Using logistic regression to predict developer responses to Coverity Scan bug reports

Philip J. Guo (pg@cs.stanford.edu)
Advisor: Dawson Engler
Stanford University

Draft: July 6, 2008

1 Objective

This report presents the state of my ongoing work to create statistical models that can be used to make predictions about how developers will respond to bug reports issued by the Coverity tool in the Open Source Scan project. I present models that can predict the following probabilities for a given report based on properties of the report itself and especially on the development history of the file/module where the report indicates a possible bug:

1. Probability that a report is inspected (triaged)
2. Probability that an inspected report is actually resolved (bugfix patch submitted)

My purpose for creating these models is two-fold:

- **Descriptive** - Their parameters indicate possible factors that are correlated with developer responses to bug reports, which can lead to insights about the open source software development process
- **Predictive** - They can be directly used to rank or prioritize future bug reports when presenting to developers

2 Summary of results (for the impatient)

This report ended up being much longer than I had originally intended, so I will summarize the salient findings up-front:

- The type of Coverity checker (e.g., `OVERRUN_STATIC`) that flags a report is by far the largest factor in determining the probability that it will be inspected or resolved (§4.2.1)
- Reports affecting younger files (especially those less than 2 years old) are more likely than those affecting older files to be inspected (§4.2.3)
- Reports affecting smaller files and directories are more likely to be inspected (§4.2.4)
- Reports affecting files with previous reports that were found to be true bugs are more likely to be inspected (§4.2.6)
- Reports affecting files with previous reports that were found to be false positives are less likely to be inspected (§4.2.6)
- Files that have never been patched are sometimes more likely to have their reports inspected (§4.2.5)
- The longer it takes for somebody to inspect a report, the less likely it is that it will be marked as a true bug or eventually resolved (§5.2)
- The number of patches, modified lines, and developers for a file are less important than its age for predicting whether a report affecting that file will be inspected (§4.3)
- The top-level directory name (e.g., `drivers/`, `kernel/`) of the file affected by a report influences its probability of being inspected (§4.2.2)

3 Methodology

I used the method of *logistic regression* (§3.1) to create models that predict the probability of response variables (§3.3) being true based on a variety of explanatory variables (§3.4) in a dataset consisting of Coverity Scan bug reports for Linux (§3.2).

3.1 (Informal) introduction to logistic regression

Informally, a logistic regression model is an equation that relates the conditional probability of an event Y occurring to a weighted combination of values for variables $x_1, x_2, x_3, ..., x_N$. Y is called the **response variable** while the various x's are called **explanatory variables**. The regression equation has the following form:

$$Pr(Y|x_1, x_2, x_3, ..., x_N) \sim \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + ... + \beta_N x_N$$

The exact form of the equation is a bit more complicated (to ensure that the probability lies between 0 and 1), but this simplified form is sufficient to convey the general intuition. The coefficient (β) in front of each explanatory variable determines the strength and direction of correlation between it and the response variable Y. (β_0 is called the intercept term and is not associated with any variable.)

For example, let's say that the following model predicts the probability that a person will enjoy watching the cartoon *Bugs Bunny*:

$$Pr(ENJOY\_BUGS\_CARTOON|AGE, IS\_AMERICAN, IS\_MALE) \sim \\ -0.01 + (-2.3 \times AGE) + (1.3 \times IS\_AMERICAN) + (0.2 \times IS\_MALE)$$

AGE is a continuous variable while $IS\_AMERICAN$ and IS_MALE are (boolean) categorical variables that map to either 0 or 1. Logistic regressions are extremely flexible because the explanatory variables can be continuous (e.g., weight), ordinal (e.g., preference level), or categorical (e.g., blood type).

Here the negative coefficient (-2.3) preceding AGE implies that older people are less likely to enjoy watching *Bugs Bunny*. The positive coefficients in front of $IS\_AMERICAN$ and IS_MALE imply that Americans and boys, respectively, are more likely to enjoy *Bugs Bunny*. However, notice that the IS_MALE coefficient (0.2) is significantly smaller than the $IS\_AMERICAN$ coefficient (1.3), which implies that gender is less strongly correlated with probability of enjoying *Bugs Bunny* than nationality is (i.e., boys are only slightly more likely than girls to enjoy it, but Americans are much more likely than non-Americans to).

3.1.1 Interpretation of logistic regression model coefficients

The first question that many people might ask is how to interpret the numerical values of the coefficients: *What does* $(1.3 \times IS\_AMERICAN)$ *mean? Does it mean that Americans are 1.3 times more likely to like Bugs Bunny than non-Americans?* Unfortunately, the answer isn't that straightforward. Unlike linear regressions (whose coefficients lend themselves to a direct multiplicative interpretation), for logistic regressions, it is more difficult (but still possible) to directly interpret the magnitude of a coefficient. Thus, I will often simply interpret the *sign* as indicating a positive or negative effect and try to indicate magnitude using other means like building contingency tables from the raw data.

3.1.2 Fitting a model to a dataset

To create a logistic regression model, we must choose four components:

1. Response variable – Y (must be binary)
2. Explanatory variables – $x_1, x_2, x_3, ..., x_N$
3. Interaction terms *(optional)* – consist of the product of two or more explanatory variables and is given its own coefficient (e.g., $\beta_{13} x_1 x_3$). When the level of one explanatory variable alters the effects of other explanatory variables on the response variable, there is an *interaction* between them; standard practice suggests adding a product term to account for such effects.
4. Model coefficients – $\beta_0, \beta_1, \beta_2, \beta_3, ..., \beta_N$

The first three must be picked by a human (using a combination of domain-specific intuition, trial-and-error, and sometimes computer assistance), but the last one can (and often should) be determined by a computer statistical package that learns from a *training dataset* (e.g., for my *Bugs Bunny* example, the data could come from a survey of 100 randomly-chosen people). Each entry in the dataset contains values for the explanatory and response variables, and the computer selects coefficients resulting in a model that provides the best fit (minimizes some error metric similar to linear regression) of explanatory variables to the response variable.

3.1.3 Assessing model quality

Once we have created a model, how do we assess how 'good' it is (both in absolute terms and when compared with competing models)? There are several classes of quality metrics:

1. **Reduction in deviance** – The *residual deviance* is a measure of how 'far off' a particular model is from the ideal model that perfectly fits the training dataset (0 deviance is ideal). The *null deviance* is the amount of deviance in a model containing only the intercept term β_0 and is a measure of the worst-possible model for predicting a given response variable (independent of choices of explanatory variables) since it doesn't take any explanatory variables into account.

   The difference between the residual and null deviances indicates how much the explanatory variables helped to improve the model's fit. The larger the reduction in deviance, the better the model fits the training dataset. To determine whether a particular reduction is statistically significant, a *p-value* can be obtained from an **analysis of deviance chi-square test**.

2. **Parsimony** – A model with fewer explanatory variables and interaction terms is usually better, given that it has comparable residual deviance to a more complex model. Simpler models yield more intuitive justifications and are less likely to overfit the training dataset. A commonly-used quality metric called **AIC** (*Akaike's Information Criterion*) augments the residual deviance measure with the number of explanatory variables and assigns a lower (better) score to simpler models.

3. **Classification accuracy** – A good model must accurately classify members of the dataset it was trained on. In my *Bugs Bunny* example, the model provides the probability that each person in the training dataset will like watching *Bugs* based on his/her age, gender, and nationality. We can set a threshold probability, only above which a person is classified as liking *Bugs*. For example, with a threshold of 0.5, we will classify a person with a model-calculated probability of 0.7 as liking *Bugs* (but with a threshold of 0.8, we will classify that same person as *not* liking *Bugs*).

So what threshold is optimal? There is no clear-cut answer; it depends on the relative costs of getting false positives (e.g., mistakenly predicting that a person likes *Bugs* when he/she actually does not) versus false negatives (e.g., mistakenly predicting that a person does not like *Bugs* when he/she actually does).

A common way to visualize the trade-offs of different thresholds is by using an **ROC curve**, a plot of the *true positive rate* (# true positives / total # positives) versus the *false positive rate* (# false positives / total # negatives) for all possible choices of thresholds. For example, a high threshold will yield a low false positive rate (since only samples with very high probabilities will be classified as positive) but also a low true positive rate.

A model with good classification accuracy should have significantly more true positives than false positives at all thresholds. The **area under the ROC curve** quantifies model classification accuracy; the higher the area, the greater the disparity between true and false positives, and the stronger the model in classifying members of the training dataset. An area of 0.5 corresponds to a model that performs no better than random classification, and an area of 1 is ideal (any area above 0.9 is extremely impressive, though).

4. **Prediction accuracy** – One primary function of a model is to make predictions about new, unknown data. Classification accuracy on the training dataset alone cannot validate a model's goodness; using computerized exhaustive search techniques, it can be easy to create complex models that provide extremely high classification accuracy (e.g., with ROC curve areas above 0.9) but *overfit* the training dataset.

   The true test of model quality is how well it does when presented with new data that it was not trained on. Unfortunately, it is often not feasible to collect new data, so a method called **cross-validation** is widely employed to approximate prediction accuracy. In cross-validation, a certain percentage of the dataset (e.g., 20%) is hidden and the rest is used to train the model. Then the model is run to make predictions about the hidden portion of the dataset (which was *not* involved in training), and the error rate is recorded. This process can be repeated numerous times and the error rates can be averaged to provide scores for prediction accuracy.

In summary, a ‘good’ model should have relatively few explanatory variables (parsimony reduces the chances of overfitting), fit the training set data points well (have low deviance), and have strong predictive powers (high ROC curve area and low cross-validation error rates).

3.2 Dataset

For this study, I used 2,090 Coverity Scan bug reports obtained from scanning the Linux source code base between Feb. 2006 and Dec. 2007. All bug reports were released to Linux developers on the Scan website, where they could triage and deal with them individually. Each report contains the following information:

- The file where the bug occurred
- The checker that flagged the bug (e.g., `USE_AFTER_FREE`, `NULL_RETURNS`)
- The date the report was issued
- A timestamped sequence of status changes for the report, as marked by Linux developers who logged into the Scan website. Each report starts as `UNINSPECTED`, and developers change its status to other labels when deemed appropriate:
  - `UNINSPECTED` – no developer has ever investigated this report
  - `PENDING` – someone has investigated this report but cannot determine whether it is a true bug or a false positive
  - `FALSE` – someone has marked this report as a false positive
  - `BUG` – someone has marked this report as a true bug but has not yet taken steps towards resolving it
  - `IGNORE` – someone has marked this report as a true bug but will not fix it for whatever reason
  - `RESOLVED` – someone has marked this report as a true bug and also committed a patch that supposedly fixes it
- The number of days since the report was issued until somebody first inspects it carefully enough to change its status (null if never inspected)
- The number of days since the report was issued until somebody resolves it (null if never resolved)

3.3 Response variables

I will use logistic regression to predict probabilities for the following response variables, which are derived from the final status field of each bug report:

- `INSPECTED` – final status is *not* `UNINSPECTED`, which means that somebody has triaged the report
- `RESOLVED` – final status is `RESOLVED`, which means that somebody has checked in a patch that supposedly fixes the bug described by the report

Recall from §1 that my goal is to create models to predict the following probabilities:

1. $\Pr(\texttt{INSPECTED}\ | x_1, x_2, x_3, ..., x_N)$
2. $\Pr(\texttt{RESOLVED} \mid \texttt{INSPECTED}\ , x_1, x_2, x_3, ..., x_N)$

The explanatory variables $x_1, x_2, x_3, ..., x_N$ are drawn from the set described in §3.4.

3.4 Explanatory variables

I have augmented each report in my dataset with a multitude of variables related to the properties and development history of the file and directory where it flagged a potential bug. I have manually chosen these variables based upon reading related work, discussions with others, and my own hypotheses about what factors might influence the response variables of §3.3. Of course, not all of these variables will end up going into my models (unless I'm either very lucky or totally overfit the dataset); this is just the candidate pool that serves as a starting point for model building. I now present the explanatory variables grouped into categories:

3.4.1 Static file properties

- `file_num_lines` – the number of lines in the file at the time the report occurred
- `file_num_lines.log` – natural logarithm of `file_num_lines` (log transforms are standard practice for making strongly-skewed distributions more symmetric, which helps in making model residuals more symmetric and thus reducing deviances)

- `file_has_gt_500_lines` – `TRUE` iff `file_num_lines > 500` (if `FALSE`, then this is a 'small' file)
- `file_has_gt_2000_lines` – `TRUE` iff `file_num_lines > 2000` (if `TRUE`, then this is a 'large' file)

It is common practice to derive categorical (binary) variables from continuous variables (e.g., `file_has_gt_2000_lines`) to provide greater opportunities for model building. The boundary values are usually chosen through a combination of intuition and manually finding interesting points in the data distribution (e.g., splitting at the 'knee' of a long tail curve).

