Bachelor Thesis: Automatic Token Classification for Unknown Languages

Joel Guggisberg, Jan Kurš

1 Introduction

Given code of an unknown programming language, attempt to automatically recognize which are the keywords of the language.

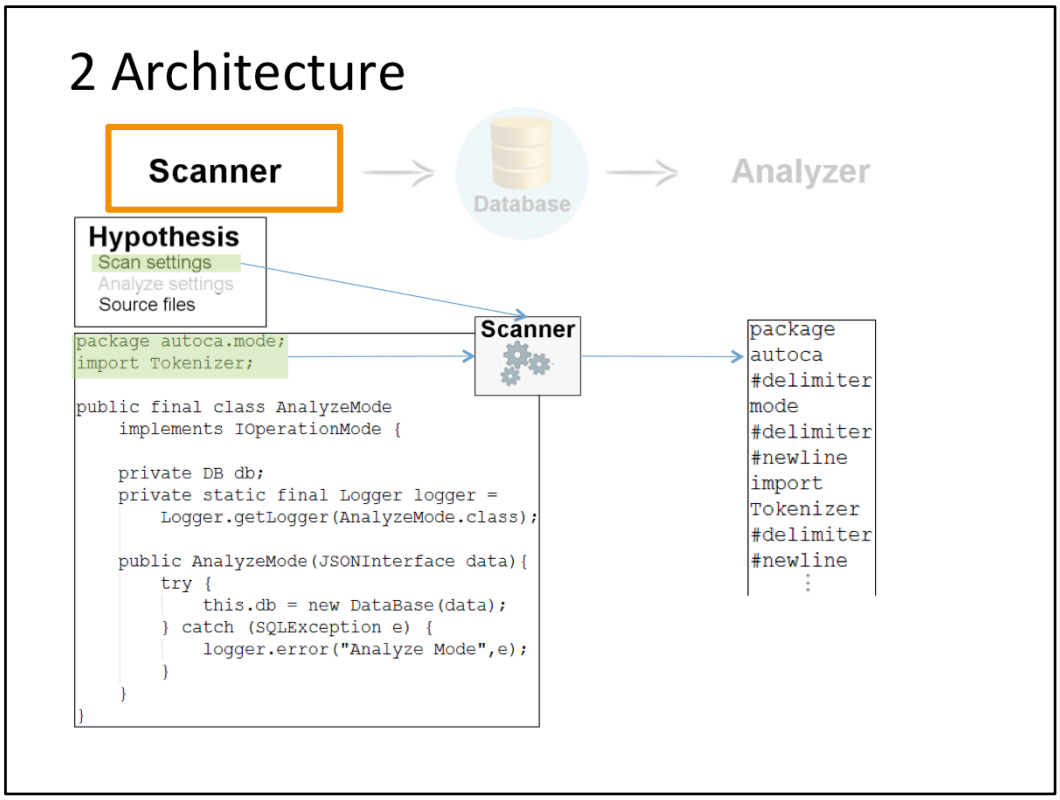
Example:

Keywords

```
package autoca mode
import Tokenizer

public final class AnalyzeMode
    implements IOperationMode

    private DB db
    private static final Logger logger
        Logger getLogger AnalyzeMode class

    public AnalyzeMode JSONInterface data
        try
            this db new DataBase data
        catch SQLException e
            logger error Analyze Mode e
```

Assume you have code of an unknown programming language,
The progeamm developped tries to automatically find the keywords of the language.

So first we are given code of an unknown programming language, like the example on the right. (click)
The first task is to split it into tokens: which by definition is is a meaningful character sequences.(click)
After that programms tries to identify the keywords of the language. Here identified in blue.



To find said keywords, hypothesis on how they can be found in all those tokens had to be made.

