Software testing and quality assurance for natural language processing

Kevin Bretonnel Cohen
University of Colorado School of Medicine
Biomedical Text Mining Group Lead

kevin.cohen@gmail.com
http://compbio.ucdenver.edu/Hunter_lab/Cohen
Urban Oregonian dialect

- Vn.tv -> Vnv planting/planning
- Vn[labial] -> ũ[V[labial]] informal → ũ[formal]
- Mandatory gesundheit
- Mumbling

*Please interrupt if I am not intelligible*
Outline

• Basic principles of and approaches to software testing
  – Catalogues
  – Fault models
  – Equivalence classes
• Black-box testing
• White-box testing
• User interface testing
• Testing frameworks
• Metamorphic testing
• Error seeding
• Special issues of language
PART I: GENERAL SOFTWARE TESTING

Linguists make good software testers
Definitions

• **Bug**: Any error

• **Testing**: Looking for bugs

• **Evaluation**: Global description of performance (precision/recall/F-measure)

• **Quality assurance**: Larger activity of assuring that software is of high quality—requirements, documentation, testing, version control, etc.
Definitions

- **Bug**: “A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A [bug], if encountered during execution, may cause a failure of the component or system.” (Spillner et al. 2011)
Definitions

- **Bug**: “An error is a mistake made by a person, resulting in one or more faults, which are embodied in the program’s text. The execution of the faulty code will lead to zero or more failures.” (Marick 1995)
Definitions

- **Bug**: “...an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unexpected ways.” (Wikipedia)
Definitions

• **Bug:**
  - User interface errors
  - Error handling
  - Boundary-related errors
  - Calculation errors
  - Initial and later states
  - Control flow errors
  - Errors in handling or interpreting data
  - Race conditions
  - Load conditions
  - Hardware
  - Source and version control

(Kaner et al. 1999)
Definitions

• **Bug**: Anything that makes you look stupid.
  - Errors in functionality
  - Errors in documentation
  - Errors in requirements
  - Minor errors in output (misspellings, language usage)
Definitions

• (Levels of maturity of) **software testing**:

  0. Testing is ignored.
  1. Testing is demonstrating that the software works.
  2. Testing is demonstrating that the software doesn’t work.
  3. Testing is providing information.
  4. Testing is fully integrated into the development project.
Definitions

• (Levels of maturity of) **software testing**:
  0. Testing is ignored.
  1. Testing is demonstrating that the software works.
  2. Testing is demonstrating that the software doesn’t work.
  3. **Testing is providing information**.
  4. Testing is fully integrated into the development project.
Definitions

• **Evaluation**: What we commonly do in natural language processing.
  - Determine one or a small number of values for metrics (precision/recall/F-measure, BLEU score, ROUGE, etc.)
  - Good for global characterization of program performance
  - Bad for understanding what the program is good at, what it’s bad at, how to improve it
Definitions

• **Quality assurance**: Broad activity of ensuring that software is of high quality. Includes requirements, testing, version control, etc.
Why academics don’t think about this

“I’m doing research, not building applications.”
A Scientist’s Nightmare: Software Problem Leads to Five Retractions

*Flipping fiasco.* The structures of MsbA (purple) and Sav1866 (green) overlap little (*left*) until MsbA is inverted (*right*).
“For scientific work, bugs don’t just mean unhappy users who you’ll never actually meet: they mean retracted publications and ended careers. It is critical that your code be fully tested before you draw conclusions from results it produces.”

--Rob Knight, CU
Some basic principles of testing

- Every program has bugs.
Some basic principles of testing

• Industry average: 1-25/1000 lines of code after release

• Microsoft:
  – 10-20/1000 lines of code found during testing
  – 0.5/1000 after release
Some basic principles of testing

• It is not possible to test a program completely.
  – Every non-trivial program has an effectively infinite number of possible test cases

• The art and science of testing is to accomplish it with the minimum number of tests/amount of resources.
Some basic principles of testing

• Linguists are good software testers, even for software that is not linguistic
• Linguists are good partners for testing software that is linguistic
• Analogy to field/descriptive linguistics...
Some basic principles of testing

- Fault model: “...relationships and components of the system under test that are most likely to have faults. It may be based on common sense, experience, suspicion, analysis, or experiment. Each test design pattern has an explicit fault model.”

Binder (2000), my emphasis
Some basic principles of testing

• Test case:
  – Specified input
  – Specified result

• Specifications should come from “requirements,” if they are explicitly available
Some basic principles of testing

• Common type of bug: some combinations of options do not work correctly.
Some basic principles of testing

Learning options:
- `e [b,c]` - select between classification [c], regression [r], and preference ranking [p] (see [Joachims, 2002a])
  (default classification)
- `-c` float - C: trade-off between training error and margin (default [orig. x*B]^-1)
- `-v` [5.] - epsilon width of tube for regression (default 0.1)
- `-j` float - Cost: cost-factor, by which training errors on positive examples outweight errors on negative

<table>
<thead>
<tr>
<th>Transduction options (see [Joachims, 1999b], [Joachims, 2002a]):</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-r</code> [0..1] - fraction of unlabeled examples to be classified into the positive class (default is the ratio of positive and negative examples in the training data)</td>
</tr>
</tbody>
</table>

Kernel options:
- `-t` int - type of kernel function:
  0: linear (default)
  1: polynomial (\(s^d\))
  2: radial basis function \(\exp(-\gamma ||a-b||^2)\)
  3: sigmoid \(\tanh(s a^T b + c)\)
  4: user defined kernel from kernel.h
- `-d` int - parameter d in polynomial kernel
- `-g` float - parameter \(\gamma\) in rbf kernel
- `-s` float - parameter s in sigmoid/poly kernel
- `-t` float - parameter c in sigmoid/poly kernel
- `-a` string - parameter of user defined kernel

Optimization options (see [Joachims, 1999b], [Joachims, 2002a]):
- `-q` [2..] - maximum size of QP-subproblems (default 10)
- `-n` [0..q] - number of new variables entering the working set in each iteration (default n = q). Set n <= q to prevent size-increasing.
- `-m` [5..] - size of cache for kernel evaluations in KB (default 40)

http://svmlight.joachims.org/
Some basic principles of testing

• \((-z (c,r,p)) \times (-c \text{ float}) \times \ldots\)

• Some simplifying assumptions:
  – Reals/floats: 3 values only
  – Range of integers: (e.g. zero to 100): 3 values only
  – Ignore any switch that takes an arbitrary string
  – Ignore output options
Some basic principles of testing

• 92,980,917,360 possible test cases
• At one per second: 2,948 years
• ...and imagine hand-calculating the results...

You need a plan
Some basic principles of testing

• Consider this application:
  – Takes three integer values representing the lengths of the sides of a triangle as input...
  – …and determines whether the triangle is scalene, isosceles, or equilateral.

• List all of the inputs that you can think of for this application.
Some basic principles of testing

• **Scalene triangle**: 3 sides of unequal length and 3 unequal angles.

• **Isosceles triangle**: 2 sides of equal length and 2 equal angles.

• **Equilateral triangle**: 3 sides of equal length and 3 angles of 60 degrees.
Some basic principles of testing

1. A valid scalene triangle. (Note that this would not include all possible sets of three non-zero integers, since there are no triangles with sides of e.g. 1, 2, and 3.)

2. A valid isosceles triangle.

3. A valid equilateral triangle.

4. At least three test cases defining isosceles triangles with all possible permutations of the two equal sides.
Some basic principles of testing

5. A length of zero for at least one side.

6. A negative integer for at least one side.

7. Three integers greater than zero such that the sum of two of the lengths is equal to the third (should not return isosceles).
Some basic principles of testing

8. At least three test cases of the preceding type covering all permutations of the location of the side equal to the sum of the other two sides.

9. Three integers greater than zero such that the sum of two of the sides is less than the length of the third.

10. All three permutations of the preceding test case.
Some basic principles of testing

11. All three sides are zero.
12. At least one non-integer value.
13. The wrong number of input values.
14. Is there a specified output for every one of the test cases listed above?
Some basic principles of testing

• Highly experienced professional software developers come up with about 7.8 of these 14 tests.
• Most people come up with far less.
Some basic principles of testing

• The purpose of testing is to find problems.

