Utilizing Inductive Logic to Fix Bugs in Python Scripts

Data Science capstone project

Latif.Masud@spsmail.cuny.edu

2018

Contents

[Problem Description 2](#_Toc514202974)

[Current State 2](#_Toc514202975)

[Links 3](#_Toc514202976)

[Hypothesis 3](#_Toc514202977)

[Introduction 3](#_Toc514202978)

[Recognizing Syntax Typos 3](#_Toc514202979)

[Collecting Function Names 3](#_Toc514202980)

[Measuring Distance Between Strings 4](#_Toc514202981)

[Hamming Distance 4](#_Toc514202982)

[Levenshtein Distance 5](#_Toc514202983)

[Jaro–Winkler Distance 6](#_Toc514202984)

[Algorithms Analysis 7](#_Toc514202985)

[Putting It All Together 7](#_Toc514202986)

[Classifying Python Errors 7](#_Toc514202987)

[Building Classification System 8](#_Toc514202988)

[Classifying Errors 9](#_Toc514202989)

[Generate Line Number for Error 10](#_Toc514202990)

[Recognize Python Error 10](#_Toc514202991)

[Error Classification with Action 10](#_Toc514202992)

[Fixing Python Exceptions 11](#_Toc514202993)

[Examples 12](#_Toc514202994)

[NameError 13](#_Toc514202995)

[Script 13](#_Toc514202996)

[Output 13](#_Toc514202997)

[TypeError 13](#_Toc514202998)

[Script 13](#_Toc514202999)

[Output 13](#_Toc514203000)

[IndexError 14](#_Toc514203001)

[Script 14](#_Toc514203002)

[Output 14](#_Toc514203003)

[ZeroDivisionError 14](#_Toc514203004)

[Script 14](#_Toc514203005)

[Output 14](#_Toc514203006)

[Conclusion 14](#_Toc514203007)

# Problem Description

In the quality assurance process, one of the most time-consuming elements is the process of analyzing the result of a test and making a decision based on the result. With current machine learning and artificial intelligence techniques, it is possible to let machines make some of the decisions, and thus cut down the amount of time it takes to get through test cases.

In particular, machines are already capable of detecting when an error has occurred that is related to code. For example, a machine knows when a script has run a bad line of code, and will terminate running code past this point. Currently, human inspection is required to observe the error, and then kick off the test case again. However, if a machine were to be taught what error signatures look like, then it itself could identify the error, make the modification, and kick off the test case again.

# Current State

One of the areas that seems a perfect fit to solve the problem described is Inductive Logic Programming (ILP). An inductive logic is a type of logic that is based on probability instead of definite facts (which would be considered deductive reasoning). An example of deductive reasoning would be: all humans are mortal, Socrates is human; therefore, Socrates is mortal. An example of inductive reasoning would be: 90% of all humans are right handed, Adam is a human; therefore, Adam is right handed.

There are a number of universities and research paper that explain both the math and concept behind ILP, but I was not able to find a paper that directly linked the usage of ILP to solve error in Python. Still, one of the best papers I came across was a paper by Dr. Hendrik Blockeel written in 2002. In the paper, he noted that most research studies in ILP at that time focused on decisions based on a single variable ILP, whereas his paper defined approaches to multi-variable ILP decision making.

There were a number of Python packages and open source code bases that implemented ILP, but they focused on solving relational problems such as the finding the relationship between two nodes based on some given paraments, or methods to keep track of data in a database.

### Links

* Dr. Blockeel’s Paper: <https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume16/blockeel02a-html/node1.html>
* IDL Implementation in Python: <https://github.com/johntrimble/foil-python>

# Hypothesis

In one sentence: Utilizing machine learning techniques with inductive logic programming can be used to cut down on human error in Python scripts.

The application of these techniques would lead machines having to rely on human presence to solve menial problems. For example, if a developer accidentally typed “prnt” instead of “print”, a machine would be able to figure out via ILP that the developer really meant “print” since it is a very close match. By allowing the machine to apply a bit of intelligence to the problem, the developer would save time and thus be more productive.

