# Risk-Assessment Model

## Data Synthesis

**1.1 - Justification**

Early in the development, we decided that the predictive Risk-Assessment Model should be trained on an artificially-generated dataset of simulated projects. This decision was made due to two main reasons: first, real datasets of successful software projects are not readily available (or at least, we were unable to find such datasets in our research), and second, if such a dataset does exist, then it is unlikely to include the same metrics our model tracks, so it would be of limited use for training. However, the decision to artificially generate a dataset introduces a significant element of bias into our system, as the model may only learn the relationship(s) present in the dataset. As such, it is crucial that the generated data reflects real-world software development projects. Alternatively, if the project were of a larger scale or were allocated more time, then we could pursue obtaining our own dataset via surveys sent to project managers and software development companies.

**1.2 - Project Simulation**

The main component of the simulation is the SimProject object which represents a single artificial project, whose state is sampled regularly throughout its progression. Initial hard metrics such as budget and deadline are randomly-generated within a given range, while soft metrics relating to the team and their experience are generated arbitrarily. As such, projects can be generated for a specific range of budgets/deadlines, but the remaining attributes may take any values, which provides variation in the dataset.

The *.simulate()* method emulates the lifetime of the project, with a new state (dataframe row) being recorded once per simulated day. Over the simulation, the progress of the project is maintained and incremented via a function on the team experience and soft metrics, with some randomness. The project is deemed to be complete and the simulation stops when that progress reaches 100%. However, in some cases, projects may experience “negative” progress, a probabilistic event that is more likely to occur in teams with low morale. Similarly, while the cost of development is regularly updated as it progresses, projects may also incur “unexpected costs” via a similar probabilistic mechanism. These features aim to replicate some of the challenges faced by projects, as development may not progress entirely as expected or as planned. Furthermore, a project may also be cancelled prematurely, with a probability inversely related to the managerial support. When a project exceeds its budget or deadline, the support metric begins to decline geometrically, thus increasing the chance of a cancellation. When a project is cancelled, it is considered to be a failure, as no viable product was produced.

There are several other simulated components of interest, which were influenced by relevant research. First, the main cost faced by the simulated projects is that of the development team (I.e. wages), which is modelled as a function of the experience rank of the developers. The relationship is shown in Table 1, but can be expressed as: .

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Rank** | **Experience** | **Hourly Wage** | **x Daily Hours** | **= Daily Wage** | **Calculated Daily Wage** |
| 1 | Beginner | £15-20 | 8 | £120-160 | £75 + £75x1 = £150 |
| 5 | Advanced | £40-60 | 8 | £320-480 | £75 + £75x5 = £450 |

**Table 1**: Developer Costs by Experience. Hourly Wages provided by

(**SOURCE**: <https://www.approvedindex.co.uk/software-developers/software-development-prices>)

Next, from the code-base metrics, the simulation is only concerned with two areas in particular. First, the total number of code commits is simulated, as the change in commit frequency can be compared to estimate the relative activity of the development team. Moreover, our research found a study investigating the relationship between commit frequency and project success [**Source**: *https://www.academia.edu/36780849/The\_empirical\_commit\_frequency\_distribution\_of\_open\_source\_projects*] which measured project activity by the following formula:

Since the study only concerned Open-Source projects and our system is more suited to commercial, closed-source project, we decided to adapt the formula to consider the commit frequency for the last month (30 days) instead of 6 months. In particular, this metric is used to evaluate

The second modelled code metric is the rate at which known bugs are resolved via commits, in order to provide an estimation of the quality of the project’s code and the engagement of the development team with known issues. For each developer, the number of commits is generated probabilistically, based on developer experience and soft metrics such as the team commitment and morale; then for each commit, the developer has a probability of introducing a new bug or fixing an existing a bug. As the project progresses, the resolution rate tracks the ratio of resolved bugs over the total number of known bugs. One limitation of this approach is that defect rate may only provide a partial view of the code, as this metric’s accuracy depends on issues being identified and flagged [**Source**: *https://www.sealights.io/code-quality/defect-density-context-is-king/*].

Finally, at various points throughout the simulation, generic data-manipulation functions -namely, Exponential or Sigmoid functions - are used to control the distribution of some project quantities (for example, probabilities or costs). To be specific, the exponential function is useful for reducing the likelihood of large unexpected costs being generated, while an exponential-decay function reduces the effect of developer experience on the likelihood of producing bugs (but prevents the returned value from reaching zero). In a similar manner, some division operations may produce arbitrarily large floats in the range [0, infinity), so the Sigmoid function is used to map the input value to the range [0,1] (with a mean of 0.5), thus allowing for more meaningful comparison.