• “The purpose of finding problems is to get them fixed.” (Kaner et al. 1999)
Some basic principles of testing

• Testing requires planning.
  – A one-page plan is better than no plan at all!
Some basic principles of testing

• Tests should include both “clean” and “dirty” inputs.
  – “Clean” inputs are “expected”
  – “Dirty” inputs are not “expected”

• 5:1 ratio of dirty:clean tests in “mature” testing organizations, 1:5 ratio of dirty:clean tests in immature testing organizations (McConnell 2004)
Basic principles of test case design

- Good tests:
  - ...have “a reasonable probability of finding an error”
  - ...are “not redundant”
  - ’are the “best of breed”’
  - ...are “neither too simple nor too complex”
  - ...make “program failures obvious”

(Kaner et al. 1999)
Basic principles of test case design

• **Good tests:**
  - Run fast
  - Are independent of each other
    • No test should rely on a previous test
    • Order-independent
Basic principles of test case design

- Equivalence classes: Classes of test inputs that you expect to give the same result (should find the same bug)
- Boundary conditions: Values at the “edges” of ranges or sets
Basic principles of test case design

• Equivalence classes:
  – All test the same functionality
  – All will catch the same bug
  – If one won’t catch a bug, the others won’t, either

(Kaner et al. 1999)
Basic principles of test case design

• Make a list or a table
• Don’t forget “dirty” classes!
  – Always include:
    • null input
    • empty input
  – …whenever possible.
## Basic principles of test case design

<table>
<thead>
<tr>
<th>Input or output event</th>
<th>Clean equivalence classes</th>
<th>Dirty equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter a number</td>
<td>Numbers between 1 and 99</td>
<td>empty, null, 0, &gt; 99, &lt; 1, Letters and other non-numeric characters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter a word</td>
<td>Capital letters, Lower-case letters, Non-Latin letters</td>
<td>empty, null, non-alphabetic characters</td>
</tr>
</tbody>
</table>

(Adapted from Kaner et al. 1999)
Basic principles of test case design

• Finding equivalence classes:
  – Ranges
    • Clean: within range
    • Dirty: below the low end of the range
    • Dirty: above the high end of the range
    • Dirty: different data type (e.g. non-numeric)
  – Groups
    • Positive and negative numbers
    • Zero
    • Rational numbers and integers
    • ASCII and non-ASCII character sets
Basic principles of test case design

• Look for combinations of equivalence classes
  – E.g. one parser fails just in case:
    1. ...it is provided with pre-POS-tagged input, and...
    2. ...there is white space at the end of the line.
  – One program for finding addresses failed just in case country was Canada and country name was written in all upper-case letters
Basic principles of test case design

• Some equivalence classes for a named entity recognition system for gene names:
  – Characteristics of names
  – Characteristics of context
What would constitute an equivalence class?

- For coreference:
  - Proper/common/pro- nouns
  - Singular/plural
  - Morphological/conjoined plurals
  - Proper with/without initials
  - Syntactic role (subject/object/oblique)
  - Same/different sentence
  - Length
Basic principles of test case design

- Boundary conditions
  - Best test cases from an equivalence class are at the boundaries
  - Some error types (e.g. incorrect equalities, like $>$ instead of $\geq$) only cause failures at boundaries
  - Programs that fail at non-boundaries usually fail at boundaries

(Kaner et al. 1999)
Basic principles of test case design

• A bug in the GIZA++ word alignment package:

“When training the HMM model, the matrix for the HMM trellis will not be initialized if the target sentence has only one word. Therefore some random numbers are added to the count. This bug will also crash the system when linking against [the] pthread library. We observe different alignment and slightly lower perplexity after fixing the bug.”

Gao and Vogel (2008)
What constitutes a boundary case for language?

- **Length matters**
  - Parser performance varies with length
  - Named entity recognition: GM performance drops off at $L = 5$

- **Depth**
  - Morphological derivation
  - Syntactic
Length is important for understanding Bulgarian plurals and biomedical named entity recognition...
Length effect in a hybrid biomedical NER system

Kinoshita et al. (2005)
Yeh et al. (2005)
...and the problem of telling when two names refer to the same thing

Recall: # correctly identified/# possible correct
Precision: # correctly identified/# identified

**F-measure is balanced precision and recall: 2*P*R/(P+R)**

Hirschman (2005)
What constitutes a boundary case for language?

- How many args on a variable-adiicity predicate?
- Ambiguity
  - Syntactic
  - POS ambiguity: 1-11 (J&M)
  - Word sense
Basic principles of test case design

- Other approaches to test case building
  - Combinations of equivalence classes, boundary conditions, etc.
  - State transitions
  - Load testing
  - “Error guessing”
  - Dumb monkey/smart monkey
Organizing your tests: test suites

- Test suite: A group of tests that are run together
- Easier to interpret results
- First level of division: clean/dirty tests
- Consider separating by equivalence class
- Consider combinations of equivalence classes
Organizing your tests: test suites

E.g., for code implementing calculation of an Index of Syntactic Complexity:

```java
@Test
public void nullAndEmptyInput() {
    // test cases here
}

@Test
public void subordinatingConjunctions() {
    // test cases here
}

@Test
public void whPronouns() {
    // test cases here
}
```
Organizing your tests: test suites

```java
@Test
public void verbForms() {
    // test cases here
}

@Test
public void nounPhrases() {
    // test cases here
}

@Test
public void parenthesesAndBraces() {
    // test cases here
}

@Test
public void exampleSentenceFromPaper() {
    assertEquals(21, isc.isc(input1));
}
```
Organizing your tests: test suites

• E.g., one parser fails in the following cases:
  – Input contains POS tag HYPH (segmentation fault)
  – Input file contains [ or ] instead of –LRB- or –RRB- (abort trap)
  – Input contains the token / (parse failure)

• What equivalence classes would catch these? How else might they be scattered among tests?
Combination testing

• Imagine program with
  – three command-line variables
  – ...and 100 possible values for each variable
  \[100 \times 100 \times 100 = 1,000,000\text{ test cases}\]

• Determine five equivalence classes for each and pick best representative of each
  \[5 \times 5 \times 5 = 125\]
Combination testing using the all-singles technique

• “Complete testing” defined as every value of every variable being used in at least one test
## Combination testing using the all-singles technique

<table>
<thead>
<tr>
<th>Test case</th>
<th>Variable 1</th>
<th>Variable 2</th>
<th>Variable 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>A</td>
<td>1</td>
<td>α</td>
</tr>
<tr>
<td>Test case 2</td>
<td>B</td>
<td>2</td>
<td>β</td>
</tr>
<tr>
<td>Test case 3</td>
<td>C</td>
<td>3</td>
<td>γ</td>
</tr>
<tr>
<td>Test case 4</td>
<td>D</td>
<td>4</td>
<td>δ</td>
</tr>
<tr>
<td>Test case 5</td>
<td>E</td>
<td>5</td>
<td>ε</td>
</tr>
</tbody>
</table>

Adapted from Kaner, Bach, and Pettichord (2002)
Combination testing using the all-singles technique

• Shortcoming of all-singles: important, obvious variable configurations will be missed
• Partial solution is to add these
Combination testing using the all-pairs technique

- “Complete testing” defined as every pair of values for every variable being used at least once
Combination testing using the all-pairs technique

<table>
<thead>
<tr>
<th>Test case 1</th>
<th>Variable 1</th>
<th>Variable 2</th>
<th>Variable 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 2</td>
<td>A</td>
<td>1</td>
<td>α</td>
</tr>
<tr>
<td>Test case 3</td>
<td>A</td>
<td>2</td>
<td>β</td>
</tr>
<tr>
<td>Test case 4</td>
<td>A</td>
<td>3</td>
<td>γ</td>
</tr>
<tr>
<td>Test case 5</td>
<td>A</td>
<td>4</td>
<td>δ</td>
</tr>
<tr>
<td>Test case 6</td>
<td>B</td>
<td>1</td>
<td>β</td>
</tr>
<tr>
<td>Test case 7</td>
<td>B</td>
<td>2</td>
<td>ε</td>
</tr>
<tr>
<td>Test case 8</td>
<td>B</td>
<td>3</td>
<td>δ</td>
</tr>
<tr>
<td>Test case 9</td>
<td>B</td>
<td>4</td>
<td>α</td>
</tr>
</tbody>
</table>

