Mining Software Repository

CS454 AI-Based Software Engineering

Shin Yoo | COINSE@KAIST

Data Mining

(from Wikipedia https://en.wikipedia.org/wiki/Data_mining)

- “The process of extracting and finding patterns in massive data sets”
- Why are we interested in data mining in software engineering? Certain questions cannot be easily answered using analytical frameworks.
  - Is this change likely to introduce a fault? (defect prediction)
  - Who can handle this bug report the best? (bug triage)
  - Who can review this code change? (code review assignment)
- We can try to learn patterns from historical data to answer such questions.

Components of Data Mining Pipeline

- Data sources: Where do you get your data, and why? Your data source should be supported by your problem formulation as well as your hypothesis.
- Feature engineering: What do you exactly extract from your data source? How many features, and why?
- Labels and Ground Truths: What do you train your model with?
- Evaluation: How do you know that your trained model is any good?

Data sources in Software Engineering

Open source, agile, etc —> Abundance of data!

- Version control: commits, diffs, blame, timeline…
- Issue tracker: textual contents, response patterns, severity, triage…
- Code review: PR contents, discussion metadata, reviewer assignments…
- CI: test results, logs…
- AppStore: user reviews
- …and of course the code itself.

Mining Software Repository

…has been rapidly established as a sub-topic in early 2000s

- Classical papers on some of the representative problems:
  - **Defect Prediction**: “Predicting Defects for Eclipse” (https://doi.org/10.1109/PROMISE.2007.10), “A large-scale empirical study of just-in-time quality assurance” (https://doi.org/10.1109/TSE.2012.70)
  - **Bug Triage**: “Who should fix this bug?” (https://doi.org/10.1145/1134285.1134336)
  - **Flaky Test Detection**: “Just-in-time Flaky Test Detection via Abstracted Failure Symptom Matching” (https://doi.org/10.1109/ICSME58944.2024.00078)
  - **Code as Natural Language**: “On the Naturalness of Software” (https://doi.org/10.1145/2902362)
  - …and many others.

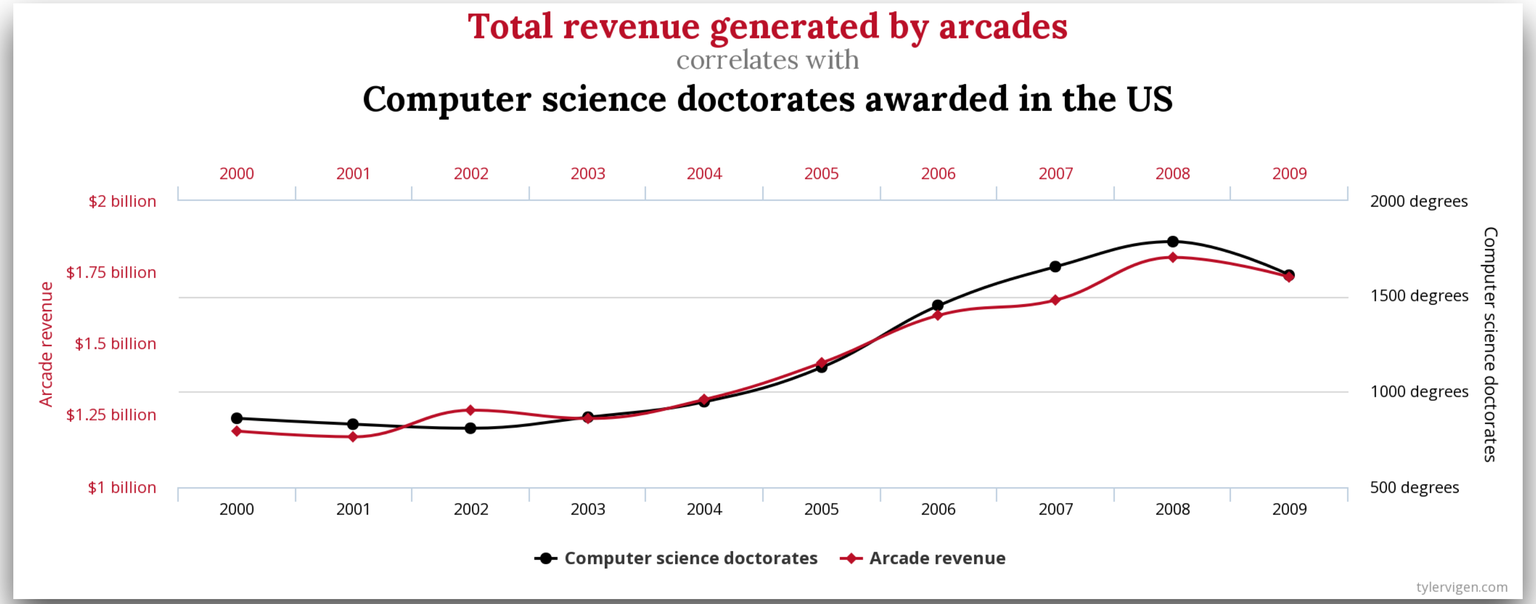


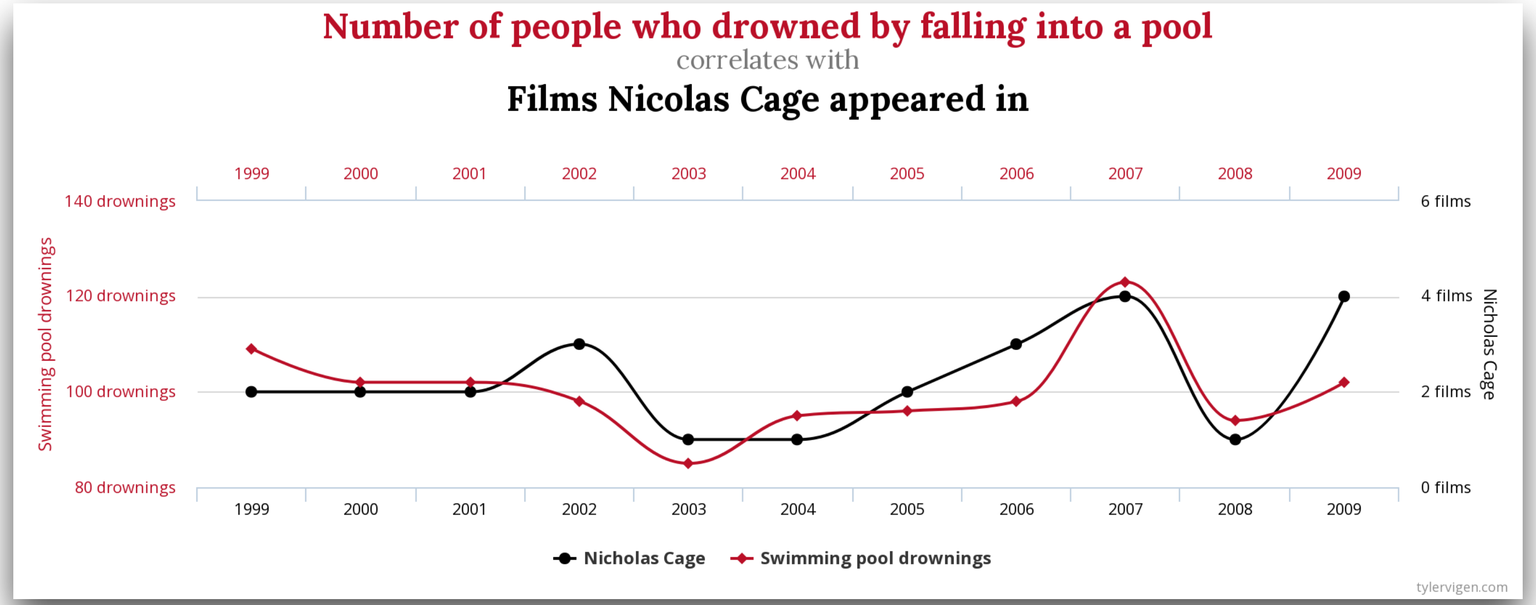
“It is the age of big data! Shouldn’t we collect as much data as possible, then make ML algorithms to figure out secret patterns automatically?”

Uh, no.

Data Dredging

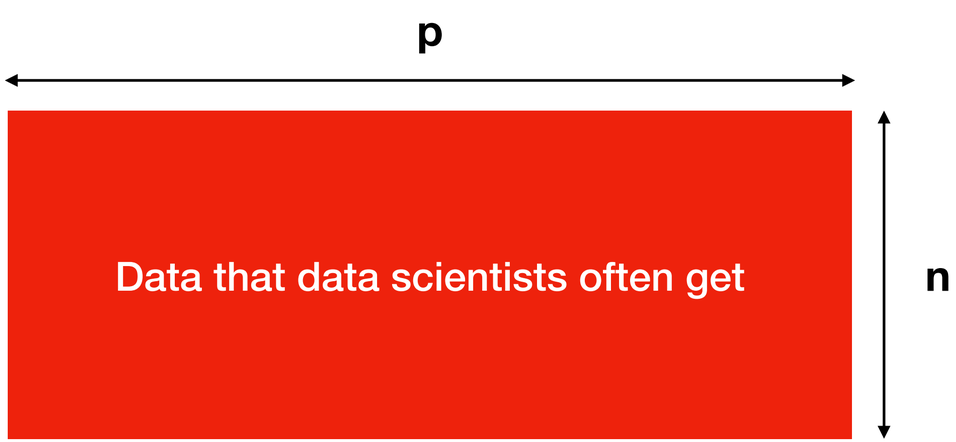
- Misuse of data analysis to find patterns in data that can be presented as statistically significant, when in fact there is no real underlying effect (Wikipedia)
  - Testing/modelling hypotheses suggested by data
  - Bias in the data collection
  - Correlation, not causation





Spurious Correlations (http://tylervigen.com/spurious-correlations)





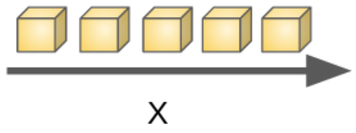
Big *p* small *n* problem!