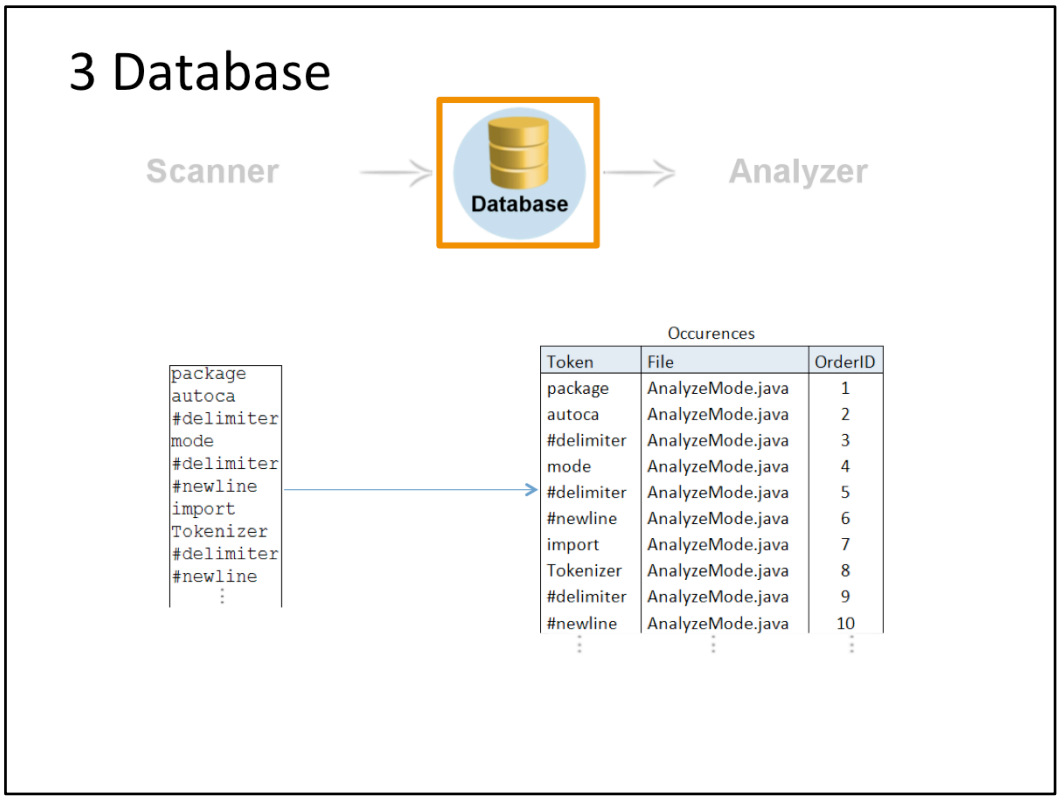
The reach our goal of the program, we had to automatically test different hypothesis on source code provided.
Our first attempt at the architecture of the program failed becaue it was not customizable
and the runtime increased exponentially when more data was added.

This new architecture runs linearly on any amount input and is designed to be customizable
so we can easily change and test steps in these hypothesis as we progress.

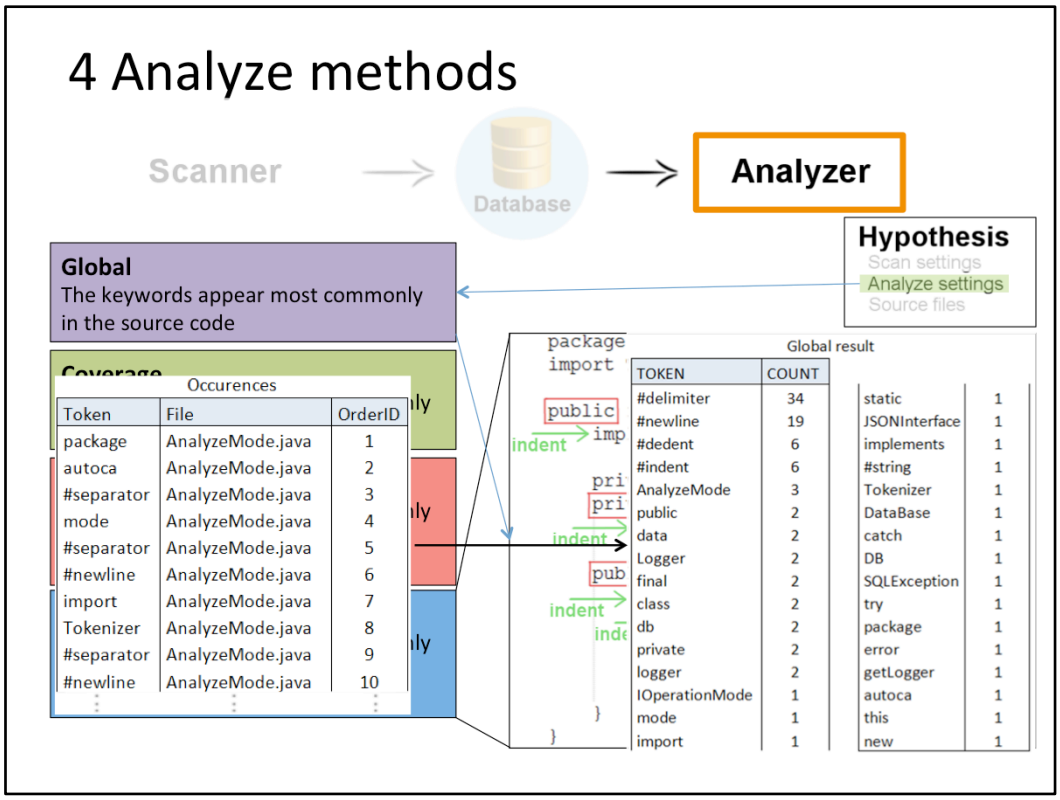
These hypothesis that we want to verify are described by an object called hypothesis. (click)

