ナイーブベイズ分類器 (Naive Bayes Classifier) の GoLang による実装

最終更新日：2020/01/05

---

1. はじめに

「分類器」とは、ある文書を入力するとその文書が属するカテゴリを推定し出力するシステムである。本稿はこの分類器をナイーブベイズを用いたモデルで示し、そして GoLang による実装例を示す。

本稿では多項分布のパラメータ推定として最尤推定値 (MLE) と 期待事後確率推定値 (EAP 推定値) を使用するが、具体的な計算については別紙を参照されたい。

参考：多項分布のパラメータ推定

---

2. 推定

本節では、与えられた文書に対するカテゴリの推定について示す。

2.1 PredictCat (暫定版)

ある文書 D が与えられたときに、それがカテゴリ C に属する確率を $P(C|D)$ と表す。

カテゴリが n 個あり、$C_1, \ldots, C_i, \ldots, C_n$ のいずれかのとき、 ある文書 D の属するカテゴリは $P(C_i|D)$ が最大となる C_i で与える。

例：ニュースのカテゴリ群が $\{$社会,政治,国際,スポーツ,科学$\}$ の 5 種類とする。 あるニュースの文書 D が与えられそれぞれのカテゴリの確率が以下のとき、 D の属するカテゴリは $P(C_i|P)$ が最大の値となる「スポーツ」となる。

C_i	$P(C_i\|D)$
社会	0.12
政治	0.17
国際	0.05
スポーツ	0.51
科学	0.15

$P(C_i|D)$ を求める関数を ProbCatGivenDoc とする。 ある文書がどのカテゴリに属するかを推定する関数 PredictCat は すべてのカテゴリについて ProbCatGivenDoc を計算していき、 その値が最大となるときのカテゴリを決定すればよいので、 実装は次のようになる。

```
// catList : カテゴリのリスト
var catList []TypeCat

// PredictCat : 与えられた文書 doc のカテゴリを推定する関数。暫定版。
func PredictCat(doc TypeDoc) (cat TypeCat) {
        cat = catList[0]
        maxValue := ProbCatGivenDoc(doc, cat)
        for i:=1; i<len(catList); i++ {
                result := probCatGivenDoc(doc, catList[i])
                if result > maxValue {
```

```
                        maxValue = result
                        cat = catList[i]
                }
        }
        return
}
```

上の実装は暫定版である。後ほど最終版を示す。

2.2 ProbCatGivenDoc (暫定版) --- $P(C_i|D)$

ある文書 D がカテゴリ C_i に属する確率 $P(C_i|D)$ はベイズの定理により次のように表される。

$$P(C_i|D) = \frac{P(C_i)P(D|C_i)}{P(D)}$$

ここで、$P(C_i), P(D|C_i), P(D)$ はそれぞれ次の確率を示す。

$P(C_i)$	カテゴリー C_i である確率
$P(D\|C_i)$	カテゴリー C_i に文書 D が含まれる確率
$P(D)$	文書 D が成立する確率

また $P(D)$ は以下を満たす。

$$P(D) = \Sigma_i\{P(C_i)P(D|C_i)\}$$

従って、$P(D)$ は C_i に関係なく固定の値であること、また、 カテゴリの推定は $P(C_i|D)$ の大小関係のみに基づいていることから、次の比例関係にのみ注目すればよい。

$$P(C_i|D) \propto P(C_i)P(D|C_i)$$

$P(C_i)$ を計算する関数 ProbCat と $P(D|C_i)$ を計算する関数 ProbDocGivenCat があるとすれば、 $P(C_i|D)$ の「比」を計算する関数 ProbCatGivenDoc の実装は次のような形になる。

```
// ProbCatGivenDoc : 文書 doc がカテゴリ cat に含まれる確率の比を求める関数。暫定版。
func ProbDocGivenCat(doc TypeDoc, cat TypeCat) float64 {
        return ProbCat(cat) * ProbDocGivenCat(doc, cat)
}
```

