

Using Neural Networks for Document Classification

Grant Guenter*, Geordie Parappilly†, Weijun Qiu‡, Andy Soobrian§ and Fabian Volz¶

*57336133 t1b9 grantguenter@gmail.com

†33948134 n1u9a geordie.p@hotmail.com

‡42701136 d8x8 weijun.q@gmail.com

§51982122 o3j8 andys12358@gmail.com

¶73138159 u2p0b f.volz@live.de

INTRODUCTION

We have implemented an artificial neural network in Erlang, trained it, and then applied it to the document classification problem, that is the problem of determining its author upon input of a text passage.

The end product allows users to, given a text document, determine if the style of the document matches any of the authors included in the dataset that the algorithm has been trained on. Given sufficient examples of an author's work, it should also be possible to train the system to recognize new authors who have styles that are sufficiently distinct from those that have already been input. Since training a neural network often requires processing large data sets, we have strived for a concurrent implementation which makes use of all the available processing power.

The value of this project is manifold. It demonstrates, through the application to machine learning, which has gained much popularity recently, that concurrency is a topic of extreme relevance these days. It shows that Erlang features good built-in support for concurrent programming, making it a potentially useful language. The project also allows us to display our ability to rapidly learn an unknown programming language and exploit its benefits to write a substantial program.

This document is a detailed description of what we have accomplished and how we did it. It is self-contained and partially repeats what has already been stated in the proof-of-concept and plan. However, in order to understand the document thoroughly, it is highly recommended to be familiar with the background research report.

The first section gives an overview of the features we have been able to implement as well as of those that are missing for the complete set of goals. In sections II–VI we then present and explain most parts of the code; the entire code can be found in [1]. The more generic and less interesting helper functions are defined in Section VII. Section VIII finally gives a small example of how the neural network code can be used.

I. THE FINAL PROJECT

We have made significant achievements in trying to complete the final project and we have been able to implement all the core functionality which is required to classify documents. More precisely, we have been able to implement the following:

- *Document acquisition:* Having downloaded a copy of Project Gutenberg's RDF catalog, the user can use

the function `find_author_works` to obtain a list of integers that correspond to the books written by a specific author. This list can then be used to call `process_file_numbers` to download the documents. See Section II.

- *Document preprocessing:* After downloading the documents, the user can repeatedly call `process_file` to extract the real content, removing prefaces etc. The text of each chapter is written into a file. See Section III.
- *Text parsing:* The user can use the function `chapter_stats` on a chapter text to compute its statistics. What is returned by this function can be used as an input for an artificial neural network, that is to say it returns the X value of a training sample (X, Y) . The Y value should be the j -th standard unit vector in \mathbb{R}^r , if this chapter was written by the j -th author and the user wants to train the network on r authors. See Section IV.
- *Neural network training:* Once the user has all training samples, he can use the function `training_session_parallelism` to get a trained neural network. He might want to customize this function to get better results. See Section V.
- *Classification:* The user can compute the statistics of an unclassified chapter and then use the previously obtained network and the `feed_forward` function to try to determine its author. See Section VI.

As can be seen, our project provides very powerful tools for document classification and we successfully accomplished the low-risk goals. Unfortunately, the following features of the full set of goals are still missing:

- *Node parallelism:* We have only been able to implement the training session parallelism strategy for the backpropagation algorithm. The node parallelism strategy would require to spawn an Erlang process for every node in the network. Each node would have to store a list of the process identifiers for the nodes of the previous layer and—if also the feedforward procedure should be parallelized—the subsequent layer. This approach makes it necessary to send and receive many messages between the processes (compared to the actual computations), thus it might not be very efficient. Still it would have been interesting to see how a more

complicated architecture is facilitated by Erlang's concurrency features.

- *All-automatic network training:* In theory, once the user has a copy of the RDF catalog, the network could be trained from a list of author names alone. This would only require to correctly put the aforementioned, provided functions together. Instead of going through the entire process and manually keeping track of the training samples, the user would only have to call two functions, one for document acquisition, preprocessing, parsing and training, and one for classifying.

II. DOCUMENT ACQUISITION

We source our documents from Project Gutenberg, an online collection of books which are in the public domain. Unfortunately, the main website of Project Gutenberg does not allow automated access. In order to search through the collection or to download files, it is necessary to obtain a local copy of the RDF catalog provided here: <http://www.gutenberg.org/wiki/Gutenberg:Feeds>. Because this catalog is unsorted, we use linear search to read each RDF file and determine if it matches the desired author, using the `find_author_works` function of the `rdf_processor` module.

```

1 -module(rdf_processor).
2 -export([find_author_works/1]).
3
4 % Input: String AuthorName, in format "Last Name, First
   name" ex. "Dickens, Charles"
5 % Output: list of Natural, each element in the list
   corresponds to a file number of one of the author's
   works.
6 find_author_works(AuthorName)
7   -> find_author(AuthorName, 1, []).
8
9 find_author(_, 50543, Works) -> Works;
10 find_author(AuthorName, 50283, Works) -> find_author(
   AuthorName, 50284, Works);
11 find_author(AuthorName, 50465, Works) -> find_author(
   AuthorName, 50466, Works);
12 find_author(AuthorName, 50541, Works) -> find_author(
   AuthorName, 50542, Works);
13 find_author(AuthorName, Num, Works) ->
14   DIR = "C:\\Users\\Grant\\Desktop\\Gutenberg_RDF_
   Files\\rdf-files.tar\\cache\\epub", %%
   the location of the RDF files from Project
   Gutenberg
15   FileAuthor = file_read_author(make_path_local(
   DIR, Num)),
16   if
17     FileAuthor == AuthorName ->
18     find_author(AuthorName, Num+1, [
   Num|Works]);
19   true ->
20     find_author(AuthorName, Num+1,
   Works)
21   end.
22
23 file_read_author(FilePath) ->
24   case file:read_file(FilePath) of
25     {ok, File} -> {ok, File};
26     {error, _} -> File = FilePath
27   end,
28   FileString = binary_to_list(File),
29   case
   string:str(FileString, "<pgterms:
   name>") == 0 of
30     true ->
31       "Error:_No_Author_Found";
32     false ->
33       string:sub_string(FileString, (
   string:str(FileString, "<

```

```

pgterms:name>")+14), (string:
str(FileString, "</pgterms:
name>")-1))
34 end.
35
36 make_path_local(DIR, Num) ->
37   string:concat(string:concat(string:concat(string:
   concat(string:concat(DIR, "\\"),
   integer_to_list(Num)), "\\pg"),
   integer_to_list(Num)), ".rdf").