The first step when running such a test is to input the hypothesis object into the scanner.

Now Lets pass our sample code from before into the scanner to see what it does:
So first the scanner loads the scanner settings from the hypothesis object, describing what a tokens is.
Then it scans the source code files and tokenizes them into a list of tokens.



The results from the Scanner are then stored relationally in the database.
The main table is the occurences table, which holds the exact position of each token in the structure of the code files.

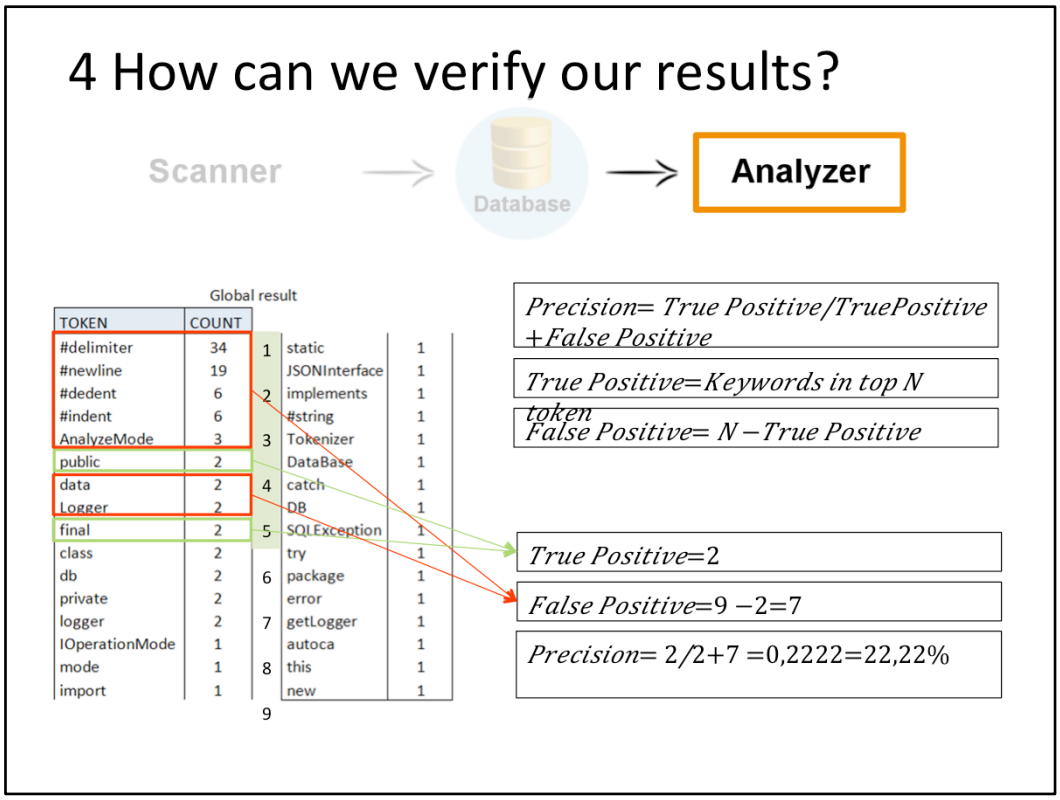


In the analyzer are currently 4 hypothesis implemented on where in the code the keywords can be found.

