# (peek @Clojure)

## Agenda

In part 1 we will be covering a very basic and well known Clojure environment setup, have a very high-perspective overview over the Clojure language (key points & philosophy) and a quick warm-up/ramp-up on the core basics.

In part 2 we'll be going over the Classification problem and a very basic classification algorithm (Nearest Neighbor Classifier) and use Clojure's opinionated ways to implement it.

In the last part we explore a small but efficient library – Compojure + Ring – to expose our classifier as a basic web service. If time permits we'll explore migrating to a microservice architecture.

# Part 1

### Setup

This involves having emacs + CIDER + leiningen.

Follow the setup steps at https://github.com/alex-gherega/clojure-env-setup.

After all the steps make use Leiningen to make a new Clojure project:

**~> lein new *project_name***

Navigate inside project_name directory and launch emacs. Start the REPL with C-F11 (Ctrl + F11). You should see a REPL window where you can start type in Clojure code and get it evaluated on the spot.

### Clojure overview

Clojure is a language designed by Rich Hickey in 2007. First stable release was in 2009.

Rich Hickey is a fan of CLisp, has tons of experience with Java technologies, Lisp, FP, Database systems and many more.

He began the Cognitect startup where Clojure is continuously forged together with other cool libs and projects (Datomic, spec, Arachne, etc.)

Reason of Clojure:

> wanted a Lisp with concurency and symbiotic with a powerful platform.

Clojure's philosphy:

> be expressive, simple and always practical.

Clojure's key aspects:

keeps best features of lisp (homoiconic, clean sintax, nil-punning and many more).

extends the first class citizenship to other datastructures as well

works with an existing platform (it is not one!)

very practical support for concurrency

not backward compatible

it always works it hardest to extract abstractions (callability, sequences, reducers, transducers)

it keeps a very close eye on performance

it provides a dynamic environment

Clojure buzzwords to burn into your memory:

**CORE:** REPL, macros, datatypes, evaluation, transients, transducers, reducers, multimethods +hierarchi, protocols, concurency + STM ...

**LIBS:** core.async, spec, Luminus/Pedestal/Arachne, Datomic, clj-time, kibit, clojure-android, neko ….

Have a look at these: clojure.org, clojure-toolbox.com, clojuredocs.org, cognitect.com, dev.clojure.org

## Clojure ramp-up/warm-up

Syntax: Always prefix notation.

When you see **(some-name …)** this means some-name points to a function, whatever is after **some-name** in the brackets represent the arguments with which the above function will be called.

REPL:

Read Print Eval Loop – says it all doesn't it? It's similar to every other REPL you've encountered except for the particularities of lisp + JVM (the reader is similar to a Lisp-1 reader) and the code is compiled dynamically to bytecode and run on the JVM.

This together with the code-as-data “inheritance” from Lisp means that you get tremendous power – Clojure code is something you can change and interact with.

The reader:

You get 3 main syntactic constructs atomic literals (42, \c, “a”), s-expressions these are symbolic expressions which have synactic meaning, and forms which are valid s-

expressions (i.e. expressions that evaluate to a single value).

Whenever this is not the case we deal with something called a special-form (e.g. if, let, def, quote,loop, recur, fn, do, var, throw, try, .(dot) , new, set!)

Everything else is a function call. You get nice syntactic-sugar for various constructs so you get more succinct code. These are called reader-macros (e.g. {“k1” 42 “k2” 24} is the same as calling the hash-map function with “k1” 42 “k2” 24 arguments).

Data-structures:

Atomic literals:

- numbers are Java numbers (Long, BigDecimal, Double) and ratio;
  - these all get autoboxed but not arbitrary precision on all operations
- strings are Java Strings
- characters (you got it) Java Characters
- true/false/nil

Collections:

- sets, maps, vectors and lists
- they are part of the code-as-data (they do not get converted to lists)
- Clojure integrates them as part of the language through the Sequence abstraction

Lambdas:

you create an anonymous function using fn

Vars:

use the special form def to associate a symbol and a Var

def+fn => defn your most used macro!

Another abstraction: Callability (maps, sets, vectors are all also functions).

The same abstraction is used for symbols and keywords:

→ symbols are names (this is different from Lisp where symbols are practically Vars); when used as functions over maps the look themselves up as the key

→ keywords are practically things that evaluate to themselves; they require a “:” before the name (can be almost anything); again the implement the ICallable interface so the act as functions over maps as well

Immutability:

Clojure keeps almost everything immutable (but it does so with performance in mind; so for collections you don't get full copies – you get persistent datastructures with structural sharing).

But you do get mutable storage via Vars, Refs, Agents and Atoms.