25 test cases needed

Adapted from Kaner, Bach, and Pettichord 2002
Combination testing using the all-pairs technique

<table>
<thead>
<tr>
<th></th>
<th>Variable 1</th>
<th>Variable 2</th>
<th>Variable 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>A</td>
<td>1</td>
<td>α</td>
</tr>
<tr>
<td>Test case 2</td>
<td>A</td>
<td>2</td>
<td>β</td>
</tr>
<tr>
<td>Test case 3</td>
<td>A</td>
<td>3</td>
<td>γ</td>
</tr>
<tr>
<td>Test case 4</td>
<td>A</td>
<td>4</td>
<td>δ</td>
</tr>
<tr>
<td>Test case 5</td>
<td>A</td>
<td>5</td>
<td>ε</td>
</tr>
<tr>
<td>Test case 6</td>
<td>B</td>
<td>1</td>
<td>β</td>
</tr>
<tr>
<td>Test case 7</td>
<td>B</td>
<td>2</td>
<td>ε</td>
</tr>
<tr>
<td>Test case 8</td>
<td>B</td>
<td>3</td>
<td>δ</td>
</tr>
<tr>
<td>Test case 9</td>
<td>B</td>
<td>4</td>
<td>α</td>
</tr>
</tbody>
</table>

25 test cases needed

Adapted from Kaner, Bach, and Pettichord 2002
Combination testing using the all-pairs technique

• Algorithm for building set of test cases
  – Program with three variables:
    • $V_1 = \{A, B, C\}$
    • $V_2 = \{1, 2\}$
    • $V_3 = \{\alpha, \beta\}$

$3 \times 2 \times 2 = 12$
Combination testing using the all-pairs technique

• Arrange variables in columns, in descending order of number of values (V1, V2, V3 or V1, V3, V2)

• Create table with V1 * V2 rows

• First column contains each value of V1 repeated V2 times

• (Hint: leave a blank row—this is hard)

• Second column contains each value of V2
Combination testing using the all-pairs technique

<table>
<thead>
<tr>
<th>Variable 1</th>
<th>Variable 2</th>
<th>Variable 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Adapted from Kaner, Bach, and Pettichord 2002
Combination testing using the all-pairs technique

- For third variable, rotate
  - $\alpha, \beta, \gamma, \delta, \varepsilon$
  - $\beta, \gamma, \delta, \varepsilon, \alpha$
  - $\gamma, \delta, \varepsilon, \alpha, \beta$
  - $\delta, \varepsilon, \alpha, \beta, \gamma$
  - $\varepsilon, \alpha, \beta, \gamma, \delta$
Combination testing using the all-pairs technique

<table>
<thead>
<tr>
<th>Variable 1</th>
<th>Variable 2</th>
<th>Variable 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>α</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>β</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>β</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>α</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>α</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>β</td>
</tr>
</tbody>
</table>

Adapted from Kaner, Bach, and Pettichord 2002
Combination testing using the all-pairs technique

<table>
<thead>
<tr>
<th>Test case</th>
<th>Variable 1</th>
<th>Variable 2</th>
<th>Variable 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>A</td>
<td>1</td>
<td>Alpha</td>
</tr>
<tr>
<td>Test case 2</td>
<td>A</td>
<td>2</td>
<td>Beta</td>
</tr>
<tr>
<td>Test case 3</td>
<td>A</td>
<td>3</td>
<td>Gamma</td>
</tr>
<tr>
<td>Test case 4</td>
<td>A</td>
<td>4</td>
<td>Delta</td>
</tr>
<tr>
<td>Test case 5</td>
<td>A</td>
<td>5</td>
<td>Epsilon</td>
</tr>
<tr>
<td>Test case 6</td>
<td>B</td>
<td>1</td>
<td>Beta</td>
</tr>
<tr>
<td>Test case 7</td>
<td>B</td>
<td>2</td>
<td>Gamma</td>
</tr>
<tr>
<td>Test case 8</td>
<td>B</td>
<td>3</td>
<td>Delta</td>
</tr>
<tr>
<td>Test case 9</td>
<td>B</td>
<td>4</td>
<td>Epsilon</td>
</tr>
<tr>
<td>Test case 10</td>
<td>B</td>
<td>5</td>
<td>Alpha</td>
</tr>
<tr>
<td>Test case 11</td>
<td>C</td>
<td>1</td>
<td>Gamma</td>
</tr>
<tr>
<td>Test case 12</td>
<td>C</td>
<td>2</td>
<td>Delta</td>
</tr>
<tr>
<td>Test case 13</td>
<td>C</td>
<td>3</td>
<td>Epsilon</td>
</tr>
<tr>
<td>Test case 14</td>
<td>C</td>
<td>4</td>
<td>Alpha</td>
</tr>
<tr>
<td>Test case 15</td>
<td>C</td>
<td>5</td>
<td>Beta</td>
</tr>
<tr>
<td>Test case 16</td>
<td>D</td>
<td>1</td>
<td>Delta</td>
</tr>
<tr>
<td>Test case 17</td>
<td>D</td>
<td>2</td>
<td>Epsilon</td>
</tr>
<tr>
<td>Test case 18</td>
<td>D</td>
<td>3</td>
<td>Alpha</td>
</tr>
<tr>
<td>Test case 19</td>
<td>D</td>
<td>4</td>
<td>Beta</td>
</tr>
<tr>
<td>Test case 20</td>
<td>D</td>
<td>5</td>
<td>Gamma</td>
</tr>
<tr>
<td>Test case 21</td>
<td>E</td>
<td>1</td>
<td>Epsilon</td>
</tr>
<tr>
<td>Test case 22</td>
<td>E</td>
<td>2</td>
<td>Alpha</td>
</tr>
<tr>
<td>Test case 23</td>
<td>E</td>
<td>3</td>
<td>Beta</td>
</tr>
<tr>
<td>Test case 24</td>
<td>E</td>
<td>4</td>
<td>Gamma</td>
</tr>
<tr>
<td>Test case 25</td>
<td>E</td>
<td>5</td>
<td>Delta</td>
</tr>
</tbody>
</table>
Combination testing using the all-pairs technique

- Algorithm works for up to five variables
- Beyond that, additional rows have to be added
- Ordering decisions are difficult after third variable
- May still need to add specific test cases
- ...still, reduction in number of test cases is very large (125 to 25 for 3-variable, 5-value case)
Finite-state approaches to software testing

• Often applied to GUI testing or other interactive programs

• Utility in non-interactive programs, as well:
  – Operating system
  – Input files/resources do/don’t exist
  – Output files do/don’t exist
  – Permissions
Finite-state approaches to software testing

- E.g. one parser throws a segmentation fault if it is passed an input file name on the command line and that file does not exist.
FSAs as Graphs

- Let’s start with the sheep language from Chapter 2 of Jurafsky and Martin – /baa+!/
Sheep FSA

- We can say the following things about this machine
  - It has 5 states
  - \(b, a, \text{ and } !\) are in its alphabet
  - \(q_0\) is the start state
  - \(q_4\) is an accept state
  - It has 5 transitions
But Note

- There are other machines that correspond to this same language

![Diagram](attachment:diagram.png)
More Formally

- You can specify an FSA by enumerating the following things.
  - The set of states: $Q$
  - A finite alphabet: $\Sigma$
  - A start state
  - A set of accept/final states
  - A transition function that maps $Q \times \Sigma$ to $Q$
About Alphabets

• Don’t take term *alphabet* word too narrowly; it just means we need a finite set of symbols in the input.

• These symbols can and will stand for bigger objects that can have internal structure.
Yet Another View

- The guts of FSAs can ultimately be represented as tables.

If you’re in state 1 and you’re looking at an a, go to state 2.