# Introduction

In order to create a self-healing Python scripts, we need to do two things: recognize typos (i.e. human errors) and recognize actual Python errors. If we can achieve the step of recognizing the errors, then we can use them to build a classification system and typecast errors. Once this is done, we can then train a classifier with existing array of datasets to build a machine learning algorithm that can heal the scripts.

# Recognizing Syntax Typos

To recognize whether or not a line contains a syntax errors, we need to do the following:

* Have the predefined Python keywords available[[1]](#footnote-1). Alternatively, we can also use Python’s predefined keywords library.
* Have all the function names the file contains be available for comparison.
* Utilize various techniques to measure the distance between two strings to find the most probable option, which we will look at below.

## Collecting Function Names

To collect all the function names in a Python file, I wrote the following code to extract the names.

#### Code

|  |
| --- |
| def find\_str(str\_to\_search, sub\_str):  try:  index = str\_to\_search.index(sub\_str)  except:  return -1  return index  def extract\_functions(filepath):  sample\_file = open(filepath, "r")  sample\_file\_contents = sample\_file.read()  sample\_file.close()  sample\_file\_lines = sample\_file\_contents.split("\n")  function\_names = []  for line in sample\_file\_lines:  if find\_str(line, "def") != -1:  function\_parts = line.split("def ")  func\_name = function\_parts[1].split("(")[0]  function\_names.append(func\_name)  return function\_names |

#### Results

* Running the file listed in the Appendix section returns the following list of functions: ['hello', 'add', 'subtract', 'multiply']

## Measuring Distance Between Strings

Inductive Logic Programming, as stated in sections above, is the approach of making a decision based on probability. In our case, we want to know how close two words are to one another in terms of letters and pattern. For example, “prin” should return a high degree of similarity when compared to “print”. On the flipside, comparing “prin” to “if” or “for” should return very low similarity scores. There are numerous ways of performing this task, but we will look at three of the most commonly used ways below.

### Hamming Distance

**Definition**: the distance between two strings of equal length is the number of positions at which the corresponding symbols are different. [1][[2]](#footnote-2)

To calculate hamming distance, we iterate through the characters of a string, and score a zero if the character is the same on both strings, and a 1 for when the characters are different. Some examples are:

* Cat and Cut would have a distance of 1
* Prin\_ and Print would also have a score of 0
* Prin and for would have a score of 4

As we can see, the lower the hamming distance, the more similar the two strings are to one another, and thus we could potentially use the hamming distance to potentially fix a typo.

#### Code

|  |
| --- |
| def hamdist(str1, str2):  diffs = 0  for ch1, ch2 in zip(str1, str2):  if ch1 != ch2:  diffs += 1  return diffs |

#### Results

* hamdist(“cut”, “cat”) -> 1
* hamdist(“prin”, “for”) -> 3
* hamdist(“print”, “some\_random\_string”) -> 4

### Levenshtein Distance

**Definition**: Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.[[3]](#footnote-3)

To calculate the levenshtein distance between two strings, we simply look for how many substitutions need to be made in order to get one string to look like the other. Some examples would be:

* “cut” and “cat” would require one change.
* “pront” and “print” would require one change
* “faree” and “false” would require two changes

#### Code[[4]](#footnote-4)

|  |
| --- |
| import numpy as np  def levenshtein(seq1, seq2):  size\_x = len(seq1) + 1  size\_y = len(seq2) + 1  matrix = np.zeros ((size\_x, size\_y))  for x in xrange(size\_x):  matrix [x, 0] = x  for y in xrange(size\_y):  matrix [0, y] = y  for x in xrange(1, size\_x):  for y in xrange(1, size\_y):  if seq1[x-1] == seq2[y-1]:  matrix [x,y] = min(  matrix[x-1, y] + 1,  matrix[x-1, y-1],  matrix[x, y-1] + 1  )  else:  matrix [x,y] = min(  matrix[x-1,y] + 1,  matrix[x-1,y-1] + 1,  matrix[x,y-1] + 1  )  return (matrix[size\_x - 1, size\_y - 1]) |

