論文調査報告：
Machine Learning:
The High-Interest Credit Card of Technical Debt

国立情報学研究所　石川　冬樹
f-ishikawa@nii.ac.jp / @fyufyu
http://research.nii.ac.jp/~f-ishikawa/

自己紹介

- 石川冬樹（35）
  - 国立情報学研究所 コンテンツ科学研究系 准教授
  - 何かしらの「ソフトウェアがすべきこと・してくれること」の研究
    - 仕様，サービスAPI/SLA，法律とか
    - 分析・検証・推論・変換・・・
  - VDMとか形式手法を中心に，産業界の皆さんに伝えたり一緒に追求したりしている人（トップエスイー，日科技連）

f-ishikawa@nii.ac.jp / @fyufyu
http://research.nii.ac.jp/~f-ishikawa/

今回紹介する論文

- Machine Learning: The High-Interest Credit Card of Technical Debt
  - 直訳すると

  機械学習：技術的負債の高利クレジットカード

  At: SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)

  By: D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young (Google, Inc)

  - ほとんどGoogleの経験・解法を報告している感じ

Technical Debt

- 6/4 14時 Wikipediaより引用

技術的負債

技術的負債（英: **Technical debt**）とは、行き当たりばったりなソフトウェアアーキテクチャと、余裕のないソフトウェア開発が引き起こす結果のことを指す新しい比喩である。「設計上の負債(design debt)」とも言う。

1992年ウォード・カニンガムが、技術的な複雑さと債務の比較を経験報告で初めて行った。

> 最初のコードを出荷することは、借金をしに行くことと同じである。小さな負債は、代価を得て即座に書き直す機会を得るまでの開発を加速する。危険なのは、借金が返済されなかった場合である。品質の良くないコードを使い続けることは借金の利息としてとらえることができる。技術部門は欠陥のある実装や、不完全なオブジェクト指向などによる借金を目の前にして、立ち尽くす羽目になる[1]。

Joshua Kerievsky は彼の影響ある著作の一つ*Refactoring to Patterns*において、彼が「設計上の借金」と呼ぶアーキテクチャの手抜きによるコストに関して類似した議論を示した[2]。

開発の中で先送りされるのは、文書化、テストコードの記述、ソースコード中の積み残し(TODO)項目の解決やコンパイラの警告、静的コード解析ツールの解析結果への対応などである。他にも、技術的負債の例として、組織で共有されない知識や、複雑すぎて変更が難しいコードなどがある。

1. 概要

- スピードを優先した結果抱える「負債」
  - 必ずしも悪いこととは限らない
  - 「清算」：リファクタリング，ユニットテストのカバレッジ向上，不要なコードの削除，依存関係の削減，APIの制限，ドキュメントの向上など
- 機械学習の便利なパッケージ
  - 従来のソフトウェアと同様の「負債」と同様の対応

に加えて

  - システムレベルでの隠れた「負債」

⇨ 本論文では避けるべきリスク要因・パターンを議論

2. 境界の浸食

- 従来：境界による保守性の向上
  - カプセル化・モジュール化
  - 隔離された変更
  - 不変条件・整合性条件の明示
- 機械学習
  - そもそも望ましい挙動が，外部データへの依存なしにソフトウェアロジックだけではうまく実装できないために使っている
  - ⇨ 不変条件を，気まぐれなデータから分離する術はほとんどない

2.1. 絡み合い

- 機械学習は結局複数のデータ源を混ぜ合わせるので，絡み合いを産み独立した改良を不可能に
- n個のデータがあるとする
  - うち１個のデータの分布が変われば，他のデータの重要度・重み・利用法が変わる
  - 新しいデータが加わっても，データが１つなくなっても同じ
- ハイパーパラメーターについても同じ
  - 訓練のためのサンプリング方法，収束の閾値など

⇨ 予測の結果が少しあるいは劇的に変わる

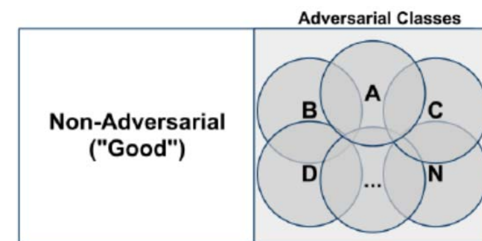
CACE: Changing Anything Changes Everything

