決定木による同音異義語の誤り検出とその修正

執筆者：野澤　洋一 (96T6055H)

指導教官：新納　浩幸

平成12年3月1日

目 次

第1章 序論 **5**

1.1 研究概要 5

1.2 同音異義語問題について 5

1.3 本論文の構成 6

第2章 従来手法 **7**

2.1 決定リスト 7

2.2 同音異義語問題への適用 9

2.2.1 証拠の設定 9

2.2.2 頻度の獲得 9

2.2.3 予測力の算出 9

2.2.4 決定リストの作成 10

第3章 本手法 **11**

3.1 決定木 11

3.2 C4.5 12

3.2.1 C4.5について 12

3.2.2 分割統治法 13

3.2.3 例 13

3.2.4 テストの評価 15

3.2.5 利得基準 16

3.2.6 利得比基準 18

3.2.7 テストの候補 19

3.2.8 連続属性に関するテスト 20

3.3 同音異義語問題への適用 20

3.3.1 決定木の問題点 20

3.3.2 問題適用への対策 21

3.3.3 決定木作成のための手順 21

第4章 実験 **23**

4.1 実験の設定 23

4.2 決定リストによる実験 24

4.2.1 実験手順 24
4.2.2 実験結果 27
4.3 決定木による実験 29
4.3.1 実験手順 29
4.3.2 実験結果 34

第5章 考察 **36**
5.1 決定リストによる誤り判定の結果について 36
5.2 決定木による誤り判定結果について 38
5.3 採用した証拠の個数と正解率の関係について 40

第6章 まとめ **49**

付 録A 日本語EUC文字列⇔ASCII文字列変換テーブル **51**

付 録B 決定木を使用した同音異義語判定プログラムdecideの主要部 **62**

図目次

2.1 決定リストの表現形式例 7

3.1 {債券, 債権}における決定木の例 11
3.2 多数の値をとる決定木の表現形式例 21

4.1 「債券」における判定結果ファイルの例 28
4.2 債券, 債権に対応する名称ファイルの例（採用証拠数300） 30
4.3 図4.2に対応するデータファイルの例 32
4.4 {債券, 債権}においてC4.5の出力した決定木の例 33
4.5 図4.4に対応するif-thenルール 34

5.1 決定リストの改善後の表現形式例 38
5.2 採用した証拠の数を変化させたときの平均正解率の変動 41
5.3 {債券, 債権}における正解率の変動 42
5.4 {解放, 開放}における正解率の変動 42
5.5 {協調, 強調}における正解率の変動 43
5.6 {自信, 自身}における正解率の変動 44
5.7 {感心, 関心}における正解率の変動 44
5.8 {体外, 対外}における正解率の変動 45
5.9 {運航, 運行}における正解率の変動 45
5.10 {同志, 同士}における正解率の変動 46
5.11 {過程, 課程}における正解率の変動 46
5.12 {実効, 実行}における正解率の変動 47
5.13 {食料, 食糧}における正解率の変動 47
5.14 {傷害, 障害}における正解率の変動 48

表目次

2.1 “plant” における決定リストの例 8

3.1 小さな訓練集合の例 14
3.2 事例の最終的な分割と対応する決定木 15

4.1 単語「債権」のトレーニングデータ 24
4.2 単語「債券」に対応する証拠頻度リストの例 25
4.3 {債券, 債権} に対応する決定リストの例 26
4.4 単語「債券」における、ある事例に対して決定リストから得られた判定例 27
4.5 決定リストによる判定の正解率 28
4.6 決定リストと決定木の正解率の比較 35
4.7 属性値のとり得る個数を変化させたときの正解率の変化 35

5.1 最も良い結果の決定リスト（{過程, 課程}） 37
5.2 最も悪い結果の決定リスト（{食料, 食糧}） 37

第1章　序論

1.1　研究概要

日本語文章の記述の際には文法・記述・漢字変換のような部分で誤りが発生する場合がある．それ故，そのような誤りを自動的に校正する，もしくはそれを支援するシステムが有益であるのは明らかである．しかし，文章中の誤りには多種多様な種類やレベルがあり，それらを一様に扱える高機能且つ高精度な文章校正システムの実現は難しい．

そこで本研究では文章中に発生し得る誤りのうち，特に同音異義語の書き誤りのみに対象を絞り，同音異義語の書き誤りを自動検出するための手法を考察するものとする．

ここで同音異義語とは，{債券, 債権}のように同じ平仮名表記を持つ単語の集合であると定義する．また，同音異義語の中から正しい単語を選択する問題を同音異義語問題とする．

同音異義語問題の例
「あの国は現在深刻なしょくりょう問題を抱えている.」
“しょくりょう” ⇒ “食料” or “食糧” or ...?

従来より同音異義語問題を解くために決定リストを用いた手法があるが，その学習できる規則は単純であり，もう少し複雑な規則を扱える手法によって判定の精度を上げられる見込みがある．そこでより複雑なルールの学習が可能な手法として決定木を適用する．

本研究では同音異義語の誤り検出とその修正に決定木を適用し，従来手法である決定リストの結果と比較を行うことで決定木の有効性を検証する．また，決定木を同音異義語問題のような属性値の多数ある問題へ適用する際の対策法を提案する．

1.2　同音異義語問題について

先述のように，同音異義語を同じ平仮名表記を持つ単語の集合であると定義し，同音異義語の中から正しい単語を選択する問題を同音異義語問題とする．

同音異義語問題に対しては，従来，前後の一文字から判別する手法があったが，その判別力には限界がある．また複合語内の同音異義語に対して，前後の文字や単語を調べる手法もあるが，この場合は複合語にない同音異義語には対処できない．また，対象となる単語の回りの語列，品詞列などを手がかりとして，同音異義語問題を解消する試みもあるが，手がかりの質や量の問題のほかに，手がかりの組み合わせ方の問題も残っている．

同音異義語問題における平仮名表記を単語，漢字表記を語義ととらえれば，同音異義語問題は語義選択の問題と等価である．このため，従来より提案されてきた語義選択の問題に対する種々の統計的手法を用いて同音異義語問題を解くことができる [1].

1.3 本論文の構成

本論文では始めに，従来より同音異義語問題を解くのに使用されてきた手法の説明をする（第２章）．その後に本研究で使用する手法である決定木の説明を行い（第３章），各手法を同音異義語問題に適用した実験の解説と結果を示し（第４章），そしてその結果に対する考察を行う（第５章）．

また、付属として実験に使用したプログラムのソースの中から主要と思われる部分の中で、日本語 EUC 文字列と ASCII 文字列を相互に変換するテーブルの実装部と、決定木を判定するためのクラスライブラリの実装部を載せた．

第2章　従来手法

ここでは従来より使用されてきた手法である決定リストについての説明を行う.

2.1　決定リスト

決定リストとは従来より語義選択問題に適用されてきたクラス分類手法の一つである.

特定の辞書に基づいて意味タグ（語義）が付与されたコーパスであれば，品詞付けと同様の手法により語義付与が可能かもしれないが，意味タグつきコーパスを作成するコストは非常に大きい．そこで，Yarowskyは，語義が付与された少数の初期データ（seed collocation）から，教師なし学習により語義選択のための確率モデルを作製する方法を提案した [4]．その確率モデルが決定リストである.

決定リストは図2.1に示すようなif-thenのルールを学習する手法といえる.

```
if (前の自立語 == "共同") return 債権;
if (前の自立語 == "株式") return 債券;
if (後ろの自立語 == "問題") return 債権;
...
return 債権;
```

図 2.1: 決定リストの表現形式例

図2.1から分かるように，決定リストの規則の表現形式は容易であり，直感的で分かりやすい構造をしている．語義の選択に影響を与えるための証拠（evidence）がリスト状に並んでおり，その順番は証拠の強さ（予測力）の順番で降順にソートされたものである．証拠としては，様々な単語共起の形（単語・品詞及びその位置関係）が考慮され，予測力はその証拠 $evidence_i$ のもとで，語義 $sense_a$ と語義 $sense_b$ が選ばれる確率との対数尤度比で表される.

$$\log\left(\frac{P(sense_a|evidence_i)}{P(sense_b|evidence_i)}\right)$$

テキスト中の単語の語義は，文脈中に存在する様々な証拠から総合的に判断するのではなく，最も予測力が高い一つの証拠に基づいて決定される．例えば表2.1に英単語 “plant” の語義選択に関する決定リストの例を示す．

表 2.1: “plant” における決定リストの例

共起単語（証拠）	予測力	語義
plant growth	10.12	LIVING
car（ $\pm k$ 語以内）	9.68	FACTORY
plant height	9.64	LIVING
union（ $\pm k$ 語以内）	9.61	FACTORY
equipment（ $\pm k$ 語以内）	9.54	FACTORY
assembly *plant*	9.68	FACTORY
nuclear *plant*	9.68	FACTORY
...	...	...

英単語 “plant” には「植物（LIBING）」と「工場（FACTORY）」という2つの意味がある．この決定リストでは，“growth” や “height” という語が “plant” の直後にあれば「植物」を意味し，“car” や “union” が周辺（ $\pm k$ 語以内）にあれば「工場」を意味するとしている．

2.2 同音異義語問題への適用

2.2.1 証拠の設定

決定リストを作成するために，文脈情報（証拠）を設定する．本研究では証拠として以下の3つの文脈情報を設定した．なお，実験で使用されるトレーニングデータ及びテストデータには前提として日本語2バイト文字列以外は出現しないものとし，実際の単語分割には形態素解析プログラム JUMAN を利用した．

- 直前の自立語 W：aW と表記する．
- 直後の自立語 W：bW と表記する．
- 前後に現れる自立語 W：近いものから最大3つずつ取り出し，cW と表記する．

2.2.2 頻度の獲得

証拠 evd_j が存在しているときに同音異義語 $\{w_1, w_2, \cdots, w_n\}$ 内の単語 w_i が出現している頻度 $freq(w_1, evd_j)$ をコーパスから得る．

具体的な例を示す．例えば，同音異義語を { 食料, 食糧 } とし，以下の2つの例文を見る．

例文1 「明日に備えて食料の買い出しに行かなければならない.」

例文2 「食糧問題について新聞各紙で取り上げられていた.」

例文1からは「食料」に対する証拠として，

“a 備えて”, “b 買い”, “c 明日”, “c 備えて”, “c 買い”, “c 出し”, “c 行か”

が取り出される．また，例文2からは「食糧」に対する証拠として，

”b 問題”, ”c 問題”, ”c ついて”, ”c 新聞”

が取り出される．以上のように証拠を取り出して加算し，最終的に頻度 $freq(w_1, evd_j)$ を求める．

2.2.3 予測力の算出

最終的な決定リストを求めるには，上記の頻度 $freq(w_1, evd_j)$ をもとに予測力を求め，予測力の強い順にソートする．

証拠 evd_j が生じている場合に，単語が w_i である予測力 $est(w_i|evd_j)$ を対数尤度比を用いて以下のように定義する．

$$est(w_i|evd_j) = \log\left(\frac{P(w_i|evd_j)}{\sum_{k\neq i} P(w_k|evd_j)}\right)$$

ここで，証拠 evd_j のもとで自立語 w_i が選ばれる確率 $P(w_i|evd_j)$ を以下のように近似する．

$$P(w_i|evd_j) = \frac{freq(w_i, evd_j) + \alpha}{\sum_{k=1} freq(w_k, evd_j) + \alpha}$$

上式の α は， $freq(w_i, evd_j) = 0$ の場合に起る 0 除算の不具合を回避するために設定した．本研究では $\alpha = 0.1$ とする．

2.2.4 決定リストの作成

ある証拠 evd_j が発生しているときに同音異義語 $\{w_1, w_2, ..., w_j\}$ の中から最も適したものを選択するには，$est(w_1, evd_j)$, $est(w_2, evd_j)$, ..., $est(w_j, evd_j)$ の中で最も値の大きな $est(w_k, evd_j)$ を採用すればよい．そして単語 w_k を証拠 evd_j が現れたときの解答とする．

実際の決定リスト作成時には各 evd_j に対して， evd_j が現れたときの解答 w_{k_j} を求め，予測力 $est(w_{k_j}, evd_j)$ が高い順のリストを作成する．これが決定リストとなる．

以上で決定リストを求めることができるが，あまりに弱い予測力をもつ事項をリストに含めても，実際にそれが正しい選択を示しているかどうかは疑わしい．そこでそのような無意味な事項をリストから除外するために特別な証拠 $default$ を設定し，予測力に閾値を設けることにする．

具体的には $freq(w_i, default)$ を w_i の総頻度とすれば，この総頻度から得られる予測力は

$$P(w_i|evd_j) = \frac{freq(w_i, evd_j) + \alpha}{\sum_{k=1} freq(w_k, evd_j) + \alpha}$$

と表すことができる．そこで，同音異義語の各単語における総頻度の予測力 $est(w_1, default)$, $est(w_2, default)$, ..., $est(w_j|default)$ の中から最も大きい値である $est(w_k, default)$ を取り出せば，それがその決定リストにおける閾値となる．当然ながらこの事項はリストの最終に置かれ，リストの上端より一致する証拠を探して見つからなかった場合は， w_k が解答として得られる．

第3章　本手法

3.1　決定木

決定リストは文脈上の1つの情報からクラスを判定しており，規則の形としては非常に単純である．そのためもう少し複雑な規則を学習できる決定木を利用した方が精度が上がることが期待できる．

決定木もクラス分類手法の一種であり，図3.1のようなif-thenのルールを学習する手法と捉えられる[2]．図2.1と比較するとif-thenのルールの中にif-thenのルールを入れ子に記述できる形であり，決定リストよりも複雑な記述形式を提供していることがわかる．

```
if (自立語"日本" ∈ 証拠の集合) {
  if (後ろの自立語 == "問題") return 債権;
  else return 債券;
} else if (自立語"信用" ∈ 証拠の集合) {
  if (自立語"共同" ∈ 証拠の集合) {
  if (自立語"責任" ∈ 証拠の集合) {
    ...
   }
   ...
}  else return 債券;
   ...
} else return 債権;
```

図 3.1: {債券, 債権}における決定木の例

3.2 C4.5

3.2.1 C4.5について

決定木を同音異義語問題に適用する際に必要な決定木作成ソフトウェアには，本研究ではパッケージソフト C4.5[3] を採用した．

AI手法である C4.5 は，記録された膨大な分類データを調べ，特定の例を一般化することによりモデルを帰納的に作る方法の1つで，記録の中にあるパターンを見つけて解析することによって，2番目の分類モデルを作るコンピュータプログラムである．

- 属性 — 値の記述1つのオブジェクトあるいは事例 (case) に関するすべての情報は，あらかじめ決められた性質あるいは属性 (attributes) によって表現できなくてはならない．それぞれの属性は離散値をとる．しかし，ある事例を表現するために使われた属性のタイプは，別の事例で別なタイプとして扱われてはならない．この制約のため，オブジェクトが変化するような構造を持つような問題領域は扱うことができない．

