2017/4/21

システム創成プロジェクトⅠ（画像認識） 演習資料

九州工業大学 情報工学部 システム創成情報工学科
演習担当：尾下 真樹、齊藤 剛史、徳永 旭将、宮野 英次

1. 本演習の目的

本演習では、手書き文字画像を認識するプログラムを作成する。本演習の主な目的は以下の２つである。
・ 前の講義で学んだ特徴量を使ってデータを判別する基本的な方法について、実際にプログラムを作成してみることによって、より理解を深める。
・ 文字画像認識プログラムの開発を通じて、実践的なプログラムの開発方法を学ぶ。特に本演習では、ある程度規模の大きいプログラムを開発するときに、どのようにクラス設計などを行っていけば良いかというオブジェクト指向のソフトウェア設計の考え方も学習する。

2. プログラムの仕様

プログラムを設計・開発するときには、まず、自分がこれからどのようなプログラムを作成しようとしているのかをきちんと整理することが必要である。特に、そのプログラムが何を入力データとして、内部でどのような処理を行い、最終的にどのようなデータを出力するのか、というデータの流れが重要である。
以下、具体的なクラス設計に入るまえに、これから作ろうとするプログラムの概要を整理してみる。

プログラムに必要とされる機能
- サンプル画像を読み込んで文字画像認識テストを実行
- 特徴量の計算方法を変更
- 閾値の計算方法を変更
- 全サンプル画像の特徴量の分布や閾値をグラフで表示（確認用）
- 各サンプル画像の特徴量を表示（確認用）

文字画像認識テストの処理の流れ（シナリオ）
入力データ：
　学習用画像（グループ０の画像集合とグループ１の画像集合）
　評価用画像（グループ０の画像集合とグループ１の画像集合）
出力データ：
　誤認識率
処理手順
1. 学習処理
　1-1. 両グループの各学習用画像の特徴量を計算する
　1-2. 特徴量の分布から、２つのグループを識別するための閾値（識別境界）を計算する
2. 認識処理
　各評価用画像の特徴量を計算して、閾値（識別境界）から、どちらのグループに属するかを判定
3. どれだけの画像が間違ったグループに属すると判定されたかの誤認識率を計算して出力

基本的には、あらかじめ用意された画像データをもとに自動的に認識文字画像処理を行い、結果として誤認識率を表示する。また、特徴量の計算方法や、閾値の計算方法として複数の方法があるので、これらを変更しながら比較できるようにする。さらに、プログラムの動作確認や、より細かい結果の評価のために、各

文字画像の特徴量の計算結果や全画像の特徴量の分布と閾値などを表示確認できる機能も加える（図 1)。
今回は全体のプログラムを Java アプレットとして作成する。テスト用の画像データとして、「８」「Ｂ」のそれぞれ 55 枚の手書き文字をスキャンしたファイル（GIF 形式、256×256 ドット）を用意する。なお、このように、正確な評価を行うためには本来は学習用画像と評価用画像を分けることが望ましいが、とりあえず今回のプログラムでは、同じ画像ファイルを学習用と評価用に使用することとする。