#### Results

* levenshtein(“cut”, “cat”) -> 1.0
* levenshtein(“pront”, “print”) -> 1.0
* levenshtein(“pront”, “false”) -> 5.0

### Jaro–Winkler Distance

**Definition**: the Jaro–Winkler distance is a string metric for measuring the edit distance between two sequences. The Jaro distance between two words is the minimum number of single-character transpositions required to change one word into the other.[[5]](#footnote-5)

The Jaro-Winkler distance is a bit difficult to explain, but from my research, it seems to be one of the core metrics used to calculate the similarity of two strings. For the code, I used a pre-written Python library called pyjarowinkler to calculate the scores.

#### Code

|  |
| --- |
| from pyjarowinkler import distance  def jaro\_winkler(seq1, seq2):  return distance.get\_jaro\_distance(seq1, seq2, winkler=True, scaling=0.1) |

#### Results

* jaro\_winkler(“cut”, “cat”) -> 0.8
* jaro\_winkler(“prnt”, “print”) -> 0.95
* jaro\_winkler(“prnt”, “false”) -> 0.0

### Algorithms Analysis

As we can see, all three of these distance measurement algorithms offer us different types of insights into how two strings are different. We would use the Jaro-Winkler distance as our core metric to drive our decision making and get good results by itself. But since both the Levenshtein and Hamming distance offer additional information about the *type* of mismatch there is between two strings, we could use that information to fine tune predictions.

## Putting It All Together

Now that we have ways of measuring how far two strings are from one another and know what to measure against, the code below actually does the measurements, and returns the most likely word that needs to be used instead.

[TODO: Implement Hamming and Levenshtein distances into function]

#### Code

|  |
| --- |
| def get\_python\_keywords():  keywords\_file = open(keywords\_file\_path, "r")  keywords\_file\_content = keywords\_file.read()  keywords\_file.close()    return keywords\_file\_content.split("\n")  def closest\_keyword(filepath, seq):  keywords = extract\_functions(filepath) + get\_python\_keywords()  highest\_score = -1  closest\_word = ""  for word in keywords:  score= jaro\_winkler(seq, word)  if score > highest\_score:  highest\_score = score  closest\_word = word    return closest\_word |

#### Results

* closest\_keyword(filepath, “prtn”) -> print
* closest\_keyword(filepath, “ade”) -> add
* closest\_keyword(filepath, “sbutract”) -> subtract

# Classifying Python Errors

Recognizing Python errors are somewhat simplified due to Python’s Exceptions module. This module documents all of the different ways that a piece of Python code can break. There are a myriad of options, but for this project, I will only focus on attempting to classify a few simple ones to prove that the concept works.

#### Code

|  |
| --- |
| def classify\_error(code\_string):  try:  exec(code\_string)  except IndexError:  return "IndexError" # Index in an array doesn't exist  except TypeError:  return "TypeError" # Attempting to mix two types that can't operate with one another  except ZeroDivisionError:  return "ZeroDivisionError" # When dividing by zero  except NameError:  return "NameError" # when a name/variable does not exist |

#### Results

* classify\_error("1/0") -> ZeroDivisionError
* arr = [1]; classify\_error(“arr[6]”) -> IndexError

As we can see, the Exceptions module makes the scope of this project far easier. We can easily run a piece of code and figure out the classification for the error. Using this information, we can build a classification system, and then train it.

# Building Classification System

To build a classification system, we would first fix all of the typos with the script below. The script goes through, and inductively replaces all the typos found, with the closest Jaro-Winkler match. Once this is done, the script can be run.

