修 士 論 文

ウェブブラウザにおける既読コンテンツの検出・表示手法に関する研究

A Study of Methods for Detecting and Showing Already-read Contents in Web Browsers

2017 年　2 月 3 日

指導教員　　相田　仁　教授

東京大学大学院
工学系研究科 融合情報学コース

37-146456　　早 乙 女　高 大

目次

図目次

表目次

第1章

序論

1.1 はじめに

インターネット等の情報技術は21世紀に入って急速に普及した．その中でも検索エンジンはインターネットにおいて情報を検索する際には不可欠な技術となっている．しかしながら，検索エンジンを実装しようとする際の課題として，重複した内容のページや検索エンジンスパム等が指摘されている [1]．検索エンジンによる検索結果を利用する際には上位に表示されたウェブサイトから順次閲覧することになるが，その最中，ユーザーは既に読んだものとは異なる内容を期待している．そのような状況で重複した内容のページが検索結果に現れると，ユーザーは同一内容のページを何度も見ることとなり，利便性が損なわれる．

一例として，あるエラーメッセージが表示され，解決策を求めているという状況を考える．図1.1はGoogleにおいて

firefoxで接続がリセットされる

というキーワードで検索を行った結果の上位10件(左半分が1位から5位，右半分が6位から10位)を示しているが，そのうち3番目，および10番目の項目は，6番目の項目“OKWave”を転載したものである．(10番目の“楽天 みんなで解決！ Q&A”のサイト内には“Powered by OKWave”の表記がある．また，これら3項目のURLには文字列“7053807”が共通して含まれており，同一のデータベースを利用しているものと考えられる．)

そのような重複コンテンツへの対処方法として，検索エンジン側におけるアルゴリズムの修正，ウェブブラウザ上の拡張機能による表示の調整などが挙げられる．

本研究では，ウェブブラウザ上の拡張機能により現在のページの表示内容の記録，および既読コンテンツの検出・表示を行う．本研究の最終的な目標は，既読コンテンツを自動的に検出・表示し，ユーザーがこのような重複コンテンツを読み飛ばすことを補助することにある．

このシステムの実現可能性を検証すべく，ウェブブラウザの拡張機能，および既読コンテンツの記録・検出用サーバからなるシステムを構築し，実際に記録・検出を行い，その性能を評価する．

図 1.1: 複数のウェブサイトに転載されたコンテンツの例

1.2　本論文の構成

本論文は 7 つの章で構成されている．第 1 章では序論を述べた．第 2 章では関連する研究と，利用した基礎的な技術について説明する．第 3 章では，提案するシステムの概要について説明する．第 4 章では，提案したシステムの実装と詳細な仕様について説明する．第 5 章では，複数の実験を行い，目的とするシステムや本研究の最終的な目標の実現可能性について検討を行う．第 6 章では，実験の結果について考察する．第 7 章では，本研究のまとめと今後の課題について述べる．

第2章

背景

本章では，研究の背景として，検索エンジンにおいて現れる重複コンテンツの例，関連する研究，および利用する基礎的な技術について説明する．

2.1 背景と関連研究

2.1.1 重複した内容

重複するコンテンツの例として，次のものが挙げられる．

引用・転載 ユーザーに対するサービスの一環として，他のウェブサイトの内容を転載する場合がある．例えば，オンライン百科事典サイト「Wikipedia」の内容は「goo Wikipedia 記事検索」や「Weblio 辞書」などに転載されている．前者はポータルサイト「goo」のサービスの一環としての転載である．また，後者は特定の語句に対し複数の辞書を横断検索できるサービスであり，その一環として Wikipedia の該当する記事の一部を転載している．

複数サイト間の情報共有 (コンテンツシンジケーション) ユーザーに対する情報の露出を増やすために，複数のサイトが同一の内容を掲載する場合がある．一例として，冒頭にも挙げたナレッジコミュニティ・Q&A 投稿サイト「OKWave」とそのパートナーサイトがある．OKWave では「パートナーサイト」として他の企業が Q&A サイトを手軽に構築できるシステムの貸出を行っているが，構築されたパートナーサイトの Q&A データベースは元の OKWave の Q&A データベースと共有されているため，同一の質問とその回答が検索結果に複数現れることにつながる．

複数の情報源のまとめ (キュレーション) 複数の情報源を 1 ページにまとめて読みやすくするサイトやサービスは多数存在する．Livedoor News や Excite News は複数の報道機関からのニュースを 1 サイトにまとめるサービスである．また，NAVER まとめは様々な情報をユーザーが独自に収集して組み合わせ，一つの「まとめ」として公開できるサービスである．複数の情報源とは限らないものの，電子掲示板や SNS の投稿を情報源とし，それら取捨選択して公開している「まとめブログ」は無数に存在する．

また，重複はしていないが，似通ったコンテンツが多くなる例としては，同一の主題や同一の製品について説明している複数のページがある．例えば，ある事件が発生した際に

その事件について報じる複数のニュースサイト，ある製品に対する複数のレビュー記事，ある医薬品についてその作用・用法を説明する複数のサイトなどの状況が考えられる．

2.1.2　検索エンジンスパム (Spamdexing)

ロボット型検索エンジンにおいては，独自のアルゴリズムにより検索結果の表示順序の決定や無関係なサイトの除外などが行われる．しかしながら，このアルゴリズムの欠陥を突くことにより，実際には検索キーワードと無関係なページを検索結果の上位に表示させる行為が行われる場合がある．このような行為は検索エンジンスパム (search engine spam, spamdexing) と呼ばれる．この名称はスパムメール (迷惑メール, email spam) からの類推である．

[1] では，ほとんどのユーザーは検索エンジンの結果のうち最初のページ (通常は上位 10 件) のみを調べるため，自サイトの訪問者を増やしたいウェブサイトの管理者は上位 10 件以内に自分のサイトを表示させるために検索エンジンスパムを行う場合があることが指摘されている．

2.1.3　ブラウザ拡張機能

特定のウェブサイト A が別のウェブサイト B の内容を転載していることが既知であるならば，転載先のウェブサイトを検索時に除外して検索したり，検索結果において非表示にしたりする方法が有効であり，実際にそのような機能を実現するブラウザ拡張機能が提供されている．

Personal Blocklist[2] は，Google の検索結果からドメイン名を指定して特定のサイトを除外する機能を持つ Chrome 用拡張機能である．Google の検索結果から特定のサイトを除外する機能は，かつて Google 側のサービスとして “Blocked Sites” という名前で提供されており，ブロックされた回数の多いサイトの検索結果における順位を下げるなどサービスの質の改善のために利活用することが意図されていたが，このサービスは現在廃止されており，その代わりとして [2] が提供されている．

2.1.4　ユーザースクリプト

Greasemonkey[3], Tampermonkey[4] は現在表示中のウェブページに任意の JavaScript コード (ユーザースクリプト) を挿入することでウェブページの外観や挙動を変更する機能を持つ拡張機能である．これらの拡張機能を用い，検索結果において特定のウェブサイトを除外する機能を持つユーザースクリプトは多数開発・公開されている．その例として Google Hit Hider by Domain[5] や Google Domain Blocker[6] 等が挙げられる．

しかし，他サイトの転載であることが実際にアクセスするまで分からないウェブサイトの場合は，これらの拡張機能やユーザースクリプトでは対応できない．また，ブラウザ開発元が運営する配布サイト (Mozilla Add-ons, Chrome Web Store など) での公開に際して事前審査を受ける必要のある拡張機能と異なり，公開されているユーザースクリプトは必ずしも安全であるとは限らない．[7] では，Greasemonkey の仕様に言及した上で，かつてユーザースクリプトの公開・配布場所として運営されていた `userscripts.org` におけるユーザースクリプトの約 2 割は個人情報を盗み見る等悪意のあるユーザースクリプトであることを指摘している．また，このような悪意のあるユーザースクリプトでなくとも，

脆弱性により使用者が危険に晒される場合があることも指摘している．

2.2　基礎技術

ここでは，既読コンテンツを検出するために必要な類似度判定の尺度，および類似度判定を効率よく行うライブラリである SimString について説明する．

2.2.1　N-gram

文章や単語の意味を比較しようとする際には「文脈ベクトル」が用いられる．文脈ベクトルとは，文章を特定の規則に基づいて分割したとき，単語毎の出現率により与えられるベクトルである．

主要な文脈ベクトルの生成方法に N-gram がある．N-gram とは，ある文字列に含まれる，最大 N 文字までの部分文字列のである．$1 \leq N \leq 3$ の範囲が一般的に用いられ，それぞれ unigram, bigram, trigram と呼ばれる．例えば，文字列‘既読コンテンツ’に対する trigram には次の 9 要素
{‘既’, ‘既読’, ‘既読コ’, ‘読コン’, ‘コンテ’, ‘ンテン’, ‘テンツ’, ‘ンツ’, ‘ツ’}
があり，その出現回数は全て 1 である．

文脈ベクトル間の類似度の計算に用いられる類似度関数には，集合に対するもの (ある要素が含まれるかどうかのみ考慮する) と，数値ベクトルに対するものがある．何度も用いられる単語や要素であるほど重要度が高いものと考えられるため，本稿では後者を採用する．

2.2.2　N-gram に対する類似度関数

類似度関数の一つにコサイン類似度がある．数値ベクトルに対するコサイン類似度 (Cosine Similarity) は，文字列 x，y の文脈ベクトルをそれぞれ $\boldsymbol{x}, \boldsymbol{y}$ とするとき，次式により与えられる．[8]

$$\cos(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{|\boldsymbol{x}||\boldsymbol{y}|}$$

例として，x =‘既読コンテンツ’，y =‘未読コンテンツ’に対する trigram による文脈ベクトルに対するコサイン類似度を計算する．

$\boldsymbol{x}$ と $\boldsymbol{y}$ はそれぞれ 9 つの要素を含んでおり，その出現回数は全て 1 である．したがって，

$$|\boldsymbol{x}| = |\boldsymbol{y}| = \sqrt{9} = 3$$

となる．また，$\boldsymbol{x}$ と $\boldsymbol{y}$ に共通する要素は
{ 読コン, コンテ, ンテン, テンツ, ンツ, ツ }
であり，双方の要素の出現回数は全て 1 であるから，

$$\boldsymbol{x} \cdot \boldsymbol{y} = 6$$

となる．したがって，文字列「既読コンテンツ」,「未読コンテンツ」の trigram による文脈ベクトルに対するコサイン類似度は 2/3 となる．

また，同一の文字列や，文脈ベクトルの各要素の出現回数の比率が等しい文字列が与えられた場合，$\boldsymbol{x}$ と $\boldsymbol{y}$ は互いに平行となるため，コサイン類似度は 1 となる．