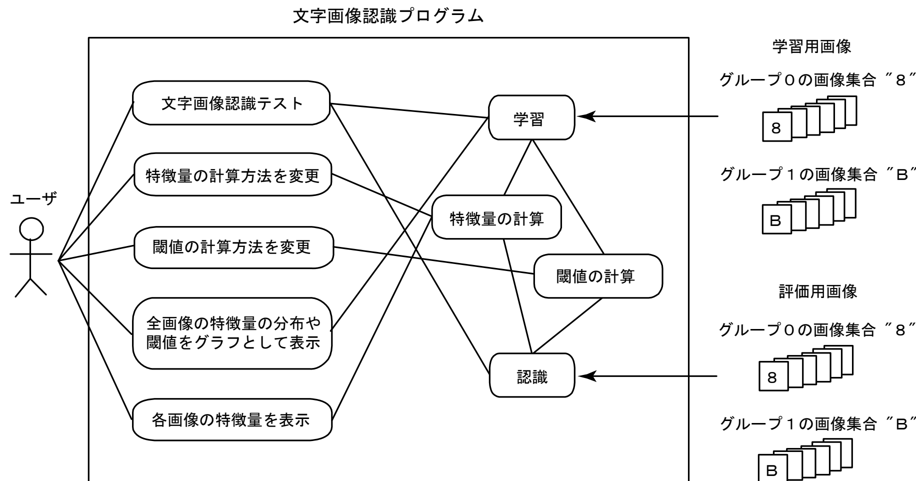

図 1　プログラムに必要な各機能の関係（ユースケース図）

3. プログラムの設計

3.1. 全体の設計

全体の設計として、まず、プログラム全体を、文字画像認識を行う機能を持ったクラスと、その機能をテストするためのクラスに大きく分けることにする。一般にプログラム設計時には、後からなるべく再利用しやすいようにクラス設計を行うことが望ましい。そこで、文字画像認識の機能だけを独立したクラスとして設計する（CharecterRecognizer クラスとする)。それに加えて、今回のテストプログラムを実現するアプレットクラス（RecognizerApp クラスとする）を作成し、ユーザの操作などに応じてそのクラスから文字画像認識クラスの処理を呼び出すようにする。
ここで重要なのは、それぞれのクラスの役割と、インターフェースをきちんと決めることである。ここが正しく決まっていないと後の設計を進めることができない。また、このような基本的な部分を後から変更しようとすると大幅な修正が必要になってしまう。

図 2 および 図 3 に、上記の考え方にもとづいてクラス設計を行った例を示す。
図 2 は、クラス間の関連、および、文字画像認識クラスの主要なメソッドを示した図（クラス図）である。JApplet クラスと BufferedImage クラスは、それぞれ Java の標準ライブラリのクラスである。RecognitionApp クラスは、アプレットであるので、JApplet クラスから派生させる。また、RecognitionApp クラスは、画像認識に使用する画像オブジェクト（BufferedImage クラスのオブジェクト）の配列をデータとして持つ。BufferedImage クラスについては、次節で説明する。画像認識クラス（CharacterRecognizer クラス）の主要な機能は、２種類の画像の集合から学習を行う処理（train() メソッド）と、学習結果に基づき新たに画像が与えられたときにその画像がどちらに属するかを認識する処理（recognize() メソッド）の２つであると考えられるので、それぞれの処理に相当するメソッドを定義する。
図 3 は、これらのクラスを使用して認識処理を行うときの処理の流れを表した図（シーケンス図）である。RecognitionApp クラスは、初期化処理・画像データの読み込みを行った後、読み込んだ学習用画像を引数として CharacterRecognizer.train() メソッドを呼び出し、学習処理をおこなう。その後、評価用画像のそれぞれ画像について CharacterRecognizer.recognize()メソッドを呼び出して認識処理を行い、どの程度の精度で認識できたかを計算し、画面に表示する。