(click)The first one is the Global hypothesis it rates the token higher the more they appear in the source code.
(click) the second one is coverage it rates token higher the more they appear per file
(click) the third is Newline it rate it rates token higher the more they appear on the first line
(click) the fourth is indent it rates token higher the more they appear on at the first position of a line before an inden.
To explain this here is a picture(click)

Lets go back to our example that we scanned and saved in the dateabas and pass it to the analyzer.
The analyze settings in the hypothesis object contains information on which hypothesis we want to execute.

For our example we use the global hypothesis , to generate this global result table on the right.
Note that in the global result table the likelihood for a token to be a keyword increases by its count.
So we would say using global hypothesis our token that is most likely to be a keyword is #delimiter

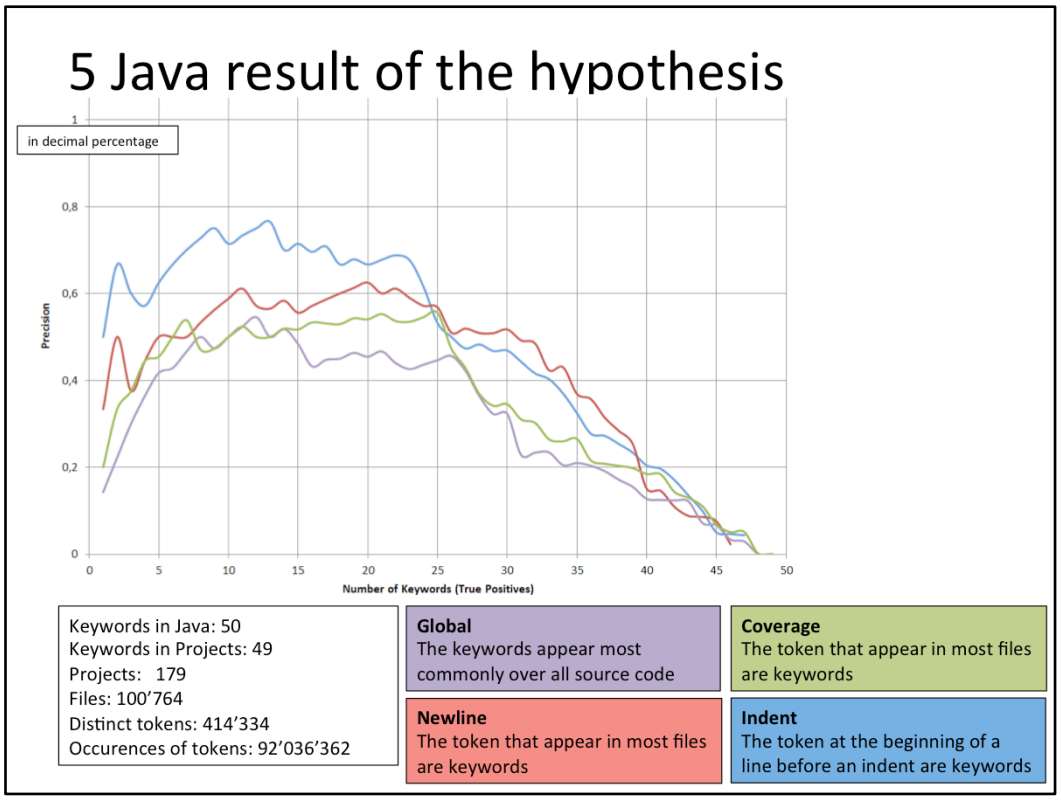


So now we have an orderd table with rated tokens of which the topmost are supposed to be keywords, but how do we verify them?

We achieved this by finding the real java keywords in the result table and calculating the precision at their position.

Lets calculate the precision for the keyword «final» as an example:
1) First find final
2) Count position
3) Calculate the precision

So the precision to find 2 keywords is 22,22 percent.