**1.3 - Success Evaluation and Propagation**

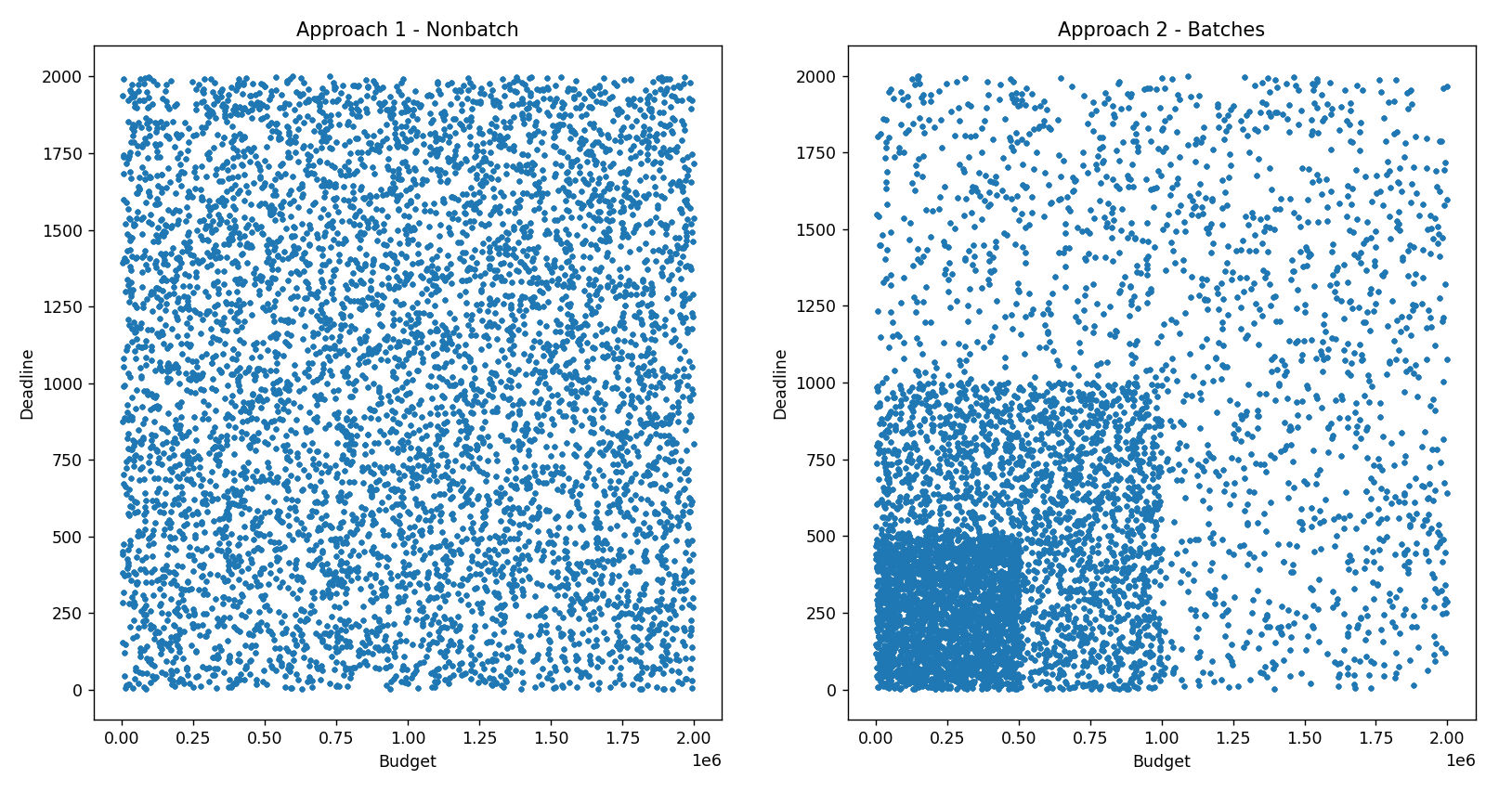
A critical element of the Data-Generation involves determining class-labels for the simulated projects upon their completion; in other words, this is the problem of deciding whether to label the project as a Success (1) or Failure (0). We decided that the success of a project can be considered in terms of five components: Finance, Timescale, Code, Team, and Management. When the project is complete, each of these components is evaluated separately based on the last state to be recorded (I.e. the completion state). This action produces five scores (each between 0 and 1) which can be combined by the *SuccessReport* object into a single overall evaluation for the project’s success. To be precise, a weighted-sum is applied, in order to emphasise the significance of the Finance and Time aspects in the overall result, before being normalised by dividing by the maximum possible score if all aspects were 1.