3.4.2 Static module properties

Most directories in the Linux code base contain source files that together implement one piece of functionality (usually along with a `Makefile`). Thus, I will use directories as an approximation to **modules**:

- `dir_num_files` – the number of source files (`*.[chS]`) in the directory where the report occurred at the time it occurred
- `dir_has_gt_5_files` – `TRUE` iff `dir_num_files > 5` (if `FALSE`, then this is a 'small' module)
- `dir_has_gt_50_files` – `TRUE` iff `dir_num_files > 50` (if `TRUE`, then this is a 'large' module)
- `dir_num_lines` (and `dir_num_lines.log`) – total number of lines in all source files in directory at the time the report occurred (and its natural log)
- `toplevel_dirname` – the top-level directory name of the file where the report occurred (16 categories, e.g., `drivers/`, `fs/`, `kernel/`)

3.4.3 File development history properties

I have only been able to obtain the source control version history of Linux starting in Feb. 2002, which is $\sim$ 5 years before when the Coverity scans occurred. Thus, files created before Feb. 2002 have incomplete (censored) date information. I used a sentinel value of 3,000 days ($\sim$ 8 years) for `file_age` for files created before Feb. 2002. Similarly, I set 3,000 for `file_days_since_last_patch` and `file_days_since_last_monofile_patch` if there are no patches to a file after Feb. 2002.

- `file_age` – the number of the days the file has existed at the time the report occurred
- `file_age_in_years` – same as `file_age`, except in years, rounding down to the nearest integer and converted into a categorical variable (with categories `0`, `1`, `2`, etc.)
- `file_days_since_last_patch` – at the time the report occurred, how many days has it been since the most recent patch was committed for this file?
- `file_days_since_last_monofile_patch` – at the time the report occurred, how many days has it been since the most recent patch that *only affected* this file ('monofile patch') was committed? I hypothesize that patches affecting only one file are usually more significant for that file than patches affecting multiple files.
- `file_num_patches` (and `file_num_patches.log`) – number of patches to this file between Feb. 2002 and the time the report occurred (and natural log, adding 1 to prevent `NaN`'s resulting from `0` values)
- `file_has_gt_20_patches` – `TRUE` iff `file_num_patches > 20`
- `file_has_gt_100_patches` – `TRUE` iff `file_num_patches > 100`
- `file_num_patches_1_month_prior` – # patches to this file in the 1 month prior to the report date, an indication of its level of recent development activity.
- `file_num_patches_6_months_prior` – # patches to this file in the 6 months prior to the report date
- `file_num_monofile_patches` – # patches that *only affected* this file ('monofile patch') between Feb. 2002 and the report date
- `file_num_monofile_patches_1_month_prior` – # monofile patches in the 1 month prior to the report date

- `file_num_monofile_patches_6_months_prior` – # monofile patches in the 6 months prior to the report date
- `file_num_mod_lines` (and `file_num_mod_lines.log`) – total number of lines modified (inserted + deleted) in this file in all patches from Feb. 2002 to the report date (and natural log, adding 1 to prevent `NaN` resulting from `0` values)
- `file_num_mod_lines_1_month_prior` – # modified lines in the 1 month prior to the report date
- `file_num_mod_lines_6_months_prior` – # modified lines in the 6 months prior to the report date

3.4.4 Developer properties

- `file_num_authors` (and `file_num_authors.log`) – the number of unique developers who have written patches for this file from Feb. 2002 to the report date (and natural log, adding 1 to prevent `NaN` resulting from `0` values)
- `file_has_gt_1_authors` – `TRUE` iff `file_num_authors > 1` (this is a key boundary since many files only have 1 developer)
- `file_has_gt_5_authors` – `TRUE` iff `file_num_authors > 5`
- `file_has_gt_15_authors` – `TRUE` iff `file_num_authors > 15`
- `file_num_authors_1_month_prior` – # unique developers who have written patches for this file in the 1 month prior to report date
- `file_num_authors_6_months_prior` – # unique developers who have written patches for this file in the 6 months prior to report date

3.4.5 Code churn metrics

- `file_percentage_churn` =

  `file_num_mod_lines / file_num_lines`
- `file_has_gt_100_percentage_churn` =

  `TRUE` iff `file_percentage_churn > 1`

- `file_percentage_churn_1_month_prior` =
  `file_num_mod_lines_1_month_prior / file_num_lines`
- `file_percentage_churn_6_months_prior` =
  `file_num_mod_lines_6_month_prior / file_num_lines`

3.4.6 Module development history properties

I have derived the following variables for modules (directories) by summing up the respective values for all source files (`*.[chS]`) in each directory (the variable names should be self-explanatory). These might sometimes provide over-approximations (e.g., patches are often over-counted because a single patch might affect multiple files in the same directory), but I am not too concerned with this inaccuracy for now.

- `dir_num_mod_lines`
- `dir_num_mod_lines_1_month_prior`
- `dir_num_mod_lines_6_months_prior`
- `dir_num_patches`
- `dir_num_patches_1_month_prior`
- `dir_num_patches_6_months_prior`
- `dir_num_monofile_patches`
- `dir_num_monofile_patches_1_month_prior`
- `dir_num_monofile_patches_6_months_prior`

3.4.7 Coverity report properties

The following are properties about the Coverity reports themselves and developer responses to them:

- `checker` – the type of checker that flagged the bug (12 categories, e.g., `DEADCODE`, `OVERRUN_DYNAMIC`, `USE_AFTER_FREE`)

- `days_before_inspection` – the number of days between the report date and the *first time* somebody inspects and marks it with some status other than `UNINSPECTED` (null if never inspected)

- `num_prev_inspected_reports_in_file` – at the time that *this report* was issued, how many *other reports* in the **same file** had already been inspected?

- `num_prev_inspected_reports_in_dir` – at the time that *this report* was issued, how many *other reports* in the **same directory** had already been inspected?

- `num_prev_inspected_reports_in_file_FACTOR` =

  `num_prev_inspected_reports_in_file` converted into a categorical variable (with categories `0`, `1`, `2`, etc.) to try to find non-linear relations

- `num_prev_inspected_reports_in_dir_FACTOR` =

  `num_prev_inspected_reports_in_dir` converted into a categorical variable

- `file_has_prev_inspected_reports` =

  `TRUE` iff `num_prev_inspected_reports_in_file > 0`

- `dir_has_prev_inspected_reports` =

  `TRUE` iff `num_prev_inspected_reports_in_dir > 0`

I have also added two sets of variables that are analogous to those just presented in this section. These variables have names like:

- `file_has_prev_inspected_BUGGY_reports`

- `file_has_prev_inspected_FALSE_reports`

The `*_BUGGY_reports_*` variables only count previous reports that have been inspected and *found to be true bugs*, while the `*_FALSE_reports_*` variables only count previous reports that have been inspected and *found to be false positives.*

4 Predicting whether a report will be inspected

This section describes my attempts to create logistic regression models to predict whether a given Coverity Scan report will be inspected by developers. The process of creating these models compelled me to analyze the effects of certain explanatory variables in greater detail, so I will also present the findings of these investigations.

4.1 Single regression models

The simplest type of logistic regression model involves only one explanatory variable; as a starting point, I first created separate models using each explanatory variable in §3.4 to predict `INSPECTED`. Table 1 shows a subset of those variables that provided reasonably good fits (with low *p-values* in an analysis of deviance chi-square test). Each row shows one explanatory variable, its regression equation coefficient, and two metrics of model quality from §3.1.3: chi-square *p-value* indicating the significance of reduction in deviance (lower is better) and ROC area indicating classification accuracy (higher is better). Note that non-binary categorical variables (e.g., `checker`, `file_age_in_years`, and `toplevel_dirname`) do not have one coefficient but rather have as many coefficients as there are categories, so they are not listed in Table 1 for brevity.

Although all of these single regression models have low *p-values*, their ROC areas aren't at all impressive. Recall that a model that randomly classifies the response variable will have an average ROC area of 0.5; most of these models don't improve upon that baseline by much (0.65 for `checker` is respectable, though). In other words, most explanatory variables have a statistically significant correlation with inspection probability, but their correlations are fairly weak. A common warning about employing statistical techniques is that, given a large enough sample size (there are 2,090 reports in my dataset), even tiny effects will likely be deemed statistically significant; statistical significance alone does not mean that an effect has *practical significance*.

Full results are shown in Tables 13 and 14 in the Appendix. Note that just because a particular explanatory variable *alone* does not result in a strong model does not mean that it will not be useful when combined with other variables. As a commonly-accepted heuristic, any explanatory variable whose *p-value* in single regression is less than 0.3 could be a viable candidate

Explanatory variable	coefficient	p-value	ROC area
`checker`	N/A	3.2e-32	0.65
`file_num_authors.log`	−0.35	1.8e-12	0.59
`file_has_gt_1_authors`	−1.06	1.9e-11	0.55
`file_num_patches.log`	−0.24	2.6e-11	0.58
`file_age_in_years`	N/A	2.1e-10	0.59
`file_age`	−0.00024	4.4e-10	0.59
`file_has_gt_15_authors`	−0.59	2e-09	0.56
`file_num_mod_lines.log`	−0.13	2.2e-09	0.56
`file_has_gt_20_patches`	−0.53	7e-09	0.56
`file_num_authors`	−0.02	9.1e-08	0.59
`file_has_gt_5_authors`	−0.51	1.6e-07	0.55
`file_days_since_last_monofile_patch`	0.00022	3.1e-07	0.53
`file_days_since_last_patch`	0.00043	7.7e-07	0.54
`file_has_gt_2000_lines`	−0.45	1.7e-06	0.55
`toplevel_dirname`	N/A	2.7e-06	0.57
`file_num_lines.log`	−0.18	9.8e-05	0.56
`dir_num_patches_6_months_prior`	−0.00059	0.00017	0.54
`dir_num_patches`	−8.6e-05	0.0002	0.57
`dir_num_patches_1_month_prior`	−0.0021	0.00042	0.54
`file_num_authors_1_month_prior`	−0.11	0.00045	0.54
`file_has_gt_100_percentage_churn`	−0.31	0.00057	0.54
`dir_num_mod_lines`	−1.58e-06	0.00097	0.56
`file_num_authors_6_months_prior`	−0.039	0.0011	0.54
`file_num_patches_6_months_prior`	−0.0094	0.005	0.54
`dir_num_lines.log`	−0.098	0.006	0.53

Table 1: Selected explanatory variables that provided good fits for a single logistic regression model to predict `INSPECTED`.

`checker`	Percent (and number) of reports			
	inspected		uninspected	
`DEADCODE`	**82%**	(205)	18%	(45)
`FORWARD_NULL`	50%	(250)	50%	(250)
`NEGATIVE_RETURNS`	38%	(16)	62%	(26)
`NULL_RETURNS`	51%	(76)	49%	(72)
`OVERRUN_DYNAMIC`	**100%**	(6)	0%	(0)
`OVERRUN_STATIC`	**78%**	(222)	22%	(62)
`RESOURCE_LEAK`	53%	(161)	47%	(141)
`REVERSE_INULL`	57%	(144)	43%	(108)
`REVERSE_NEGATIVE`	**69%**	(9)	31%	(4)
`SIZECHECK`	20%	(1)	80%	(4)
`UNINIT`	**84%**	(53)	16%	(10)
`USE_AFTER_FREE`	48%	(109)	52%	(116)
Total	60%	(1252)	40%	(838)

Table 2: Percent and number of inspected reports for different checker types, which are significantly different across checkers at $p << 0.01$ according to a chi-square test. Entries above the overall percentage of 60% are in bold.

for including in a multiple regression model (a topic further explored in §4.5).

4.2 Impactful explanatory variables

The following types of variables lead to well-fitting single regression models according to Table 1 and can be used as a starting point for generating multiple regression models: checker type, top-level directory name, file age, and file/module size.

4.2.1 Checker type

Table 1 shows that `checker` results in the strongest single regression model by far, with the lowest *p-value* and highest ROC area of 0.65.

To explore the effects of checker type in more detail, I have created a *contingency table* to compare inspection rates across checkers. The resulting Table 2 shows the percentage and number of reports that were inspected and uninspected for each of the dozen checker types.

The inspection percentages differ significantly across checkers, as confirmed by a chi-square test. If developers inspected bug reports without regard to the checker type, then the inspection percentages for each checker would likely be nearly identical and fail the chi-square test.