2.1. 絡み合い

- 考えられる対策（1）： 各モデルを孤立させてアンサンブル

  ※ 複数の学習手法を組み合わせる手法

  - 対象問題でそもそも有用な場合や，孤立によるモジュール化の利益がよほど大きい場合
  - 例：Googleでの「よくない」広告の検出では，様々な種類のマイナーな広告を扱うため，アンサンブルがそもそも有用だった

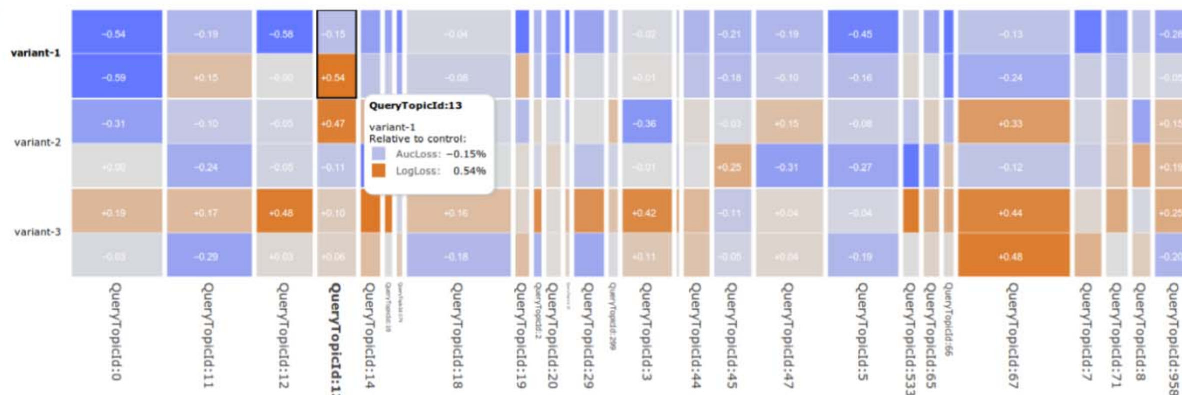


Sculley et al., Detecting Adversarial Advertisements in the Wild, KDD 2011

  - コストは大きくスケールしないかも・また各モデル内では同じ問題が残る

2.1. 絡み合い

- 考えられる対策（2）： 多次元，様々なスライシングでの可視化ができると有用



McMahan et al., Ad Click Prediction: a View from the Trenches, KDD 2013

- 考えられる対策（3）： 予測精度の変化がトレーニングにおける目的関数においてコストとして扱われるようにより高度な正則化を用いる

Lavoie et al., History Dependent Domain Adaptation, Domain Adaptation Workshop 2011

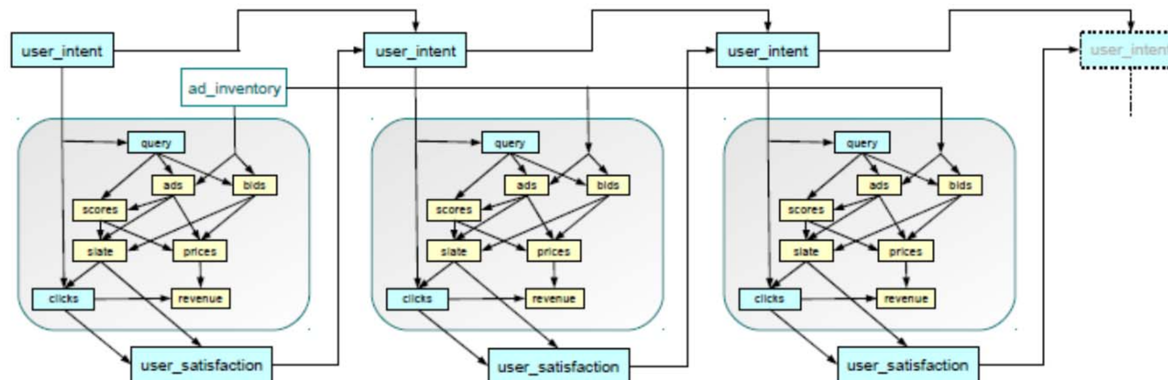
2.1. 絡み合い

- いずれにしても
  - 機械学習では先天的に存在する問題
  - Ver. 1.0のリリースより保守の方がはるかに難しい

⇨ Ver. 1.0の納期への圧力の中でも，どの程度重視するか注意深く考慮すべき

2.2. 隠れたフィードバックループ

- 実世界から学習するシステムは，通常フィードバックループに含まれる
  - 例：ニュース見出しのクリック率予測
    - ユーザのクリックが訓練に使われる
    - 一方でユーザのクリックは過去の予測に依存する
  - ➡難しいが自然と追求される課題ではある