```

Once it is determined which files belong to a given author, the files can be downloaded from one of Project Gutenberg's download mirror sites using the `process_file_numbers` function from the `html_processor` module.

```

1 -module(html_processor).
2 -import(httpc, [request/1]).
3 -export([process_file_numbers/1]).
4
5 % Input: list of Natural, the file numbers of the
   desired documents.
6 % Downloads, processes and writes the files to location
   DIR.
7 process_file_numbers([]) -> inets:stop(), ok;
8 process_file_numbers([Head|Tail]) ->
9   inets:start(),
10   Mirror = "http://www.mirrorservice.org/sites/ftp
   .ibiblio.org/pub/docs/books/gutenberg/",
11   DIR = "C:\\Users\\Grant\\Erlang_Workspace\\
   DocumentProcessor\\Documents", %% Where you
   want the ORIGINAL, downloaded documents to
   be stored
12   Name = integer_to_list(Head),
13   ShortPath = string:concat(string:concat(DIR, "\\")
   , Name),
14   FilePath = string:concat(ShortPath, ".txt"),
15   case string:equal("error:_no_txt_file",
   FileContents) of
16     false -> file:write_file(FilePath,
   FileContents),
17     paragraph:remove_preface_etc(FilePath),
18     paragraph:process_file(string:concat(DIR
   , "\\") , Name),
19     inets:stop(),
20     process_file_numbers(Tail);
21     true -> process_file_numbers(Tail)
22   end.
23 read_URL(URL) ->
24   [BinURL|_] = re:replace(URL, ".txt", ""),
25   BaseURL = binary_to_list(BinURL),
26   case httpc:request(get, {string:concat(BaseURL, ".
   txt"), [], [], []} of
27     {ok, {{_Version1, 200, _ReasonPhrase1},
   _Headers1, Body1}} -> Body1;
28     _Else1 ->
29     case httpc:request(get, {string:concat(BaseURL, "
   -0.txt"), [], [], []} of
30       {ok, {{_Version2, 200, _ReasonPhrase2},
   _Headers2, Body2}} -> Body2;
31       _Else2 ->
32       case httpc:request(get, {string:concat(
   BaseURL, "-1.txt"), [], [], []} of
33         {ok, {{_Version3, 200, _ReasonPhrase3},
   _Headers3, Body3}} -> Body3;
34         _Else3 ->
35         case httpc:request(get, {string:concat(
   BaseURL, "-2.txt"), [], [], []} of
36           {ok, {{_Version4, 200, _ReasonPhrase4},
   _Headers4, Body4}} -> Body4;
37           _Else4 ->
38           case httpc:request(get, {string:concat(
   BaseURL, "-3.txt"), [], [], []} of
39             {ok, {{_Version5, 200, _ReasonPhrase5},
   _Headers5, Body5}} -> Body5;
40             _Else5 ->
41             case httpc:request(get, {string:concat(
   BaseURL, "-4.txt"), [], [], []} of
42               {ok, {{_Version6, 200, _ReasonPhrase6},
   _Headers6, Body6}} -> Body6;
43               _Else6 ->