Curse of Dimensionality

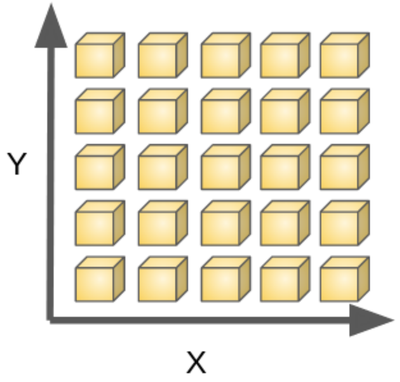
- Various phenomena that arise when analysing and organising data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience (Wikipedia).
- Having more features = having data in higher dimensional space

Data Density

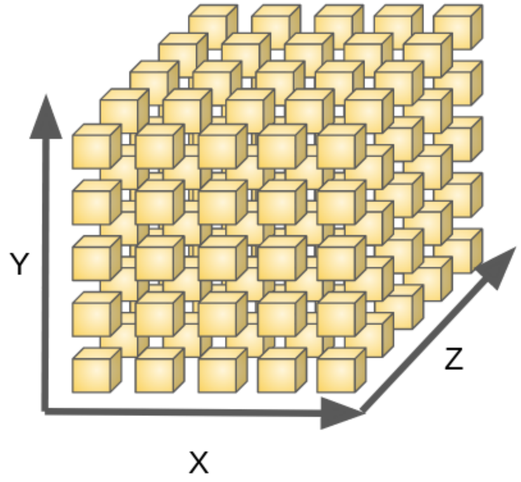
Suppose we take N samples from different dimensions



One cube contains
N/5 samples.



One cube contains
N/25 samples.

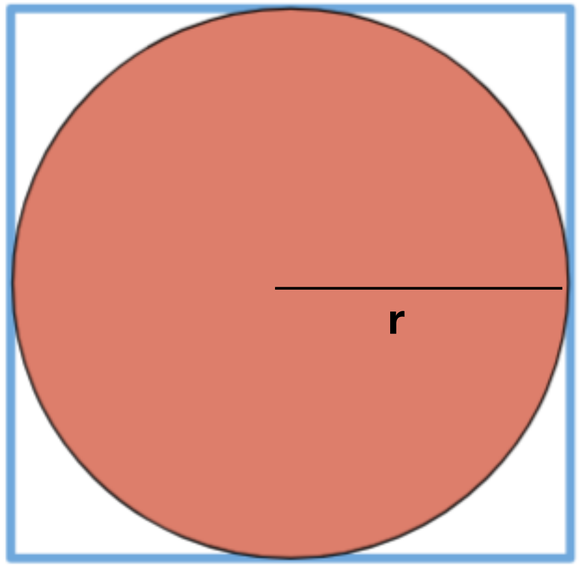


One cube contains
N/125 samples.

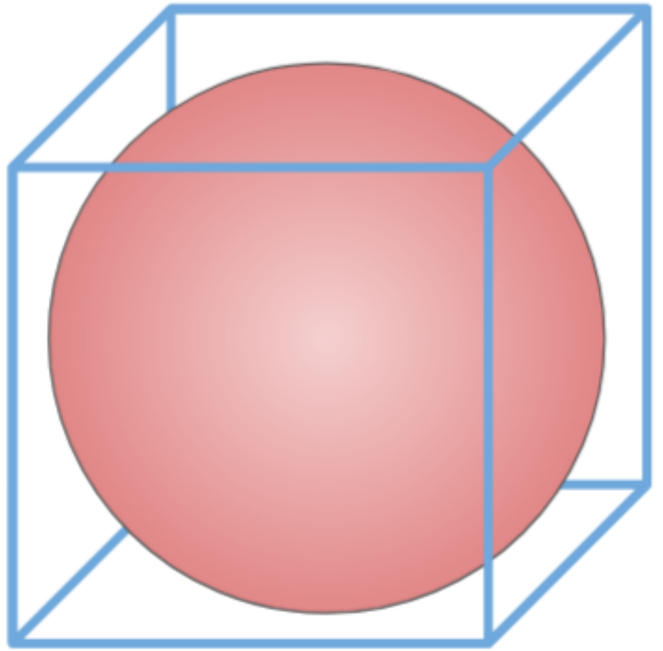
https://medium.freecodecamp.org/the-curse-of-dimensionality-how-we-can-save-big-data-from-itself-d9fa0f872335

“Geometric Insanity”

A



B



Square Area: $(2r)^2$
Circle Area: πr^2
Circle / Square ≃ **78.5%**

Cube Vol.: $(2r)^3$
Sphere Vol.: $4\pi r^3/3$
Circle / Square ≃ **52.4%**

https://medium.freecodecamp.org/the-curse-of-dimensionality-how-we-can-save-big-data-from-itself-d9fa0f872335

“Geometric Insanity”

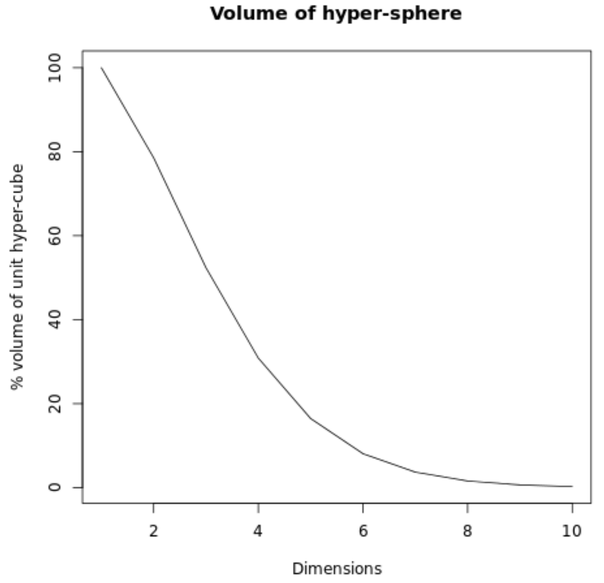
Hypervolume of an n dimension hypercube:

$$(2r)^n$$

Hypervolume of an n dimension hypersphere:

$$\frac{\pi^{n/2} r^n}{\Gamma(\frac{n}{2}+1)}$$

Euler’s Gamma



Hypersphere / Hypercube (%) = ?

$$\frac{\pi^{\frac{n}{2}}}{2^n \Gamma(\frac{n}{2}+1)} \cdot 100$$

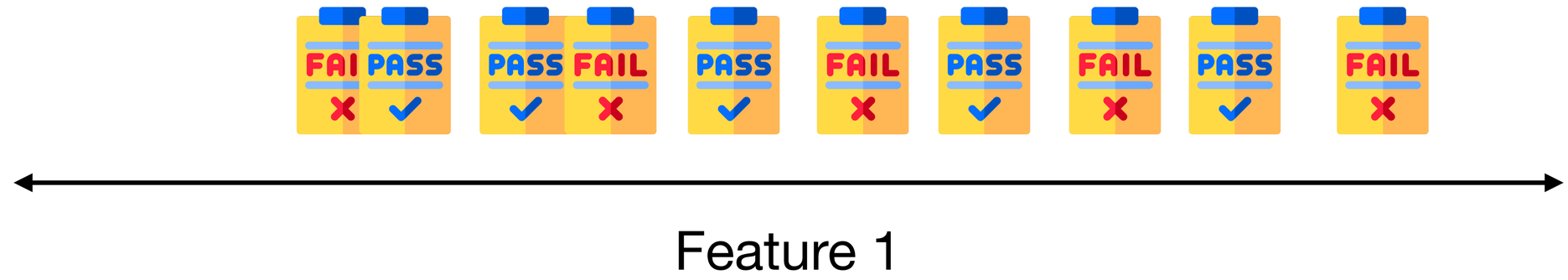
Smaller and smaller chance of two data-points being within the distance of 2r.

Most points are near the “corners” and far away from each other!