Reports from certain checker types are more likely than others to be inspected, perhaps because developers deem them more critical to fix or simply have an easier time distinguishing between true bugs and false positives (recall that to 'inspect' a bug report means more than to merely glance at it; one must be confident enough to triage it as a true bug, false positive, or, at worse, mark it as unsure to defer to another developer). The `DEADCODE`, `OVERRUN_DYNAMIC`, `OVERRUN_STATIC`, `REVERSE_NEGATIVE`, and `UNINIT` checkers have above-average inspection rates.

4.2.2 Top-level directory

Table 1 shows that the top-level directory (`toplevel_dirname`) also provides a fairly strong single regression model. The corresponding contingency table (Table 3) shows inspection rates differing across top-level directories.

Note that most of these directories contain relatively few files and thus few reports, so their percentage deviations from the average aren't too significant. For the four directories with large numbers of reports, `drivers`, `fs`, and `sound` have inspection rates around the global average of 60%, but `net` has a much lower rate of 47%.

The general insight here is that developers prioritize different parts of the codebase unequally; some parts might be deemed more security-critical or are simply under more active development. Of course, the exact details here are only relevant to Linux, but the idea of using top-level directory names as explanatory variables for model-building can be useful for studying other software projects as well.

toplevel_dirname	Percent (and number) of reports			
	inspected		uninspected	
arch/	77%	(23)	23%	(7)
block/	80%	(4)	20%	(1)
crypto/	67%	(2)	33%	(1)
drivers/	63%	(748)	37%	(432)
fs/	58%	(180)	42%	(128)
include/	71%	(5)	29%	(2)
init/	100%	(1)	0%	(0)
ipc/	78%	(7)	22%	(2)
kernel/	50%	(18)	50%	(18)
lib/	25%	(2)	75%	(6)
mm/	62%	(10)	38%	(6)
net/	47%	(159)	53%	(181)
scripts/	100%	(8)	0%	(0)
security/	83%	(5)	17%	(1)
sound/	60%	(79)	40%	(53)
usr/	100%	(1)	0%	(0)
Total	60%	(1252)	40%	(838)

Table 3: Percent and number of inspected reports according to top-level directory, which are significantly different across directories at $p << 0.01$ according to a chi-square test.

# and % of reports	`file_age_in_years`							Total
	0–1	**1–2**	2–3	3–4	4–5	5–6	> 6	
inspected	**71%** (391)	**70%** (115)	53% (85)	57% (191)	52% (107)	52% (17)	54% (346)	60% (1252)
uninspected	29% (159)	30% (50)	47% (76)	43% (142)	48% (98)	48% (16)	46% (297)	40% (838)

Table 4: Inspection rates for reports in files of various ages, which are significantly different across years at $p << 0.01$ according to a chi-square test. Entries above the overall percentage of 60% are in bold.

4.2.3 File age

One intuition about software development is that younger files (files created more recently) are under more active development than older files, so developers might be more responsive to bug reports affecting younger files.

Table 1 shows that the age of the file where a report occurs can be a good predictor for whether it will be inspected (via the continuous variable `file_age` and its derived categorical variable `file_age_in_years`). The negative coefficient (-0.00024) for `file_age` suggests that developers are more likely to inspect reports affecting younger files. Note that its small magnitude might be due to the fact that `file_age` is measured in units of *days* (rather than larger units like months or years).

The contingency table for `file_age_in_years` (Table 4) clearly shows that files less than 2 years old have a much higher inspection rate than their older counterparts.

In addition to simply dividing up by years using `file_age_in_years`, I thought it would be informative to also divide up files into 'young' and 'old' based on an arbitrary threshold of how many days each has been alive (using `file_age`). Let's say that the threshold is 100 days. Then I can create the following 2x2 contingency table where each entry contains the number of reports satisfying its respective criteria:

	`file_age < 100` (young)	`file_age >= 100` (old)
# inspected	a	b
# uninspected	c	d

The inspection rate is $a/(a+c)$ for young files and $b/(b+d)$ for old files.

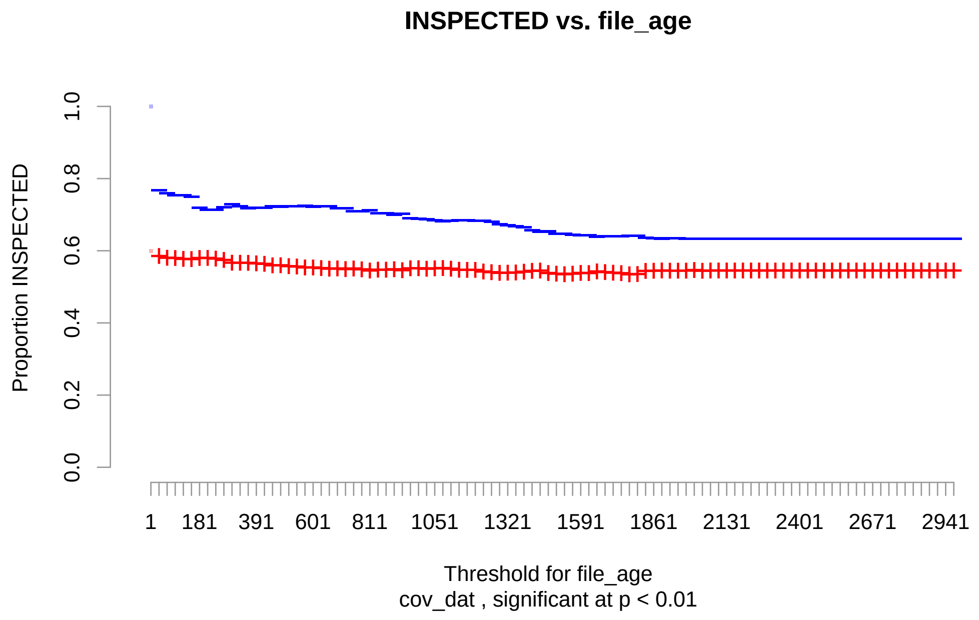


Figure 1: Inspection rates for various thresholds of `file_age`. For significant differences with chi-square $p < 0.01$, files older than threshold marked in red $+$, and files younger than threshold marked in blue $-$.

A chi-square test can be used to determine whether there is a statistically significant difference in inspection rates between young and old files for this given choice of threshold.

But which threshold should I choose? Well, why stop at just one threshold? Figure 1 shows the inspection rates for old vs. young files plotted for a wide range of thresholds of `file_age`. The high-order bit of this figure is that the red $+$ marks (inspection rates for old files) are consistently *below* the blue $-$ marks (inspection rates for young files), which means that for (almost) all possible choices of thresholds, younger files are *more likely to be inspected* than older files. These results corroborate Table 4, but that shouldn't be surprising since `file_age_in_years` is derived directly from `file_age`.

It might now seem like there is strong evidence to suggest that developers are more likely to inspect reports in younger files than in older files, but there is a bit more to this story. It turns out that my dataset actually contains two different kinds of Coverity Scan reports:

INSPECTED $\sim$ file_age			
	coefficient	p-value	ROC area
Initial scan	-2.52e-05	0.69	0.52
Subsequent scan	-0.00043	1.2e-16	0.65
All	-0.00024	4.4e-10	0.59
INSPECTED $\sim$ file_age_in_years			
	coefficient	p-value	ROC area
Initial scan	N/A	0.055	0.56
Subsequent scan	N/A	2.8e-14	0.64
All	N/A	2.1e-10	0.59

Table 5: Single logistic regression models for file age versus `INSPECTED` for initial and subsequent scan reports.

- **Initial scan reports** – 970 reports released together to developers on 2006-02-24, resulting from scanning the entire Linux codebase for the first time in the Coverity Scan project
- **Subsequent scan reports** – 1120 reports released in small batches between 2006-02-24 and 2007-12-01, which only include *new bugs* found in existing files and bugs found in *newly added* files

Figure 2 visualizes the distribution of reports across time. Almost half of all reports were released as a batch on 2006-02-24 at the start of the Coverity Scan project, and the rest of the reports were released periodically to the Scan website every few weeks or so as the entire Linux codebase was re-scanned.

To produce Figure 3, I split up my dataset into two separate groups and calculated inspection rates vs. `file_age` thresholds separately for initial and subsequent scan reports.

For initial scan reports, there was *no statistically significant difference* in inspection rates, as indicated by all data points being muted-colored dots. Even though the rates weren't identical, they weren't 'different enough' according to a chi-square test with the significance threshold set to $p = 0.01$. This fact suggests that when the initial batch of reports were released, developers did not really bias their inspections based on how old the files were at that time.

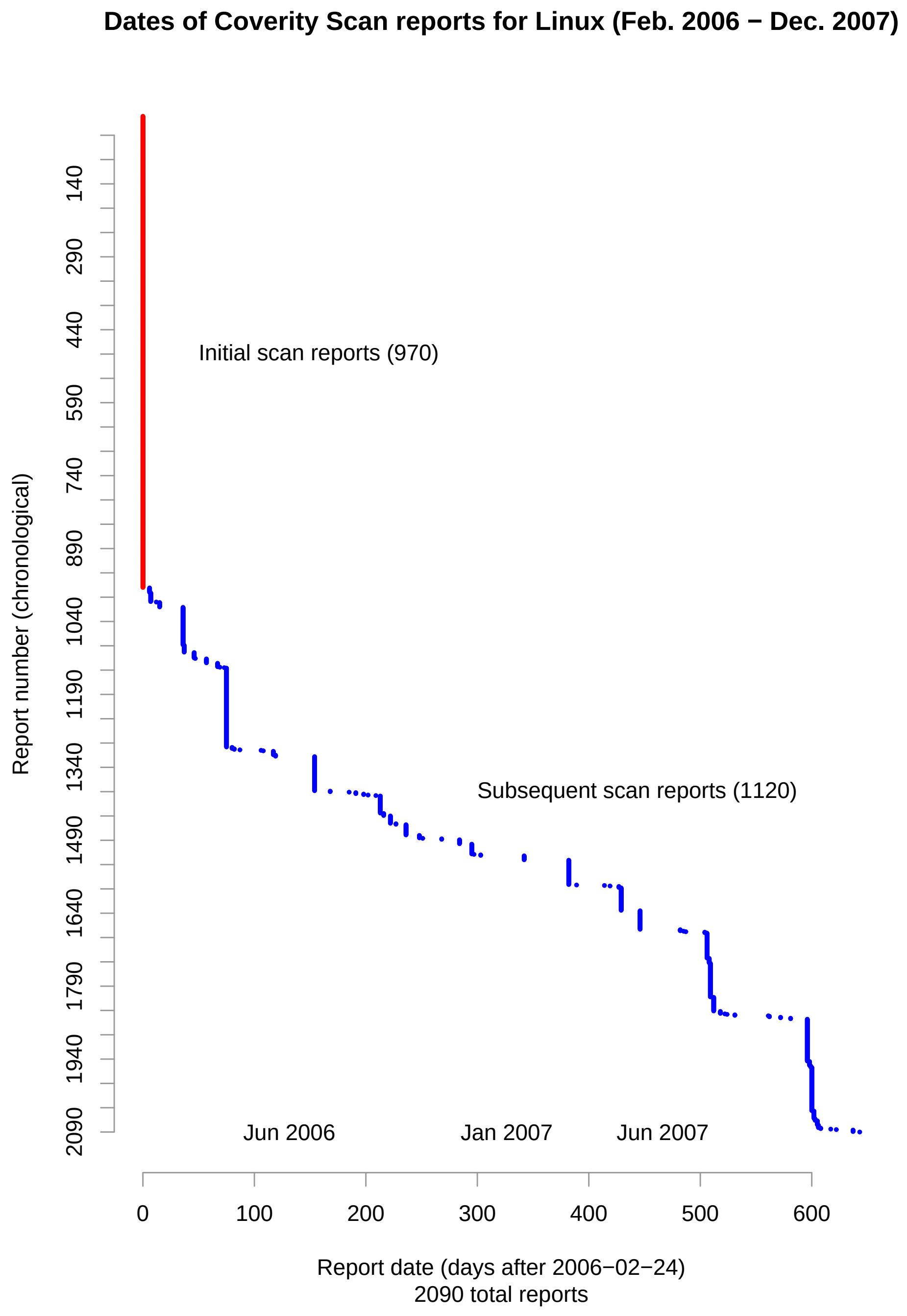
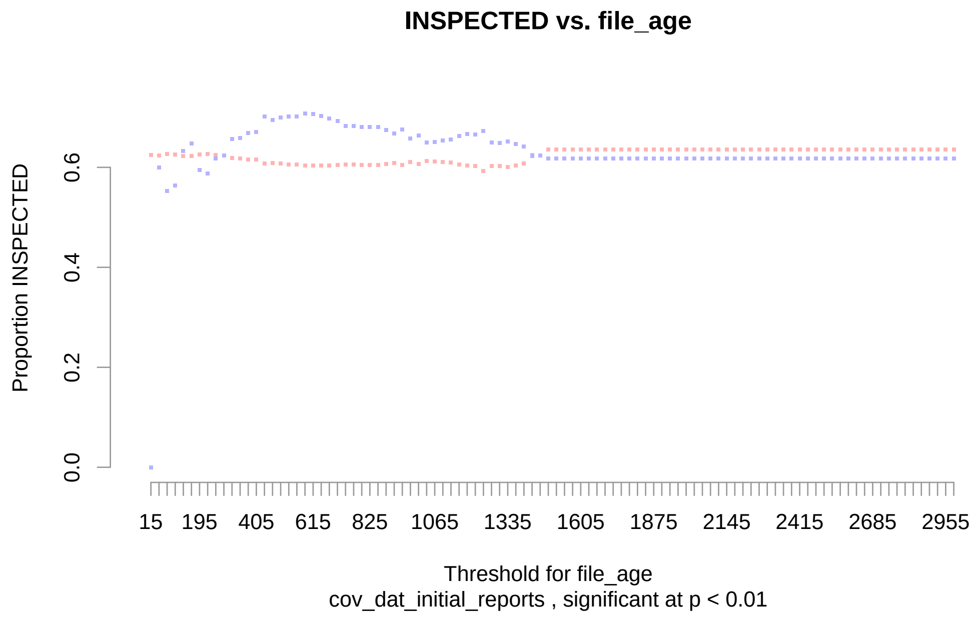


Figure 2: Distribution of all Coverity Scan reports across time.



