In [293]:
```
from numpy import array

import pandas as pd
import numpy as np
import re

import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import CountVectorizer

START = "<s> "
STOP = "</s>"
UNK = "<UNK>"
```

In [318]:
```
# import libraries needed, read the dataset
import nltk, re, pprint, string
from nltk import word_tokenize, sent_tokenize

def add_sentence_tokens(sentences):
    return ['{}{} {}'.format(START, s, STOP) for s in sentences]

def replace_tringletons(tokens):
    vocab = nltk.FreqDist(tokens)
    return [token if vocab[token] >= 3 else UNK for token in tokens]

def preprocess(sentences):
    sentences = add_sentence_tokens(sentences)
    tokens = ' '.join(sentences).split(' ')
    tokens = replace_tringletons(tokens)
    return tokens

```

In [359]:
```
# file = open('/Users/nehakardam/Documents/UWclasses /CSE NLP/A3/data_A
#file = open('/Users/nehakardam/Documents/UWclasses /CSE NLP/A3/data_A3
file = open('/Users/nehakardam/Documents/UWclasses /CSE NLP/A3/data_A3/
train = [l.strip() for l in file.readlines()]

```

In [361]:
```
unigram = preprocess(train,1)
bigrams = nltk.ngrams(unigram, 2)
trigrams = nltk.ngrams(unigram, 3)
```

In [362]:
```
freq_uni = nltk.FreqDist(unigram)
freq_bi = nltk.FreqDist(bigrams)
freq_tri = nltk.FreqDist(trigrams)
```

In [363]:
```
len(freq_uni)
```

Out[363]: 9599

In [364]:
```
lm = LanguageModel(train, 1)
print("Vocabulary size: {}".format(len(lm.vocab)))
perplexity = lm.perplexity(train)
print("Model perplexity: {:.3f}".format(perplexity))
```

```
Vocabulary size: 9599
Model perplexity: 537.947
```

In [367]:
```
lm = LanguageModel(train, 2)
print("Vocabulary size: {}".format(len(lm.vocab)))
perplexity = lm.perplexity(train)
print("Model perplexity: {:.3f}".format(perplexity))
```

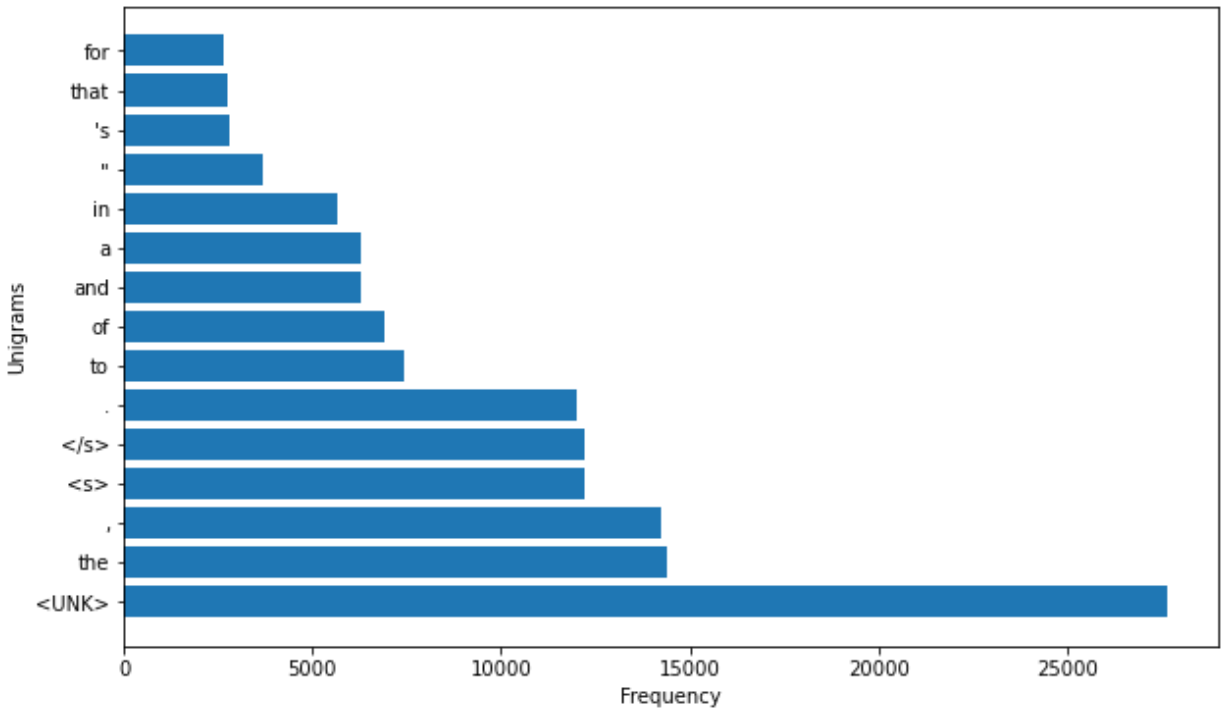
```
Vocabulary size: 9599
Model perplexity: 670.571
```