Curse of Dimensionality

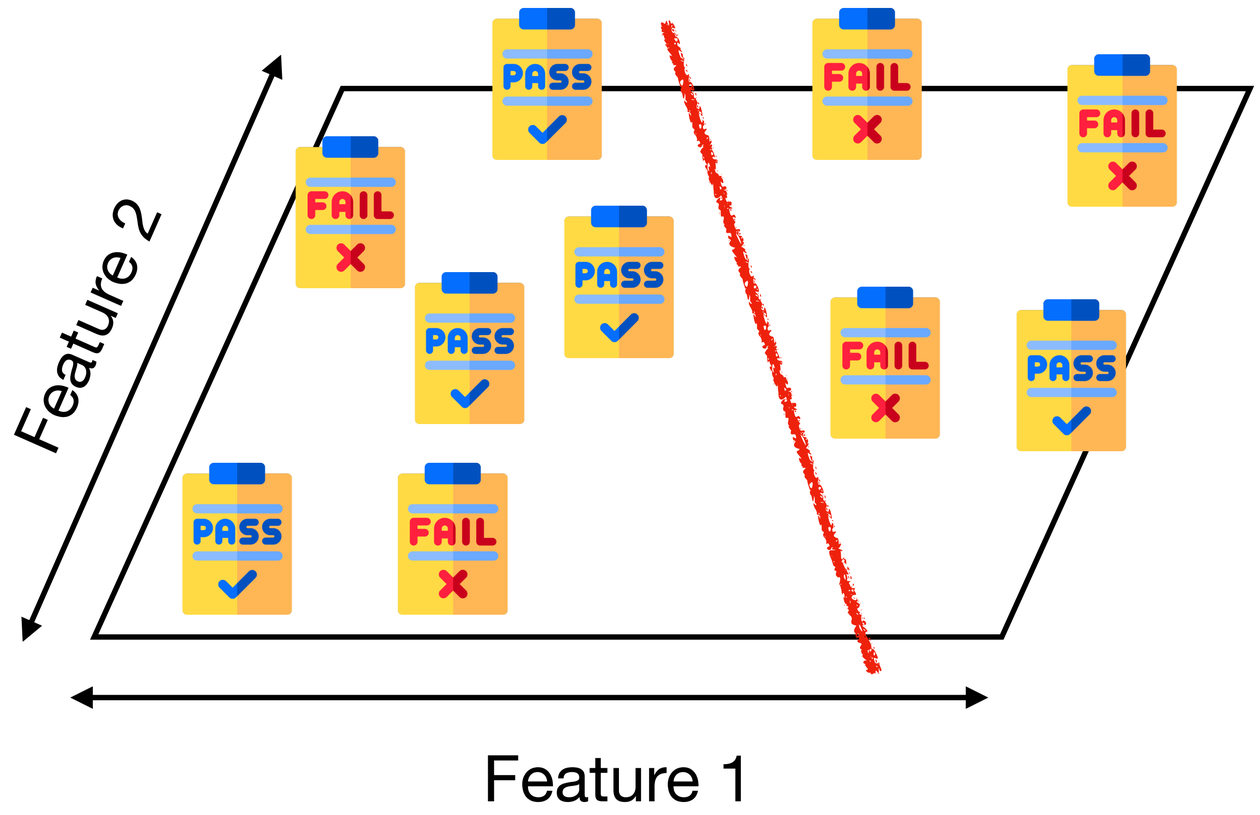
- Intuitively, the higher the dimension is, the lower the data density is.
- Formally, the higher the dimension is, the sparser everything **has to** become (mathematics says so).
- This has a direct and critical impact on machine learning.
- Let’s assume that we have a JIT-DP model.

A 1-feature JIT-DP



Suppose we have 10 samples.
Using a single feature clearly cannot separate pass and fail.

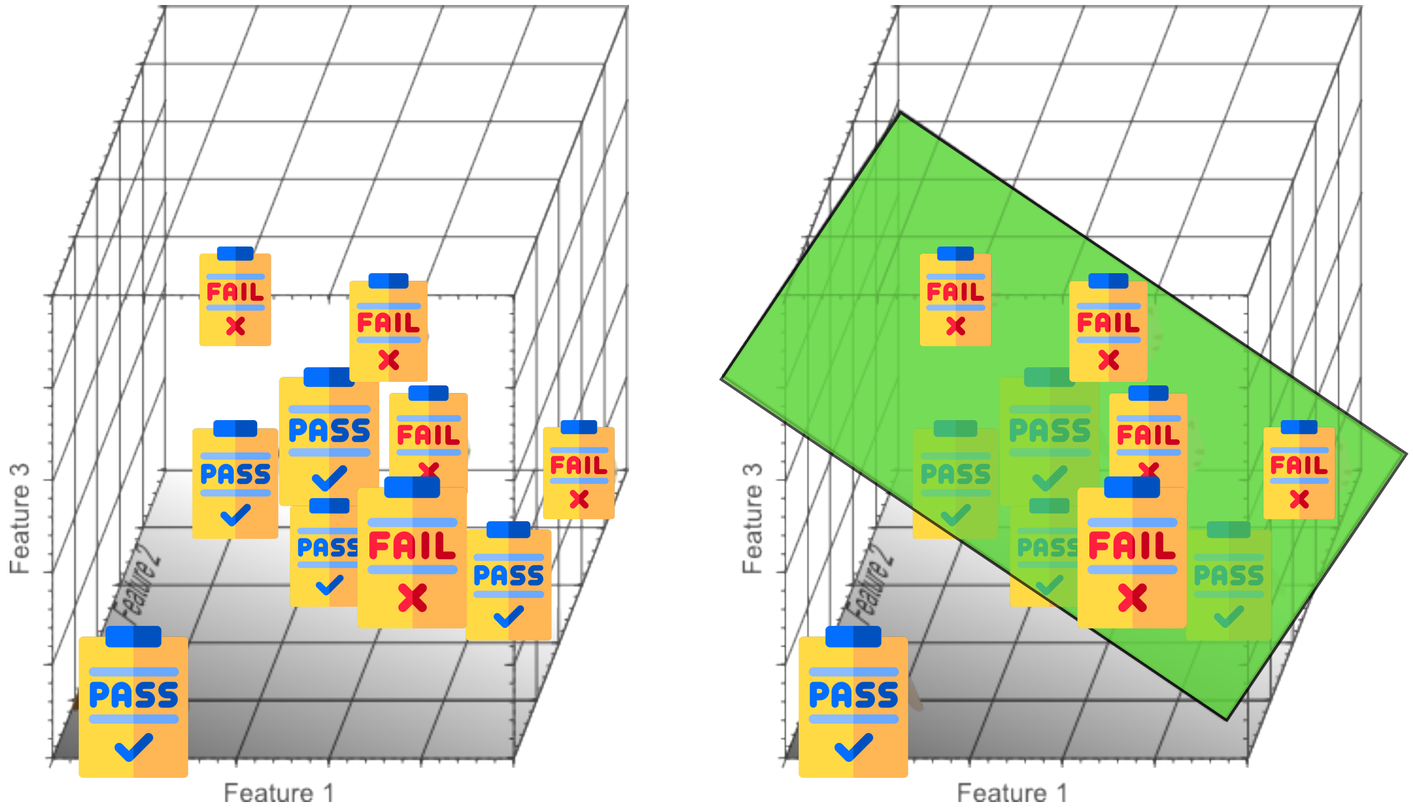
A 2-feature JIT-DP



So we add one more feature.
Slightly better, but no linear separation of two yet.

A 3-feature JIT-DP

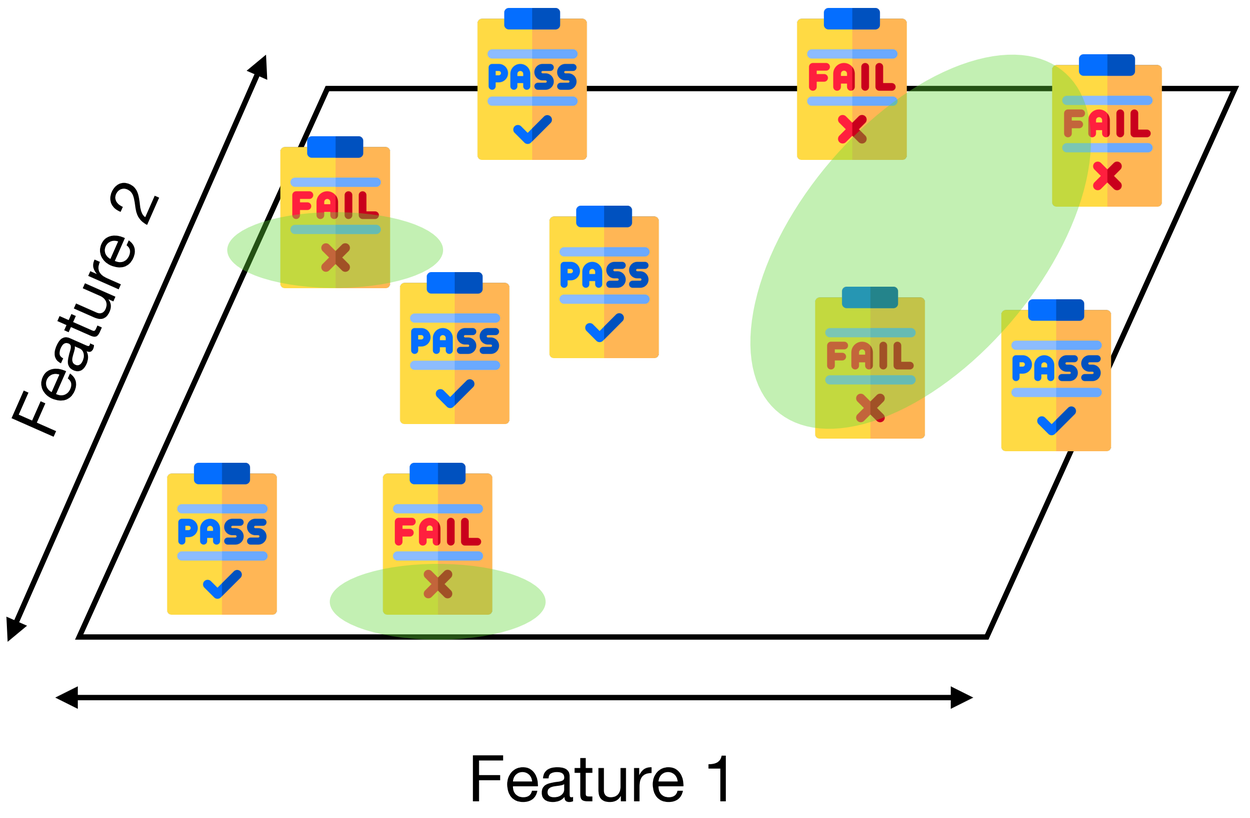
Perfect…?