2.2.3　レーベンシュタイン距離

2.2.1 節にて述べた「文脈ベクトル」とそれに対する「類似度関数」によらない類似度判定手法として「レーベンシュタイン距離 (Levenshtein Distance)」がある．レーベンシュタイン距離は編集距離の一種であり，文字の挿入，置換，削除等の操作によりある文字列 x を 別の文字列 y に変更しようとした際に必要な操作の最小回数として与えられる．[9]

例えば，文字列「既読コンテンツ」を「未読コンテンツ」に改変するためには，最低でも先頭の「既」を「未」へと置換する操作が必要である．したがって，この 2 文字列間のレーベンシュタイン距離は 1 となる．一方，文字列「既に読んだ内容」を「まだ読んでいない内容」に改変するためには，最低でも 3 文字の置換，3 文字の挿入が必要であるため，この 2 文字列間のレーベンシュタイン距離は 6 となる．

また，同一の文字列が与えられた場合，レーベンシュタイン距離は 0 となる．

ただし，同じ類似度の文章であっても，一度に比較する文章量によりレーベンシュタイン距離は変化する．例えば，10 文字のうち 5 文字が異なる文章の組と，100 文字のうち 5 文字のみが異なる 2 つの文章の組では，レーベンシュタイン距離はともに 5 であるが，後者の組がより類似しているといえる．そこで，レーベンシュタイン距離により類似度を算出する際には，算出したレーベンシュタイン距離を，対象となる 2 つの文章の文字数のうち大きい方で割るという正規化を行う．

2.2.4　SimString

SimString[10] は，類似文字列検索 (文字列集合の中から，ある文字列と一定以上の類似度を持つものを探し出す操作) を高速に行うためのライブラリである．

SimString が対応している類似度関数は，コサイン係数，ジャッカード係数，ダイス係数，およびオーバーラップ係数である．また，文字列の類似度を計算するための特徴量として，2.2.1 節にて述べた文字単位の N-gram に対応している．

SimString を用いて類似文字列を検索する際の大まかな流れは次の通りである．

データベース作成　最初に，検索対象とする文字列の集合 (bag of words) を読み込ませ，データベースを作成する．本研究においては，事前に記録されていた (既に読まれた) 文字列の集合を読み込ませる．

検索　類似度の閾値や類似度関数を指定して実際の検索を行う．

SimString では，類似文字列検索においては類似度が閾値以上であるか否かのみに着目することにより高速化を図っているため，類似度の最大値を直接求めることはできない．

具体的には，与えられた閾値から検索対象とする文字列の N-gram の範囲を事前に求めることによりデータベースの検索範囲を限定している．また，クエリ文字列とデータベース中の特定文字列との間の類似度に関しては，類似度が閾値に達するために必要な共通 N-gram の最小個数をあらかじめ求め，共通する N-gram の個数がその範囲内にあるか否かにより与えられた条件を満足するかどうかを判断しており，類似度を直接求めているわけではない．

コサイン類似度についての例を示す．閾値を α，比較対象とするデータベース中の文字列・現在類似度を求めている文字列の N-gram 長をそれぞれ p, q とすると，コサイン類似度の定義から，コサイン類似度が α 以上となる可能性がある p の範囲は次式で表される．

$$\alpha^2 q \leq p \leq \frac{q}{\alpha^2}$$

この範囲を外れている比較対象文字列の類似度は α に達することはないため，検索対象から除外される．この条件は必要条件であり，この条件を満足する文字列であったとしても，類似度が閾値に達するとは限らない．

一方，コサイン類似度の定義から，コサイン類似度が α 以上となる共通 N-gram の個数は $\alpha\sqrt{pq}$ 以上である．これは必要十分条件である．

SimString は C++用ライブラリであるが，SWIG を経由することで Python/Ruby から直接呼び出すことが可能である．また，それ以外の言語であってもコマンドラインインタフェースを用いて SimString を利用可能である．

第3章

提案手法

3.1 既読コンテンツの記録・検出

3.1.1 スタンドアローンモデル

ここでは，2節にて示した基礎技術による既読コンテンツ検出，および類似度に基づいた書式変更の流れについて示す．

最初に，ユーザーが現在読んでいるページを読み終わった際に，そのページの内容を適当な単位に分割し，ページのURLや閲覧日時などの付加情報とともに記録する．

その後にユーザーが別のページを読もうとした際に，同様に現在のページの内容を分割した後，現在の内容と記録されていた内容の文脈ベクトルについて類似度関数を適用する，現在の内容と記録されていた内容間のレーベンシュタイン距離を算出するなどの方法により，既読コンテンツとの間で類似度の算出を行う．

最後に，現在のページの内容のうち，記録された内容との類似度がある閾値を超えたものに対し，一律に同一の書式を適用するか，または類似度に応じて書式を変化させる．

これらの一連の処理は，現在ユーザーが操作している端末上で動作しているウェブブラウザ上で完結するため，既読コンテンツの記録・検出に際して外部との通信は行われない．本研究ではこの実装をスタンドアローンモデルと定義する．本モデルの概略を図3.1に示す．

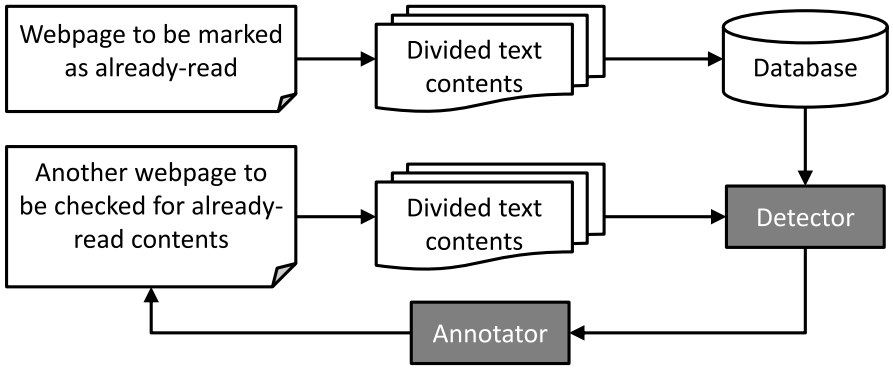

図3.1: 提案手法の概略

3.1.2　クライアントサーバモデル

一方，スタンドアローンモデルの一連の処理のうち，記録や検出の処理 (図 3.1 において “Database” および “Detector” で示される部分) をブラウザの外部で動作するサーバプログラム上で行うことも可能である．この場合は，ブラウザ拡張機能はユーザインタフェースの提供，および現在のページ内容の取得のみを行う．

この方法では前者と比較して，ブラウザ拡張機能の仕様に束縛されない実装が可能となり，より自由度が高まる．しかしながら，サーバプログラムをローカルマシン上ではなく外部のサーバ上で動作させた場合 (クラウドサービスとしての動作) は，記録・検出の際に既読コンテンツをその都度サーバにアップロードすることにより帯域幅の圧迫につながる可能性があることに加え，その動作は「ユーザーの読んだコンテンツ」を外部に送信するということに外ならないため，既読コンテンツを暗号化して保存するなどプライバシーに配慮した実装が必要となる．本研究ではこの実装をクライアントサーバモデルと定義する．

3.2　既読コンテンツの表示手法

既読コンテンツとの類似度が高い部分に対して行う書式設定方法には，図 3.2 のようなものが考えられる．

背景色変更 (網かけ)　背景色を文字色に近づけることにより，既読コンテンツを目立たなくさせる．方法 1 では類似度がある閾値以上の部分に対して同じ背景色を適用する．方法 2 では類似度に応じた濃度で網かけする．

文字色変更　背景色変更とは逆に，文字色を背景色に近づけることにより，既読コンテンツを目立たなくさせる．方法 3 では類似度がある閾値以上の部分に対して同じ文字色を適用する．方法 4 では類似度に応じて文字色を変更する．

文字サイズ変更　既読コンテンツを縮小表示することにより目立たなくさせる．方法 5 では類似度がある閾値以上の部分に対して，文字を一律に縮小する．方法 6 では類似度に応じて文字サイズを変更する．

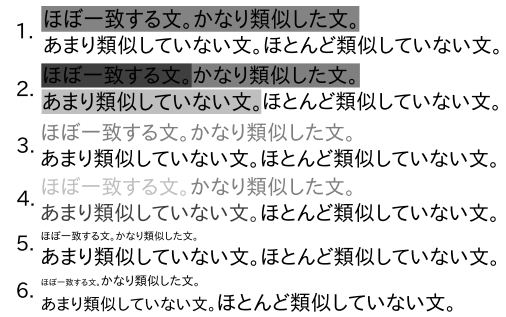

図 3.2: 表示手法

第4章

実装

本章では，第3章にて述べた提案手法における両モデルの拡張機能，およびクライアントサーバモデルにおける記録・検出用サーバプログラムの実装について述べる．

4.1 拡張機能のみによる実装(スタンドアローンモデル)

3章にて示した提案手法に基づき，既読コンテンツを検出し，未読と思われるコンテンツを優先的に表示するプログラムを，Mozilla Add-on SDK による Mozilla Firefox 用の拡張機能 (Extension) として実装した．この実装は主に「コンテンツ記録部」「検出・表示部」の2つに分けられる．

4.1.1 コンテンツ記録部

コンテンツ記録部は，現在のページを既に読んだものとして，Mozilla Add-on SDK において提供されている高レベル API である simple-storage を用いて，当該ページの全ブロックレベル要素が持つテキストノードを要素単位で記録する．図 4.1 の上部にコンテンツ記録部を示す．

例えば，次の HTML に対しては，「既読コンテンツです．」「未読コンテンツです．」の2文字列が記録される．

```
<p><em>既読</em>コンテンツです．</p>
<p><a href="foo.html">未読</a>コンテンツです．<img src="bar.png"></img></p>
```

また，よりきめ細かい検出を行うために，各要素の持つテキストノードについて，それらを句読点や空白で分割したものを同時に記録する．例えば，

> この文章は，まだ読まれていない．

という文字列は，元の文字列と共に「この文章は」「まだ読まれていない」という2文字列に分割されて記録される．

要素単位，句読点単位のいずれの場合も，記録の前に「全ての全角英数を半角に，連続する空白文字や改行を単一の半角スペースに置換する」という正規化を行う．

4.1.2　検出・表示部

検出・表示部は，現在のページに対して，記録したコンテンツとの類似度を算出する．類似度を算出しようとするページの各ブロックレベル要素について，またはそれらを 4.1.1 節と同様に句読点単位で分割したものについて，記録されたコンテンツとの間の類似度 (2.2.2 節にて述べた類似度関数，および 2.2.3 節にて述べたレーベンシュタイン距離) を算出する．図 4.1 の下部に検出・表示部を示す．