In [366]:
```
lm = LanguageModel(train, 3)
print("Vocabulary size: {}".format(len(lm.vocab)))
perplexity = lm.perplexity(train)
print("Model perplexity: {:.3f}".format(perplexity))
```

```
Vocabulary size: 9599
Model perplexity: 1592.478
```

In [357]:
```
# import math

# def compute_prob(freq, N):
#     print ("processing")
#     return { ngram: count / N for ngram, count in freq.items() }

# def perplexity(ngrams, freq):
#     N = len(unigram)
#     prob = compute_prob(freq, N)
#     probabilities = [prob[ngram] for ngram in ngrams]
#     print(probabilities)
#     return math.exp((-1/N) * sum(map(math.log, probabilities)))
```

In [358]:
```
uni_k = list(freq_uni.keys())
uni_v= list(freq_uni.values())
bi_k = list(freq_bi.keys())
bi_v= list(freq_bi.values())
tri_k = list(freq_tri.keys())
tri_v= list(freq_tri.values())

bi_k_str = ['_'.join(i) for i in bi_k]
tri_k_str = ['_'.join(i) for i in tri_k]
unigram_count_dict = dict(zip(uni_k,uni_v))
bigram_count_dict = dict (zip(bi_k_str,bi_v))
trigram_count_dict = dict (zip(tri_k_str,tri_v))

unigram_data_items = unigram_count_dict.items()
unigram_data_list = list(unigram_data_items)
unigram_df = pd.DataFrame(unigram_data_list,columns=['Unigram','count'])
unigram_df = unigram_df.sort_values(by=['count'],ascending=False,ignore_
plt.rcParams["figure.figsize"] = (10,6)
y=unigram_df['Unigram']
x=unigram_df['count']
plt.barh(y, x)
plt.ylabel("Unigrams")
plt.xlabel("Frequency")
# plt.title("Bigram Frequency Distribution")
plt.show()
```