- まえもって定義されたクラス

  事例が割り当てられるべきカテゴリはまえもって準備されていなくてはならない．これは，機械学習における教師付き (supervised) 学習に対応し，解析によって事例の適当なグループを見つける教師なし学習とは対照的である．

- 離散クラス

  クラスは，事例がそのクラスに属するか属さないかを決めることができるように，はっきりと定義できる必要がある．さらに，事例の数はクラスの数より十分多い必要がある．

- データが十分あること

  帰納的一般化は，データの中から同じようなパターンを見つけることによって行われる．もし，たまたま，はっきりと区別できるパターンを見つけることができなければ，この方法は失敗してしまう．この区別は一般に統計的テストによって行っているので，このテストが意味があるためには十分な事例が必要である．必要となるデータの数は，属性やクラスの数と分類しようとするモデルの複雑さなどの要因から決まる．これらの要因が増えると，信頼性のあるモデルを作るために必要なデータの数は増える．簡単なモデルであれば，小数のデータから作ることができる．しかし，複雑な分類モデルを作るには，一般に数百から数千のトレーニング事例が必要となる．

- “論理的” 分類モデル

C4.5は，決定木あるいはプロダクションルールの集合を作ることができるのみである．1つのクラスは，属性の値に関する記述を集めた1つの論理的な表現で表すように制限されている．この制限を満足していない分類モデルの一般的な形の1つとして，線形判別 (linear discriminant) がある．線形判別では，属性からの寄与を重み付けしてから加え，閾値と比較する．クラスの記述は論理的というより算術的である．

このプログラムのアルゴリズムを以下に示す．

3.2.2 分割統治法

訓練事例の集合 T から決定木を構成する．クラスを $\{C_1, C_2, ..., C_k\}$ と表す．Tがどのような事例になるかにより，以下の 3 つの可能性が考えられる．

- T は少なくとも1つ以上の事例を含み，しかも，そのすべての事例が1つのクラス C_j に属する場合:

  この場合は，T に対する決定木は1つの葉だけからなり，それにクラス C_j のラベルを付与する．

- T が全く事例を含まない場合:

  この場合も決定木は1つの葉からなるが，それに付与すべきクラス名は T 以外の情報に基づいて決めなければならない．例えば，問題領域の知識に基づいて，最も頻繁に現れるクラスを選ぶことができる．C4.5 では，事例が全く与えられなかった葉に付与すべきクラスとして，その親ノードの中で最も頻繁に現れたクラスを用いる．

- T が種々のクラスに属する事例を含む場合:

  この場合は，T を部分集合に分割して，各部分集合ができるだけ単一のクラスに属するように改善する．例えば T 10 というテストが選ばれたとして，それは1つの属性に基づいて，相異なる値 $\{ O_1, O_2, ..., O_n \}$ を出力とするとしよう．このとき，テスト結果が O_i となるような T 内の事例の集合を T_i とすれば，T は部分集合 $T_1, T_2, ..., T_n$ に分割される．T に対する決定木は，テスト T 10 を行う決定ノードと各出力値に対応する枝からなり，同様な手順がさらに再帰的に繰り返される．すなわち，その i 番目の枝の先には，訓練事例の部分集合 T_i に基づいて同様な手順で構成された決定木がつながれる．

3.2.3 例

訓練事例の集合の分割は，すべての部分集合が単一のクラスに属するようになるまで繰り返される．このプロセスを説明するために一例を表3.1に示す．この小

さな訓練集合では，4種の属性と2つのクラスがある．

これらの事例は単一のクラスに属しているわけではないので，分割統治法によりそれらの分割を試みる．ここで”天候”のテストを選んだとしよう．その出力値は，”晴れ”と”曇り””雨”の3通りある．”曇り”のグループはすべて開催のクラスからなるが，”晴れ”と”雨”のグループでは複数のクラスが混在している．そこで，さらに”晴れ”のグループを湿度が75また”雨”のグループを強風か否かのテストで分割すれば，その結果得られる各グループはすべて単一のクラスからなるようにできる．こうして最終的に得られた分割の様子と，それに対応する決定木を表3.2に示す．

表 3.1: 小さな訓練集合の例

天候	温度 ($^\circ F$)	湿度 (%)	強風?	クラス
晴れ	75	70	真	開催
晴れ	80	90	真	中止
晴れ	85	85	偽	中止
晴れ	72	95	偽	中止
晴れ	69	70	偽	開催
曇り	72	90	真	開催
曇り	83	78	偽	開催
曇り	64	65	真	開催
曇り	81	75	偽	開催
雨	71	80	真	中止
雨	65	70	真	中止
雨	75	80	偽	開催
雨	68	80	偽	開催
雨	70	96	偽	開催

表 3.2: 事例の最終的な分割と対応する決定木

事例の分割：

- 天候 = 晴れ：
  - 湿度 ≤ 75：

天候	温度 ($^\circ F$)	湿度 (%)	強風?	クラス
晴れ	75	70	真	開催
晴れ	69	70	偽	開催

  - 湿度 > 75：

天候	温度 ($^\circ F$)	湿度 (%)	強風?	クラス
晴れ	80	90	真	中止
晴れ	85	85	偽	中止
晴れ	72	95	偽	中止

- 天候 = 曇り：

天候	温度 ($^\circ F$)	湿度 (%)	強風?	クラス
曇り	72	90	真	開催
曇り	83	78	偽	開催
曇り	64	65	真	開催
曇り	81	75	偽	開催

- 天候 = 雨：
  - 強風 = 真：

天候	温度 ($^\circ F$)	湿度 (%)	強風?	クラス
雨	71	80	真	中止
雨	65	70	真	中止

  - 強風 = 偽：

天候	温度 ($^\circ F$)	湿度 (%)	強風?	クラス
雨	75	80	偽	開催
雨	68	80	偽	開催
雨	70	96	偽	開催

3.2.4 テストの評価

各ノードで T を分割するテストをどのように選んでも，それが真の意味での分割であれば，最終的には単一のクラスからなる部分集合への分割が得られる．こ

こで，真の意味での分割とは，空にならない部分集合 $\{ T_i \}$ が少なくとも2つ以上できるような分割のことである．もっとも，そのほとんどすべての部分集合がただ1つの訓練事例しか含まないことがあるかもしれない．しかし，決定木を構成する目的は，単にそのような分割を何でもよいから見つけることでなく，問題領域の構造を明らかにしたり予測能力を獲得したりすることにある．そのためには，一つ一つの葉において十分な数の事例が必要であり，したがって，あまり細かく分割しすぎないことが望ましい．理想を言えば，最終的に小さな木が得られるように各段階でテストを選びたい．

3.2.5 利得基準

事例の集合 S に対して，$freq(C_i, S)$ は S の中でクラス C_i に属する事例の数を表す．また，集合 S に含まれる事例数を $|S|$ と表す．

事例の集合 S からランダムに1つの事例を選びだし，それがクラス C_j に属していると知らせたとする．このメッセージの確立は，

$$\frac{frec(C_j, S)}{|S|}$$

であり，それが伝える情報量は，

$$-\log_2 \left(\frac{frec(C_j, S)}{|S|} \right) \text{ビット}$$

となる．このようなクラスの所属関係に関するメッセージの平均情報量を求めるために，S 内での頻度で重み付けしてクラス全体に対する平均を求めると，

$$info(S) = -\sum_{j=1}^{k} \frac{frec(C_j, S)}{|S|} \times \log_2 \left(\frac{frec(C_j, S)}{|S|} \right) \text{ビット}$$

を得る．この量は集合 S のエントロピーとも呼ばれる．これを訓練事例の集合 T に適用すれば，$info(T)$ は T 内のある1つの事例が属するクラスを同定するのに必要な情報量の平均値となる．

さて，テスト X の n 通りの結果に合わせて T が分割された後について，同様な評価を考えよう．このときクラスを同定するのに必要な情報量の期待値は，部分集合上で荷重平均をとって，

$$info_X(T) = \sum_{j=1}^{n} \frac{|T_i|}{|T|} \times info(T_i)$$

となる．これらの差

$$gain(X) = info(T) - info_X(T)$$

は，テスト X で T を分割することによって獲得される情報量を表す．この情報量の利得は，テスト X とクラスとの相互情報量とも呼ばれる．これを最大にするようにテストを選ぶ基準を，**利得基準**と呼ぶ．

具体例として，表3.1の訓練集合を再び使用する．ここには，9事例の”開催”と5事例の”中止”の2つのクラスがあり，

$$info(T) = -\frac{9}{14} \times \log_2 \frac{9}{14} - \frac{5}{14} \times \log_2 \frac{5}{14} = 0.940 \text{ビット}$$

である (これは，T 内の1事例のクラスを同定するために必要な情報量の期待値を表す)．また，”天候”を用いて T を3つの部分集合に分割した後の結果は，

$$
\begin{aligned}
info_X(T) &= \frac{5}{14} \times (-\frac{2}{5} \times \log_2 \frac{2}{5} - \frac{3}{5} \times \log_2 \frac{3}{5}) \\
&\quad + \frac{4}{14} \times (-\frac{4}{4} \times \log_2 \frac{4}{4} - \frac{0}{4} \times \log_2 \frac{0}{4}) \\
&\quad + \frac{5}{14} \times (-\frac{3}{5} \times \log_2 \frac{3}{5} - \frac{2}{5} \times \log_2 \frac{2}{5}) \\
&= 0.694 \text{ビット}
\end{aligned}
$$

したがって，このテストによる情報量の利得は，$0.940 - 0.694 = 0.246$ ビットとなる．さて，天候の代わりに強風か否かの属性によって T を分割したとしよう．そうすれば，開催3事例と中止3事例，開催6事例と中止2事例からなる2つの部分集合が得られたであろう．同様な計算により，

$$
\begin{aligned}
info_X(T) &= \frac{6}{14} \times (-\frac{3}{6} \times \log_2 \frac{3}{6} - \frac{3}{6} \times \log_2 \frac{3}{6}) \\
&\quad + \frac{8}{14} \times (-\frac{6}{8} \times \log_2 \frac{6}{8} - \frac{2}{8} \times \log_2 \frac{2}{8}) \\
&= 0.892 \text{ビット}
\end{aligned}
$$

で，利得は0.048ビットとなり，これは先のテストで得られる利得より少ない．したがって，利得基準は，強風か否かの後者のテストよりも，天候についての前者のテストを選ぶことになる．

3.2.6 利得比基準

利得基準は，多数の値を取るテストを偏重する欠陥を持つ．このことは，患者の身分証明を属性として用いるような仮想的な医療診断タスクを考えてみればわかる．身分証明は患者一人一人を識別するためのものであるから，これを属性として用いて訓練事例集合を分割すれば，1事例だけからなるたくさんの部分集合が得られることになる．当然のことながら，これらの1事例からなる部分集合は単一のクラスに含まれるので，$info_x(T) = 0$ となる．したがって，この属性を用いて訓練事例の集合を分割すれば，情報量の利得は最大になる．しかしながら，クラスの予測能力を獲得しようという観点からは，このような分割は全く無益である．

利得基準に伴うこのような偏重は，多数の値を取ることによって得られた利得部分を調整することによって，矯正することができる．ある事例に関して，それがどのクラスに属するかではなく，そのテスト結果自体を伝えるメッセージの情報量を考える．$info(S)$ の定義からの類推により，分割情報量を

$$
split\ info(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times \log_2 \left(\frac{|T_i|}{|T|} \right)
$$

と定める．これは T を n 個の部分集合へ分割することによって得られる全情報量を表す．一方，情報量利得は，そのうちのクラス分けにかかわる部分の情報量を表す．したがって，利得比

$$gain\ ratio(X) = \frac{gain(X)}{split\ info(X)}$$

は，分割によって得られる情報量のうち，有益な部分，すなわち，クラス分類に役立つ部分の割合を表す．ところで，分割が自明な分割に近いときは，分割情報量の値が小さいために利得比の値が不安定になる．そこで，**利得比基準**では，全テスト中で情報量利得が少なくとも平均以上であるという制約下でこの利得比を最大にするテストを選ぶ．

明らかに，この評価基準の下では，患者の身分証明という属性が高く評価されることはない．前述のように，クラスの数を k とすると上式の分子 (情報量利得) は高々 $\log_2(k)$ となる．一方，訓練事例数を n とすれば，テスト結果も n 通りに分割されるので，分母は $\log_2(n)$ となる．

ここで，訓練事例数 n はクラス数 k よりずっと大きいので，利得比は小さな値となる．

前述の例の議論に戻る．天候に関するテストは，訓練事例を 5 個，4 個，5 個の部分集合に分割する．この分割情報は，

$$-\frac{5}{14} \times \log_2 \frac{5}{14} - \frac{4}{14} \times \log_2 \frac{4}{14} - \frac{5}{14} \times \log_2 \frac{5}{14}$$

すなわち，1.577 ビットと計算される．また，前述したように，利得は 0.246 であるので，利得比は $\frac{0.246}{1.577} = 0.156$ となる．

3.2.7 テストの候補

テストの候補の集合が与えられれば，評価基準により各候補を評価して，その中で最良のテストを選び出すことができる．

通常，分類器を構成するシステムでは，テストの形式を定めて，その形式で表されるすべてのテストを調べる．1つのテストでは1つの属性だけを扱うのが普通である．なぜならば，それによって木が理解しやすくなるし，また，複数の属性を一度に扱ったときに起こるような組み合わせ爆発の問題が避けられるからである．

C4.5 は，次の 3 種類のテストを生成する仕組みを持っている．

- 離散的な属性に関する "標準的" なテスト．各属性値をそのまま出力値として，1つの枝に対応させる．
- 離散的な属性に基づく，より複雑なテスト．属性値をいくつかにグループ分けして出力値を割り当てる．

- 連続値を取る属性 A に対して，閾値 Z との比較により，$A \leq Z$ または $A > Z$ に分割するテスト．

これらのテストは，訓練事例を分割する際の利得比に基づいて統一的に評価される．また，自明に近い分割を避けるために，『分割の際，少なくとも2つの部分集合 T_i はある最小限の数の事例を含まねばならない』という制約を加えることは有益である．このような制約は，特に訓練集合 T が小さいときに効果的である．

3.2.8 連続属性に関するテスト

まず，考えている属性 A の値によって訓令事例集合 T をソートする．その値は有限個しかないので，それらを大きさの順に並べて $\{v_1, v_2, ..., v_m\}$ と表そう．v_i と v_{i+1} の間にある閾値はどれも同じ効果を持ち，属性 A の値が $\{v_1, v_2, ..., v_i\}$ に含まれるか $\{v_{i+1}, v_{i+2}, ..., v_m\}$ に含まれるかにより事例を分割する．したがって，A に基づく分割法は $m-1$ 通りしかなく，そのすべてを調べることは可能である．