類似度を算出した後，現在のコンテンツに対する「既読として記録したコンテンツ」の類似度に関して，図 3.2 のように，その最大値に応じて書式を設定するか，または類似度がある閾値を超えた場合に一律に書式を設定するなどの方法でユーザーに通知する．

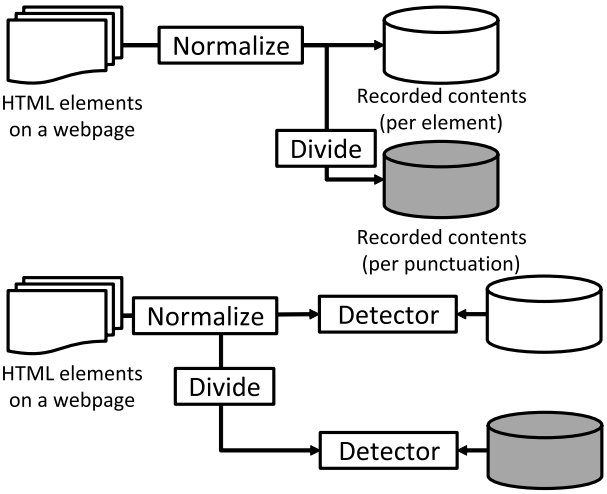

図 4.1: スタンドアローンモデルの概略

4.1.3　検出・表示部における書式設定処理の流れ

要素単位の書式設定は，該当する要素に `jQuery.css()` を用いて CSS によるスタイルを設定することにより行う．その具体的な処理の流れを以下に示す．

ここでは，類似度に応じた書式設定を行う際の当該要素に対する既読コンテンツの最大類似度を S_{max} とする．閾値以上の類似度を持つ要素に対し一律に書式設定を行う場合は $S_{max} = 0.5$ として処理を行う．

網かけ (背景色変更) の場合は，最初にその要素の前景色 (文字色) を取得する．ここで取得に失敗した場合は前景色を暫定的に「黒」として扱う．続いて背景色を前景色と同一の色に設定し，透明度 (RGBA カラーモデルの `alpha` 値) を $0.8S_{max}$ に設定する．この処理により，完全に一致している (類似度 1 の) 場合の背景色の `alpha` 値は 0.8 となり，不一致 (類似度 0) の場合は 0 となる．

alpha 値が 0 の場合は完全に透明，そこから `alpha` 値が大きくなるにつれて透明度が下がっていき，1 となると完全に不透明となることから，図 3.2 の状況においては，完全に不一致である場合の表示は元のままであり，類似度が高くなるにつれて背景色が暗くなっていくが，完全に一致している場合でも元の文字列が完全に見えなくなることはない．

文字色変更の場合は，当該要素の不透明度 (`opacity` プロパティ) を $1 - 0.8S_{max}$ と設定す

る．この処理により，$S_{max} = 1$(完全一致) の場合の不透明度は 0.2 となり，$S_{max} = 0$(不一致) の場合は 1 となる．

不透明度を変化させた際の表示の変化は `alpha` 値の場合と同様である．したがって，図 3.2 の状況においては，完全に不一致である場合の `opacity` は 1 であることから表示は元のままであり，類似度が高くなるにつれて文字色が薄くなっていく．しかし，完全に一致している場合でも `opacity` は 0.2 であることから元の文字列が完全に見えなくなることはない．

文字サイズ変更の場合は，CSS の%値を用い，フォントサイズ (`font-size` プロパティ) を基準値の $1 - 0.5S_{max}$ 倍に設定する．この処理により，$S_{max} = 1$(完全一致) の場合はスタイル `font-size:50%;` が設定され，$S_{max} = 0$(不一致) の場合は `font-size:100%;` が設定される．

したがって，図 3.2 の状況においては，完全に不一致である場合の表示は元のまま，類似度が高くなるにつれて文字サイズが小さくなっていき，完全に一致している場合は元の大きさの 1/2 となる．

ここまでは要素単位での書式設定の際の処理について説明した．句読点単位の書式設定の場合については，そのままでは該当する部分の書式を直接設定することが不可能であるため，その文字列が含まれる要素から該当する文字列を検索し，`<span></span>` タグで囲むという処理を行った後，追加された `span` 要素に対して前述の書式設定を行う．

4.2　ユーザインタフェース

本実装では，メニューバーに追加されたボタンを押すと表示されるパネルから全ての機能を呼び出すことができる．図 4.2 の水平線より上部が記録用，下部が検出用メニューとなる．

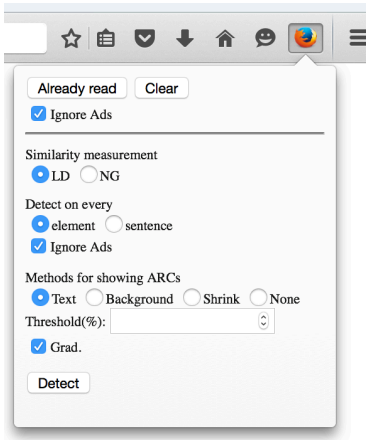

図 4.2: スタンドアローンモデルのユーザインタフェース

4.3　拡張機能と検出用サーバを組み合わせた実装 (クライアントサーバモデル)

続いて，クライアントサーバモデルによる実装について述べる．第 3 章における提案手法のうち現在表示されているページの文字列データを取得する部分，および取得された文字列データを分割する部分については 4.1 節にて述べたスタンドアローンモデルと同様であるが，取得された文字列データは外部のサーバに送信され，記録・検出はサーバ上で行われる．

本研究では，HTTP サーバプログラムとして nginx を，クライアントからの記録・検出要求を CGI(Common Gateway Interface) を介して処理するためのプログラミング言語として Perl を，記録されたサイトやその閲覧日時を保持するためのデータベース管理システムとして MySQL を，類似度算出のためのプログラムとして SimString をそれぞれ用いた．

2.2.4 節で述べたように，SimString は SWIG を用いることにより C++以外のプログラミング言語から呼び出すことが可能である．しかし，SimString 開発者による Perl のサポートが行われていないため，現時点では直接コマンドラインインタフェースを呼び出し標準入出力経由で実行結果をやりとりする仕様としている．

また，クライアント側 (拡張機能側) の実装は複数ブラウザに対応する新しいシステムである WebExtensions として書き直している．

現状では，公開サーバ上で処理を行うことに対してはプライバシー・セキュリティ上の危険性が大きいため，記録・検出用サーバはプライベートネットワーク上で稼動させ，記録・検出用サーバへの接続は，同一のネットワーク上にある公開サーバへのポートフォワーディングを伴う SSH 接続 (トンネリング) により行うこととしている．クライアント，公開サーバ，および記録・検出用サーバへの通信経路を図 4.3 に示す．具体的には，実際にコンテンツを閲覧しているコンピュータ上の `localhost:10080` に向けて通信を行うと，その通信は公開サーバを介して記録・検出用サーバの 80 番 (HTTP) ポートに到達するよう設定している．図 4.3 において，クライアントから公開サーバまでの SSH 通信 (破線矢印) は暗号化されている．また，番号が書かれた長方形は通信に用いられ接続を待ち受けるポート番号を表し，斜線が描かれた長方形は TCP/IP プロトコルスタックにより接続されている間のみ自動的に割り当てられるポート (エフェメラルポート) を表す．

図 4.3 はクライアントが記録・検出用サーバとは別のネットワーク上に存在する場合を想定して描かれているが，クライアントと記録・検出用サーバが同一のネットワーク上に存在する場合であっても設定変更が不要となるようゲートウェイサーバを設定している．

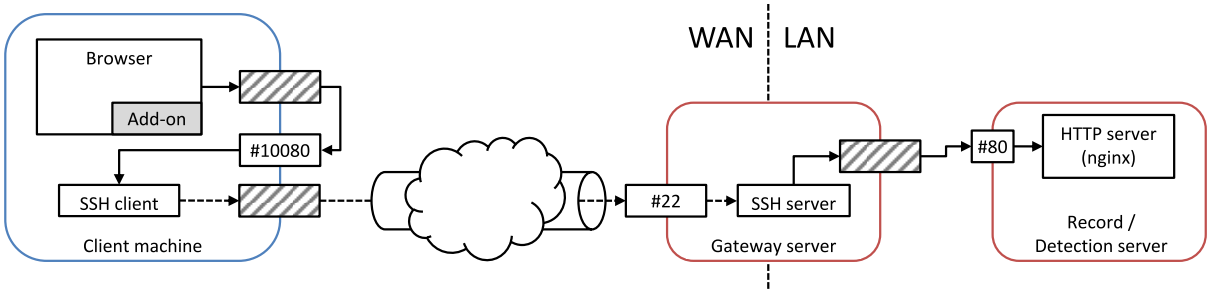

図 4.3: クライアントサーバモデルの概略

4.4　ユーザインタフェース

本実装では，スタンドアローンモデルと同様にパネルから全ての機能を呼び出すことができる．図 4.4 において水平線で分けられた上段が資格情報入力画面，中段が記録用メニュー，下部が検出用メニューとなる．

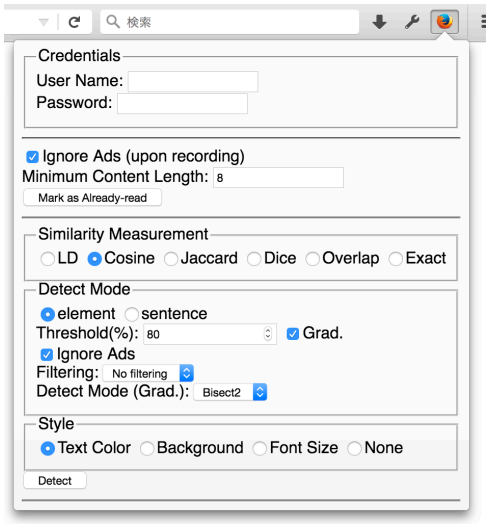

図 4.4: クライアントサーバモデルのユーザインタフェース

4.4.1　既読コンテンツデータベースの仕様

この実装では，複数ユーザーによる利用を想定し，既読コンテンツ情報を格納するテーブルを利用者の人数分作成し，記録・検出時のリクエストに含まれる資格情報 (ユーザ名・パスワード) に基づいて使用するテーブルを切り替えるという仕様とした．具体的には，新規ユーザーを作成すると，既読コンテンツの情報を格納するデータベース `ArcdTestDB` 中に `ArcdTbl_[UserName]` という名前のテーブルを作成し，該当するユーザーにそのテーブルへのアクセス権を付与する．