Bottou et al., Counterfactual reasoning and learning systems: the example of computational advertising, JMLR 2013

2.2. 隠れたフィードバックループ

- 隠れたフィードバックループ
  - 例：先週１週間のユーザのクリック数x_{week}を学習の入力に利用
  - 予測モデルを改良してクリック数の増加を見込んでも，少なくとも１週間が経過してx_{week}が適応するまでは表面には現れないかも
  - 新しいデータに応じてモデルを更新したとしても，x_{week}の扱いに対する変化はさらに遅く現れるかも
  - ➡ 簡単な改良でも，素早い実験では確認できず，改良の効果を分析することは難しくコストがかかる
- ➡ フィードバックループは注意深く探し，可能な限り取り除くべき

2.3. 未申告の消費者

- 機械学習の出力は，直接，あるいはログデータなどを通し，他のシステムの入力になりうる
- ➡ 未申告の消費者を産む
  - （通常のソフトウェアでも）予期しない密結合につながり，影響分析が困難に，すると変更が困難に
  - 例：各見出しの大きさを適切に決める部品の入力に，クリック数予測を利用
  - ➡ 新たなフィードバックループにより，簡単に全見出しが大きくなり続けるようなことが起きうる
- ➡ 何かしらの障壁がないと，さっとやってしまいがちで検出も困難

3. コード依存性よりデータ依存性

- 従来のソフトウェアでコード依存性に起因する問題がデータ依存性に応じて生じる
  - コード依存性のように分析手法・ツール（静的解析，リンクグラフ等）が使われていない

3.1. 不安定なデータへの依存性

- 一部の入力は不安定（時間経過で変わる）
  - 他の学習システムからの出力や，単語のトピック分類など参照データの暗黙的な変化
  - 依存するシステムになされる明示的な変更
- 対応策：バージョン付けされたコピー
  - 利用するデータは「凍結」する
  - 新しいデータは吟味の後に適用する
  - ただし，データの陳腐化などそれ自体のコストもあり，また複数バージョン管理が技術的な負債につながる可能性もある

3.2. 不要なデータへの依存性

- コードにおけるほぼ不要なパッケージのように，精度にほぼ貢献しない入力データがありうる

➡ 不必要に変更に対して脆弱に

  - 初期に導入されたが，以後他の入力が追加された結果，冗長になってしまっている
  - データが「一式」として取り急ぎ導入されてしまう
  - ちょっとした精度向上に心惹かれがちだが，複雑さの向上が高い場合がある

➡ 定期的にデータの評価をして手入れをすること，それがもたらす長期的な利益が把握されているような文化を創りあげること

3.3. データ依存性の静的解析

- コードについてはコンパイラやビルダをはじめとしてよく用いられている
- ツールの補助なしでは難しい
  - 大企業では，全体を把握できる人はいない
  - 例：辞書データの利用者

⇨ データのアノテーションが有用
  - Deprecated，プラットフォーム依存性，ドメイン依存性などをアノテーションに付加
  - Deprecatedに変わった際にアラートを出すなど自動で様々な管理が可能に

McMahan et al., Ad Click Prediction: a View from the Trenches, KDD 2013

3.4. 補正の連鎖

- ある問題Aに対するモデルaがあるとき，ちょっと違う問題A'を解きたい
- ついaを入力とし補正を行うようなモデルa'を学習を通して作りたくなる
  - 全く別のチームにより行え，補正モデルは小さい
- 依存関係が生まれ，aに対する精度向上の改良すらシステム全体では不利益を産むことすら
  - 補正モデルa'をさらに補正するような連鎖があるとなおさらで，どこにも手を付けられない状態に
- 最初のモデルaを様々なユースケースに対応し補正を自身で行えるように増強する方がよい

4. システムレベルのスパゲッティ

- システムレベルのアンチパターン

4.1. 接着コードパターン

95%以上！

- システムのコードの大半が，パッケージへの入出力をつなぐ・整理するためのものとなりがち
- パッケージ固有の特性に特化してしまい，しばしば解空間の構築方法を作り込んでしまう
  - パッケージは1つのシステムでいろ多くの問題を解こうとするが，実際は1つの大きな問題に様々な解法を試しうまい設計を検討したい
  - パッケージはアルゴリズム変更を容易にするが，実際は問題空間の構築方法をリファクタリングしたい
- 自分たちで再実装した方が，テストしやすさ，保守性，問題に応じた設計がよくなる