```

```

44     case httpc:request(get,{string:concat(
45         BaseURL,"-5.txt"), [],[], []} of
46     ok, {{_Version7, 200, _ReasonPhrase7}, _Headers7,
47         Body7}} -> Body7;
48     _Else7 ->
49     case httpc:request(get,{string:concat(
50         BaseURL,"-6.txt"), [],[], []} of
51     {ok, {{_Version8, 200, _ReasonPhrase8},
52         _Headers8, Body8}} -> Body8;
53     _Else8 ->
54     case httpc:request(get,{string:concat(
55         BaseURL,"-7.txt"), [],[], []} of
56     {ok, {{_Version9, 200, _ReasonPhrase9},
57         _Headers9, Body9}} -> Body9;
58     _Else9 ->
59     case httpc:request(get,{string:concat(
60         BaseURL,"-8.txt"), [],[], []} of
61     {ok, {{_Version10, 200, _ReasonPhrase10},
62         _Headers10, Body10}} -> Body10;
63     _Else10 ->
64     case httpc:request(get,{string:concat(
65         BaseURL,"-9.txt"), [],[], []} of
66     {ok, {{_Version11, 200, _ReasonPhrase11},
67         _Headers11, Body11}} -> Body11;
68     _Else11 -> "error: no_txt_file" end
69     end
70     end
71     end
72     end
73     end
74     end
75     end
76     end
77     end
78     end
79     end
80     end
81     end
82     end
83     end
84     end
85     end
86     end
87     end
88     end
89     end
90     end
91     end
92     end
93     end
94     end
95     end
96     end
97     end
98     end
99     end
100    end
101    end
102    end
103    end
104    end
105    end
106    end
107    end
108    end
109    end
110    end
111    end
112    end
113    end
114    end
115    end
116    end
117    end
118    end
119    end
120    end
121    end
122    end
123    end
124    end
125    end
126    end
127    end
128    end
129    end
130    end
131    end
132    end
133    end
134    end
135    end
136    end
137    end
138    end
139    end
140    end
141    end
142    end
143    end
144    end
145    end
146    end
147    end
148    end
149    end
150    end
151    end
152    end
153    end
154    end
155    end
156    end
157    end
158    end
159    end
160    end
161    end
162    end
163    end
164    end
165    end
166    end
167    end
168    end
169    end
170    end
171    end
172    end
173    end
174    end
175    end
176    end
177    end
178    end
179    end
180    end
181    end
182    end
183    end
184    end
185    end
186    end
187    end
188    end
189    end
190    end
191    end
192    end
193    end
194    end
195    end
196    end
197    end
198    end
199    end
200    end
201    end
202    end
203    end
204    end
205    end
206    end
207    end
208    end
209    end
210    end
211    end
212    end
213    end
214    end
215    end
216    end
217    end
218    end
219    end
220    end
221    end
222    end
223    end
224    end
225    end
226    end
227    end
228    end
229    end
230    end
231    end
232    end
233    end
234    end
235    end
236    end
237    end
238    end
239    end
240    end
241    end
242    end
243    end
244    end
245    end
246    end
247    end
248    end
249    end
250    end
251    end
252    end
253    end
254    end
255    end
256    end
257    end
258    end
259    end
260    end
261    end
262    end
263    end
264    end
265    end
266    end
267    end
268    end
269    end
270    end
271    end
272    end
273    end
274    end
275    end
276    end
277    end
278    end
279    end
280    end
281    end
282    end
283    end
284    end
285    end
286    end
287    end
288    end
289    end
290    end
291    end
292    end
293    end
294    end
295    end
296    end
297    end
298    end
299    end
300    end
301    end
302    end
303    end
304    end
305    end
306    end
307    end
308    end
309    end
310    end
311    end
312    end
313    end
314    end
315    end
316    end
317    end
318    end
319    end
320    end
321    end
322    end
323    end
324    end
325    end
326    end
327    end
328    end
329    end
330    end
331    end
332    end
333    end
334    end
335    end
336    end
337    end
338    end
339    end
340    end
341    end
342    end
343    end
344    end
345    end
346    end
347    end
348    end
349    end
350    end
351    end
352    end
353    end
354    end
355    end
356    end
357    end
358    end
359    end
360    end
361    end
362    end
363    end
364    end
365    end
366    end
367    end
368    end
369    end
370    end
371    end
372    end
373    end
374    end
375    end
376    end
377    end
378    end
379    end
380    end
381    end
382    end
383    end
384    end
385    end
386    end
387    end
388    end
389    end
390    end
391    end
392    end
393    end
394    end
395    end
396    end
397    end
398    end
399    end
400    end
401    end
402    end
403    end
404    end
405    end
406    end
407    end
408    end
409    end
410    end
411    end
412    end
413    end
414    end
415    end
416    end
417    end
418    end
419    end
420    end
421    end
422    end
423    end
424    end
425    end
426    end
427    end
428    end
429    end
430    end
431    end
432    end
433    end
434    end
435    end
436    end
437    end
438    end
439    end
440    end
441    end
442    end
443    end
444    end
445    end
446    end
447    end
448    end
449    end
450    end
451    end
452    end
453    end
454    end
455    end
456    end
457    end
458    end
459    end
460    end
461    end
462    end
463    end
464    end
465    end
466    end
467    end
468    end
469    end
470    end
471    end
472    end
473    end
474    end
475    end
476    end
477    end
478    end
479    end
480    end
481    end
482    end
483    end
484    end
485    end
486    end
487    end
488    end
489    end
490    end
491    end
492    end
493    end
494    end
495    end
496    end
497    end
498    end
499    end
500    end
501    end
502    end
503    end
504    end
505    end
506    end
507    end
508    end
509    end
510    end
511    end
512    end
513    end
514    end
515    end
516    end
517    end
518    end
519    end
520    end
521    end
522    end
523    end
524    end
525    end
526    end
527    end
528    end
529    end
530    end
531    end
532    end
533    end
534    end
535    end
536    end
537    end
538    end
539    end
540    end
541    end
542    end
543    end
544    end
545    end
546    end
547    end
548    end
549    end
550    end
551    end
552    end
553    end
554    end
555    end
556    end
557    end
558    end
559    end
560    end
561    end
562    end
563    end
564    end
565    end
566    end
567    end
568    end
569    end
570    end
571    end
572    end
573    end
574    end
575    end
576    end
577    end
578    end
579    end
580    end
581    end
582    end
583    end
584    end
585    end
586    end
587    end
588    end
589    end
590    end
591    end
592    end
593    end
594    end
595    end
596    end
597    end
598    end
599    end
600    end
601    end
602    end
603    end
604    end
605    end
606    end
607    end
608    end
609    end
610    end
611    end
612    end
613    end
614    end
615    end
616    end
617    end
618    end
619    end
620    end
621    end
622    end
623    end
624    end
625    end
626    end
627    end
628    end
629    end
630    end
631    end
632    end
633    end
634    end
635    end
636    end
637    end
638    end
639    end
640    end
641    end
642    end
643    end
644    end
645    end
646    end
647    end
648    end
649    end
650    end
651    end
652    end
653    end
654    end
655    end
656    end
657    end
658    end
659    end
660    end
661    end
662    end
663    end
664    end
665    end
666    end
667    end
668    end
669    end
670    end
671    end
672    end
673    end
674    end
675    end
676    end
677    end
678    end
679    end
680    end
681    end
682    end
683    end
684    end
685    end
686    end
687    end
688    end
689    end
690    end
691    end
692    end
693    end
694    end
695    end
696    end
697    end
698    end
699    end
700    end
701    end
702    end
703    end
704    end
705    end
706    end
707    end
708    end
709    end
710    end
711    end
712    end
713    end
714    end
715    end
716    end
717    end
718    end
719    end
720    end
721    end
722    end
723    end
724    end
725    end
726    end
727    end
728    end
729    end
730    end
731    end
732    end
733    end
734    end
735    end
736    end
737    end
738    end
739    end
740    end
741    end
742    end
743    end
744    end
745    end
746    end
747    end
748    end
749    end
750    end
751    end
752    end
753    end
754    end
755    end
756    end
757    end
758    end
759    end
760    end
761    end
762    end
763    end
764    end
765    end
766    end
767    end
768    end
769    end
770    end
771    end
772    end
773    end
774    end
775    end
776    end
777    end
778    end
779    end
780    end
781    end
782    end
783    end
784    end
785    end
786    end
787    end
788    end
789    end
790    end
791    end
792    end
793    end
794    end
795    end
796    end
797    end
798    end
799    end
800    end
801    end
802    end
803    end
804    end
805    end
806    end
807    end
808    end
809    end
810    end
811    end
812    end
813    end
814    end
815    end
816    end
817    end
818    end
819    end
820    end
821    end
822    end
823    end
824    end
825    end
826    end
827    end
828    end
829    end
830    end
831    end
832    end
833    end
834    end
835    end
836    end
837    end
838    end
839    end
840    end
841    end
842    end
843    end
844    end
845    end
846    end
847    end
848    end
849    end
850    end
851    end
852    end
853    end
854    end
855    end
856    end
857    end
858    end
859    end
860    end
861    end
862    end
863    end
864    end
865    end
866    end
867    end
868    end
869    end
870    end
871    end
872    end
873    end
874    end
875    end
876    end
877    end
878    end
879    end
880    end
881    end
882    end
883    end
884    end
885    end
886    end
887    end
888    end
889    end
890    end
891    end
892    end
893    end
894    end
895    end
896    end
897    end
898    end
899    end
900    end
901    end
902    end
903    end
904    end
905    end
906    end
907    end
908    end
909    end
910    end
911    end
912    end
913    end
914    end
915    end
916    end
917    end
918    end
919    end
920    end
921    end
922    end
923    end
924    end
925    end
926    end
927    end
928    end
929    end
930    end
931    end
932    end
933    end
934    end
935    end
936    end
937    end
938    end
939    end
940    end
941    end
942    end
943    end
944    end
945    end
946    end
947    end
948    end
949    end
950    end
951    end
952    end
953    end
954    end
955    end
956    end
957    end
958    end
959    end
960    end
961    end
962    end
963    end
964    end
965    end
966    end
967    end
968    end
969    end
970    end
971    end
972    end
973    end
974    end
975    end
976    end
977    end
978    end
979    end
980    end
981    end
982    end
983    end
984    end
985    end
986    end
987    end
988    end
989    end
990    end
991    end
992    end
993    end
994    end
995    end
996    end
997    end
998    end
999    end
1000   end