各テーブルの構造は次の通りである．

id　整数型，一意の ID

date　日付時刻型，ページの閲覧日時

url　可変長文字列型，ページの URL

elem_path　可変長文字列型，既読コンテンツの実体 (JSON 形式ファイル) のファイル名

既読コンテンツの実体は JSON 形式ファイルとして保存される．以下にその構造を示す．

```
{
  "url": "http://www.example.com/",
  "elem": [
    {
      "ignored": false,
      "raw": "",
      "raw_token": "",
```

```
      "punc": [],
      "punc_token": [],
      "selector": ["html", "body", "p:nth-of-type(2)"]
    }, ...
  ]
}
```

このファイルが表すオブジェクトの各要素は次の通りである．

url　文字列，現在閲覧中のページの URL

elem　各 HTML 要素を記述するオブジェクトからなる配列

elem を構成するオブジェクトの各要素は次の通りである．

ignored　この要素が 5.2 節にて述べる「非コンテンツ部」として無視されるかどうかを表す真偽値

raw　この要素全体の文字列データ (正規化済)

raw_token　raw を形態素解析し，分かち書きした文字列

punc　raw を句読点で分割した配列

punc_token　raw_token の各要素を形態素解析し，分かち書きした文字列からなる配列

selector　各要素の一意なセレクタを表す配列

4.4.2　記録時の処理の流れ

最初に記録時の処理の流れについて説明する．現在のページを既に読んだものとして記録する際の，クライアントおよびサーバ側の処理の流れを図 4.5 に示す．

最初に，拡張機能のみによる実装と同様の処理により，現在表示されているページの文字列データ，およびその文字列が含まれる HTML 要素の一意な CSS セレクタを取得し，JavaScript オブジェクトの形で保持する．

続いて，このオブジェクトにユーザ認証のための資格情報を付加した上で，JavaScript における標準 API である JSON.stringify() を用いて JSON 形式の文字列に変換し，XMLHttpRequest を用いて記録・検出用サーバ上の record.cgi へと送信する (1).

要求を受け取ったサーバプログラムは record.cgi にその入力を渡す．record.cgi は，最初に各要素の文字列データについて形態素解析エンジン MeCab[11] を用いた形態素解析を行い，その結果を現在のページ情報に付加する (2)．これは将来的に形態素解析に基づく類似度判定を行うためのものである．

続いて，当該ページ情報を，UUID(汎用一意識別子) により重複しないファイル名とした JSON 形式ファイルとして保存し (3)，そのファイル名と現在時刻，現在のページの URL を既読コンテンツテーブルに追記する (4)．データベースの肥大化を避けるため，追記を行う際には事前に既読コンテンツテーブルから url フィールドが現在の URL と完全に一致するレコードを全て削除する．

4.4.3　検出時の処理の流れ

続いて，検出時の処理の流れについて説明する．現在のページから既読コンテンツを検出する際の，クライアントおよびサーバ側の処理の流れを図 4.5 に示す．

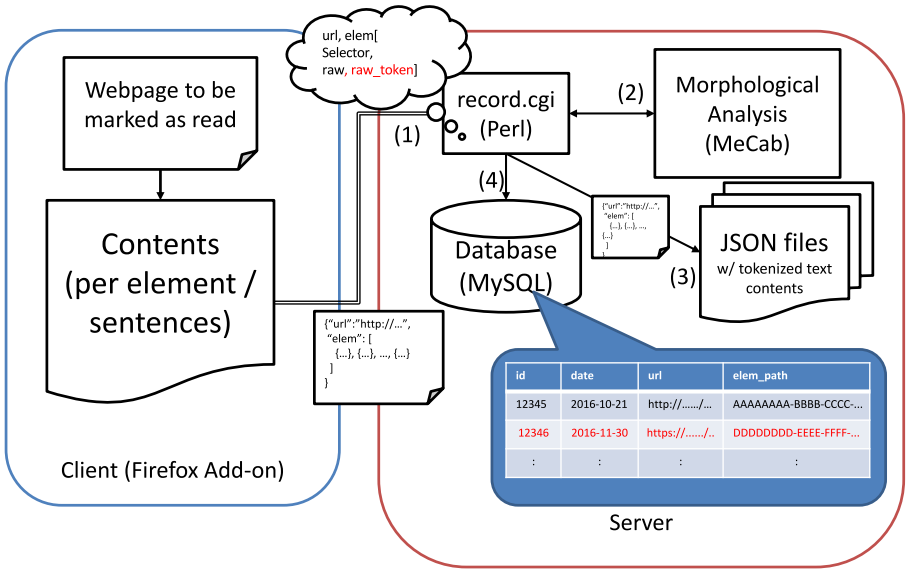

図 4.5: 記録時における処理の流れ

現在のページの文字列データおよび一意な CSS セレクタの取得については記録時と同様である．

取得した文字列データを含む JavaScript オブジェクトに類似度関数や検出単位等を含むパラメータ，および資格情報を付加した上で JSON 形式の文字列に変換し，XMLHttpRequest を用いて記録・検出用サーバ上の `detect.cgi` へと送信する (1)．

要求を受け取った `detect.cgi` は，最初に受け取った JSON ファイルからパラメータを取得する．次に，既読コンテンツテーブルから既読コンテンツの実体が格納されている JSON 形式ファイルのリストを取得する (2)．ここでは，パラメータに基づいて，例えば「直近 1 ヶ月」というような期間指定検索を行うことも可能である．

続いて，データベースから得られた既読コンテンツファイルに順次アクセスし，要素単位，または句読点単位の文字列データを読み込む (3)．

続いて，読み込んだデータについて，SimString による類似文字列検索用一時データベースを作成する (4)．この一時データベースは全ての検出処理が完了するまで保持される．

引き続き，SimString により，各文字列に対して一定以上の類似度を持つ文字列が存在するかどうか，順次検索する (5)．

SimString による検索の終了後，その結果に基づき現在のページ情報に類似度の情報 (図中 `raw_sim`, `punc_sim`) を付加し (6)，JSON 形式へと変換した上でクライアントへと返す (7)．`raw_sim` は現在の要素全体の類似度を表す数値である．また，`punc_sim` は句読点単位で分割された各文字列の類似度を表す数値からなる配列である．

類似度検索の結果をサーバから受けとった Firefox アドオンは，その結果に含まれる要素単位，あるいは句読点単位の類似度に基づき，書式設定を行う (8)．

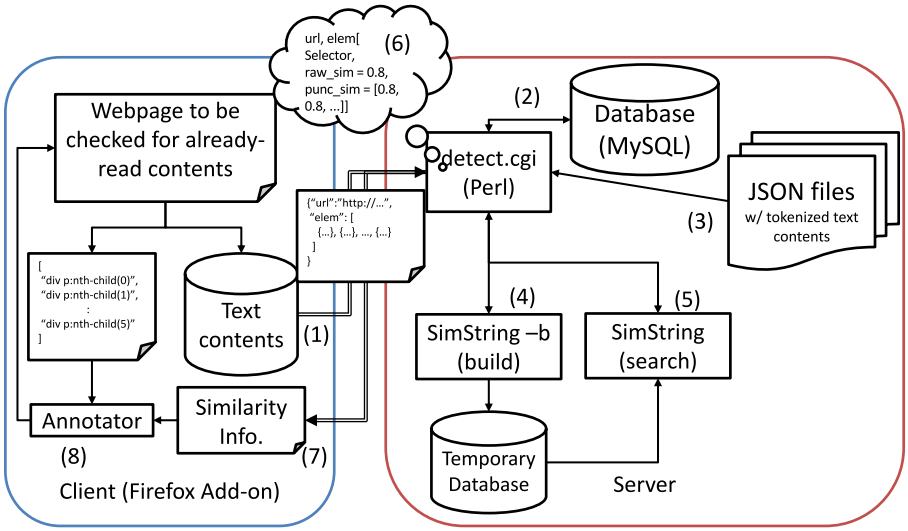

図 4.6: 検出時における処理の流れ

4.4.4　連続書式変化の場合の SimString の実行方法

類似度に応じた書式設定を行う場合，スタンドアローンモデルでは類似度の最大値を直接求めることが可能である．しかし，クライアントサーバモデルの場合はそのような処理は不可能である．これは，2.2.4 節でも述べた通り，SimString ではあらかじめ類似度の閾値を指定し，その条件を満足する文字列を検索することは可能であるが，類似度の最大値を直接求めることは不可能であることによる．

したがって，類似度に応じた書式設定を行う場合は，類似度を段階的に変化させて SimString を繰り返し実行することが必要となる．この場合の SimString の実行方法として，次のものが考えられる．

1. 線形探索 (Linear search)
2. 二分探索 (Bisection search)
3. 傾斜配分探索 (Inclined search)

SimString では，同一の類似度閾値・SimString データベースを使用している限り，1 回のコマンド実行で複数のクエリ文字列について一度に類似文字列探索を行うことができる．

例えば，コサイン類似度が 0.8 以上の既読コンテンツがある文字列を検索したい場合は，類似度判定を行いたい文字列を 1 行 1 文字列として query_strings.txt に記述した上で次のコマンドを実行すると，クエリ文字列ごとに条件を満たした既読コンテンツの数が出力される．(SimString は標準入力から 1 行 1 文字列としてクエリ文字列を読み込む．)

実行するコマンド：

```
simstring -p -u -d already_read_contents.db -t 0.8 -s cosine < query_strings.txt
```

出力：

```
0 strings retrieved (0.000549 sec)
                :
0 strings retrieved (2.5e-05 sec)
Total number of queries: 37
```

```
Seconds per query: 0.000124838
Number of retrieved strings per query: 0
```

実際の類似文字列探索処理は，一時テキストファイルからの入力の代わりに，パイプを介して検索対象文字列を逐次入力することにより行われる．

1. の線形探索は，類似度の閾値を 0 から 1 の間で線形的に変化させるものである．この方法では，類似度を一定量ずつ大きくしていった上で，その類似度に対して上記の SimString による類似文字列検索を実行する．その結果 1 以上の値が出力された場合は，そのクエリ文字列の元となる文字列データに対して当該類似度を設定する．この処理は閾値が 1 に達するか，条件を満たす既読コンテンツが見つからなくなるまで繰り返し行われる．この方法では n 回のループにより $1/n$ 刻みの検出が行える．この実装では $n = 10$ とする．

2. の二分探索は，ある閾値における結果に応じて次回の検索時の閾値・検索範囲を狭めていくものである．

二分探索による類似度算出処理の擬似コードをアルゴリズム 1 に示す．

Algorithm 1 Calculate similarity by bisection method

```
threshold ⇐ 0.5
p ⇐ 0
tick ⇐ 0.5
while p < depth do
  tick ⇐ tick/2
  Search for strings whose similarity is larger than threshold
  if Similar strings found then
    threshold ⇐ threshold + tick
  else
    threshold ⇐ threshold − tick
  end if
  p ⇐ p + 1
end while
return threshold − tick
```

