# ULL Mini-Project: Sampling a TSG for Numerals

**Tran Cong Nguyen**
10867481
University of Amsterdam
cong.tran@student.uva.nl

**Lautaro Quiroz**
10849963
University of Amsterdam
lautaro.quiroz@student.uva.nl

**Roger Wechsler**
10850007
University of Amsterdam
roger.wechsler@student.uva.nl

## Abstract

In this project we extend the second assignment, by taking a deeper look into the Metropolis-Hastings sampling algorithm and how the likelihood of the dataset is computed. We begin by presenting a proper likelihood calculation by using inside probabilities, and present a comparison to previous results. Furthermore, by using inside probabilities, and thus, consider every possible parsing tree, we analyze how initial grammar ambiguity can impact results and affect the grammar refinement process.

## 1 Introduction

This report presents an extension to the second assignment, in which we used the Metropolis-Hastings algorithm in order to train a randomly initialized Tree Substitution Grammar (TSG) to better represent a corpus of numerals extracted from the Penn World Street Journal (Penn WSJ) treebank[1]. A very broad overview of sampling TSGs for general language data is provided in [1].

For the Metropolis-Hastings algorithm, in each iteration we must compute the likelihood of the data before and after making a random change to the TSG, and decide whether to keep this change or to discard it. Originally, the calculation of the likelihood of the data was an approximation carried out by using the highest probability of the derivations (Viterbi). We now extend this idea and compute the likelihood of the data by considering the inside probability of the strings, which considers all possible derivations of a string and not only the most likely. We expect that having a better estimation of the data likelihood will lead to a better final TSG. Later on, we analyse results obtained when using different initial Context Free Grammar (CFG) which differ on the level of ambiguity they present.

## 2 Methodology

### 2.1 Numeral extraction

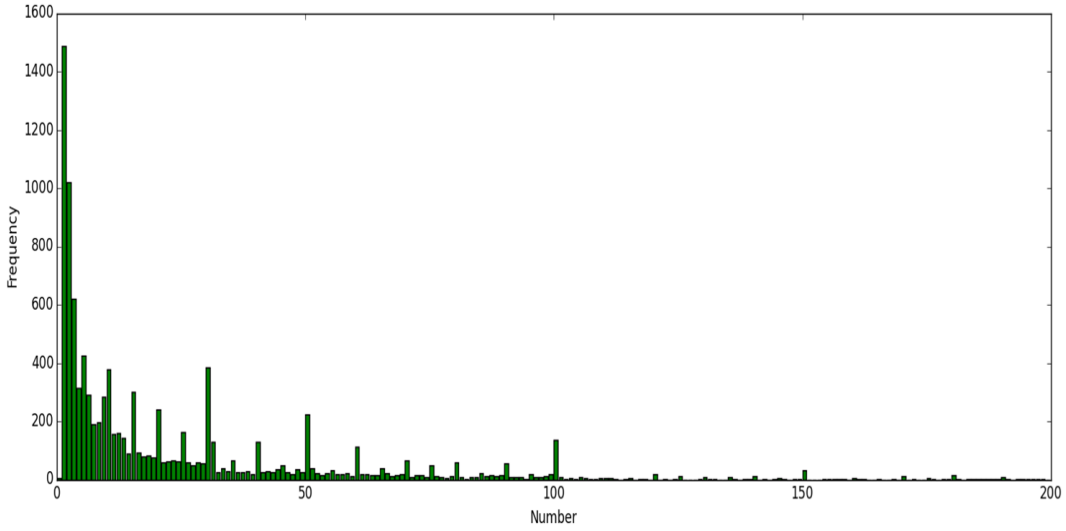
The numerals are obtained from the Penn WSJ treebank. We used the version without traces, marks, and punctuations, up to length 40. As the numeral data in the treebank is marked with ”CD” tags,

[1] https://www.cis.upenn.edu/~treebank

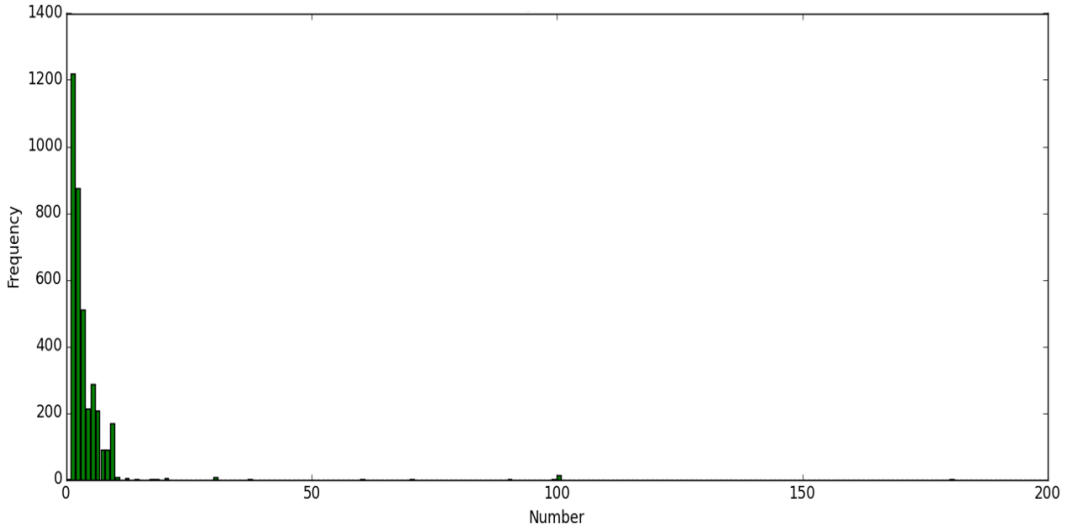
the first phase of the extraction process involves looking for all "CD" tags and extracting their values. Next, decimal numbers are dropped as we are only interested in natural numbers. Finally, we normalize the data by converting alphabetic numbers like *twenty-four* into their number forms as digits (*24*). Numbers representing years or decades are also converted. As composite numbers like *five thousand* consist of multiple "CD" tags, an additional preprocessing step is required to ensure the correct parsing of such numbers.

Below are a few examples of the extracted values inside "CD" tags and their normalized values:

- "12" $\rightarrow 12$.
- "two hundred"$\rightarrow 200$.
- "1.45 million"$\rightarrow 1,450,000$.
- "early-1980s"$\rightarrow 1980$.




(a) Natural numbers in number form.




(b) Natural numbers in alphabetic form.

Figure 1: Distribution of natural numbers in $[0, 200)$ extracted from Penn WSJ treebank.

Figure 1 shows the frequencies of natural numbers in range $[0, 200)$ from the Penn WSJ treebank in number form (1a) and alphabetic form (1b). We can see from the two sub-figures that these distributions have the shape of a negative logarithm function. It is not as clear in the alphabet case though because the amount of data is relatively insufficient. All in all, they both show a trend that small numbers between 0 and 10 tend to have high frequencies, compared to higher numbers.

Figure 2 extends the range to $[0, 4\,000)$, and with both forms combined into a single histogram. This figure shows another interesting property of the distribution. It has a high peak at around $2\,000$. It is because numbers representing years and decades such as "1980" mostly concentrate around this value.

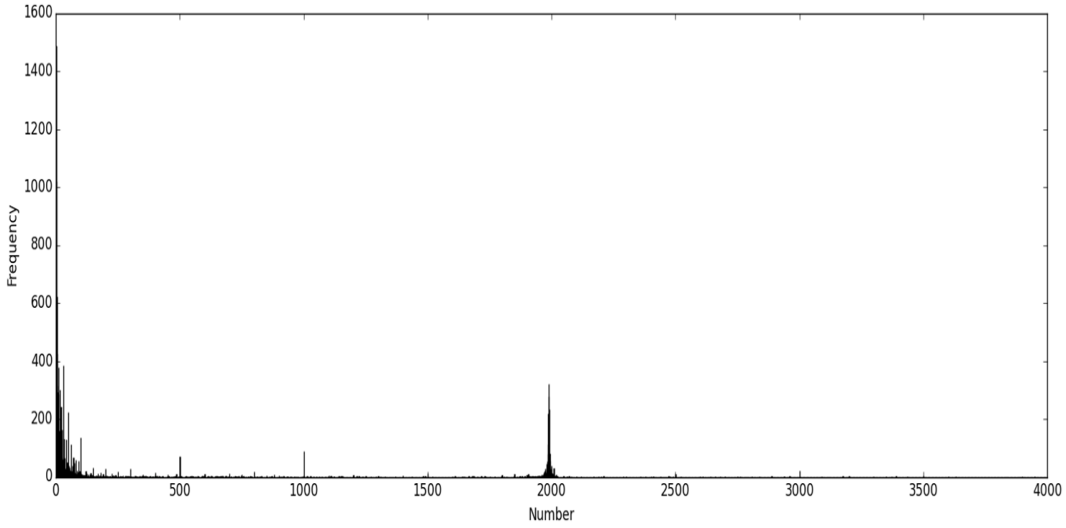



Figure 2: Natural numbers (both alphabetic and digit forms) in $[0, 4\,000)$ in Penn WSJ treebank.

### 2.2 Procedure

After extracting the numerals from the WSJ treebank, we assign a parse to each string using the CDEC decoder [2]. The CDEC decoder provides a framework for parallel parsing of a source and target langauge used in machine translation. By duplicating the grammars for both languages, it can also be used to parse monolingual data. The framework is optimal for our project as it provides a fast parsing approach using CFG grammars and as the likelihood of the string (i. e. the inside probability of the root symbol of the parse chart) can easily be returned by the parser.

Once each numeral has been assigned a random parse from the initial CFG, the parses are randomly split into elementary trees by assigning a random substitution site marker. This is done by deciding whether to split a tree at each possible non-terminal with a small probability (i. e. 0.3 for our experiments).