If you’re in state 1 and you’re looking at a b, go to state 0.

If you’re in state 1 and you’re looking at a !, go to state 4.

If you’re in state 1 and you’re looking at a e, go to state 3.

If you’re in state 1 and you’re looking at a !, go to state 2.

If you’re in state 1 and you’re looking at a 2,3, go to state 3.

If you’re in state 1 and you’re looking at a 4, go to state 4.

If you’re in state 1 and you’re looking at a 1, go to state 0.

If you’re in state 1 and you’re looking at a 2, go to state 2.

If you’re in state 1 and you’re looking at a 3, go to state 3.

If you’re in state 1 and you’re looking at a 4, go to state 4.

If you’re in state 1 and you’re looking at a 5, go to state 5.

If you’re in state 1 and you’re looking at a 6, go to state 6.

If you’re in state 1 and you’re looking at a 7, go to state 7.

If you’re in state 1 and you’re looking at a 8, go to state 8.

If you’re in state 1 and you’re looking at a 9, go to state 9.

If you’re in state 1 and you’re looking at a 0, go to state 0.

If you’re in state 1 and you’re looking at a 1, go to state 1.

If you’re in state 1 and you’re looking at a 2, go to state 2.

If you’re in state 1 and you’re looking at a 3, go to state 3.

If you’re in state 1 and you’re looking at a 4, go to state 4.

If you’re in state 1 and you’re looking at a 5, go to state 5.

If you’re in state 1 and you’re looking at a 6, go to state 6.

If you’re in state 1 and you’re looking at a 7, go to state 7.

If you’re in state 1 and you’re looking at a 8, go to state 8.

If you’re in state 1 and you’re looking at a 9, go to state 9.

If you’re in state 1 and you’re looking at a 0, go to state 0.

If you’re in state 1 and you’re looking at a 1, go to state 1.

If you’re in state 1 and you’re looking at a 2, go to state 2.

If you’re in state 1 and you’re looking at a 3, go to state 3.

If you’re in state 1 and you’re looking at a 4, go to state 4.

If you’re in state 1 and you’re looking at a 5, go to state 5.

If you’re in state 1 and you’re looking at a 6, go to state 6.

If you’re in state 1 and you’re looking at a 7, go to state 7.

If you’re in state 1 and you’re looking at a 8, go to state 8.

If you’re in state 1 and you’re looking at a 9, go to state 9.

If you’re in state 1 and you’re looking at a 0, go to state 0.
Generative Formalisms

- **Formal Languages** are sets of strings composed of symbols from a finite set of symbols.
- Finite-state automata define formal languages (without having to enumerate all the strings in the language).
- The term **Generative** is based on the view that you can run the machine as a generator to get strings from the language.
Generative Formalisms

• FSAs can be viewed from two perspectives:
  – Acceptors that can tell you if a string is in the language
  – Generators to produce all and only the strings in the language
Review of FSA diagrams
Case study in FSAs: Chilibot

Chen and Sharp (2004)
Case study in FSAs: Chilibot

Chen and Sharp (2004)
Case study in FSAs: Chilibot

Chen and Sharp (2004)
Chilbot searches PubMed literature database (abstracts) about specific relationships between proteins, genes, or keywords. The results are returned as a graph (see examples). We support several different search methods.

**Search for relationship between two genes, proteins or keywords**

<table>
<thead>
<tr>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDNF &amp; TRKB</td>
</tr>
<tr>
<td>BDNF &amp; polymorphism</td>
</tr>
<tr>
<td>BDNF &amp; modulate</td>
</tr>
</tbody>
</table>

**Search for relationships between many genes, proteins, or keywords**

Example:
- Apoptosis
- TrkB
- BDNF
- Nur77

**Search for relationships between two lists of genes, proteins, or keywords**

Example:
- List 1: Apoptosis, Cancer
- List 2: BDNF, Creb, TrkB

**Advanced options**
- Automated hypothesis generation [graph]
- Restricting context using keywords
- Providing your own synonyms
- Modifying synonyms provided by Chilbot
- Color coding nodes with gene expression values
- Special search: modulation
Case study in FSAs: Chilibot

• Initial states:
Case study in FSAs: Chilibot

- Three paths through the FSA
  - One clean (gene/gene/Search)
  - Two dirty
Case study in FSAs: Chilibot

- Both dirty paths catch a bug that suggests that overall, the builders did not do a good job of validating inputs.
Case study in FSAs: Chilibot

BDNF & TRKB

Retrieving synonyms..

Retrieving abstracts .. |BDNF/TRKB * ............

Performing linguistic analysis ..
Case study in FSAs: Chilibot

Retrieving synonyms...

Retrieving abstracts.. No relevant literature was found. Exit. view query History.
White-box testing

• **Black-box testing**
  – “Functional” testing
  – No knowledge of software internals
  – Completely based on requirements

• **White-box testing**
  – Access to software internals
  – Still rooted in requirements!
  – Static and dynamic analysis
Code coverage defined

• How much of your code was executed when you ran a given set of tests?
  – How many lines were executed...
  – ...branches were traversed
  – ...functions/classes were called
  – ...

Code coverage defined

Test cases

• No blood pressures
• Blood pressure present, equals 152
• Blood pressure present, equals 150
• ...equals 90
• ...equals 88
• Two blood pressures present

```perl
if ($line =~ /<bps>(\d+)</bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE";
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE";
    } else {
        print "NORMOTENSIVE";
    }
} // close while-loop through line
```
Test cases
• No blood pressures

```perl
if ($line =~ /<bps>(\d+)</bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE";
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE";
    } else {
        print FEATURES "NORMOTENSIVE";
    }
} // close while-loop through line
```
Code coverage defined

Test cases

• No blood pressures
• Blood pressure present, equals 150

```perl
if ($line =~ /<bps>(\d+)</bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE";
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE";
    } else {
        print FEATURES "NORMOTENSIVE";
    }
} // close while-loop through line
```
Code coverage defined

Test cases

• No blood pressures
• Blood pressure present, equals 150
• Blood pressure present, equals 152

```perl
if ($line =~ /<bps>(\d+)</bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE";
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE";
    } else {
        print FEATURES "NORMOTENSIVE";
    }
} // close while-loop through line
```
Test cases

• No blood pressures
• Blood pressure present, equals 150
• Blood pressure present, equals 152
• ...equals 90

```perl
if ($line =~ /<bps>(\d+)</bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE"
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE"
    } else {
        print FEATURES "NORMOTENSIVE"
    }
} // close while-loop through line
```
Test cases

• No blood pressures
• Blood pressure present, equals 150
• Blood pressure present, equals 152
• ...equals 90
• ...equals 88

```perl
if ($line =~ /<bps>(\d+)</bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE";
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE";
    } else {
        print "NORMOTENSIVE";
    }
} // close while-loop through line
```
Code coverage defined

Test cases

- No blood pressures
- Blood pressure present, equals 150
- Blood pressure present, equals 152
- ...equals 90
- ...equals 88
- Two blood pressures present

```perl
if ($line =~ /bps>(\d+)bps>/) {
    $blood_pressure = $1;
    if ($blood_pressure > 150) {
        print FEATURES "HYPERTENSIVE";
    } elsif ($blood_pressure < 90) {
        print FEATURES "HYPOTENSIVE";
    } else {
        print "NORMOTENSIVE";
    }
} // close while-loop through line
```
Types of coverage

• **Line coverage or statement coverage:** Goal is to execute every line of code.

• **Branch coverage or decision coverage:** Goal is to go down each branch.

• **Condition coverage:** Goal is to make each atomic part of a conditional take every possible value.

• **Condition coverage > branch coverage > line coverage (very weak)**
Working with white box testing

• First step is to build a control flow diagram.

Figure from Spillner et al. (2011)
Working with white box testing

• All statements (nodes) can be reached by a single test case. What is it?

Figure from Spillner et al. (2011)
Working with white box testing

- All statements (nodes) can be reached by a single test case. What is it?

a, b, f, g, h, d, e

Figure from Spillner et al. (2011)
Working with white box testing

- To reach branch coverage, what additional test cases do we need?