通常，閾値として各区間の中点を選ぶ．したがって，i 番目の閾値は，

$$\frac{v_i + v_{i+1}}{2}$$

となる．

C4.5は，閾値として中点そのものを選ばずに，訓練事例集合上で A が取り得る値の中でその中点を越えない最大のものを選ぶ．こうすることによって，木や規則で用いられる閾値は，実際にデータ中に現れている“意味ある”値であると保証できる．

3.3 同音異義語問題への適用

ここでは決定木を同音異義語問題へ適用する際に発生する問題と，それを解消するために工夫した対策法を述べる．

3.3.1 決定木の問題点

同音異義語の誤り検出とその修正の問題に決定木を利用する場合，属性が非常に多くの値を取るという問題がある．例えば目的の同音異義語を判定するための証拠として「前の自立語」という属性を考える．この場合，この属性がとり得る値の数，つまり自立語の種類というのは多数出現することが容易に予想できる．こうした状況では少数の事例しか持たない多数の分割が起こり，分割情報量は小さくなる．分割情報量の値が小さくなると，*gain* の値が少し違うだけで *gain ratio* の値が大きく変動することになる．結果的に *gain ratio* の値が不安定になりやす

くなり，決定木作成上の要である利得比基準の効果が信用できないため，適正な決定木を作成できないことになる．

また，そのように属性が多数の値をとる状況下での決定木は，図2の if-then ルールの形で述べれば，図3.2のような if-then の形が生じることに対応している．

```
if (前の自立語 == 単語1) {
  return 債券;
} else if (前の自立語 == 単語2) {
  ...
} else if (前の自立語 == 単語100000) {
  return 債券;
} else return 債権;
```

図 3.2: 多数の値をとる決定木の表現形式例

図2と図3.2を比較すれば,「決定リストよりも複雑なルールを学習できる」という決定木の利点を失ってしまっていることが分かる．

3.3.2 問題適用への対策

本研究の目的は，上記の問題に対処した決定木を利用することで，決定リストよりも精度の良い同音異義語の誤り検出とその修正を行うことである．そのためには属性の値の種類が多数存在する状況でも対応できるような工夫が必要である．

今回の実験では属性の値の種類を押さえるために，多数の属性の値の中から頻出するものを選出し，それら以外の値を other という1つの値にまとめる方法をとることにした．

3.3.3 決定木作成のための手順

- 決定木で同音異義語問題を解くための文脈情報を設定する．

  ここでは前述の決定リストと同じ設定を用いる．

- 同音異義語 $\{w_1, w_2, ..., w_j\}$ 内の単語 w_i と証拠 evd_j とが共起する頻度 $freq(w_i, evd_j)$ をコーパスから得る．

  これも前述の決定リストと同じものが使えるため，流用することができる．

- 出現する証拠の中から頻出するものを選出する．

頻出する証拠を上位から一定個数だけ採用する．結果の同音異義語とそれに対する，コーパスから得た証拠列とを記述したファイルより各証拠の出現頻度を計算し，その頻度順にソートしたものを用意する．

第4章　実験

本研究で行った実験について説明する．

4.1　実験の設定

まず同音異義語として，誤りやすいと思われる以下の12組を用意した．

{債券，債権}，{解放，開放}，{協調，強調}，
{自信，自身}，{感心，関心}，{体外，対外}，
{運航，運行}，{同志，同士}，{過程，課程}，
{実効，実行}，{食料，食糧}，{傷害，障害}

実験に利用するトレーニングデータの作成には'94年度毎日新聞を，テストデータの作成には'91年度日経新聞を使用する．トレーニングデータおよびテストデータの種類は今回用意した同音異義語の単語に対応する分だけ作成するので，それぞれ２４個分あることになる．テストデータで生じている同音異義語は表記が全て正しいと仮定して同音異義語の判定を行う．

証拠として採用する情報にはいろいろ考えられるが，今回は「自立語の表記」と「同音異義語に対するその自立語の出現位置」という情報を設定した．また出現位置については際限なく範囲をとっても意味がないので，「同音異義語に対する自立語の位置 ±3」という範囲をもうけた．

トレーニングデータとテストデータの記述形式は同じものを採用した．その記述形式は以下のようなものである．

直前の自立語　直後の自立語　±3 語以内の自立語１　±3 語以内の自立語２
...　±3 語以内の自立語６

これが一つ当りの同音異義語に対して抜き出した，証拠となる自立語の集合である．また，±3 語以内の位置に対して自立語が出現しなかった場合，その位置には “NULL” を置いて自立語の代用とする．

実際に同音異義語｛債券，債権｝の「債券」に対するトレーニングデータの例を図4.1に示す．

コーパスから証拠を抜き出すために形態素解析プログラム JUMAN を利用した．また，JUMAN が解析した分割結果は全て正しいと仮定する．

表 4.1: 単語「債権」のトレーニングデータ

偽造	が	偽造	ワリコー	金融債	が	や	都
ロンバードレート	貸付	ロンバードレート	%	５・７５	貸付	同	６・７５
背景	の	背景	回復	景気	の	を	受けて
根拠	動き	根拠	NULL	NULL	動き	だ	NULL
こと	が	こと	決めた	売りオペ	が	十四	,
しかし	の	しかし	NULL	NULL	の	急落	連れて
ニューヨーク	市場	ニューヨーク	月	二	市場	総額	約
米国	で	米国	政府	中国	で	を	発行
最近	格付け	最近	ウォール街	ただ	格付け	,	情報
ため	の	ため	いる	なって	の	大	暴落
短期	表面	短期	程度	年	表面	は	次第に
こと	価格	こと	いう	する	価格	が	なる
...	...	...	...	...	...	...	...

4.2 決定リストによる実験

ここでは実際に本研究で行った，従来手法である決定リストによる実験の手順を説明し，その結果を示す．

4.2.1 実験手順

STEP1 証拠頻度ファイルの作成

トレーニングデータから決定リスト作成に必要な対数尤度比を計算するために，その前準備として証拠頻度を計算し，その結果を記述したファイルを作成する．また，前述のように

- 直前の自立語 W：aW
- 直後の自立語 W：bW
- 前後 ±3 語以内に現れる自立語 W：cW

と，表記に “a,b,c” のいずれかの位置情報を付与する．

実際に同音異義語 {債券，債権} の「債券」に対応するトレーニングデータから，位置情報を付与した証拠の頻度を計算した結果の例を表 4.2 に示す．ここでは出現した証拠の種類を確認しやすくするために，位置情報を含めた表記の順にソートするものとする．

STEP2 決定リストファイルの作成

次に，先ほど作成した証拠頻度リストを記述したファイルから対数尤度比を利用した予測力を算出し，その予測力の順にソートして決定リストを作成する．

表 4.2: 単語「債券」に対応する証拠頻度リストの例

証拠	出現頻度
a あおり	1 個
a あと	1 個
a ある	3 個
a いったん	2 個
a いる	1 個
a おり	1 個
a きっかけ	3 個
a こと	4 個
...	...
c 連合	1 個
c 連動	1 個
c 六	1 個
c 六・〇	1 個

例えば，同音異義語{債券,債権}については以下のようになる．まず{債券,債権}に対応する証拠頻度リストのファイルを開き，{債券,債権}の総頻度を調べるためにそれぞれの頻度の総計を調べる．ここで「債券」及び「債権」の証拠の頻度総計がそれぞれ 1808，3819 であったとすると，証拠 default が出現したときのそれぞれの同音異義語の条件付き確率は，近似式を利用して

$$P(\text{債券}|default) = \frac{freq(\text{債券}|default) + \alpha}{freq(\text{債権}|default) + freq(\text{債券}|default) + \alpha} = \frac{1808 + 0.1}{5627 + 0.1} = 0.3213$$

$$P(\text{債権}|default) = \frac{freq(\text{債権}|default) + \alpha}{freq(\text{債権}|default) + freq(\text{債券}|default) + \alpha} = \frac{3819 + 0.1}{5627 + 0.1} = 0.6787$$

となることがわかる．次に，それぞれの予測力を計算すると，

$$est(\text{債券}|default) = \log\left(\frac{P(\text{債券}|default)}{P(\text{債権}|default)}\right) = \log\left(\frac{0.3213}{0.6787}\right) = -1.0789$$

$$est(\text{債権}|default) = \log\left(\frac{P(\text{債権}|default)}{P(\text{債券}|default)}\right) = \log\left(\frac{0.6787}{0.3213}\right) = 1.0522$$

となる．ここで予測力が最大になるものをその証拠の予測力とし，そのときの同音異義語の表記を証拠に対する解答とする．ここでは証拠 default に対して予測力 1.0522，解答「債権」が得られたことになる．

総頻度の予測力の計算が終わったら，一般の証拠の予測力の計算に移る．算出方法は総頻度と同様である．例えば証拠「c もの」に対し,「債券」の事例が 5,「債権」の事例が 2 だったとすれば，証拠「c もの」が出現したときの条件付き確率はそれぞれ

$$P(\text{債券}\,|\text{c もの}) = \frac{freq(\text{債券}\,|\text{c もの}) + \alpha}{freq(\text{債権}\,|\text{c もの}) + freq(\text{債券}\,|\text{c もの}) + \alpha} = \frac{5 + 0.1}{7 + 0.1} = 0.7183$$

$$P(\text{債権}\,|\text{c もの}) = \frac{freq(\text{債権}\,|\text{c もの}) + \alpha}{freq(\text{債権}\,|\text{c もの}) + freq(\text{債券}\,|\text{c もの}) + \alpha} = \frac{2 + 0.1}{7 + 0.1} = 0.2958$$

となり，それぞれの予測力は

$$est(\text{債券}\,|\text{c もの}) = \log\left(\frac{P(\text{債券}\,|\text{c もの})}{P(\text{債権}\,|\text{c もの})}\right) = \log\left(\frac{0.7183}{0.2958}\right) = 1.2800$$

$$est(\text{債権}\,|\text{c もの}) = \log\left(\frac{P(\text{債権}\,|\text{c もの})}{P(\text{債券}\,|\text{c もの})}\right) = \log\left(\frac{0.2958}{0.7183}\right) = -1.2800$$

となる．これより結果的に証拠「c もの」の予測力は 1.2800 で，解答は「債券」であることが得られる．

このとき，先に算出した default の予測力を下回る証拠については，意味のないリスト要素として決定リストには挿入しない．

すべての証拠について予測力の計算とリストへの挿入が終わったら，今度は予測力について降順にソートする．これが最終的な決定リストとなる．{ 債券, 債権 } 予測力の順にソートした決定リストの例を表 4.3 示す．

表 4.3: { 債券, 債権 } に対応する決定リストの例

順位	証拠	債券	債権	解答	予測力
1	c 処理	0	54	債権	9.0795
2	b 処理	0	37	債権	8.5353
3	c 債権	0	33	債権	8.3707
4	c 不良	1	313	債権	8.1530
5	a 不良	1	304	債権	8.1109
6	c 多額の	0	27	債権	8.0822
7	c 額	0	26	債権	8.0279
8	c 償却	0	26	債権	8.0279
...	...	...	...	...	...
1194	default	1808	3819	債権	1.0522

STEP3 誤り判定

決定リストが作成されたら，テストデータから各同音異義語について誤り判定を行うことができる．テストデータもトレーニングデータと同様の形式で，一つの同音異義語に証拠が

直前の自立語　直後の自立語　±3 語以内の自立語 1　±3 語以内の自立語 2
... ±3 語以内の自立語 6

と並べて記述されている．これらの証拠を使って，先ほど作成した決定リストから各証拠に対応する予測力とその解答を調べることで誤り判定を行う．

例えば，単語「債券」のテストデータにおいて，次のような証拠列を得たとする．

もっとも の もっとも NULL NULL の 上昇 小幅

これに位置情報を付与した表記を使って，{債券,債権}に対応する決定リストの上位から一致するものを探索する．その結果，以下の表4.4が得られたとする．表

表 4.4: 単語「債券」における、ある事例に対して決定リストから得られた判定例

証拠	予測力	解答	正否
aもっとも	1.0522(default)	債権	×
bの	1.7551	債権	×
cもっとも	1.0522(default)	債権	×
cの	1.4769	債権	×
cNULL	1.0522(default)	債権	×
c上昇	5.6725	債券	○
c小幅	1.0522(default)	債権	×

4.4からわかるように，この事例では最も予測力の大きい証拠「c上昇」の解答に従う．つまりここでの同音異義語の表記は「債券」が正しいため，この事例は正解となる．このように判定して出力した判定結果ファイルの例を図4.1に示す．

このような判定を各同音異義語のテストデータ対して行い，同音異義語の事例数に対してどれだけ正解数を出したかを正解率として計算する．

4.2.2 実験結果

今回テストデータ作成に使用した'94年度毎日新聞では，同音異義語は平均494組現われていた．また，得られた決定リストの証拠の個数(決定リストのサイズ)は平均2081個であった．

この決定リストを使用した同音異義語の誤り判定の結果を表4.5に示す．それにより，この実験では決定リストを利用した同音異義語の判定の平均正解率が85.47%で

同音異議語「債券」の書き誤り判定

正しい:○　誤り:×
○　:もっとも の もっとも NULL NULL の 上昇 小幅
○　:ニューヨーク 相場 ニューヨーク NULL NULL 相場 下落 .
○　:証券 売買 証券 三 岡 売買 部長)
○　:部長 , 部長 売買 債券 , 短期 金利
○　:為替 も 為替 先物 指数 も 小 動き
×　:3 は 3 広がる 空気 は 圏 で
○　:上昇 売ら 上昇 原油 懸念 売ら こと 考え
○　:ロンバード 貸付 ロンバード 近く ドイツ 貸付) を
×　:NULL ボックス NULL NULL NULL ボックス 圏 もみ合い
...

図 4.1: 「債券」における判定結果ファイルの例

あることがわかる. 今回用意したの同音異義語はその意味の近さなどから判定に難しいと思われるものを用意しているため, それを考慮すれば同音異義語問題に適用した例としてはかなり良い結果を得られたと考えられる.