Linearly separable with the addition of the third feature.
Problem solved?

A 3-feature JIT-DP

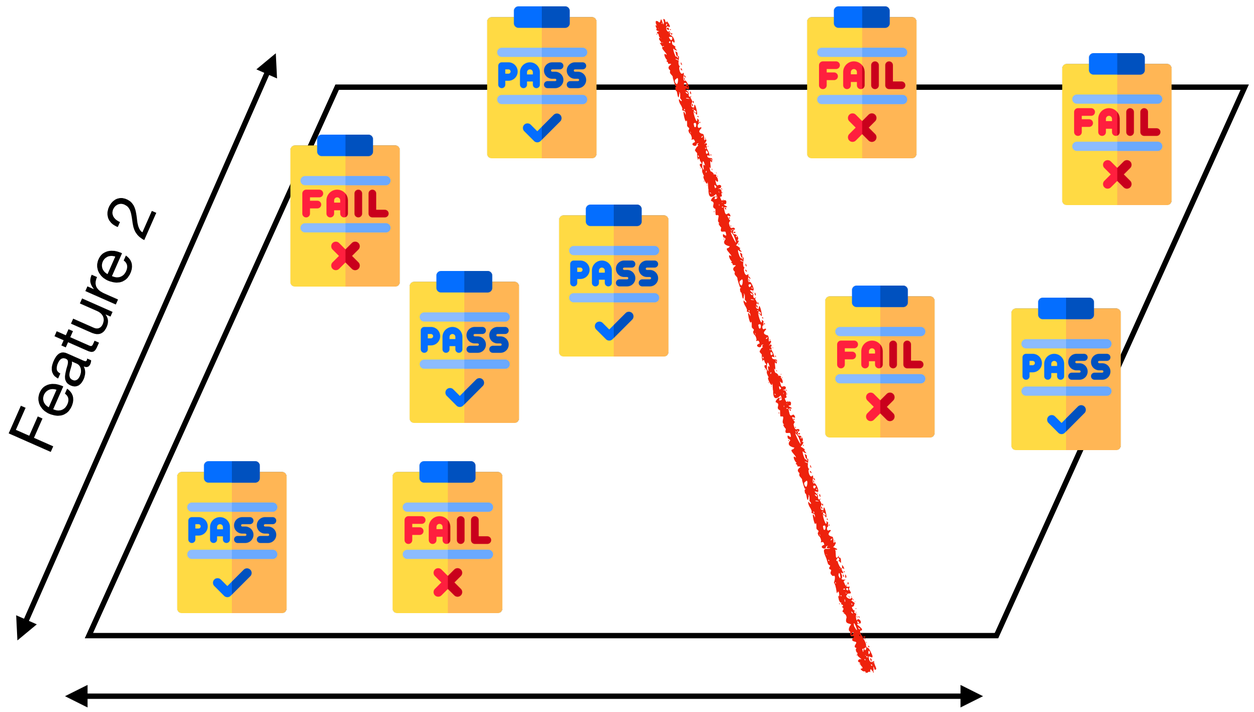
- The sparsity in higher dimension makes it exponentially easier to fit a separable hyperplane.
- However, if we project the result back to a lower dimension, this actually has the effect of using a sophisticated non-linear classifier in the low dimension.
- We are learning the training data, rather than patterns —> overfitting.



When projected back to a lower dimension, it becomes obvious that we learnt the location of training data, rather than any pattern.

A Robust Feature Engineering

- To be more robust against unseen data, a non-perfect low level classifier may be preferable.
- In general, we want a sparse feature set, with each feature definitely contributing unique information. This improves robustness as well as interpretability.



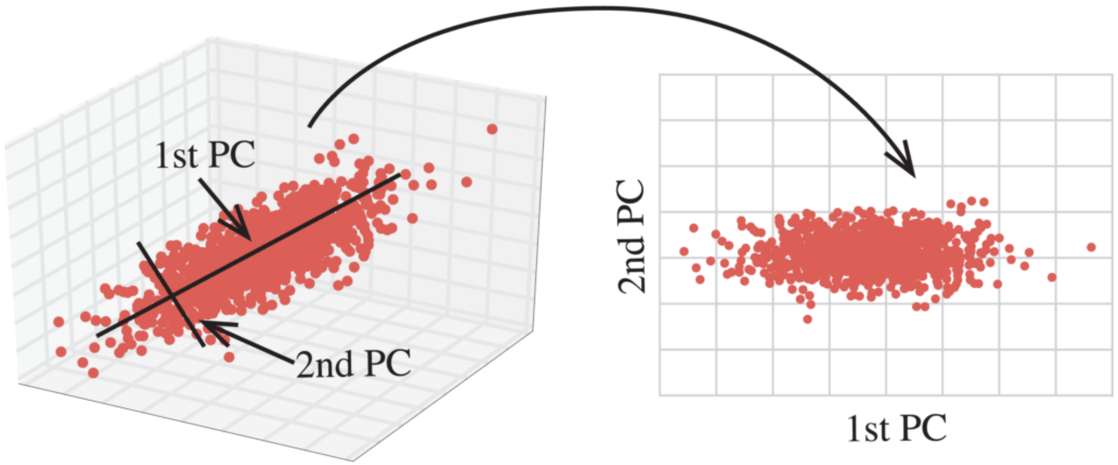
Feature Engineering

Concrete Techniques

- Dimensionality Reduction: project your data to a smaller dimension, thereby reducing the number of features. Examples: PCA
- Embedded Methods: in linear models, your loss function can directly embedded feature weight regularisation (i.e., reduce the weights as much as possible)
- Wrapper Methods: selecting the feature set becomes the outer layer optimisation problem.
- Filter Methods: you filter out features that do not make enough contribution - this depends on quantitative measure of information gain from each feature.

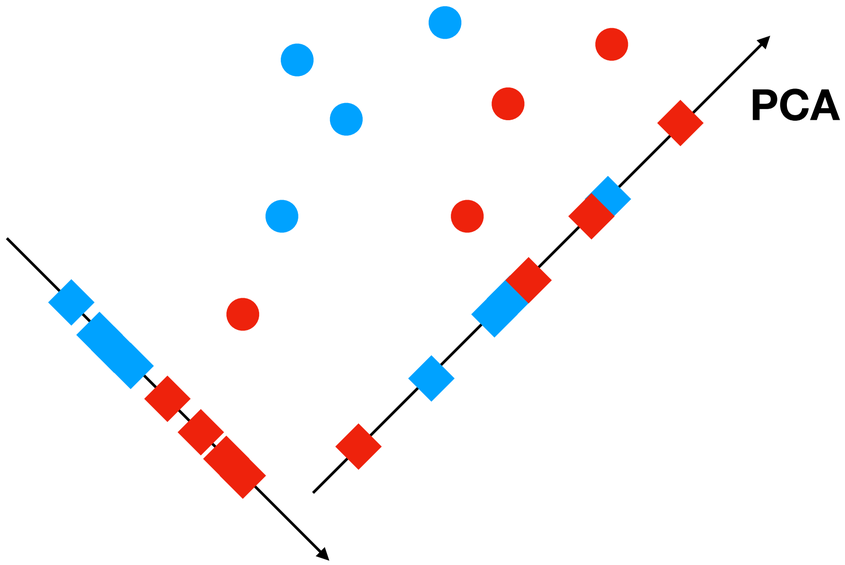
Principal Component Analysis (PCA)

- Defines a set of Principal Components, i.e., directions of different degrees of variance
  - 1st: direction of the greatest variance in the data
  - 2nd: direction of the second greatest variance in the data, perpendicular to the 1st
  - …
- Find n << d components (dimensions), and project every data point to these dimensions