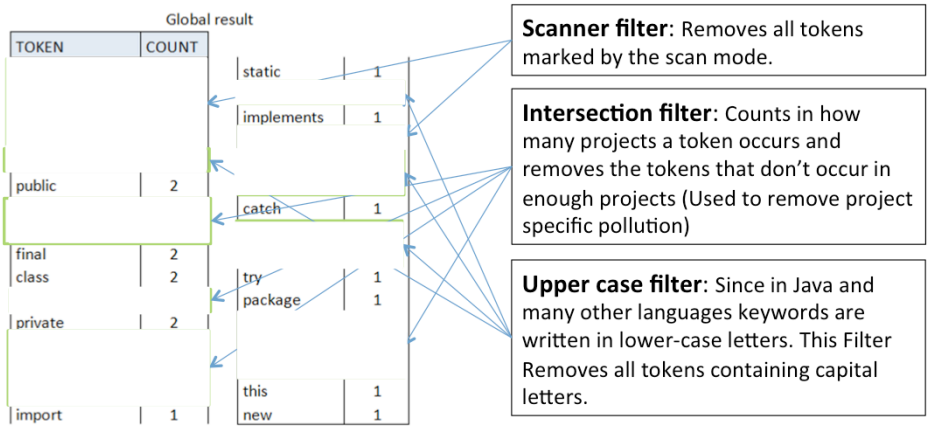
Now let's have a look at the results of the actual data.
We used was 179 java projects from github which after the scanner resulted in 92 million entries in the main table of the database.
Java has 50 keywords and in the 179 project we found 49.

This Graph shows the precision as calculated on the slide before for each Java keyword in each hypothesis:

Lets say we want to find the top 20 keywords with each hypothesis.
the Global hypothesis in purple gives us a precision of 45%
the Coverage in green 52%
the Newline in red 62%
and Indent in blue 78%

6 Filters

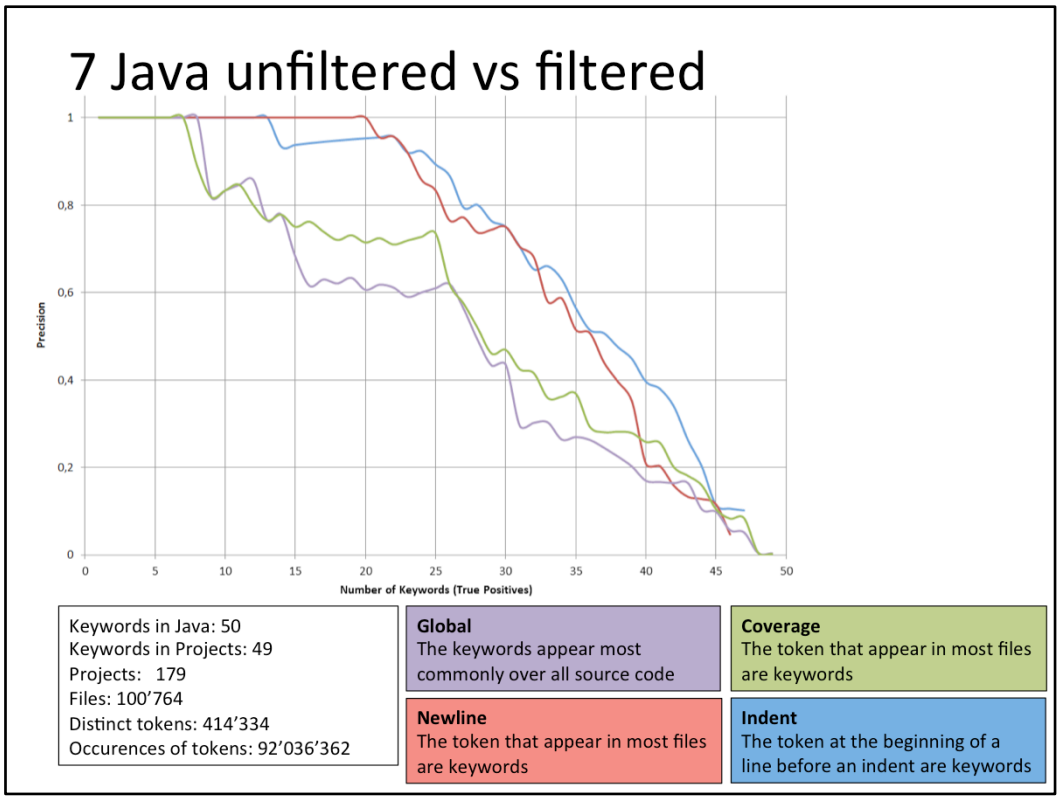
How can we improve those results?



The question is now where to go from here, how are we going to improve the results?

We compared the result tables and tried to find commonalities between wrongly identified keywords.
Finally we ended up implementing three filters which are applied after each analysis to improve the results.