However, our design set out that the Machine-Learning model would be a classifier, rather than a regression model and indeed, we want to determine if a project will be a success, rather than calculating a score. As such, we apply so-called *binarisation* to the 6 calculated scores. Each component is allocated a minimum threshold for success, and if the score exceeds that threshold, then the component is marked as success (1); otherwise, the component is marked a failure (0). As mentioned previously, if the project is cancelled, then all 6 of these bits are set to 0. Then, after identifying the six success bits, this data is propagated backwards, being written to all prior states of that project, such that they act as “foresight” for the result of the project.

**1.4 - Data Generation**

After designing a mechanism for producing labelled data, the next step was to produce a dataset using those procedures. It was decided that the dataset should be stored in CSV format, which can be easily translated to and from a Pandas dataframe. It is important to note, though, that our intention was for the model to handle projects at varying stages of development, so we decided that each generated project should be sampled multiple times (in order to reduce the number of projects which are required). However, we also wanted to avoid oversampling a small set of projects which would lead to a higher chance of the model overfitting to the given data, rather than generalising. Our testing consisted of generating 3-5 samples per project, ultimately settling on the lower-end for greater variation in the data-set.

Next, we encountered some challenges with the scope of the dataset. The primary goal of the model is to generalise and be applicable for a range of project types and sizes, so we began by randomly initialising projects with budgets in the range 100 to 2,000,000 and deadlines from 1 to 2000 days. However, the uniformity of this dataset was found to be an issue, as most projects actually have a budget in the range of $50k-$250k (**Source**: <https://www.uptech.team/blog/software-development-costs)> with an expected duration of 4-12 months (120-360 days) (**Source**: *[https://www.brainspire.com/blog/how-long-does-it-take-to-build-custom-software-for-a-business](https://www.brainspire.com/blog/how-long-does-it-take-to-build-custom-software-for-a-business).)*[).](https://www.brainspire.com/blog/how-long-does-it-take-to-build-custom-software-for-a-business).) Therefore, we decided to reflect this distribution in the training data-set, by increasing the concentration of projects in this range. Although, to ensure we did not restrict the model’s generalisability, the dataset still includes some projects outside of this interval.

****

*Figure 1: An example of uniform data-generation vs. clustered data-generation*

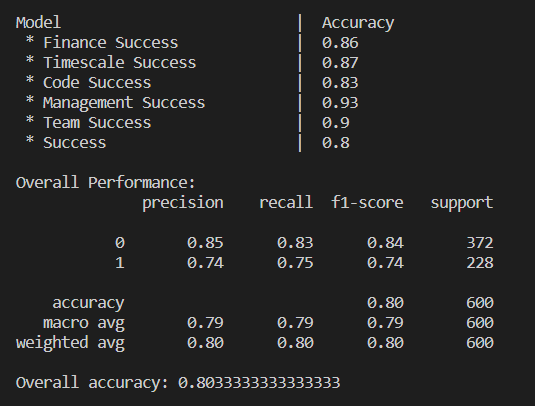
## Model Training

### 2.1 - Model Structure

The Risk-Assessment-Model consists of six sub-models, each trained to predict a single Success component; in other words, one model recognises Finance success, while another recognises Timescale success,… etc. Each model is independently trained on the same data-set and considers all columns (even those which are not explicitly linked to the component being predicted). As an example, we previously discussed how the team soft metrics (morale and communication) influence the rate of commits and bugs which are recorded for a project, so a model trained only on a subset of the project fields may not observe this relationship. Therefore, our design increases the likelihood that the model witnesses the true relationship between the attributes, rather than only being exposed to the fields which appear to be relevant. This element is especially important if the Simulation were to become more complex or real project data were to be used for training, as the innate relationships would become more complex.

Next, we discussed in the design that a Logistic Regression model how would be most suitable for the Risk-Assessment and initial testing demonstrated that such a model produced good results for each component. However, during development, we learned that a Logistic Regression model does not support retraining (also known as partial fitting) so the model could not learn from data provided by users of the website upon project completion. After some research, the next closest solution was deemed to be an SGDClassifier.