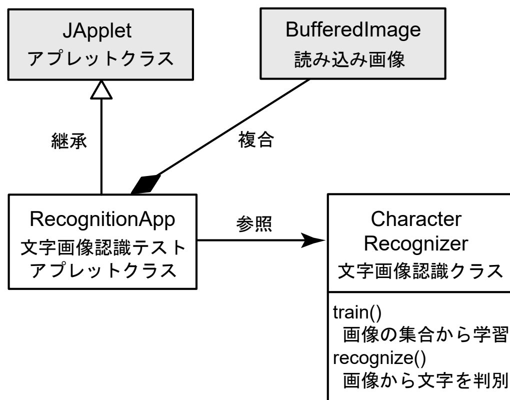

図 2　クラス構成（クラス図）

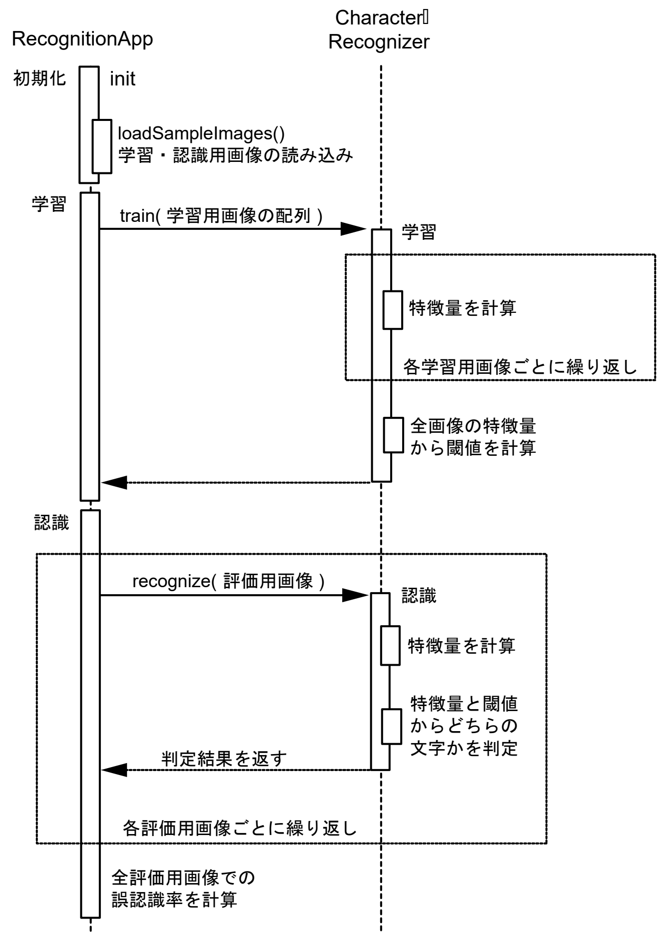

図 3　主な処理の流れ（シーケンス図）

なお、図 2 や 図 3 は、UML（Unified Modeling Language）という記述法に従って、プログラムの設計を分かりやすく整理したものである。UML は、オブジェクト指向プログラムの設計を記述するための標準的な表記法で、最近、ソフトウェア開発の分野で広く使われつつある。UML の書き方を理解しておけば、プログラムの設計時に自分の考えを分かりやすく整理したり、また、多人数で設計を検討するときにスムーズに意思疎通を行うことができる。最近は UML 関連の書籍も多数出版されているので、興味があれば勉強してみると良い。

以上の設計に従って、２つのクラスの定義を記述すると以下のようになる。

```
//
//  文字画像認識テスト アプレット
//
public class RecognitionApp extends JApplet
{
	// 文字画像認識モジュール
	protected CharacterRecognizer   recognizer;

	// サンプル画像
	protected BufferedImage   sample_images0[];
	protected BufferedImage   sample_images1[];

	// 初期化処理
	public void   init()
	{
		// 文字画像判別モジュールの生成
		recognizer = new CharacterRecognizer();
```

```
			// 全サンプル画像の読み込み
			loadSampleImages();

			// サンプル画像を使った文字画像認識のテスト
			recognitionTest();
		}

		// 描画処理
		public void	paint( Graphics g )
		{
			// 誤認識率を画面に描画する
		}

		// サンプル画像の読み込み
		public void	loadSampleImages()
		{
			// サンプル画像を読み込む処理をここに記述
			// sample_images0, sample_images1 にサンプル画像の配列を格納する
		}

		// サンプル画像を使った文字画像認識のテスト
		public void	recognitionTest()
		{
			// サンプル画像をもとに文字画像認識オブジェクトの学習を実行
			recognizer.train( sample_images0, sample_images1 );

			// サンプル画像を使って誤認識率を調べる
			// sample_images0, sample_images1 の各画像につき
			// recognizer.recognize( image ) メソッドを呼び出して
			// 正しく判定できるかどうかを調べる
		}
}


//
//	文字画像認識クラス
//
class	CharacterRecognizer
{
		// 与えられた２グループの画像データを判別するような特徴量の閾値を計算
		public void	train( BufferedImage[] images0, BufferedImage[] images1 )
		{
			// 各入力画像の特徴量を計算

			// 特徴量の分布をもとに２つのグループを識別するような閾値を計算

		}

		// 学習結果に基づいて与えられた画像を判別
		//（判別した画像の種類として 0 or 1 の値を返す）
		public int	recognize( BufferedImage image )
		{
			// 入力画像の特徴量を計算

			// 閾値に基づいてどちらのグループに属するかを判定

			return	0;
		}
}
```

3.2. Java の画像クラスの利用

Java 標準ライブラリには、画像を扱うためのクラスがいくつか存在する。最も単純な Image クラス（java.awt.Image）は、画像オブジェクトを画面に描画するといった単純な機能は利用できるが、内部のピクセルデータにアクセスするような高度な操作はできない。本プログラムでは、画像の特徴量を計算するた

めに、読み込んだサンプル画像のピクセルデータにアクセスする必要がある。そこで、Image クラスの代わりに、このような用途に適した BufferedImage クラス（java.awt.image.BufferedImage）を利用することにする。

また、画像を読み込むための方法として、Java の Image I/O ライブラリを使用する。Applet クラスにも、画像を読み込むためのメソッドとして getImage() が存在するが、対応している画像フォーマットは gif, jpg, PNG のみで、BMP 画像などは読み込むことができない。後々の実験では、BMP 画像を使用できた方が都合が良いので、BMP 画像の読み込みにも対応した Image I/O ライブラリを使用することにする。