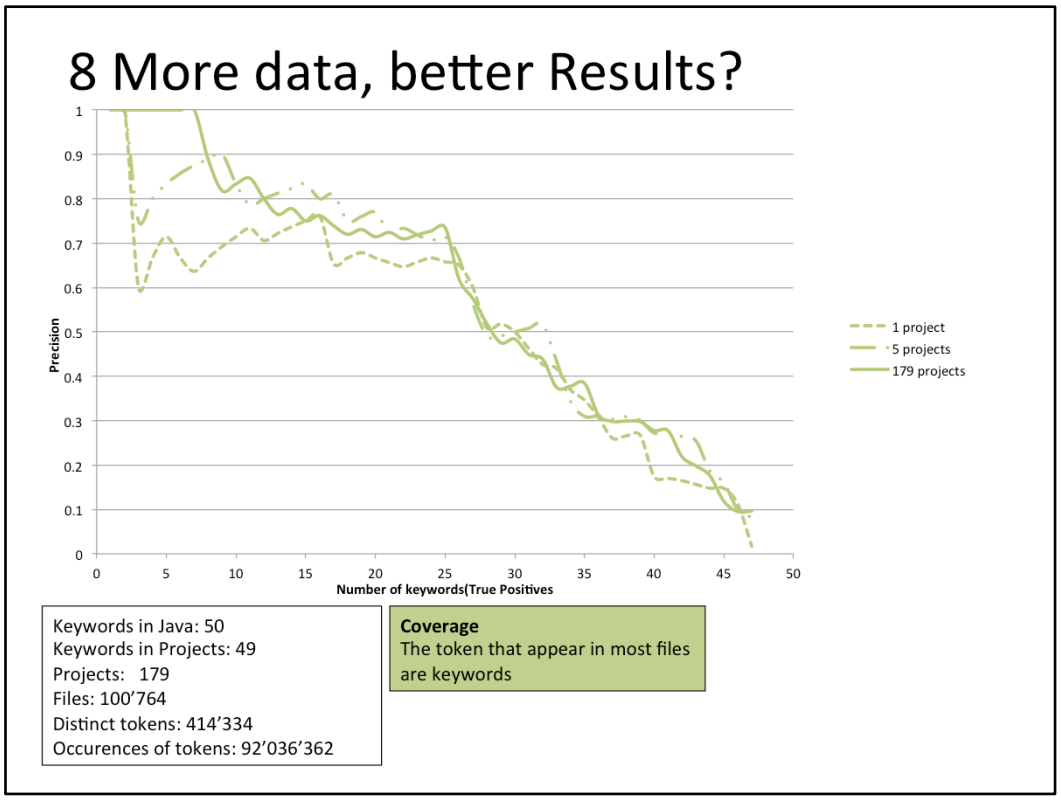
First the scanner filter, which gets rid of the tokens tagged by the scanner.
Second the Intersection filter, it count in how many projects a token occurs and removes tokens that dont occur
In enough projects.
Finally the Upper case filter: which removes tokens containing capital letters.



Here are the unfiltered results again and as you can see the precision to find 1 keyword is roughly 40% only.