上の実装は暫定版である。後ほど最終版を示す。

2.3 ProbCat --- $P(C_i)$

$P(C_i)$ はカテゴリー C_i である確率である。 ここでは全文書における、カテゴリー C_i に属す文書の割合とみなし、 単純に文書の個数の割合で考えることにする。 $P(C_i)$ を求める関数 ProbCat は次の実装で与えることができる。

```
var numAllDocs int // すべての文書の数
var numDocsCat map[TypeCat]int // 各カテゴリごとの文書の数

// ProbCat : カテゴリ cat の確率＝カテゴリ cat の文書の全文書に対する割合
func ProbCat(cat TypeCat) float64 {
    return float64(numDocsCat[cat]) / float64(numAllDocs)
}
```

2.4 ProbDocGivenCat --- $P(D|C_i)$

$P(D|C_i)$はカテゴリー C_i （に属する文書群）に文書 D が含まれる確率である。 しかし文書 D がカテゴリーに属する文書と一致するケースがほぼないと考えられるため、 文書数だけでは計算することができない。 ここで文書 D に出現する単語群 w_j に注目し、次の仮定をおくことにする。

- 文書は単語の並びである
- 文書中にある単語が現れる確率は他の単語が現れる確率に依存せず独立である
- 文書中にある単語が現れる確率は文書中の位置に依存しない

確率 $P(w_j|C_i) = \theta_j$ を、文書 D に含まれる単語 w_j がカテゴリ C_i に出現する確率とし、単語 w_j が文書 D に出現する個数を n_j とすれば、 $P(D|C_i)$ は次のような多項分布関数で表される。

$$P(D|C_i) = f(n_1, \cdots, n_m; \theta_1, \cdots, \theta_m) = \theta_1^{n_1} \cdots \theta_m^{n_m}$$

$$(1 \leq j \leq m,\ 0 < \theta_j < 1,\ \Sigma_j \theta_j = 1)$$

確率 $P(w_j|C_i) = \theta_j$ を求める関数を ProbWordGivenCat とするとき、 $P(D|C_i)$ を求める関数 ProbDocGivenCat は次のようになる。

```
// TypeDoc : 文書の型。各単語の出現個数のmap。
type TypeDoc map[TypeWord]int
// 例：doc[word] = 単語 word が文書 doc に含まれる個数

// ProbDocGivenCat : 文書 doc がカテゴリ cat に含まれる確率
func ProbDocGivenCat(doc TypeDoc, cat TypeCat) (r float64) {
        r = 1.0
        for word, num := range doc {
                r *= math.Pow(ProbWordGivenCat(word, cat), float64(num))
        }
        return
}
```

2.5 ProbWordGivenCat --- $P(w_j|C_i)$

単語の出現確率 $P(w_j|C_i) = \theta_j$ は$P(D|C_i)$ のパラメータである。実測した単語 w_j の出現数n_jをもとに、$P(D|C_i)$ を下に示すような尤度関数 $L(\theta_1, \cdots, \theta_m)$ とみなし、これが最大になる$\theta_1, \cdots, \theta_m$ を推定することになる。

$$L(\theta_1, \cdots, \theta_m) = P(D|C_i) = \theta_1^{n_1} \cdots \theta_m^{n_m}$$

まず最尤推定値 (Maximum Likelihood Estimator) を使うこととして確率 $P(w_j|C_i) = \theta_j$ を推定する関数 ProbWordGivenCat の実装を考えるが、ここで２つの案が考えられる。

- **案1**：カテゴリー C_i に属する文書に出現する単語群に、文書 D に出現する単語群がどれだけ含まれるかで考える。

$$P(w_j|C_i) = \theta_j = \frac{\text{単語 } w_j \text{ が } C_i \text{ に属す文書における出現回数の合計}}{C_i \text{ に属す文書に出現する全単語の出現回数の合計}}$$

- **案2**：文書 D に含まれる各単語を含む文書がどれだけカテゴリー C_i に属しているかで考える。

$$P(w_j|C_i) = \theta_j = \frac{\text{単語 } w_j \text{ を含みかつ } C_i \text{ に属す文書の個数}}{C_i \text{ に属す全文書の個数}}$$

