1 Part 1

1.1 Helper Functions

First I provide the implementations of two helper functions that are used for computing the linear interpolation and absolute discount smoothing functions. For context, m_model is a hashmap that maps strings to Token objects. Bigrams are simply stored as two unigrams separated by a space. Calling getValue() on the returned object gives the number of times the N-gram appears in the corpus the model was trained on.

```
public double calcMLProb(String want, String given) {
    if (m_model.containsKey(given)) {
        double numerator = m_model.containsKey(given + " " + want) ?
            m_model.get(given + " " + want).getValue():0.0;
        double denomenator = m_model.get(given).getValue();
        return numerator / denomenator;
    }
    else {
        return 1.0/unique_unigram_count;
    }
}

double additiveSmoothedUnigramProb(String token) {
    double count = m_model.containsKey(token) ? m_model.get(token).getValue() :
        0.0;
    double numerator = count + add_delta;
    double denomenator = total_unigram_count + add_delta * unique_unigram_count;

    return numerator/denomenator;
}
```

1.2 Linear Interpolation Smoothing Implementation

```
public double linearInterpolation(String want, String given) {
    double term1 = m_lambda * calcMLProb(want, given);
    double term2 = (1.0 - m_lambda) * additiveSmoothedUnigramProb(want);

    return term1 + term2;
}
```

1.3 Absolute Discounting Smoothing

In the following code "followers" is a hashmap that maps each unigram to a list of the unique unigrams that occur immediately after it in the training corpus. If the model has encountered the unigram then it is guaranteed to be a key in the followers map but it's possible that the

list it maps to is empty. This happens when a word happens to only appear at the end of documents. This happened with the word "andalouse."

```
public double absoluteDiscounting(String want, String given) {
    //Unseen given reverts to unigram model
    if (!m_model.containsKey(given)) {
        return additiveSmoothedUnigramProb(want);
    }
    // number of occurences of w_i-1
    double denom = m_model.get(given).getValue();

    //occurrences of bigram
    double count = m_model.containsKey(given + " " + want) ? m_model.get(given +
        " " + want).getValue() : 0.0;

    // max of occurrences of bigram - delta and 0
    double numert1 = Math.max(0.0, count - m_delta);

    //number of unique followers of w_i-1
    double S = followers.containsKey(given) ? followers.get(given).size() : 0.0;

    //If S is 0, such as when a word only occurs at the end of a document, revert
        to unigram model
    double lambda = S != 0.0 ? (S * m_delta)/denom : 1;

    return (numert1/denom) + lambda * additiveSmoothedUnigramProb(want);
}
```

1.4 Top Ten Words

1.4.1 Linear Interpolation Smoothing

but,0.0838202335
and,0.0610019058
i,0.0535692082
the,0.0535650506
as,0.0333443833
food,0.0265242443
it,0.0183092619
thing,0.0167145
for,0.0166456228
we,0.0153231237

1.4.2 Absolute Discount Smoothing

but,0.0921253211

```
and,0.0643654197
i,0.0564011309
the,0.0539974605
as,0.0366084907
food,0.028944764
it,0.0185250587
thing,0.0184190056
for,0.0173101632
too,0.0166266827
```

1.5 Observations

Interestingly, on this data set both of the smoothing words produce the exact same top ten most probable words. The exact probabilities are of course slightly different but the order is the same.
It is less surprising that almos all of the words would traditionally be considered stopwords. In fact, of the ten words listed, only two do not appear on Smart system's stopword list which was used in MP1. These words are "thing" and "food." Neither of these words are surprising, the first is a relatively common word in causal English and it is intuitive that the word "food" would occur with high probability in a corpus dealing with restaurant reviews.

2 Part 2

2.1 Sampling Procedure Implementation

The following code is used to generate ten fifteen word long sentences from the linear interpolation smoothing bigram language model. The same code is used to generate sentences from the absolute discounting language model simply replacing the relevant function calls. In order to generate sentences from the unigram we simply repeat the code below to generate the first word as many times as we need to complete the sentence.

```
sentences = new ArrayList<>();
  //Generate 10 sentences
      for (int j = 0; j < 10; j++) {
          // Generate the first word from the smoothed unigram model
          ArrayList<String> sen = new ArrayList<String>();
          double prob = Math.random();
          double likelihood = 1.0;
          boolean wordAdded = false;
          for(String token:analyzer.unigrams) {
              prob -= model.additiveSmoothedUnigramProb(token);
              if (prob<=0) {
                  sen.add(token);
                  likelihood = model.additiveSmoothedUnigramProb(token);
                  wordAdded = true;
                  break;
```

```
            }
        }
        if (!wordAdded) {
            sen.add(analyzer.unigrams.get(analyzer.unigrams.size()-1));
        }

    //Sample remaining words from the bigram model
        for (int i = 0; i < 14; i++) {
            prob = Math.random();
            wordAdded = false;
            for(String token:analyzer.unigrams) {
                prob -= model.linearInterpolation(token,
                    sen.get(sen.size()-1));
                if (prob<=0) {
                    likelihood *= model.absoluteDiscounting(token,
                        sen.get(sen.size()-1));
                    sen.add(token);
                    wordAdded = true;
                    break;
                }
            }
            if (!wordAdded) {
                sen.add(analyzer.unigrams.get(analyzer.unigrams.size()-1));
            }
        }
        sen.add(String.valueOf(likelihood));
        sentences.add(sen);
     }

     for (ArrayList<String> sen : sentences) {
        for (String word : sen) {
            absgenWriter.print(word + " ");
        }
        absgenWriter.println("");
     }
     absgenWriter.close();
```