We use this initial TSG as the input to the Metropolis-Hastings algorithm. Algorithm 1 shows the pseudocode for that sampling method. It simply makes a random change to the data set by adding or removing a substitution site marker, which results in splitting or combining elementary trees. Based on the newly introduced changes, the likelihood of the dataset is computed. If it is higher than the one of the dataset without the changes, i. e. if the changes increase the likelihood of the data, the changes are always accepted. Otherwise the changes might be accepted nevertheless with a probability that is the ratio of the two likelihoods.

**Data**: Initial TSG
**repeat**
  Introduce a random change;
  Compute the new likelihood with and without the change;
  **if** *newLikelihood >oldLikelihood or with probability newLikelihood / oldLikelihood* **then**
    Apply change;
  **end**
**until** *iteration limit achieved*;

**Algorithm 1:** Metropolis-Hastings algorithm

When computing the likelihood of the dataset, we use the CDEC decoder that returns the likelihood of the strings. For that step, the TSG has to be converted into a CFG that can be used for parsing the data. The TSG can be transformed into a CFG by constructing rules that write from the root node to the leaf nodes. The internal structure of the elementary trees is thus lost, but for the actual parsing only the leaf nodes are important. The parameters, i. e. the probabilities of the rules are set using a maximum likelihood approach, i. e. counting how often the rules occur in the dataset from the respective root nodes.

The probability of a derivation $\psi$ is the product of all rules used for that derivation. The likelihood of a string $w$ is then the sum over all possible derivations for that string:

$$p(\psi) = \prod_{r \in R} p(r)^{f_r(\psi)}$$

$$p(w) = \sum_{\psi \in \Psi} p(\psi)$$

Using the inside-outside framework, the likelihood corresponds to the inside probability of the root node in the chart. In contrast, the Viterbi calculation implies getting the probability of the most probable derivation of a string:

$$p^*(w) = \max_{\psi \in \Psi} p(\psi)$$

### 2.3 Grammar

We compared three different grammars that we use for parsing the dataset in the beginning. These grammars present various levels of ambiguity. In the first case we used a deterministic grammar, where each numeral has only one parse tree associated; then we moved on to a grammar with more than one parse tree per string; and finally, we tried a "highly ambiguous" grammar, which allows even more possible parses for one numeral. As the computation of the inside probabilities takes into account all possible trees, we are interested in how deterministic and redundant grammars might affect the likelihood of the dataset.

Table 1, Table 2, and Table 3 show the grammar rules of the three grammars.

| Root | Derivation |
|---|---|
| S | [0] or [1] or ... or [9] |
| S | [1] [P] or [2] [P] or ... or [9] [P] |
| P | [0] or [1] or ... or [9] |
| P | [0] [P] or [1] [P] or ... or [9] [P] |

Table 1: Deterministic grammar

| Root | Derivation |
|---|---|
| S | [D] [D] or [S1] [S2] |
| S1 | [NZ] [S2] or [NZ] |
| S2 | [D] [S2] or [D] |
| D | [0] or [1] or ... or [9] |
| NZ | [1] or [2] or ... or [9] |

Table 2: Ambiguous grammar

| Root | Derivation |
|---|---|
| S | [Z] or [NZ] or [S1] [S2] |
| S1 | [NZ] or [NZ] [S2] or [S1] [S2] |
| S2 | [Z] or [NZ] or [Z] [Z] or [Z] [NZ] or [NZ] [Z] or [NZ] [NZ] |
| NZ | [1] or [2] or ... or [9] |
| Z | [0] |

Table 3: “Highly ambiguous” grammar

## 3 Experiments

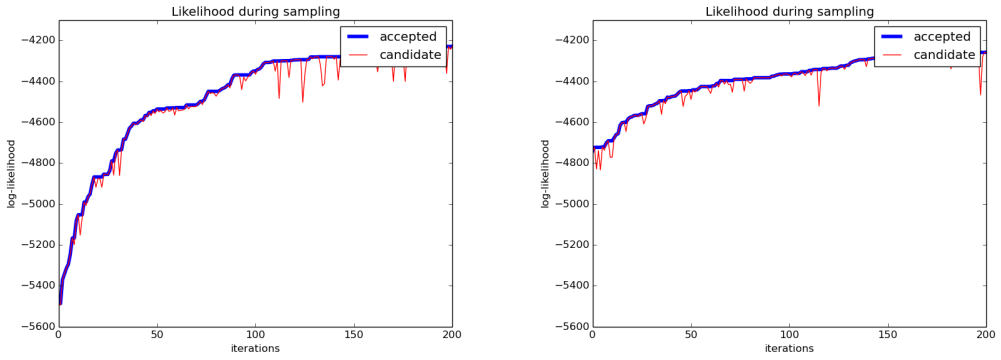
In these experiments, we initialize the TSG by placing the substitution markers with a probability of 0.3, and run the Metropolis-Hastings algorithm for 600 iterations.

The CDEC decoder was used to get initial parses of the corpus, as well as the computations of the Viterbi and inside probabilities.

### 3.1 Viterbi vs Inside probabilities comparison

Figure 3 shows a comparison between the Viterbi and Inside method of the likelihood evolution while running Metropolis-Hastings.

As it is shown in the graphs, the likelihood that is calculated using the sum of all derivations of a string is higher during the initial iterations, but towards the end, both algorithms seem to converge to the same value. We expect it to be higher or equal since summing up more than just the best probability will always be at least equal or higher. If we take a look at the final state of the grammar we can see how almost each numeral has its own rule, which leads to both the Viterbi and the inside method to compute the same likelihood value for that string.



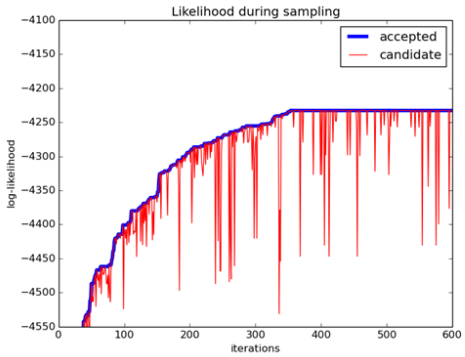

(a) Likelihood during sampling with Viterbi

(b) Likelihood during sampling with Inside

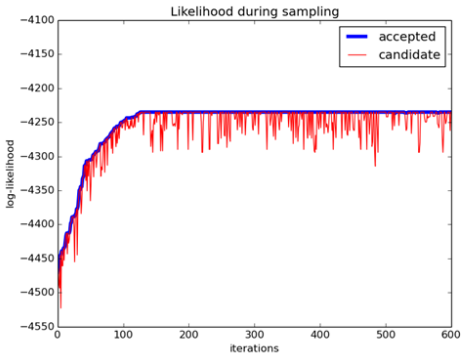
Figure 3: Comparison of likelihood during training between Viterbi and Inside probabilities

### 3.2 Grammar ambiguity comparison

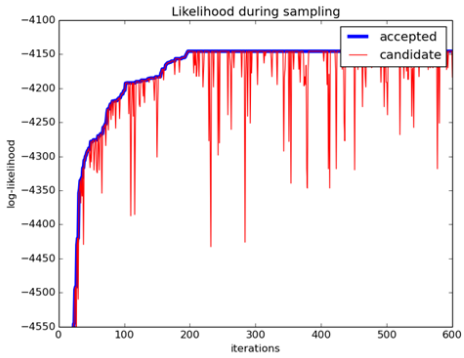
Figure 4 shows the likelihood of the entire dataset per Metropolis-Hastings iteration while using a deterministic, ambiguous, and highly ambiguous grammar. As we can see, the three cases converge to a similar final value; the highly ambiguous grammar seems to be performing better than the rest, however. The iteration number in which the likelihoods stabilize might be, mostly, due to the random initialization of the TSGs rather than a grammar ambiguity effect.




(a) Likelihood during sampling for the deterministic grammar



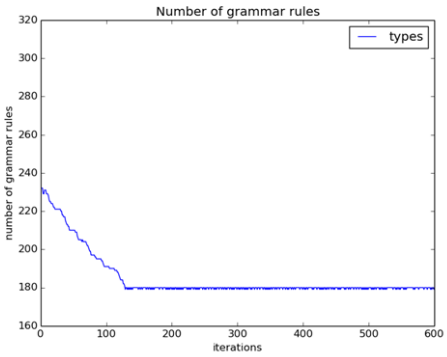

(b) Likelihood during sampling for the half ambiguous grammar



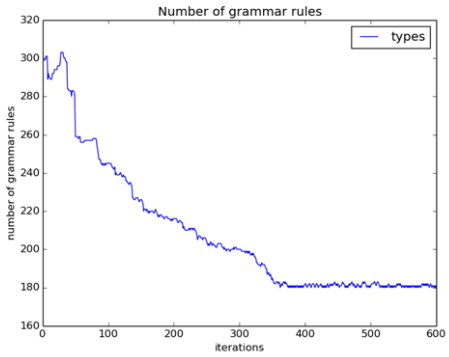

(c) Likelihood during sampling for the ambiguous grammar

Figure 4: Comparison of likelihood during training between grammars

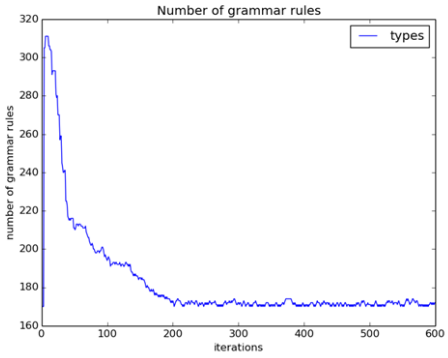
Figure 5 and Figure 6 show the number of used rules and number of existing elementary trees per iteration of the sampling algorithm when using the three different grammars. The curves show the natural trend of grammar refinement algorithms, decreasing over time and reaching a convergence point after a few iterations. The convergence point might be the result of the TSG initialization.




