Extracting Phrases with Chunking

Extracting Phrases with Chunking

This is a simple chunker based on an averaged perceptron. Here, accuracy is not key since we have a method that's tolerable to noise. And furthermore, a chunk in and of itself is completely useless (at least for my purposes). The chunks here are used to generate

1. Possible phrases of interest and

2. Factoring sentences into noun phrase verb phrase noun phrase.

The accuracy of 2) is not paramount, it simply needs to be good enough to feed a later hypothesis generation phase. A human will then be tasked to filter the outputs. If the signal is above the noise then that's all that matters.

```
let formChunks chunks =
  let _, final_chunk, built_chunks =
   chunks |> Array.fold (fun ((lasttag, curphrase, phrases) as state) (w, pos) ->
         match splitstrWith "-" pos, curphrase with
          | [|_; chunktag|], _  ->
            if lasttag = chunktag then chunktag, (w,chunktag)::curphrase, phrases
            else chunktag, [w,chunktag], curphrase::phrases
          | _ , [] -> state
          | _ , c -> "", [], c::phrases) ("", [],[])
  final_chunk :: built_chunks |> List.rev |> List.toArray
```

The above method works to simply join a series of matching tags (for tagged words) together. E.g. "NP"; "NP"; "VP" => "NP" "NP"; "VP"... etc.

```
let formChunksAny chunks =
  let _, final_chunk, built_chunks =
   chunks |> Array.fold (fun ((lasttag, curphrase, phrases) as state) (w, chunktag) ->
            if lasttag = chunktag then chunktag, (w,chunktag)::curphrase, phrases
            else chunktag, [w,chunktag], curphrase::phrases) ("", [],[])

  final_chunk :: built_chunks |> List.rev |> List.toArray
```

This method is identical to the last but further removes the restriction that the word must be tagged.

Joining Chunks.

```
let joinchunksToStringGen f phraseList =
   phraseList |> Array.mapFilter (List.rev >> f >> joinTokens) (String.length >> (<) 1)
```

Joins chunks to string, dropping tags and any word whose len is < 1.
Sample Output:

```
[|[|"Henry"; "was"; "an early king"; "of"; "France"|];
  [|"the boy"; "will throw"; "the ball"|]...

```

joinchunksToStringTags keeps tags.

```
///phrase list as input
let joinChunksToString phraseList = joinchunksToStringGen (List.map fst) phraseList

///phrase list as input
let joinChunksToStringTags phraseList = joinchunksToStringGen id phraseList
```

Sample Output:

```
[|[|"(Henry, NP)"; "(was, VP)"; "(an, NP) (early, NP) (king, NP)";
  "(of, PP)"; "(France, NP)"|];
  [|"(the, NP) (boy, NP)"; "(will, VP) (throw, VP)"; "(the, NP) (ball, NP)"|];...
```

```
let joinChunksToSummaryGen innermap phraseList =
   phraseList |> Array.filterMap (function [_, "NP"] | [_, "VP"] -> true | [] | [_,_] -> fal
                                 (List.rev >> innermap)

let joinChunksToSummary joinChar phrases = joinChunksToSummaryGen (List.map fst >> joinToken
```

Example usage:

```
let chunks = sentWords |> Array.Parallel.mapi (fun i s ->
      let t = predictlabelChunker tagtableChunk avec_chunk pos.[i] s
      Array.zip s t)

let gs = chunks |> Array.map getFocusPhrases
let qq = gs.[3] |> Array.filter (List.isEmpty >> not)
let qs = gs |> Array.filter (fun x -> x |> Array.filter (List.isEmpty >> not) |> Array.lengt
qs.[2]
let groupedChunks = chunks |> Array.map formChunks

let sections =  groupedChunks |> Array.map (joinChunksToSummary " ")

let allchunks = groupedChunks
                |> Array.concat
                |> Array.filterMap
```

```
                     (List.isEmpty >> not)
                     (fun phraseList ->
                          phraseList
                          |> List.rev
                          |> List.map fst
                          |> joinTokens,
                               snd phraseList.Head) // |> Seq.g


let chunk_tagmap = Map allchunks

let chunk_tag = allchunks |> Seq.groupBy snd |> Seq.mapGroupByWith (Set.ofSeq >> Set.toArray
let chunk_map = Map chunk_tag

chunk_tag |> Array.map (keepLeft (Array.sortByDescending String.length))

chunk_map.["ADJP"] |> Array.sortByDescending (String.length)

sections |> Array.map (breakParagraphs splitwSpace 3 9 "\n" >> snd >> trim)

let paras = sections |> set |> Set.toArray
```

A more lenient grouping of chunks. Allows:

- verbs after nouns
- nouns after verbs.
- adverbs after verbs
- adjp after verb

```
let groupChunkFlexible allowadverb_verb chunk =
   let _, last_chunk, built_chunks =
      chunk |> Array.fold (fun ((lasttag, curphrase, phrases) as state) (w, pos) ->
            match splitstrWith "-" pos, curphrase with
             | [|_; chunktag|], _  ->
               if lasttag = chunktag || (chunktag = "VP" && lasttag = "NP") || (chunktag = "
                  || (allowadverb_verb && (chunktag = "ADJP" || chunktag = "ADVP") && (lastt
               then chunktag, (w,chunktag)::curphrase, phrases
               else chunktag, [w,chunktag], curphrase::phrases
             | _ , [] -> state
             | _ , c -> "", [], c::phrases) ("", [],[])
   last_chunk::built_chunks |> List.rev |> List.toArray


let chunks0 = chunks |> Array.map (groupChunkFlexible true >> joinChunksToSummary ", ")

```

```
Array.zip (chunks0 |> Array.map (Seq.filter ((<>) ' ') >> Seq.length)) (sections |> Array.ma

let chunks9 = chunks |> Array.map (groupChunkFlexible true
                                    >> joinChunksToSummaryGen (fun phrases ->
                                            phrases |> List.map fst
                                                    |>  joinTokens,
                                            Seq.mode (phrases |> List.map snd)))

let paras0 = chunks9 |> Array.concat |> set |> Set.toArray
```