### 2.2 - Performance Evaluation & Export



*Figure 2: Classification Report for the Overall Success model and Accuracy scores for each component model (when run on the Test data-set)*

The performance of the model was primarily evaluated using the Classification Report generated by sklearn. This report contains the Precision, Recall and F1-Score for each class (Success-1 or Failure-0). Additionally, the overall accuracy was considered for each model, also provided by sklearn. In testing, we aimed to achieve 75-80% accuracy, although we encountered some difficulty with low recall of the model for Failed projects. This issue was ultimately related to the coverage of the data-set and an imbalance in the distribution of project classes (insufficient numbers of failures).

To export the trained models for further testing and use by other modules, we used *Joblib*, which is claimed to be faster for large models than its alternative, pickle [**SOURCE**: *[https://medium.com/nlplanet/is-it-better-to-save-models-using-joblib-or-pickle-776722b5a095](https://medium.com/nlplanet/is-it-better-to-save-models-using-joblib-or-pickle-776722b5a095].)*[].](https://medium.com/nlplanet/is-it-better-to-save-models-using-joblib-or-pickle-776722b5a095].)

### 2.3 - Avoiding Overfitting

A common issue which plagues many Machine Learning solutions is that of overfitting, which occurs when a model approximates the training dataset too closely and struggles to generalise to unseen data. Unfortunately, the risk of overfitting is increased when a model is trained on many independent input columns **[SOURCE]**. It is essential that this issue is detected and handled appropriately, otherwise the model would be of little use for generating predictions for new projects during the operation of the website.

The first way we monitored for overfitting was through maintaining separate training and test datasets. After each round of training, an internal test file *testBulk.py* evaluated the performance of the model on the test dataset, a completely unseen set of projects. These results acted as a benchmark for the generalised performance of the model. However, performance-validation alone was thought to be insufficient - especially when the test data was produced via the same means as the - so we also introduced a form of Cross-Validation. At a basic level, Cross-Validation splits the training dataset into a fixed number of batches (or “folds”), one of which acts as a test dataset while the model is trained on the rest [**Source**: <https://www.v7labs.com/blog/overfitting>]. However, as we mentioned earlier, our dataset consists of multiple samples per project, so a random choice of samples could have allowed a project to appear in *both* the training and the test fold (a form of leakage). Therefore, we decided to use *GroupKFold* which chooses samples such that the same Project ID cannot appear in both training and test fold. This decision ensures that the results of the Cross-Validation and the claimed accuracy of the trained model is verifiable and we can be sure that the model generalises well.

## Risk-Assessment Predictions

### 3.1 - Confidence Probability

The model we have described in Section 2 is capable of classifying projects into Success or Failure but our system was required to provide the probability of a given project succeeding. As such, rather than using the predictions provided by the model directly, we instead consider the confidence probabilities of the model, which represent the likelihood that the project falls into the Failure class or Success class respectively. These probabilities are obtained using the model’s *predict\_proba()* function. However, this change alone is not sufficient, since the predictions provided by the model are not perfect and our testing found that generally around 20% of the time the model will give the wrong prediction class for a project. With this detail in mind, we applied a Conditional Probability formula, using Bayes’s Law to determine the total probability of success, with respect to the model’s accuracy. The formula we applied can be summarised as follows:

The True-Success-Rate (TSR) represents the probability that the model correctly predicts a success for a project which succeeds (I.e. a True-Positive-Rate for the Success class). The False-Failure-Rate (FFR) represents the probability that the model incorrectly predicts a failure for a project which succeeds (I.e. the False-Positive-Rate for the Failure class). If we denote Success by S and Failure by F (such that TS = True Success and FF = False Failure), then we have the following definitions:

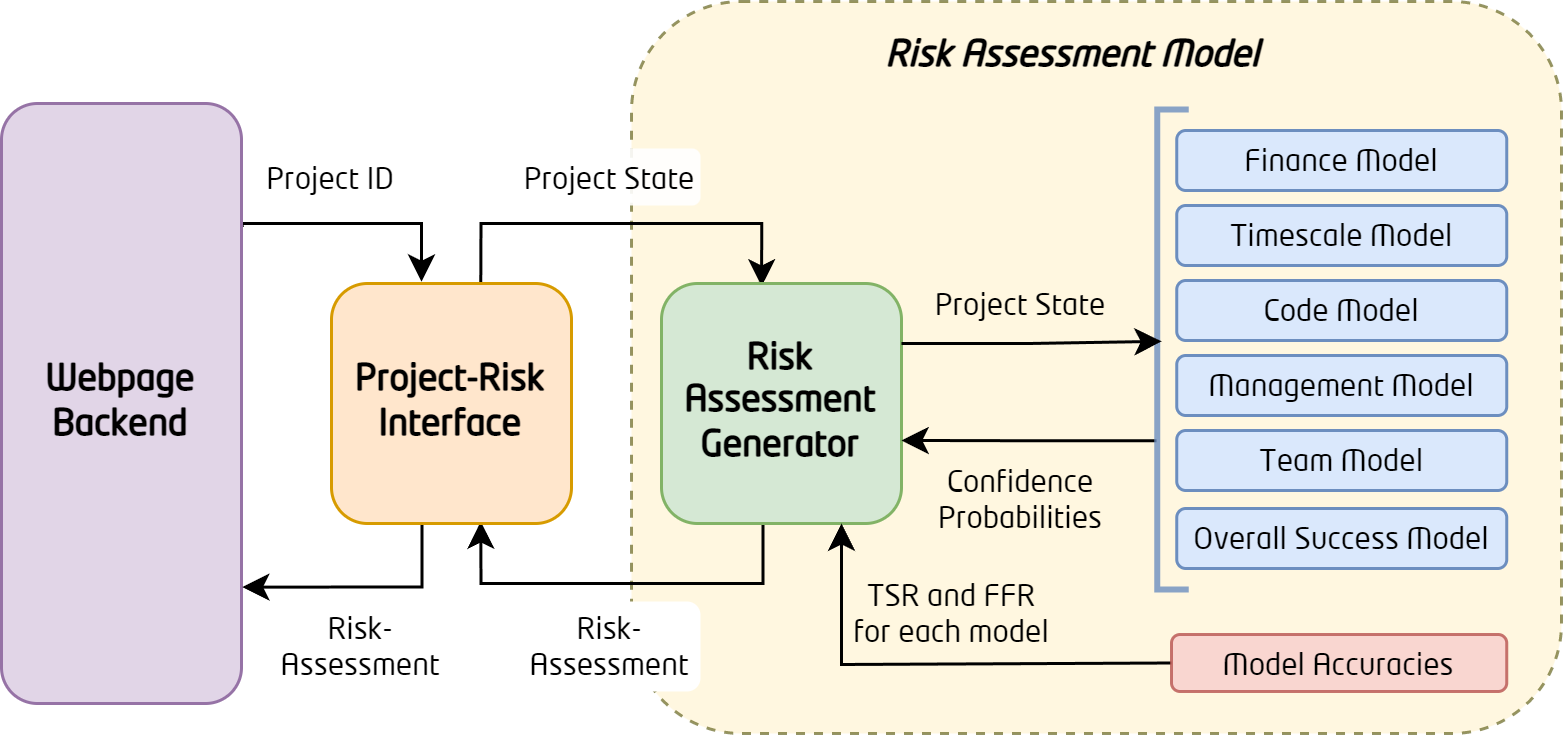
(I.e. out of all projects classified as a Success, how many are actually a Success?)

(I.e. out of all projects classified as a Failure, how many are *not* actually a Failure?)

Moreover, both TSR and FFR are easily calculated using the confusion matrix generated when each sub-model is tested against the independent test-set. These values are then written to text files adjacent to the trained models, and they can be reloaded when the Risk-Assessment is calculated.

### 3.2 - Risk Assessment Generator

In an ideal scenario, the interaction between the main website and the Risk-Assessment and Machine-Learning system should be minimal, and in fact, the Risk-Assessment system has been designed to act as a black-box, obfuscating the details of how the Success probabilities are actually calculated. To be precise, the Risk-Assessment-Generator (RAG) acts as an intermediary step, receiving a project as input and returning the Risk-Assessment object, which contains the probabilities. However, the model requires the project in a specific format with both stored and derived attributes labelled precisely, so we also introduced another intermediary step (known as *ProjectRiskInterface*) which only requires a Project’s database ID in order to return the Risk-Assessment (by making a call to an instance of RAG). Figure 2 shows the exchange of data between components to produce a Risk-Assessment.



*Figure 2: Interaction between Webpage and Risk-Assessment-Model*

## Evolution & Improvements

### 4.1 - Relearning

Need partial-fit;

To avoid overfitting to new data, only limited samples (3) are taken from each completed project

Train once a week, all-together

Success values obtained from final survey, not from evaluation formulae

### 4.2 - **Improvements**

Big Improvements possible in accuracy of data-generation

-> more fields; more complex relationships modelled (heat=Crunch Time)

-> could also continue to simulate projects after completion

Feature Reduction

Model could consider more granularity in assessment

E.g. scale of success (1-5) rather than binary