#### Code

|  |
| --- |
| def fix\_typos(error\_script\_path):  file\_ref = open(error\_script\_path, "r")  file\_contents = file\_ref.read()  file\_ref.close()  file\_lines = file\_contents.split("\n")  new\_lines = ""  for line in file\_lines:  calls = line.split("(")  #print(line)    fixes = []  for call in calls:  # checking for the call not being a def  if (find\_str(call, "def") == -1) and (find\_str(call, ",") == -1) \  and (find\_str(call, ")") == -1) :  parms = call.split(" ")  parms = list(filter(None, parms))  if len(parms) > 0:  function\_name = parms[0]  closest = closest\_keyword(error\_script\_path, function\_name)  fixes.append((function\_name, closest))    fixed = line  for fix in fixes:  fixed = line.replace(fix[0], fix[1])    new\_lines += fixed + "\n"    return new\_lines |

#### Result

For this part, I can input the following small python script, which has two errors that are highlighted:

|  |
| --- |
| def add(a,b):  return a + b  print(ade(1,2))  prit("hello") |

The output is:

|  |
| --- |
| def add(a,b):  return a + b  print(add(1,2))  print("hello") |

We can see that all four the function calls in the file were properly recognized, and in the case of the two typos, they were fixed! The next step is to reconstruct the script, and then run it.

# Classifying Errors

For this portion, I decided to focus on four specific types of Python errors:

* **IndexError**: when attempting to go to an index in array that doesn’t exist.
* **TypeError**: when attempting to mix two types that cannot operate with one another.
* **ZeroDivisionError**: when dividing by zero
* **NameError**: when a name/variable does not exist

For each of these errors, I first needed to recognize the exact error and the precise line that the error occurs in, which is what the following two functions do.

## Generate Line Number for Error

This is a pretty simple piece of code. All it does is loop through each line in a file, and when it hits an error, it stops executing, and returns the line number in the file, and the specific line.

|  |
| --- |
| def error\_line(code):  n = 1  for line in code.split("\n"):  try:  exec(line)  n = n + 1  except:  break  return (n, line) |

## Recognize Python Error

This function utilizes Python’s linecache library to produce the specific error string that a Python Exception raises, and also some other useful information such as the file that the error occurs in, and the relative line number that the error occurs at.

|  |
| --- |
| import linecache  import sys  def identify\_exeception():  exc\_type, exc\_obj, tb = sys.exc\_info()  f = tb.tb\_frame  lineno = tb.tb\_lineno  filename = f.f\_code.co\_filename  linecache.checkcache(filename)  line = linecache.getline(filename, lineno, f.f\_globals)  #print('EXCEPTION FOUND IN ({}, LINE {} "{}"): {}'.format(filename, lineno, line.strip(), exc\_obj))  return (filename, lineno, line.strip(), exc\_obj) |
|  |

## Error Classification with Action

This function extends the initial classification function shown in earlier part. The function takes in the body of code, runs it, and then classifies the error like before, but now, it also returns the specific location the error happened, and provides custom information for each error type:

* For NameError: returns the unknown variable name.
* For TypeError: returns the correct type, and the line the error occurred in.
* For ZeroDivisionError: returns the line the error occurred in.
* For IndexError: returns the line the error occurred in.

|  |
| --- |
| import re  def classify\_error(code\_string):  try:  exec(code\_string)  except NameError:  exception = identify\_exeception()  err\_vars = re.findall(r'\'\w+\'', str(exception[3]))  return ("NameError", err\_vars[0].replace("'", ""))    except TypeError:  exception = identify\_exeception()  line = error\_line(code\_string)  error\_type = re.findall(r'(must be )(\w+)', str(exception[3]))[0][1]  return ("TypeError", (error\_type, line))    except ZeroDivisionError:  line = error\_line(code\_string)  return("ZeroDivisionError", line)    except IndexError:  line = error\_line(code\_string)  return("IndexError", line)    return (None, "") |
|  |

## Fixing Python Exceptions

This is the last function in the process. At this point, I now have ways to fix all of the possible typos that may have arisen, and a couple of helper functions defined in the steps above that can recognize the line where an error happens, the exact classification, and some custom information.