4.2. パイプラインジャングル

- 接着コードの一種として新しい入力が増えるために，データの抽出，結合，サンプリング処理の「ジャングル」になる
  - 結合テスト，誤りを見つけるのが大変
    ※ Paxosなど分散処理の側面も含む

➡ 白紙の状態からデータ収集全体を考えて再設計することが結果としては速い

※ 接着コードやパイプラインジャングルは，研究部門と開発部門の過度な分離が原因になっていることも：「象牙の塔」から出てくるパッケージが完全ブラックボックスになってしまう（Googleでは融合チーム）

Spector et al., Google's hybrid approach to research, CACM 2012

4.3. 使われない実験コード

- 他のアルゴリズムや調整を試す誘惑のために，接着コード・パイプラインジャングルが悪化
  - この手のdead codeは積み重なると大きな負債
    - 例： 45分で4億ドル以上の損失 http://www.sec.gov/News/PressRelease/Detail/PressRelease/1370539879795

⇨ 定期的に検証し取り除くことが有用

⇨ もっと踏み込むと，実験コードが複数のモジュールにからまることなく分離されるよう，APIが設計されていることが望ましい

Googleでは何万行も削除しAPI再設計，よかった

4.4. 設定に関する負債

- 大きなシステムには大量の設定があり，機械学習自体のコードより大きくなりがち，その割に抽象化やユニットテストは軽視されがち
  - このデータはこの期間は誤っているので使わない
  - 実験時のデータDは実環境ではデータD'とD''に代替
  - 機能Zを有効にするときはメモリ上限を増やすべき
  - 機能Qは機能Rより先に立ち上げる必要がある
  - ・・・

⇒ 不変条件アサーションの注意深い設定が重要
コピペが多いのでdiff・可視化も重要
コードの変更と同等にレビューなど行うべき

5. 外の世界の変化

- 機械学習は外の世界と直接やり取りするのが魅力的
- だが外の世界は滅多に安定しない

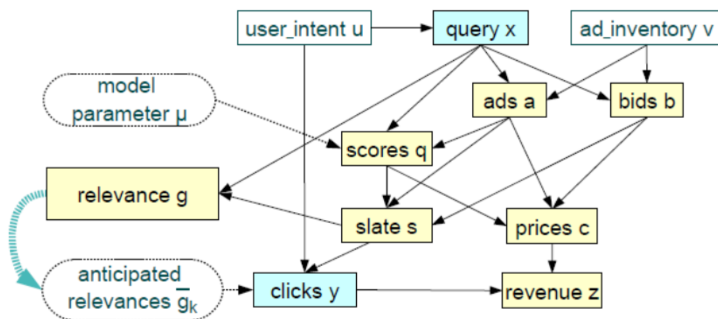
5.1. 動的なシステムでの固定閾値

- しばしば閾値の設定が必要だが手動になりがち
  - Spamかどうかの判定
- ⇨ 妥当性確認用データからの学習が有用

5.2. 相関が相関でなくなるとき

- 相関する特徴量の影響を識別することは難しい
  - よく共起する2つの特徴量が，実は片方からの因果関係しかないとしてもその相関に頼ってもいいかな？
  - シンプトンのパラドックス，交絡などよく知られた難しさ
- 共起がなくなると，予測の振る舞いが大きく変わってしまう

➡ 因果関係の詳細分析手法もある



Bottou et al., Counterfactual reasoning and learning systems: the example of computational advertising, JMLR 2013

5.3. 監視とテスト

- ユニットテスト，end-to-endのテストは重要だが，変化する世界と向き合うシステムでは，リアルタイムでのライブ監視が欠かせない
- ➡ 問題は何を監視するか？
- 予測の偏り： 予測された値の分布と観察された値の分布を比べるくらいでも非常に有用
  - 様々な次元で偏りをスライシングすると分析に役立つし自動アラートもできる
- 行動の制限：健全性検査として，行動に制限を設けることも有用
  - 人の介入を促す，ただ過剰発火しないように

まとめ

- 機械学習システムで，ときには驚く形で技術的負債を産む領域を紹介
  - 機械学習がダメだとか，これらは完全に取り除かなければならない，ということではない
  - 短期的には気にしないで速く行動した方がよいかもしれないが，長期的にはすぐに管理不能になることは認識しなければならない
- エンジニアも研究者もともに！