この方法では，n 回のループにより 2^{-n} 刻みの検出が行える．しかし，この方法では対象の文字列ごとに閾値が独立して変化する関係上，1 回のコマンド実行で複数のクエリ文字列を一度に処理することはできない．したがって，1 つの文字列に対して n 回 SimString を実行するという処理を，全ての文字列に対して繰り返すことが必要となる．この実装では $n = 4$ とする．

3. の傾斜配分探索は，類似度の高い既読コンテンツを探索する処理により多くの計算機資源を与えるものである．基本的な処理の流れは「線形探索」と同様である．この方法の「線形探索」との相違点は，n 回目の SimString の実行における閾値 T_n を次のように設定することである．

$$T_n = 1 - 2^{-n}$$

この方法では，n が大きくなるにつれて T_n は 1 に漸近する．また，類似度の間隔は最終的に出力される類似度に依存し，1 に近いほど小さくなる．この実装では $n = 4$ とする．

第5章

評価

本章では，提案手法の第4章による実装の有効性を確認するため，いくつかの実験を行った．

5.1 スタンドアローンモデルにおける検出性能評価

5.1.1 検出対象の選定

本節では，第2章において述べた重複するコンテンツや類似したコンテンツのうち，既読コンテンツの検出対象として選定した具体的な例を示す．

重複はしていないが似通ったコンテンツが多くなる例としては，ある同一製品について説明するサイトとして特定の医薬品の作用・用法等を説明するサイトを用いた．また，同一の主題について説明するサイトとして，ある事件について報じるニュースサイト2社についての記事を用いた．

また，重複コンテンツが発生する引用・転載サイトとして，Wikipediaの内容を転載したサイトや，多数のニュースサイトを1箇所に転載したサイトを用いた．

いずれの例においても，一方のページの内容を既に読んだものとして記録し，もう一方のページにおいて既読コンテンツの検出を行うという操作を行い，平均の類似度(レーベンシュタイン距離およびコサイン類似度)を求めた．

5.1.2 同一製品について説明している複数のサイト

ある医薬品について，その作用・用法等を説明するサイト5つについて，4.1節による実装を用い，あるサイトを基準とした他の4サイトの類似度を算出した．

実験の流れは次の通りである．ここで，5つのサイトをそれぞれA-Eとする．

1. 対象のページ5つをFirefoxに読み込む
2. Aの内容を記録する
3. A以外のページについて，レーベンシュタイン距離およびコサイン類似度を算出する
4. 記録されたコンテンツを全て削除する
5. B-Eに対して手順2-4を繰り返す

3種類の医薬品についてこれらの手順を行い，特定の記録対象および検出対象における3種類分の実行結果について，まとめて平均値を算出した．

紹介サイト 5 つのうち 2 サイト間の平均レーベンシュタイン距離を表 5.1 に，平均コサイン類似度を表 5.2 にそれぞれ示す．各表において，行番号は内容を記録したサイトを，列番号は記録された内容に基づき類似度の算出を行ったサイトを表す．また，上段は要素単位，下段は句読点単位の検出による結果である．

なお，表 5.1 では値が小さいほど類似度が高いことを表し，表 5.2 では値が 1 に近いほど類似度が高いことを表す．

サイト D を基準とした他サイトの類似度，サイト B-E を基準としたサイト A の類似度は，レーベンシュタイン距離，コサイン類似度のいずれにおいても，比較する単位の差異により共通の変化の傾向が見られた．

また，比較する単位を句読点単位とした場合について見ると，サイト A および E を基準とした他サイトの平均レーベンシュタイン距離は，いずれも要素単位のものより大きくなった．

	A	B	C	D	E
A		.400688	.396473	.320590	.302865
		.440060	.562170	.464614	.351725
B	.375787		.413358	.346406	.323442
	.394400		.443295	.336105	.294258
C	.336268	.385257		.333383	.315399
	.421698	.339467		.330935	.296609
D	.403517	.429575	.444057		.352855
	.388309	.367153	.494061		.294953
E	.358127	.411563	.427519	.347042	
	.491321	.457500	.575659	.441785	

表 5.1: 2 サイト間の正規化レーベンシュタイン距離の平均値

	A	B	C	D	E
A		.118577	.188876	.306427	.156342
		.114032	.153166	.464216	.229823
B	.124681		.151229	.185107	.139744
	.090801		.168324	.097633	.170462
C	.197419	.173911		.170940	.148671
	.098567	.119094		.093702	.167518
D	.136882	.098953	.132313		.092190
	.282152	.099138	.102242		.188049
E	.185711	.148428	.171053	.169201	
	.076533	.087374	.080884	.084306	

表 5.2: 2 サイト間のコサイン類似度の平均値

5.1.3　別のサイトの内容を転載しているサイト

あるウェブサイトが別のウェブサイトの内容を転載している最初の例として，Wikipedia 日本語版と，ポータルサイトのサービスの一環としてその内容を転載している「goo Wikipedia 記事検索」が挙げられる．これらの 2 サイトについて，転載元 (Wikipedia) のある記事を既読として記録し，転載先 (goo Wikipedia) において既読コンテンツの検出を行い，レーベンシュタイン距離が 10 以下，またはコサイン類似度が 0.8 以上の部分を表示した．その結果を表 5.1 に示す．

表 5.1 のうち左側が転載元，右側が転載先である．要素単位であればいずれの方式であっても，転載されたコンテンツは既読として検出されている．一方，句読点で分割した場合，既読として検出されたコンテンツは 2 割程度であった．

(以下，‘LD’ はレーベンシュタイン距離を，‘CS’ は trigram による文脈ベクトルに対するコサイン類似度を表すものとする．)

また，ページの全ての内容を転載した上記サイトの他に，ページの一部分を引用している 3 サイトを加え，Wikipedia の元の記事を基準として類似度を算出した．表 5.3 に平均の類似度を示す．

どの方式であっても，元サイトを基準とした転載サイトの平均類似度は，いずれの引用サイトの平均類似度をも上回った．

ウェブサイトの転載のもう一つの例として，多数のニュースサイトを転載し，同一サイトでまとめて閲覧できるようにしたサービス「Livedoor News」を挙げる．産経新聞から

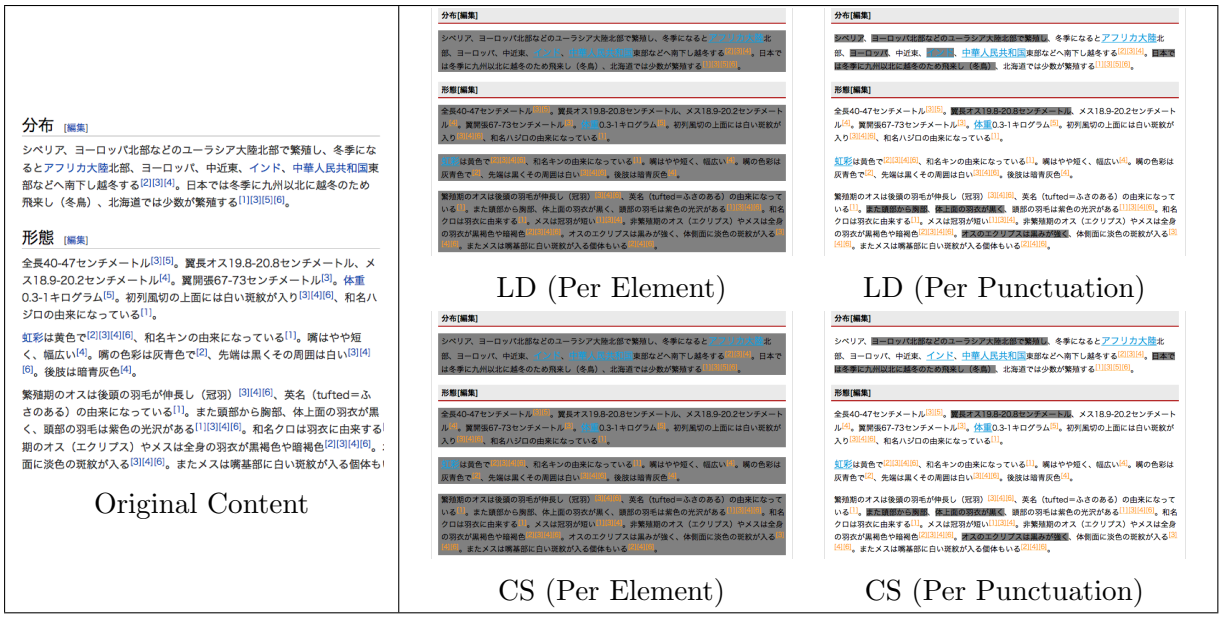

図 5.1: 転載サイトに対する既読コンテンツ検出の結果

	Reproduced	Quoted A	Quoted B	Quoted C
LD	.647504	.862270	.885704	.916880
	.693850	.848955	.837688	.830763
CS	.481817	.105649	.187705	.165848
	.277880	.095252	.076253	.062338

表 5.3: Wikipedia クローンにおける平均類似度

Livedoor News へと転載された，ある事件について報じた記事に対して，転載元を「既読コンテンツ」として記録し，転載先の記事に対して検出処理を行うという，Wikipedia の転載記事と同様の実験を行った．

転載元の記事は 2 ページに分割されていたが，転載先では 1 ページにまとめられている．転載元にて 1 ページ目のみ既読として記録を行った後，転載先の記事において既読コンテンツの検出を行うと，どの方法を用いても元の記事の 1 ページ目にあたる部分が既読として検出された．(図 5.2 では，「公務執行妨害で現行犯逮捕された．」までが 1 ページ目，「京大では 4 日」以降が 2 ページ目である．)

