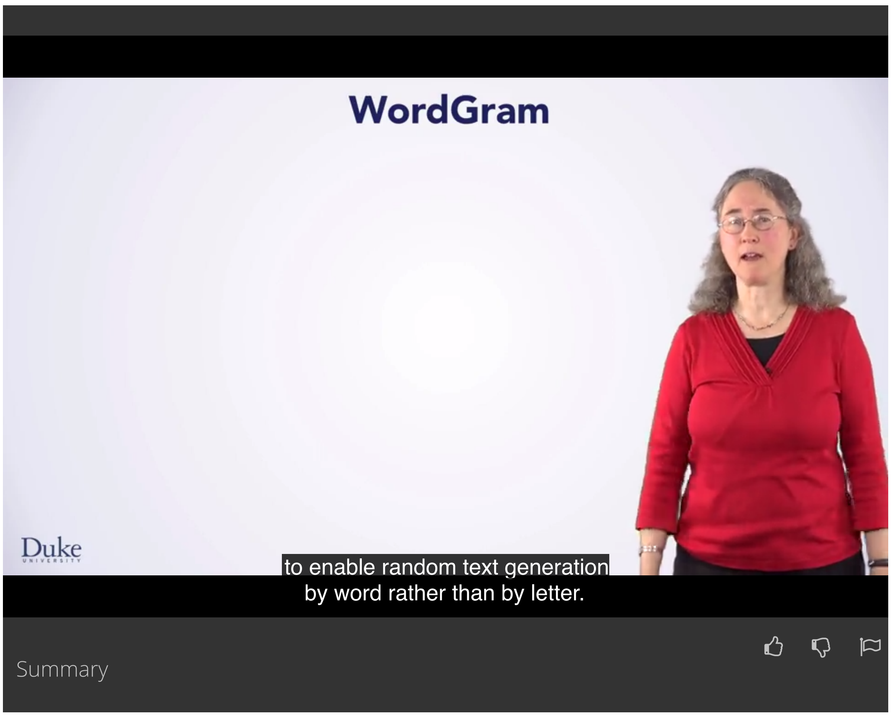
Generating Random Text

Word N-Grams

Review



Have a question? Discuss this lecture in the week forums.

Interactive Transcript

Search Transcript English

0:03
Hi, we've just completed the design and implementation of the class WordGram, to enable random text generation by word rather than by letter. This extends the predictive and occasionally amusing text generation, that's the basis for spam detection, predictive text in search engines and more.

0:24
We used word at a time, rather than letter at a time, to illustrate new concepts. But a good initial design with the IMarkovModel interface allowed our new classes to work with existing and previously tested client programs. We developed MarkovWordOne first, it used one word to predict the next. This was an easy transition from the latter Markov programs to using word gram. It's a good idea in general to take small steps in developing new programs with new ideas and changing design. After the small step works, and has been tested, take another step in extending designs. As you take steps, be sure to use the seven step process for algorithms as that's warranted.

1:11
We looked at a new class, WordGram, and it's internal representation and behavior. Our design and implementation of MarkovWordOne used strings, rather than characters, and was successful. But we had to design and implement the WordGram class to extend our order one MarkovWord program to order two, three, and more. However, we could develop the WordGram class with familiar designs, testings, programs, and interfaces.

1:40
We tested WordGram outside of the context of Markov text generation, but we developed used cases with Markov to help guide the design of WordGram. We tested .toString() and constructors first, because it's tough to test and debug when you can't create and print an object.

1:59
When it was time to implement .wordAt() and .length() methods, the similarity to similar methods in the string class helped.

2:07
We also had to understand how to implement .equals() and .hashCode(), but only .equals() was needed in our code. .hashCode() would be needed in more advanced uses of WordGram.

2:20
We encountered a few new ideas in implementing WordGram. The .wordAt() method threw an exception when an index was out of bounds. You can't index an array or a string with values like -1, or the length of the array. We made WordGram behave like String does, throwing an exception that might help the programmer when bad indexes are made.

2:44
Typically, bad indexes don't occur in relief software, but they do in the development of new software as it's being tested. But understanding exceptions is part of being a software engineer or programmer.

2:58
We wrote functions that we didn't call. Other methods might call them as .toString is called when an object is printed. As we saw sometimes, .toString is called explicitly, as when appending to a string builder.

3:13
When inserting it into a HashMap the .hashCode method gives an index for what we call a bucket, where objects are stored. The .equals() method helps distinguish objects that might end up in the same bucket. Happy programming.

Downloads

Lecture Video mp4
Subtitles (English) WebVTT
Transcript (English) txt

Would you like to help us translate the transcript and subtitles into additional languages?