

}

}

}

// Process the batch of datarecords

for r in bpr\_start..bpr\_end {

let dr: &DataRecord = &bpr.individual\_features\_list[usize::try\_from(r).unwrap()];

if dr.discrete\_features.is\_some() {

for feature in dr.discrete\_features.as\_ref().unwrap() {

match self.feature\_mapper.get(&feature.0) {

Some(f\_info) => {

let idx = f\_info.index\_within\_tensor as usize;

let flat\_index: usize = r \* cols + idx;

if flat\_index < v.len() && idx < cols {

v[flat\_index] = \*feature.1;

}

}

None => (),

}

if self.discrete\_features\_to\_report.contains(feature.0) {

self.discrete\_feature\_metrics

.with\_label\_values(&[feature.0.to\_string().as\_str()])

.observe(\*feature.1 as f64)

}

}

}

}

bpr\_start = bpr\_end;

}

InputTensor::Int64Tensor(

Array2::<i64>::from\_shape\_vec([rows, cols], v)

.unwrap()

.into\_dyn(),

)

}

fn get\_user\_embedding(

&self,

bprs: &[BatchPredictionRequest],

batch\_ends: &[usize],

) -> InputTensor {

InputTensor::FloatTensor(

self.get\_embedding\_tensors(bprs, self.user\_embedding\_feature\_id, batch\_ends)

.into\_dyn(),

)

}

fn get\_eng\_embedding(

&self,

bpr: &[BatchPredictionRequest],

batch\_ends: &[usize],

) -> InputTensor {

InputTensor::FloatTensor(

self.get\_embedding\_tensors(bpr, self.user\_eng\_embedding\_feature\_id, batch\_ends)

.into\_dyn(),

)

}

fn get\_author\_embedding(

&self,

bpr: &[BatchPredictionRequest],

batch\_ends: &[usize],

) -> InputTensor {

InputTensor::FloatTensor(

self.get\_embedding\_tensors(bpr, self.author\_embedding\_feature\_id, batch\_ends)

.into\_dyn(),

)

}

}

impl Converter for BatchPredictionRequestToTorchTensorConverter {

fn convert(&self, batched\_bytes: Vec<Vec<u8>>) -> (Vec<InputTensor>, Vec<usize>) {

let bprs = batched\_bytes

.into\_iter()

.map(|bytes| {

BatchPredictionRequestToTorchTensorConverter::parse\_batch\_prediction\_request(bytes)

})

.collect::<Vec<\_>>();

let batch\_ends = bprs

.iter()

.map(|bpr| bpr.individual\_features\_list.len())

.scan(0usize, |acc, e| {

//running total

\*acc = \*acc + e;

Some(\*acc)

})

.collect::<Vec<\_>>();

let t1 = self.get\_continuous(&bprs, &batch\_ends);

let t2 = self.get\_binary(&bprs, &batch\_ends);

//let \_t3 = self.get\_discrete(&bprs, &batch\_ends);

let t4 = self.get\_user\_embedding(&bprs, &batch\_ends);

let t5 = self.get\_eng\_embedding(&bprs, &batch\_ends);

let t6 = self.get\_author\_embedding(&bprs, &batch\_ends);

(vec![t1, t2, t4, t5, t6], batch\_ends)

}

}