Figure from Spillner et al. (2011)
Working with white box testing

• a, b, c, d, e
• a, b, f, g, i, g, h, d, e
• a, k, e
Working with white box testing

- Crucial difference is in treatment of ELSE conditions—100% branch coverage will find missing statements, 100% line coverage will not.
Working with white box testing

- Full coverage not the same as taking all paths through the code
- Generally can’t detect “sins of omission”
- Won’t necessarily catch all classes of errors—STILL NEED TO THINK ABOUT EQUIVALENCE CLASSES AND OTHER TOOLS FROM BLACK-BOX TESTING
Unit testing via an interface is a combination of white-box testing and black-box testing, sometimes known as “translucent box testing” or “gray box testing”
You haven’t tried working with control flow diagrams, but want to know what your coverage is:

- Java: Cobertura
- Perl: Devel::Cover
- Python: PyDEV or Coverage.py

Testing done without measuring code coverage typically tests only 50-60% of statements (Wiegers 2002 in McConnell 2004)
Testing frameworks

- Provides functionality for running and reporting on tests
- Specific to individual computer languages
  - Java: JUnit
  - Perl: Test::Simple
  - Python: PyUnit
Testing frameworks

• Typically include a setup/”teardown” method
• Think back to initial states
Testing frameworks: JUnit

• Command-line version provides textual output

• Graphical versions available for IDEs ("interactive" or "integrated" development environment) such as Eclipse

```java
java junit.textui.TestRunner org.example.antbook.junit.SimpleTest
```

Time: 0.01

OK (1 tests)

java.sun.com/developer/Books/javaprogramming/ant/ant_chap04.pdf
Various methods allow different kinds of comparisons and assertions about their results

```java
@Test
public void testSize() {
    HashMap hm = new HashMap();
    assertEquals(0, hm.size());
    hm.add("1", "one");
    assertEquals(1, hm.size());
}
```
Testing frameworks: JUnit

• Optional comments that are output in case of failure

```java
assertEquals("New HashMap should return size 0", 0, hm.size());
```
Testing frameworks: JUnit

• Very partial list:
  – assertEquals()
  – assertTrue()
  – assertFalse()
  – assertArrayEquals()
  – assertNotNull()
  – assertNull()
User interface testing

• Designer is striking a balance between:
  – Functionality
  – Time to learn how to use the program
  – How well the user remembers how to use the program
  – Speed of performance
  – Rate of user errors
  – User’s satisfaction with the program

(Kaner et al. 1999)
User interface testing

- Applies to all kinds of user interfaces:
  - Graphical
  - Command line
Test-driven programming

• Write tests **first**—they can define the requirements.
• Leads to more easily testable code.
• “Writing test cases before the code takes the same amount of time and effort as writing the test cases after the code, but it shortens defect-detection-debug-correction cycles.” (McConnell 2004)
Error seeding

• Methodology for estimating adequacy/efficiency of testing effort
• Add errors in the code (e.g. flip comparators)
• How many of the seeded errors are found?
Metamorphic testing

• Characterize broad changes in behavior that should correspond to specific changes in inputs
• Requires considerable domain knowledge
Open-Source tools for automating testing

• **Performance/load testing:**
  – Apache Jmeter
  – Open STA

• **Functionality testing:**
  – HttpUnit
  – Selenium
  – Water
  – Bad Boy
Software is special, and so is language

PART II: CASE STUDIES
Simple(ish) case: numerical calculations
Simple(ish) case: checking linguistic values
More interesting case: checking for state
Software is special, and so is language

PART III: SPECIAL ISSUES OF LANGUAGE AND NLP
“Software is different...”

- If you are driving your car and...
  - ...you adjust the volume downwards on the radio...
  - ...while making a left turn...
  - ...with the windshield wipers set to intermittent...
- ...the muffler falls off.
...and so is language...

• Verbs that take infinitival complements take *to*-phrases, unless:
  
  • ...the verb is *try*...
    • I’ll try to convince you.
    • I’ll try and convince you.
    • I want to convince you.
    • * I want and convince you.
  
  • ...and it’s *not* inflected.
    • * I’m trying and convince you.
    • * I tried and convince you.
...and software testers know it.

- What counts as a boundary case?
- What would constitute an equivalence class?
- What things interact?
The naturally occurring data assumption: a paraphrase, but not a caricature

- What do you mean, testing? You test your code by running it over a large corpus.
The system under test

- OpenDMAP (Hunter et al. 2008)
- Highest performer on one of the BioCreative PPI task measures (Baumgartner et al. 2008)
- Semantic parser

{interaction} := [interactor1] {interaction-verb} the? [interactor2]
Not too big, not too trivial

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser</td>
<td>3,982</td>
</tr>
<tr>
<td>Rule-handling</td>
<td>2,311</td>
</tr>
<tr>
<td>Configuration</td>
<td>731</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7,024</strong></td>
</tr>
</tbody>
</table>

Table 1: Distribution of lines of code in the application.
Table 2: Distribution of functional tests.

<table>
<thead>
<tr>
<th>Test type</th>
<th>Number of tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>85</td>
</tr>
<tr>
<td>Pattern/rule</td>
<td>67</td>
</tr>
<tr>
<td>Patterns only</td>
<td>90</td>
</tr>
<tr>
<td>Slots</td>
<td>9</td>
</tr>
<tr>
<td>Slot nesting</td>
<td>7</td>
</tr>
<tr>
<td>Slot property</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>278</td>
</tr>
</tbody>
</table>
# Methods: What the developer did

<table>
<thead>
<tr>
<th>Rule</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^*$</td>
<td>$B$</td>
</tr>
<tr>
<td></td>
<td>$A$</td>
</tr>
<tr>
<td></td>
<td>$AAA$</td>
</tr>
<tr>
<td></td>
<td>$AB$</td>
</tr>
<tr>
<td></td>
<td>$BA$</td>
</tr>
<tr>
<td></td>
<td>$BAB$</td>
</tr>
<tr>
<td></td>
<td>$AAAB$</td>
</tr>
<tr>
<td></td>
<td>$AAAB$</td>
</tr>
<tr>
<td></td>
<td>$BAAA$</td>
</tr>
<tr>
<td></td>
<td>$BAAA$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Materials: The corpus & rules

• BioCreative II protein-protein interaction task document collection
  – 3.9 million words
• 98 semantic grammar rules
Methods: Assessing coverage

- Cobertura (Mark Doliner, cobertura.sourceforge.net)
  - Whole application
  - Parser package alone
  - Rules package alone
  - Line coverage
  - Branch coverage
  - Class coverage
Methods: Three conditions

• Test suite versus corpus
• Varying the size/contents of the rule set
• Varying the size of the corpus
## Experiment 1: Test suite versus corpus

<table>
<thead>
<tr>
<th>Metric</th>
<th>Functional tests</th>
<th>Corpus, all rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall line coverage</td>
<td>56%</td>
<td>41%</td>
</tr>
<tr>
<td>Overall branch coverage</td>
<td>41%</td>
<td>28%</td>
</tr>
<tr>
<td>Parser line coverage</td>
<td>55%</td>
<td>41%</td>
</tr>
<tr>
<td>Parser branch coverage</td>
<td>57%</td>
<td>29%</td>
</tr>
<tr>
<td>Rules line coverage</td>
<td>63%</td>
<td>42%</td>
</tr>
<tr>
<td>Rules branch coverage</td>
<td>71%</td>
<td>24%</td>
</tr>
<tr>
<td>Parser class coverage</td>
<td>88% (22/25)</td>
<td>80% (20/25)</td>
</tr>
<tr>
<td>Rules class coverage</td>
<td>100% (20/20)</td>
<td>90% (18/20)</td>
</tr>
</tbody>
</table>
Experiment 2: Varying the size/contents of the rule set

<table>
<thead>
<tr>
<th>Corpus, all rules</th>
<th>nominal rules</th>
<th>verbal rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>41%</td>
<td>41%</td>
<td>41%</td>
</tr>
<tr>
<td>28%</td>
<td>28%</td>
<td>28%</td>
</tr>
<tr>
<td>41%</td>
<td>41%</td>
<td>41%</td>
</tr>
<tr>
<td>29%</td>
<td>29%</td>
<td>29%</td>
</tr>
<tr>
<td>42%</td>
<td>42%</td>
<td>42%</td>
</tr>
<tr>
<td>24%</td>
<td>24%</td>
<td>24%</td>
</tr>
<tr>
<td>80% (20/25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90% (18/20)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The table below shows the results of line and branch coverage for different packages when examining individual classes.