(a) Number of used rules during sampling for the deterministic grammar

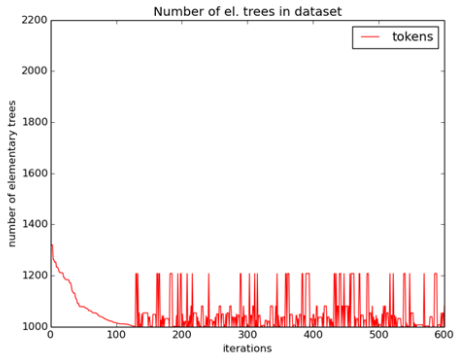



(b) Number of used rules during sampling for the half ambiguous grammar

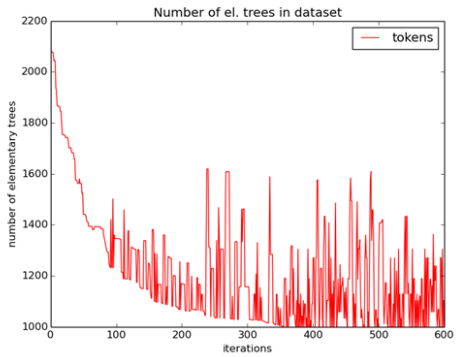



(c) Number of used rules during sampling for the highly ambiguous grammar

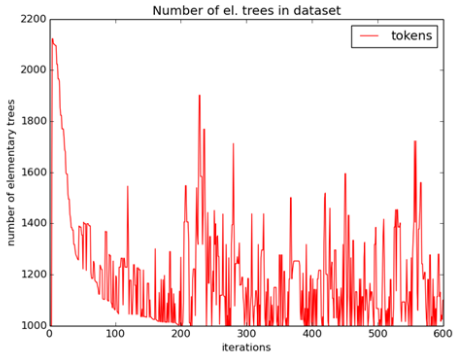
Figure 5: Comparison of the number of used rules during training between grammars




(a) Number of trees during sampling for the deterministic grammar



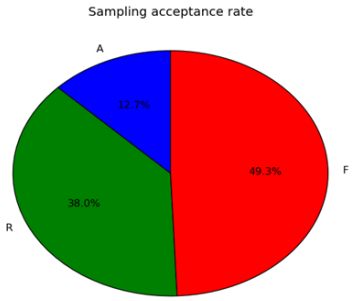

(b) Number of trees during sampling for the half ambiguous grammar



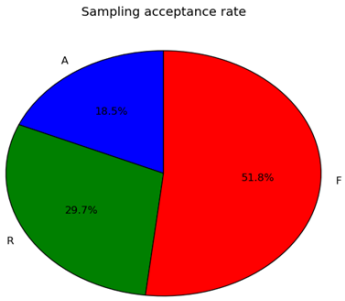

(c) Number of trees during sampling for the highly ambiguous grammar

Figure 6: Comparison of the number of trees during training between grammars

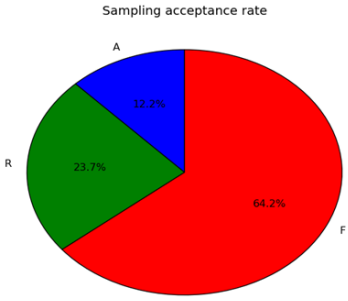
Figure 7 shows the overall percentage of accepted changes (those which produced a dataset likelihood improvement), forced changed (those which even when they produced a lower dataset likelihood, were accepted with a probability $p < newLikelihood \ / \ oldLikelihood$), and rejected changes. As it can be seen in the graphs, while the percentage of accepted changes does not vary significantly, the forced changes increases with ambiguity. As changes produce a new likelihood close to the old one, then the probability of accepting the change will tend to 1. This is most likely to occur during the iterations previous to convergence.




(a) Change acceptance rate for the deterministic grammar




(b) Change acceptance rate for the half ambiguous grammar




(c) Change acceptance rate for the highly ambiguous grammar

Figure 7: Comparison of the change acceptance rates between grammars

### 3.3 Sampling from the grammars

In this experiment, we sample 200 000 natural numbers from the true distribution obtained directly from the treebank as well as from the trained grammars to see how effective each type of grammar is in the task of generating natural numbers.

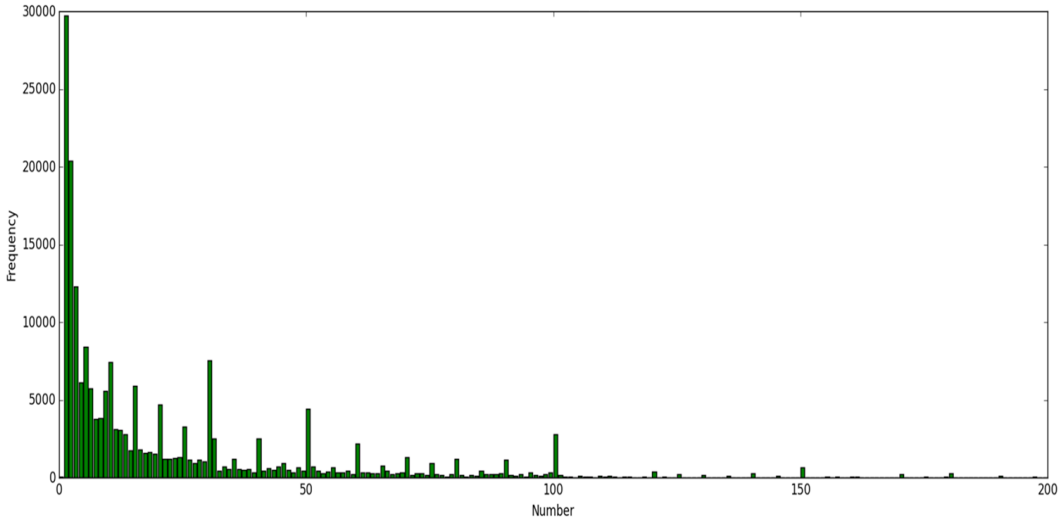
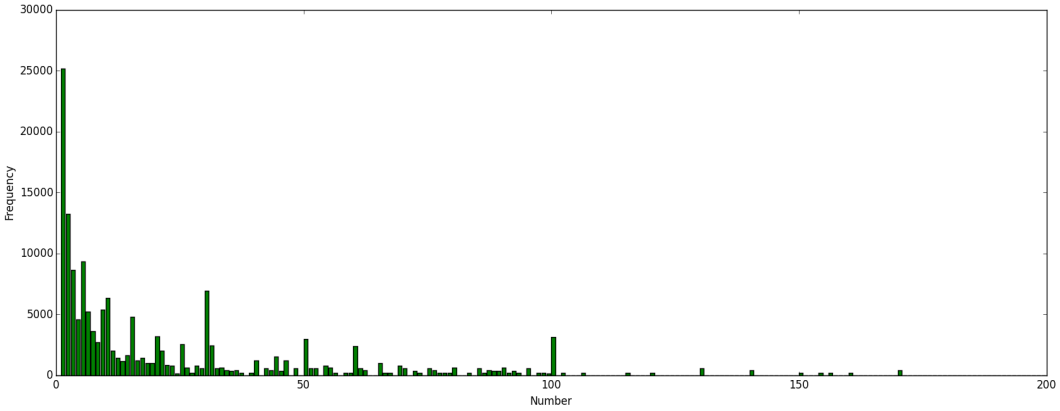


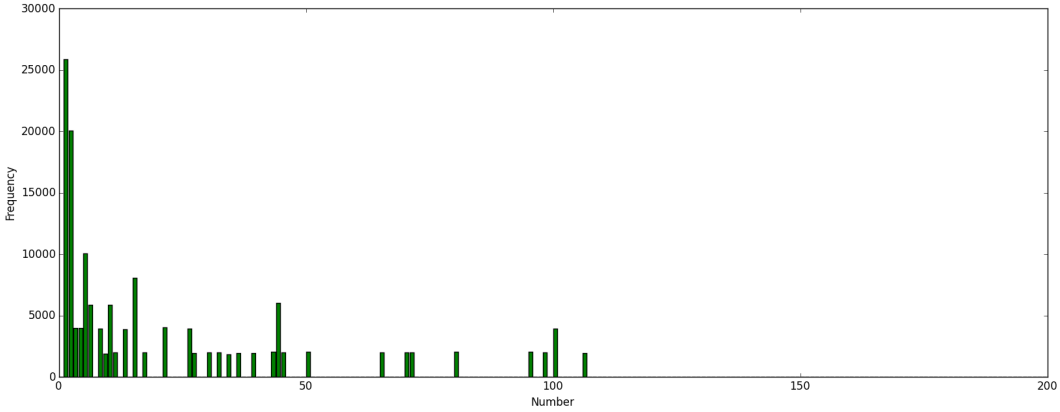

Figure 8: $200\,000$ natural numbers in range $[0, 200)$ sampled from the true distribution.

Figure 8 shows the distribution of $200\,000$ natural numbers sampled from the true distribution, which is the sum of those in Figures 1a and 1b. We can see the numbers generated follow the true distribution really closely and their histogram also has the shape of a negative logarithm function.

