XYZ

Kristy James

July 13, 2015

1 Langauge Models

1.1 Formula used

1.1.1 Code

```
__author__ = 'Kristy'

import math
################### OVERALL LIKELIHOOD EQNS ########################
def mle_eqn(my_lm, order, fullterm):
    #print('fullterm', fullterm)
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    #print('hist',history)
    return my_lm.ngrams[order].get(fullterm, 0) \
           / \
           my_lm.ngrams[order - 1].get(history,0)

def add_one_eqn(my_lm, order, fullterm):
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    unigram_vocab = my_lm.vocabsize[1]
    potential_ngrams = math.pow(unigram_vocab, order)
    return  (my_lm.ngrams[order].get(fullterm,0) + 1) \
            / \
            (my_lm.ngrams[order - 1].get(history,0) + potential_ngrams)

def add_alpha_eqn(my_lm, order, fullterm, alpha=0.05):
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    unigram_vocab = my_lm.vocabsize[1]
    potential_ngrams = math.pow(unigram_vocab, my_lm.order)
    return  (my_lm.ngrams[order].get(fullterm,0) + alpha) \
            / \
```

```
            (my_lm.ngrams[order - 1].get(history,0) + (alpha * potential_ngram

def good_turing_eqn(my_lm, order, fullterm):
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)

    unigram_vocab = my_lm.vocabsize[1]
    potential_ngrams = math.pow(unigram_vocab, my_lm.order)

    original_count = my_lm.ngrams[order].get(fullterm, 0)
    expected_count = (original_count +1) * \
                     (  my_lm.count_of_counts[order].get(original_count + 1 ,
                      /
                      (  my_lm.count_of_counts[order].get(original_count, pot
    return expected_count \
           / \
           my_lm.ngrams[order - 1][history]


def linear_interpolation(my_lm, order, fullterm):
    #probs and params in ascending order
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    probs_by_order = [my_lm.lm_params.lm_eqn(my_lm, my_lm.order, history[order
    params = my_lm.lm_params.parameters
    if len(params) != len(probs_by_order):
        print("There are more lambda values than probabilities from counts!")
    elif abs(sum(params)-1) > 0.001:
        print("Smoothing params sum to a number other than one!")
    else:
        return sum([params[x] *  probs_by_order[x] for x in range(1, order+1)

def recursive_interpolation(my_lm, order, fullterm):
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    probs_by_order = [my_lm.lm_params.lm_eqn(my_lm, my_lm.order, history[order
    params = my_lm.lm_params.parameter
    if len(params) != len(probs_by_order):
        print("There are more lambda values than probabilities from counts!")
    elif abs(sum(params)-1) > 0.001:
        print("Smoothing params sum to a number other than one!")
    else:
        def recur_int(probs_by_order, params):
            if len(probs_by_order) ==1:
                return probs_by_order.pop() * params.pop()
            else:
```

```
                higher_order = probs_by_order.pop() * params.pop()
                return higher_order + recur_int(probs_by_order, params)
        return recur_int(probs_by_order, params)

def recursive_backoff(my_lm, order, fullterm, d):
    '''This is only a temporary equation - in reality d should be conditioned
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    counts_by_order = [my_lm.ngrams[x].get((history[order-x:]+ word)) for x in
    probs_by_order = [my_lm.lm_params.lm_eqn(my_lm, my_lm.order, history[order
    params = my_lm.lm_params.parameter
    if len(params) != len(probs_by_order):
        print("There are more lambda values than probabilities from counts!")
    elif abs(sum(params)-1) > 0.001:
        print("Smoothing params sum to a number other than one!")
    else:

        def recur_bo(counts_by_order, probs_by_order, d):
            current_count = counts_by_order.pop()
            current_prob = probs_by_order.pop()

            if current_count > 0:
                return d * current_prob

            else:
                return (1-d) * recur_bo(counts_by_order, probs_by_order, d)
        return recur_bo(counts_by_order, probs_by_order, d)

################### SMOOTHING PARAMETER ESTIMATION EQNS ######################
def witten_bell(my_lm, order, fullterm):
    history, word = tuple(fullterm[:-1]), tuple(fullterm[-1])
    fullterm = tuple(fullterm)
    counts_by_order = [my_lm.ngrams[x].get((history[order-x:]+ word)) for x in
    other_futures = [my_lm]
    params = []
    #TODO
    return params

def none_eqn(*args):
    return []
```