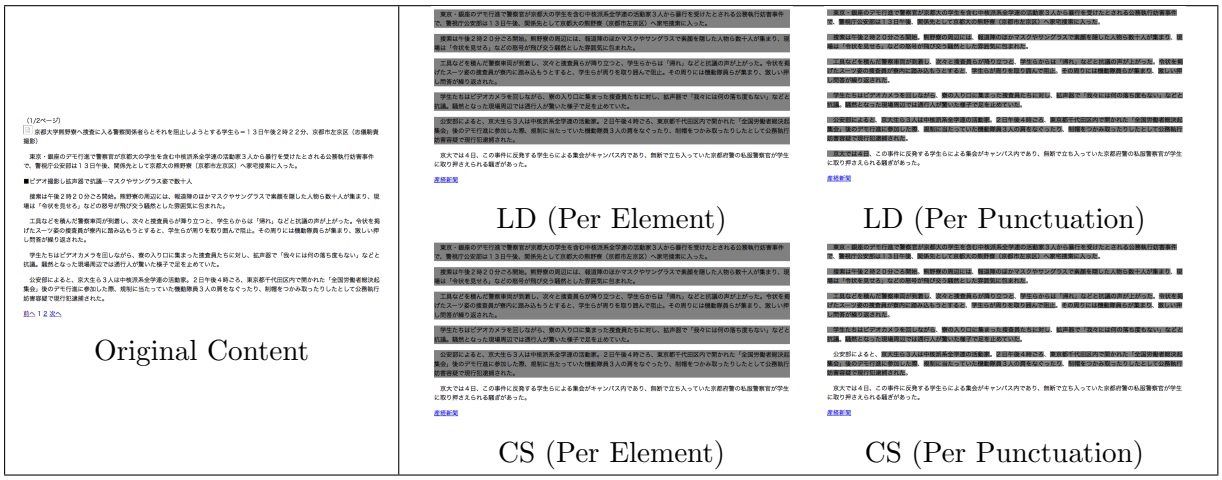

図 5.2: 転載サイトに対する既読コンテンツ検出の結果

5.1.4　同一主題について述べている複数の記事

同一主題について述べている複数の記事として，ある研究についての日本経済新聞 (日経) の記事を既読として記録し，同じ研究について報じている NHK NEWS WEB 内の記事 (転載ではない) に対して既読コンテンツの検出を行った．

その結果，要素単位ではレーベンシュタイン距離が 10 以下となる，またはコサイン類似度が 0.8 以上となることはなかった．一方，句読点単位では 1 割程度の領域についてレーベンシュタイン距離が 10 以下となったが，コサイン類似度が 0.8 以上となることはなく，2 割程度の領域が 0.3 以上 0.8 未満となった．

5.2　記録や検出時における非コンテンツ部の影響

第 4 章において述べた実装では，メニューや関連記事，広告など，ページの主な内容ではない部分が多くを占めるページであっても，それらが共に記録される．例えば，5.1.3 節における実験では，転載元のコンテンツを記録する際に，「メニューバー」や「他言語版へのリンク」なども記録されていた．本研究では，当該ページの主題とは無関係なこれらの部分を「非コンテンツ部」(NCP: Non-content part) と呼ぶこととする．NCP の例を 5.4 に示す．この図において NCP は赤く塗りつぶされた部分となる．

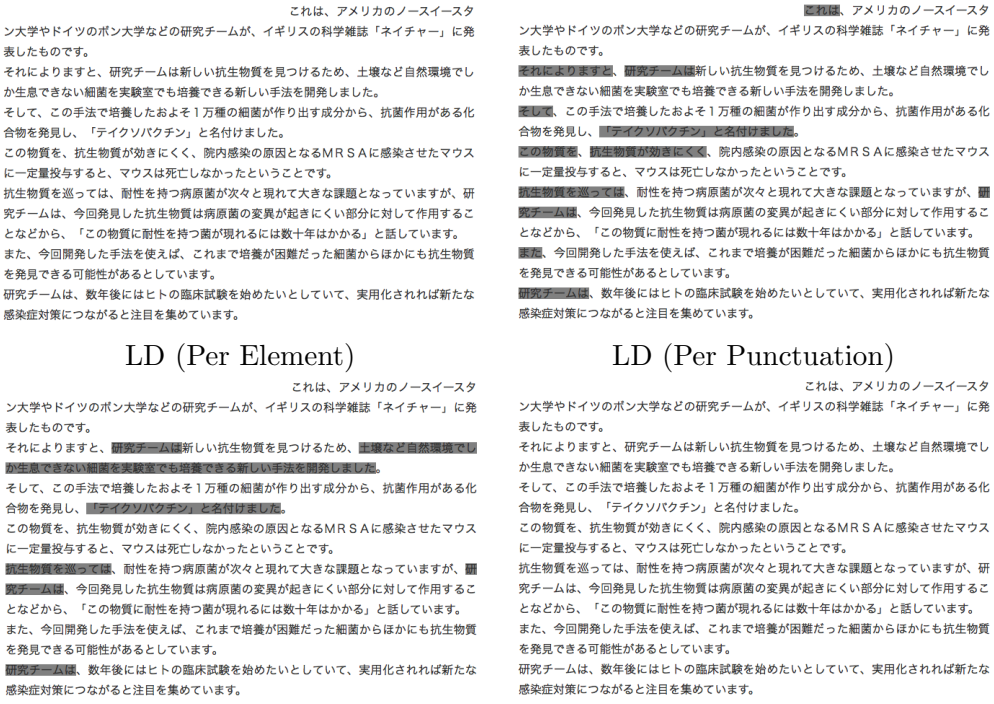

図 5.3: 同一主題のサイトに対する既読コンテンツ検出の結果

図 5.4: 非コンテンツ部の例

これはコンテンツを記録する際に発生する事象であるが，既読コンテンツの検出の際にも，NCP が多いページほど検出および書式設定に長時間を要するものと考えられる．実際に 5.1.3 節において用いたページでは，書式設定が完了するまでに数秒の待ち時間が発生し，ブラウザの応答停止が発生することもあった．

表 5.4 に 5.1.3 節における類似度指標，検出単位，NCP 除外の有無，および表示方法による処理時間と，NCP を除外することによる時間短縮率を示す．5.1.3 節における転載先ページには，そのページの主題とは外れる「広告」「広告・アクセス解析用スクリプト」「人気記事」などの NCP が存在する．それらを除外することにより処理時間が平均で 2.61%短縮された．

Meas.	Unit	Ignore NCP	Detection only	Text Color	Background	Font Size
LD	Elem.	No	1.452	1.457	1.518	1.516
		Yes	1.418 (-2.34%)	1.501 (+3.02%)	1.476 (-2.77%)	1.493 (-1.52%)
	Punc.	No	1.091	4.166	4.272	4.232
		Yes	1.071 (-1.83%)	4.183 (+0.41%)	4.211 (-1.43%)	4.182 (-1.19%)
CS	Elem.	No	1.294	1.453	1.330	1.280
		Yes	1.254 (-3.09%)	1.304 (-10.3%)	1.238 (-6.92%)	1.265 (-1.17%)
	Punc.	No	2.724	5.863	5.873	5.688
		Yes	2.606 (-4.33%)	5.786 (-1.31%)	5.593 (-4.77%)	5.559 (-2.27%)

表 5.4: NCP を無視することによる所要時間の変化 [s]

5.3　実装間の性能比較

第 4 章において示した「スタンドアローンモデル」と「クライアントサーバモデル」という実装の差異により所要時間にどのような影響が出るかを調べるために，5.1.3 節におけるニュースサイトの転載記事を用い，以下の条件で既読コンテンツの検出・表示を行い，検出および書式設定に必要な所要時間を計測した．

固定条件　書式設定: 背景色，類似度関数: コサイン，検出単位: 文単位

変動条件　検出方法: スタンドアローン/クライアントサーバモデルの各方法

スタンドアローンモデルおよびクライアントサーバモデルにおける所要時間を表 5.5 に，クライアントサーバモデルにおいて背景色変更を行った際の検出方法による所要時間の変化を図 5.5 に示す．図 5.5 のうち右側は，左側の「コサイン類似度 0.8 を閾値とした場合」の部分を拡大したものである．また，図 5.5 のうち “F.C.” は文字色，“B.C.” は背景色，“Size” は文字サイズを変化させた場合の所要時間を表す．

この結果から，既読コンテンツの検出部分を外部のサーバに置くことにより，同じ検出処理であってもより高速に実行できることが分かった．特に，コサイン類似度 0.8 を閾値とした一律書式設定においては，スタンドアローンモデルでは検出および書式設定に 3 分程度かかっており実用的ではなかったものが，クライアントサーバモデルを用いることにより 2 秒程度に短縮され，実用に足る処理速度となった．しかし，類似度に応じた書式設定の所要時間に着目すると，双方の実装間で変化は見られず，50 秒から 60 秒程度となっている．また，図 5.5 から，クライアントサーバモデルにおける類似度に応じた書式設定の所要時間は，どのような書式設定を行うかにかかわらず 49〜50 秒という値を示した．

	Stand-alone		Client-server	
Style	BG	None	BG	None
Threshold 80%	172.243	174.934	1.749	1.273
Gradation (Direct)	242.574	179.130	N/A	N/A
Gradation (Linear)	N/A	N/A	53.606	3.965
Gradation (Bisection)	N/A	N/A	92.700	42.099
Gradation (Incl.)	N/A	N/A	51.097	1.515

表 5.5: 実装による既読コンテンツ検出・表示の所要時間の変化 [s]

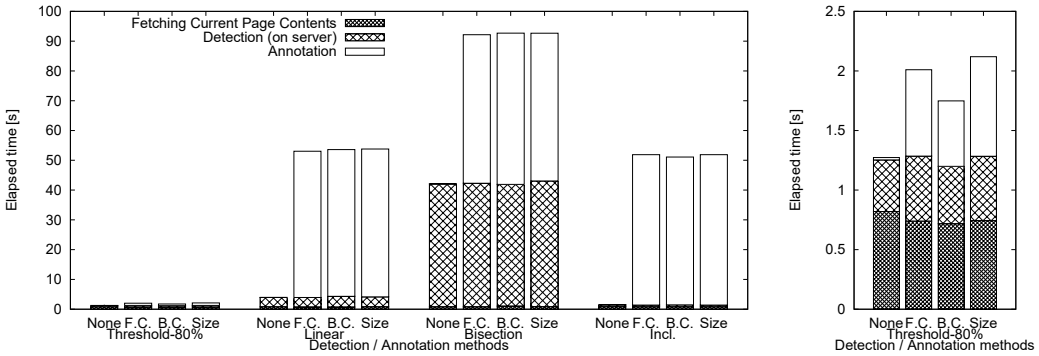

図 5.5: 検出方法による既読コンテンツ検出・表示の所要時間の変化 [s]

5.4　書式設定処理時間

次に，検出単位による書式設定処理時間の変化を調べるために，次の条件で書式設定に必要な所要時間を計測した．

固定条件　書式設定: 背景色，非コンテンツ部: 無視する，類似度関数: コサイン

変動条件　検出単位: 要素単位/句読点単位，検出方法: クライアントサーバモデルの閾値 80%/線形探索

所要時間の計測方法は，console.time()，および console.timeEnd() の組を書式設定処理の前後に挿入することで行った．処理時間を計測したい部分に `console.time("timer_name")`・`console.timeEnd("timer_name")` を挿入することにより，コンソールに経過時間を出力することができる．

実験の結果を表 5.6 に示す．

	Threshold 80%	Gradation
Per Element	0.023	0.070
Per Punc.	0.523	48.676

表 5.6: 既読コンテンツの書式設定所要時間 [s]