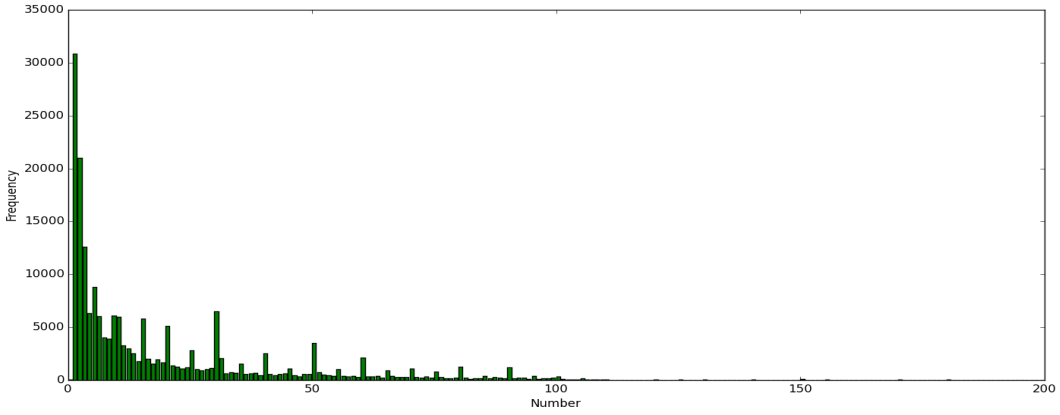
Figure 9 shows $200\,000$ numbers sampled from the three grammars that we work throughout our project. All three grammars capture the characteristics of the true distribution having really high peak towards zero. Both the highly ambiguous grammar in Figure 9a and deterministic grammar in Figure 9c produce distributions following the true distribution closely. However, as the deterministic grammar favors shorter elementary trees, the peak at number $1$ of the histogram this grammar induces has frequency of approximately $30\,000$, which is closer to what Figure 8 of the true distribution than in the case of the highly ambiguous grammar, having frequency of $1$ only about $25\,000$. On the other hand, the half ambiguous grammar in Figure 9b is not applicable for the task, as the generated distribution only has peaks at a few numbers, while having almost zero frequency for the rest.




(a) Highly ambiguous grammar.




(b) Half ambiguous grammar.




(c) Deterministic grammar.

Figure 9: $200\,000$ natural numbers in range $[0, 200)$ sampled from the three grammars.

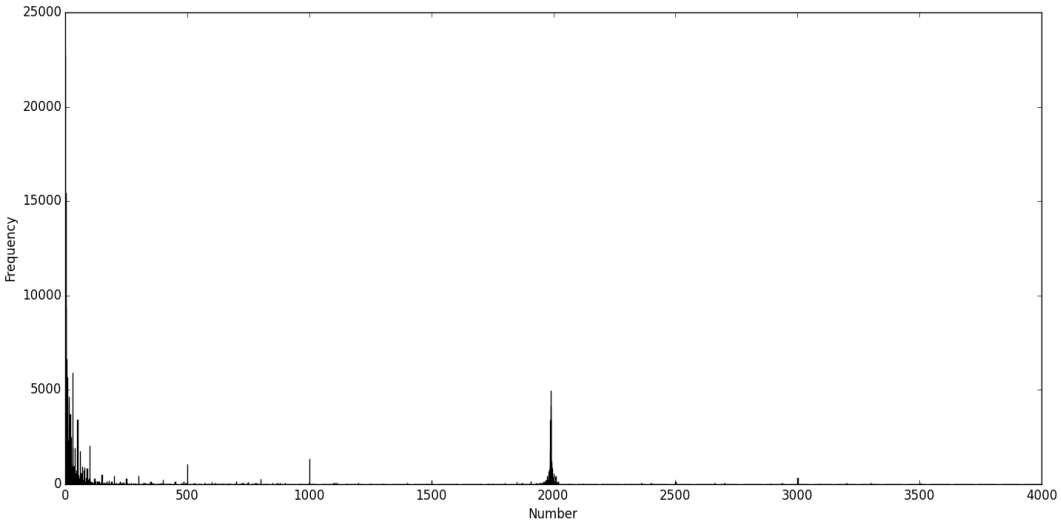
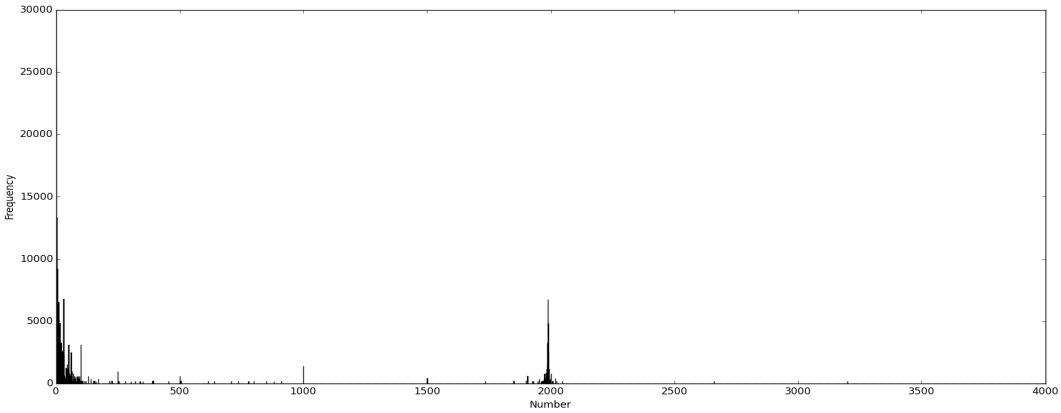


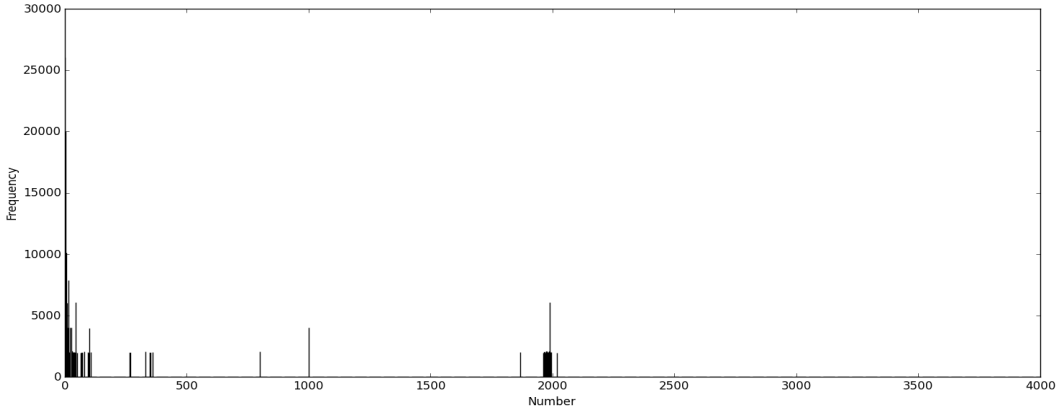

Figure 10: $200\,000$ natural numbers in range $[0, 4000)$ sampled from the true distribution.

Next, we perform the same task as the previous one, but this time the sample range if $[0, 4000)$. Figure 10 shows $200\,000$ natural numbers sampled from the true distribution in Figure 2.

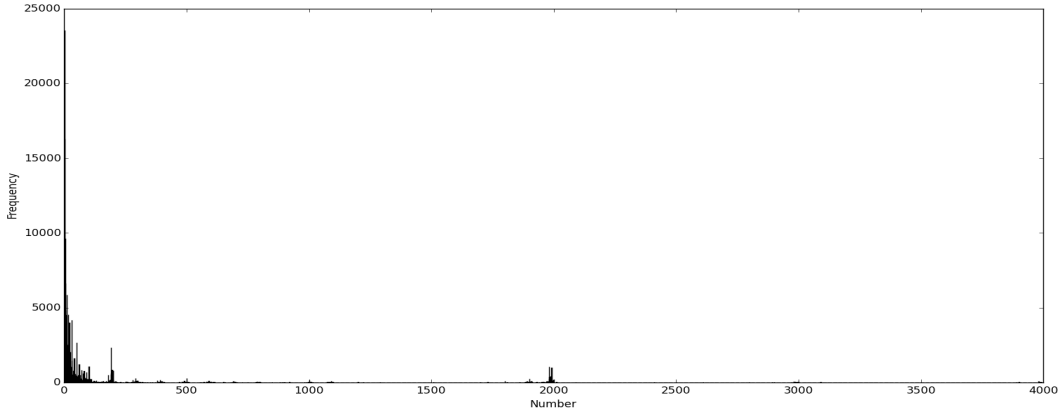
Sampling from the three grammars is illustrated in Figure 11. We can see that the half ambiguous grammar in Figure 11b still has the phenomenon of having high peaks at some certain numbers and almost zero everywhere else. Even though it manages to capture the numbers concentrated around $2\,000$, the relative frequencies of these numbers do not follow those in the true distribution. The highly ambiguous grammar in Figure 11a has the best performance out of of three since it can capture the peak at $2\,000$ very well. In the case of the deterministic grammar in Figure 11c, because this grammar favors shorter rules over long ones, it has difficulty generating numbers exceeding one thousand. Even though some numbers around $2\,000$ are generated, their distribution is far from the true distribution.




(a) Highly ambiguous grammar.




(b) Half ambiguous grammar.




(c) Deterministic grammar.

Figure 11: $200\,000$ natural numbers in range $[0, 4000)$ sampled from the three grammars.

## 4 Conclusions

In this work we present two likelihood computation methods for assessing how well grammars represent a dataset: (1) Viterbi probablity, i. e. using only the probability of the best derivation and (2) Inside probability, which calculates the entire likelihood of a string. Using those computations in a Metropolis-Hastings sampling method, we are able to induce a Tree Substitution Grammar while maximizing the likelihood of the data under that grammar. When comparing the results of the two likelihood calculations, we can see that the same final result is achieved, as shown by the likelihood of the whole dataset. The main explanation is that the grammars start to overfit to the dataset after some sampling iterations. This results in grammar rules that mostly rewrite to terminal symbols and that thus do not allow for much composition anymore. This also means that most of the numerals in the dataset only have one possible parse anyway in the end and thus the Viterbi probability is the entire likelihood of the string.

