There are now several open-source projects to do Bayesian spam filtering. Your own implementation is always encouraged!

Apply Bayesian (or other algorithms) with unigram or bigram on at least 10 training examples (both spam and ham emails) to find rules; based on attributes (features) you think are significant. Use at least 5 testing examples (both spam and ham emails) to test and verify the accuracy of your found rules. Here we require a relatively high accuracy (expect >80%). If your result accuracy is low, you should re-think about the features you have selected.

Doing the evaluation is easy, if a bit tedious (requiring 300 page evaluations for the complete 10 documents × 3 engines × 10 queries). Explaining the differences is more difficult. Some things to check are whether the good results in one engine are even in the other engines at all (by searching for unique phrases on the page); check whether the results are commercially sponsored, are produced by human editors, or are algorithmically determined by a search ranking algorithm; check whether each engine does the features mentioned in the next exercise.

The simplest approach is to look for a string of capitalized words, followed by “Inc” or “Co.” or “Ltd.” or similar markers. A more complex approach is to get a list of company names (e.g. from an online stock service), look for those names as exact matches, and also extract patterns from them. Reporting recall and precision requires a clearly-defined corpus which should include at least 10 news articles with a number of company names mentioned in each.

One example code:

\[0-9]*\s(([A-Z\(\&\)*]+[a-z.\(\)\:;]*\s)+(?=Company\|company\|Inc\|Inc\|Co\|Corporation\|

The good regular expression has both accuracy and recall around or above 50%. But a little lower than that is fine.

There are a wide variety of acceptable answers. Here are ours:

**Grammar and Syntax** Java: formally defined in a reference book. Grammaticality is crucial; ungrammatical programs are not accepted. English: unknown, never formally defined, constantly changing. Most communication is made with “ungrammatical” utterances. There is a notion of graded acceptability: some utterances are judged slightly ungrammatical or a little odd, while others are clearly right or wrong.

**Semantics** Java: the semantics of a program is formally defined by the language spec-
ificação. More pragmatically, one can say that the meaning of a particular program is the JVM code emitted by the compiler. English: no formal semantics, meaning is context dependent.

**Pragmatics and Context-Dependence** Java: some small parts of a program are left undefined in the language specification, and are dependent on the computer on which the program is run. English: almost everything about an utterance is dependent on the situation of use.

**Compositionality** Java: almost all compositional. The meaning of “A + B” is clearly derived from the meaning of “A” and the meaning of “B” in isolation. English: some compositional parts, but many non-compositional dependencies.

**Lexical Ambiguity** Java: a symbol such as “Avg” can be locally ambiguous as it might refer to a variable, a class, or a function. The ambiguity can be resolved simply by checking the declaration; declarations therefore fulfill in a very exact way the role played by background knowledge and grammatical context in English. English: much lexical ambiguity.

**Syntactic Ambiguity** Java: the syntax of the language resolves ambiguity. For example, in “if (X) if (Y) A; else B;” one might think it is ambiguous whether the “else” belongs to the first or second “if,” but the language is specified so that it always belongs to the second. English: much syntactic ambiguity.

**Reference** Java: there is a pronoun “this” to refer to the object on which a method was invoked. Other than that, there are no pronouns or other means of indexical reference; no “it,” no “that.” (Compare this to stack-based languages such as Forth, where the stack pointer operates as a sort of implicit “it.”) There is reference by name, however. Note that ambiguities are determined by scope—if there are two or more declarations of the variable “X”, then a use of X refers to the one in the innermost scope surrounding the use. English: many techniques for reference.

**Background Knowledge** Java: none needed to interpret a program, although a local “context” is built up as declarations are processed. English: much needed to do disambiguation.

**Understanding** Java: understanding a program means translating it to JVM byte code. English: understanding an utterance means (among other things) responding to it appropriately; participating in a dialog (or choosing not to participate, but having the potential ability to do so).