DESIGN AND ANALYSIS OF ALGORITHMS

LAB 10: Algorithm Design Paradigms in Practice

I. Ideas

1. Greedy technique

- ✓ The greedy algorithm makes a locally optimal solution. In some cases, this is enough to get a globally optimal solution.
- ✓ The choice at each step must be feasible: satisfy the requirements.
- ✓ Each step reduces the problem to a smaller problem.
- ✓ Locally optimal: Each step is Greedy and tries local optimization, that is, chooses best among local choices based on some metric.
- ✓ Short-sighted.
- ✓ Easy to construct, simple to implement
- ✓ Irrevocable: once choice is made, there is no going back.
- ✓ May or may not give optimal (best) solution.

2. Dynamic Programming

- ✓ "Programming" relates to planning/use of tables, rather than computer programming.
- ✓ Solve smaller problems first, record solutions in a table; use solutions of smaller problems to get solutions for bigger problems.

- ✓ Differs from Divide and Conquer in that it stores the solutions, and subproblems are "overlapping"

II. Exercises

1. Analyze the complexity (**Θ**-notation) of greedy search decoder algorithm, then implement it in pure Python without using external libraries, such as numpy
2. Analyze the complexity (**Θ**-notation) of beam search decoder algorithm, then implement it in pure Python without using external libraries, such as numpy
3. Write a Python function for randomly generating sequence of M words over a vocabulary of N words, where M, N are integers
4. Test programs from exercises 1, 2 with 5 different inputs generated by the Python function in exercise 3.

III. Search decoder algorithms[1]

Natural language processing tasks, such as caption generation and machine translation, involve generating sequences of words.

Models developed for these problems often operate by generating probability distributions across the vocabulary of output words and it is up to decoding algorithms to sample the probability distributions to generate the most likely sequences of words.

In this tutorial, you will discover the greedy search and beam search decoding algorithms that can be used on text generation problems.

After completing this tutorial, you will know:

[1] Download at: https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/

The problem of decoding on text generation problems.

The greedy search decoder algorithm and how to implement it in Python.

The beam search decoder algorithm and how to implement it in Python.

1. Greedy Search Decoder

A simple approximation is to use a greedy search that selects the most likely word at each step in the output sequence.

This approach has the benefit that it is very fast, but the quality of the final output sequences may be far from optimal.

We can demonstrate the greedy search approach to decoding with a small contrived example in Python.

We can start off with a prediction problem that involves a sequence of 10 words. Each word is predicted as a probability distribution over a vocabulary of 5 words.

```
# define a sequence of 10 words over a vocab of 5 words
data = [[0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
```

```
                [0.5, 0.4, 0.3, 0.2, 0.1]]

data = array(data)
```

We will assume that the words have been integer encoded, such that the column index can be used to look-up the associated word in the vocabulary. Therefore, the task of decoding becomes the task of selecting a sequence of integers from the probability distributions.

The argmax() mathematical function can be used to select the index of an array that has the largest value. We can use this function to select the word index that is most likely at each step in the sequence. This function is provided directly in numpy.

The *greedy_decoder()* function below implements this decoder strategy using the argmax function.

```
# greedy decoder

def greedy_decoder(data):

        # index for largest probability each row

        return [argmax(s) for s in data]
```

Putting this all together, the complete example demonstrating the greedy decoder is listed below.

```
from numpy import array

from numpy import argmax



# greedy decoder

def greedy_decoder(data):
```

```
        # index for largest probability each row
        return [argmax(s) for s in data]


# define a sequence of 10 words over a vocab of 5 words
data = [[0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1]]
data = array(data)
# decode sequence
result = greedy_decoder(data)
print(result)
```

Running the example outputs a sequence of integers that could then be mapped back to words in the vocabulary.

```
[4, 0, 4, 0, 4, 0, 4, 0, 4, 0]
```

2. Beam Search Decoder

Another popular heuristic is the beam search that expands upon the greedy search and returns a list of most likely output sequences.

Instead of greedily choosing the most likely next step as the sequence is constructed, the beam search expands all possible next steps and keeps the *k* most likely, where *k* is a user-specified parameter and controls the number of beams or parallel searches through the sequence of probabilities.

We do not need to start with random states; instead, we start with the *k* most likely words as the first step in the sequence.

Common beam width values are 1 for a greedy search and values of 5 or 10 for common benchmark problems in machine translation. Larger beam widths result in better performance of a model as the multiple candidate sequences increase the likelihood of better matching a target sequence. This increased performance results in a decrease in decoding speed.

The search process can halt for each candidate separately either by reaching a maximum length, by reaching an end-of-sequence token, or by reaching a threshold likelihood.

Let's make this concrete with an example.

We can define a function to perform the beam search for a given sequence of probabilities and beam width parameter *k*. At each step, each candidate sequence is expanded with all possible next steps. Each candidate step is scored by multiplying the probabilities together.

The *k* sequences with the most likely probabilities are selected and all other candidates are pruned. The process then repeats until the end of the sequence.

Probabilities are small numbers and multiplying small numbers together creates very small numbers. To avoid underflowing the floating point numbers, the natural logarithm of the

probabilities are added together, which keeps the numbers larger and manageable. Further, it is also common to perform the search by minimizing the score. This final tweak means that we can sort all candidate sequences in ascending order by their score and select the first k as the most likely candidate sequences.

The *beam_search_decoder()* function below implements the beam search decoder.

```
# beam search
def beam_search_decoder(data, k):
	sequences = [[list(), 0.0]]
	# walk over each step in sequence
	for row in data:
		all_candidates = list()
		# expand each current candidate
		for i in range(len(sequences)):
			seq, score = sequences[i]
			for j in range(len(row)):
				candidate = [seq + [j], score - log(row[j])]
				all_candidates.append(candidate)
		# order all candidates by score
		ordered = sorted(all_candidates, key=lambda tup:tup[1])
		# select k best
		sequences = ordered[:k]
	return sequences
```

We can tie this together with the sample data from the previous section and this time return the 3 most likely sequences.

```
from math import log

from numpy import array

from numpy import argmax



# beam search
```

```
def beam_search_decoder(data, k):
	sequences = [[list(), 0.0]]
	# walk over each step in sequence
	for row in data:
		all_candidates = list()
		# expand each current candidate
		for i in range(len(sequences)):
			seq, score = sequences[i]
			for j in range(len(row)):
				candidate = [seq + [j], score - log(row[j])]
				all_candidates.append(candidate)
		# order all candidates by score
		ordered = sorted(all_candidates, key=lambda tup:tup[1])
		# select k best
		sequences = ordered[:k]
	return sequences


# define a sequence of 10 words over a vocab of 5 words
data = [[0.1, 0.2, 0.3, 0.4, 0.5],
```

```
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1],
                [0.1, 0.2, 0.3, 0.4, 0.5],
                [0.5, 0.4, 0.3, 0.2, 0.1]]
data = array(data)
# decode sequence
result = beam_search_decoder(data, 3)
# print result
for seq in result:
        print(seq)
```

Running the example prints both the integer sequences and their log likelihood.

Experiment with different k values.

```
[[4, 0, 4, 0, 4, 0, 4, 0, 4, 0], 6.931471805599453]
[[4, 0, 4, 0, 4, 0, 4, 0, 4, 1], 7.154615356913663]
```

```
[[4, 0, 4, 0, 4, 0, 4, 0, 3, 0], 7.154615356913663]
```