Although the likelihood is steadily increased during sampling, the final grammars are not able to generalize anymore, which is a typical case of overfitting encountered in machine learning methods. One possible to way to counteract this problem is to use a Bayesian approach that allows for penalization of undesired grammar structures.

Afterwards, we investigated the impact of grammar ambiguity. Trying different ambiguous grammars with the Viterbi method should not have an impact, as we would always be picking the most likely parse for each string; but when the inside probabilities are applied, we take into account all of the parsing trees. As shown in the results, there is only a slight difference between the deterministic and the ambiguous grammar, but it becomes more evident with the highly ambiguous one.

By sampling from the approximated distributions, we can conclude that the highly ambiguous grammar works best in generating numbers following the true distribution, while the less ambiguous grammar performs worst in all cases. If only small numbers are concerned, the deterministic grammar is a good candidate for its simplicity while still able to produce a distribution close to the true one.

## References


[1] COHN, Trevor; BLUNSOM, Phil and GOLDWATER, Sharon. *Inducing tree-substitution grammars*. In: *The Journal of Machine Learning Research* 11 (2010), pp. 3053–3096.

[2] DYER, Chris; LOPEZ, Adam; GANITKEVITCH, Juri; WEESE, Johnathan; TURE, Ferhan; BLUNSOM, Phil; SETIAWAN, Hendra; EIDELMAN, Vladimir and RESNIK, Philip. *cdec: A Decoder, Alignment, and Learning framework for finite-state and context-free translation models*. In: *Proceedings of ACL*. 2010.

## Appendix A Implementation details

Listing 1: `datareader.py`

```
import numpy
import matplotlib.pyplot as plt
import random

class CorpusReader:
    def __init__(self):
        self.lookup = dict()
        self.lookup['one'] = 1
        self.lookup['two'] = 2
        self.lookup['three'] = 3
        self.lookup['four'] = 4
        self.lookup['five'] = 5
        self.lookup['six'] = 6
        self.lookup['seven'] = 7
        self.lookup['eight'] = 8
        self.lookup['nine'] = 9
        self.lookup['ten'] = 10
        self.lookup['eleven'] = 11
        self.lookup['twelve'] = 12
        self.lookup['thirteen'] = 13
        self.lookup['fourteen'] = 14
        self.lookup['fifteen'] = 15
        self.lookup['sixteen'] = 16
        self.lookup['seventeen'] = 17
        self.lookup['eighteen'] = 18
        self.lookup['nineteen'] = 19
        self.lookup['twenty'] = 20
        self.lookup['thirty'] = 30
        self.lookup['forty'] = 40
        self.lookup['fifty'] = 50
        self.lookup['sixty'] = 60
        self.lookup['seventy'] = 70
        self.lookup['eighty'] = 80
        self.lookup['ninety'] = 90
        self.lookup['hundred'] = 100
        self.lookup['thousand'] = 1000
        self.lookup['million'] = 10 ** 6
        self.lookup['billion'] = 10 ** 9
        self.lookup['trillion'] = 10 ** 12
        self.reset()


    def reset(self):
        self.count_num = dict()
        self.count_alpha = dict()
        self.count_total = dict()


    def read_data(self, filename, tag='CD'):
        '''
        Read number data from a corpus or treebank
```

```
        :param filename: Name of file to be read
        :param tag: Look for numbers inside this tag, if None is
            supplied,
                    numbers are assumed to be one per line
        '''
        self.reset()

        with open(filename) as fileobject:
            for line in fileobject:
                number_groups = []

                # If a tag is provided, read their values first
                if tag is not None:
                    extendedtag = tag + ' "'
                    current_number = ''
                    next_tag_idx = line.find(extendedtag)
                    while next_tag_idx != -1:
                        line = line[next_tag_idx+len(tag)+2:]
                        next_quote_idx = line.find('"')
                        if current_number:
                            current_number += ' '
                        current_number += line[:next_quote_idx]

                        next_tag_idx = line.find(extendedtag)
                        if next_tag_idx > len(current_number) + 4:
                            # The two tags are too far away
                            number_groups.append(current_number)
                            current_number = ''
                    if current_number:
                        number_groups.append(current_number)

                    i = 0
                    while i < len(number_groups):
                        # Post-process
                        if number_groups[i].find('/') != -1:
                            number_groups.remove(number_groups[i])
                        else:
                            number_groups[i] = number_groups[i].
                                replace(',', '')
                            number_groups[i] = number_groups[i].
                                replace('-', ' ')
                            i += 1
                else:
                    number_groups.append(line)

                for numberstr in number_groups:
                    numberstr = numberstr.lower()
                    numberstr = numberstr.strip()
                    numbers = numberstr.split(' ')
                    if len(numbers) == 1:
                        numbers = numbers[0]
                        if numbers in self.lookup:
                            number = self.lookup[numbers]
```

```
                self.count_alpha = self.add(self.
                    count_alpha, number)
                self.count_total = self.add(self.
                    count_total, number)
            else:
                try:
                    number = int(numbers)
                    self.count_num = self.add(self.
                        count_num, number)
                    self.count_total = self.add(self.
                        count_total, number)
                except:
                    continue
        elif len(numbers) == 2:
            number = 0
            isfloat = [False, False]
            firstnum = 0
            secondnum = 0

            try:
                firstnum = float(numbers[0])
                isfloat[0] = True
            except Exception:
                pass

            try:
                secondnum = float(numbers[1])
                isfloat[1] = True
            except Exception:
                pass

            if isfloat[0] and isfloat[1]:
                self.count_total = self.add(self.
                    count_total, int(firstnum))
                self.count_total = self.add(self.
                    count_total, int(secondnum))
                self.count_num = self.add(self.
                    count_num, int(firstnum))
                self.count_num = self.add(self.
                    count_num, int(secondnum))
            else:
                if isfloat[0] and not isfloat[1]:
                    if numbers[1] in self.lookup:
                        number = int(firstnum * self.
                            lookup[numbers[1]])
                    else:
                        number = int(firstnum)
                    self.count_total = self.add(self.
                        count_total, number)
                    self.count_alpha = self.add(self.
                        count_alpha, number)
                elif not isfloat[0] and not isfloat
                    [1]:
                    adding = True
```

```
                    if numbers[0] not in self.lookup
                        or numbers[1] not in self.
                        lookup:
                        if numbers[1][-1] == 's':
                            try:
                                number = int(numbers
                                    [1][:-1])
                            except Exception:
                                adding = False
                    else:
                        number = self.lookup[numbers
                            [0]] * self.lookup[numbers
                            [1]]
                    if adding:
                        self.count_total = self.add(
                            self.count_total, number)
                        self.count_alpha = self.add(
                            self.count_alpha, number)


def get_statistics(self, limit=3000):
    keys = range(limit)
    number_form = []
    alphabetic_form = []
    total = []
    for number in keys:
        if number in self.count_num:
            number_form.append(self.count_num[number])
        else:
            number_form.append(0)

        if number in self.count_alpha:
            alphabetic_form.append(self.count_alpha[number])
        else:
            alphabetic_form.append(0)

        if number in self.count_total:
            total.append(self.count_total[number])
        else:
            total.append(0)
    return number_form, alphabetic_form, total


def getIndexFromProb(self, probList, randomValue):
    probArray = numpy.array(probList)
    probArray = probArray * 1. / numpy.sum(probArray, axis=0)
    cumprob = numpy.cumsum(probArray)
    return numpy.size(cumprob, 0) - numpy.count_nonzero(
        cumprob > randomValue)


def sample(self, limit=4000, size=20000, uniformprob=0.001):
    samples = numpy.zeros(limit)
    _, _, total = self.get_statistics(limit)
```

```
        for _ in range(size):
            if numpy.random.sample() < uniformprob:
                # Sample uniformly
                samples[numpy.random.randint(0, limit)] += 1
            else:
                samples[self.getIndexFromProb(total, numpy.random.
                    sample())] += 1
        return samples

    def add(self, dictToAdd, key):
        if key not in dictToAdd:
            dictToAdd[key] = 0
        dictToAdd[key] += 1

        return dictToAdd

    def get_sampled_dataset(self, size=1000):
        dataset = []

        for n, c in self.count_total.items():
            dataset += [str(n)] * c

        random.shuffle(dataset)

        return dataset[:size]


if __name__ == '__main__':
    # Initialize the reader
    reader = CorpusReader()

    # Read numeral data
    reader.read_data('wsj01-21-without-tags-traces-punctuation-m40
        .txt', 'CD')


    # Limit to range [0,200)
    limit = 4000

    # Get distributions
    number_form, alphabetic_form, total = reader.get_statistics(
        limit=limit)

    keys = range(limit)
    size = 200000

    # Plot the true distribution
    plt.xlabel('Number')
    plt.ylabel('Frequency')
    plt.title('Distribution of natural numbers in alphabetic form
        in [0,' + str(limit) + ') from the treebank')
    plt.bar(keys, total, color='g')
    plt.show()
```

```
    # Sample from the distribution
    samples = reader.sample(limit, size=size, uniformprob=0)


    # Plot the sampled distribution
    plt.xlabel('Number')
    plt.ylabel('Frequency')
    plt.title(str(size) + ' natural numbers in [0, ' + str(limit)
        + ') sampled from the true distribution')
    plt.bar(keys, samples, color='g')
    plt.show()
    pass
```