```

III. DOCUMENT PREPROCESSING

After documents are downloaded and stored, they can be partially processed, removing prefaces, introductions etc, and splitting them into chapters, each of which is stored as a separate file, in preparation for parsing. This partial processing is accomplished using the `process_file` function of the `paragraph` module.

```

1 -module(paragraph).
2 -export([process_file/2]).
3 -import(re,[split/3,replace/4]).
4 %document storage location: C:\Users\Grant\Erlang
   Workspace\DocumentProcessor\Documents
5
6 %FilePath is the directory that the file is in. Name is
   the name of the file, without extension ("practice",
   "testFile", etc).
7 process_file(FilePath,Name)->
8     {ok, File} = file:read_file(string:concat(string:
9         concat(FilePath,Name), ".txt")),
10     process_paragraphs(remove_rn(File),Name).
11
12 remove_preface_etc(FullFilePath)->
13     {ok, File} = file:read_file(FullFilePath),
14     FileString = binary_to_list(File),
15     TrimmedFile = string:substr(FileString,
16         get_start(FileString)),
17     file:write_file(FullFilePath, TrimmedFile).
18
19 get_start(File)->
20     %% Starters = ["\r\n\r\nCHAPTER_1","\r\n\r\nCHAPTER
   _ONE","\r\n\r\nCHAPTER_I","\r\n\r\nSTAVE_ONE"],
21     case string:str(File, "\r\n\r\nCHAPTER_1") of
22     0 ->

```

```

21     case string:str(File, "\r\n\r\nCHAPTER_ONE") of
22     0 ->
23     case string:str(File, "\r\n\r\nCHAPTER_I") of
24     0 ->
25     case string:str(File, "\r\n\r\nSTAVE_ONE") of
26     0->1;
27     N->N end;
28     N->N end;
29     N->N end;
30     N->N end.
31
32 process_paragraphs(Document,Name)->
33     Paragraphs = split(Document, "CHAPTER",[{return,
34         list}]),
35     write_files(lists:map(fun remove_rn/1,
36         Paragraphs),string:concat(Name,"_"),"0").
37
38 %DIR is the location that you want to write the
   individual paragraph files to.
39 write_files([],_,_)>ok;
40 write_files([Head |Paragraphs], Name, Num)->
41     DIR = "C:\\Users\\Grant\\ErlangWorkspace\\
   DocumentProcessor\\Documents\\", %%Where
   you want the PARSED documents to be stored
42     file:write_file(string:concat(string:concat(DIR,
43         string:concat(Name, Num)), ".txt"), Head),
44     {N,_} = string:to_integer(Num),
45     write_files(Paragraphs, Name, integer_to_list(N
46         +1)).
47
48 remove_rn(Document)->
49     replace(Document, "\r\n", "_", [{return, list}]).

```

IV. TEXT PARSING

Once the document has been preprocessed, they should be turned into more structured data by extracting words and special characters. The user can call `chapter_stats` to compute the statistics for a chapter. For this step we use a helper function `isalpha`, which returns whether the input character is a letter (a-z, A-Z), a special character (comma, semicolon, exclamation mark, ...) or neither.

```

1 % Input: Character Ch
2 % Output: true if Ch is a letter; Ch if Ch is a special
   character; false otherwise
3 isalpha(Ch) -> case lists:member(Ch, ",; \!-.") of
4     true -> Ch;
5     false -> case lists:member(Ch, lists:seq(97, 97+25) ++
6         lists:seq(65, 65+25)) of
7         true -> true;
8         false -> false
9     end
10 end.

```

The `parse` function takes a string and turns it into a list of words (i.e. a continuous sequence of characters for which `parse` returns true) and special characters:

```

1 % Extracts words and special characters.
2 parse(Text) -> parse_(Text, "", false, []).
3
4 parse_("", Word, Alpha, Result) -> Result;
5 parse_([Ch|Rest], Word, Alpha, Result) ->
6     case isalpha(Ch) of
7     true -> parse_(Rest, [Ch|Word], true, Result);
8     false -> if Alpha -> parse_(Rest, "", false, [{word,
9         lists:reverse(Word)}|Result]);
10         true -> parse_(Rest, "", false, Result)
11     end;
12     N -> if Alpha -> parse_(Rest, "", false, [{char, N
13         }|[{word, lists:reverse(Word)}|Result]]);
14         true -> parse_(Rest, "", false, [{char, N
15         }|Result])
16     end
17 end.

```

Note that the output is reversed. So if we, for example, call parse on “Abc, def. GH-IJ kl!” and reverse the output, we get the following.

```
1 [{word, "Abc"},
2  {char, 44},
3  {word, "def"},
4  {char, 46},
5  {word, "GH"},
6  {char, 45},
7  {word, "IJ"},
8  {word, "kl"},
9  {char, 33}]
```

Since case sensitivity does not really make sense for our purposes, we might modify the function so that all words are lowercased.