表 4.5: 決定リストによる判定の正解率

同音異義語	決定リストのサイズ	出現総数	正解率 (%)
{債券, 債権}	1194	3781	73.18
{解放, 開放}	1907	2462	88.46
{協調, 強調}	3340	5937	89.29
{自信, 自身}	4104	2690	85.64
{感心, 関心}	2677	2757	94.02
{体外, 対外}	1092	1592	94.54
{運航, 運行}	941	1623	69.13
{同志, 同士}	1945	1328	91.11
{過程, 課程}	1735	1473	95.32
{実効, 実行}	2170	1683	89.07
{食料, 食糧}	1609	1558	64.38
{傷害, 障害}	2259	1373	92.35
平均	2081	2355	85.47

4.3 決定木による実験

ここでは実際に本研究で行った，本手法である決定木による実験の手順を説明し，その結果を示す．

4.3.1 実験手順

STEP1 属性の取りうる値をまとめる．

決定木を同音異義語問題に適用するために，本研究では「頻出自立語x個+other」という分類を行ってまとめることで，属性値の取りうる種類を押さえる工夫をする．その為，まずトレーニングデータから証拠の頻度を計算し，得られた情報を頻度順にソートする．その後に採用する種類だけ上位から取り，その他をotherとして扱う．

実際には，決定木作成に使用したパッケージソフトウェアc4.5は日本語文字列の処理を意識して設計されていないため，正常な処理を行うためには日本語EUC文字列で記述された自立語を別のASCII文字列へ変換するテーブルを用意する必要があった．そこで今回の実験ではC++で変換テーブルオブジェクトを実装し，メソッドを介してEUC文字列と，それに対応するASCII文字列（1から始まる正数の数列）の相互変換を行っている．C4.5に渡す必要のある全ての日本語文字列はこの変換テーブルオブジェクトを使用して変換した．

STEP2 名称ファイル(.names)の作成

C4.5で決定木作成の処理を行うために必要とする基本的なファイルに名称ファイルが挙げられる．このファイルにはクラス，属性，属性値の名称が格納されており，このファイルの内容はC4.5が決定木を作成する際の指針となる．

名称ファイルはそれぞれのエントリーが行の先頭から始まり，ピリオドで終わり，その繰り返しで構成される．空行やスペース，タブはファイルを読みやすくするために使用され，重要な意味を持たない．行中に現れる縦棒はどこにあってもその行の残りの部分を無視されることを意味し，ファイル中に注釈をつけるために使用される．

名称ファイルの最初にはクラス名を記述する．それぞれのクラス名はコンマで区切られていて，行末にはピリオドを打つ．必ずクラス名は2つ以上なければならず，またその順番は任意でよい．

ファイルの残りの部分は属性のエントリーからなる．各属性に対するエントリーは1つである．属性のエントリーは属性名で始まり，そのあとコロンで続き，そして，その属性が取りうる値のタイプが続く．4種類のタイプが指定できる．

- ignore：属性値は無視される．

- continuous：属性値は数値であり，整数か浮動小数点のいずれかであることを示す．
- discrete N （ N は正の数値である）：属性値は離散値を取り，N 個よりも多くの値は取らないことを示す．
- コンマで区切られた名称のリスト：属性は離散値であり，記述された名称のいずれかを取る（データをチェックできるのでdiscreteよりも望ましい）．クラス名と同様に属性値の順序は任意である．

今回属性として割り当てられるのは証拠としての自立語をASCII文字列に変換したものであり，それゆえ離散値である．よって4種類のタイプのうち，コンマで区切られた名称のリストを使用する．

例として，同音異義語{債券,債権}に対応させて作成した名称ファイルを図4.2挙げる．

```
1,2.
attribute#1: 1, 2, 3, ..., 298, 299, 300, other.
attribute#2: 1, 2, 3, ..., 298, 299, 300, other.
attribute#3: 0, 1.
attribute#4: 0, 1.
...
attribute#301: 0, 1.
attribute#302: 0, 1.
```

図 4.2: 債券,債権に対応する名称ファイルの例（採用証拠数300）

図4.2において，始めの行はクラスであり，これは同音異義語の表記にあてはまる．ここでは日本語文字列を変換デーブルを介して数字の文字列に変換しているため，同音異義語の語義を2つと限定した条件である本実験において実際に記述される値は1と2である．また，次行から属性の名前と取りうる値を記述する．属性の順序は

attribute#1（直前の自立語）
attribute#2（直後の自立語）
attribute#3（頻出自立語１）
attribute#4（頻出自立語２）
...

となっている．頻出自立語の数は採用する個数の設定を変化させることで，それに従って増減する．

「直前の自立語」と「直後の自立語」は採用した自立語の個数だけあることになる．属性値の文字列変換テーブルはクラスのそれとは別なものを用意するため，ここでは新たに1から記述し，採用した自立語の個数だけ数値を並べる．例えば，採用した自立語の数を100個としたならば，ここで「直前の自立語」と「直後の自立語」が取りうる値の種類は101個となり，また名称ファイルに記述される名称列は1, 2, ..., 100, otherとなる．

属性「頻出自立語 N 」とは，証拠として採用された頻出上位 N 個目の自立語を表し，その取りうる値は，トレーニングデータの事例にその頻出上位 N 個目の自立語が存在していれば1，していなければ0を取るようにした．よって頻出自立語についての記述が始まる4行目以下は採用した自立語の個数分だけ，属性名とその取りうる値0,1が記述されることになる．

STEP3 データファイル (.data) の作成

名称ファイルの作成が終わったら，今度はデータファイルの作成に移る．データファイルとは決定木とルール集合の両方，またはいずれかを構築するための訓練事項を記述したファイルである．各行あたり1つの事例が記述され，それは全ての属性に対する値と事例のクラスを含む．またその値及び事例はコンマで区切られ，最後にピリオドがつけられる．属性値の順序は名称ファイルに記述された属性の順序と一致させなければならない．事例の順序は任意でかまわない．

実験において，一つの訓練事例に対応して記述されるデータ列は以下のようになる．

直前の自立語, 直後の自立語, 頻出自立語1, 頻出自立語2, ..., クラス名.

これは，最後のクラス名を除いて先述の名称ファイルに記述された属性の順序と一致する．また，最後のクラス名はこの訓練事例を得たときの同音異義語の表記である．当然ながら数字の文字列に変換されたものなので，ここには1か2が記入される．

例として，同音異義語{債券, 債権}の例として挙げた名称ファイル（図4.2）に対応させて作成したデータファイルを図4.3に挙げる．

STEP4 決定木の生成

名称ファイルとデータファイルがあれば，それを利用してC4.5にて決定木を出力することができる．C4.5は機能も豊富であり，木の生成に関する諸パラメータの調整や評価方法の種類の選択，テスト回数の変更などを行うことができる．だが今回の実験では特に調整は行わず，全ての同音異義語に対し一貫して標準のオプションで実験を行った．ここで標準のオプションとは，

- テスト集合を指定しない．

```
67, 5, 0, 0, ..., 0, 0, 0, 0, 0, 1
128, 68, 0, 0, ..., 0, 0, 0, 0, 0, 1
280, 2, 0, 1, ..., 0, 0, 0, 0, 0, 1
other, 207, 0, 0, ...,0, 0, 0, 0, 0, 1
...
44, 2, 0, 1, ..., 0, 0, 0, 0, 0, 2
1, 3, 1, 0, ..., 0, 0, 0, 0, 0, 2
1, 3, 1, 0, ..., 0, 0, 0, 0, 0, 2
```

図 4.3: 図 4.2 に対応するデータファイルの例

- 離散的な属性値のテストのためにグループ化をしない.
- 少なくとも 2 つの部分集合に対して，事例の重みの和が何らかの最小値以上でなければならない.
- 決定木の枝仮に影響を与えるパラメータ $CF = 25\%$
- 繰り返し木を成長させるウインドウ処理を行える木の数=10
- ウインドウ処理を起動し，初期のウインドウに含まれる事例の数を，訓練事例の 20%と訓練事例の数の平方根の 2 倍のうちの大きい方の数にする.
- ウインドウ処理を起動し，それぞれの繰り返しにおいてウインドウに追加される事例の最大値をウインドウの大きさの 20%とする.
- 事例分割の評価基準に利得比基準を用いる.

という条件を暗黙的に指定したことになる.

C4.5 の出力する決定木はその形式から見て二種類ある．一つは C4.5 自身が直接利用できる，決定木の構造が入ったバイナリファイル（.tree）であり，もう一つは標準出力に出力するキャラクタ表示の木である．実際に決定木を利用して誤り判定を行うためには，C4.5 の出力した木の意味を解釈しなければならない．そのため，どちらかの出力形式を選択して利用する必要がある.

前者の場合，C4.5 のソースコードからそのデータ構造を把握し，同音異義語の誤り判定プログラムにそのデータ構造を利用できるように実装すれば良い．しかしそのデータ構造はやや難解であり，ソースコードを理解し正確に実装できる段階まで辿り着くには時間がかかると思われる．そのため本実験の判定用プログラムには後者の形式を利用する．具体的には，

1. 判定用プログラムが子プロセスとして C4.5 を起動.

2. C4.5に決定木を作成させる．その結果として標準出力に出力されたキャラクタ木をプロセス間通信を利用して読み取る．

3. プログラム側でキャラクタ木を解釈して決定木を利用なデータ形式に再構築する．

4. その後テストデータを一事例ずつ読み取り，決定木を利用して判定する．

といったことを行うように設計した．

STEP5 誤り判定

同音異義語債券,債権においてC4.5の出力した決定木の例を図4.4に示す．

```
attribute#3 = 1: 2 (314.0/2.6)
attribute#3 = 0:
|   attribute#22 = 1: 2 (21.0/1.3)
|   attribute#22 = 0:
|   |   attribute#32 = 1: 2 (15.0/1.3)
|   |   attribute#32 = 0:
|   |   |   attribute#41 = 1: 2 (10.0/1.3)
|   |   |   attribute#41 = 0:
...
```

図 4.4: {債券,債権}においてC4.5の出力した決定木の例

この例は，クラスの文字列が1:債券,2:債権に対応しているとすると，次のif-then表現形式と意味的に同等である．ただし，attributeの番号を頻出自立語の番号に言い換えるときには，attribute#1とattribute#2が直前と直後の自立語に割り当てられていることに留意し，その2つ分だけマイナスする必要がある．

この様に決定木の解釈を行うことによって同音異義語の誤り判定のルールを構築することができる．また，判定の結果出力と平均正解率の算出については決定リストと同様の方法で行う．

以上の手順を踏まえた上で実験を行い，決定リストの判定の正解率と比べてみる．

属性が多数の値を取ることで決定木の利得比基準の値が信用できなくなることへの対策として，取りうる値を「頻出自立語x個+other」種類としたが，一般的に決定木の属性値が100種類を越えると利得比基準の値に問題が生じて高い正解率は望めないと言われているため，標準で100個と設定した．また，まとめる個数による正解率の変化を調べてみるため，100個から20刻みで200個まで変化させ，同様の実験を行って正解率を計算してみた．

```
if(頻出自立語No.1が事例の証拠集合に含まれる)
  return 債権;
else {
  if(頻出自立語No.20が事例の証拠集合に含まれる)
    return 債権;
  else {
    if(頻出自立語No.30が事例の証拠集合に含まれる)
      return 債権;
    else {
      if(頻出自立語No.39が事例の証拠集合に含まれる)
        return 債権;
      else {
        ...
```

図 4.5: 図 4.4 に対応する if-then ルール

4.3.2 実験結果

まず，属性の取りうる値の種類を 100 個に固定したうえで実験を行った．そのときの結果を，決定リストでの結果と対比し表 4.6 に示す．表 4.6 より，個々の同音異義語の観点ではそれぞれの手法に優劣があるものの，平均正解率は決定リストが決定木を上回っていることがわかる．

次に，属性の取りうる値の種類を 100 から 20 刻みで 200 まで増加させた場合の実験を行った．結果を表 4.7 に示す．表 4.7 より正解率の変化は同音異義語によって様々だが，属性値のとり得る値を増やすにつれて平均正解率がやや上昇していることがわかる．

表 4.6: 決定リストと決定木の正解率の比較

同音異義語	出現総数	決定リストでの正解率 (%)	決定木での正解率 (%)
{債券，債権}	3781	73.18	73.21
{解放，開放}	2462	88.46	77.01
{協調，強調}	5937	89.29	85.31
{自信，自身}	2690	85.64	84.94
{感心，関心}	2757	94.02	92.13
{体外，対外}	1592	94.54	94.66
{運航，運行}	1623	69.13	68.76
{同志，同士}	1328	91.11	91.49
{過程，課程}	1473	95.32	94.23
{実効，実行}	1683	89.07	86.69
{食料，食糧}	1558	64.38	67.59
{傷害，障害}	1373	92.35	93.30
平均	2354	85.47	83.42

表 4.7: 属性値のとり得る個数を変化させたときの正解率の変化

同音異義語	正解率 (%)					
	100	120	140	160	180	200
{債券，債権}	73.21	76.88	76.46	76.36	76.43	76.49
{解放，開放}	77.01	78.23	77.82	77.82	79.04	79.00
{協調，強調}	85.31	85.77	86.10	85.95	86.14	86.24
{自信，自身}	84.94	84.76	84.87	84.54	85.06	84.65
{感心，関心}	92.13	92.13	91.80	91.73	91.73	91.73
{体外，対外}	94.66	94.28	94.22	94.10	94.35	94.35
{運航，運行}	68.76	68.76	70.12	72.64	72.64	72.64
{同志，同士}	91.49	91.27	88.70	88.63	87.95	88.18
{過程，課程}	94.23	94.03	93.96	93.89	93.89	93.89
{実効，実行}	86.69	87.46	87.46	87.46	88.24	88.24
{食料，食糧}	67.59	64.89	66.88	67.33	67.65	67.91
{傷害，障害}	93.30	93.66	93.66	93.66	93.66	93.88
平均	83.42	83.97	83.99	84.06	84.31	84.33

第5章　考察

5.1　決定リストによる誤り判定の結果について

決定リストについて，表から判定の平均正解率が85.47%と，同音異義語問題に適用した例としてはかなり良い結果を得られたと考えられる．最も結果の良かった同音異義語は{過程，課程}で，正解率が95.32%である．それに対して最も結果の悪かった同音異義語は{食料，食糧}で，正解率が64.38%であった．

{過程，課程}などのように，今回用意した同音異義語の中で語義の違いが比較的分かりやすいものは正解率が高い結果となった．この実験で誤り判定に使用される証拠は，目的となる同音異義語の前後 ±3 後以内の自立語である．そのため，表記に対して出現する自立語の種類や頻度に違いが出れば，その語義の判定も容易になる．

そこで，最も結果の良かった{過程，課程}及び最も結果の悪かった{食料，食糧}の決定リストを比較してみるため，表5.1と表5.2にそれぞれ示して比較してみる．

表5.1と表5.2を比較してすぐに分かることは，証拠一つ当りに含まれるトレーニングデータの事例数に大きな違いがあることである．

最も良い結果を残した{過程，課程}について，最上位の予測力を持つの証拠に含まれるトレーニングデータの事例数は48である．しかし，これに対して最も悪い結果を残した{食料，食糧}における最上位の予測力を持つ証拠に含まれる事例数は23と，その差は大きい．このことから{食料，食糧}が正解率64.38%しか出せなかった原因の一つとして，トレーニングデータから十分な事例が得られなかったことが考えられる．