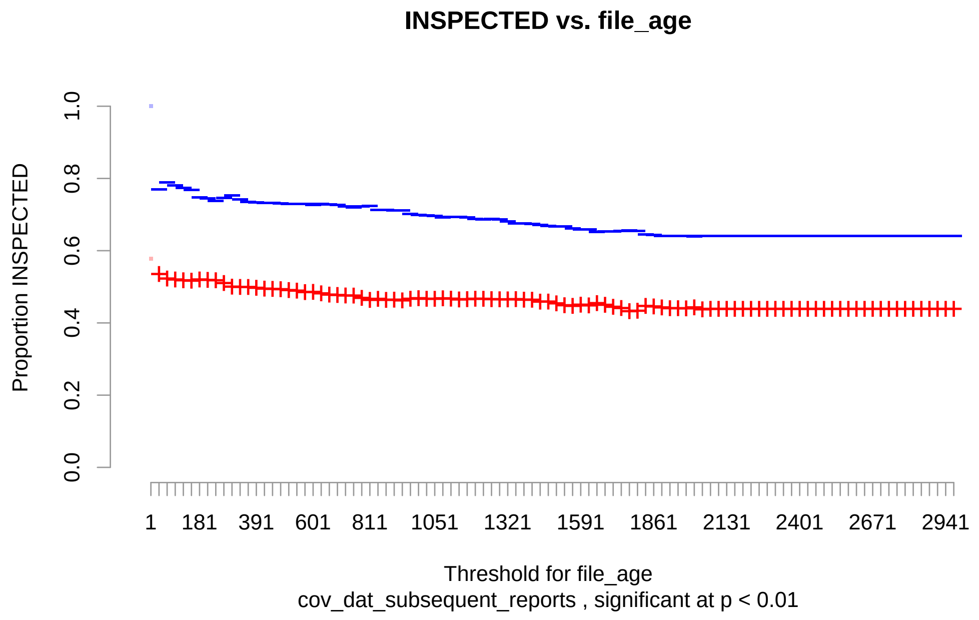


Figure 3: Inspection rates for various thresholds of `file_age` for *initial scan* reports (top) and *subsequent scan* reports (bottom). For significant differences with chi-square $p < 0.01$, files older than threshold marked in red $+$, and files younger than threshold marked in blue $-$. Muted-colored dots are for chi-square $p \geq 0.01$.

In contrast, for subsequent scan reports, there is a striking difference in inspection rates between old and young files. This fact suggests that when batches of new reports arrived incrementally, developers were *more likely to inspect reports in younger files.*

The contrast between initial and subsequent reports suggests that it might be useful to create separate models for each dataset. The models I generated for Table 5 corroborates Figure 3: Note the far smaller *p-values* and larger ROC areas in models for the subsequent scan reports, which demonstrate that file age variables generate much better models for subsequent scan reports than for initial scan reports.

4.2.4 File/module size

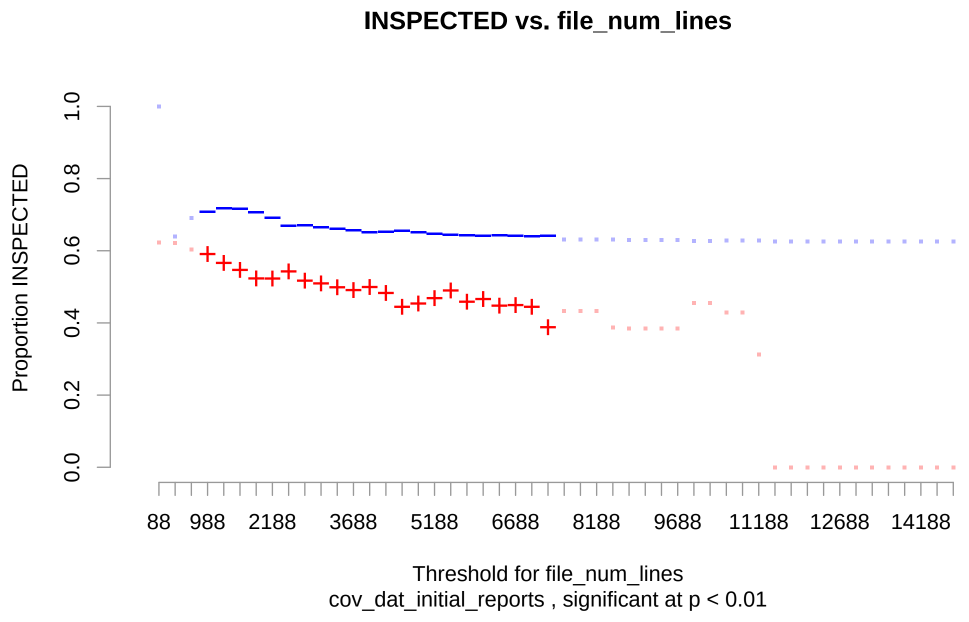
Does the size of the file/module where a report occurs affect its probability of inspection? Size (e.g., # lines) is usually correlated with code complexity, so it might be easier to triage bug reports in smaller (simpler) files/modules.

Once again dichotomizing the data using thresholds, I generated Figure 4, which shows that for initial scan reports, *smaller files are more likely to be inspected* than larger files, but for subsequent scan reports, there isn't a statistically significant difference in inspection rates (most data points are muted-colored dots).

These results suggest that when developers were initially presented with a large batch of 970 reports on 2006-02-24, they favored inspecting reports in smaller files, perhaps because they were easier to triage, but when presented with new reports in the subsequent months, file size was overshadowed by other factors (most notably file age).

For instance, out of all initial scan reports, 70% of reports in files with fewer than 2000 lines were inspected while only 50% of reports in files greater than 2000 lines were inspected.

The models I created for Table 6 corroborate that variables related to file/module size create far better models for initial scan reports than for subsequent scan reports (with smaller *p-values* and larger ROC area). File size seems to be a better predictor for inspection rates than directory size, perhaps because it more directly measures the complexity of the portion of code where the report affects. `file_num_lines` fares the best in terms of ROC area, while the other two variables have much weaker predictive powers, even though they still have strong (small) *p-values*.



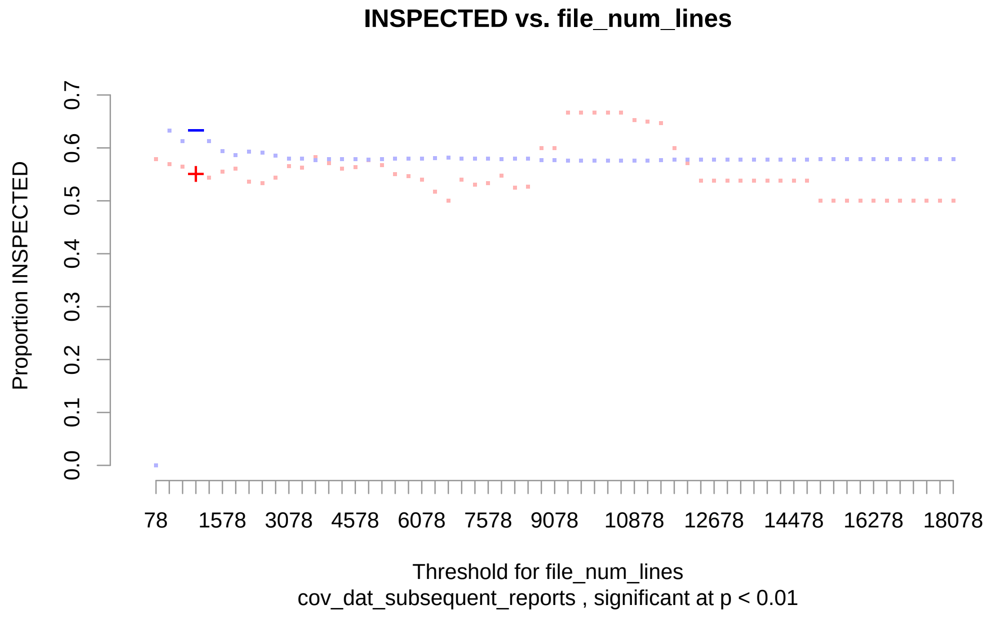


Figure 4: Inspection rates for various thresholds of `file_num_lines` for *initial scan* reports (top) and *subsequent scan* reports (bottom). For significant differences with chi-square $p < 0.01$, files larger than threshold marked in red +, and files smaller than threshold marked in blue −. Muted-colored dots are for chi-square $p \geq 0.01$.

`INSPECTED` ~ `file_num_lines`			
	coefficient	p-value	ROC area
Initial	−0.00015	4.2e-07	0.6
Subsequent	−2e-05	0.39	0.53
All	−6.9e-05	0.00017	0.56

`INSPECTED` ~ `dir_num_lines`			
	coefficient	p-value	ROC area
Initial	−2.27e-06	0.0034	0.54
Subsequent	−8.25e-07	0.31	0.54
All	−1.47e-06	0.0085	0.53

`INSPECTED` ~ `dir_num_files`			
	coefficient	p-value	ROC area
Initial	−0.0015	0.032	0.54
Subsequent	−0.00028	0.67	0.52
All	−0.00078	0.10	0.52

Table 6: Single logistic regression models for file/module size versus `INSPECTED` for initial and subsequent scan reports.

initial scan reports		
	0 patches	>0 patches
inspected	67% (4)	62% (601)
uninspected	33% (2)	38% (363)

subsequent scan reports		
	0 patches	>0 patches
inspected	85% (69)	56% (578)
uninspected	15% (12)	44% (461)

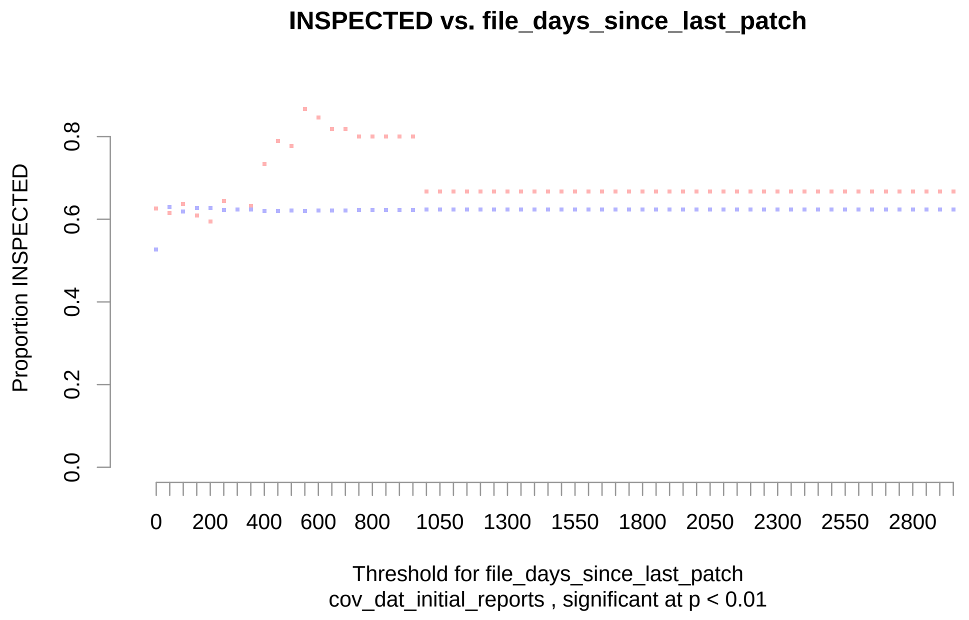
Table 7: Inspection rates for reports in files with zero versus non-zero patches.