The (reversed) result of parse can then be used to calculate paragraph statistics using the stats function. It calculates how often each token appears, the sum of the lengths of all words, the number of words, and the lengths of the sentences.

```
1 % Computes basic statistics.
2 % SentenceLength is reversed.
3 stats([], CountMap, WordLengthSum, WordCount,
4       SentenceLength)
5 -> {CountMap, WordLengthSum, WordCount, lists:nthtail(1,
6       SentenceLength)};
7 stats([T|Rest], CountMap, WordLengthSum, WordCount,
8       SentenceLength)
9 -> NewCountMap = maps:put(T, maps:get(T, CountMap, 0) +
10 1, CountMap),
11 [CurrentLength|RestSL] = SentenceLength,
12 case T of {word, Word} -> stats(Rest, NewCountMap,
13 WordLengthSum + length(Word), WordCount + 1, [
14 CurrentLength+1|RestSL]);
15 {char, Ch} when Ch == \$. orelse Ch == \$?
16 orelse Ch == \$! -> stats(Rest, NewCountMap, WordLengthSum, WordCount,
17 [0|SentenceLength]);
18 _Else -> stats(Rest, NewCountMap, WordLengthSum, WordCount,
19 SentenceLength)
20 end.
21 % Parse text and compute stats.
22 stats(Text) -> stats(lists:reverse(parse(Text)), maps:
23 new(), 0, 0, [0]).
```

The output of stats(lists:reverse(parse(`Abc, def. GH-IJ kl!`))), maps:new(), 0, 0, [0]) is, for example:

```
1 [{#{char, 33} => 1,
2  {char, 44} => 1,
3  {char, 45} => 1,
4  {char, 46} => 1,
5  {word, "Abc"} => 1,
6  {word, "GH"} => 1,
7  {word, "IJ"} => 1,
8  {word, "def"} => 1,
9  {word, "kl"} => 1},
10 12, 5,
11 [3, 2]]
```

This function is called once for each paragraph, and the results are combined in a function called chapter_stats, which computes the statistics of a chapter. They are then used as input for the artificial neural network. We have currently decided on the following characteristics; they have already been used in [2]:

- type-token-ratio
- mean word length

- mean sentence length
- standard deviation of sentence length
- mean paragraph length
- chapter length
- number of commas, semicolons, quotation marks, exclamation marks, hyphens, “and”s, “but”s, “however”s, “if”s, “that”s, “more”s, “must”s, “might”s, “this”s, “very”s per word.

Thus, the input for the network is a 21-tuple of real numbers.

The code for chapter_stats is as follows.

```
1 % The main function for text parsing. Called by the user
2 .
3 % Calculate the statistics of a chapter.
4 % Paragraphs must be separated by \r\n\r\n.
5 chapter_stats(Text)
6 -> Paragraphs = re:split(Text, "\r\n\r\n", [{return,
7 list}]),
8 ParagraphStats = lists:map(fun stats_/1, Paragraphs
9 ),
10 {CountMaps, WordLengthSums, WordCounts,
11 SentenceLengths} = unzip4(ParagraphStats),
12 CountMap = map_merge(CountMaps),
13 WordLengthSum = lists:sum(WordLengthSums),
14 WordCount = lists:sum(WordCounts),
15 SentenceLength = lists:append(SentenceLengths),
16 MeanSentenceLength = mean(SentenceLength),
17 {unique_words(CountMap) / WordCount, % type-token-
18 ratio
19 WordLengthSum / WordCount, % mean word length
20 MeanSentenceLength, % mean sentence length
21 standard_deviation(SentenceLength,
22 MeanSentenceLength), % sd of sentence length
23 mean(WordCounts), % mean paragraph length
24 WordCount, % chapter length
25 maps:get({char, $,}, CountMap, 0) / WordCount, %
26 comma density
27 maps:get({char, $;}, CountMap, 0) / WordCount, %
28 semicolon density
29 maps:get({char, $"}, CountMap, 0) / WordCount, %
30 quotation mark density
31 maps:get({char, $!}, CountMap, 0) / WordCount, %
32 exclamation mark density
33 maps:get({char, $-}, CountMap, 0) / WordCount, %
34 hyphen density
35 maps:get({word, "and"}, CountMap, 0) / WordCount, %
36 "and" density
37 maps:get({word, "but"}, CountMap, 0) / WordCount, %
38 "but" density
39 maps:get({word, "however"}, CountMap, 0) /
40 WordCount, % "however" density
41 maps:get({word, "if"}, CountMap, 0) / WordCount, %
42 "if" density
43 maps:get({word, "that"}, CountMap, 0) / WordCount,
44 % "that" density
45 maps:get({word, "more"}, CountMap, 0) / WordCount,
46 % "more" density
47 maps:get({word, "must"}, CountMap, 0) / WordCount,
48 maps:get({word, "might"}, CountMap, 0) / WordCount,
49 maps:get({word, "this"}, CountMap, 0) / WordCount,
50 maps:get({word, "very"}, CountMap, 0) / WordCount}.
```

V. THE BACKPROPAGATION ALGORITHM

The backpropagation algorithm is used to train the artificial neural network so that it can later be used to classify documents. The theory behind it has already been described in Section IV-A of the background research report. We give a very brief summary of what has to be computed: If L_1, \dots, L_k are the layers of a neural network, if $p = |L_1|$, $r = |L_k|$, $L_k = \{B_1, \dots, B_r\}$, and if (X, Y) is a training sample with $X = (x_1, \dots, x_p)$, $Y = (y_1, \dots, y_r)$, then compute the output O_s of every node s using the feedforward algorithm (Section

VI). Compute the δ -values and the gradients for the last layer using the equations

$$\delta_{B_j} = (O_{B_j} - y_j)O_{B_j}(1 - O_{B_j}),$$

$$\frac{\partial E}{\partial w_{tB_j}} = \delta_{B_j} O_t$$

for all $j \in \{1, \dots, r\}$ and every $t \in L_{k-1}$. The term w_{ab} denotes the weight of the arc (a, b) , where a and b are nodes. Then compute for all $s \in L_{j-1}$ and $t \in L_j$

$$\delta_t = O_t(1 - O_t) \sum_{u \in L_{j+1}} \delta_u w_{tu},$$

$$\frac{\partial E}{\partial w_{st}} = \delta_t O_s,$$

iteratively for $j = k - 1, \dots, 2$. Finally update

$$w_{ij} := w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}}$$

for all arcs (i, j) , where $\alpha > 0$ is the learning rate.

We decided to first compute all δ -values and postpone the computation of the gradients to when the weights are updated.