リストの下部を見てみると，予測力に大きな差が出ていることが分かる．{食料，食糧}において，予測力の低い証拠まで採用されている理由は，defaultにおいて「食料」と「食糧」の総頻度が拮抗しているためにdefaultの予測力が低いからである．だがそれを考慮しても，{過程，課程}と同程度の順位において，1500番付近では予測力に数倍ものの差が出ている．つまり，一つの証拠に対して「食料」と「食糧」の事例数が同程度出てしまうことが原因である．これは表記に対して語義の区別が意味的に近く，直感的に判別がつきにくいということを表す．この原因から考えられることは，トレーニングデータの事例数を増やしても，同程度の事例がでてしまう場合，多少の正解率向上は見られてもそれほどの解決にはならないであろうということである．

表 5.1: 最も良い結果の決定リスト（{過程，課程}）

順位	証拠	過程	課程	解答	予測力
1	c 修士	0	48	課程	8.9099
2	a 修士	0	46	課程	8.8487
3	c 博士	0	45	課程	8.8170
4	a 博士	0	40	課程	8.6475
5	c 修了	0	40	課程	8.6475
6	c 大学院	0	39	課程	8.6110
7	c その	37	0	過程	8.5353
8	c 決定	37	0	過程	8.5353
9	a 決定	36	0	過程	8.4959
10	c いく	31	0	過程	8.2808
...	...	...	...	...	...
1732	c れて	4	1	過程	1.8981
1733	c 進める	4	1	過程	1.8981
1734	c 文化	1	4	課程	1.8981
1735	default	4398	1185	過程	1.8919

表 5.2: 最も悪い結果の決定リスト（{食料，食糧}）

順位	証拠	食料	食糧	解答	予測力
1	a 新	0	23	食糧	7.8518
2	b 事務所	0	22	食糧	7.7880
3	c 事務所	0	22	食糧	7.7880
4	b 法	0	14	食糧	7.1396
5	c 法	0	14	食糧	7.1396
6	c 代わる	0	13	食糧	7.0335
7	c 法案	0	13	食糧	7.0335
8	c 自給	0	11	食糧	6.7945
9	c 食糧管理法	0	11	食糧	6.7944
10	a 主要	0	9	食糧	6.5078
...	...	...	...	...	...
1616	c 月	4	5	食糧	0.3149
1617	c 人	5	4	食料	0.3149
1618	c と	17	14	食料	0.2783
1619	default	1897	2272	食糧	0.2602

この問題を改善するためには，より複雑なルールを学習する工夫が必要である．今回用いた決定リストの学習する形式は図3.1の通りである．そこで，図5.1のような工夫を決定リストに用いたらどうだろうか．今回用いた決定リストは一時的

```
if (前の自立語 == "共同" && 後ろの自立語 == "責任")
  return 債権;
if (前の自立語 == "株式" && ... && ... && ... && ...)
  return 債券;
if (後ろの自立語 == "問題" && ... && ... && ... )
  return 債権;
...
return 債権;
```

図 5.1: 決定リストの改善後の表現形式例

な条件にマッチすれば解答を得られるような形式だったが，図5.1においては，条件部が複数条件の積で表現されている．この工夫を用いれば，より複雑なルールを学習することができ，決定リストが表現できる形式も多彩になる．

理論的には，これは決定木の表現形式と同等となるはずである．図5.1における複数条件の積で表現された条件文は，条件の積の回数に相当する深さを持った決定木のノードに対応する．つまり，この工夫からも分かることは，決定木の表現形式は決定リストの表現形式と同等，または内包しているということである．

だが，現実的にはこの工夫を用いることは難しい．何故なら，積を取る条件の個数が事前にわからなければ，決定リストを作成できないからである．つまり，この工夫の元に決定リストを作成するには条件の積の個数に上限を設けなければならないという制限がつく．このことは，決定木がその木の深さにとらわれず，無制限に条件の積を取れることを考えれば大きな制約となる．

5.2 決定木による誤り判定結果について

属性の取りうる値の種類を100と固定して本手法である決定木を用いた場合，決定リスにおいて最も結果の悪かった同音異義語{食料，食糧}(正解率64.38%)について，表4.6から決定木の結果(正解率67.59%)で3.21%ほど改善されていることが読み取れる．このことから，{食料，食糧}のように直感的に語義の意味の区別をつけにくいような同音異義語に関して効果があることが分かる．

またその効果から決定木の平均正解率も期待したのだが，実際は決定リストの85.47%に対して83.42%と，2.05%ほど下回ってしまった．

ここの同音異義語の観点から表4.6を見てみると，決定リストにおいて最も結果

の悪かった{食料，食糧}の次に悪い結果である{運航，運行}の正解率69.13%に対して，決定木で68.76%と，思ったように効果がでない場合もあった．また，{解放，開放}について決定リストでは88.46%なのに対し，決定木では77.01%と，大きく下回る結果のものもあった．

理論的には決定木は決定リストの学習の表現形式を内包しているため，正解率は決定リストを上回ることが予想されていたが，実際にはいくつかの同音異義語でその効果があったものの，全体的には決定リストをやや下回った．この原因と考えられるものとしては，

1. 属性の取りうる値の種類が多いため，決定木作成時の要である利得比基準の値が信用できなくなってしまい，適切な木を作成できなかった．

2. 多数の属性値の種類への対策として行った「頻出自立語上位x個+other」にまとめる方法が上手く効果を発揮しなかった．

3. 決定木よりも決定リストのほうが適切な手法といえる同音異義語の種類が存在した．

といったことが挙げられる．

考えられる原因の1番目について．同音異義語の種類によっては，属性の取りうる値の種類を「上位100種類+other」個におさえたとしても，利得比基準に影響が出てしまった可能性がある．一般的には決定木について，属性の値が100種類を超えても正解率が良くならなければ，その問題に対して決定木で解いてもいい結果を出せる見込みはないといわれている．そのことから，同音異義語問題を同音異義語の種類まで掘り下げた観点で捕らえれば，問題によっては決定木に適さず，むしろ証拠の種類の多さに制限をうけない決定リストの方が適している場合があると考えられる．

原因の2番目として，多数の属性値の種類への対策として行った今回の工夫を挙げた．今回の工夫は証拠として出現した自立語の頻度を計算し，その上位から一定個数採用するという簡単な方法である．しかし，この方法だと単純に「出現頻度が高い」という理由のみで採用することになり，語義が似通った同音異義語の場合だとどちらにも共通の証拠として出現してしまう恐れがある．こういった証拠は明らかに「強い」証拠力をもつとはいい難いため，この証拠の有無による事例の判定は誤る可能性も高い．通常この類いの属性による事例分割は利得比基準の値は小さく算出されるため，その属性による分割の優先度は低くなるものと考えられる．だが，同音異義語問題のように属性の取る値が単語表記そのものになってしまう場合，どうしても利得比基準の信頼性の問題が発生してしまう．それ故，属性分割の評価が正しく行われないケースも存在することを考えなければならない．

結局，原因の3番目も含め，同音異義語問題で前後±3語以内の自立語の出現頻度のみで高い判定精度を望むことには限度があり，例え頻出する自立語を上位

から採用するような対策をとっても解決にはならない同音異義語が存在する，というのが今回の工夫で得た結論である．よって同音異義語問題において現段階では単純に属性の取り得る値の種類を押さえる工夫のみでは正解率の高い決定木の生成は難しいため，決定リストや証拠の出現順序を考慮して隠れマルコフモデルといった別の手法を補助的に用いる工夫が有効と思われる．

5.3 採用した証拠の個数と正解率の関係について

採用した証拠の個数を標準の100個から20刻みで200個まで変化させた場合の平均正解率の変化について示したグラフを図5.2に示す．また，それぞれの同音異義語の正解率の変化をグラフにしたものを図5.3から図5.14に示す．

図5.2により100個から200個へ増やすにつれて平均正解率が上昇していることがわかる．しかし個々の同音異義語からの観点からは，グラフを見てみると以下の3つのパターンに分けられる．

- 正解率が上昇する同音異義語
- 正解率が下降する同音異義語
- その他の同音異義語

まず正解率が上昇するものについては，{債券，債権}（図5.3），{協調，強調}（図5.5），{運航，運行}（図5.9）などが挙げられる．

また，正解率が下降するものについては，{自信，自身}（図5.6），{感心，関心}（図5.7），{過程，課程}（図5.11）などが当てはまる．

その他の同音異義語とは，上記の2種類のどちらにも当てはめにくいタイプの変化を見せたものである．{体外，対外}（図5.8），{食料，食糧}（図5.13）などがこの種類の同音異義語で，グラフを見ると増加と減少を交えたランダムな変化を見せていることが分かる．

以上の3つのパターンにおいて共通していることとして，その変化を見せた背景には，

- 採用した自立語の種類を増やしたことにより決定木の表現するルールがより豊富な種類を表現できるようになり，判定の精度が上昇する．
- 採用した証拠となる自立語の数の増加，即ち属性の取りうる値の種類が増えてしまうことにより，利得比基準の値が信用できなくなり適切な決定木の作成ができずに判定の精度が下降する．

という2つの相反する要因が絡んでいると思われる．

例えば採用した証拠の個数を増やしたことで正解率が上昇した{債券，債権}などは，適切な木が作成できなると行った影響よりも，証拠の数を増やしたこと

による精度の向上の効果の方が高かったと考えられる．このまま証拠の数を増やし続ければ正解率がこのまま上昇し続けそうだが，実際は限度がある．何故なら，証拠の数を増やし続けることで「頻出自立語上位 x 個+other」という属性の取りうる値をまとめた工夫が無駄になり，根本的な問題に戻ってしまうからである．

逆に下降した例で{自信，自身}などは，証拠の数を増やすことよりも，決定木がいかに適切なものなのか，ということの方が重要であったと思われる．このグループに該当する同音異義語は，恐らく採用した証拠の個数が標準の 100 個のときでさえも利得比基準に問題が生じている可能性がある．{体外，対外}などは，証拠の個数が 100 個のときで正解率が 94.66%と高く，既に適正な決定木が作成されていたと考えられる．この様な高い正解率の同音異義語の多くが証拠の個数を増やすにつれて正解率を下げている傾向があり，やはり利得比基準の問題が浮彫りになっている．

また，上記の 2 つに当てはまらないタイプとして{体外，対外}などは，正解率の変化ががあまり安定していないことから，やはり利得比基準の値が信用できなくなったことの影響を強く受けているものと考えるのが妥当だろう．

これらのことから，同音異義語問題において採用する証拠の個数を増やしたことによって平均正解率の上昇が見られたものの，利得比基準の値が信用できなくなる問題は無視できないと言える．各同音異義語の観点では，いくつかの種類ではかえって正解率が低下するなど，その問題がむしろ明確にでている．よって証拠の個数を増やすことが単純に決定木の精度を上げることに繋がるとは言えず，各同音異義語の性質に合った方法で個別に対応する必要があると考えられる．