<table>
<thead>
<tr>
<th>Package</th>
<th>Line coverage $\geq$</th>
<th>Line coverage $&gt;$</th>
<th>Branch coverage $\geq$</th>
<th>Branch coverage $&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes in parser package</td>
<td>21/25</td>
<td>14/25</td>
<td>19/21</td>
<td>18/21</td>
</tr>
<tr>
<td>Classes in rules package</td>
<td>19/20</td>
<td>6/20</td>
<td>11/11</td>
<td>11/11</td>
</tr>
</tbody>
</table>

Table 4: When individual classes were examined, both line and branch coverage were always higher with the functional tests than with the corpus. This table shows the magnitude of the differences. $\geq$ indicates the number of classes that had equal or greater coverage with the functional tests than with the corpus, and $>$ indicates just the classes that had greater coverage with the functional tests than with the corpus.
“Closure properties”
Time is money

- Developer-written tests: median run time **11 seconds**
- Corpus: median run time **4 hours 27 minutes 51 seconds**
• Evaluation is not bad—running a giant corpus through your application is good!

• **Code coverage is not perfect:**
  – Cannot detect “sins of omission”
  – May not detect looping problems
  – Can’t detect faulty requirements

• **Ours was not good! But as soon as we tried to increase it...**
Code coverage defined

- Affected by the test suite and the code — code can be difficult to achieve coverage of, or not

```java
if (i_count >= 1) {
    // do something
} else if ((i_count) < 1) {
    // do something else
} else {
    // this code is not reachable
}
```
Code coverage defined

• Affected by the test suite and the code — code can be difficult to achieve coverage of, or not

```c
if (i_count >= 1) { do_something(); } else if (i_count < 1) { do_something_else(); }
```
Running a large body of data through your code is not bad!

- BioNLP company building statistical language modeling toolkit
- Health record documents scarce, small
- Misinterpretation of 0 only showed up when customer tried our product on 1M words of text
…but, test suites have many advantages

- Faster to run
- Systematic coverage of all functions
- Control of redundancy
- Coverage of rare and “dirty” conditions
- Control of data—easy to interpret
- Easy to add test cases found from large data sets!
Test suite generation for named entity recognition

- fuculokinase
- Trp-1
- BRCA1
- Heat shock protein 60
- calmodulin
- dHAND
- suppressor of p53

- cheap date
- lush
- ken and barbie
- ring
- to
- the
- there
- a
Test suite generation for named entity recognition

• Entities
  – NAT1, myoglobin, ...

• Contexts

NAT1 polymorphisms may be correlated with an increased risk of larynx cancer.

<> polymorphisms may be correlated with an increased risk of larynx cancer.
Generation process

• Plug entities into contexts

NAT1 polymorphisms may be correlated with an increased risk of larynx cancer.

Insulin polymorphisms may be correlated with an increased risk of larynx cancer.

Test data
Equivalence classes and boundary conditions for entities

• Four categories:
  – Orthographic/typographic
  – Morphosyntactic
  – Source features
  – Lexical resource features
Equivalence classes and boundary conditions for entities

- Orthographic/typographic features:
  - Length: characters for symbols, whitespace-tokenized words for names
  - Case:
    - All upper-case
    - All lower-case
    - Upper-case-initial only
    - Mixed

- Nat-1
- N-acetyltransferase 1
- Pray For Elves
- putative tumor suppressor 101F6
- INNER NO OUTER
- Out at first
Equivalence classes and boundary conditions for entities

- One system missed every one-word name
- One system missed lower-case-initial names in sentence-initial position
- One system only found multi-word names if each word was upper-case-initial
Equivalence classes and boundary conditions for entities

- Orthographic/typographic features:
  - Numeral-related:
    - Whether or not entity name contains a numeral
    - Whether numeral is Arabic or Roman
    - Position of numeral within the entity:
      - Initial
      - Medial
      - Final

- 18-wheeler
- elongation factor 1 alpha
- androgen-induced 1
- angiotensin II
Equivalence classes and boundary conditions for entities

- One system missed numerals at the right edge of names
- One system only found multi-word names if there was an alphanumeric postmodifier

- alcohol dehydrogenase 6
- spindle A
Equivalence classes and boundary conditions for entities

- **Punctuation-related features**
  - Whether or not entity contains punctuation
  - Count of punctuation marks
  - Which punctuation marks (hyphen, apostrophe, etc.)

- **One system missed names, but not symbols, that contained hyphens**

- **One system missed names containing apostrophes whenever they were in genitives**

- **N-acetyltransferase 1**
- **Nat-1**
- **e(olfC)**
- **5’ nucleotidase precursor**
- **corneal dystrophy of Bowman's layer type II (Thiel-Behnke)**
Equivalence classes and boundary conditions for entities

• Greek-letter-related features
  – Whether or not entity contains Greek letter
  – Position of the letter
  – Format of the letter

• Two systems had format-related failures

• PPAR-delta
• beta1 integrin
• [beta]1 integrin
Equivalence classes and boundary conditions for entities

- Morphosyntactic features
  - Name or symbol
  - Function words
    - Present or absent
    - Number of function words
    - Position in the entity

- One system performed well on symbols but did not recognize any names at all.

- $N$-acetyltransferase 1
- $NAT1$
- scott of the antarctic
Equivalence classes and boundary conditions for entities

- Features related to inflectional morphology
  - Whether or not entity contains:
    - Nominal number morphology
    - Genitive morphology
    - Verbal participial morphology
  - Positions of words in entities that contain these morphemes

- bag of marbles
- Sjogren's syndrome nuclear autoantigen 1
- apoptosis antagonizing transcription factor
Equivalence classes and boundary conditions for entities

- **Source or authority features**
  - Database
  - Website
  - Document identifier of document in which observed

- **Lexicographic features**
  - Presence in a lexical resource
  - OOV or in vocabulary for a language model
Equivalence classes and boundary cond. for contexts

• True positive context, or challenging false positive
Equivalence classes and boundary cond. for contexts

• Features for true positive contexts:
  – Count of slots
  – Sentential position of slot(s)
  – Typographic context (tokenization/punctuation)
  – List context
  – Appositive
  – Syntactic features
Equivalence classes and boundary cond. for contexts

• Features for true positive contexts
  – Syntactic/semantic features:
    • Preceding word a species name?
    • Following word a keyword?
    • Preceding word POS:
      – Article?
      – Adjective?
      – Conjunction?
      – Preposition?
Equivalence classes and boundary cond. for contexts

• Features for challenging false positive sentences
  – Keywords
  – Orthographic/typographic features of a token in the sentence
  – Morphological features of apparent word endings such as -in, -ase
Evaluation

- Generated simple test suites varying only:
  - Entity length
  - Case
  - Hyphenation
  - Sentence position
Evaluation

• 5 EI systems
  – AbGene (ours)
  – Yapex
  – KeX
  – CCP (ours)
  – Ono et al. (EI only)
Can we find bugs?
Bugs in every system

• Fine on symbols, except when lc-initial and sentence-initial
• Fine with medial numbers, fine with final letters, failed on final numbers
  – Glucose 6 phosphate dehydrogenase
  – Alcohol dehydrogenase 6
  – Protein kinase C
• Fine with apostrophes except in genitives
• Fine on symbols with hyphens, failed on names with hyphens
• Missed every possible one-word name
Bugs in every system

• Fine on symbols, except when lc-initial and sentence-initial
• Fine with medial numbers, fine with final letters, failed on final numbers
  – Glucose 6 phosphate dehydrogenase
  – Alcohol dehydrogenase 6
  – Protein kinase C
• Fine with apostrophes except in genitives
• Fine on symbols with hyphens, failed on names with hyphens
• Missed every possible one-word name
You can’t predict P/R/F for a corpus based on this, but...
Predictions on a single system

Can we predict performance on an equivalence class?