A. Backpropagation: A Sequential Implementation

The sequential backpropagation was implemented in a function `train_network` that takes a neural network and a training sample (X, Y) and returns a new neural network with updated weights so that the regression function value at X is now closer to Y .

```
1 % Returns the network with updated weights.
2 train_network({network, Layers, Layersize, Weights}, X,
3   Y)
4   -> Output = feed_forward(#network{layers=Layers,
5     layersize=Layersize, weights=lists:append(Weights)},
6     X),
7     Deltas = compute_deltas_init(lists:reverse(Weights),
8       lists:reverse(Output), Y),
9     Alpha = 0.08
10    NewWeights = update_weights(Alpha, Output, Deltas,
11      Weights),
12    #network{layers=Layers, layersize=Layersize, weights=
13      NewWeights}.
```

The function first uses the feedforward algorithm (which was already shown in Section IV of the proof-of-concept) to compute the output of the network given X . This output, together with the target value Y can be used to compute the δ -values for each node—this is done in `compute_deltas_init`. Finally the function calls `update_weights` to update the weights and returns the new network. The learning rate Alpha can be adjusted.

Note that, unlike in the proof-of-concept, the weights of a network are now grouped by layer so that the field `weights` is a list of lists. This is why we call `lists:append` in line 3. We also have a function `feed_forward` which does that automatically.

We now show the `compute_deltas_init` function. It computes the δ -values for the nodes of the output layer and then calls `compute_deltas` which recursively computes the δ -values of all other layers. This “distinction” between

the last layer and the other layers is due to the fact that the δ -values of the output layer depend on the target value, whereas those of the other layers do not (at least not directly; cf. the equations above).

```
1 % Computes the delta values. The layers of the input
2   network must be reversed.
3 compute_deltas_init(Weights, [Output|ORest], Target)
4   -> Deltas = map(fun (X, Y) -> (X-Y)*X*(1-X) end,
5     Output, Target),
6     compute_deltas([Deltas], Weights, ORest).
7
8 % Recursively computes the delta values. The layers of
9   the input network must be reversed.
10 % Requires the delta values of the next layer.
11 % Called by compute_deltas_init.
12 compute_deltas(Deltas, [], []) -> Deltas;
13 compute_deltas([Deltas|DRest], [Weights|WRest], [Output|
14   ORest])
15   -> ThisLayerSize = length(Weights),
16     PreviousLayerSize = length(Deltas),
17     AWeights = partitionList(Weights, ThisLayerSize),
18     NewDeltas = map(fun (O, NodeWeights) -> O*(1-O) *
19       lists:foldl(fun ({X, Y}, Acc) -> Acc + X*Y end,
20         0, myzip(Deltas, NodeWeights)) end, Output,
21     AWeights), % The delta values for this layer.
22     compute_deltas([NewDeltas|Deltas|DRest], WRest,
23       ORest).
```

Note that, since backpropagation works from the last layer to the first, we require that the list of weights and output values is reversed. This makes recursion easier.

With the output and the δ -values we can easily update the weights:

```
1 update_weights(Alpha, Output, Delta, Weights)
2   -> map3(fun (Os, Ds, Ws) -> map3(fun (O, D, W) -> W -
3     Alpha * D * O end, repeat(Os, length(Ds)),
4     repeat_each(Ds, length(Os)), Ws) end, Output,
5     lists:nthtail(1, Delta), Weights).
```

With these function definitions, if we want to train a neural network, we only have to repeatedly call `train_network`.

B. Backpropagation: Training Session Parallelism

Recall the idea behind training session parallelism (cf. Section IV-B of the background research report): Start with an artificial neural network and a training data set $X_1, \dots, X_n \in \mathbb{R}^p$, $Y_1, \dots, Y_n \in \mathbb{R}^r$. For each $i = 1, \dots, n$, train the network with the sample (X_i, Y_i) (using `train_network`). If the training error is sufficiently small or after a predefined number of iterations, stop. Otherwise, repeat the previous step. This entire procedure is known as a training session. With training session parallelism, several such training sessions are performed in parallel, each with different initial (random) neural networks. Once all sessions have terminated, we can choose the network with the smallest regression error on the training data.

Since we already have the `train_network` function and given Erlang’s concurrency features, training session parallelism can be implemented easily.

```
1 % Train N networks with training data Xs, Ys (lists) and
2   return the best network.
3 training_session_parallelism(Xs, Ys, N)
4   -> training_session_parallelism_(Xs, Ys, N, N).
5
6 % Train N networks with training data Xs, Ys (lists) and
7   return the best network.
8 % N-K sessions have already been started.
```

```

7 training_session_parallelism_(Xs, Ys, N, 0)
8   -> receive_results(Xs, Ys, N); % all sessions have
      been started; now collect the results
9 training_session_parallelism_(Xs, Ys, N, K)
10  -> InputLayerSize = length(lists:nth(1, Xs)), % the
      size of the first and last layer are determined by
      the training data
11      OutputLayerSize = length(lists:nth(1, Ys)),
12      HiddenLayerSize = round(1.5*InputLayerSize), % can
      be adjusted
13      Network = #network{layers=3, layersize=[
        InputLayerSize, HiddenLayerSize, OutputLayerSize
      ], weights=[random_list(InputLayerSize*
        HiddenLayerSize), random_list(HiddenLayerSize*
        OutputLayerSize)]}, % can be adjusted
14      NumberOfIterations = 500000, % can be adjusted
15      spawn(ann, train_concurrently, [self(), Network,
        myzip(Xs, Ys), NumberOfIterations]), % start
        training session in new thread
16      training_session_parallelism_(Xs, Ys, N, K-1). %
        start the remaining K-1 sessions recursively

```

In this case we use a neural network with three layers. The size of the first layer is the dimension of the input data; the size of the last layer is the dimension of the target data—you have no choice. However, you could decide to have more than one hidden layer or to change the size of the hidden layer. The initial weights are chosen uniformly at random from the interval $(0,1) \subseteq \mathbb{R}$ using `random_list`, which we will define later. Additionally you could decide to change the number of iterations for each training session.