一つの文書には異なる単語が複数含まれることが簡単に予想されることから、案 2 では $\Sigma_j \theta_j = 1$ の条件を満たすことができない。ここでは案 1 を実装する。 numWordInCat[C_i][w_j] を文書 D に含まれる単語 w_j がカテゴリ C_i に含まれる個数とすれば、$P(w_j|C_i)$ を計算する関数 ProbWordGivenCat は次のように与えることができる。

```
// numWordInCat: ある単語があるカテゴリに含まれる個数
var numWordInCat map[TypeCat]map[TypeWord]int
// 例：numWordInCat[cat][word] = 単語 word がカテゴリ cat に含まれる個数

// numAllWordsInCat : カテゴリに含まれる個数
var numAllWordsInCat map[TypeCat]int
// 例：numAllWordsInCat[cat] = カテゴリ cat に含まれる単語の個数

// ProbWordGivenCat : 単語 word がカテゴリ cat に含まれる確率
func ProbWordGivenCat (word TypeWord, cat TypeCat) float64 {
    return float64(numWordInCat[cat][word])/float64(numAllWordsInCat[cat])
}
```

2.6 ProbWordGivenCat (スムージング拡張版) --- $P(w_j|C_i)$

上の実装では、文書の中に一つでもカテゴリ C_i に含まれない単語が存在すると、他の単語の確率が高いものだったとしても、全体として $P(D|C_i)$ が 0 となってしまうという問題がある。 これを回避するため「加算スムージング」（あるいは「ラプラススムージング」）を使う。重複のない全単語の個数を m とする。

$$P(w_j|C_i) = \theta_j = \frac{\text{単語 } w_j \text{ が } C_j \text{ に属す文書における出現回数の合計 } +1}{C_j \text{ に属す文書に出現する全単語の出現回数の合計 } +m}$$

これは期待事後確率推定値 (Expected a Posterior Estimator; EAP 推定値) に相当する。 この推定値は特に標本数が少ない場合に効果があり、標本数が増えるにつれて先の最尤推定値に近づいていく。

```
var numAllWords int // 全単語数

// ProbWordGivenCat : 単語 word がカテゴリ cat に含まれる確率(スムージング拡張版)
func ProbWordGivenCat (word TypeWord, cat TypeCat) float64 {
    num := float64(numWordInCat[cat][word] + 1)
    sum := float64(numAllWordsInCat + numAllWords)
    return num/sum
}
```

上記スムージングを施しても $\Sigma_j \theta_j = 1$ の条件を満たすことに注意。

LogProbDocGivenCat --- $\log P(D|C_i)$

上の関数 ProbDocGivenCat の実装では、単語数が多いと分母の値が非常に大きくなりアンダーフローが起きる恐れがあるので、これを回避すべく対数をとる。

$$\log P(D|C_i) = \log f(n_1, \cdots, n_m; \theta_1, \cdots, \theta_m) = n_1 \log \theta_1 + \cdots + n_m \log \theta_m$$

```
// TypeDoc : 文書の型。各単語の出現個数のmap。
type TypeDoc map[TypeWord]int
// 例：doc[word] = 単語 word が文書 doc に含まれる個数
```

```
// LogProbDocGivenCat : 文書 doc がカテゴリ cat に含まれる確率の対数
func LogProbDocGivenCat(doc TypeDoc, cat TypeCat) (r float64) {
        r = 0.0
        for word, num := range doc {
                r += float64(num) * math.Log(ProbWordGivenCat(word, cat))
        }
        return
}
```

2.7 LogProbCatGivenDoc --- $\log P(C_i|D)$

以上を踏まえると、冒頭に示した関数 ProbCatGivenDoc のアンダーフローを考慮した対数版 LogProbCatGivenDoc は次のようになる。

```
// LogProbCatGivenDoc : 文書 doc がカテゴリ cat に含まれる確率の比の対数
func LogProbDocGivenCat(doc TypeDoc, cat TypeCat) float64 {
        return math.Log(ProbCat(cat)) + LogProbDocGivenCat(doc, cat)
}
```

2.8 PredictCat (最終版)

冒頭に示した関数 PredictCat は LogProbCatGivenDoc を使って次のように実装される。

```
// PredictCat : 与えられた文書 doc のカテゴリを推定する
func PredictCat(doc TypeDoc) (cat TypeCat) {
        cat = catList[0]
        maxValue := LogProbCatGivenDoc(doc, cat)
        for i:=1; i<len(catList); i++ {
                result := LogProbCatGivenDoc(doc, catList[i])
                if result > maxValue {
                        maxValue = result
                        cat = catList[i]
                }
        }
        return
}
```