2.2 Generated Sentences

2.2.1 Linear Interpolation Smoothing

- Score: 1.1235820330329619E-28
  the maiz asado con jamon iberico is good door i you ll be the tsunami
- Score: 1.2177970041868793E-35
  those load up were wait i think of mexico drink and two of wait i

- Score: 2.162186479412217E-30
  club the park is long night if you a spici puttanesca sauc to make
- Score: 1.377198011869393E-29
  did nt like a place to own beverag and grit were byob and was so
- Score: 3.13774306704666E-29
  great atmospher is a maxwel st loui you ll find to this with it be
- Score: 2.541096886331702E-32
  my the now thank breakfast is food is veri attent to start out of a
- Score: 1.940104215451961E-31
  is amaz next destin base off the duck here for more sushi restaur to be
- Score: 5.893054360935125E-31
  like we had an offtim it had some were guid to hang over the bar
- Score: 1.0928950590235426E-31
  experi major point we were decent convers and wacki interior is cool the qualiti
- Score: 1.4678665332394338E-31
  the greatest and you were amaz fresh servic is on a prime beef came with

2.2.2 Absolute Discount Smoothing

- Score: 9.092884856801813E-34
  particular yummi sauc which were unhelpfulunfriend in all the menuth cake shake for food usual
- Score: 1.6211419157489424E-35
  them over the bodi a soda and i e not a would nt be frugal
- Score: 4.3477265193119425E-28
  was realli good but even for a littl goat chili such a garlic is nt
- Score: 4.861087599629066E-37
  wing are better than medium but menu but then the walk downtown usual goe formango
- Score: 4.618945798600749E-30
  fact that it had and thought that came out you can i dig are amaz
- Score: 1.9281435973558804E-35
  of of experi in my meal but to loud i love itit s good sauc
- Score: 1.5071173203664913E-26
  told my mouth best deep dish difficult and i ve had the same thing on

- Score: 3.803025186269628E-30
  in a of but i was ordinari mickey d say this place and foh manag
- Score: 1.812614924932321E-30
  you get a grumpi but it was a review the that it s cream i
- Score: 1.9201368389214673E-27
  like the food go to find a great valu for brunch twice the kick to

3 Part 3

3.1 Perplexity Implementation

```
ArrayList<Double> uni_perp = new ArrayList<>();
ArrayList<Double> lin_perp = new ArrayList<>();
ArrayList<Double> abs_perp = new ArrayList<>();

for(Post doc : analyzer.m_reviews) {
    ArrayList<String> sen = doc.getTokens();
    if (sen.size() == 0) {
        continue;
    }

    Double unip = Math.log(model.additiveSmoothedUnigramProb(sen.get(0)));
    Double linp = Math.log(model.additiveSmoothedUnigramProb(sen.get(0)));
    Double absp = Math.log(model.additiveSmoothedUnigramProb(sen.get(0)));
    Double len = (double)sen.size();

    for(int i = 1; i < sen.size(); i++) {
        unip += Math.log(model.additiveSmoothedUnigramProb(sen.get(i)));
        linp += Math.log(model.linearInterpolation(sen.get(i), sen.get(i-1)));
        absp += Math.log(model.absoluteDiscounting(sen.get(i), sen.get(i-1)));
    }

    unip = Math.pow(Math.E, (-1.0/len)*unip);
    linp = Math.pow(Math.E, (-1.0/len)*linp);
    absp = Math.pow(Math.E, (-1.0/len)*absp);

    uni_perp.add(unip);
    lin_perp.add(linp);
    abs_perp.add(absp);
}
```

After creating these lists of perplexities I simply loop over them to calculate average and standard deviation in the standard way.

3.2 Statistics

- Unigram Model
  Average Perplexity: 1697.702549130934
  Standard Deviation: 52214.17073720482

- Linear Interpolation Model
  Average Perplexity: 1292.3483808787837
  Standard Deviation: 57199.64071808456

- Absolute Discounting Model
  Average Perplexity: 1856.1064380479763
  Standard Deviation: 76167.87397254999

3.3 Conclusions

From the perplexity scores we can conclude that the linear interpolation model on average has the lowest perplexity on a given document in the test folder. However, this does not mean in general that the linear interpolation model is the best choice for all applications concerning this data. For example, the unigram model had a lower standard deviation and in general given only the means and standard deviations we cannot tell what the distribution as a whole looks like.
There are a few particularly interesting things we can note from the perplexity scores. The primary one is that the unigram model actually performed better in terms of average perplexity than the absolute discounting model and had a lower standard deviation than either of the alternatives. This is unexpected because in general we expect N-gram models to improve in performance as N increases.
In general the models have very large standard deviations compared to their averages. On closer inspection of the perplexities for individual documents we can see that the median value for each of the models is actually in the hundreds rather than the thousands and the averages and standard deviations are inflated by a few extremely unlikely individual documents with perplexities in the hundreds of thousands.