After the (partly) random neural network is created, `training_session_parallelism_` creates a new Erlang process (line 15; cf. Section II of the background research report; note that `train_concurrently` is in the `ann` module) which does one training session with that network and the training data. The new process is also given the ID of the current process (`self()`) so that it will be able to send back the results. Then the remaining $K - 1$ sessions are started recursively. Once all sessions are started, it calls `receive_results` to *receive* the N networks and choose the best:

```

1 % Collect N results. Xs, Ys: training data.
2 receive_results(Xs, Ys, N) -> receive_results_(Xs, Ys, N
  , 0, 9999999999).
3
4 receive_results_(Xs, Ys, 0, BestNetwork, SmallestError)
5   -> {BestNetwork, SmallestError}; % when all networks
      have been received, return the one with the
      smallest training error
6 receive_results_(Xs, Ys, N, BestNetwork, SmallestError)
7   -> receive
8       NewNetwork
9       -> NewError = compute_training_error(NewNetwork
        , Xs, Ys),
10         if NewError < SmallestError ->
            receive_results_(Xs, Ys, N-1, NewNetwork
              , NewError);
11         true -> receive_results_(Xs, Ys, N-1,
            BestNetwork, SmallestError)
12       end
13     end.

```

In order to complete the code, all that is left to do is define `train_concurrently`. This function repeatedly calls `train_network` from V-A and after the final iteration it sends the network back to the “main process” using the `!` operator.

```

1 train_concurrently(Process, Network, TrainingSamples, 0)

```

```

2   -> Process ! Network;
3 train_concurrently(Process, Network, TrainingSamples, N)
4   -> train_concurrently(Process, lists:foldl(fun ({X, Y
      }, Acc) -> train_network(Acc, X, Y) end, Network,
      TrainingSamples), TrainingSamples, N-1).

```

The call to `lists:foldl` does an entire round of training with all the samples being used once. Note that `TrainingSamples` is a tuple, with the first element being a list of the X values of the training samples, and the second element being a list of the Y values. It comes from the call to `myzip` in line 15 of the code for `training_session_parallelism_`.

Just like `myzip` and other helper functions, the code for `compute_training_error` is shown in VII.

VI. THE FEEDFORWARD ALGORITHM

In this section we present our implementation of the feedforward algorithm for multilayer perceptrons as it was implicitly given in section III-A of our background research report.¹ It is not only needed to classify the unknown documents, but also for the backpropagation algorithm, which is why our implementation returns the output of every node instead of just the output of the nodes in the output layer.

We first define the data structure for a neural network. It consists of the number of layers (an integer), the size of each layer (a list of integers), and the weights (a list of integers).

The weights must be carefully ordered: If layer l has n nodes $(l,1), \dots, (l,n)$ and layer $l+1$ has m nodes $(l+1,1), \dots, (l+1,m)$, we use the following ordering of the arcs in $L_l \times L_{l+1}$ ($1 \leq l < k$)

$$((l,1), (l+1,1)), \dots, ((l,n), (l+1,1)), ((l,1), (l+1,2)), \dots, ((l,n), (l+1,m))$$

to list the weights of the arcs in $L_1 \times L_2, \dots, L_{k-1} \times L_k$ (in this order).

The implementation of the feedforward algorithm works by recursion over the layers: We want to compute the output of a k -layer network when given some input. Once we have computed the output of the first layer (by applying the sigmoid function), we combine them according to the weights of the arcs between the first and the second layer. This gives us the input to the second layer, and we have reduced the task to computing the output of a $(k-1)$ -layer network.

```

1 -record(network, {layers, layersize, weights}).
2
3 testann() -> #network{layers=3, layersize=[2,1,2],
  weights=[[1,1],[1,1]]}.
4
5 % Weights is a list of lists
6 feed_forward({network, Layers, Layersize, Weights},
  Input) -> feed_forward(#network{layers=Layers,
  layersize=Layersize, weights=list:append(Weights)},
  Input).
7
8 % Weights is a list
9 feed_forward({network, Layers, Layersize, Weights},
  Input)
10 -> if Layers == 1 -> [lists:map(fun sigmoid/1, Input)];
11     true -> InputSize = length(Input),

```

¹We use the same notation.

```

12         Output = lists:map(fun sigmoid/1,
13                               Input),
14         NextLayerSize = lists:nth(2,
15                                   Layersize),
16         NextInput = compute_next_input(
17                               Output, NextLayerSize, Weights)
18
19         NewWeights = lists:nthtail(length(
20                               Output) * NextLayerSize,
21                                   Weights),
22         [Output|feed_forward(#network{
23                               layers = Layers-1, layersize =
24                               lists:nthtail(1, Layersize),
25                               weights = NewWeights},
26                               NextInput)]
27     end.
28
29 % Combines the output of the current layer to compute
30 % the input to the next layer.
31 compute_next_input(Output, 0, Weights) -> [];
32 compute_next_input(Output, NextLayerSize, Weights)
33 -> NextInput = lists:foldl(fun ({Out, Wei}, Sum) -> Sum
34     + Out * Wei end, 0, myzip(Output, Weights)),
35     [NextInput|compute_next_input(Output, NextLayerSize
36     -1, lists:nthtail(length(Output), Weights))].
37
38 % Like the regular zip, but the lists can have different
39 % lengths.
40 myzip([], Bs) -> [];
41 myzip(As, []) -> [];
42 myzip([A|As], [B|Bs]) -> [{A, B}|myzip(As,Bs)].
43
44 sigmoid(X) -> 1 / (1 + math:exp(-X)).

```

VII. HELPER FUNCTIONS

Here we show the code for some helper functions. They are minor extensions to existing Erlang functions or easy functions which are not inherently related to neural networks, or both. The provided comments should be enough to understand the code or at least their purpose.

```

1 % Merge all maps, adding the values of common keys.
2 map_merge([]) -> maps:new();
3 map_merge([X|[]]) -> X;
4 map_merge([X|Y|Rest])
5 -> map_merge([maps:fold(fun (K, V, AccIn) -> maps:put(
6     K, maps:get(K, AccIn, 0) + V, AccIn) end, X, Y)|
7     Rest]).

```

```

1 % Like the regular unzip, but List is a list of 4-tuples
2 unzip4(List) -> lists:foldr(fun ({A, B, C, D}, {As, Bs,
3     Cs, Ds}) -> {[A|As], [B|Bs], [C|Cs], [D|Ds]} end,
4     {[[], [], [], []]}, List).