4.2.5 Number of days since most recent patch

Figure 5 shows that the number of days since the most recent patch to a file has no significant bearing on the inspection rate for initial scan reports but is positively correlated with the inspection rate for subsequent scan reports. This finding is a bit strange because it implies that, for subsequent scan reports at least, reports for files that were patched *less recently* were more likely to be inspected.

Upon some manual investigation, I realized that this effect might be an artifact of me using a sentinel value of 3,000 days for `file_days_since_last_patch` for files that were *never patched* (i.e., files that were never patched always seem to be above all thresholds). After splitting up the data into files with 0 patches and >0 patches (Table 7), it becomes apparent that subsequent scan reports in files with 0 patches have a much higher inspection rate — 85% vs. 56% — passing a chi-square test with flying colors. In contrast, because there are extremely few files with 0 patches amongst initial scan reports, even though their proportions are slightly different, it fails the chi-square test ($p = 0.84$).

Interpreted in this new light, the results in Figure 5 simply indicate that reports in files that have *never been patched* (with a really, really 'large' # days since last patch) have a greater probability of being inspected. Perhaps developers are more attuned to bug reports in 'green' files that have been recently added to the repository and never patched, but I have not investigated in more detail.



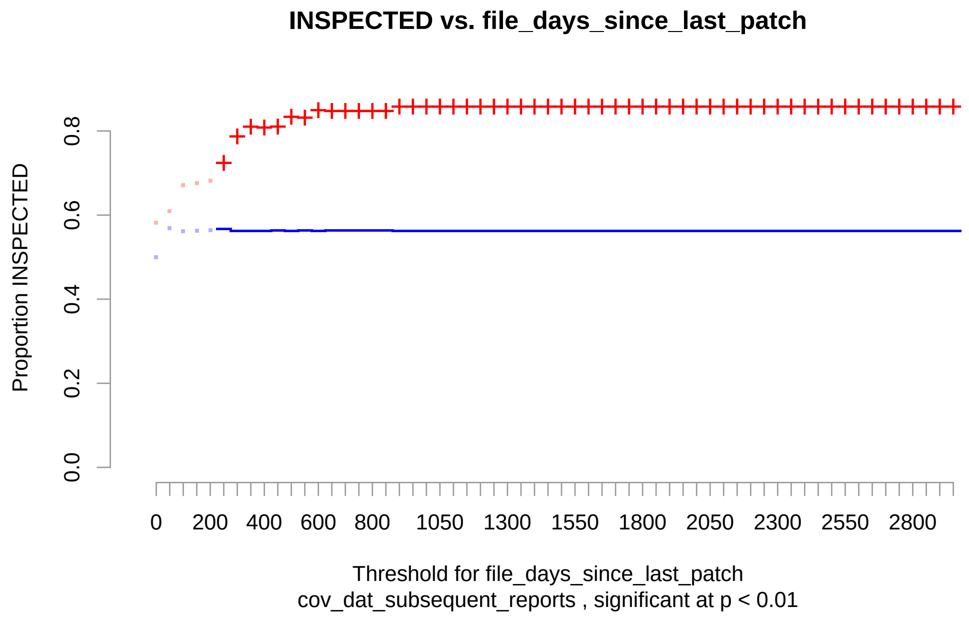


Figure 5: Inspection rates for various thresholds of `file_days_since_last_patch` for *initial scan* reports (top) and *subsequent scan* reports (bottom). Files with # days significantly larger than threshold marked in red +, smaller than threshold marked in blue −.

Explanatory variable	coefficient	p-value	ROC area
`file_has_prev_inspected_reports`	−0.21	0.12	0.52
`file_has_prev_inspected_BUGGY_reports`	0.20	0.23	0.51
`file_has_prev_inspected_FALSE_reports`	−0.098	0.57	0.51
`dir_has_prev_inspected_reports`	−0.50	0.00023	0.55
`dir_has_prev_inspected_BUGGY_reports`	−0.25	0.039	0.53
`dir_has_prev_inspected_FALSE_reports`	−0.45	0.00025	0.55
`num_prev_inspected_reports_in_dir`	−0.016	0.028	0.55
`num_prev_inspected_BUGGY_reports_in_dir`	−0.008	0.59	0.52
`num_prev_inspected_FALSE_reports_in_dir`	−0.061	0.00036	0.57

Table 8: Single logistic regression models for number of previous inspected reports versus `INSPECTED`, only for subsequent scan reports.

4.2.6 Previous inspected reports (subsequent scan only)

For subsequent scan reports, it would be reasonable to assume that previous inspections are indicative of future inspections. If at the time that some report arrives, there are previous reports in the same file/module that have *already been inspected*, it might influence the probability of that report being inspected. For this particular dataset, it turns out that this effect is present but fairly weak.

Table 8 shows single logistic regression models built from binary variables indicating whether the file/module that a report affects has any previous inspected reports. The negative coefficients for `file_has_prev_inspected_reports` and `dir_has_prev_inspected_reports` seem counter-intuitive. Is it really true that when a file/module has previous inspected reports, the probability of inspection for future reports actually *decreases*? My intuition suggests that having previous inspected reports should actually *increase* the probability of inspection.

One hypothesis is that many reports are inspected and then marked as false positives, so developers would be less likely to pay attention to future reports in those same files/modules. To test this hypothesis, I looked at whether a file/module had previous inspected reports that were marked as true bugs or false positives using the `*_BUGGY_*` and `*_FALSE_*` variables.

The regression equation coefficient turns out to be *positive* (0.20) for `file_has_prev_inspected_BUGGY_reports` and negative for `file_has_prev_` `inspected_FALSE_reports` (-0.098), which means that files with previous reports that were inspected and marked as true bugs were *more likely* to have their future reports inspected, while files with previous reports that were marked as false positives were *less likely* to have their future reports inspected, corroborating my hunch. However, the *p-values* and ROC areas for both are fairly weak, so this effect isn't at all substantial.

On the module (directory) level, both coefficients are negative, but the one for inspected buggy reports (-0.25) is less negative than the one for inspected false positive reports (-0.45).

Finally, the *number* of previous inspected reports actually matters: The *p-value* for `num_prev_inspected_FALSE_reports_in_dir` is 0.00036 (with the highest ROC area of the bunch, 0.57), while the *p-value* for `num_prev_inspected_BUGGY_reports_in_dir` is 0.59, which is nowhere near statistically significant. This means that *the more false positives found in previous reports in the same module, the less likely that future reports will be inspected*, while there is no significantly noticeable effect for reports found to be true bugs.

4.3 Redundant explanatory variables

There are many more variables in Table 1 that I have not yet examined in detail, most notably those dealing with numbers of patches, modified lines, and developers. I surmised that these variables might be dependent on `file_age`, so it would be redundant to also include them in a model. For instance, younger files are likely to have fewer patches, modified lines, and developers. One way to test for redundancy is to create a multiple logistic regression model and to separately assess the significance of the effect of each explanatory variable.

4.3.1 Number of patches

`file_age` and `file_num_patches` are decently well-correlated, with a (linear) *Pearson's r* of 0.4 and (nonlinear) *Spearman's rho* of 0.64 (1 is perfect positive correlation), which means that there is a good chance that it is only necessary to include one of them in a model rather than both.

Table 9 shows that when `file_age` and `file_num_patches` are both used together to create a multiple regression model, `file_age` greatly reduces the deviance from 2814.8 to 2775.9 (earning it a tiny *p-value* of 4.4e-10), but

INSPECTED ∼ file_age + file_num_patches			
Term	coefficient	deviance	p-value
NULL		2814.8	
file_age	−0.00024	2775.9	4.4e-10
file_num_patches	−0.00028	2775.8	0.71

Table 9: Multiple regression model showing that `file_num_patches` is unnecessary once `file_age` is already in the model.

INSPECTED ∼ file_age + file_num_mod_lines			
Term	coefficient	deviance	p-value
NULL		2814.8	
file_age	−0.00024	2775.9	4.4e-10
file_num_mod_lines	4.6e-06	2775.7	0.67

Table 10: Multiple regression model showing that `file_num_mod_lines` is unnecessary once `file_age` is already in the model.

`file_num_patches` does almost nothing to the deviance (earning it a huge *p-value* of 0.71). In other words, once `file_age` is in the model, adding `file_num_patches` doesn't provide any more substantial gains, so it is redundant.

4.3.2 Number of modified lines

`file_age` and `file_num_mod_lines` are also decently well-correlated, with a (linear) *Pearson's r* of 0.3 and (nonlinear) *Spearman's rho* of 0.55, although the correlation is not as strong as with `file_num_patches`.

Still, Table 10 shows that using `file_num_mod_lines` doesn't fare much better than using `file_num_patches` once `file_age` is in the model, so it too is redundant.

4.3.3 Number of developers

`file_age` and `file_num_authors` are *strongly* correlated, with a (linear) *Pearson's r* of 0.57 and (nonlinear) *Spearman's rho* of 0.72.

`INSPECTED` $\sim$ `file_age + file_num_authors`			
Term	coefficient	deviance	p-value
`NULL`		2814.8	
`file_age`	-0.00024	2775.9	4.4e-10
`file_num_authors`	-0.01	2771.2	0.03

Table 11: Multiple regression model showing that `file_num_authors` might actually be able to work together with `file_age` to create a stronger model.

`INSPECTED` $\sim$ `file_age + file_num_authors`			
Only considering *initial scan* reports			
Term	coefficient	deviance	p-value
`NULL`		1284.7	
`file_age`	9.7e-7	1284.5	0.69
`file_num_authors`	-0.006	1284.0	0.45
Only considering *subsequent scan* reports			
Term	coefficient	deviance	p-value
`NULL`		1525.5	
`file_age`	-0.0004	1457.0	1.2e-16
`file_num_authors`	-0.003	1456.7	0.61

Table 12: Multiple regression models for initial scan (top) and subsequent scan (bottom) reports showing that, in fact, `file_num_authors` is redundant.

Strangely, though, Table 11 actually shows that `file_num_authors` seems to *not* be redundant with `file_age` because it can further decrease the deviance of the model by a statistically significant amount, earning it a respectable *p-value* of 0.03.

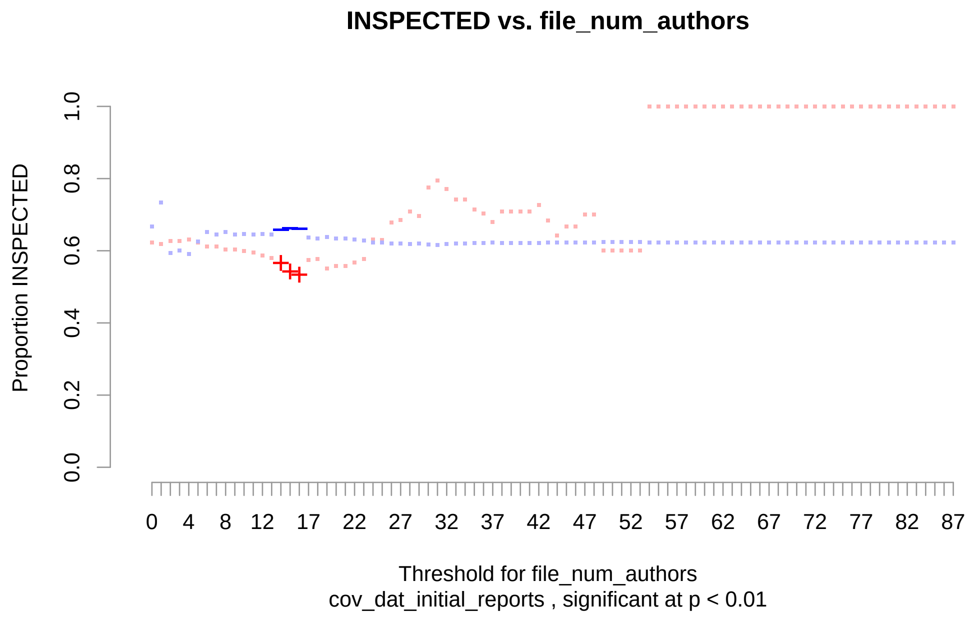
However, before concluding that `file_num_authors` has a significant effect on inspection rates independent of `file_age`, let's look again at its contrasting effects for initial vs. subsequent scan reports, as shown in Figure 6.