Hypothesis: performance on an equivalence class in a structured test suite predicts performance on that equivalence class in a corpus.
Predictions on a single system

Method:

1. Run several simple test suites through the system
   • Length, case, hyphenation, sentence position
2. Make predictions
3. Run corpora through the system
   • BioCreative
   • PMC (PubMed Central)
Predictions on a single system

1. BAD R: Numerals in initial position
   - 12-LOX, 18-wheeler

2. BAD R: Contain stopwords
   - Pray for elves, ken and barbie

3. BAD R: Sentence-medial upper-case-initial

4. BAD R: 3-character-long symbols

5. GOOD R: Numeral-final names
   - Yeast heat shock protein 60
### Results

#### BioCreative

<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>57</td>
<td>17</td>
<td>.17</td>
<td>.41</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>38</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>556</td>
<td>278</td>
<td>512</td>
<td>.67</td>
<td>.52</td>
</tr>
<tr>
<td>5</td>
<td>284</td>
<td>251</td>
<td>72</td>
<td>.53</td>
<td>.80</td>
</tr>
</tbody>
</table>

#### PubMed Central

<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>.44</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1.0</td>
<td>.33</td>
</tr>
<tr>
<td>4</td>
<td>163</td>
<td>64</td>
<td>188</td>
<td>.72</td>
<td>.46</td>
</tr>
<tr>
<td>5</td>
<td>108</td>
<td>54</td>
<td>46</td>
<td>.67</td>
<td>.70</td>
</tr>
</tbody>
</table>

- **BioCreative**
  - P = .65
  - R = .68
- **PMC**
  - P = .71
  - R = .62
Are there general principles of NER test suite construction?
Concept recognition systems

• **Given:**
  – Gene Ontology (~32,000 concepts)
  – *In mice lacking ephrin-A5 function, cell proliferation and survival of newborn neurons...* (PMID 20474079)

• **Return:**
  – GO:0008283 cell proliferation

• **Performance tends to be low**
Two paradigms of evaluation

• Traditional approach: use a corpus
  • Expensive
  • Time-consuming to produce
  • Redundancy for some things...
  • ...underrepresentation of others (Oepen et al. 1998)
  • Slow run-time (Cohen et al. 2008)

• Non-traditional approach: structured test suite
  • Controls redundancy
  • Ensures representation of all phenomena
  • Easy to evaluate results and do error analysis
  • Used successfully in grammar engineering
Goals

• Big picture: How to evaluate ontology concept recognition systems?
• Narrow goal of this work: Test hypothesis that we can use techniques from software testing and descriptive linguistics to build test suites that overcome the disadvantages of corpora and find performance gaps
• Broad goal of this work: Are there general principles for test suite design?
Methods

• Experiment 1: Build a structured test suite and apply it to an ontology concept recognition system
• Experiment 2: Compare to other test suite work (Cohen et al. 2004) to look for common principles
Structured testing defined

• Systematic exploration of paths and combinations of features in a program
  – Theoretical background: set theory
  – Devices: state machines, grammars
How to build a structured test suite

• Steps: list factors that might affect system performance and their variations

• Assemble individual test cases that instantiate these variations...

• ...by using insights from linguistics and from how we know concept recognition systems work
  – Structural aspects: length
  – Content aspects: typography, orthography, lexical contents (function words)...

• ...to build a structured set of test cases
Structured test suite

Canonical
• GO:0000133 Polarisome
• GO:0000108 Repairosome
• GO:0000786 Nucleosome
• GO:0001660 Fever
• GO:0001726 Ruffle
• GO:0005623 Cell
• GO:0005694 Chromosome
• GO:0005814 Centriole
• GO:0005874 Microtubule

Non-canonical
• GO:0000133 Polarisomes
• GO:0000108 Repairosomes
• GO:0000786 Nucleosomes
• GO:0001660 Fevers
• GO:0001726 Ruffles
• GO:0005623 Cells
• GO:0005694 Chromosomes
• GO:0005814 Centrioles
• GO:0005874 Microtubules
Structured test suite

Features of terms
• Length
• Punctuation
• Presence of stopwords
• Ungrammatical terms
• Presence of numerals
• Official synonyms
• Ambiguous terms

Types of changes
• Singular/plural variants
• Ordering and other syntactic variants
• Inserted text
• Coordination
• Verbal versus nominal constructions
• Adjectival versus nominal constructions
• Unofficial synonyms
Structured test suite

• Syntax
  – induction of apoptosis ➔ apoptosis induction

• Part of speech
  – cell migration ➔ cell migrated

• Inserted text
  – ensheathment of neurons ➔ ensheathment of some neurons
Methods/Results I

- Gene Ontology, revision 9/24/2009
- Canonical: 188
- Non-canonical: 117

- Observation:
  - 5:1 “dirty” versus 5:1 “clean” is mark of “mature” testing

- Applied publicly available concept recognition system
Results

- No non-canonical terms were recognized
- 97.9% of canonical terms were recognized
  - All exceptions contain the word *in*
- What would it take to recognize the error pattern with canonical terms with a corpus-based approach??
Methods/Results II

• Compare dimensions of variability to those in Cohen et al. (2004)
  – Applied structured test suite to five named entity recognition systems
  – Found errors in all five

• 11 features in Cohen et al. (2004), 16 features in this work
Methods/Results II

• Shared features were:
  – Length
  – Numerals
  – Punctuation
  – Function/stopwords
  – Syntactic context
  – Canonical form in source

• Shared boundary conditions: length and punctuation

• ....so, linguistics is important
Some basic principles of testing

• Fault model: “…relationships and components of the system under test that are most likely to have faults. It may be based on common sense, experience, suspicion, analysis, or experiment. Each test design pattern has an explicit fault model.”

Binder (2000), my emphasis
A fault model for ontology mapping, alignment, and linking systems that work on lexical methods
Mapping, Alignment, & Linking of Ontologies (MALO)

- Brenda Tissue Ontology
- Cell Type Ontology
- Gene Ontology
- Chemical Entities of Biological Interest
Is this Cell Type term the concept that is being referred to in this GO term?
Lexical methods for MALO

• exact match

Example:

CL: T cell ➔ GO: T cell proliferation
Lexical Methods for MALO

- exact match
- synonyms

Example:

BTO: T-lymphocyte  →  GO: negative T-cell selection

synonym: T-cell
Lexical methods for MALO

- exact match
- synonyms
- text processing

Example:

CH: spermatogonium → GO: spermatogonial cell division

Stemming: using an algorithm which determines the morphological root of a word by removing suffixes.
Lexical Methods for MALO

- exact match
- synonyms
- text processing

Example:

❌ CH: L-lysinate → GO: L-lysine metabolism
Results from an ontology linking system

Correctness of ontology terms linked to GO using a stemming technique

- CELL TYPE: 80%
- CHEBI: 70%
- BRENDA: 10%

Percentage
Driving questions

• Why does one ontology show different performance?
• What characteristics do these errors have?
• How much do particular error types contribute to overall system performance?
• How can we compare this system to other systems?
• How do we make our system better?
Fault Model

Definition:
An explicit hypothesis about potential sources of error in a system (Binder, 1999)

Method:
- hypothesize sources of error
- categorize errors (481) by source
- calculate inter-judge agreement
- analyze results from applying fault model
- make suggestions for system improvement based on analysis of fault model results
Lexical MALO fault model

- Lexical ambiguity errors
  - biological polysemy
  - General English polysemy
  - ambiguous abbreviations
  - phrase boundary mismatch

- Text processing errors
  - tokenization
  - removal of digits
  - removal of punctuation
  - removal of stop words
  - stemming

- Errors resulting from matching metadata
Inter-judge agreement

- Lexical ambiguity errors
  - biological polysemy 98%
  - General English polysemy 80%
  - ambiguous abbreviations 99%
  - phrase boundary mismatch

- Text processing errors
  - Tokenization 27%
  - removal of digits 100%
  - removal of punctuation 100%
  - removal of stop words 100%
  - stemming 100%

- Errors resulting from matching metadata 100%
Inter-judge agreement

- Lexical ambiguity errors
  - biological polysemy 98%
  - General English polysemy 80%
  - ambiguous abbreviations 99%
  - phrase boundary mismatch ----

- Text processing errors
  - tokenization 27%
  - removal of digits 100%
  - removal of punctuation 100%
  - removal of stop words 100%
  - stemming 100%

- Errors resulting from matching metadata 100%
Lexical ambiguity errors

Example:

**BTO: cone**
Def: A mass of ovule-bearing or pollen-bearing scales or bracts in trees of the pine family ...

**GO: cone cell fate commitment**
Def: The process by which a cell becomes committed to become a cone cell

biological polysemy
56%

ambiguous abbreviation
44%
Lexical ambiguity errors

Example:

**CL**: band form neutrophil
**Synonym**: band

Def: A late neutrophilic metamyelocyte in which the nucleus is in the form of a curved or coiled band, not having acquired the typical multilobular shape of the mature neutrophil.