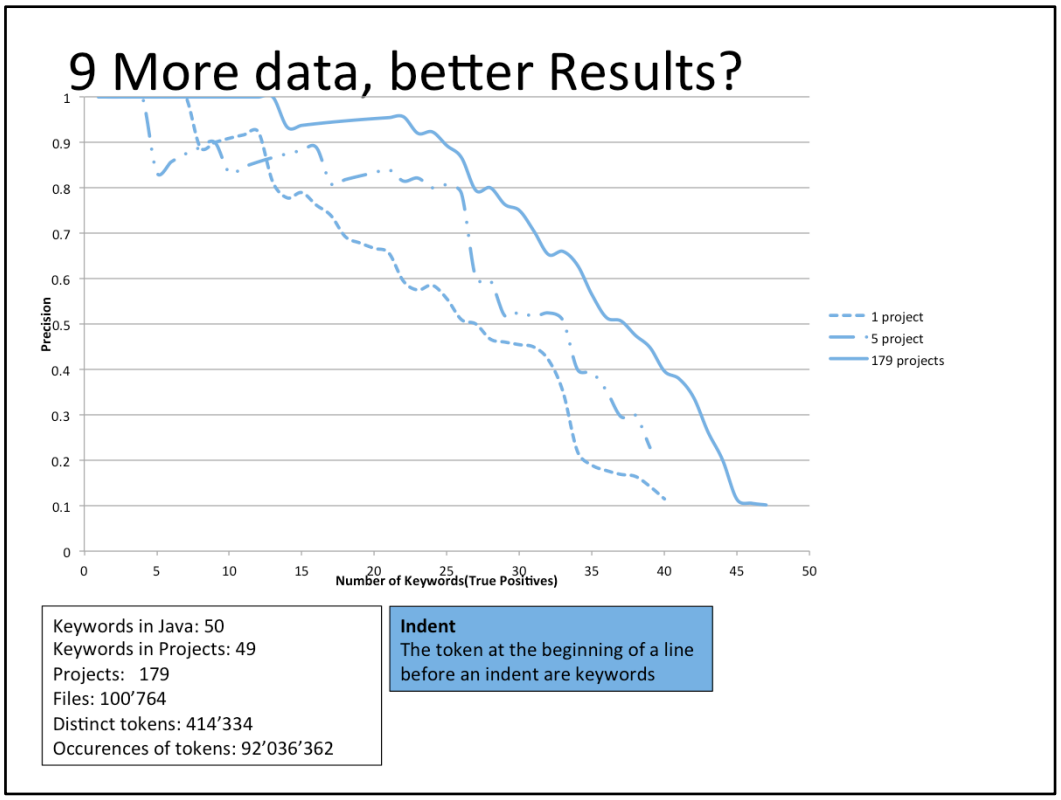
Compared to the filtered ones(click), the precision incresed greatly for the first 20 keywords.
For our newline hypothesis it is 100% for the first 20 keywords.

Still if we want to find all the keywords we arent very successful since the precision drops greatly after 20 tokens.



How can we improve the precision from 20 keywords upwards?
We tried to find out how the amount of data influences these hypothesis to see whether even more data would bring better results.

This graph shows how the coverage hypothesis reacts to the changes in data.
It shows more data seems not to improve the result of the coverage hypothesis.



Next we see how the amount of data influences the indent hypothesis.

The dotted line shows the a single project,
the dotted with lines in between shows 5 random projects and
the normal line shows the result of the 179 projects.

If we look at the lines of the 3 data sets, the precision for keywords between 20 to 40 does raise about 10 to 15 % with more data.
So the graph seems to indicate that more data can yield better results with the right hypothesis,
but also that finding all 49 keywords with high precision is unlikely using these methods.

10 Summary

Architecture

Runtime: exponential to linear

Static design to highly customizable design

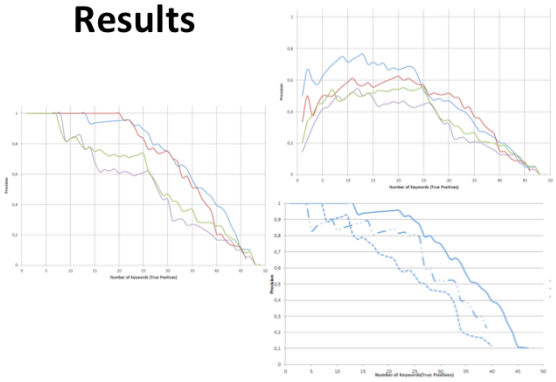
Hypothesis

Indent The token at the beginning of a line before an indent are keywords	**Newline** The token that appear in most files are keywords
Global The keywords appear most commonly over all source code	**Coverage** The token that appear in most files are keywords

Filters

Scanner filter: Removes all tokens marked by the scan mode.
Intersection filter: Counts in how many projects a token occurs and removes the tokens that don't occur in enough projects
Upper case filter: This Filter Removes all tokens containing capital letters.

Results



To sum up the presentation:

Upper left corner: We changed the Architecture by improving the runtime from exponential to linear, the old static design got replaced by a new cutsomizable and extendable one.

New Hypothesis and test can be added and verified easily.

Upper right : The 4 hypothesis we implemented were indent, newline global and coverage of which indent and newline gave the best results.

Bottom left : To improve the results we introduced 3 Filters: Scanner, Intersection and Upper case filter. Which got applied to the results of the 4 hypotesis

Bottom right :Introducing the filters improved the first results greatly but only for the first 20 tokens. The graph of the indent hypothesis suggest that with a lot more data we can get even better results this way. Reaching 50 token seems unlikely with current methods.