Listing 2: `sampling_numbers.py`

```
import cPickle as pickle
import numpy
import matplotlib.pyplot as plt

class Tree:
    def __init__(self, root, left=None, right=None):
        self.root = root
        self.left = left
        self.right = right


    def terminal(self):
        return self.isdigit(self.root)


    def leaf(self):
        return True if self.left is None and self.right is None
            else False


    def fullygrown(self):
        if self.leaf():
            if self.terminal():
                return True
            else:
                return False
        else:
            return True if self.left.fullygrown() and (self.right
                is None or self.right.fullygrown()) else False


    def isdigit(self, value):
        try:
            int(value)
            return True
        except Exception:
            return False
```

```
class Sampler:
    def __init__(self, final_treeFrequency):
        self.rule_S = []
        self.rule_S1 = []
        self.rule_S2 = []
        self.rule_D = []

        self.rule_S_count = []
        self.rule_S1_count = []
        self.rule_S2_count = []
        self.rule_D_count = []

        for key, value in final_treeFrequency.items():
            if key[0] == '(':
                key = key[1:-1]
            root = key[:2]
            if root == 'S ':
                self.rule_S.append(key)
                self.rule_S_count.append(value)
            elif root == 'S1':
                self.rule_S1.append(key)
                self.rule_S1_count.append(value)
            elif root == 'S2':
                self.rule_S2.append(key)
                self.rule_S2_count.append(value)
            elif root == 'D ':
                self.rule_D.append(key)
                self.rule_D_count.append(value)


    def getleftstr(self, treestr):
        firstleft = treestr.find('(')

        if firstleft == -1:
            return 0, len(treestr)-1

        leftcount = 1
        for i in range(firstleft+1, len(treestr)):
            if treestr[i] == '(':
                leftcount += 1
            elif treestr[i] == ')':
                leftcount -= 1
            if leftcount == 0:
                return firstleft, i


    def parse(self, treestr):
        if not treestr:
            return None
        treestr = treestr.strip()
        if treestr[0] == '(':
            treestr = treestr[1:-1]

        root = treestr[:treestr.find(' ')]
```

```
        leftbracket = treestr.find('(')

        if leftbracket == -1:
            spaceidx = treestr.find(' ')
            if spaceidx != -1:
                # Base case, for example 'NZ 4'
                innervalue = treestr[spaceidx+1:]
                return Tree(root, Tree(innervalue))
            else:
                # Base case ending with non-terminal, for example
                    '(S2)'
                return Tree(treestr)

        leftlidx, leftridx = self.getleftstr(treestr)
        leftstr = treestr[leftlidx:leftridx+1]
        rightstr = treestr[leftridx+2:]
        rightstr = rightstr[rightstr.find('('):]

        return Tree(root, self.parse(leftstr), self.parse(rightstr
            ))

    def getnumberstr(self, tree):
        '''
        Get the number a tree represents
        :param tree: A fully-grown tree
        '''
        if tree is None:
            return ''
        if tree.terminal():
            return tree.root
        numberstr = ''
        numberstr += self.getnumberstr(tree.left)
        numberstr += self.getnumberstr(tree.right)
        return numberstr


    def getnumber(self, tree):
        return int(self.getnumberstr(tree))


    def sample(self, limit=4000, size=10000):
        '''
        Sample natural numbers from the grammar
        :param limit: Cut off at this limit
        :param size: Number of samples to return
        '''
        samples = numpy.zeros(limit)
        for _ in range(size):
            number = self.getnumber(self.generate_tree())
            if number < limit:
                samples[number] += 1
        return samples
```

```
    def getIndexFromProb(self, probList, randomValue):
        probArray = numpy.array(probList)
        probArray = probArray * 1. / numpy.sum(probArray, axis=0)
        cumprob = numpy.cumsum(probArray)
        return numpy.size(cumprob, 0) - numpy.count_nonzero(
            cumprob > randomValue)


    def expand_tree(self, tree):
        while not tree.fullygrown():
            if tree.leaf():
                if not tree.terminal():
                    root = tree.root
                    if root == 'S1':
                        treestr = self.rule_S1[self.
                            getIndexFromProb(self.rule_S1_count,
                            numpy.random.sample())]
                    elif root == 'S2':
                        treestr = self.rule_S2[self.
                            getIndexFromProb(self.rule_S2_count,
                            numpy.random.sample())]
                    elif root == 'D':
                        treestr = self.rule_D[self.
                            getIndexFromProb(self.rule_D_count,
                            numpy.random.sample())]
                    subtree = self.parse(treestr)
                    tree.left = subtree.left
                    tree.right = subtree.right
                    return tree
            else:
                if not tree.left.fullygrown():
                    tree.left = self.expand_tree(tree.left)
                if tree.right is not None and not tree.right.
                    fullygrown():
                    tree.right = self.expand_tree(tree.right)
        return tree


    def generate_tree(self):
        treestr = self.rule_S[self.getIndexFromProb(self.
            rule_S_count, numpy.random.sample())]
        print treestr
        return self.expand_tree(self.parse(treestr))




if __name__ == '__main__':
    inal_dataset, final_rootFrequency, final_treeFrequency,
        final_grammar = pickle.load(open('deterministic_final.pkl'
        , 'rb'))
#     initial_dist = pickle.load(open('test_initial_dist.pkl', 'rb
    '))
```

```
    # Limit to range [0,200)
    limit = 4000

    keys = range(limit)
    size = 200000


    # Sample from the grammar
    sampler = Sampler(final_treeFrequency)
    samples = sampler.sample(limit=limit, size=size)



    # Plot the sampled distribution
    plt.xlabel('Number')
    plt.ylabel('Frequency')
    plt.title(str(size) + ' natural numbers in [0, ' + str(limit)
        + ') sampled from the deterministic grammar')
    plt.bar(keys, samples, color='g')
    plt.show()
```

Listing 3: `evaluate_ds.py`

```
__author__ = 'rwechsler'
import constants
from parser import Parser
import re
import cPickle as pickle
import os
from datareader import CorpusReader

#------------ Begin Params ----------#
CDEC_PATH = constants.CDEC_PATH
WEIGHT_FILE = constants.WEIGHT_FILE
INITIAL_INI = constants.INITIAL_INI
TMP_DATA_DIR = constants.TMP_DATA_DIR
#------------ End Params ------------#



def get_dataset_likelihood(raw_dataset, grammar):
    '''
    Computes the likelihood of the whole dataset
    '''

    # write grammar_file

    outfile = open(TMP_DATA_DIR + "tmp_grammar.cfg", "w")
    outfile.write(grammar)
    outfile.close()

    # write init_file

    infile = open(INITIAL_INI, "r")
```

```
    init = infile.read()

    infile.close()

    new_init = re.sub("grammar=.*", "grammar=" + TMP_DATA_DIR + "
        tmp_grammar.cfg", init)

    outfile = open(TMP_DATA_DIR + "tmp_init.ini", "w")
    outfile.write(new_init)
    outfile.close()

    parser = Parser(TMP_DATA_DIR + "tmp_init.ini", CDEC_PATH)

    likelihood = 0

    prob_f = parser.get_inside_string

    for s in raw_dataset:
        likelihood += prob_f("␣".join(s))

    # delete tmp_files
    os.remove(TMP_DATA_DIR + "tmp_grammar.cfg")
    os.remove(TMP_DATA_DIR + "tmp_init.ini")

    return likelihood


_, _, _, final_grammar1 = pickle.load(open('results/comp_vit_final
    .pkl', 'rb'))
_, _, _, final_grammar2 = pickle.load(open('results/
    comp_noinside_final.pkl', 'rb'))


reader = CorpusReader()
reader.read_data('wsj01-21-without-tags-traces-punctuation-m40.txt
    ', 'CD')

dist = reader.sample(limit=4000, size=1000)
raw_dataset = []
for i, n in enumerate(dist):
    raw_dataset += [str(i)] * n


print get_dataset_likelihood(raw_dataset, final_grammar2)
print get_dataset_likelihood(raw_dataset, final_grammar1)
```

Listing 4: `old_parser.py`

```
#coding: utf8
import cdec
import numpy as np
```

```
def create_cdec_grammar(root_counts, tree_counts):
    grammar = ""
    for tree in tree_counts:
        root = tree.split()[0]

        logprob = np.log(float(tree_counts[tree])/root_counts[root
            ])




class Parser(object):

    def __init__(self, grammar, weight_file):
        # Load decoder width configuration
        self.decoder = cdec.Decoder(formalism='scfg')

        # Read weights
        self.decoder.read_weights(weight_file)

        self.grammar = grammar


    def get_inside_string(self, string):
        forest = self.decoder.translate(string, grammar=self.
            grammar)
        I = forest.inside()
        return I[-1]

    def get_best_parse(self, string):
        forest = self.decoder.translate(string, grammar=self.
            grammar)
        return forest.viterbi_trees()[0][1:-1]

    def get_random_parse(self, string):
        forest = self.decoder.translate(string, grammar=self.
            grammar)
        return list(forest.sample_trees(1))[0][1:-1]



if __name__ == "__main__":

    grammar_f = open("initial_grammar", "r")
    grammar = grammar_f.read()
    grammar_f.close()

    parser = Parser(grammar, "example.weights")


    test = "0"
```

```
    print parser.get_inside_string(test)

    print parser.get_best_parse(test)

    print parser.get_random_parse(test)
```

