```
(ns n-gram.letters.hierarchical-dirichlet
       (:require [n-gram.misc.misc-functions :refer :all]
                 [n-gram.letters.file-reader :refer :all]
                 [n-gram.letters.letter-maker :refer :all]
                 [n-gram.letters.letter-predictor :refer :all]))

(defn K "Returns K (MacKay and Peto Equation 34)" [alpha]
  (let [count-vals (vals all-char-counts)]
  (+ (sum  (map #(Math/log (/ (+ % alpha) alpha)) count-vals)) (* 0.5 (sum (map
#(/ % (* alpha (+ % alpha))) count-vals))))))

(def K-memo "Memoized K" (memoize K))

(defn F-w2-given-w1  "Returns all counts of bigrams with word w2 as the second
word" [w2] (zipmap (map key all-char-pair-counts)
(map #(if (= (str (second (key %))) w2) (val %) 0.0) all-char-pair-counts)))

(def F-w2-given-w1-memo "Memoized F-w2-given-w1" (memoize F-w2-given-w1))

(defn V-i "Returns the number of entries in row 'i' of F-w2-given-w1 that are
non-zero" [F-w2-w1] (sum (map #(if (< % 1) 0 1) F-w2-w1)))

(def V-i-memo "Memoized V-i" (memoize V-i))

(defn N-f-i "Returns the number of contexts w1 such that F-w2-given-w1 is
greater than or equal to f"
  [f F-w2-w1] (sum (map #(if (>= f %) 0 1) F-w2-w1)))

(def N-f-i-memo "Memoized N-f-i" (memoize N-f-i))

(defn find-F-max "Returns the largest F such that N-f-i is greater than zero"
[F-w2-w1 f] (if (> (N-f-i-memo f F-w2-w1) 0) (find-F-max F-w2-w1 (inc f)) f))

(def find-F-max-memo "Memoized find-F-max" (memoize find-F-max))

(defn G-i "Returns G-i (MacKay and Peto Equation 32)" [i] (let [F-w2-w1 (F-w2-
given-w1-memo i) F-vals (vals F-w2-w1) F-min 2 F-max (find-F-max-memo F-vals F-
min) f-vector (take (- F-max (dec F-min)) (iterate inc F-min))]
                (sum (map #(/ (N-f-i-memo % F-vals) (- % 1)) f-vector))))

(def G-i-memo "Memoized G-i" (memoize G-i))

(defn H-i "Returns H-i (MacKay and Peto Equation 33)" [i] (let [F-w2-w1 (F-w2-
given-w1-memo i) F-vals (vals F-w2-w1) F-min 2 F-max (find-F-max-memo F-vals F-
```

```
min) f-vector (take (- F-max (dec F-min)) (iterate inc F-min))] (sum (map #(/
(N-f-i-memo % F-vals)(Math/pow (- % 1) 2)) f-vector))))

(def H-i-memo "Memoized H-i" (memoize H-i))

(defn u-i-MP "Returns the optimal value for u-i" [alpha i] (let [F-w2-w1 (vals
(F-w2-given-w1-memo i)) V (V-i-memo F-w2-w1) K-alpha (K-memo alpha) G (G-i-memo
i) H (H-i-memo i)] (/ (* 2 V) (+ K-alpha (- G)

(java.lang.Math/sqrt (+ (Math/pow (- K-alpha G) 2) (* 4 H V))))))))

(def u-i-MP-memo "Memoized u-i-MP" (memoize u-i-MP))

(defn alpha-MP "Returns alpha based on the u-i values inputted" [u] (sum (vals
u)))

(def alpha-MP-memo "Memoized alpha" (memoize alpha-MP))

(def initial-alpha 6)

(defn find-all-u [alpha] (zipmap (keys all-char-counts) (map #(u-i-MP-memo
alpha %) (keys all-char-counts))))

(def find-all-u-memo "Memoized find-all-u" (memoize find-all-u))

(defn find-optimum-u ([] (let [u (find-all-u-memo initial-alpha) alpha (alpha-
MP-memo u)] (if (equal initial-alpha alpha 2) u (find-optimum-u alpha))))
  ([old-alpha] (let [u (find-all-u-memo old-alpha) alpha (alpha-MP-memo u)] (if
(equal old-alpha alpha 2) u (find-optimum-u alpha)))))

(def find-optimum-u-memo "Memoized find-optimum-u" (memoize find-optimum-u))


(defn find-all-m [alpha u] (zipmap (keys u) (map #(/ % alpha) (vals u))))

(def find-all-m-memo (memoize find-all-m))

(defn lambda [F-w1 alpha] (/ alpha (+ F-w1 alpha)))

(def lambda-memo (memoize lambda))

(defn f-w2-w1 [F-w2-w1 F-w1] (if (= 0 F-w2-w1) 0 (/ F-w2-w1 F-w1)))

(def f-w2-w1-memo (memoize f-w2-w1))
```

```
(defn find-m-i [i m] (m i))

(def find-m-i-memo (memoize find-m-i))

(def u-optimum (find-optimum-u-memo))

(def alpha-optimum (alpha-MP-memo u-optimum))

(def m-optimum (find-all-m-memo alpha-optimum u-optimum))

(defn P ([w2 w1 alpha m] (let [F-w2-w1 (get-count-memo all-char-pair-counts
(str w1 w2)) F-w1 (get-count-memo all-char-counts w1) m-i (find-m-i-memo w2 m)
the-f-w2-w1 (f-w2-w1-memo F-w2-w1 F-w1) lambda-w1 (lambda-memo F-w1 alpha)]
                    (+ (* lambda-w1 m-i) (* (- 1 lambda-w1) the-f-w2-w1))))
([w2 w1] (let [F-w2-w1 (get-count-memo all-char-pair-counts (str w2 w1)) F-w1
(get-count-memo all-char-counts w1) m-i (find-m-i-memo w2 m-optimum) the-f-w2-w1
(f-w2-w1-memo F-w2-w1 F-w1) lambda-w1 (lambda-memo F-w1 alpha-optimum)]
                   (+ (* lambda-w1 m-i) (* (- 1 lambda-w1) the-f-w2-w1)))))

(def P-memo (memoize P))

(defn log2 [n]
  (/ (Math/log n) (Math/log 2)))

(defn loop_sum_probs-2 [text] (let [prob (P (second text)  (first text))]
       (if (> (count text) 2) (+ (log2 prob) (loop_sum_probs-2 (rest text)))
                              (log2 prob))))

(defn ell [n] (let [probs (cond (= n 2) (loop_sum_probs-2 letters))] (- (*
probs (/ 1 (count letters))))))

(defn perplexity [n] (Math/pow 2 (ell n)))
```