その結果から，いずれの書式設定方法を用いた場合でも呼び出される処理や関数はほぼ同じである (唯一異なる部分は類似度の値) にもかかわらず，句読点単位の処理であり，

かつ連続的書式設定の場合の所要時間が極端に長いことがわかった．

5.5　既読コンテンツの量による検出時間

このアドオンは使用を続けると蓄積された既読コンテンツの量が多くなり，その結果検出に必要な所要時間が延びるものと考えられる．

蓄積された既読コンテンツの量が処理時間に与える影響を調べるために，任意の大きさの既読コンテンツデータベースを作成し，それに対して検出を行いサーバ上の所要時間を計測する実験を行った．

5.5.1　実験方法

実験の流れは大きく分けて実験用既読コンテンツデータベースの作成，および実際の計測作業に分けられる．

実験用既読コンテンツデータベースの作成手順は次の通りである．

最初に，任意の名前の一時ユーザーを作成し，そのユーザーとしてある 1 ページのみを record.cgi を用いて記録する．その結果 1 ページのみが記録されたテーブルが作成される．

次に，`scaltest_num` (この実験では $num = 5, 10, 15, \ldots, 100$) というユーザーを作成し，一時ユーザー用テーブルの内容を当該ユーザー用の既読コンテンツテーブル `ArcdTbl_scaltest_num` に対して num 個挿入する．

この作業を行ったテーブルには見かけ上複数の既読コンテンツが記録されているように見えるが，実際には同一の既読コンテンツファイルを参照している．

所要時間の計測は，検出サーバ上で Wireshark(ネットワークアナライザ) を稼動させた上で次の条件で既読コンテンツの検出を行い，Wireshark に表示されるタイムスタンプを見ることにより行った．所要時間は検出サーバにリクエストの最初のパケットが到達してから，それに対する応答 (ACK) を送信し終わるまでの時間差とする．

固定条件　書式設定: 無し (検出のみ)，非コンテンツ部: 無視する，類似度関数: コサイン，時間によるフィルタリング: 無し

変動条件　検出単位: 要素単位/句読点単位，検出方法: 閾値 80%/線形探索/二分探索/傾斜配分探索，既読コンテンツ数: $5, 10, 15, \ldots, 100$

実際の計測作業では，書式設定を伴う既読コンテンツ検出を行った場合は一度ページの再読み込みを行わないと再び検出を行うことはできないが，ページの書式設定を行わない場合は連続して検出を行うことができることを利用し，パネルを開いたままにした状態でパラメータを変更しながら Detect ボタンを押すことを繰り返す．

本実験の一時テーブルに記録された既読コンテンツファイルには 62 個の HTML 要素データが含まれており，句読点単位で分割された文字列データは合計で 1247 個存在する．

5.5.2　結果

実験の結果を図 5.6 に示す．図 5.6 の点線部分は各系列の線形回帰による近似曲線である．また，図 5.6 のうち右側のものは左側のうち $0 \leq y \leq 14$ の部分を拡大したものである．

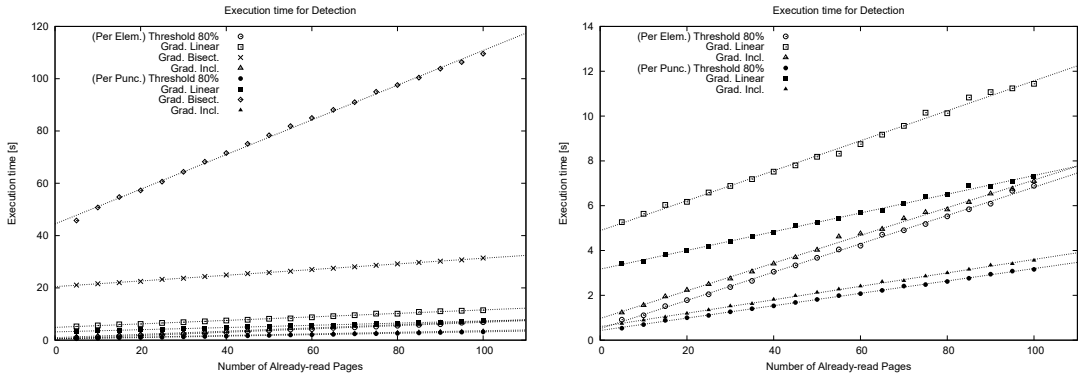

図 5.6: 既読コンテンツの量による所要時間の変化 [s]

いずれの条件においても，所要時間は既読コンテンツの量により線形的に変化した．また，いずれの条件においても決定係数 $R^2 > 0.995$ であった．

1 回の既読コンテンツファイル処理あたりの所要時間に着目すると，「二分探索」以外における所要時間は 10ms オーダーであり，「二分探索」では 100ms オーダーであった．

Mozilla Firefox は，事前に許可 (オプトイン) したユーザーに対して，ブラウザの動作状況を匿名の統計情報として収集する機能[1] を備えており，その統計情報は一般に公開[2] されている．この統計情報によると，ブラウザを開いてから (セッションが復元されてから) 読み込まれるページの数の中央値は延べ 11.17 ページであった．これを一日あたりに閲覧するページ数と見なした場合，1 週間分の既読コンテンツを最短 2.5 秒程度で検出できることが示された．

[1] https://support.mozilla.org/ja/kb/send-performance-data-improve-firefox
[2] https://telemetry.mozilla.org/

第6章

考察

6.1 既読コンテンツの表示について

6.1.1 実装の際に発生し得る問題点

HTML 要素単位で検出を行う場合は，4.1.3 節にて述べたように，算出された類似度に応じて該当する要素の書式を変更することで 3.2 節で述べた表示方法は容易に実装することができた．

しかし，それよりも小さい単位で検出・表示を行おうとした場合，該当する文字列を，それを含む要素の中から指定しなければならない．具体的には，4.1.3 節で述べたように，表示上該当する文字列の元となっている HTML ソース中の文字列を span タグで囲み，その span 要素に対して書式設定を行うこととなる．その際，当該 HTML ソース中に何らかの要素の開始タグが存在するが，それに対する終了タグが当該 HTML ソースの外部にある場合，span タグを挿入することで，HTML 要素の入れ子構造が崩れてしまうことになる．

実際に，5.1.3 節において行った実験では，要素単位の検出では転載先のサイトにおいて転載元コンテンツを検出・表示することに成功したが，句読点単位では検出は行なわれたものの書式設定に失敗している領域が発生していた．例えば，図 5.1 において，

> 全長 40-47 センチメートル [3][5]

という文字列は転載元・転載先の双方に存在し，レーベンシュタイン距離は 0，trigram による文脈ベクトルに対するコサイン類似度は 1 であり，完全に一致していることを示していたが，既読コンテンツとしては表示されなかった．

6.1.2 書式設定のコスト

スタンドアローンモデルによる 5.1.3 節での実験では，特に句読点単位の検出において，書式が設定されるまでに最大で 5.9 秒の待ち時間が発生した．しかし，要素単位の検出では，NCP の除外の有無，および表示方法にかかわらず，2 秒以内に処理が終了した．特に，句読点単位の書式設定を行った際の検出・書式設定を合わせた所要時間は，平均で要素単位の 3.56 倍であった．また，句読点単位の検出であっても，検出のみで書式設定を行わない場合の処理時間は 3 秒以内に収まった．

これは「要素単位の書式設定」が該当する要素の書式を設定するという処理のみで完了するのに対し，「句読点単位」では，該当する部分を記述している HTML 要素を検索し，

書式設定のために span 要素を追加するという高コストな処理を行う必要があるためであると考えられる．

また，クライアントサーバモデルによる実験では，図 5.5, および表 5.6 が示すように，句読点単位の書式設定であっても類似度に応じて書式を段階的に変化させるかどうかにより書式設定の所要時間が大幅に増大することがわかった．該当する処理を記述するソースコードがほとんど同一であるにもかかわらずこのような差異が生じる原因は不明である．

6.1.3　類似度関数の値と実際の類似度

類似度関数の値と実際の文章の類似度は必ずしも線形的ではない．そのため，類似度に応じて書式の値を線形的に変化させると，類似してはいない部分に書式設定が行われ，可読性が損なわれる．

例えば，次の 2 文

> 効果を最大限に引き出すことが大切です
> 眠気を催すことがあるので

間のコサイン類似度は 0.158 であるが，この二つの文に共通する trigram は‘すこと’のみである．

6.2　SimString による既読コンテンツの検出処理について

SimString 以外の処理が十分高速であり処理時間が無視できるものと仮定する．また，SimString が呼び出されてから類似文字列検索を行う準備が整うまでの時間を t_0，一つのクエリ文字列あたりの処理時間を t_1，Web ページに含まれる検出対象 HTML 要素数を n，句読点単位で分割された文字列が一つの要素あたり平均 m 個含まれるものとすると，一つの Web ページにおいて既読コンテンツを検出する (類似度が閾値以上であるか否かを求める) ための所要時間は表 6.1 で与えられる．

	要素単位	句読点単位
線形探索	$nt_1 + t_0$	$mnt_1 + t_0$
二分探索	$n(t_1 + t_0)$	$mn(t_1 + t_0)$

表 6.1: 検出の際の所要時間 (一律)

さらに，類似度に応じた書式設定を行うために類似度を求める場合は，その精度を p とすると，所要時間は表 6.2 で与えられる．(例えば，0.1 から 1 までの 10 段階の場合は，$p = 0.1$ となる．)

	要素単位	句読点単位
線形探索	$\frac{1}{p}(nt_1 + t_0)$	$\frac{1}{p}(mnt_1 + t_0)$
二分探索	$-n(t_1 + t_0)\log_2 p$	$-mn(t_1 + t_0)\log_2 p$

表 6.2: 検出の際の所要時間 (可変)

$t_0 = t_1$ かつ m, n が十分に大きいものと仮定すると，$p \leq 1/4$ ならば二分探索が有利であるが，実際には二分探索を行うことにより所要時間が増大した．

二分探索による類似度算出を実装した段階では $t_0 \ll t_1$ を仮定していたが，実際にはその仮定は誤りであり $t_1 \ll t_0$ であったため，所要時間が増大したものと考えられる．

6.3　既読コンテンツ検出における形態素解析

2.2.1 節における文脈ベクトルとして，N-gram ではなく形態素解析により分割された単語のベクトルを用いる実装の有効性について論ずる．

6.3.1　形態素解析器

Firefox 拡張機能上で形態素解析を行う場合は，Mozilla Firefox 用の拡張機能と同じく JavaScript で実装されており，親和性が高いと考えられる形態素解析器 kuromoji.js[12] を用いることができる．4.1 節の実装を行う過程で kuromoji.js を一部改変の上組み込み，本稿の執筆時点では，任意のページにおいて形態素解析を行い，そのデータを保存することまでが可能となっている．