The function below takes into account all of the previous parts, and utilizes the custom information provided to take a specific action for each error:

* **NameError**: finds the closest Jaro-Winkler relative and replaces the invalid variable with it.
* **TypeError:** converts to the suggested type provided by the Python Exceptions library
* **ZeroDivisionError:** attempts to divide by one to solve the problem
* **IndexError:** attempts to access the very last element in the array to try to alleviate the problem.

If any of these routes do not work, we assume that the inductive logic has failed, and thus needs human intervention to resolve the error.

|  |
| --- |
| import re  def fix\_errors(error\_script\_path):  file\_ref = open(error\_script\_path, "r")  code = file\_ref.read()  file\_ref.close()  error\_type, data = classify\_error(code)  if error\_type == "NameError":  replacement = closest\_keyword(error\_script\_path, data)  code = code.replace(data, replacement)  print(code)  elif error\_type == "TypeError":  correct\_type, line\_info = data  line\_number, line\_txt = line\_info  line\_vars = line\_txt.split(" ")[1:]  filtered = []  for t in line\_vars:  if t.isalnum():  line\_txt = line\_txt.replace(t, correct\_type+"("+t+")")  lines = code.split("\n")  lines[line\_number -1] = line\_txt  new\_code = "\n".join(lines)  print(new\_code)  elif error\_type == "ZeroDivisionError":  line\_txt = data[1].replace("/0", "/1")  lines = code.split("\n")  lines[line\_number -1] = line\_txt  new\_code = "\n".join(lines)  print(new\_code)  elif error\_type == "IndexError":  line\_number, line\_txt = line\_info  lines = code.split("\n")  lines[line\_number -1] = re.sub("([\(\[]).\*?([\)\]])", "\g<1>-1\g<2>", line\_txt)  new\_code = "\n".join(lines)  print(new\_code) |
|  |

# Examples

All of the examples shown below were generated by running the fix\_errors function shown above wit the following sample scripts.

## NameError

### Script

|  |
| --- |
| def add(a,b):  return a + b  print(ade(1,2))  print("hello") |
|  |

### Output

|  |
| --- |
| Message:  EXCEPTION FOUND IN (<ipython-input-137-9c3564cb92cf>, LINE 5 "exec(code\_string)"): name 'ade' is not defined  **Response**:  def add(a,b):  return a + b  print(add(1,2))  print("hello") |
|  |

## TypeError

### Script

|  |
| --- |
| x = "2"  y = x + 1  name = "latif"  name + 5 |
|  |

### Output

|  |
| --- |
| Message:  EXCEPTION FOUND IN (<ipython-input-137-9c3564cb92cf>, LINE 5 "exec(code\_string)"): must be str, not int  **Response**:  x = "2"  y = str(x) + str(1)  name = "latif"  name + 5 |
|  |

## IndexError

### Script

|  |
| --- |
| xs = [1,2,3]  print(x[5]) |
|  |

### Output

|  |
| --- |
| xs = [1,2,3]  print(xs[-1]) |
|  |

## ZeroDivisionError

### Script

|  |
| --- |
| x = 1  y = x/0 |
|  |

### Output

|  |
| --- |
| x = 1  y = x/1 |
|  |

# Conclusion

Overall, this capstone went in a very different direction than I expected even half way through the semester. My original goal was to utilize machine learning with a Bayesian model to statistically “teach” a system the most correct route to take to solve a certain Exception. This route was something that never happened, so I decided to hard code my actions for each classification. With that plan, I was able to solve some of the common errors found in Python scripts.

1. Source: <https://www.programiz.com/python-programming/keyword-list> [↑](#footnote-ref-1)
2. Source: <https://en.wikipedia.org/wiki/Hamming_distance> [↑](#footnote-ref-2)
3. Source: <https://en.wikipedia.org/wiki/Levenshtein_distance> [↑](#footnote-ref-3)
4. Source: <http://stackabuse.com/levenshtein-distance-and-text-similarity-in-python/> [↑](#footnote-ref-4)
5. Source: <https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance> [↑](#footnote-ref-5)