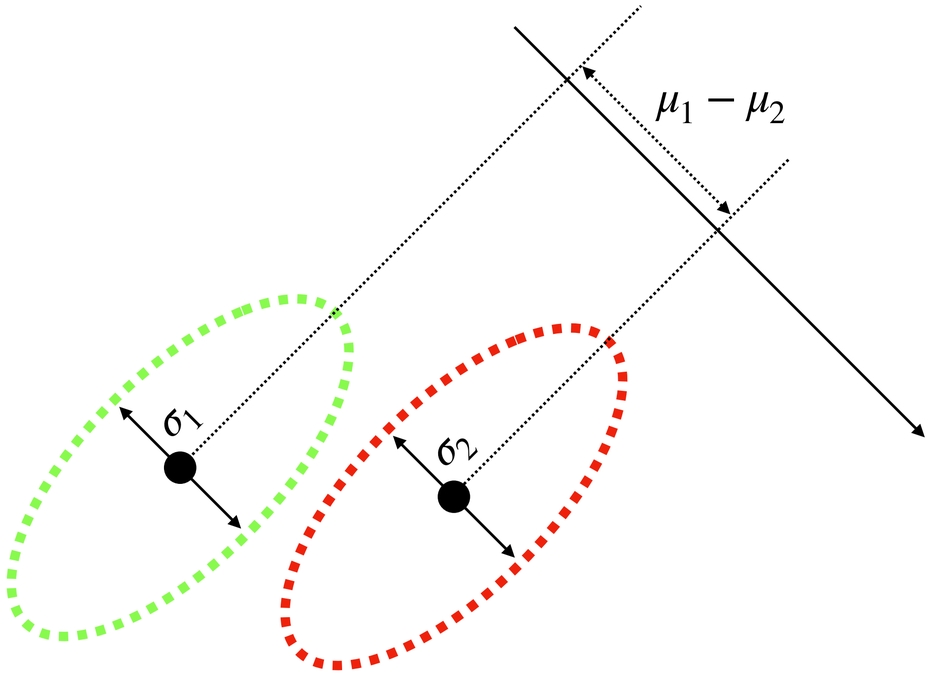
PCA and Classification

- PCA may hurt classifiers badly.
- It only looks at coordinates, and not the class labels.
- Depending on a problem, this may result in a very unfortunate decomposition.



Linear Discriminant Analysis

- LDA: find a dimension that:
  - Maximises separation between means of projected classes, and
  - Minimises variance within projected classes
- LDA is supervised (i.e., requires labels), whereas PCA is unsupervised.



$$\max\left(\frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 + \sigma_2^2}\right)$$

Embedded Methods

i.e., Coefficient Regularisation in Linear Models

- i.e., We want the coefficients in linear models to be as small as possible. This helps:
  - Avoid overfitting: we will not make the current training set have too much impact on the final decision.
  - Model interpretation: we will be able to read relative importance of each features more clearly (linear weights = impact of a unit change, i.e., ± 1, of the corresponding feature)
    - Multicollinearity: correlations between features in regression models

Regularisation

- Given a linear model f, we add regularisation $R(f)$:

$$\min_f \sum_{i=1}^{n} \left(V(f(x_i), y_i) + \lambda R(f) \right)$$

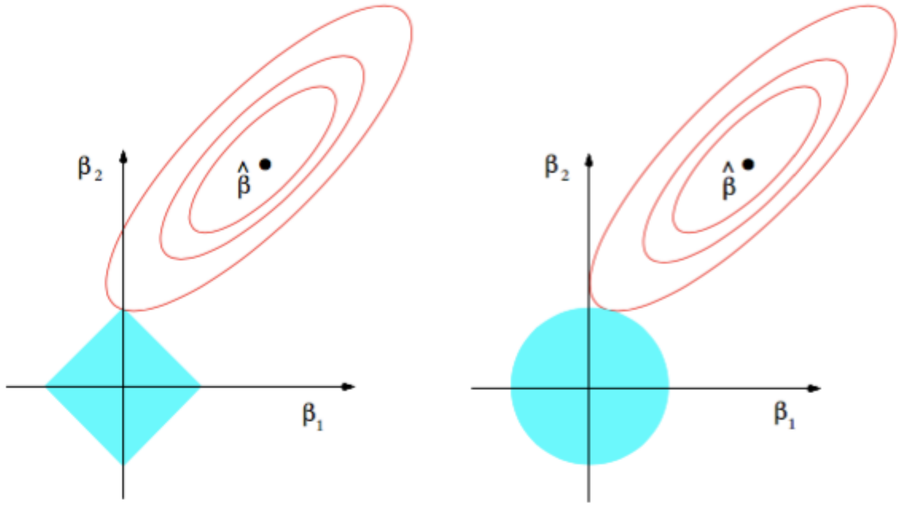
  - L1 (Lasso): $\min \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$
  - L2 (Ridge): $\min \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$

L1 vs. L2

Likely to produce sparse output
Need some tricks

Non-sparse output
Easy to compute (differentiable)

$|\beta_1| + |\beta_2| \leq t$



Contours of unregularised loss

$\beta_1^2 + \beta_2^2 \leq t$

FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

Larger $\lambda =$ **smaller** t

Regularisation Weight

$$\min_f \sum_{i=1}^{n} \left(V(f(x_i), y_i) + \lambda R(f) \right)$$

$\lambda = 0 \ (t \to \infty)$

No regularisation
= becomes OLS
(Ordinary Least Square)

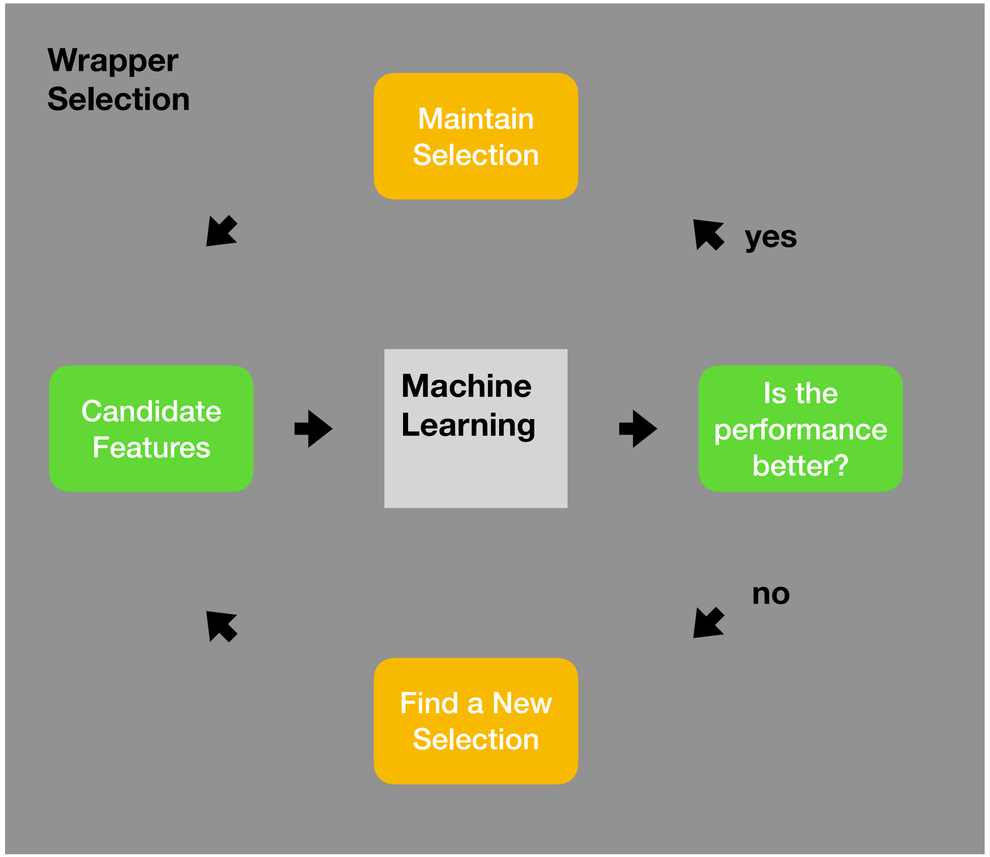
Finding λ:
only possible via
cross-validation