Table 12 shows what happens when multiple regression models are created separately for initial and subsequent scan reports. As expected, for the initial scan reports, neither `file_age` nor `file_num_authors` has any significant effect on inspection rates (*p-values* of 0.69 and 0.45, respectively). However, for the subsequent scan reports, `file_num_authors` is, in fact, redundant, with a *p-value* of 0.61. Sadly, the apparent benefit of `file_num_authors` in Table 11 now disappears.

4.4 Useless explanatory variables

Empirical studies often only report what trials worked well, but it can be useful to also report what didn't work so well. The following types of explanatory variables made for poor models for predicting `INSPECTED`:

- **Number of monofile patches** – simply using the number of patches worked better
- **Code churn**, especially short-term metrics like percentage churn in the past 1 or 6 months – a related study with FindBugs at Google (Ruthruff, Penix, et. al. ICSE 2008) also found that code churn was a noisy factor that made for poor models
- **Number of modified lines**, especially short-term metrics like # mod. lines in the past 1 or 6 months – file age and number of patches worked better



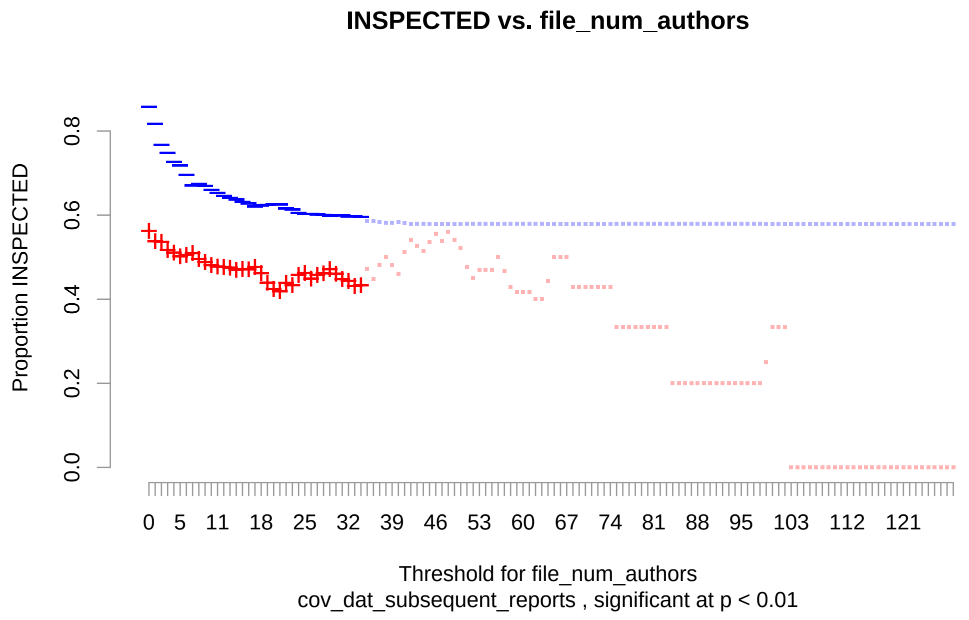


Figure 6: Inspection rates for various thresholds of `file_num_authors` for *initial scan* reports (top) and *subsequent scan* reports (bottom). For significant differences with chi-square $p < 0.01$, files with more authors than threshold marked in red $+$, files with fewer authors than threshold marked in blue $-$. Muted-colored dots are for chi-square $p \geq 0.01$.

4.5 Multiple regression models

Leveraging the insights developed in §4.2, §4.3, and §4.4, I've built multiple regression models that are stronger than the single regression models of §4.1. There is often no single 'best' model in practice, so I will present a few that fit my dataset relatively well and have some chances of generalizing.

My strategy was to start with a set of explanatory variables that worked reasonably well by themselves and then add more variables and see whether they each decrease the residual deviance by a statistically significant amount. I favored parsimonious models (with fewer variables) because they are more likely to generalize to new datasets.

4.5.1 Basic model

If I take the four most impactful variables as determined in §4.2 and jam them together into a model, they end up working quite well together:

```
INSPECTED ~
checker + toplevel_dirname + file_age + file_num_lines.log
```

Term	coefficient	deviance	p-value
`NULL`		2814.8	
`checker`	N/A	2637.2	3.2e-32
`toplevel_dirname`	N/A	2600.3	0.0013
`file_age`	−0.00013	2583.2	3.6e-5
`file_num_lines.log`	−0.27	2558.4	6.3e-7

ROC area: 0.70
Deviance: 2558

Each variable helps to decrease the deviance of the model by a statistically significant amount (with $p << 0.01$), so they don't 'step on each other's toes'. A four-variable model is fairly parsimonious, and each seems to exert an independent effect on the inspection probability. The ROC area is 0.70, which is slightly better than the 0.65 achieved by a model that simply uses `checker` alone. This means that the other 3 variables actually add relatively little to classification power, although they do reduce model deviance by significant amounts (i.e., they create a 'tighter' fit but don't allow the model to classify members of the training dataset with much higher accuracy).

Figure 7 shows the ROC curve for this model, which is color-coded according to the threshold. A particular choice of threshold marks a point on

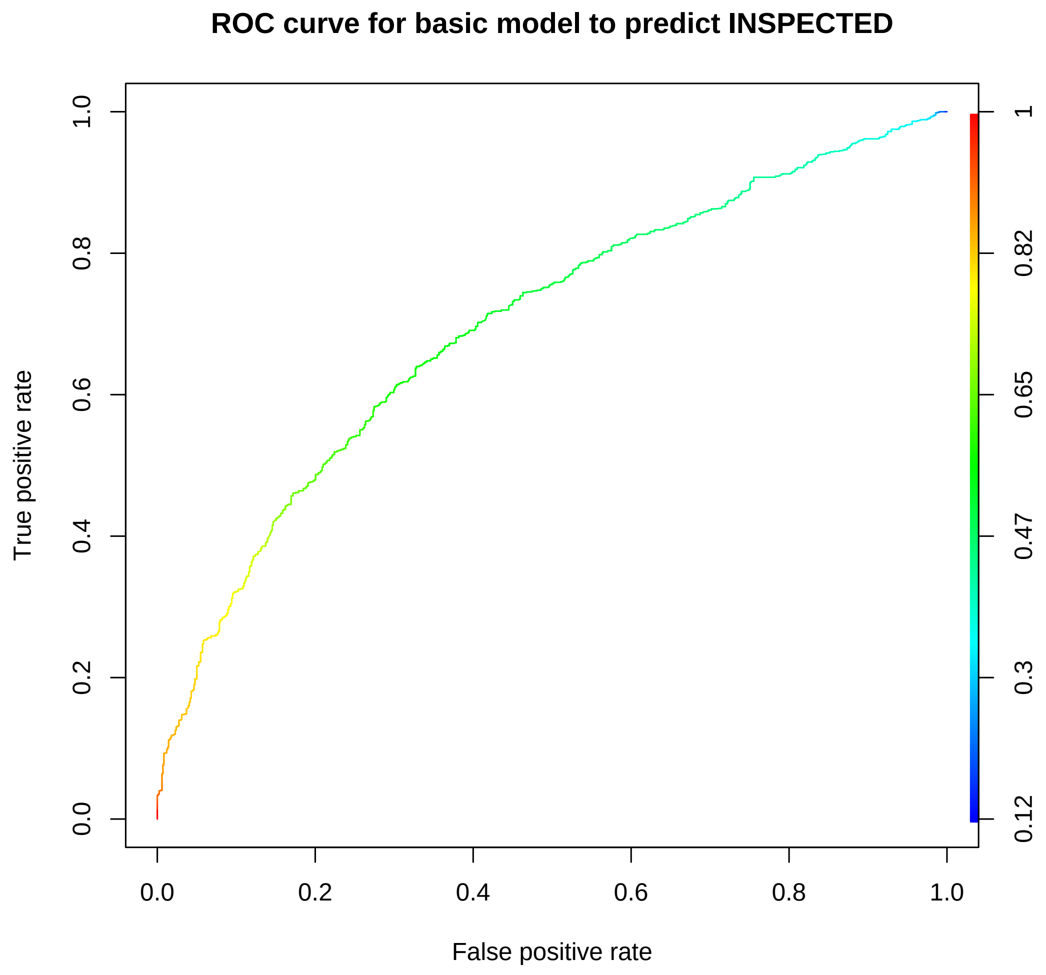


Figure 7: ROC curve for basic model (area = 0.70).

the curve, which uniquely maps to true and false positive rates. Blue (low threshold) points appear in the upper-right corner while red (high threshold) points appear in the lower-left corner. Recall that the threshold is the probability at which the classifier predicts a report to be `INSPECTED`. For instance, with a low threshold of, say 0.1, almost all reports are classified as `INSPECTED` (as long as the model predicts an inspection probability of ≥ 0.1); this results in a near-perfect true positive rate (almost all inspected reports are properly classified) but also in a high false positive rate (almost all reports that weren't inspected are falsely classified as inspected). In contrast, with an unbelievably high threshold of, say 1, no reports are classified as `INSPECTED`; this results in a zero false positive rate (which might seem promising), but unfortunately also results in a zero true positive rate (because nothing is classified as inspected).

A strong model should strive to attain the highest possible true positive rate while keeping the false positive rate relatively low; a model that does so will have *a high arch* and thus a high ROC area. The higher the arch of the curve, the greater the area, which means greater classification accuracy on the training dataset. A model making random predictions results in an area of ~ 0.5. This model has an area of 0.7, which is respectable but not amazing (models with areas approaching 0.9 are exceptional).

4.5.2 Enhanced model (found by computer search program)

There are many more possible explanatory variables that I could add to the basic model with the hopes of improving its reduction in deviance and ROC area. I first tried adding variables manually and assessing their effects, but then I decided to automate the process by writing a program to search through the space of possible models. This is a fairly straightforward search algorithm that incrementally adds new variables and interaction terms and assesses whether each one reduces the deviance of its predecessor model by a statistically significant amount according to the analysis of deviance chi-square test.

Because the search space is exponential (an exhaustive search would consider all possible subsets of explanatory variables and their interactions), my intent was never to wait for my algorithm to terminate, but rather to run it for long enough (overnight to a few days) so that I could inspect at least a few dozen generated models. Thus, I used an iterative deepening strategy, which favors generating simpler models (shallower solutions in the search tree) before more complicated ones but doesn't have the immense memory requirements of breadth-first search. In contrast, when I initially tried using depth-first search, it always generated overly-complex models because its directive is to return the deepest solutions first.

A significant danger in using computer programs to search for models is that they often find overly-complex models that overfit the dataset (i.e., by trying enough possibilities, some are bound to fit well just by pure luck). Thus, it's always important to have a human 'in the loop' to manually assess whether variables in computer-generated models make intuitive sense.

Here is a reasonable enhanced model that my program found:

```
INSPECTED ~
checker + toplevel_dirname + file_age +
(file_age_in_years * file_num_lines.log) + dir_num_mod_lines
```

Term	coefficient	deviance	p-value
`NULL`		2814.8	
`checker`	N/A	2637.2	3.2e-32
`toplevel_dirname`	N/A	2600.3	0.0013
`file_age`	−7.3e-04	2583.2	3.6e-5
`file_age_in_years`	N/A	2564.2	0.0043
`file_num_lines.log`	−0.39	2543.0	4e-6
`dir_num_mod_lines`	−1.3e-06	2539.1	0.05
`file_age_in_years:file_num_lines.log`	N/A	2523.0	0.01

ROC area: 0.71
Deviance: 2523

This model contains two extra variables and a pair of variables that are bound together in an *interaction term* (recall its definition from §3.1.2): `(file_age_in_years * file_num_lines.log)`. My search program found that the effects of these two variables complemented one another, so it added their cross-product as a derived variable. In this case, the effects of number of lines in a file vary for files of different ages, so adding these interaction effects to the model made it stronger.

However, this more complicated model only improved upon the ROC area of the basic model by 0.01 (0.71 vs. 0.70), but the deviance has been decreased from 2558.4 to 2523.0, which is decent. I still don't have a good intuition about whether the incremental improvements of this computer-generated model outweigh its added complexity.