Listing 5: `parser.py`

```
#coding: utf8
import cdec
import numpy as np
import subprocess
import re
import sys

def create_cdec_grammar(root_counts, tree_counts):
    terminals = ['0','1','2','3','4','5','6','7','8','9']
    grammar = ""
    for tree in tree_counts:
        root = tree.split()[0]

        logprob = np.log(float(tree_counts[tree])/root_counts[root
            ])

        stree = re.sub(r" (\w+?) ", r" \1)", tree + ")")
        stree = re.sub(r"\((\w+)\)", r"\1)", stree)

        leaves = re.findall(r"[ |\(](\w+?)\)", stree)
        #leaves = re.findall(r"[^\(](\w+)[\)]", stree)
        i = 0
        for j, l in enumerate(leaves):
            if l not in terminals:
                i += 1
                leaves[j] = "[" + l + "," + str(i) + "]"

        RH = " ".join(leaves)

        rule = "[" + root.lstrip("(") + "]" + " ||| " + RH + " ||| 
            " + RH + " ||| LogProb=" + str(logprob) + "\n"

        grammar += rule

    return grammar



class Parser(object):

    def __init__(self, config_file, path_cdec):
        self.config_file = config_file
        self.path_cdec = path_cdec


    def get_inside_string(self, string):
```

```
        parsing = subprocess.Popen([self.path_cdec, "-c", self.
            config_file, "-z"], stdin=subprocess.PIPE, stdout=
            subprocess.PIPE, stderr=subprocess.STDOUT)
        results = parsing.communicate(input=string + "\n")
        result = results[0]
        try:
            return float(re.search(r"log\(Z\): (.*)", result).
                group(1))
        except AttributeError:
            raise Exception("PARSE FAIL: " + result)


    def get_best_parse(self, string):
        parsing = subprocess.Popen([self.path_cdec, "-c", self.
            config_file, "-z"], stdin=subprocess.PIPE, stdout=
            subprocess.PIPE, stderr=subprocess.STDOUT)
        result = parsing.communicate(input=string + "\n")[0]
        try:
            return re.search(r"tree: (\(.+\))\n", result).group(1)
                [1:-1]
        except AttributeError:
            raise Exception("PARSE FAIL: " + result)

    def get_max_likelihood_string(self, string):
        parsing = subprocess.Popen([self.path_cdec, "-c", self.
            config_file, "-z"], stdin=subprocess.PIPE, stdout=
            subprocess.PIPE, stderr=subprocess.STDOUT)
        result = parsing.communicate(input=string + "\n")[0]
        try:
            return float(re.search(r"Viterbi logp: (.*)", result).
                group(1))
        except AttributeError:
            raise Exception("PARSE FAIL: " + result)


    def get_random_parse(self, string):
        # TO-DO: implement
        return None



if __name__ == "__main__":


    parser = Parser("initial.ini", "/home/rwechsler/
        PycharmProjects/cdec/decoder/cdec")


    test = "1 2 3"


    print parser.get_inside_string(test)

    print parser.get_best_parse(test)
```

```
    print parser.get_max_likelihood_string(test)


    #tree_counts = {"S (NZ 3) (S2 S2)": 1}
    #root_counts = {"S": 1}

    #print create_cdec_grammar(root_counts, tree_counts)
```

Listing 6: `plot.py`

```
from __future__ import division
from matplotlib import pyplot as plt

# read data

infile = open("results/comp_noinside_results.txt", "r")
data = []
for line in infile:
    data.append(tuple(line.strip().split()))

infile.close()

x = [tup[0] for tup in data]

# plot accepted likelihood

accepted_ll = [tup[3] for tup in data]
candidate_ll = [tup[2] for tup in data]

plt.plot(x, accepted_ll, label="accepted", linewidth=4)
plt.plot(x, candidate_ll, color="r", label="candidate")
plt.legend()
plt.title("Likelihood during sampling")
plt.xlabel("iterations")
plt.ylabel("log-likelihood")
plt.show()



et_types = [tup[4] for tup in data]
et_tokens = [tup[5] for tup in data]


#plt.plot(x, et_types, label="types")
plt.plot(x, et_tokens, color="r", label="tokens")
plt.legend()
plt.title("Number of el. trees in dataset")
plt.xlabel("iterations")
plt.ylabel("number of elementary trees")
plt.show()


plt.plot(x, et_types, label="types")
plt.legend()
```

```
plt.title("Number of grammar rules")
plt.xlabel("iterations")
plt.ylabel("number of grammar rules")
plt.show()


counts = {"A": 0, "F": 0, "R": 0}

labels = [tup[1] for tup in data[1:]]

for l in labels:
    counts[l] += 1

ls = []
fracs = []

for l, c in counts.items():
    ls.append(l)
    print l, c / (len(data) - 1)
    fracs.append(c / (len(data) - 1))
plt.pie(fracs, labels=ls, startangle=90, autopct='%1.1f%%')
plt.title("Sampling acceptance rate")
plt.show()
```