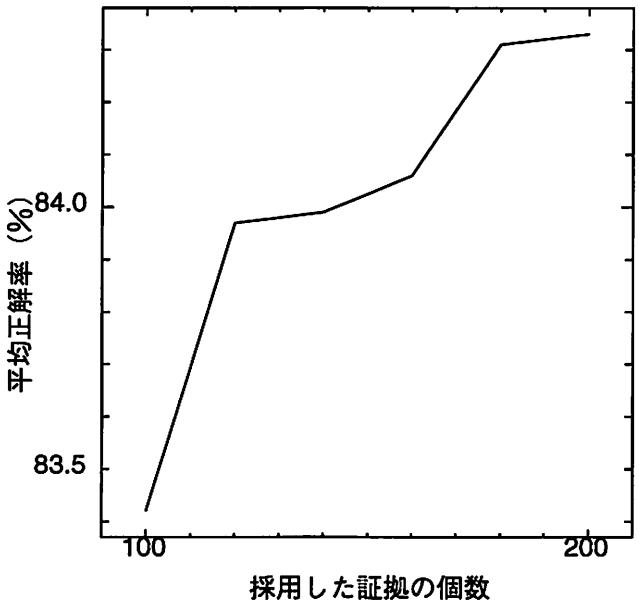

図 5.2: 採用した証拠の数を変化させたときの平均正解率の変動

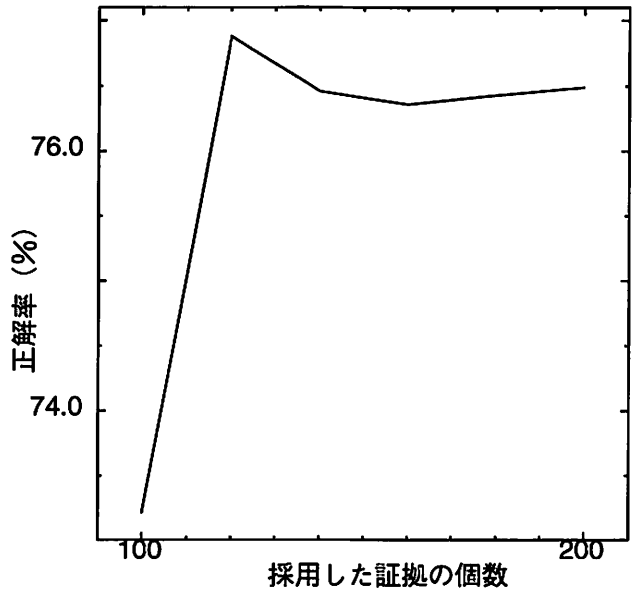

図 5.3: {債券，債権} における正解率の変動

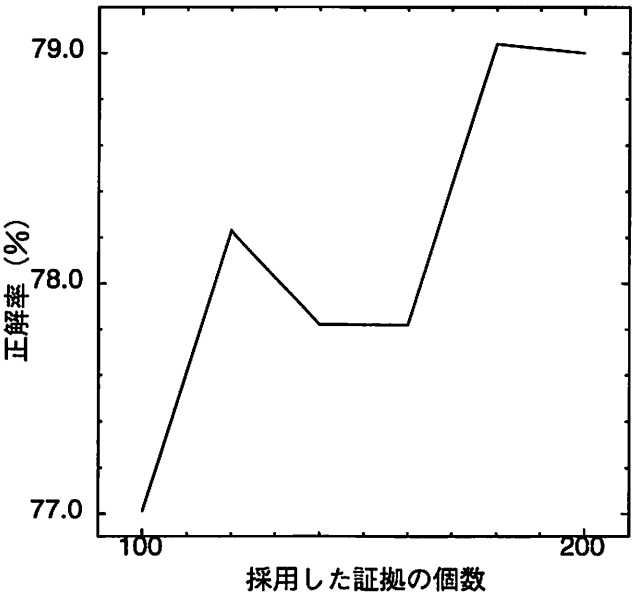

図 5.4: {解放，開放} における正解率の変動

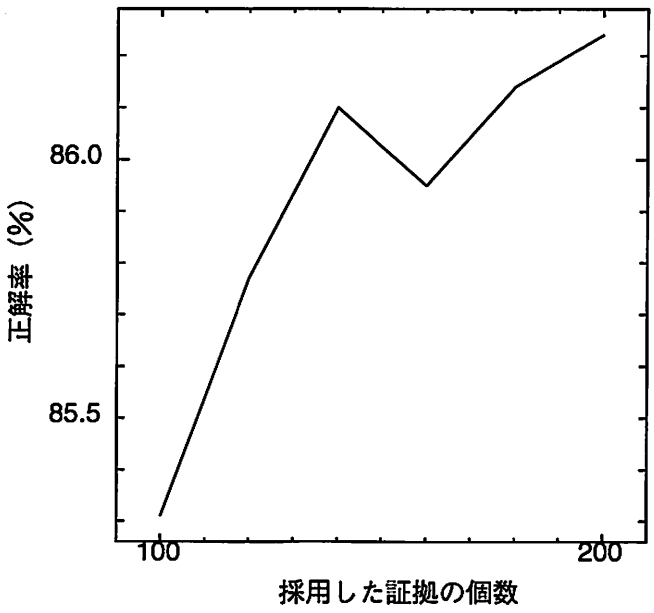

図 5.5: {協調，強調} における正解率の変動

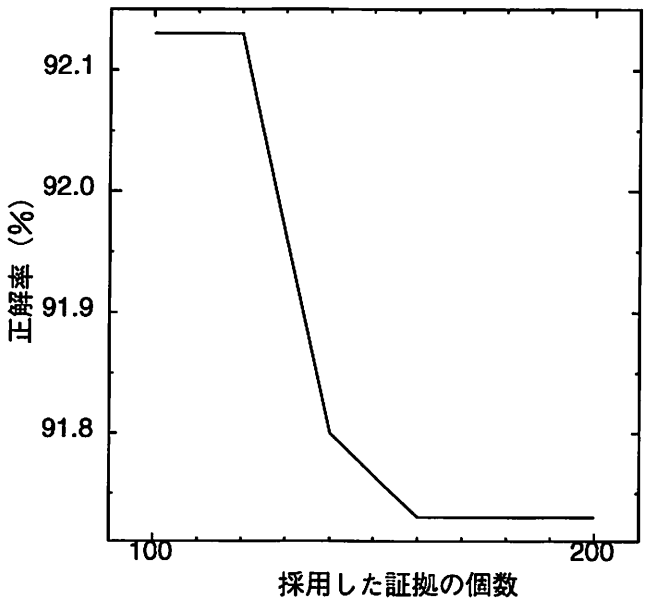

図 5.6: {自信，自身} における正解率の変動

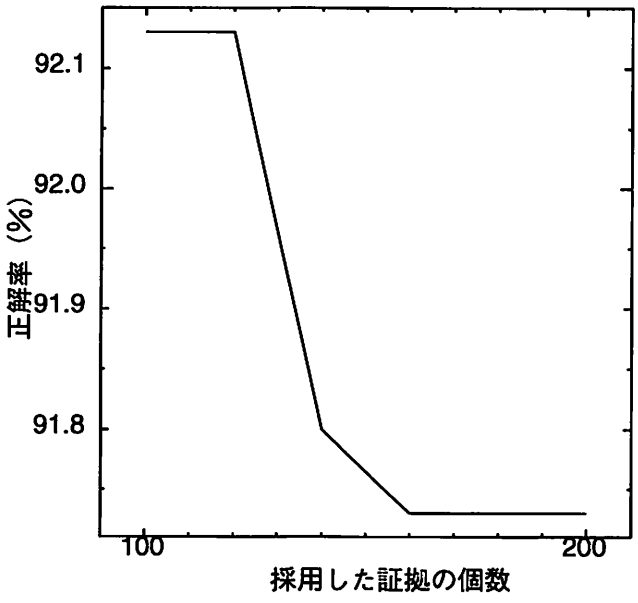

図 5.7: {感心，関心} における正解率の変動

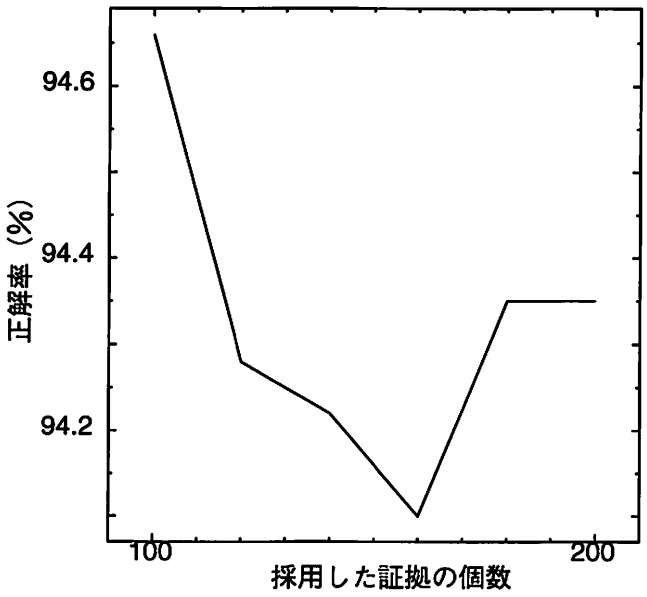

図 5.8: {体外，対外} における正解率の変動

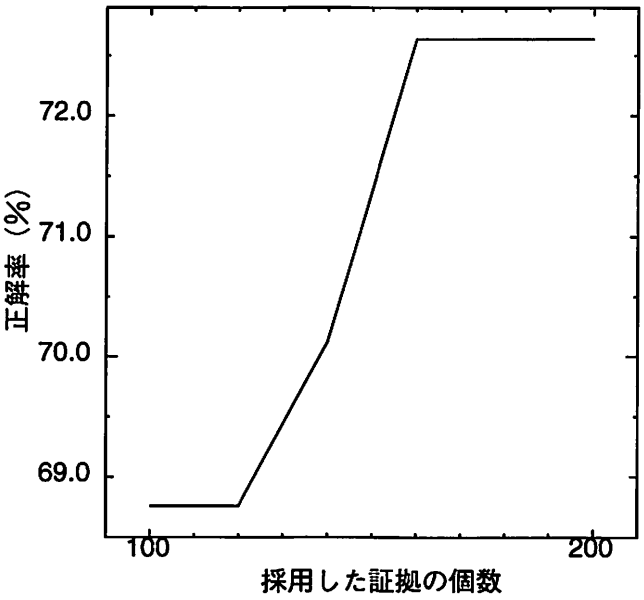

図 5.9: {運航，運行} における正解率の変動

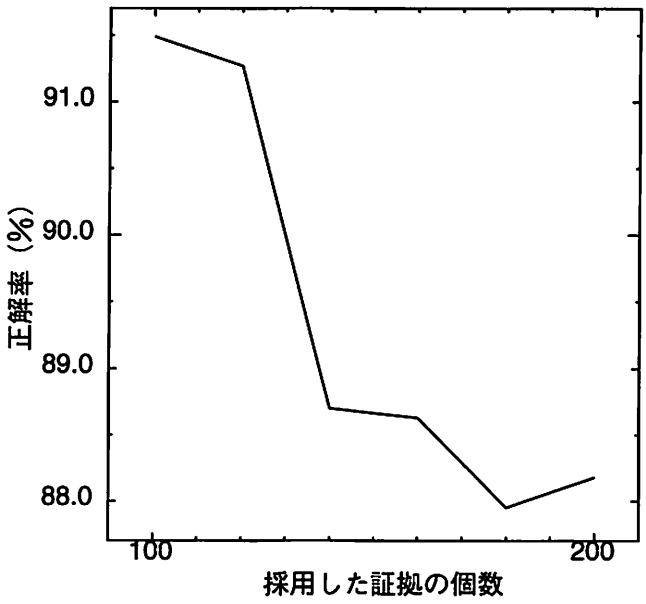

図 5.10: {同志，同士}における正解率の変動

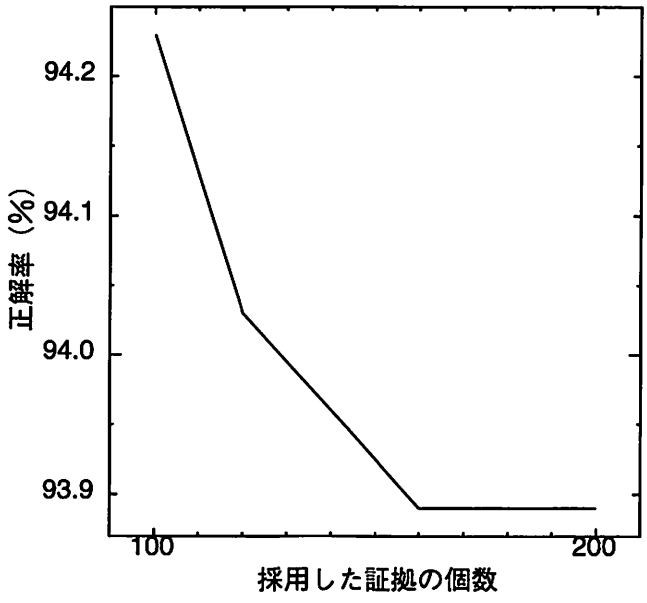

図 5.11: {過程，課程}における正解率の変動

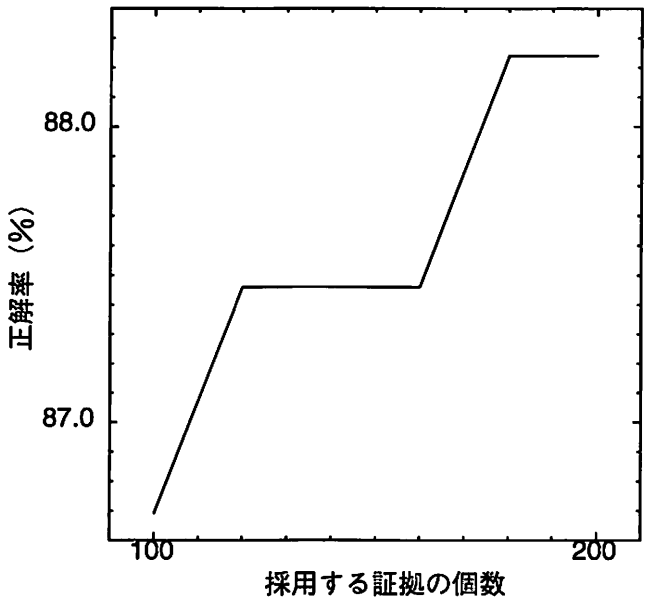

図 5.12: {実効，実行} における正解率の変動

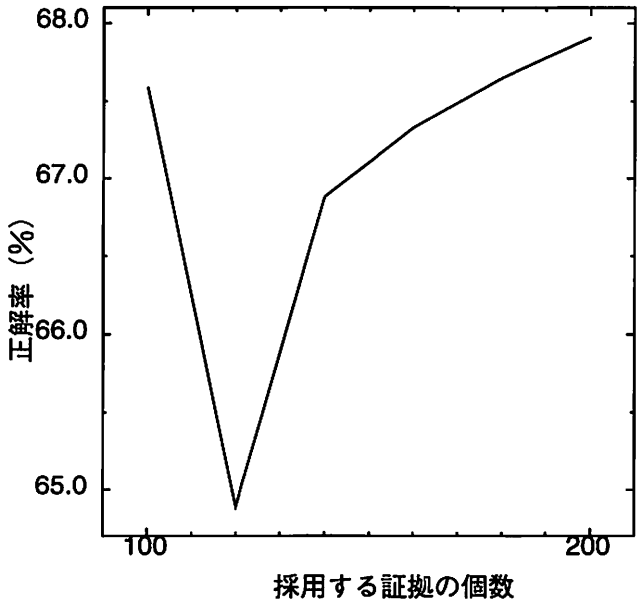

図 5.13: {食料，食糧} における正解率の変動

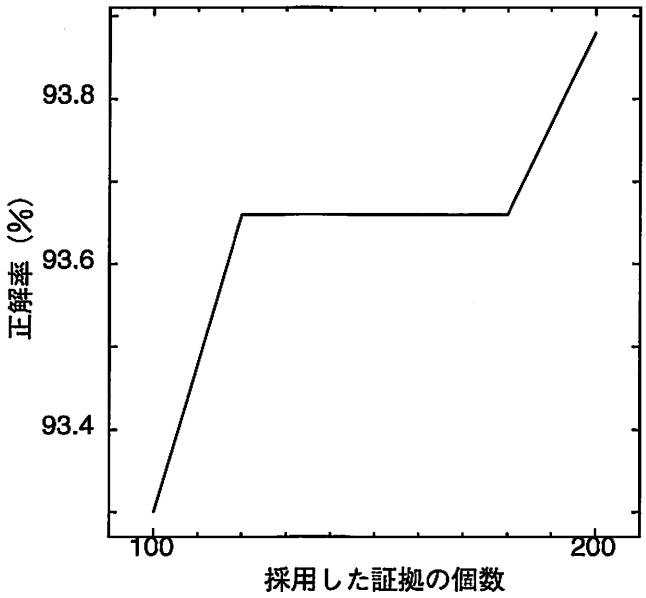

図 5.14: {傷害，障害}における正解率の変動

第6章　まとめ

本研究の目的は，決定木の手法を利用して同音異義語の誤り検出とその修正をすることにある．実際に本研究では以下の事柄を行った．

- 従来手法である決定リストによる同音異義語の誤り検出と修正のシステムを作成した．
- 決定木を同音異義語問題に用いるための工夫を提案し，それによる同音異義語の誤り検出と修正のシステムを作成した．
- 各々のシステムによる実験結果を比較し，決定木の有効性を検証した．
- 決定木にて属性値のとり得る値と正解率の関係について実験し考察した．

残された課題として，以下のことが挙げられる．

- 今回の工夫から得られた結果を元に，さらに良い結果を出せる決定木を作成できるような属性値の選び方を考案する．
- 属性の取りうる値の種類の変化による正解率の変動によってグループ化した同音異義語の性質について，より細かい分析を行う．

謝辞

本研究を進めるにあたって，その元となった理論の御指導及び論文作成における多大な御助言を新納 浩幸 教官 (茨城大学工学部システム工学科) に賜りました．その深い感謝の意をここに表します．

また，本研究に用いたトレーニングデータとテストデータ作成の元データには，毎日新聞'94 年度版および，日本経済新聞'91 年度版を使用しました．利用を許可していただいた毎日新聞社と日本経済新聞社に深く感謝致します．