[12] は，XMLHttpRequest を用いて，形態素解析用の辞書ファイルを外部から非同期的にダウンロードするという仕様となっている．辞書ファイルを設置するサーバの帯域幅の問題から，現状では現在使用中のシステム (localhost) 上で HTTP サーバプログラムを動作させる必要がある．

外部から辞書ファイルをダウンロードする仕様であることから，スタンドアローンモデルによる実装であっても，kuromoji.js を用いる場合はクライアントサーバモデルと同様の事象が発生することが考えられる．

一方，4.3 節のように，外部のサーバ上で記録・検出を行う場合は使用する言語の制約が取り除かれるため，MeCab が利用可能となる．

6.3.2　形態素解析済データの類似度

形態素解析されたデータ間の類似度を算出する方法として，tf-idf 法 [13] による単語の重要度に基づくベクトル空間法がある．

TF(term frequency, 単語出現頻度) とは，ある文章を構成する単語の出現回数の全単語の出現回数に対する比である．文章 d_j における単語 t_i の TF $tf_{i,j}$ は，文章 d_j における単語 t_k の出現回数を $n_{k,j}$ とすると，次式により与えられる．

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

また，IDF(inverse document frequency, 逆文書頻度) は，ある単語を含む文章の全文章の個数に対する割合の逆数の対数をとったものである．単語 t_i の IDF は，総文章数を $|D|$，単語 t_i が出現する文章数を $|\{d : d \ni t_i\}|$ とすると，次式により与えられる．

$$idf_i = \log \frac{|D|}{|\{d : d \ni t_i\}|}$$

tf-idf は，これらの式で与えられる TF と IDF の積である．tf-idf 方法はある文章において繰り返し用いられる単語ほど重要度が高いが，一部の文章のみに含まれる単語の重要度は様々な文章に含まれる単語よりは高くなるという考え方に基づく．

TF は文章ごとにその形態素解析データのみの各単語数および総単語数を数えることにより容易に計算可能である．しかし，IDF の計算には，その都度過去の全解析データを走査する必要があり，非常に高コストである．

そのため，各単語の IDF を MeCab を用いて事前に計算しておき，tf-idf の計算にはそれを参照するという方法を検討していた．しかし，IDF の計算コストが予想以上に高コストであった．元のデータ (コーパス) として日本語版 Wikipedia の記事内容のダンプを用いた場合，実験環境の性能の問題により数日経過しても計算が完了しなかった．

第7章

結論

7.1 まとめ

本研究では，検索エンジンを利用するにあたってしばしば問題となる重複コンテンツに対し，それらのコンテンツを読み飛ばすことを補助する既読コンテンツの検出・表示システムの提案，およびウェブブラウザ用拡張機能による本システムの構築を目指した．また，既読コンテンツの蓄積・検出部分を外部のサーバに移動した場合の性能の変化についての評価を行った．

最初にウェブブラウザ用拡張機能として実装したシステムに対し評価実験を行い，どのような検出方式であっても記事の大部分が他の転載であるページは，一部分のみを引用するサイトに比べて高い類似度が算出されること，および元ページの一部分のみ既読である場合であってもその部分を既読コンテンツとして検出可能であること，既読コンテンツの検出に際して，広告などのページの主題とは外れる部分を除外することで所要時間が平均して2.61%短縮されることが示された．しかし，所要時間の面では，ページ内容や検出単位等の条件により，検出および書式設定の所要時間が数十秒から数分に達することがあるため，実用的なシステムとはいえない．

そこで，拡張機能とサーバプログラムを組み合わせた実装を行い，従来の実装との性能比較を行ったところ，句読点単位での検出時間が従来の実装と比較して0.73%から23.5%程度に短縮された．また，直近1週間程度の既読コンテンツであれば，検出方法を工夫することにより10秒以下で検出が完了し，実用に足ることが示された．しかし，類似度に応じて連続的に書式を設定する処理を句読点単位で行った場合，所要時間がそれ以外のものと比較して極端に大きくなるため，そのような条件下においては未だ実用的ではない．

7.2 今後の課題

7.2.1 コサイン類似度以外の類似度関数

文字列間の類似度を算出するための類似度関数には，本研究で使用したコサイン係数以外にジャッカード係数，ダイス係数，およびオーバーラップ係数等がある．既読コンテンツの検出をこれらの類似度関数を用いて行うことにより，検出結果に有意な差が出る可能性がある．

また，スタンドアローンモデルにおいてはレーベンシュタイン距離による類似度算出

を未だ実装していないため，そのための高速な類似度算出アルゴリズムの実装が必要となる．

7.2.2　多言語対応

本研究においては，記録・検出対象となるページとして，日本語のページのみを想定していた．それ以外の言語への対応をするにあたっては，類似度を算出する元となる特徴量の取り方に再考の余地がある．

具体的には，日本語や中国語のように分かち書きを行わない言語の場合は文字 N-gram が利用できるが，英語のように分かち書きを行う言語の場合は文字単位ではなく，単語間 N-gram により特徴量をとる必要がある．

7.2.3　形態素解析

単に文字 N-gram の類似度を用いて既読コンテンツを検出するだけではなく，形態素解析により分割された単語の類似度を利用することで，より正確な検出が期待できる．しかし，形態素解析されたデータ間の類似度を算出するために単語の重要度を得る方法として知られている tf-idf 法は非常に高コストであるため，一般的なパーソナルコンピュータ上で実行することは現実的ではない．したがって，既に算出済の IDF データを利用することが必要となる．

7.2.4　様々なサイトにおける検証

本研究において記録・検出対象としたページはオンライン百科事典やニュース記事の転載サイト，ある製品に関する複数の記事などごく限られたものであった．より正確な性能評価を行うためには，より幅広い範囲のウェブサイトにおいて性能評価を行う必要がある．

7.2.5　大規模なテスト

本システムの使用感，特に表示手法に関するどれほど見やすかったか・効果的だったか等の評価は，利用者の主観的な評価に頼らなければならない．したがって，正確な性能評価を行うためには，幅広く被験者を募る必要があるものと考えられる．

7.2.6　プライバシー保護

クライアントサーバモデルによる実装において既読コンテンツ蓄積・検出用サーバを外部に配置した場合，本システムはクラウドサービスとなる．しかし，そのような運用を行う場合，既読コンテンツをその都度サーバーにアップロードすることによる帯域幅の圧迫の可能性があることに加えて，「ユーザーの読んだコンテンツ」を外部に送信することによるプライバシーの侵害という懸念がある．

したがって，TLS(SSL) を用いてクライアント-サーバ間の通信が第三者に読み取られないようにする，データベースの内容をユーザー毎に暗号化し，サーバ管理者であったとしてもその内容を読み取れないようにするなど，適切なプライバシー保護手段を講じる必要がある．

謝辞

本論文の執筆にあたり，常日頃から厳しくも優しいご指導，ご鞭撻を賜りました指導教官の相田仁教授に深く感謝致します．

また，古宇田フミ子助教授には常日頃から研究の進め方や生活に関する助言をして頂くなど，研究全体にわたり熱心にご指導して頂きました．心より感謝致します．

日頃の研究活動を支えてくださった矢谷浩司准教授，千葉新吾技官に厚く御礼申し上げます．

秘書の元岡みさ子氏には，事務面でのサポートに限らず，日頃から温かい言葉をかけて下さいました．厚く御礼申し上げます．

研究室での生活を支えてくださった相田研究室の方々にも感謝致します．

そして，これまで私を支えてくださった家族，友人，その他全ての方々にも深く感謝致します．皆様の助けがあって始めて本論文を完成させることができました．本当にありがとうございました．

2017年2月3日
早乙女 高大

参考文献

[1] M.R. Henzinger, R. Motwani, and C. Silverstein, “Challenges in web search engines,” SIGIR Forum, vol.36, no.2, pp.11–22, Sept. 2002. http://doi.acm.org/10.1145/792550.792553

[2] “Personal Blocklist”. https://chrome.google.com/webstore/detail/nolijncfnkgaikbjbdaogikpmpbdcdef.

[3] A. Lieuallen, A. Boodman, and J. Sundstrm, “Greasemonkey”. http://www.greasespot.net/.

[4] J. Biniok, “Tampermonkey”. https://tampermonkey.net/.

[5] J. Scher, “Google Hit Hider by Domain”. https://greasyfork.org/ja/scripts/1682-google-hit-hider-by-domain-search-filter-block-sites.

[6] P. Jobson, “Google Domain Blocker”. https://github.com/pjobson/pjUserscripts.

[7] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen, “Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets,” Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp.525–530, ASIA CCS ’14, ACM, New York, NY, USA, 2014. http://doi.acm.org/10.1145/2590296.2590311

[8] G. Salton, A. Wong, and C.-S. Yang, “A vector space model for automatic indexing,” Communications of the ACM, vol.18, no.11, pp.613–620, 1975.

[9] D. Sankoff and J.B. Kruskal, “Time warps, string edits, and macromolecules: the theory and practice of sequence comparison,” Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B., vol.1, pp.37–39, 1983.

[10] 岡崎直観，辻井潤一，“高速な類似文字列検索アルゴリズム，” 情報処理学会創立50周年記念全国大会，pp.1C–1，2010． http://www.chokkan.org/software/simstring/

[11] T. Kudo, “Mecab: Yet another part-of-speech and morphological analyzer,” 2005. http://mecab.sourceforge.net/.

[12] T. Asano, “kuromoji.js,” 2015. https://github.com/takuyaa/kuromoji.js.

[13] G. Salton and M.J. McGill, Introduction to Modern Information Retrieval, McGraw-Hill, Inc., New York, NY, USA, 1986.

発表文献

早乙女 高大, 相田 仁, “ウェブブラウザにおける既読コンテンツの検出・表示手法の検討, ” 情報処理学会 第 104 回ドキュメントコミュニケーション合同研究発表会 (発表予定)

付録A

実験環境

A.1　スタンドアローンモデル

スタンドアローンモデル (4.1 節) の実行環境は次の通り.
MacBook Pro (Retina, 13-inch, Mid 2014)

CPU　Intel®Core^{TM}i5 @ 2.6 GHz

メモリ　DDR3-1600 8192MB

オペレーティングシステム　OS X Yosemite 10.10.5

A.2　クライアントサーバモデル

スタンドアローンモデル (4.3 節) の実行環境は次の通り.
クライアント: MacBook Pro (スタンドアローンモデルに同じ)
サーバ: 自作機

CPU　Intel®Core^{TM}i7-860 @ 2.8GHz

メモリ　DDR3-1333 8192MB

オペレーティングシステム　Debian GNU/Linux 8 (jessie)