// A feature value can be one of these

enum FeatureVal {

Empty,

U8Vector(Vec<u8>),

FloatVector(Vec<f32>),

}

// A Feture has a name and a value

// The name for now is 'id' of type string

// Eventually this needs to be flexible - example to accomodate feature-id

struct Feature {

id: String,

val: FeatureVal,

}

impl Feature {

fn new() -> Feature {

Feature {

id: String::new(),

val: FeatureVal::Empty

}

}

}

// A single inference record will have multiple features

struct Record {

fields: Vec<Feature>,

}

impl Record {

fn new() -> Record {

Record { fields: vec![] }

}

}

// This is the main API used by external components

// Given a serialized input, decode it into Records

fn decode(input: Vec<u8>) -> Vec<Record> {

// For helping define the interface

vec![get\_random\_record(), get\_random\_record()]

}

// Used for testing the API, will be eventually removed

fn get\_random\_record() -> Record {

let mut record: Record = Record::new();

let f1: Feature = Feature {

id: String::from("continuous\_features"),

val: FeatureVal::FloatVector(vec![1.0f32; 2134]),

};

record.fields.push(f1);

let f2: Feature = Feature {

id: String::from("user\_embedding"),

val: FeatureVal::FloatVector(vec![2.0f32; 200]),

};

record.fields.push(f2);

let f3: Feature = Feature {

id: String::from("author\_embedding"),

val: FeatureVal::FloatVector(vec![3.0f32; 200]),

};

record.fields.push(f3);

let f4: Feature = Feature {

id: String::from("binary\_features"),

val: FeatureVal::U8Vector(vec![4u8; 43]),

};

record.fields.push(f4);

record

}