In [360]:

```
#!/bin/env python

import argparse
from itertools import product
import math
import nltk
from pathlib import Path

#!/bin/env python

import nltk

SOS = "<s> "
EOS = "</s>"
UNK = "<UNK>"

def add_sentence_tokens(sentences, n):
    """Wrap each sentence in SOS and EOS tokens.

    For n >= 2, n-1 SOS tokens are added, otherwise only one is added.

    Args:
        sentences (list of str): the sentences to wrap.
        n (int): order of the n-gram model which will use these senten
    Returns:
        List of sentences with SOS and EOS tokens wrapped around them.

    """
    sos = SOS * (n-1) if n > 1 else SOS
    return ['{}{} {}'.format(sos, s, EOS) for s in sentences]

def replace_singletons(tokens):
    """Replace tokens which appear only once in the corpus with <UNK>.

    Args:
        tokens (list of str): the tokens comprising the corpus.
    Returns:
        The same list of tokens with each singleton replaced by <UNK>.

    """
    vocab = nltk.FreqDist(tokens)
    return [token if vocab[token] >= 3 else UNK for token in tokens]

def preprocess(sentences, n):
    """Add SOS/EOS/UNK tokens to given sentences and tokenize.

    Args:
        sentences (list of str): the sentences to preprocess.
        n (int): order of the n-gram model which will use these senten
    Returns:
        The preprocessed sentences, tokenized by words.

    """
    sentences = add_sentence_tokens(sentences, n)
    tokens = ' '.join(sentences).split(' ')
    tokens = replace_singletons(tokens)
```

```
    return tokens


def load_data(data_dir):
    """Load train and test corpora from a directory.

    Directory must contain two files: train.txt and test.txt.
    Newlines will be stripped out.

    Args:
        data_dir (Path) -- pathlib.Path of the directory to use.

    Returns:
        The train and test sets, as lists of sentences.

    """
    train_path = data_dir.joinpath('train.txt').absolute().as_posix()
    test_path  = data_dir.joinpath('test.txt').absolute().as_posix()

    with open(train_path, 'r') as f:
        train = [l.strip() for l in f.readlines()]
    with open(test_path, 'r') as f:
        test = [l.strip() for l in f.readlines()]
    return train, test


class LanguageModel(object):
    """An n-gram language model trained on a given corpus.

    For a given n and given training corpus, constructs an n-gram langu
    model for the corpus by:
    1. preprocessing the corpus (adding SOS/EOS/UNK tokens)
    2. calculating (smoothed) probabilities for each n-gram

    Also contains methods for calculating the perplexity of the model
    against another corpus, and for generating sentences.

    Args:
        train_data (list of str): list of sentences comprising the trai
        n (int): the order of language model to build (i.e. 1 for unigi
        laplace (int): lambda multiplier to use for laplace smoothing

    """

    def __init__(self, train_data, n, laplace=1):
        self.n = n
        self.laplace = laplace
        self.tokens = preprocess(train_data, n)
        self.vocab  = nltk.FreqDist(self.tokens)
        self.model  = self._create_model()
        self.masks  = list(reversed(list(product((0,1), repeat=n))))

    def _smooth(self):
        """Apply Laplace smoothing to n-gram frequency distribution.

        Here, n_grams refers to the n-grams of the tokens in the traini
        while m_grams refers to the first (n-1) tokens of each n-gram.
```

```
114
115         Returns:
116             dict: Mapping of each n-gram (tuple of str) to its Laplace-
117             probability (float).
118
119         """
120         vocab_size = len(self.vocab)
121
122         n_grams = nltk.ngrams(self.tokens, self.n)
123         n_vocab = nltk.FreqDist(n_grams)
124
125         m_grams = nltk.ngrams(self.tokens, self.n-1)
126         m_vocab = nltk.FreqDist(m_grams)
127
128         def smoothed_count(n_gram, n_count):
129             m_gram = n_gram[:-1]
130             m_count = m_vocab[m_gram]
131             return (n_count + self.laplace) / (m_count + self.laplace *
132
133         return { n_gram: smoothed_count(n_gram, count) for n_gram, coun
134
135     def _create_model(self):
136         """Create a probability distribution for the vocabulary of the
137
138         If building a unigram model, the probabilities are simple relat
139         of each token with the entire corpus.
140
141         Otherwise, the probabilities are Laplace-smoothed relative freq
142
143         Returns:
144             A dict mapping each n-gram (tuple of str) to its probabili
145
146         """
147         if self.n == 1:
148             num_tokens = len(self.tokens)
149             return { (unigram,): count / num_tokens for unigram, count
150         else:
151             return self._smooth()
152
153     def _convert_oov(self, ngram):
154         """Convert, if necessary, a given n-gram to one which is known
155
156         Starting with the unmodified ngram, check each possible permuta
157         with each index of the n-gram containing either the original to
158         when the model contains an entry for that permutation.
159
160         This is achieved by creating a 'bitmask' for the n-gram tuple,
161         each flagged token for <UNK>. Thus, in the worst case, this fun
162         possible n-grams before returning.
163
164         Returns:
165             The n-gram with <UNK> tokens in certain positions such that
166             contains an entry for it.
167
168         """
169         mask = lambda ngram, bitmask: tuple((token if flag == 1 else "<
170
```

```
        ngram = (ngram,) if type(ngram) is str else ngram
        for possible_known in [mask(ngram, bitmask) for bitmask in sel
            if possible_known in self.model:
                return possible_known

    def perplexity(self, test_data):
        """Calculate the perplexity of the model against a given test

        Args:
            test_data (list of str): sentences comprising the training
        Returns:
            The perplexity of the model as a float.

        """
        test_tokens = preprocess(test_data, self.n)
        test_ngrams = nltk.ngrams(test_tokens, self.n)
        N = len(test_tokens)

        known_ngrams  = (self._convert_oov(ngram) for ngram in test_ng
        probabilities = [self.model[ngram] for ngram in known_ngrams]

        return math.exp((-1/N) * sum(map(math.log, probabilities)))

    def _best_candidate(self, prev, i, without=[]):
        """Choose the most likely next token given the previous (n-1)

        If selecting the first word of the sentence (after the SOS tok
        the i'th best candidate will be selected, to create variety.
        If no candidates are found, the EOS token is returned with pro

        Args:
            prev (tuple of str): the previous n-1 tokens of the senten
            i (int): which candidate to select if not the most probabl
            without (list of str): tokens to exclude from the candidat
        Returns:
            A tuple with the next most probable token and its correspo

        """
        blacklist  = ["<UNK>"] + without
        candidates = ((ngram[-1],prob) for ngram,prob in self.model.it
        candidates = filter(lambda candidate: candidate[0] not in blac
        candidates = sorted(candidates, key=lambda candidate: candidat
        if len(candidates) == 0:
            return ("</s>", 1)
        else:
            return candidates[0 if prev != () and prev[-1] != "<s>" el

    def generate_sentences(self, num, min_len=12, max_len=24):
        """Generate num random sentences using the language model.

        Sentences always begin with the SOS token and end with the EOS
        While unigram model sentences will only exclude the UNK token,
        will also exclude all other words already in the sentence.

        Args:
            num (int): the number of sentences to generate.
            min_len (int): minimum allowed sentence length.
```

```
            max_len (int): maximum allowed sentence length.
        Yields:
            A tuple with the generated sentence and the combined probal
            (in log-space) of all of its n-grams.

        """
        for i in range(num):
            sent, prob = ["<s>"] * max(1, self.n-1), 1
            while sent[-1] != "</s>":
                prev = () if self.n == 1 else tuple(sent[-(self.n-1):]
                blacklist = sent + (["</s>"] if len(sent) < min_len els
                next_token, next_prob = self._best_candidate(prev, i, w
                sent.append(next_token)
                prob *= next_prob

                if len(sent) >= max_len:
                    sent.append("</s>")

            yield ' '.join(sent), -1/math.log(prob)