```

```

1 % Counts the number of unique words (entries with key {
2     word, _}).
3 unique_words(CountMap)
4 -> lists:foldl(fun (X, Acc)
5     -> case X of {word, _} -> Acc + 1;
6     _Else -> Acc end end,
7     0, maps:keys(CountMap)).

```

```

1 % Compute the mean of the elements in Xs.
2 mean(Xs) -> lists:foldr(fun (X, Acc) -> X + Acc end, 0,
3     Xs) / length(Xs).

```

```

1 % Compute the standard deviation of the elements in Xs.
2 % XMean = mean(Xs)
3 standard_deviation(Xs, XMean) -> math:sqrt(lists:foldr(
4     fun (X, Acc) -> (X-XMean)*(X-XMean) + Acc end, 0, Xs
5     ) / length(Xs)).

```

```

1 % Compute the regression error for the training data set
2 compute_training_error(Network, [], []) -> 0;
3 compute_training_error(Network, [X|Xs], [Y|Ys])
4 -> Output = lists:last(feed_forward(Network, X)),
5     lists:sum(map(fun (T, O) -> (T-O)*(T-O) end, Y,
6     Output)) + compute_training_error(Network, Xs,
7     Ys).

```

```

1 % N-fold concatenation of Xs
2 repeat(Xs, 0) -> [];
3 repeat(Xs, N) -> Xs ++ repeat(Xs, N-1).

```

```

1 % [x1,...,xK] -> [x1,...,x1,x2,...,x2,...,xK,...,xK]
2 repeat_each([], N) -> [];
3 repeat_each([X|Xs], N) -> repeat([X], N) ++ repeat_each(
4     Xs, N).

```

```

1 % Partition the list into N smaller lists by assigning
2 % the elements cyclically.
3 partitionList([1,2,3,4,5,6,7,8,9], 3) = [[1,4,7],
4     [2,5,8], [3,6,9]]
5 partitionList(List, N) -> partitionList_(List, N, N).
6 partitionList_(List, N, 0) -> [];
7 partitionList_(List, N, K) -> [partListHelp(List, N, 0)|
8     partitionList_(lists:nthtail(1, List), N, K-1)].
9 partListHelp([], N, M) -> [];
10 partListHelp([X|Xs], N, 0) -> [X|partListHelp(Xs, N, N
11     -1)];
12 partListHelp([X|Xs], N, M) -> partListHelp(Xs, N, M-1).

```

```

1 % Like the regular map, but with two lists.
2 map(Function, [], Ys) -> [];
3 map(Function, Xs, []) -> [];
4 map(Function, [X|Xs], [Y|Ys])
5 -> [Function(X, Y)|map(Function,Xs,Ys)].

```

```

1 % Like the regular map, but with three lists.
2 map3(Function, [], Ys, Zs) -> [];
3 map3(Function, Xs, [], Zs) -> [];
4 map3(Function, Xs, Ys, []) -> [];
5 map3(Function, [X|Xs], [Y|Ys], [Z|Zs])
6 -> [Function(X, Y, Z)|map3(Function,Xs,Ys, Zs)].

```

```

1 % Like the regular zip, but the lists can have different
2 % lengths.
3 myzip([], Bs) -> [];
4 myzip(As, []) -> [];
5 myzip([A|As], [B|Bs]) -> [{A, B}|myzip(As,Bs)].

```

```

1 % Return a list of length N, each element being a random
2 % number with uniform distribution on the interval
3 % (0,1).
4 random_list(0) -> [];
5 random_list(N) -> [random:uniform()|random_list(N-1)].

```

```

1 % The sigmoid function.
2 sigmoid(X) -> 1 / (1 + math:exp(-X)).

```

VIII. NEURAL NETWORK DEMONSTRATION

We give a small example on how the code from Sections V and VI can be used to train a neural network.

Suppose we want to approximate a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ and we only know the values

$$f(0,0) = 0,$$

$$f(1,2) = 0.5.$$

The corresponding training data set is $X_s = [[0,0],[1,2]]$ and $Y_s = [[0],[0.5]]$. Suppose we want to train two networks simultaneously and choose the one with the smaller training error. We can do this as follows:

```

1 Eshell V7.1
2 (ann@User)1> {Network, TrainingError} = ann:
   training_session_parallelism([[0,0],[1,2]],
   [[0],[0.5]], 2).
3 {{network,3,
4   [2,3,1],
5   [[-0.19834785545597572, 2.159548206120467,
6     3.1102682198495764,
7     -4.345304952255812, 3.9994175264983896,
8     -5.033623392917592],
9     [4.467630279109158, -9.770363242212438,
10    -11.638732867738266]]},
11 1.158818489081351e-4}

```

We see that we get a small network with three layers, and the training error as computed by `compute_training_error` is about $1.16 \cdot 10^{-4}$. We can check the goodness of the approximation on the training data using `feed_forward`:

```

1 (ann@User)2> ann:feed_forward(Network, [0,0]).
2 [[0.5, 0.5],
3  [0.7272272880121131, 0.3503460704822918,
4   0.3735299139496292],
5  [0.010756390215779843]]

```

As already said, this computes the output of all nodes ordered by layer. In this case, the regression function value at (0,0) is about 0.01 (line 4), which is close to 0.

```

1 (ann@User)3> ann:feed_forward(Network, [1,2]).
2 [[0.7310585786300049, 0.8807970779778823],
3  [0.8528485744767521, 0.17457504547237046,
4   0.1809695787141338],
5  [0.49957348102737303]]

```

The regression function value at (1,2) is about 0.4996 (line 4), which is very close to 0.5.

This should be evidence enough to show that the neural network code works and is very easy to use.

CONCLUSION

We have developed our project into a useful tool for document classification, which even features generally usable, concurrent code for neural networks. With a little more time and effort we could have implemented node parallelism and significantly increased the usability of the provided code by requiring less user interaction. Additionally the training efficiency and the goodness of approximation could certainly be increased by tweaking the network structure and the choice of the learning rate. Still we think that we have created a valuable basis for anyone who is interested in the intersection between neural networks and concurrent programming with Erlang.

REFERENCES

- [1] Our GitHub Repository. <https://github.com/guenterg/cpsc311project>, December 7, 2015.
- [2] R. C. He and K. Rasheed. Using Machine Learning Techniques for Stylometry. <http://www2.tcs.ifi.lmu.de/~ramyaa/publications/stylometry.pdf>, November 25, 2015.