$\lambda = \infty \ (t \to 0)$

All weights become 0
= severe underfitting

Wrapper Methods

Use a metaheuristic to select features - fitness is model performance



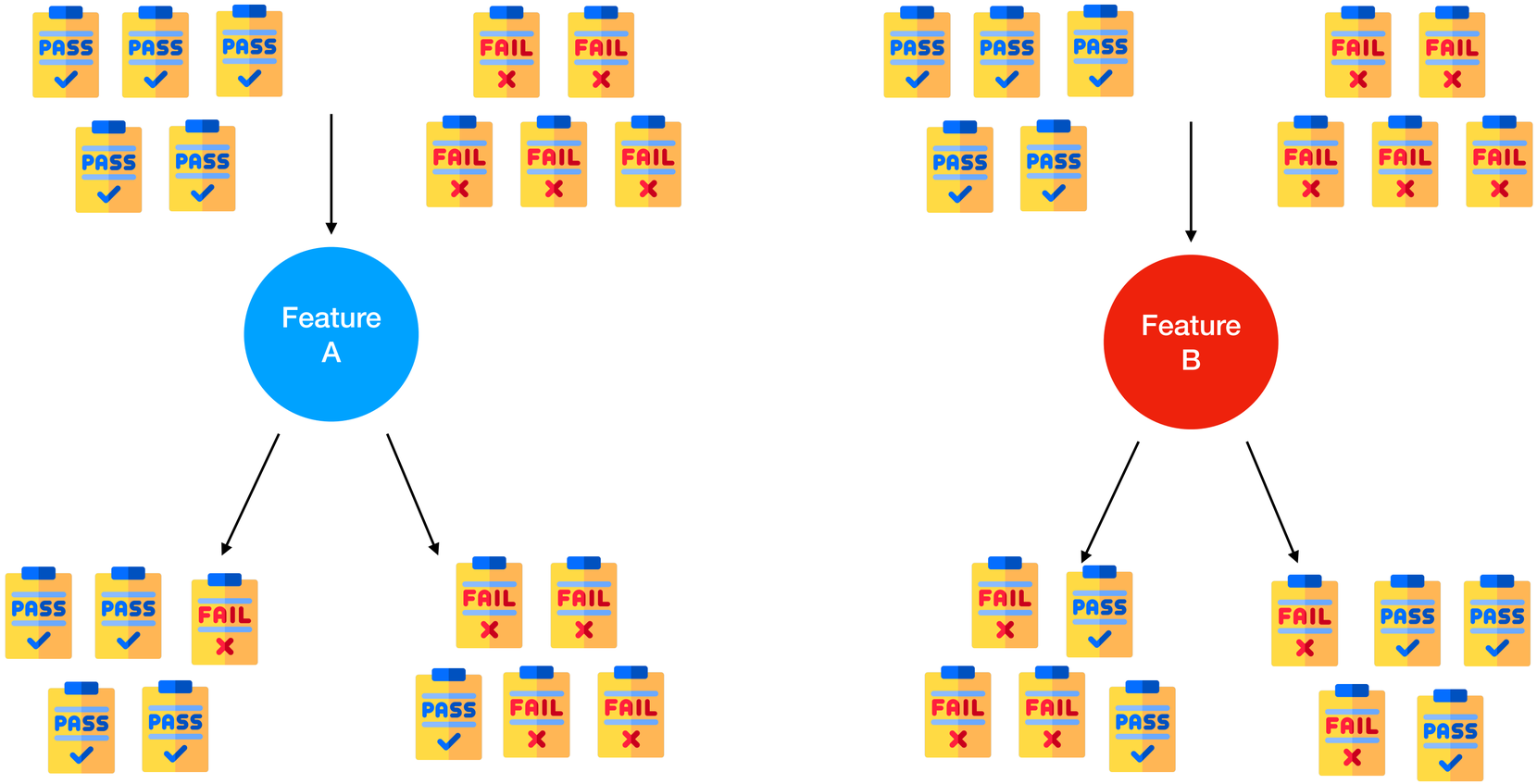
Filtering Methods

…where we filter out features that do not help much.

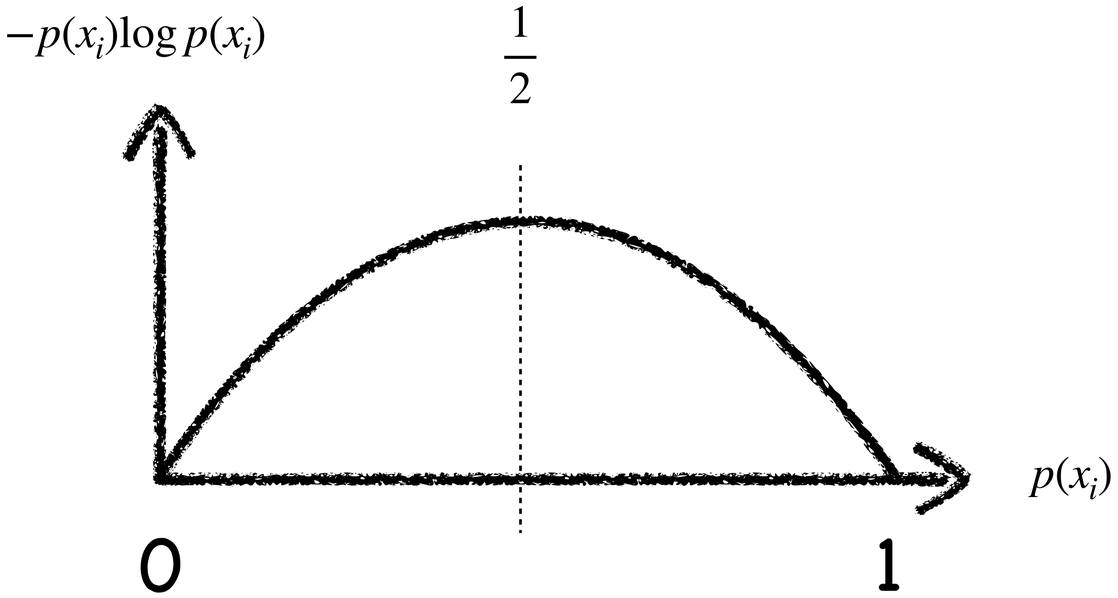
- Q: How do we quantitatively measure the contribution from a feature to a model?
  - A: Information gain(=contribution) is the decrease in class impurity. If a decision based on a feature results in purer class separation, the feature contributes more information.
  - Impurity?

Impurity Measure

Consider these decision tree nodes and their split:



Which child nodes are “purer”?
Which one is the better split?