Listing 7: `sampling.py`

```
from __future__ import division
import random
import numpy as np
import re
import sys
import cPickle as pickle
from matplotlib import pyplot as plt
from datareader import CorpusReader
from parser import Parser, create_cdec_grammar
import os
import constants
from collections import defaultdict


#---------- Begin Params ----------#
CDEC_PATH = constants.CDEC_PATH
WEIGHT_FILE = constants.WEIGHT_FILE
INITIAL_INI = constants.INITIAL_INI
TMP_DATA_DIR = constants.TMP_DATA_DIR
#---------- End Params ----------#


#---------- Begin global vars ----------#
treeFrequency = dict() # Keys: Elementary trees. Values: Frequency
    of each elementary tree
rootFrequency = dict() # Keys: Roots of the elementary trees.
   Values: Frequency of each root
newTreeFrequency = dict() # Keys: Elementary trees after Metrop-
   Hast random change. Values: Frequency of each elementary tree
```

```
newRootFrequency = dict() # Keys: Roots of the elementary trees
    formed after Metrop-Hast random change. Values: Frequency of
    each elementary tree
#------------ End global vars ------------#


def updateDictionary(parse, update=True, statistcs=True):
    '''
    Updates the frequency dictionaries
    '''

    global treeFrequency
    global rootFrequency
    global newTreeFrequency
    global newRootFrequency

    # remove head and tail parenthesis
#     parse = parse[1:len(parse)-1]
    parse = parse.strip()

    if update:
        # get root
        root = parse.split()[0]

        if statistcs: # update general stats
            # update frequency of this tree
            treeFrequency.setdefault(parse,0)
            treeFrequency[parse] += 1

            # update the root frequency
            rootFrequency.setdefault(root,0)
            rootFrequency[root] += 1

        else: # update new stats

            newTreeFrequency.setdefault(parse,0)
            newTreeFrequency[parse] += 1

            newRootFrequency.setdefault(root,0)
            newRootFrequency[root] += 1

    derivations = set()
    derivations.add(parse)

    return derivations


def parse_dataset(dataset, ini_file):

    p = Parser(ini_file, CDEC_PATH)
    parsed = []
    for string in dataset:
        parse = p.get_best_parse(' '.join(string))
        parsed.append(parse)
```

```
    return parsed



def get_dataset_likelihood(raw_dataset, root_counts, tree_counts,
    viterbi=False):
    '''
    Computes the likelihood of the whole dataset
    '''

    # generate grammar
    grammar = create_cdec_grammar(root_counts, tree_counts)

    # write grammar_file

    outfile = open(TMP_DATA_DIR + "tmp_grammar.cfg", "w")
    outfile.write(grammar)
    outfile.close()

    # write init_file

    infile = open(INITIAL_INI, "r")
    init = infile.read()

    infile.close()

    new_init = re.sub("grammar=.*", "grammar=" + TMP_DATA_DIR + "
        tmp_grammar.cfg", init)

    outfile = open(TMP_DATA_DIR + "tmp_init.ini", "w")
    outfile.write(new_init)
    outfile.close()

    parser = Parser(TMP_DATA_DIR + "tmp_init.ini", CDEC_PATH)

    likelihood = 0

    prob_f = parser.get_inside_string

    if viterbi:
        prob_f = parser.get_max_likelihood_string


    for s in raw_dataset:
        likelihood += prob_f(" ".join(s))

    # delete tmp_files
    os.remove(TMP_DATA_DIR + "tmp_grammar.cfg")
    os.remove(TMP_DATA_DIR + "tmp_init.ini")

    return likelihood

def getNonTerminals():
```

```
    '''
    Returns a list of non terminal symbols. Does not include the
        root symbol.
    '''

    return ['S1', 'S2', 'D', 'NZ', 'P'] #TODO: Fix this. Get the
        list dynamically.


def placeSubstitutionPoints(treebank):
    '''
    Receives a list of trees and randomly places subtitution
        points.
    :param treebank: Treebank corpus
    :return: Marked treebank corpus
    '''

    convertedTreebank = []

    threshold = 0.3

    for tree in treebank:
        decomposeTSG(tree,update=True,statistcs=True) # update
            statistics in rootFrequency and treeFrequency
        convertedTree = []

        tags = tree.split(' ')
        #print tags
        for tag in tags:
            #print tag
#               match = re.match(r'.*S ',n).group()
#               if tag=='(S':

            p = np.random.rand()

            if p < threshold:
                match = re.search(r'.*(?:'+'|'.join(
                    getNonTerminals()) +')', tag) # match any non
                    terminal symbol but the root
                if match:
                    newTag = match.group()+'*'
                else:
                    newTag = tag
            else:
                newTag = tag

            convertedTree.append(newTag)

        convertedTreebank.append(' '.join(convertedTree))
        decomposeTSG(' '.join(convertedTree),update=True,statistcs
            =False) # update statistics in newRootFrequency and
            newTreeFrequency

    return convertedTreebank
```

```
def getBlock(firstStarIndex, tree):
    '''
    Retrieves the elementary tree of a marked tee
    :param firstStarIndex: Index of the '*' symbol in the tree
    :param tree: tree containing the elementary tree to be
        extracted
    :return:
    '''
    # Get first previous parenthesis
    for i in range(firstStarIndex):
        firstParenthesis = tree.find('(',firstStarIndex-i,
            firstStarIndex)
        if firstParenthesis > -1:
            break

    # get block by counting parenthesis
    count = 0
    for i,ch in enumerate(list(tree[firstParenthesis:])):
        if ch=='(':
            count += 1
        if ch==')':
            count -= 1

        if count == 0:
            break

    lastParenthesis = firstParenthesis+i+1 # include the last
        parenthesis

    # get block
    rawBlock = tree[firstParenthesis:lastParenthesis]

    # get substitution symbol
    substitutionSymbol = tree[firstParenthesis+1:firstStarIndex]

    return rawBlock, substitutionSymbol


def decomposeTSG(tree,update=False,statistcs=False):
    '''
    Decomposes a tree which has substitution points into
        elementary trees.
    :param tree: marked tree with substitution points
    :return:
    '''

    if tree.count('*')==0:
        derivations = updateDictionary(tree,update=update,
            statistcs=statistcs)

        return derivations
```

```
    # Get first star
    firstStarIndex = tree.find('*') # returns -1 on failure

    rawBlock, substitutionSymbol = getBlock(firstStarIndex, tree)

    # remove first star from block
    block = rawBlock.replace('*','',1) # theres always gonna be,
        at least, one.

    # replace the block by the substitution symbol
    newL = tree.replace(rawBlock, ' ('+substitutionSymbol+') ', 1)

    result = set()
    # recursive call to keep decomposing
    result = result.union(decomposeTSG(newL,update=update,
        statistcs=statistcs))
    result = result.union(decomposeTSG(block,update=update,
        statistcs=statistcs))

    return result


def make_random_candidate_change(treebank, action=None):
    '''
    Given a grammar, makes a random change (adds or removes
    substitution point) and
    retrieves the new elementary trees.
    :param dataset:
    :return:
    '''

    global newRootFrequency
    global newTreeFrequency

    pChange = random.random() # add or delete star?

    option = False
    prevent = 0 # prevent the candidate search while loops from
        getting stuck
    probAdding = 0.5 # probability of making a change by adding a
        star

    if pChange < probAdding or action=='add': # add a star
        # select a parse that has a slot to add a star
        while not option:

            if prevent>len(treebank):
                break

            parseIndex = random.randint(0, len(treebank)-1) #
                select a random parse
            parse = treebank[parseIndex]
            symbols = getNonTerminals()
```

```
            slots = [symbol for symbol in symbols if parse.count('
                ('+symbol+' ')>0 or parse.count(' '+symbol+' ')>0]
            if len(slots) > 0:
                option = True

            prevent += 1

        #TODO: Check this. Before, I would move to the next
            iteration
#           if not option: # i got stuck in the while loop
#               logging.info('Couldnt find a replacement slot')
#               continue
        #TODO: not nice. -lqrz
        if len(slots)==0:
            return make_random_candidate_change(treebank, action='
                remove')

        symbol = slots[random.randint(0,len(slots)-1)] # choose
            random symbol to insert star
        countSymbol = parse.count(symbol+' ') # take care not to
            count symbol*


        pSymbol = random.randint(1,countSymbol)

        # get index of chosen symbol
        index=0
        for _ in range(pSymbol):
            index = parse.find(symbol+' ', index+1,len(parse)) #
                symbols have variable length

        # get block and substitution symbol
        rawBlock, _ = getBlock(index, parse)

        # remove first star from block
        newBlock = rawBlock.replace(symbol,symbol+'*',1) # theres
            always gonna be, at least, one.

    else: # delete a star
        # select a parse that has a star to be removed
        while not option:

            if prevent>len(treebank):
                break

            parseIndex = random.randint(0, len(treebank)-1) #
                select a random parse
            parse = treebank[parseIndex]
            if parse.count('*') > 0:
                option = True

            prevent += 1
```

```
        #TODO: Check this. Before, I would move to the next
            iteration
#         if not option: # i got stuck in the while loop
#             logging.info('Couldnt find a replacement slot')
#             continue

        # remove a star
        countStars = parse.count('*')

        #TODO: not nice. -lqrz
        if countStars==0:
            return make_random_candidate_change(treebank, action='
                add')

        pStar = random.randint(1,countStars) # choose star to
            eliminate

        # get index of chosen star
        index=0
        for _ in range(pStar):
            index = parse.find('*', index+1,len(parse))

        # get block and substitution symbol
        rawBlock, _ = getBlock(index, parse)

        # remove first star from block
        newBlock = rawBlock.replace('*','',1) # theres always
            gonna be, at least, one.

    newParses = list()
    for tree in treebank:
        newParse = tree.replace(rawBlock,newBlock)
        newParses.append(newParse)
        decomposeTSG(newParse, update=True,statistcs=False) #
            stats==True: update general

    return newParses


def metropolis_hastings(raw_dataset, old_dataset, n=1000, ap=None,
    viterbi=False, outfile=sys.stdout):
    '''
    Runs Metropolis Hastings algorithm
    '''

    global treeFrequency
    global rootFrequency
    global newTreeFrequency
    global newRootFrequency


    # likelihood before running the Metrop-Hast algorithm.
        Considering the substitution points.
```

```
old_likelihood = get_dataset_likelihood(raw_dataset,
    newRootFrequency, newTreeFrequency, viterbi=viterbi)


outfile.write("\t".join(["0", "A", str(old_likelihood), str(
    old_likelihood), str(len(newTreeFrequency.keys())), str(np
    .sum(newTreeFrequency.values()))]) + "\n")

for i in range(n):
    newTreeFrequency = dict()
    newRootFrequency = dict()
    new_dataset = make_random_candidate_change(old_dataset)
    new_likelihood = get_dataset_likelihood(raw_dataset,
        newRootFrequency, newTreeFrequency, viterbi=viterbi) #
         lqrz: by passing the old and new block we can forloop
         only once to ge the likelihood.
    if new_dataset == old_dataset:
        print "EQUAL!!"

    if new_likelihood > old_likelihood:
        outfile.write("\t".join([str(i+1), "A", str(
            new_likelihood), str(new_likelihood), str(len(
            newTreeFrequency.keys())), str(np.sum(
            newTreeFrequency.values()))]) + "\n")
        #print "accepted: ", new_likelihood, old_likelihood
        old_likelihood = new_likelihood
        old_dataset = new_dataset
        treeFrequency = dict(newTreeFrequency)
        rootFrequency = dict(newRootFrequency)
    else:
        if not ap:
            p = np.exp(new_likelihood- old_likelihood)
        else:
            p = ap
        r =np.random.binomial(1, p)
        if r:
            outfile.write("\t".join([str(i+1), "F", str(
                new_likelihood), str(new_likelihood), str(len(
                newTreeFrequency.keys())), str(np.sum(
                newTreeFrequency.values()))]) + "\n")
            #print "forced: ", new_likelihood, old_likelihood
            old_likelihood = new_likelihood
            old_dataset = new_dataset
            treeFrequency = dict(newTreeFrequency)
            rootFrequency = dict(newRootFrequency)
        else:
            # reject
            outfile.write("\t".join([str(i+1), "R", str(
                new_likelihood), str(old_likelihood), str(len(
                treeFrequency.keys())), str(np.sum(
                treeFrequency.values()))]) + "\n")
            #print "rejected ", new_likelihood, old_likelihood
            newRootFrequency = dict(rootFrequency)
            newTreeFrequency = dict(treeFrequency)
```

```
        print i, old_likelihood

    return old_dataset, old_likelihood, rootFrequency,
        treeFrequency

def run_experiment(outfile_name, limit=4000, size=10000,
    uniformprob=None, ap=None, iterations=10000, viterbi=False):


    reader = CorpusReader()
    reader.read_data('wsj01-21-without-tags-traces-punctuation-m40
        .txt', 'CD')

    if limit is None:
        raw_dataset = reader.get_sampled_dataset(size=size)
        dist = defaultdict(int)
        for n in raw_dataset:
            dist[int(n)] += 1

    else:
        dist = reader.sample(limit=limit, size=size, uniformprob=
            uniformprob)
        raw_dataset = []
        for i, n in enumerate(dist):
            raw_dataset += [str(i)] * n


    pickle.dump(dist, open(outfile_name + "_inital_dist.pkl", "wb"
        ))


    print "Parsing dataset."
    parses = parse_dataset(raw_dataset, INITIAL_INI)

    print "Adding substitution site markers"
    dataset = placeSubstitutionPoints(parses)

    outfile = open(outfile_name + "_results.txt", "w")

    print "Starting Metropolis-Hastings."
    final_dataset, final_likelihood, final_rootFrequency,
        final_treeFrequency = metropolis_hastings(raw_dataset,
        dataset, n=iterations, ap=ap, outfile=outfile, viterbi=
        viterbi)

    print "Generating final grammar"
    # generate grammar
    final_grammar = create_cdec_grammar(final_rootFrequency,
        final_treeFrequency)

    dmp = [final_dataset, final_rootFrequency, final_treeFrequency
        , final_grammar]
```

```
    pickle.dump(dmp, open(outfile_name+"_final.pkl", "wb"))

    outfile.close()


    # TODO: sample from final grammar, and plot sampled
        distribution and store it.



    print "Experiment " + outfile_name + " done."


def test_method():
    #------------------ For debugging purposes

    reader = CorpusReader()

    reader.read_data('numbers', None)
    data = reader.count_total



    parser = Parser(INITIAL_INI, CDEC_PATH)

    parses = []
    raw_dataset = []
    for s in data.keys()[:100]:
        raw_dataset.append(str(s))
        s = ' '.join(str(s))
        parses.append(parser.get_best_parse(s))

    #parses = ['S (S1 (NZ 2)) (S2 (D 3) (S2 (D 4) (S2 (D 5))))']

    dataset = placeSubstitutionPoints(parses)
    #dataset = ['S (S1 (NZ 2)) (S2* (D* 3) (S2 (D* 4) (S2 (D* 5)))
        )']
    final_dataset = metropolis_hastings(raw_dataset, dataset, n
        =100)

run_experiment("results/test_vit", limit=4000, size=100, ap=None,
    iterations=600, viterbi=True)
#run_experiment("results/10000_2000_001", subset_size=10000, ap
    =0.01, iterations=2000)
```