ただし、古い Java で、Image I/O ライブラリを使って BMP 画像を読み込むためには、Java に Advanced Imaging Image I/O Tools プラグインを追加する必要があった（プラグインを追加しない状態では、Image I/O ライブラリも gif, jpg, PNG しか読み込めなかった）。CL の端末にインストールされている最新のバージョンの Java では、最初からプラグインが組み込まれているので、インストールは不要である。CL 意外の古い Java の環境で演習を行いたいは、自分でプラグインをインストールする必要がある。

3.3. 文字画像認識クラスの設計

ここまでの設計では、文字画像認識クラス（CharacterRecognizer）の内部で、特徴量や閾値の計算を行なうことになっている。最初の概要の説明で挙げたように、特徴量や閾値の計算方法については、いくつかの方法を切り替えられるようにしたい。そこで、それぞれの機能を抽象インターフェースとして分離して、各手法を実装した別々のクラスを継承することにする。画像認識クラスは、各インターフェースを実装したオブジェクトへの参照を持つ。このオブジェクトを変更することで、文字画像認識クラスの方は修正することなく、特徴量計算方法や閾値計算方法を切り替えたり、後から追加したりすることが容易になる。

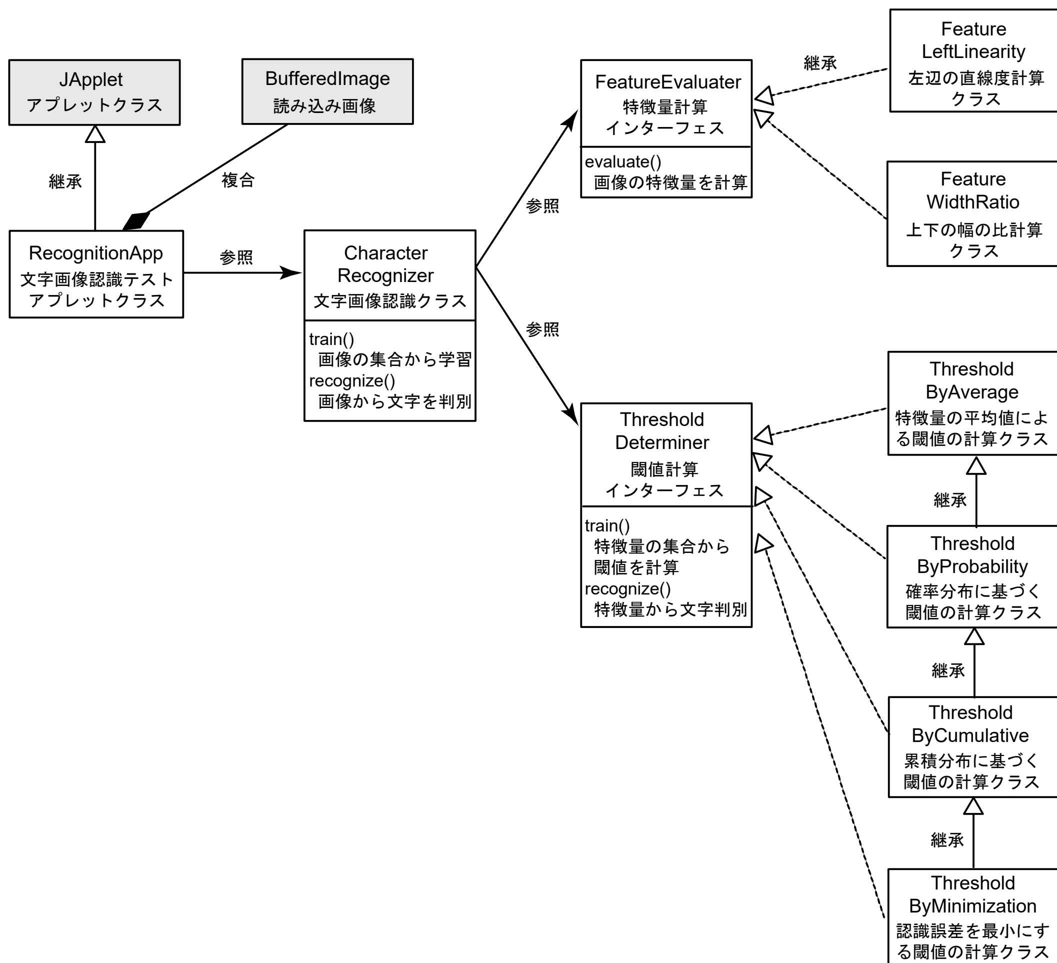

図 4　クラス構成（クラス図）

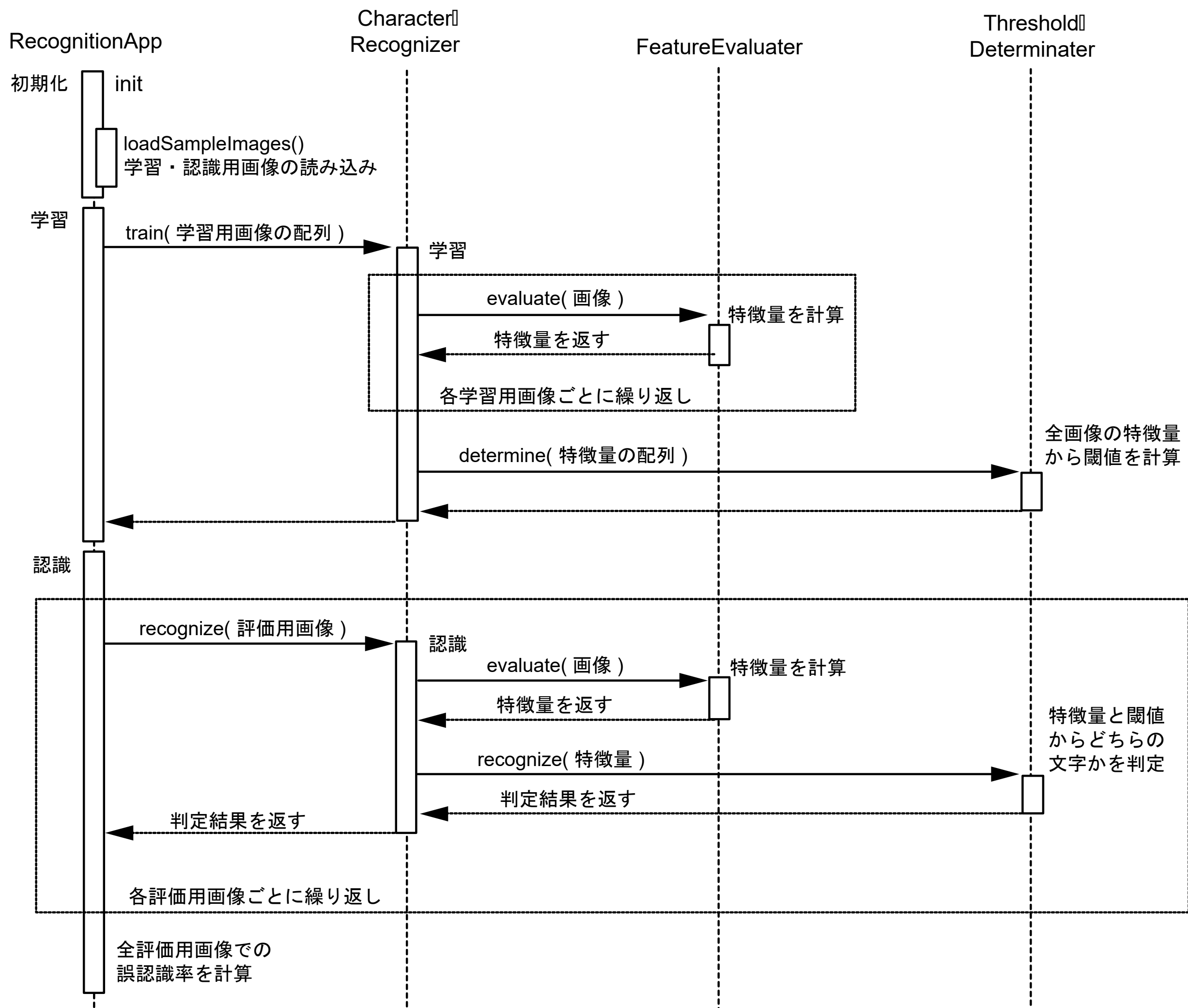

図 5　主な処理の流れ（シーケンス図）

以下に、上記の考え方にもとづいて文字画像認識クラスをさらに詳細化したサンプルコードを示す。

外部インターフェースを使用したクラス設計の例

```
//
//  文字画像認識テスト アプレット
//
public class RecognitionApp extends JApplet
{
		// 初期化処理
		public void   init()
		{
				// 文字画像判別モジュールの生成
				recognizer = new CharacterRecognizer();

				// 特徴量計算モジュール、閾値計算モジュールを設定
				recognizer.setFeatureEvaluater( new FeatureLeftLinerity() );
				recognizer.setThresholdDeterminer(new ThreshouldByAverage() );

				// 後は変更なし
		}
}
```

```
//
//   文字画像認識クラス
//
class   CharacterRecognizer
{
        // 特徴量の評価用オブジェクト
        protected FeatureEvaluater      feature_evaluater;

        // 閾値の決定用オブジェクト
        protected ThresholdDeterminer   threshold_determinater;

        // 入力画像の特徴量
        protected float   features0[];
        protected float   features1[];


        // 特徴量の評価用オブジェクトを設定
        public void   setFeatureEvaluater( FeatureEvaluater fe )
        {