最後に，システム工学科計算機応用講座の教官の方々にも深く感謝致します．

付 録A　日本語EUC文字列⇔ASCII文字列変換テーブル

```
/*******************
     s_table.h
*******************/

// source : 変換前の元々の文字列
// convert: 変換後の文字列
// 対象とするファイルは*.tblとする.

#ifndef __S_TABLE_H__
#define __S_TABLE_H__

#include <string>
#include <string.h>
#include <list>
#include <stdio.h>
#include <algorithm>
#include <fstream>
#include <function.h>
#include <excpt.h>

using namespace std;
using namespace my_excpt;

////////// prototype function //////////
void int2str(int n, char* ch, size_t size = 10);


//////////////////////////////
```

```
// StrTable
class StrTable
{
public:
  StrTable() : count(0) { }
  StrTable(const string& table_filename);
  virtual ~StrTable();

  void read(const string& table_filename);
  void write();
  void write(const string& table_filename);

  void push(const string& word);
  size_t size() const { return count - 1; }
  void clear();

  bool src2cvt(string& result, const string& source_word);
  bool src2cvt(int* result, const string& source_word);
  bool cvt2src(string& result, const string& convert_word);
  bool cvt2src(string& result, int convert_word);

  static const size_t nondef = static_cast<size_t>(-1);


private:
  struct WordPair {
    string source_word;  // 元の文字列
    size_t convert_word; // 変換後の文字列

    // 比較対象指定
    typedef enum{ CF_SRC, CF_CVT } COMP_FACTOR;
    static COMP_FACTOR comp_factor;

    bool operator<(const WordPair& arg) const;
    bool operator>(const WordPair& arg) const;
    bool operator==(const WordPair& arg) const;
  }; // WordPair

  string filename;
```

```
  size_t count;
  list<WordPair> list_wp;

}; // StrTable


//////////////////////////////
// StrTable::clear
inline
void StrTable::clear()
{
  count = 0;
  list_wp.clear();
  filename = "";

} // StrTable::clear


//////////////////////////////
// StrTable::src2cvt
inline
bool StrTable::src2cvt(int* result, const string& source_word)
{
  string str;
  bool res = src2cvt(str, source_word);
  *result = atoi(str.c_str());
  return res;

} // StrTable::src2cvt


//////////////////////////////
// StrTable::cvt2src
inline
bool StrTable::cvt2src(string& result, int convert_word)
{
  char str[10];
  int2str(convert_word, str);
  return cvt2src(result, str);
```

```
} // StrTable::cvt2src


//////////////////////////////
// int2str
inline
void int2str(int n, char* ch, size_t size)
{
  char* t = new char[size];
  int k = size - 1;
  t[k] = '\0';

  while (n > 0) {
    t[--k] = '0' + n%10;
    n/=10;
  }

  strcpy(ch, t+k);
  delete t;

} // int2str


#endif // __S_TABLE_H__

// end


/*******************
     s_table.cc
*******************/

#include "s_table.h"


////////// public //////////

StrTable::StrTable(const string& table_filename)
```

```
  : filename(table_filename)
{
  try {
    read(filename);
  }
  catch(excpt& e) {
    throw;
  }
} // StrTable::StrTable


StrTable::~StrTable()
{

} // StrTable::~StrTable


////////////////////////////
void StrTable::read(const string& table_filename)
{
  filename = table_filename;

  ifstream in(filename.c_str());
  check(in.is_open(), string("can't open the file:") + filename);

  // *.tblファイルの読み込み
  char source[50];
  size_t source_size;
  WordPair wp;

  list_wp.clear();
  while(!in.eof()) {
    in.read((char *)&source_size, sizeof(size_t));
    in.read(source, source_size);
    wp.source_word = source;
    in.read((char *)&wp.convert_word, sizeof(size_t));

    list_wp.push_back(wp);
  }
```

```
  in.close();

  if(list_wp.size()) {
    list_wp.begin()->comp_factor = WordPair::CF_SRC;
    list_wp.sort(less<WordPair>());
  }

} // StrTable::read


////////////////////////////
void StrTable::write()
{
  try {
    write(filename);
  }
  catch(excpt& e) {
    throw;
  }

} // StrTable::write


////////////////////////////
void StrTable::write(const string& table_filename)
{
  filename = table_filename;

  ofstream out(filename.c_str());
  check(out.is_open(), string("can't open the file to write:")
        + filename);

  size_t write_size;
  size_t convert;
  list<WordPair>::iterator it;
  for(it = list_wp.begin(); it != list_wp.end(); ++it) {
    write_size = it->source_word.length() + 1;
    convert = it->convert_word;
```

```
    out.write( (char*)&write_size, sizeof(size_t) );
    out.write( it->source_word.c_str(), it->source_word.length() + 1 );
    out.write( (char*)&convert, sizeof(size_t) );
  }

  out.flush();
  out.close();

} // StrTable::write


////////////////////////////
void StrTable::push(const string& word)
{
  WordPair wp;
  wp.source_word = word;

  if(list_wp.size()) { // 1回以上のpushをしている場合
    if(list_wp.begin()->comp_factor != WordPair::CF_SRC) {
      list_wp.begin()->comp_factor = WordPair::CF_SRC;
      list_wp.sort(less<WordPair>());
    }

    // 挿入するべき位置を探す.
    list<WordPair>::iterator it
      = lower_bound(list_wp.begin(), list_wp.end(), wp);

    // もし重複していなかったら挿入
    // 重複していたら無視
    if(it == list_wp.end()) {
      wp.convert_word = ++count /* convert(word) */;
      list_wp.push_back(wp);

#ifdef DEBUG
      cout << "StrTable::push : " << word << " --> " << count << endl;
#endif // DEBUG

    }
```

```
    else if(it->source_word != wp.source_word) {
      wp.convert_word = ++count /* convert(word) */;
      list_wp.insert(it, wp);

#ifdef DEBUG
      cout << "StrTable::push : " << word << " --> " << count << endl;
#endif // DEBUG

    }
  }
  else {  // まだpushしていない場合
    wp.comp_factor = WordPair::CF_SRC;
    wp.convert_word = ++count /* convert(word) */;
    list_wp.push_back(wp);

#ifdef DEBUG
      cout << "StrTable::push : " << word << " --> " << count << endl;
#endif // DEBUG

  }

} // StrTable::push


////////////////////////////
bool StrTable::src2cvt(string& result, const string& source_word)
{
  if(list_wp.begin()->comp_factor != WordPair::CF_SRC) {
    list_wp.begin()->comp_factor = WordPair::CF_SRC;
    list_wp.sort(less<WordPair>());
  }

  WordPair wp;
  wp.source_word = source_word;
  list<WordPair>::iterator it
    = lower_bound(list_wp.begin(), list_wp.end(), wp);

  if(it != list_wp.end() && it->source_word == wp.source_word) {
    char tmp[10];
```

```
    int2str(it->convert_word, tmp);
    result = tmp;
    return true;
  }
  else {
    result = "";
    return false;
  }

} // StrTable::src2cvt


////////////////////////////
bool StrTable::cvt2src(string& result, const string& convert_word)
{
  if(list_wp.begin()->comp_factor != WordPair::CF_CVT) {
    list_wp.begin()->comp_factor = WordPair::CF_CVT;
    list_wp.sort(less<WordPair>());
  }

  WordPair wp;
  wp.convert_word = atoi(convert_word.c_str());
  list<WordPair>::iterator it
    = lower_bound(list_wp.begin(), list_wp.end(), wp);

  if(it != list_wp.end() && it->convert_word == wp.convert_word) {
    result = it->source_word;
    return true;
  }
  else {
    result = "";
    return false;
  }

} // StrTable::cvt2src


////////// private //////////
```

```
// initialize static method.
StrTable::WordPair::COMP_FACTOR StrTable::WordPair::comp_factor;


bool StrTable::WordPair::operator<(const WordPair& arg) const
{
  switch(comp_factor) {
  case CF_SRC:
    return (strcmp(source_word.c_str(), arg.source_word.c_str()) < 0)
      ? true : false;
  case CF_CVT:
    return convert_word < arg.convert_word;
  default:
    return false;
  }

} // operator<


bool StrTable::WordPair::operator>(const WordPair& arg) const
{
  switch(comp_factor) {
  case CF_SRC:
    return (strcmp(source_word.c_str(), arg.source_word.c_str()) > 0)
      ? true : false;
  case CF_CVT:
    return convert_word > arg.convert_word;
  default:
    return false;
  }

} // operator>


bool StrTable::WordPair::operator==(const WordPair& arg) const
{
  switch(comp_factor) {
  case CF_SRC:
```

```
    return source_word == arg.source_word;
  case CF_CVT:
    return convert_word == arg.convert_word;
  default:
    return false;
  }

} // operator==


// end
```