4.5.3 Models for initial scan reports

Fitting the same variables in the basic model to the subset of data containing only initial scan reports results in the following model:

Only considering *initial scan* reports

`INSPECTED ~ checker + toplevel_dirname + file_age + file_num_lines.log`

Term	coefficient	deviance	p-value
`NULL`		1284.7	
`checker`	N/A	1227.9	1.4e-8
`toplevel_dirname`	N/A	1203.2	0.04
`file_age`	3.8e-5	1203.1	0.79
`file_num_lines.log`	−0.38	1180.4	1.9e-6

ROC area: 0.68
Deviance: 1180

Notice that the *p-value* for `file_age` is huge (0.79), which means it's practically useless. Recall from §4.2.3 that file age didn't matter at all for inspection rates in initial scan reports. We can thus eliminate it and form a simpler model:

Only considering *initial scan* reports

`INSPECTED ~ checker + toplevel_dirname + file_num_lines.log`

Term	coefficient	deviance	p-value
`NULL`		1284.7	
`checker`	N/A	1227.9	1.4e-8
`toplevel_dirname`	N/A	1203.2	0.04
`file_num_lines.log`	−0.38	1180.4	1.9e-6

ROC area: 0.68
Deviance: 1180

This simplified model has one fewer explanatory variable but still retains the same final deviance and ROC area. I used this model as the starting point for my computer search program and let it run overnight. The end result was a large family of overly-complicated models that fit the dataset significantly better, but I think that they might suffer from overfitting problems due to their sheer complexity. Here is one such representative computer-generated model (for brevity, I omit the individual coefficients, deviances, and *p-values*):

Only considering *initial scan* reports

```
INSPECTED ~
checker + toplevel_dirname + file_has_gt_2000_lines +
(dir_has_gt_50_files * file_num_lines.log) +
(dir_num_patches_1_month_prior * file_age_in_years) +
(dir_num_monofile_patches_6_months_prior * dir_num_patches) +
dir_num_monofile_patches_1_month_prior
```

ROC area: 0.76
Deviance: 1079

The ROC area of 0.76 is a noticeable improvement over the 0.68 of my hand-made model, and the deviance has decreased significantly as well. However, I am hard-pressed to provide an intuitive justification for why this complicated model (with 10 variables and 3 pairs of interaction terms) should generalize beyond the dataset that it was trained upon.

Even though I am wary of computer-generated models, one use for them is to *serve as inspirations* for creating new hand-made models. In this case, I noticed that `file_has_gt_2000_lines` and `dir_num_patches_1_month_prior` seemed to have noticeable effects, so I added them to my basic model:

Only considering *initial scan* reports

```
INSPECTED ~
checker + toplevel_dirname + file_num_lines.log +
file_has_gt_2000_lines + dir_num_patches_1_month_prior
```

Term	coefficient	deviance	p-value
`NULL`		1284.7	
`checker`	N/A	1227.9	1.4e-8
`toplevel_dirname`	N/A	1203.2	0.04
`file_num_lines.log`	0.04	1180.7	2.1e-6
`file_has_gt_2000_lines`	−0.96	1163.2	3e-5
`dir_num_patches_1_month_prior`	−0.0038	1158.2	0.02

ROC area: 0.70
Deviance: 1158

Adding these two variables boosted the ROC area of the basic model from 0.68 to 0.70 and decreased the deviance by a bit without increasing the complexity of the model by too much, so it seems to be an overall improvement.

4.5.4 Models for subsequent scan reports

Fitting the same variables in the basic model to the subset of data containing only subsequent scan reports results in the following model:

Only considering *subsequent scan* reports			
`INSPECTED ~ checker + toplevel_dirname + file_age + file_num_lines.log`			
Term	coefficient	deviance	p-value
`NULL`		1525.5	
`checker`	N/A	1347.0	4.8e-33
`toplevel_dirname`	N/A	1317.0	0.0047
`file_age`	-0.00013	1283.5	7.1e-9
`file_num_lines.log`	-0.27	1277.0	0.01
ROC area: 0.77 Deviance: 1277			

The ROC area of classifying subsequent scan reports is 0.77, which is much higher than the 0.70 for all reports and 0.68 for initial scan reports. This suggests that the subsequent scan data has less noise than the initial scan data.

Again I decided to let the computer generate lots of models that improved upon this basic one. The following model was one of the cleaner ones that came out of the overnight run (individual coefficients omitted for brevity):

Only considering *subsequent scan* reports

```
INSPECTED ~
checker + toplevel_dirname + file_days_since_last_patch +
(file_age * num_prev_inspected_reports_in_dir_FACTOR) +
num_prev_inspected_FALSE_reports_in_dir +
(dir_num_mod_lines_6_months_prior *
 num_prev_inspected_FALSE_reports_in_dir_FACTOR)
```

ROC area: 0.85
Deviance: 1066

The ROC area is an impressive 0.85! This model is actually not too complicated, only containing 8 variables and 2 pairs of interaction terms. Moreover, the new variables include those indicating the number of previous inspected reports, which according to §4.2.6 have a definite effect on inspection rates. I'm more confident about this computer-generated model than the one generated for initial scan reports.

The final model I want to present in this section was created by manually augmenting the basic model with relevant variables for days since most recent patch and presence of previous inspected reports according to my investigations in §4.2.5 and §4.2.6, respectively:

Only considering *subsequent scan* reports

```
INSPECTED ~
checker + toplevel_dirname + file_age + file_num_lines.log +
file_days_since_last_patch + file_has_prev_inspected_BUGGY_reports
```

Term	coefficient	deviance	p-value
`NULL`		1525.5	
`checker`	N/A	1347.0	4.8e-33
`toplevel_dirname`	N/A	1317.0	0.0047
`file_age`	−0.0003	1283.5	7.1e-9
`file_num_lines.log`	−0.21	1277.0	0.01
`file_days_since_last_patch`	0.00027	1272.2	0.03
`file_has_prev_inspected_BUGGY_reports`	0.59	1262.8	0.002

ROC area: 0.774
Deviance: 1263

Although the newly-added variables are definitely relevant, sadly the overall improvement in ROC area and deviance was barely noticeable.

The general take-home lesson from this section is that trying to tune models to squeeze out the last ounce of residual deviance or ROC area improvement is often futile. Similar to other optimization problems in practice, small tweaks don't help much after you've located the variables that have the largest effects and added those to your model. The incremental gains of manually adding new variables are often small, and non-judicious use of computer search programs can lead to overly-complex overfit models.

5 Predicting whether an inspected report will be resolved

To create models for other response variables, such as whether a report will be resolved, I could go through the same detailed steps as in §4. However, due to lack of time (and space in this document), I have opted for a simpler approach: simply letting the computer generate models for me and then picking the simplest one that looks reasonable.

In this section, I want to only use the subset of reports that have been `INSPECTED` and create a model to predict which of these reports will be `RESOLVED`.

5.1 Single regression models

Tables 15 and 16 in the Appendix show the results of creating single logistic regression models for each individual explanatory variable to assess their individual effects.

Interestingly, `toplevel_dirname` and `file_age`, which had significant effects on inspection probability, seem to have little effect here, with *p-values* of 0.53 and 0.84, respectively. I haven't investigated in detail, but perhaps once a bug has already been successfully triaged, the chances of it being fixed doesn't much depend on where the file is located or how old it is. One possibility is that once a report is put on a queue of bugs to fix, then developers will fix bugs without bias for file location or age, but what reports to triage in the first place is definitely biased by file location and age.

The other impactful variables are quite similar to the ones for `INSPECTED` presented in §4.2, but there is one particular new variable of interest: the number of days after a report has been issued before it is inspected.

5.2 Impactful variable: Days before inspection

The longer that developers wait to inspect a report, the less likely it is that it will eventually be resolved. Figure 8 shows the differences in inspection rates. Reports inspected within around the first 100 or so days are 2 to 3 times more likely to be resolved than those that developers waited longer to inspect (perhaps due to apathy or uncertainty about whether the report indicated a true bug). This variable should definitely go into a multiple

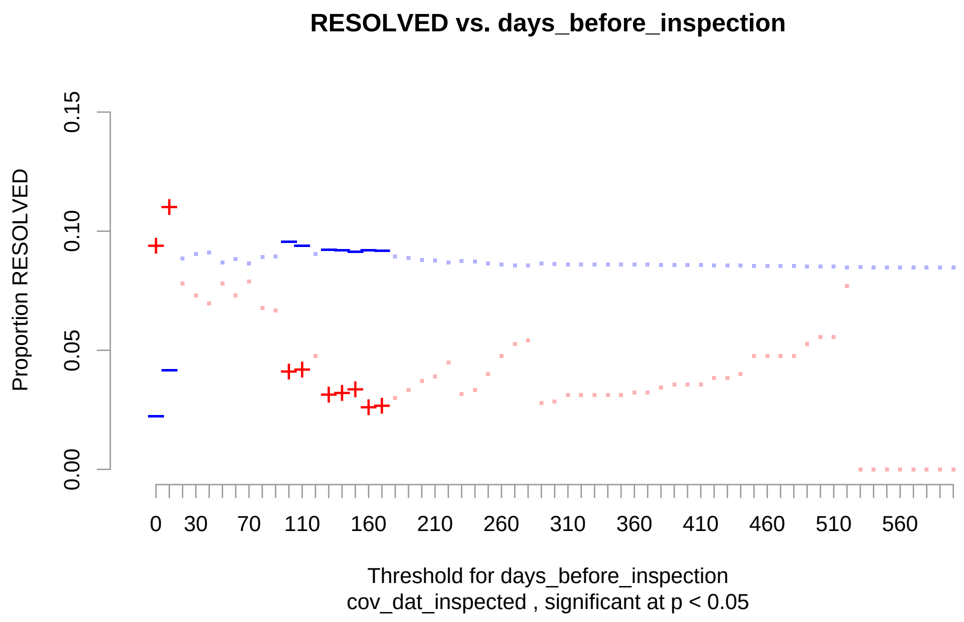


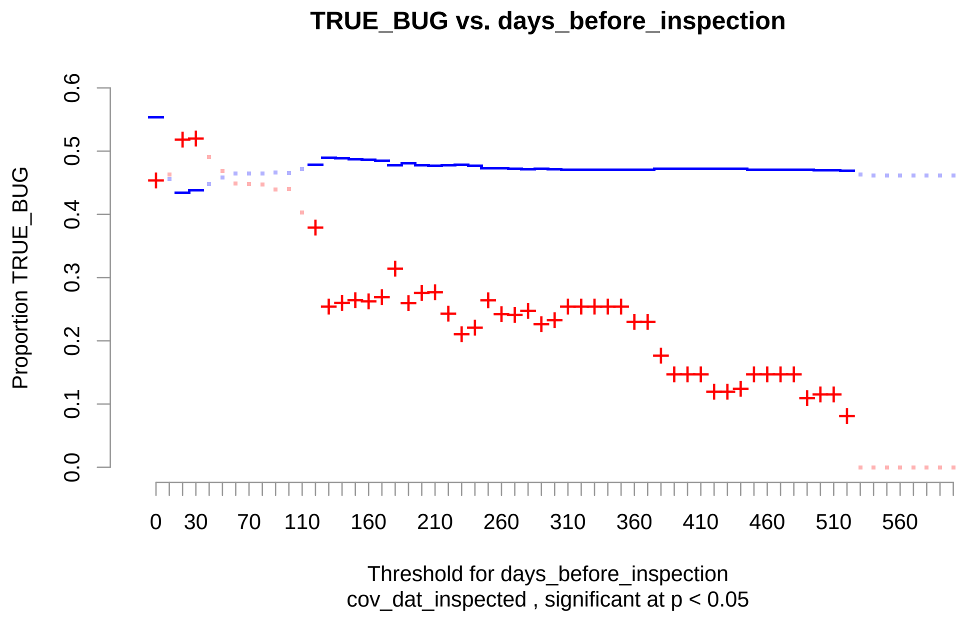
Figure 8: Resolution rates for various thresholds of `days_before_inspection` only for reports that were inspected. For significant differences with chi-square $p < 0.05$, reports with days before inspection greater than threshold marked in red $+$, less than threshold marked in blue $-$. Muted-colored dots are for chi-square $p \geq 0.05$.

regression model for predicting `RESOLVED`.

Figure 9 shows similar graphs for the proportions of reports that were marked as true bugs (top) and whose veracity could not be confirmed (bottom) split by thresholds for `days_before_inspection`. (A report is marked as a `TRUE_BUG` if its final status is either `BUG`, `IGNORE`, or `RESOLVED`, and marked as `UNSURE` if its final status is `PENDING`.)

Sure enough, as developers wait longer to inspect a report, the less likely it will be for that report to actually be a true bug and the more likely it will be that they cannot determine its true nature (the third alternative, which is to mark it as a false positive, showed no dramatic effects, so I have not plotted it). The disparities between the 'inspected quickly' and 'inspected after long delay' groups actually grow larger as the threshold increases.