                feature_evaluater = fe;
        }

        // 閾値の計算用オブジェクトを設定
        public void   setThresholdDeterminer( ThresholdDeterminer td )
        {
                threshold_determiner = td;
        }

        // 特徴量の評価用オブジェクトを取得
        public FeatureEvaluater   getFeatureEvaluater()
        {
                return   feature_evaluater;
        }

        // 閾値の計算用オブジェクトを取得
        public ThresholdDeterminer   getThresholdDeterminer()
        {
                return   threshold_determiner;
        }


        // 与えられた２グループの画像データを判別するような特徴量の閾値を計算
        public void   train( BufferedImage[] images0, BufferedImage[] images1 )
        {
                // 各入力画像の特徴量を計算
                features0 = new float[ images0.length ];
                features1 = new float[ images1.length ];
                for ( int i=0; i<images0.length; i++ )
                        features0[ i ] = feature_evaluater.evaluate( images0[ i ] );
                for ( int i=0; i<images1.length; i++ )
                        features1[ i ] = feature_evaluater.evaluate( images1[ i ] );

                // 特徴量の分布から２つのグループを識別するような閾値を決定
                threshold_determiner.determine( features0, features1 );
        }

        // 学習結果に基づいて与えられた画像を判別
        // （判別した画像の種類として  0 or 1  の値を返す）
        public int   recognize( BufferedImage image )
        {
                // 入力画像の特徴量を計算
                float   feature = feature_evaluater.evaluate( image );

                // 閾値に基づいてどちらのグループに属するかを判定
                return   threshold_determiner.recognize( feature );
        }
}
```

```
//
//  文字画像の特徴量検出クラスのインターフェース
//