Clojure made a big deal out of state vs. identity. Identity is the name you associate something with (e.g. the word Mum, or Sunday; these can and should change – my Mum is not your Mum). Each identity has state when you take a snapshot in time of the world. That state is immutable – e.g. number 42 does not change.

Recursive calls:

Because JVM has no tail optimisation => Clojure doesn's support tail call optimisation just by having a mediocre recursive call.

Hence we get loop + recur. Loop sets some local bindings and place where recur can link back to. recur is basically your recursive call (it needs to be in the tail to have hight performance; gives you an error) with new bindings for either loop or the function head.

Truthiness:

You get true and false from the Java land but you also get nil (null). nil means nothing.

Everything that is not nil or false is true.

# Part 2

### Lein explained

Leiningen is a project automation tool that let's you get going fast and in a standard way.

Lein defines a bunch of tasks (you can write your own in a lein plugin). It uses Clojure code so it is all familiar.

The usual syntax:

lein *task-name some-arguments*

e.g. lein new playground produces a default project called playground.

In it you'll find a very specific structure of files and directories.

The most important ones:

- project.clj – this is your project configuration if you will
- src/  - directory where your *.clj files are stored

- test/ - directory where your tests reside
- resources/  - for any static content you need or libs
- target/ - where your output/packed project should go.

To find out more: https://github.com/technomancy/leiningen/blob/master/sample.project.clj and https://leiningen.org/.

## Classification

Classification is a pattern matching paradigm to solve out problems like: give *this-input* associate a *label* to it from a given *set of labels*.

Usually it involves processing/modelling the input space to some sort of vector field.

Pre-processing of data means clean-up and formatting.

The set of labels is either discovered either proposed by an expert.

The task of the classifier algorithm: train on an training input set; learn a specific model over this data (there are sweet spots – too much learning leads to overfitting; too little lead to weak models – fewer degrees of freedom).

## Nearest Centroid Classifier

Training phase:

Given an input data set with data already classified: $\{(\vec{x}_1, C_1)...(\vec{x}_N, C_M)\}$ and the set of classes $C_L = \{C_1 ... C_M\}$ we then compute a centroid vector for each class:

$\mu_{C_i} = \frac{1}{C_i} \sum \vec{x}_i$ for all xi vectors in class Ci.

Prediction:

Given a new input data vector $\vec{x}$ the class estimation is given by:

$\hat{C} = argmin_{Ci} \left\| \vec{x} - \mu_{C_i} \right\|$

In other not so fancy math words (: get a mean vector from all input data for a class and then compare the new input data against a distance; the lowest one wins.

### Implement with clojure

Follow the github repos but try and workit out yourself. Don't give into temptation to look at the and of the book :)

https://github.com/alex-gherega/peek-at-clojure

Use the playground project to work out some of the cool semantics of Clojure.

The nearest-centroid-classifier has several implementations (from the trivial to the more complex/Clojure like – using Clojure idioms).

The ncc-server is for the next part.

# Part 3

### Ring

a Clojure web applications library inspired by Python's WSGI and Ruby's Rack

it allows you to construct web applications by combining modular components

these components can then be shared among a variety of applications, web servers, and web frameworks

It has a core set of libraries:

ring-core – the essential set of functions that allow you to handle requests, responses, parameters, cookies, security etc.

ring-devel – this is the place to drop any functionality for development and debug of Ring applications

ring-servlet – starting of with a Ring handler (implemented using ring-core) you can construct Java servlets

ring-jetty-adapter – basically this is where you interconnect with/over a web server (in this case Jetty – but more addapters exist).

After discussing Compojure we'll see how we use Ring by navigating through some lib spaces.

### Compojure

A light routing library for Ring.

It supports a lein plugin so you can create a new Compojure empowered project by calling:

**~> lein** new compojure *name_of_your_project*

(where *name_of_your_project* ncc-server in this case).

Looking inside project.clj you see the ring-plugin and the ring task.

The starting point in your project is the src/handler.clj file. Let's define a route!

Routes explained: you get all the methods macros followed by the route's path, parameters and body. (You can pass an optional string).

To start the server and enjoy your results:

**~> lein** ring server-headless [port]

### The NCC service

What we need to do is make the nearest-centroid-classifier project visible/available to the above compojure project, make a route so we can call the **predict** function with a specific set of arguments.

We initially return a simple string.

If time allows let's try and use hiccup or json to return eitehr an HTML or a JSON to the browser.

## Sum-up

We went throught the Clojure key aspects, got some code under our belts, and cheked out how to work with a useful library.

“We should be solving problems. We should not be building features.” - Rich Hickey [https://github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/HammockDrivenDev.md]