---

3. 学習

本節では 2 節に示したカテゴリの推定に必要なデータを作成するための「学習」について示す。

3.1 Train

教師データとして、文書 doc とそのカテゴリ cat が与えられたとする。 使用する変数に対して、以下の表に示す処理を実施する必要がある。

変数	概要	処理	直接的に依存する関数
var catList []TypeCat	カテゴリのリスト	cat が初出のときのみリストに追加	PredictCat
var numAllDocs int	すべての文書の数	numAllDocs の値をインクリメント	ProbCat

var numDocsCat map[TypeCat]int	各カテゴリごとの文書の数	numDocsCat[cat] の値をインクリメント	ProbCat
var numWordInCat map[TypeCat]map[TypeWord]int	ある単語があるカテゴリに含まれる個数	文書 doc に含まれるすべての単語 word について、numWordInCat[cat][word] の値をインクリメント	ProbWordGivenCat
var numAllWordsInCat map[TypeCat]int	カテゴリに含まれる単語の個数	文書 doc に含まれる単語の個数だけ、numAllWordsInCat[cat] の値をインクリメント	ProbWordGivenCat
var numAllWords int	全単語数(重複なし)	文書 doc について wordList を更新後、len(wordList) の値を代入。つまり、これまでに出現した重複のないすべての単語の個数を代入	ProbWordGivenCat
var wordList map[TypeWord]int	単語のリスト、各単語の出現数	文書 doc に含まれるすべての単語 word について、wordList[word] の値をインクリメント	Train

与えられた文書とカテゴリで学習する関数 Train の実装例は次のようになる。

```
// wordList : 単語のリスト
var wordList map[TypeWord]int

// Train : 文書 doc をカテゴリ cat として学習する
func Train(doc TypeDoc, cat TypeCat) {

	// カテゴリ cat が初出かどうか検査する
	_, ok := numDocsCat[cat]
	if !ok { // カテゴリ cat が初出の場合
		// カテゴリリストに追加
		catList = append(catList, cat)
		// カテゴリの文書を初期化
		numWordInCat[cat] = map[TypeWord]int{}
	}

	// すべての文書の数をインクリメント
	numAllDocs++

	// カテゴリ cat の文書の数をインクリメント
	numDocsCat[cat] = numDocsCat[cat] + 1

	// 文書 doc に出現する単語 word についてそれぞれ処理
	for word, num := range doc {
		// カテゴリ cat に含まれる単語の個数をインクリメント
		numAllWordsInCat[cat] = numAllWordsInCat[cat] + num
		// 単語 word がカテゴリ cat に含まれる個数をインクリメント
		numWordInCat[cat][word] = numWordInCat[cat][word] + num
		// 単語 word を単語リストに追加。
		wordList[word] = wordList[word] + num // 単語の出現回数
	}
```

```
        // すべての単語の重複のない個数を計算
        numAllWords = len(wordList)
}
```

---

4. 評価

本稿に示したカテゴリの推定方式について、Livedoor ニュースコーパスを用いて精度を測定してみた。9 つのカテゴリについて、Precision, Recall, F-Measure, Accuracy の指標をまとめた表を以下に示す。

カテゴリ	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
Precision	0.958084	0.898734	0.860465	0.930233	0.889535	0.932886	0.857143	0.937500	0.952941
Recall	1.000000	0.940397	0.907975	0.909091	0.987097	0.822485	0.814815	0.967742	0.885246
F-Measure	0.978593	0.919094	0.883582	0.919540	0.935780	0.874214	0.835443	0.952381	0.917847
Accuracy	0.995251	0.983039	0.973541	0.981004	0.985753	0.972863	0.964722	0.989824	0.980326

全体の指標ををまとめた表を以下に示す。

Micro Precision	0.913161
Micro Recall	0.913161
Micro F-Measure	0.913161
Macro Precision	0.913058
Macro Recall	0.914983
Macro F-Measure	0.914019
Overall Accuracy	0.980703

単純な実装内容にも関わらず、高い精度でカテゴリ推定できることがわかった。

参考：Livedoor ニュースコーパス

---

5. おわりに

　本稿では、文書のカテゴリを推定する分類器をナイーブベイズを用いたモデルで示し、そして GoLang による実装例を示した。また、Livedoor ニュースコーパスを用いて評価した結果を示した。

　今後はマルチラベル分類器に拡張していく予定である。