interface   FeatureEvaluater
{
        // 特徴量の名前を返す
        public String   getFeatureName();

        // 文字画像から１次元の特徴量を計算する
        public float   evaluate( BufferedImage image );

        // 最後に行った特徴量計算の結果を描画する
        public void   paintImageFeature( Graphics g );
}


//
//  閾値決定クラスのインターフェース
//
interface   ThresholdDeterminer
{
        // 閾値の決定方法の名前を返す
        public String   getThresholdName();

        // 両グループの特徴量から閾値を決定
        public void   determine( float[] features0, float[] features1 );

        // 与えられた特徴量からどちらの文字かを判定
        public int   recognize( float feature );

        // 特徴空間のデータをグラフに描画（グラフオブジェクトに図形データを設定）
        public void   drawGraph( GraphViewer gv );
}


//
//  文字画像の左辺の直線度を特徴量として計算するクラス
//
class   FeatureLeftLinerity implements FeatureEvaluater
{
        // FeatureEvaluater で定義されたメソッドを実装
}


//
//  特徴量の平均値による閾値の計算クラス
//
class   ThreshouldByAverage implements ThresholdDeterminer
{
        // ThresholdDeterminer で定義されたメソッドを実装
}
```

上のサンプルプログラムの ThresholdDeterminer インターフェースは、ひとまず特徴量が１次元の場合を想定した閾値の計算インターフェースである。特徴量が２次元の場合については、次節で述べる。

上記のような設計によって、文字画像認識クラス CharacterRecognizer の大部分の処理は、特徴量計算用オブジェクト FeatureEvalueter と閾値決定用オブジェクト ThresholdDeterminer で実行されることになる。ここで、RecognitionApp.init() 内のサンプルコードのように、両インターフェースを継承したオブジェクトを設定してやることで、計算方法を切り替えることが可能になる。CharacterRecognizer は、あくまで FeatureEvalueter と ThresholdDeterminer の抽象インターフェースを通じて実際の処理を呼び出しているため、計算手法を追加するときに CharacterRecognizer クラスを修正する必要はない。

なお、FeatureEvalueter クラス・ThresholdDeterminer クラスには、の設計で挙げたメソッド以外に、名前を取得するメソッド（getFeatureName()メソッド、getThresholdName() メソッド）や、特徴量の分布や閾値をグラフとして描画するメソッド（drawGraph() メソッド）を加えている。