**GO**: preprophase **band** formation

Def: The process of marking the position in the cell where cytokinesis will occur in cells that perform cytokinesis by cell plate formation.

<table>
<thead>
<tr>
<th>Biological polysemy</th>
<th>56%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambiguous abbreviation</td>
<td>44%</td>
</tr>
</tbody>
</table>
Lexical ambiguity errors

Example:

CH: thymine
Synonym: T

GO: negative regulation of CD8-positive T cell differentiation

biological polysemy
56%

ambiguous abbreviation
44%
Text processing errors

Example:

BTO: 697 cell
GO: fat cell differentiation

digit removal 51%
punctuation removal 27%
stop word removal 14%
stemming 6%
Text processing errors

Example:

BTO: 697 cell
GO: fat cell differentiation
Text processing errors

Example:

**CH:** carbon(1+)

**GO:** carbon-monoxide dehydrogenase (acceptor) activity
Text processing errors

Example:

**CH:** carbon\(^{(1+)}\)  
**GO:** carbon-monoxide dehydrogenase (acceptor) activity
Text processing errors

Example:

CL: receptor cell
GO: ... receptor on the cell ...

digit removal 51%
punctuation removal 27%
stop word removal 14%
stemming 6%
Text processing errors

Example:

**CL:** receptor cell

**GO:** ... receptor on the cell ...
Text processing errors

Example:

CH: monocarboxylates
GO: The directed movement of monocarboxylic acids into ...
## Text processing errors

**Example:**

CH: **monocarboxylates**  
GO: The directed movement of **monocarboxylic** acids into ...
Close analysis of stemming output

<table>
<thead>
<tr>
<th>Matches</th>
<th>-al</th>
<th>-ate</th>
<th>-ation</th>
<th>-e</th>
<th>-ed</th>
<th>-ic</th>
<th>-ing</th>
<th>-ize</th>
<th>-ous</th>
<th>-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>19</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>157</td>
</tr>
<tr>
<td>Incorrect</td>
<td>1</td>
<td>17</td>
<td>3</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>39</td>
</tr>
</tbody>
</table>
Close analysis of stemming output

<table>
<thead>
<tr>
<th>Matches</th>
<th>-al</th>
<th>-ate</th>
<th>-ation</th>
<th>-e</th>
<th>-ed</th>
<th>-ic</th>
<th>-ing</th>
<th>-ize</th>
<th>-ous</th>
<th>-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>19</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>157</td>
</tr>
<tr>
<td>Incorrect</td>
<td>1</td>
<td>17</td>
<td>3</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>39</td>
</tr>
</tbody>
</table>
Close analysis of stemming output

<table>
<thead>
<tr>
<th>Matches</th>
<th>-al</th>
<th>-ate</th>
<th>-ation</th>
<th>-e</th>
<th>-ed</th>
<th>-ic</th>
<th>-ing</th>
<th>-ize</th>
<th>-ous</th>
<th>-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>19</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>157</td>
</tr>
<tr>
<td>Incorrect</td>
<td>1</td>
<td>17</td>
<td>3</td>
<td>26</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>39</td>
</tr>
</tbody>
</table>
Example:

BTO: lip

GO: sphinganine metabolism
Def: The chemical reactions involving ... [http://www.chem.qmul.ac.uk/iupac/lipid/lip1n2.html#p18]
Match to metadata

Example:

BTO: lip

GO: sphinganine metabolism

Def: The chemical reactions involving ... [http://www.chem.qmul.ac.uk/iupac/lipid/lip1n2.html#p18]
Distribution of errors by ontology

Percent of types of error within an ontology

- Brenda
- Cell Type
- ChEBI

- Digit removal
- Punctuation removal
- Stop word removal
- Stemming
- Biological polysemy
- Ambiguous abbreviation
Distribution of errors by term

Number of Errors

Number of Terms

697 cell
BY-2 cell
blood plasma
T-84 cell
Reduce lexical ambiguity

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Terms removed</th>
<th>Projected increase in precision</th>
<th>Projected decrease in recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRENDA</td>
<td>697 cell, BY-2 cell, blood plasma, T-84 cell</td>
<td>27%</td>
<td>41%</td>
</tr>
<tr>
<td>Cell Type</td>
<td>band form neutrophil, neuroblast</td>
<td>4%</td>
<td>3%</td>
</tr>
<tr>
<td>ChEBI</td>
<td>iodine, L-isoleucine residue groups</td>
<td>2%</td>
<td>2%</td>
</tr>
</tbody>
</table>
Fault model is relevant to other MALO systems

<table>
<thead>
<tr>
<th></th>
<th>Stemming</th>
<th>Digit</th>
<th>Punctuation</th>
<th>Stop Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mork 03</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Sarkar 03</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Zhang 03</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Lambrix 04</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lambrix 05</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Burgun 04</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bodenreider &amp; Hayamizu 05</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burgun 05</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bodenreider 05</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luger 05</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Johnson 06</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
Fault model is relevant to other MALO systems

<table>
<thead>
<tr>
<th>The removal of</th>
<th>Stemming</th>
<th>Digit</th>
<th>Punctuation</th>
<th>Stop Word</th>
<th>Spelling</th>
<th>Case</th>
<th>Word Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mork 03</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sarkar 03</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Zhang 03</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lambrix 04</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lambrix 05</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Burgun 04</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Bodenreider &amp; Hayamizu 05</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Burgun 05</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Bodenreider 05</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Luger 05</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Johnson 06</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
Concrete suggestions and further directions

• Evaluate the suitability of each MALO technique with respect to the ontology

• Improve performance by applying text processing techniques judiciously

• Continue research that addresses inherent polysemy in language and ontologies
  – Limit search terms to 3 characters or more (Burgun and Bodenreider, 2005)
  – Investigate whether one word synonyms are useful or detrimental for detecting relationships?

• Use scalable, inexpensive methods of performance improvement that do not require domain knowledge
Conclusions

• Principled fault model can be applied consistently to MALO error data
• Applying a fault model reveals previously unknown types of error
• Both text processing and lexical ambiguity are substantial sources of error in MALO systems
• Software engineering methods and linguistic analysis are useful techniques for system evaluation
Conclusions of the tutorial: Two paradigms of evaluation

- Traditional approach: use a corpus
  - Expensive
  - Time-consuming to produce
  - Redundancy for some things...
  - ...underrepresentation of others (Oepen et al. 1998)
  - Slow run-time (Cohen et al. 2008)

- Non-traditional approach: structured test suite
  - Controls redundancy
  - Ensures representation of all phenomena
  - Easy to evaluate results and do error analysis
Conclusions of the tutorial: how to approach testing

• If your software is linguistic, consult with a linguist about how to test it
• Any planning is better than no planning
Thank you
Да отидем на вечеря
Acknowledgements

• Helen Johnson
• Biomedical Text Mining Group and Software Engineering Group in the Computational Bioscience Program, U. Colorado School of Medicine
• MapQuest.com
• Bill Baumgartner, Martha Palmer, Larry Hunter