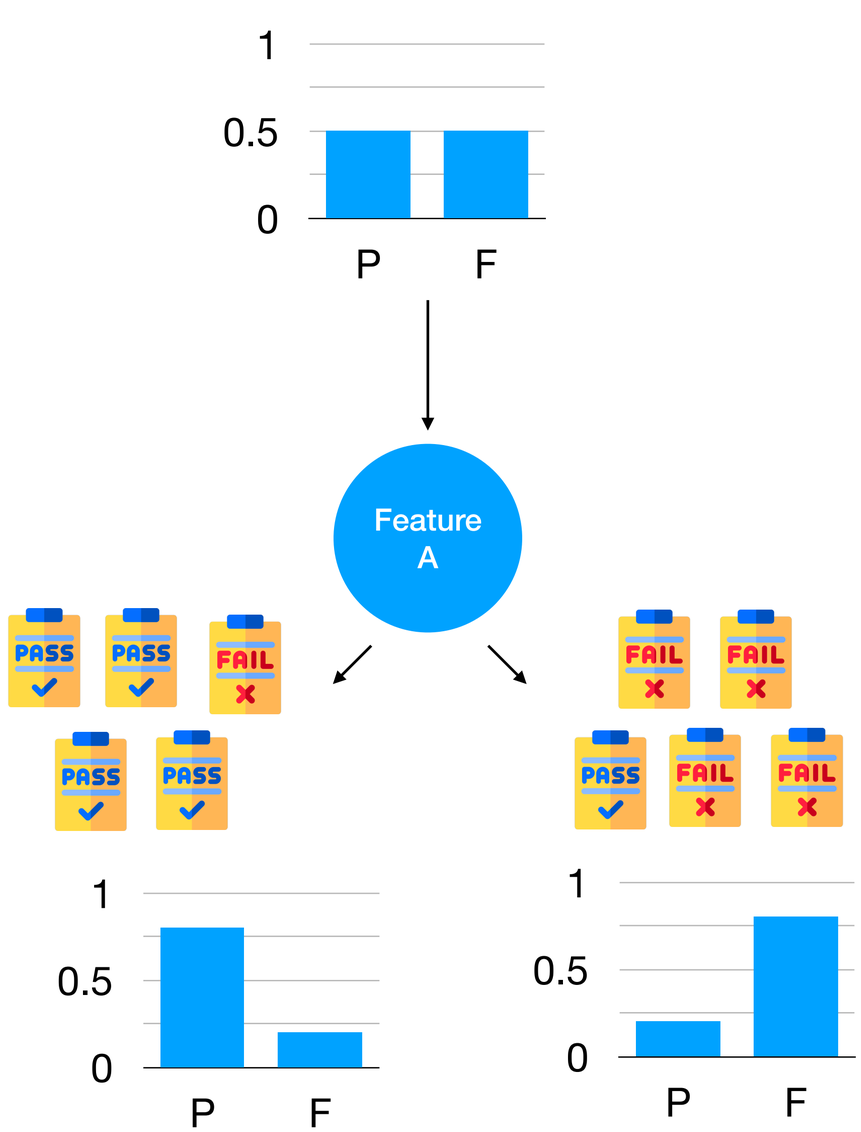


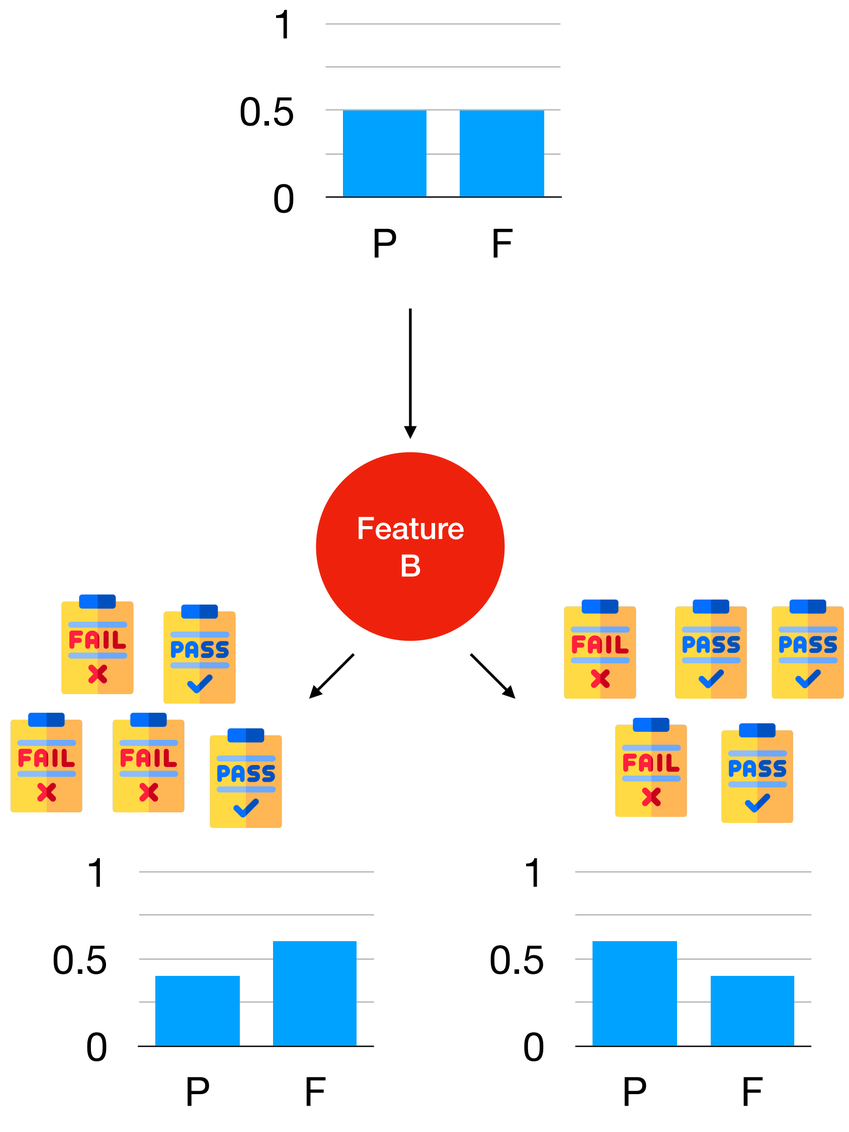
$$H(X) = -\sum_{i=1}^{n} p(x_i)\log p(x_i)$$

To reduce entropy of X is to drive p(xi) to either 0 or 1 for each x_i. The amount of reduction is our information gain.

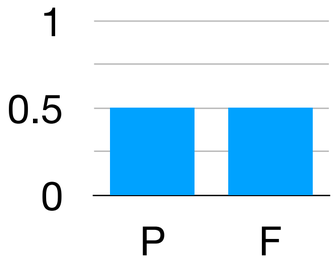
$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \left(\frac{N_j}{N} I(D_j) \right)$$

$$I_H(t) = -\sum_{c \in C} p(c \mid t) \log_2 p(c \mid t)$$





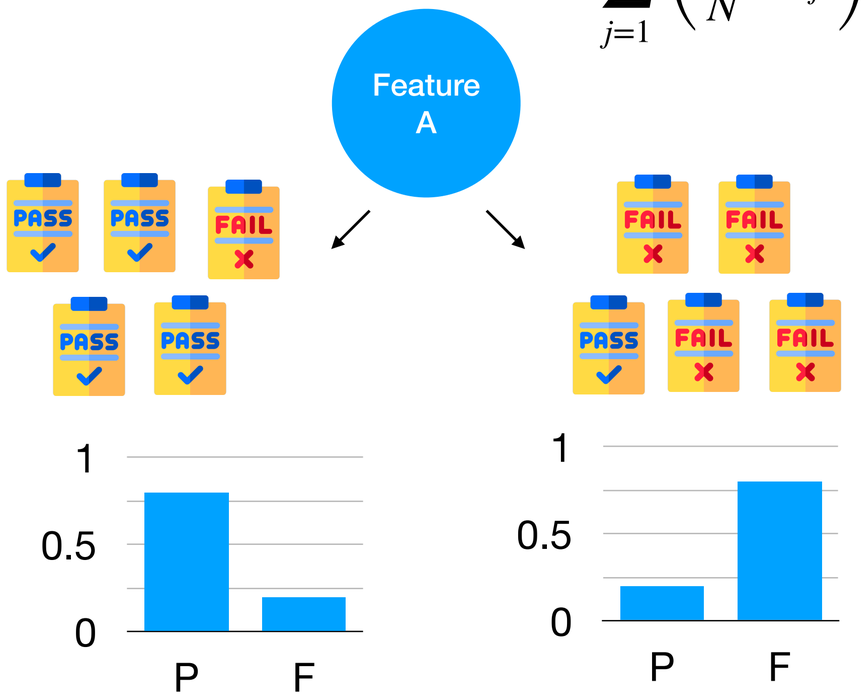
$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \left(\frac{N_j}{N} I(D_j) \right) \qquad I_H(t) = - \sum_{c \in C} p(c \mid t) \log_2 p(c \mid t)$$



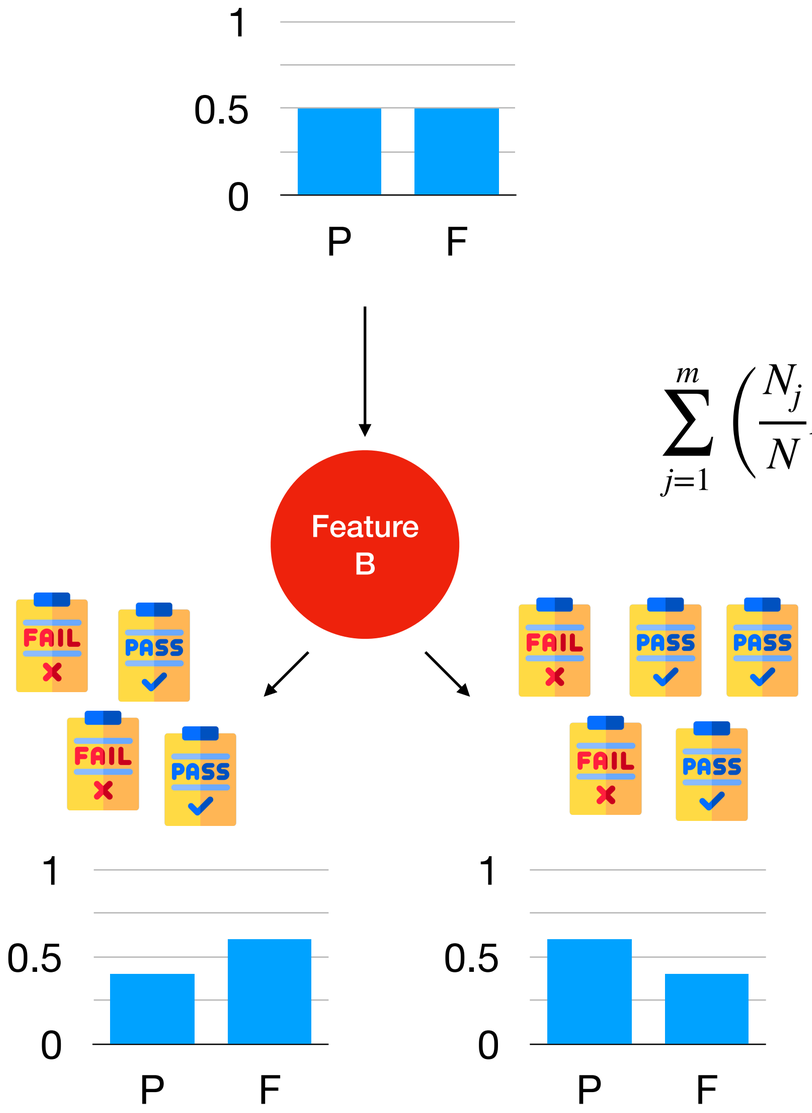
$$I(D_p) = -0.5 \log(0.5) + -0.5 \log(0.5) = 0.301$$

$$\sum_{j=1}^{m} \left(\frac{N_j}{N} I(D_j) \right) = \frac{1}{2}(-0.8 \log(0.8) - 0.2 \log(0.2)) + \frac{1}{2}(-0.2 \log(0.2) - 0.8 \log(0.8)) = 0.217$$

$$IG = 0.301 - 0.217 = 0.084$$



$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \left(\frac{N_j}{N} I(D_j) \right) \qquad I_H(t) = - \sum_{c \in C} p(c \mid t) \log_2 p(c \mid t)$$



$$I(D_p) = -0.5 \log(0.5) + -0.5 \log(0.5) = 0.301$$

$$\sum_{j=1}^{m} \left(\frac{N_j}{N} I(D_j) \right) = \frac{1}{2}(-0.6 \log(0.6) - 0.4 \log(0.4)) + \frac{1}{2}(-0.4 \log(0.4) - 0.6 \log(0.6)) = 0.292$$

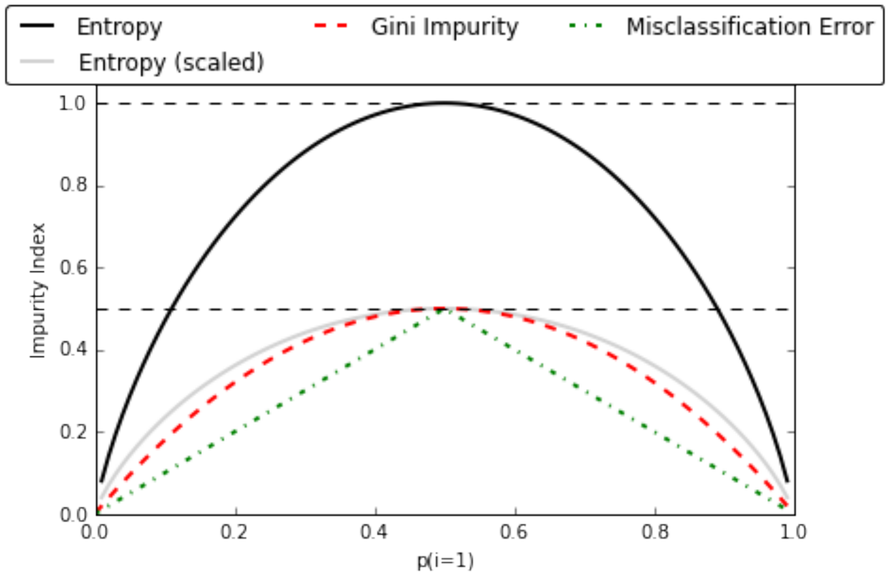
$$IG = 0.301 - 0.292 = 0.009$$

Gini Impurity

$$I_G(t) = \sum_{c \in C} p(c \mid t)(1 - p(c \mid t))$$

$$p(c \mid t) + \sum_{k \neq c} p(k \mid t) = 1 \longrightarrow$$

The *purer* the instances under t are, the smaller this becomes (because one of the possibility gets closer to 0).