```

Linear Interpolation Smoothing

Reference:https://github.com/ErolOZKAN-/Language-Modelling/blob/master/src/ngram.py (https://github.com/ErolOZKAN-/Language-Modelling/blob/master/src/ngram.py)

In [379]:

```
def ppl_interpolation(sentences, lambda_set):
    '''Returns the interpolated perplexity of the given list of sentenc
    sentences = unigram
    counter = 0
    tmp = 0
    for i in range(len(sentences)):
        sentence = sentences[i]
        for j in range(len(sentence) - 2):
            trigram_unit = (sentence[j], sentence[j + 1], sentence[j +
            if trigram_unit in trigram_prob:
                tri_gram_prob = trigram_prob[trigram_unit]
            else:
                tri_gram_prob = 0

            bigram_unit = (sentence[j], sentence[j + 1])
            if bigram_unit in bigram_prob:
                bigram_prob = bigram_prob[bigram_unit]
            else:
                bigram_prob = 0

#             unigram_prob = one_gram_add_one_prob.get(sentence[j], 0)

            prob = lambda_set[0] * unigram_prob + lambda_set[1] * bigra
            if prob == 0:
                tmp += 0
            else:
                tmp += math.log(prob, 2)
            counter += 1
    # entropy = prob of each token / number of tokens
    entropy = -1 / counter * tmp
    perplexity = math.pow(2, entropy)
    return perplexity
```

In [380]:
```
print("\n------INTERPOLATION--------")
lambda_set = []
lambda_set.append([0.5, 0.3, 0.2])
lambda_set.append([0.8, 0.1, 0.1])
lambda_set.append([0.1, 0.8, 0.1])
lambda_set.append([0.1, 0.1, 0.8])
lambda_set.append([0.6, 0.2, 0.2])
lambda_set.append([0.2, 0.6, 0.2])
lambda_set.append([0.2, 0.2, 0.6])
lambda_set.append([0.4, 0.3, 0.3])
lambda_set.append([0.3, 0.4, 0.3])
lambda_set.append([0.3, 0.3, 0.4])
lambda_set.append([0.2, 0.4, 0.4])
lambda_set.append([0.4, 0.2, 0.4])
lambda_set.append([0.4, 0.4, 0.2])
lambda_set.append([0.1, 0.4, 0.5])
lambda_set.append([0.1, 0.3, 0.6])
lambda_set.append([0.1, 0.2, 0.7])
lambda_set.append([0.05, 0.15, 0.8])
lambda_set.append([0.05, 0.05, 0.9])
for s in lambda_set:
    print("Lambda_Set: ", lambda_set)
    ppl_score = ppl_interpolation(train, s)
    print("perplexity score of test data:", ppl_score)
```

```
------INTERPOLATION--------
Lambda_Set:  [[0.5, 0.3, 0.2], [0.8, 0.1, 0.1], [0.1, 0.8, 0.1], [0.1, 0.
1, 0.8], [0.6, 0.2, 0.2], [0.2, 0.6, 0.2], [0.2, 0.2, 0.6], [0.4, 0.3, 0.
3], [0.3, 0.4, 0.3], [0.3, 0.3, 0.4], [0.2, 0.4, 0.4], [0.4, 0.2, 0.4],
[0.4, 0.4, 0.2], [0.1, 0.4, 0.5], [0.1, 0.3, 0.6], [0.1, 0.2, 0.7], [0.0
5, 0.15, 0.8], [0.05, 0.05, 0.9]]

---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-380-63efe8819287> in <module>
     21 for s in lambda_set:
     22     print("Lambda_Set: ", lambda_set)
---> 23     ppl_score = ppl_interpolation(train, s)
     24     print("perplexity score of test data:", ppl_score)

<ipython-input-379-fa79245e1b47> in ppl_interpolation(sentences, lambda_s
et)
      8         for j in range(len(sentence) - 2):
      9             trigram_unit = (sentence[j], sentence[j + 1], sentenc
e[j + 2])
---> 10             if trigram_unit in trigram_prob:
     11                 tri_gram_prob = trigram_prob[trigram_unit]
     12             else:

NameError: name 'trigram_prob' is not defined
```

In [ ]:
```
1
```