一方、このような抽象インターフェースを用いずに、CharacterRecognizer クラスだけで同様の処理を実現することも可能である。しかし、その場合、計算手法を追加する度に CharacterRecognizer を修正する必要があり、また、プログラムも複雑になってしまう。このような設計を行った場合の例を以下に示す。

外部インターフェースを使用しない CharacterRecognizer クラスの設計の例

```
//
//  文字画像認識クラス
//
class  CharacterRecognizer
{
	// 特徴量の評価処理にどの計算方法を使うかの番号
	protected int  feature_evaluater_no;

	// 閾値の決定処理にどの計算方法を使うかの番号
	protected int  threshold_determinater_no;

	// 文字画像から１次元の特徴量を計算する
	public float  evaluate( BufferedImage image )
	{
		// feature_evaluater_no により計算方法を分ける
		if ( feature_evaluater_no == 1 )
		{
			// 方法１
		}
		else if ( feature_evaluater_no == 2 )
		{
			// 方法２
		}
		else if ( ...
	}

	// 最後に行った特徴量計算の結果を描画する
	public void  paintImageFeature( Graphics g )
	{
		// 上と同様
	}

	// 両グループの特徴量から閾値を決定
	public void  determine( float[] features0, float[] features1 )
	{
		// threshold_determinater_no により計算方法を分ける
		if ( feature_evaluater_no == 1 )
		{
			// 方法１
		}
		else if ( feature_evaluater_no == 2 )
		{
			// 方法２
		}
		else if ( ...
	}

	// 与えられた特徴量からどちらの文字かを判定
	public int  recognize( float feature )
	{
		// 上と同様
	}

	// 特徴空間のデータをグラフに描画（グラフオブジェクトに図形データを設定）
	public void  drawGraph( GraphViewer gv )
	{
		// 上と同様
	}
}
```

このように、同一のインターフェースで、異なる働きをするオブジェクトを使い分けるような場合は、継承を用いてクラス設計を行った方が、分かりやすく、かつ、拡張性の高いプログラムとなる。

なお、今回の設計のように、ある特定の機能（ここでは特徴量の計算や閾値の計算）を実現するための方法としていくつかの方法があって、それらを切り替えて使いたいようなときに、インターフェースと継承を使って実現するような設計の考え方を、デザインパターンの用語で Strategy パターンと呼ぶ。デザインパターンとは、オブジェクト指向でプログラムを設計するときに頻繁に使われるようなオブジェクト同士の組み合わせ・役割分担のパターンに名前をつけたものである。代表的なデザインパターンとして、23 パターンが知られている。デザインパターンを知っておくと、自分でプログラムを設計するときに役に立ったり、他人に自分の設計を説明するときなどに（相手もデザインパターンを理解していれば）意思疎通がスムーズにいくというメリットがある。最近はデザインパターンに関する本も沢山出ているので、興味のある人は勉強してみると良い。

3.4. 文字画像認識クラスの 2 次元への拡張

ここまでの設計は、基本的に 1 次元の特徴量をもとに文字画像認識を行う機能だけであるので、２次元の特徴量に対応できるように拡張を行う必要がある。

特徴量の計算に関しては、現在の FeatureEvalueter インターフェースがそのまま使える。しかし、特徴量から閾値を計算するインターフェースについては、特徴量のデータが 1 次元から 2 次元に変わるので、ThresholdDeterminer インターフェースを追加・変更する必要があると考えられる。

本来はまず全体の設計を一通り行ってから実装に入ることが望ましいが、今回の演習では、ひとまずここまでの設計分を先に実装して、２次元の特徴量への拡張についてはその後で行うこととする。