Filtering using Decision Trees

- Train a decision tree model: you can usually specify which criterion to use (gini or entropy)
- Once training is finished, implementation usually provide normalised feature importance according to the used criterion
- Use the prioritised feature list to make a selection.

```
rf = RandomForestRegressor()
rf.fit(X, Y)
print "Features sorted by their score:"
print sorted(zip(map(lambda x: round(x, 4), rf.feature_importances_), names),
             reverse=True)
```

Labels and Ground Truths

Not as easy as you expect. Examples for DP:

- Misclassified in real life: not all bug reports are linked to real bugs.
- Implicit biases: bugs and fixes clearly marked in issue tracking systems are rare, and therefore introduce biases (https://doi.org/10.1145/1595696.1595716).
- SZZ Noise: for DP, GT labelling means connecting a commit and a bug. This can be noisy (https://doi.org/10.1109/ICSE43902.2021.00049)
- Label Timing: a commit that was labelled to be safe at some point may turn out to be buggy later (https://doi.org/10.1145/3338906.3338941)

Evaluation

Are we cheating, over-claiming, or making up solutions?

- Time-travel Leakage: if you do cross-validation, you may train with the future and evaluate with the past. Features like “# of times this file was buggy” will be affected.
- Class Imbalance: heavily imbalanced datasets can skew your evaluation (if only 1% commit is buggy, you can achieve 99% accuracy by saying that everything is not buggy). However, attempts to overcome imbalance can also affect the outcome (https://doi.org/10.1109/TSE.2018.2876537)
- Metric Choices: simple unsupervised models may outperform more complicated supervised models, if you account for the cost of collecting GT labels and training the model (https://doi.org/10.1145/2950290.2950353)

SZZ: Śilwerski, Zimmermann, and Zeller 2005

Where the DP started :)

- How do we label a commit as a Bug Inducing Commit (BIC)?
- First, identify a Bug-Fixing Commit (BFC) using issue trackers.
- Then, identify lines modified by the BFC.
- Using blame, find the most recent commit that previously added, or touched, those lines in BFC - these are the BICs.
- Noise: matchings in issue trackers, conflict resolution in VCS, refactoring commits…

Extensions to SZZ

- Various filtering heuristics have been proposed, e.g., filters for refactoring changes.
- Note that the original SZZ links BIC to failure only through BFC: there is no dynamic analysis involved.
- Fonte (https://doi.org/10.1109/ICSE48619.2023.00059) incorporates test results, so that BIC information can be available at debugging time
  - First, perform FL in the current buggy version (we know this because test failed).
  - Second, using blame, move the suspiciousness score to past versions. While doing this, filter out commits that only make stylistic changes (using AST analysis). The transferred suspiciousness decays over time (multiplier $(1-\lambda)^{depth(e,c)}$, where $depth(e,c)$ is historical depth of commit c regarding program element e.
  - Now, aggregate suspiciousness score in each version.

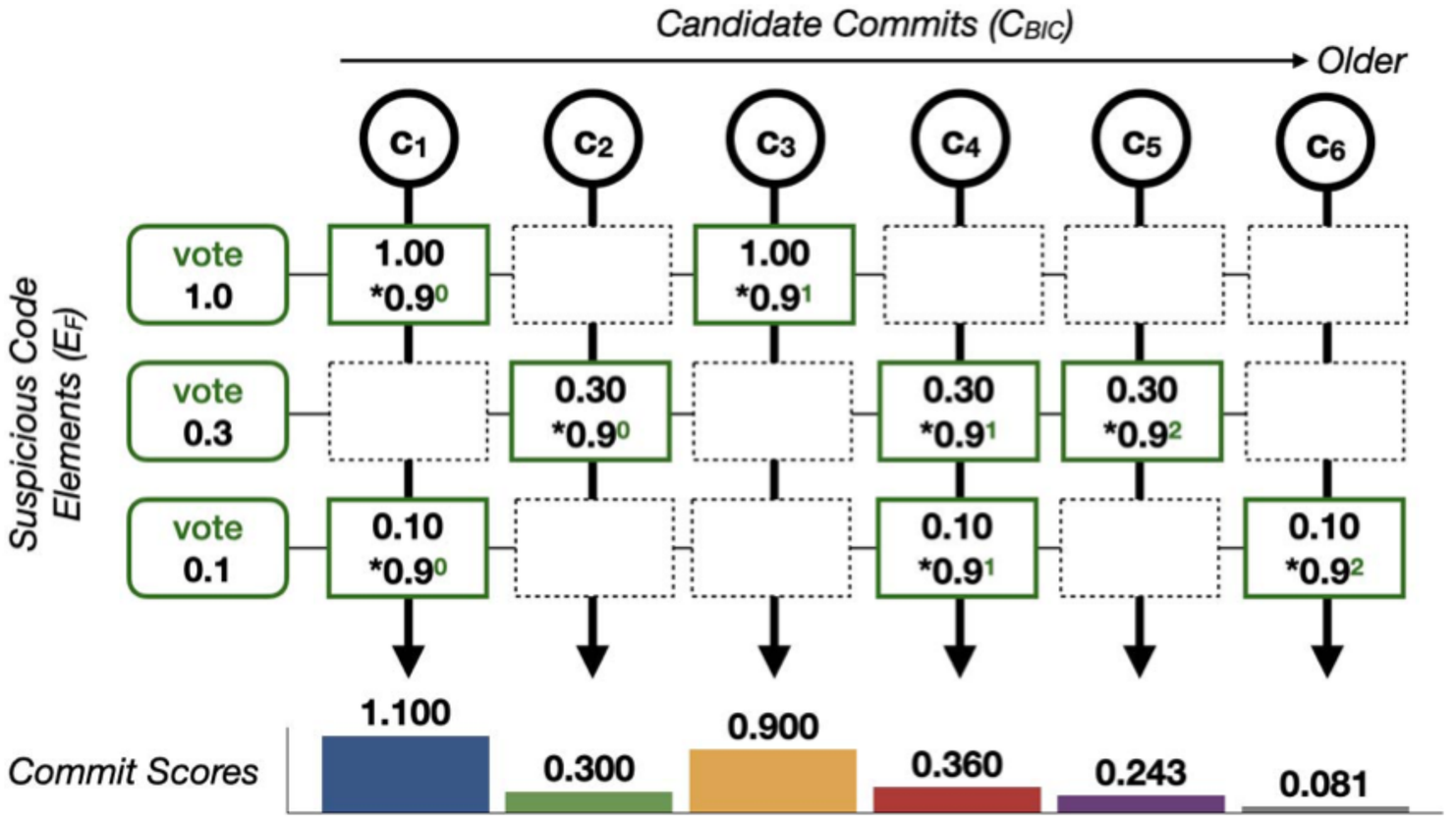


Fig. 3. Example of computing the commit scores when $\lambda = 0.1$

Fonte: Finding Bug Inducing Commits from Failures (https://doi.org/10.1109/ICSE48619.2023.00059)

Agentic BIC Identification

- SZZ-Agent (https://doi.org/10.1145/3832783.3834337): agent performs bisection over timeline, trying to evaluate each commit.
- AgenticSZZ (https://arxiv.org/abs/2602.02934): build temporal knowledge graph that connects commits and modified files based on the types of changes, and uses the graph to identify the BIC.
- AgentSZZ (https://arxiv.org/abs/2604.02665v1): the agent is given git related tools so that it can autonomously investigate the timeline.

Questions and Connections

- Why should we prefer a sparse feature set with lower accuracy to a very rich feature set that results in a higher accuracy?
- What are some of the primary reasons that SZZ-based labels for defect prediction can be noisy?
- When debugging, knowing the responsible BIC would help. Why is SZZ not appropriate for this, despite being the most famous BIC identification algorithm?

Summary

- SE is increasingly more data-driven (even before LLM). Questions that are very hard to answer analytically can be answered based on statistics.
- Be very skeptical of spurious correlations and other trivial patterns: check your hypothesis thoroughly and engineer a good feature set.
- You should prefer sparse feature set - they are more robust, less prone to overfit, and easier to interpret.
- Always put great efforts to make your GT label as clear as possible. Sometimes you cannot avoid human validation.