These numbers suggest that if developers aren't able to successfully triage a report quickly, then the odds are against them ever figuring out what to do with it.



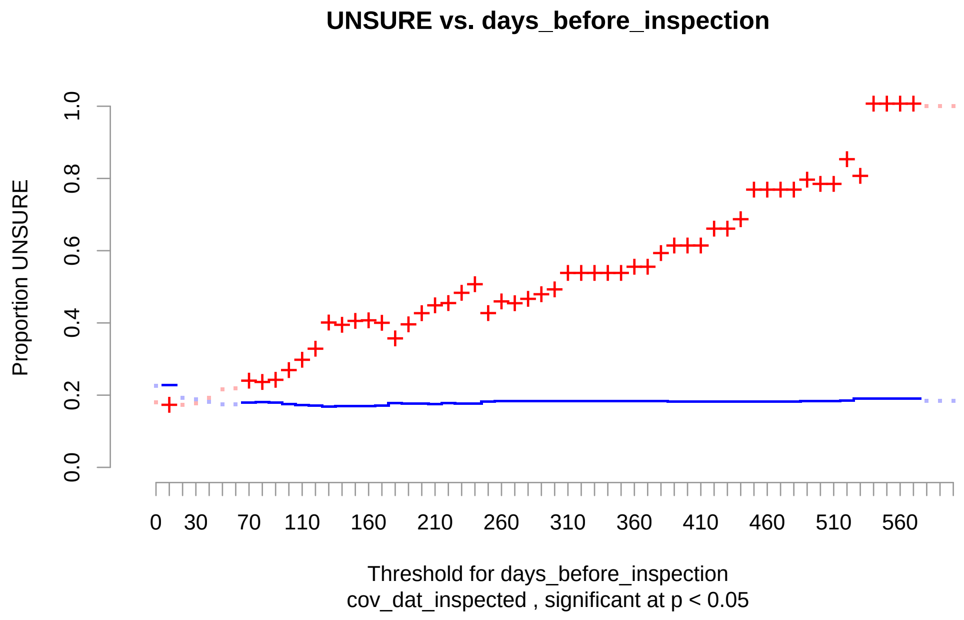


Figure 9: Proportion of reports marked as true bugs (top) and whose veracity could not be confirmed (bottom), out of all inspected reports, based on a threshold of `days_before_inspection` (greater than threshold marked with red +, less than threshold marked with blue −). Chi-square significance test set at $p = 0.05$.

5.3 Multiple logistic regression model

I will now present an example of a strong yet semi-parsimonious model that my search program found. Note that I did not start this search with any initial model as a seed; the computer started from the null model and added new variables as fit, generating numerous models and ranking them based on AIC (which is a combined measure of reduction in deviance and parsimony). I picked one particular model whose variables seemed to make sense to me:

Only considering reports that were `INSPECTED`

```
RESOLVED ~
checker +
(dir_has_prev_inspected_reports * dir_num_mod_lines_1_month_prior) +
(file_days_since_last_patch * file_has_gt_5_authors) +
days_before_inspection
```

Term	coefficient	deviance	p-value
`NULL`		726.2	
`checker`	N/A	660.3	7.4e-10
`dir_has_prev_inspected_reports`	−1.4	649.8	0.0012
`dir_num_mod_lines_1_month_prior`	−4.7e-4	648.1	0.19
`file_days_since_last_patch`	−4.7e-4	639.7	0.0037
`file_has_gt_5_authors`	1.0	636.03	0.06
`days_before_inspection`	−0.0024	631.0	0.02
`dir_has_prev_inspected_reports: dir_num_mod_lines_1_month_prior`	N/A	624.1	0.01
`file_days_since_last_patch: file_has_gt_5_authors`	N/A	607.9	5.7e-5

ROC area: 0.78

The ROC area of 0.78 is fairly respectable, and it only contains 6 variables and 2 pairs of interaction terms, which isn't too complicated. Unfortunately, due to the interaction terms, it is difficult to ascertain the nature of each variable's contributions to predicting the resolution probability by looking at its coefficient. For instance, `file_has_gt_5_authors` has a positive coefficient, but perhaps its correlation with resolution probability is still negative due to coefficients (not shown) of the interaction term (`file_days_since_last_patch * file_has_gt_5_authors`).

Reassuringly, though, `days_before_inspection` has a negative coefficient, which corroborates the hypothesis that the longer developers wait before inspecting a report, the less likely it will be resolved.

Also, `file_days_since_last_patch` has a negative coefficient, which indicates that files that have been patched less recently have a lower probability of their bug reports being resolved. This even takes into account files that have *never been patched* (with `file_days_since_last_patch` taking a sentinel value of 3,000 days). Perhaps developers are more reluctant to patch files that they haven't patched recently. (Recall that for predicting `INSPECTED`, there was actually a positive coefficient; see §4.2.5).

6 Future work

This work is currently ongoing, so there are many possible directions for future work, including:

- Performing cross-validation to determine model prediction accuracy
- Testing these models (which were built from a limited Linux dataset) on other open source projects to see how well they generalize
- Correlating Coverity Scan bug reports with developer-reported bugs
- Showing results to the Linux developer community to get their feedback and anecdotal opinions, which could lend greater veracity to quantitative research findings
- If it's possible to obtain additional Coverity Scan reports for other projects, it would be interesting to try to find bug and bugfix patterns that generalized throughout open source development

Appendix: Full data tables

	(lower is better)		(higher is better)
Explanatory variable	p-value	AIC	ROC area
checker	3.2e-32	2661	0.65
file_num_authors.log	1.8e-12	2769	0.59
file_has_gt_1_authors	1.9e-11	2773	0.55
file_num_patches.log	2.6e-11	2774	0.58
file_age_in_years	2.1e-10	2772	0.59
file_age	4.4e-10	2779	0.59
file_has_gt_15_authors	2e-09	2782	0.56
file_num_mod_lines.log	2.2e-09	2783	0.56
file_has_gt_20_patches	7e-09	2785	0.56
file_num_authors	9.1e-08	2790	0.59
file_has_gt_5_authors	1.6e-07	2791	0.55
file_days_since_last_monofile_patch	3.1e-07	2792	0.53
file_days_since_last_patch	7.7e-07	2794	0.54
file_has_gt_2000_lines	1.7e-06	2795	0.55
toplevel_dirname	2.7e-06	2792	0.57
file_num_lines.log	9.8e-05	2803	0.56
dir_num_patches_6_months_prior	0.00017	2804	0.54
file_num_lines	0.00017	2804	0.56
dir_num_patches	0.0002	2804	0.57
dir_num_patches_1_month_prior	0.00042	2806	0.54
file_num_authors_1_month_prior	0.00045	2806	0.54
file_has_gt_100_percentage_churn	0.00057	2806	0.54
dir_num_mod_lines	0.00097	2807	0.56
file_num_authors_6_months_prior	0.0011	2808	0.54
file_num_patches_6_months_prior	0.005	2810	0.54
file_num_patches	0.0051	2810	0.58
dir_num_lines.log	0.006	2811	0.53
dir_num_lines	0.0085	2811	0.53

Table 13: Single logistic regressions to predict INSPECTED, sorted by analysis of deviance test chi-square *p-values* (Part 1 of 2)

	(lower is better)		(higher is better)
Explanatory variable	p-value	AIC	ROC area
`dir_num_monofile_patches`	0.013	2812	0.55
`file_num_patches_1_month_prior`	0.015	2812	0.53
`file_has_gt_500_lines`	0.017	2813	0.52
`dir_has_gt_5_files`	0.021	2813	0.51
`dir_num_monofile_patches_6_months_prior`	0.026	2813	0.52
`dir_num_monofile_patches_1_month_prior`	0.031	2814	0.51
`file_num_monofile_patches_1_month_prior`	0.061	2815	0.51
`dir_has_gt_50_files`	0.074	2815	0.52
`file_percentage_churn`	0.083	2815	0.55
`dir_num_mod_lines_6_months_prior`	0.092	2815	0.52
`dir_num_files`	0.1	2816	0.52
`file_num_mod_lines`	0.13	2816	0.56
`dir_num_mod_lines_1_month_prior`	0.21	2817	0.53
`file_num_mod_lines_6_months_prior`	0.37	2818	0.52
`file_num_mod_lines_1_month_prior`	0.42	2818	0.52
`file_num_monofile_patches_6_months_prior`	0.44	2818	0.51
`file_has_gt_100_patches`	0.65	2818	0.5
`file_percentage_churn_6_months_prior`	0.66	2818	0.51
`file_num_monofile_patches`	0.71	2818	0.56
`file_percentage_churn_1_month_prior`	0.87	2818	0.52

Table 14: Single logistic regressions to predict `INSPECTED`, sorted by analysis of deviance test chi-square *p-values* (Part 2 of 2)

	(lower is better)		(higher is better)
Explanatory variable	p-value	AIC	ROC area
`checker`	7.4e-10	684	0.69
`dir_has_prev_inspected_reports`	0.00024	716	0.58
`file_has_gt_1_authors`	0.00066	718	0.55
`num_prev_inspected_reports_in_dir`	0.0025	721	0.58
`dir_num_mod_lines_1_month_prior`	0.0027	721	0.6
`file_days_since_last_patch`	0.0042	722	0.52
`dir_num_patches_1_month_prior`	0.0055	722	0.59
`file_has_gt_2000_lines`	0.021	724	0.55
`dir_num_lines`	0.028	725	0.55
`file_num_lines`	0.032	725	0.56
`file_num_authors.log`	0.039	725	0.54
`dir_num_mod_lines_6_months_prior`	0.041	726	0.52
`dir_num_lines.log`	0.043	726	0.55
`file_has_gt_5_authors`	0.046	726	0.55
`dir_num_monofile_patches_1_month_prior`	0.048	726	0.57
`file_num_mod_lines_6_months_prior`	0.058	726	0.5
`dir_num_mod_lines`	0.061	726	0.53
`file_has_gt_100_patches`	0.065	726	0.53
`dir_num_monofile_patches_6_months_prior`	0.075	727	0.51
`file_age_in_years`	0.076	728	0.59
`file_percentage_churn_1_month_prior`	0.083	727	0.52
`dir_num_monofile_patches`	0.084	727	0.51
`file_num_authors_6_months_prior`	0.084	727	0.58
`file_num_mod_lines_1_month_prior`	0.093	727	0.52
`file_num_lines.log`	0.094	727	0.56

Table 15: Single logistic regressions to predict `RESOLVED` for reports that were already `INSPECTED`, sorted by analysis of deviance test chi-square *p-values* (Part 1 of 2)

	(lower is better)		(higher is better)
Explanatory variable	p-value	AIC	ROC area
file_num_patches.log	0.11	727	0.52
dir_num_patches_6_months_prior	0.12	727	0.51
days_before_inspection	0.14	728	0.46
dir_num_patches	0.16	728	0.51
file_has_gt_100_percentage_churn	0.17	728	0.53
num_prev_inspected_reports_in_dir_FACTOR	0.2	757	0.62
dir_num_files	0.22	728	0.54
file_num_mod_lines.log	0.26	728	0.5
file_has_prev_inspected_reports	0.32	729	0.52
file_num_mod_lines	0.34	729	0.5
dir_has_gt_50_files	0.35	729	0.52
file_days_since_last_monofile_patch	0.35	729	0.53
file_percentage_churn_6_months_prior	0.35	729	0.48
num_prev_inspected_reports_in_file_FACTOR	0.36	736	0.53
dir_has_gt_5_files	0.38	729	0.51
num_prev_inspected_reports_in_file	0.46	729	0.52
file_num_patches_1_month_prior	0.5	729	0.52
file_percentage_churn	0.51	729	0.54
file_num_authors	0.52	729	0.54
toplevel_dirname	0.53	744	0.57
file_num_authors_1_month_prior	0.6	729	0.49
file_num_monofile_patches_1_month_prior	0.68	730	0.5
file_has_gt_20_patches	0.74	730	0.51
file_num_patches	0.75	730	0.52
file_has_gt_500_lines	0.75	730	0.51
file_has_gt_15_authors	0.76	730	0.51
file_age	0.84	730	0.54
file_num_monofile_patches_6_months_prior	0.86	730	0.47
file_num_monofile_patches	0.94	730	0.53
file_num_patches_6_months_prior	0.97	730	0.55

Table 16: Single logistic regressions to predict RESOLVED for reports that were already INSPECTED, sorted by analysis of deviance test chi-square *p-values* (Part 2 of 2)