付 録B　決定木を使用した同音異義語判定プログラムdecideの主要部

```
/********************
     dectree.h
********************/

#ifndef __DECTREE_H__
#define __DECTREE_H__

#include <vector>
#include <list>
#include <map>
#include <string>
#include <stdio.h>
#include <uc_ptr.h>
#include <uc_object.h>
#include <stdlib.h>
#include <excpt.h>
#include <iostream>
#include <assert.h>

using namespace std;
using namespace my_excpt;

// 符合
static const string math_sign_more  = ">";
static const string math_sign_less  = "<=";
static const string math_sign_equal = "=";
```

```
//////////////////////////////
// struct DTNode
// 木の要素
struct DTNode
{
  // 要素のタイプ (連続値, 離散値, 終端)
  typedef enum { CONTINUOUS, SCAT, END, NONE } NODETYPE;
  NODETYPE node_type;

  size_t node_no; /* ノード番号
                     ダミーノードと終端ノードは持たない */
  string att;     // 条件の名前

  // 条件の値

  // C4.5の吐き出す木の通りの値を入れる.
  // att sign cont_value
  double cont_value; // 連続値の場合

  typedef enum { SIGN_MORE, SIGN_LESS, SIGN_EQUAL } SIGNTYPE;
  SIGNTYPE sign_type;

  // string sign;        // 比較符合
  string scat_value; // 離散値の場合

  string result;   // 結果

  vector<DTNode *> nextnode; // 次のノード

  DTNode() : node_type(NONE), node_no(0), att(""), cont_value(0),
             sign_type(SIGN_EQUAL), scat_value(""), result("") { }
}; // DTNode


//////////////////////////////
// class DecitionData
// 求める条件データ
class DecitionData
{
```

```
public:
  DecitionData() { }

  void push(const string& att, const string& value) {
    data_map[att] = value;
  }
  void clear() {
    data_map.clear();
  }
  const map<string, string>& get_data() const {
    return data_map;
  }
  string& operator[](const string& att) { return data_map[att]; }

private:
  map<string, string> data_map;

}; // DecitionData


////////////////////////////////
// class DecTree
// 決定木を扱うクラス
class DecTree
{
public:
  DecTree() : rep(0), total_node(0) { };

  void makeTree(const vector<string>& input_buf);
  void decide(const DecitionData& data, string& result) const;

  size_t size() const { return total_node; }

private:
  ////////////////////////////////
  // class DecTree_rep
  class DecTree_rep : public uc_object {
  public:
    vector<DTNode*> node_array;    // 一段目のノードを指す配列
```

```
    vector<size_t> indent_table;    // インデント情報テーブル
    map<string, size_t> subtree_table;  // Subtree情報テーブル

    DecTree_rep() { }
    virtual ~DecTree_rep() {
      for(size_t i = 0; i < node_array.size(); ++i)
        deleteTree(node_array[i]);
    } // ~DecTree_rep()

    void deleteTree(DTNode* node) {
      if(node->node_type != DTNode::END)
        for(size_t i = 0; i < node->nextnode.size(); ++i)
          deleteTree(node->nextnode[i]);

      delete node;
      node = NULL;
    } // deleteTree

  }; // class DecTree_rep

  uc_ptr<DecTree_rep> rep;  // rep
  size_t total_node;        // ルートノードを除いたノード数

  static const size_t npos = static_cast<size_t>(-1);


  void createIndentTable(const vector<string>& input_buf);
  void createSubtreeTable(const vector<string>& input_buf);
  void createNode(DTNode* node, const vector<string>& input_buf, size_t pos);
  void nextIndentPos(const vector<string>& input_buf, size_t parent_buf_pos,
                     vector<size_t>& result) const;
  size_t countIndent(const string& buf) const;
  size_t getAttPos(const string& buf, size_t pos = 0) const;
  bool isSatis(const DTNode* node, const DecitionData& data) const;
  const DTNode* _decide(const DTNode* node, const DecitionData& data) const;

#ifdef DEBUG
  ////////// methods for debug //////////
  void showNode(DTNode* node, size_t indent = 0) const;
```

```
  void showTree(DTNode* node, size_t indent = 0) const;
#endif // DEBUG

}; // DecTree


#endif // __DECTREE_H__

// end



/********************
    dectree.cc
********************/


#include "dectree.h"


////////// public //////////


//////////////////////////////
// DecTree::makeTree
// c4.5の出力した木を作成する
// input_bufにはc4.5の標準出力の木の部分を渡す.
// 仮に空白行があっても無視する.
void DecTree::makeTree(const vector<string>& input_buf)
{
  try {
    rep = new DecTree_rep();
  }
  catch(...) {
    rep = 0;
    throw excpt("can't allocate memory in DecTree::makeTree");
  }

  createIndentTable(input_buf);    // インデントテーブルの作成
```

```
  createSubtreeTable(input_buf);  // Subtree情報テーブルの作成

  // 一段目(rep->nextnode[0]のノードの位置を取得.
  vector<size_t> nodepos;
  size_t i;

  for(i = 0; i < rep->indent_table.size(); ++i) {
    // Subtreeの方まで処理しないように考慮する.
    if(rep->indent_table[i] == npos)
      break;
    if(rep->indent_table[i] == 0) // インデント量が0の位置
      nodepos.push_back(i);
  }

  DTNode *node = NULL;
  try {
    for(i = 0; i < nodepos.size(); ++i) {
      node = new DTNode;
      createNode(node, input_buf, nodepos[i]);
      rep->node_array.push_back(node);
    }
  }
  catch(excpt& e) {
    rep = 0;
    throw;
  }
  catch(...) {
    rep = 0;
    throw excpt("can't allocate memory in DecTree::makeTree");
  }

#ifdef DEBUG
  for(i = 0; i < rep->node_array.size(); ++i)
    showTree(rep->node_array[i]);
#endif // DEBUG

} // DecTree::makeTree
```

```
////////////////////////////
// DecTree::decide
// 求める問題が data の場合の決定木での結果を求める.
void DecTree::decide(const DecitionData& data, string& result) const
{
#ifdef DEBUG
  assert(rep != 0);
#endif // DEBUG

  // case : 決め打ち木の場合は問答無用で回答を出す.

  if(rep->node_array[0]->node_type == DTNode::END) {
    result = rep->node_array[0]->result;
    return;
  }

  // 前処理

  size_t i;
  try {
    for(i = 0; i < rep->node_array.size(); ++i)
      if(isSatis(rep->node_array[i], data))
        break;
  }
  catch(excpt& e) {
    throw;
  }

  DTNode* node = rep->node_array[i];

  bool flag = false;

  while(1) {

#ifdef DEBUG
    showNode(node, 0);
#endif // DEBUG

    if(node->nextnode[0]->node_type == DTNode::END)
```

```
      break;

    for(i = 0; flag == false && i < node->nextnode.size(); ++i) {
      try {
        if(isSatis(node->nextnode[i], data)) {
          node = node->nextnode[i];
          flag = true;
        }
      }
      catch(excpt& e) {
        throw;
      }
    }
    check(flag, "適合するノードがありません. ");

    flag = false;

  } // while

  result = node->nextnode[0]->result;
} // DecTree::decide


////////// private //////////


//////////////////////////////
// DecTree::createIndentTable
// インデント情報テーブルの構築
void DecTree::createIndentTable(const vector<string>& input_buf)
{
#ifdef DEBUG
  assert(rep != 0);
  cout << "Creating Indent Table...\n";
#endif // DEBUG

  size_t count, next_pos;

  rep->indent_table.clear();
```

```
  for(size_t i = 0; i < input_buf.size(); ++i) {

    // 空行の場合や"Subtree"の場合はnposを代入する.

    if(!input_buf[i].length() ||
       input_buf[i].find("Subtree") != string::npos) {
      rep->indent_table.push_back(npos);
    }
    else {
      count = next_pos = 0;
      while((next_pos = input_buf[i].find('|', next_pos)) != string::npos) {
        ++count;
        ++next_pos;
      }
      rep->indent_table.push_back(count);
    }

#ifdef DEBUG
    cout << "line:" << i << "\t indent:" << rep->indent_table[i] << endl;
#endif // DEBUG

  } // for

} // DecTree::createIndentTable


//////////////////////////////
// DecTree::createSubtreeTable
// Subtree情報テーブルの構築
void DecTree::createSubtreeTable(const vector<string>& input_buf)
{
#ifdef DEBUG
  assert(rep != 0);
#endif // DEBUG

  rep->subtree_table.clear();

  const string st("Subtree");
```

```
  for(size_t i = 0; i < input_buf.size(); ++i) {
    if(input_buf[i].find(st) != string::npos) {

      // subtree の名前の摘出
      size_t s_p = input_buf[i].rfind('[');
      ++s_p;
      size_t e_p = input_buf[i].find(']', s_p);
      string st_name = input_buf[i].substr(s_p, e_p - s_p);

      rep->subtree_table[st_name] = i;
    }
  }

#ifdef DEBUG
  map<string, size_t>::iterator it;
  cout << "in DecTree::createSubtreeTable.\n";
  cout << "Subtree Position:\n";
  for(it = rep->subtree_table.begin(); it != rep->subtree_table.end();
      ++it) {
    cout << (*it).first << " : " << (*it).second << endl;
  }
#endif // DEBUG

} // DecTree::createSubtreeTable


////////////////////////////
// DecTree::createNode
// input_bufから再帰的にノードを構築する.
void DecTree::createNode(DTNode* node, const vector<string>& input_buf,
                         size_t pos)
{
#ifdef DEBUG
  cout << "in DecTree::createNode:\n";
  cout << "terget buffer(line " << pos << "):" << input_buf[pos] << endl;
#endif // DEBUG

  check(input_buf[pos].length() != 0,
```

```
        "空のバッファが指定されました.  in createNode");
    size_t strpos = getAttPos(input_buf[pos], 0);
    check(strpos != npos, "can't find att posision DecTree::createNode");
    size_t temp_size = input_buf[pos].length() - strpos;

    // ノードの構築
    char* temp = NULL;
    try {
      temp = new char[temp_size + 1];
    }
    catch(...) {
      throw excpt("can't allocate temp memory in DecTree::createNode");
    }

    input_buf[pos].copy(temp, temp_size, strpos);
    temp[temp_size] = '\0';

    // case: 木が決め打ちだったらENDノードをつくって終了.
    // 決め打ちの場合は':'が出現しない.

    bool not_tree = true;
    size_t i;
    for(i = 0; i < strlen(temp); ++i) {
      if(temp[i] == ':')
        not_tree = false;
    }

    if(not_tree) {
      node->node_type = DTNode::END;

#ifdef DEBUG
    cout << "終端ノードを作成しました. \n";
#endif // DEBUG

    // 結果を摘出
    char result[50];
    sscanf(temp, "%s %*s", result);
    node->result = result;
```

```
  return;
}

// 決めうちじゃない木の場合は通常処理

char att[30];
char sign[3];
char value[30];

sscanf(temp, "%s %s %s", att, sign, value);
// valueの終端に':'があったら取り除く.
for(size_t i = 0; i < strlen(value); ++i) {
  if(value[i] == ':') {
    value[i] = '\0';
  }
}

node->node_no = pos;
node->att = att;

if(sign != math_sign_equal) {  // 連続値の場合
  node->node_type = DTNode::CONTINUOUS;
  node->cont_value = atof(value);
  if(sign == math_sign_more)
    node->sign_type = DTNode::SIGN_MORE;
  else
    node->sign_type = DTNode::SIGN_LESS;
}
else { // 離散値の場合
  node->node_type = DTNode::SCAT;
  node->scat_value = value;
  node->sign_type = DTNode::SIGN_EQUAL;
}

// 子ノードの構築

// 次のたどるべきinput_buf内の位置を取得
vector<size_t> nodepos;
nextIndentPos(input_buf, pos, nodepos);
```

```
#ifdef DEBUG
  cout << "parent pos:" << pos << endl;
  cout << "next pos...\n";
  for(size_t i = 0; i < nodepos.size(); ++i)
    cout << nodepos[i] << endl;
  cout << endl;
#endif // DEBUG

  // 子ノードがない場合は終端ノードを作成
  DTNode* temp_node;
  if(!nodepos.size()) {
    try {
      temp_node = new DTNode;
    }
    catch(...) {
      throw excpt("can't allocate memories for an end node in DecTree::createNode'
    }

    temp_node->node_type = DTNode::END;
#ifdef DEBUG
    cout << "create End Node\n";
#endif // DEBUG

    // 結果を摘出
    size_t result_pos;
    for(result_pos = 0; result_pos < temp_size; ++result_pos)
      if(temp[result_pos] == ':')
        break;
    check(result_pos != temp_size,
          "can't find result! in DecTree::createNode");
    ++result_pos;

    char result[30];
    sscanf(temp + result_pos, "%s", result);
    temp_node->result = result;

    node->nextnode.push_back(temp_node);
  }
```

```
  else {   // 子ノードがあった場合
    for(size_t i = 0; i < nodepos.size(); ++i) {
      try {
        temp_node = new DTNode;
        createNode(temp_node, input_buf, nodepos[i]);
      }
      catch(excpt& e) {
        throw;
      }
      catch(...) {
        throw excpt("can't allocate memories for an new node in DecTree::createNo
      }

      node->nextnode.push_back(temp_node);
    }
  }

  delete temp; temp = NULL;
} // DecTree::createNode


////////////////////////////////
// DecTree::nextIndentPos
// 親ノードの位置parent_buf_posに対する子ノードの位置リストを求める.
void DecTree::nextIndentPos(const vector<string>& input_buf,
                            size_t parent_buf_pos,
                            vector<size_t>& result) const
{
#ifdef DEBUG
  assert(rep != 0);
#endif // DEBUG

  size_t buf_pos = parent_buf_pos;
  size_t parent_indent = rep->indent_table[parent_buf_pos];
  result.clear();

  // Subtreeがある場合の処理

  size_t s_p;
```

```
  if((s_p = input_buf[parent_buf_pos].rfind('[')) != string::npos) {
    ++s_p;
    size_t e_p = input_buf[parent_buf_pos].find(']', s_p);
    string st_name = input_buf[parent_buf_pos].substr(s_p, e_p - s_p);

    // Subtreeの開始位置
    size_t st_pos = rep->subtree_table[st_name] + 2;

#ifdef DEBUG
    cout << "in DecTree::nextIndentPos.\n";
    cout << "find Subtree option.(line " << parent_buf_pos << ")\n";
    cout << "subtree name:" << st_name << endl;
#endif

    // Subtreeの先頭ノードのインデントは0
    while(1) {
      // "Subtree"以後，空行で終了.
      if(st_pos >= input_buf.size() || !input_buf[st_pos].length()) break;
      if((input_buf[st_pos])[0] != '|')
        result.push_back(st_pos);
      ++st_pos;
    }
    return;
  }

  // 通常の処理

  size_t current_indent;
  while(++buf_pos != rep->indent_table.size()) {
    current_indent = rep->indent_table[buf_pos];

    // 別の木か場合は終了
    if(current_indent <= parent_indent)
      return;

    if(current_indent == parent_indent + 1)
      result.push_back(buf_pos);
  }
```

```
} // DecTree::findEachIndentPos


//////////////////////////////
// DecTree::countIndent
// pos の位置のインデント数を返す.
size_t DecTree::countIndent(const string& buf) const
{
  size_t count = 0;
  size_t next_pos = 0;
  while( (next_pos = buf.find('|', next_pos)) != string::npos ) {
    ++count;
    ++next_pos;
  }

  return count;
} // DecTree::countIndent


//////////////////////////////
// DecTree::getAttPos
// buf 内の pos 以降における次の att, 等・不等号, 値の位置を返す.
size_t DecTree::getAttPos(const string& buf, size_t pos) const
{
  while(pos < buf.length()) {
    if(buf[pos] != ' ' && buf[pos] != '|')
      return pos;
    ++pos;
  }
  return npos;
} // DecTree::getAttPos


//////////////////////////////
// DecTree::isSatis
// node の条件が data の条件を満たすかどうかを調べる.
// 満たしていれば true, 満たさない場合は false
bool DecTree::isSatis(const DTNode* node, const DecitionData& data) const
{
```

```
    check(node->node_type != DTNode::END,
          "exception in tracing node: here is END node");

    check(data.data_map.find(node->att) != data.data_map.end(),
          string("can't find attribute in decition data: ") + node->att);

    double data_value = 0;
    map<string, string>::const_iterator cit;

    switch(node->node_type) {
    case DTNode::CONTINUOUS : // 連続値の場合
      // data_value = atof(data.data_map[node->att].c_str());
      cit = data.data_map.find(node->att);
      data_value = atof((*cit).second.c_str());

      switch(node->sign_type) {
      case DTNode::SIGN_MORE:
        if(node->cont_value > data_value)
          return true;
        break;
      case DTNode::SIGN_LESS:
        if(node->cont_value <= data_value)
          return true;
        break;
      default:
        break;
      } // switch(node->sign)

    case DTNode::SCAT: // 離散値の場合
      cit = data.data_map.find(node->att);
      check(cit != data.data_map.end(), "条件が見つからないよ");

#ifdef DEBUG
    cout << "In isStatus...";
    cout << "att: " << node->att << endl;
    cout << "data value(0 or 1): " << (*cit).second << endl;
    cout << "scat_value(0 or 1): " << node->scat_value << endl << endl;
#endif // DEBUG
```

```
    if((*cit).second == node->scat_value) {
#ifdef DEBUG
      cout << "data_value == scat_value   match!\n";
#endif // DEBUG
      return true;
    }
#ifdef DEBUG
    else
      cout << "data_value != scat_value\n";
#endif // DEBUG

    break;

  default:
    break;
  } // switch(node->node_type)

  return false;

} // DecTree::isSatis


////////////////////////////
// DecTree::_decide
const DTNode* DecTree::_decide(const DTNode* node,
                               const DecitionData& data) const
{
  // このノードがENDノードならストップ
  if(node->node_type == DTNode::END)
    return node;

  // ENDノードでない場合，条件に合う次のノードを探す.
  size_t i;
  for(i = 0; i < node->nextnode.size(); ++i)
    if(isSatis(node->nextnode[i], data))
      return _decide(node->nextnode[i], data);

  return NULL;
} // DecTree::_decide
```

```
#ifdef DEBUG
////////// Methods for Debug //////////


//////////////////////////////
// DecTree::showNode()
// ノードを表示する(DEBUG用メソッド)
void DecTree::showNode(DTNode* node, size_t indent) const
{
  assert(node != NULL);
  string space;
  for(size_t i = 0; i < indent; ++i)
    space += ' ';

  cout << space << "node_type     :" << (int)node->node_type << endl;
  cout << space << "node_no       :" << (int)node->node_no   << endl;
  cout << space << "node_att      :" <<  node->att  << endl;
  cout << space << "cont_value    :" << node->cont_value<< endl;
  cout << space << "scat_value    :" << node->scat_value<< endl;
  cout << space << "sign          :" << (int)node->sign_type << endl;
  cout << space << "result        :" << (int)node->node_type << endl;
  cout << space << "nextnode size :" << (int)node->nextnode.size() << endl;
  cout << endl;
} // DecTree::showNode


//////////////////////////////
// DecTree::showTree()
// 木を表示する(DEBUG用メソッド)
void DecTree::showTree(DTNode* node, size_t indent) const
{
  showNode(node);

  if(node->node_type == DTNode::END) {
    cout << node->result << endl << endl << flush;
    return;
```

```
    }

    size_t i;
    if(node->node_type != DTNode::END) {
      for(i = 0; i < rep->indent_table[node->node_no]; ++i)
        cout << "|   ";

      cout << node->att << " ";

      switch(node->sign_type) {
      case DTNode::SIGN_MORE :
        cout << math_sign_more << " ";
        break;
      case DTNode::SIGN_LESS :
        cout << math_sign_less << " ";
        break;
      case DTNode::SIGN_EQUAL:
        cout << math_sign_equal << " ";
        break;
      }

      switch(node->node_type) {
      case DTNode::CONTINUOUS:
        cout << node->cont_value << " :";
        break;
      case DTNode::SCAT:
        cout << node->scat_value << " :";
        break;
      default: // DTNode::END
        break;
      }
    }

    for(i = 0; i < node->nextnode.size(); ++i){
      showTree(node->nextnode[i], indent + 1);
    }

  } // DecTree::showTree()
```

```
#endif // DEBUG


// end
```

関連図書

[1] 新納浩幸：“複合語からの証拠に重みをつけた決定リストによる同音異義語判別”, 情報処理学会, **Vol.39, No.12**, pp.3200–3205 (1998).

[2] 秋葉泰弘 フセイン・アルモアリム 金田重郎：“例からの学習技術の応用に向けて”, 情報処理学会, **Vol.39, No.2**, pp.145–151 (1998).

[3] J.R. キンラン＝著 古川康一＝監訳：“AIによるデータ解析”, トッパン (1993).

[4] Yarowsky, D 著：“Decision list for lexical ambiguity resolution”, 32th Annual Meeting of the Association for Computational Linguistics, pp.88–95(1994).