3.5. インターフェース設計

ユーザインターフェースの設計についても決めておく必要がある。

本プログラムでユーザのできる操作は、図 1 に示した通りである。基本的には、次の２つの画面表示モードを使用する。

- 特徴量の分布や閾値、文字画像認識結果を表示する画面（文字認識結果の表示）
- 各サンプル画像と特徴量の計算結果を表示する画面（特徴量の表示）

ここで、画面を２分割して両方を同時に表示することも可能であるが、使い方やプログラムがややこしく

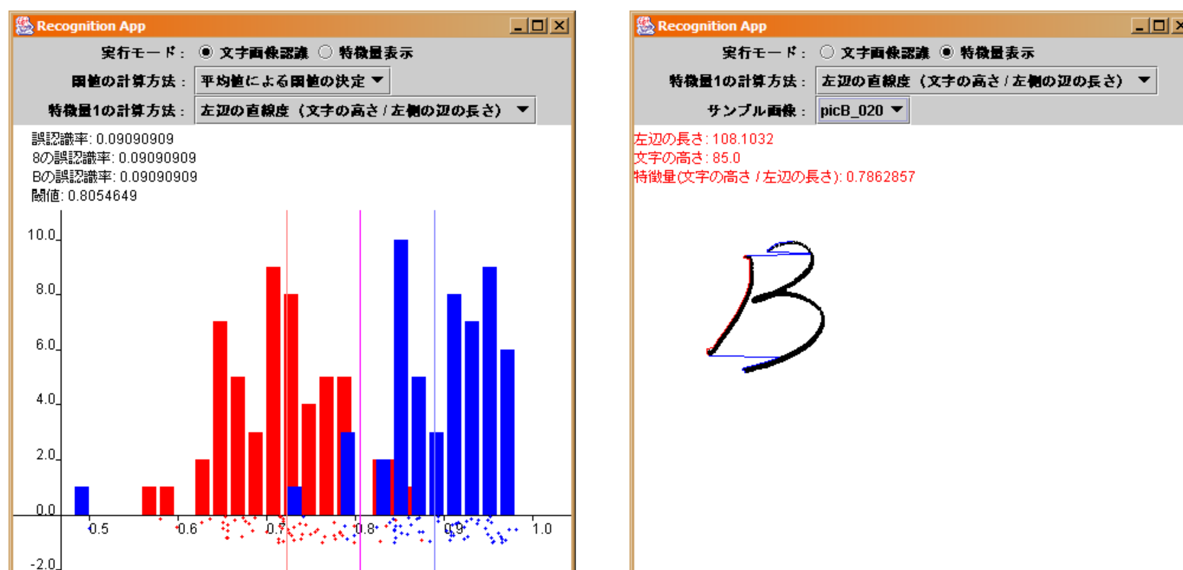

図 6　操作画面（左：文字認識結果の表示、右：特徴量の表示）

なるので、ここでは次の２つの表示モードを切り替えるようなインターフェースとする。

画面上部にはモードを切り替えるためのボタンを用意して、どちらかのボタンをクリックすることで、いつでもそのモードに移れるようにする。文字認識結果の表示中（図 6 左）は、使用する特徴量の計算方法や、閾値の計算方法をコンボボックスで切り替えられるようにする。また、特徴量の表示中（図 6 右）は、特徴量の計算方法や表示するサンプル画像を切り替えられるようにする。

3.6. グラフの描画

図 6 左の例のように、最終的な誤認識率だけでなく、特徴量の分布や閾値などをグラフとして表示して確認する機能が欲しい。そのため、ある程度一般的な用途に使えるようなグラフ描画の機能を持ったクラスを作成して、それを使用することにする。ただし、本演習でグラフ描画クラスまで各自で全部実装していると大変だと思われるので、グラフ描画クラスについては、こちらの用意したクラスを使って良い。

以下に、グラフ描画クラス（GraphViewer クラス）の主要なメソッド定義を示す。主に、描画領域の設定のためのメソッドと、描画する図形データの設定のためのメソッドが定義されている。addFigure()メソッドを呼んでグラフの図形を設定しておくと、paint()メソッドが呼ばれたときにグラフの枠を含めて全ての図形が描画されるようになっている。今回のグラフは２次元平面に限定し、図形の種類としては散布図・棒グラフ・折れ線グラフ・Ｙ軸に平行な直線に対応している。

グラフ描画クラスの主要メソッド定義

```
//
//  グラフ内での２次元座標を表すクラス
//
class  GraphPoint
{
        public float  x;
        public float  y;
}


//
//  簡易グラフ描画クラス
//
class  GraphViewer
{
        // 図形の種類
        public static final int  FIG_SCATTERED = 1; // 散布図
        public static final int  FIG_BAR = 2;       // 棒グラフ
        public static final int  FIG_LINE = 3;      // 折れ線グラフ
        public static final int  FIG_Y_LINE = 4;    // Y 軸に平行な直線（X=a の直線）

        // グラフ内の描画範囲の設定
        public void  setGraphArea( float x0, float y0, float x1, float y1, float grid_x, float grid_y );

        // 画面上の描画範囲の設定
        public void  setDrawArea( int x0, int y0, int x1, int y1 );

        // 図形データのクリア
        public void  clearFigure();

        // 図形データの追加
        public void  addFigure( int type, Color color, GraphPoint data[] );

        // グラフ全体の描画
        public void  paint( Graphics g );
}
```

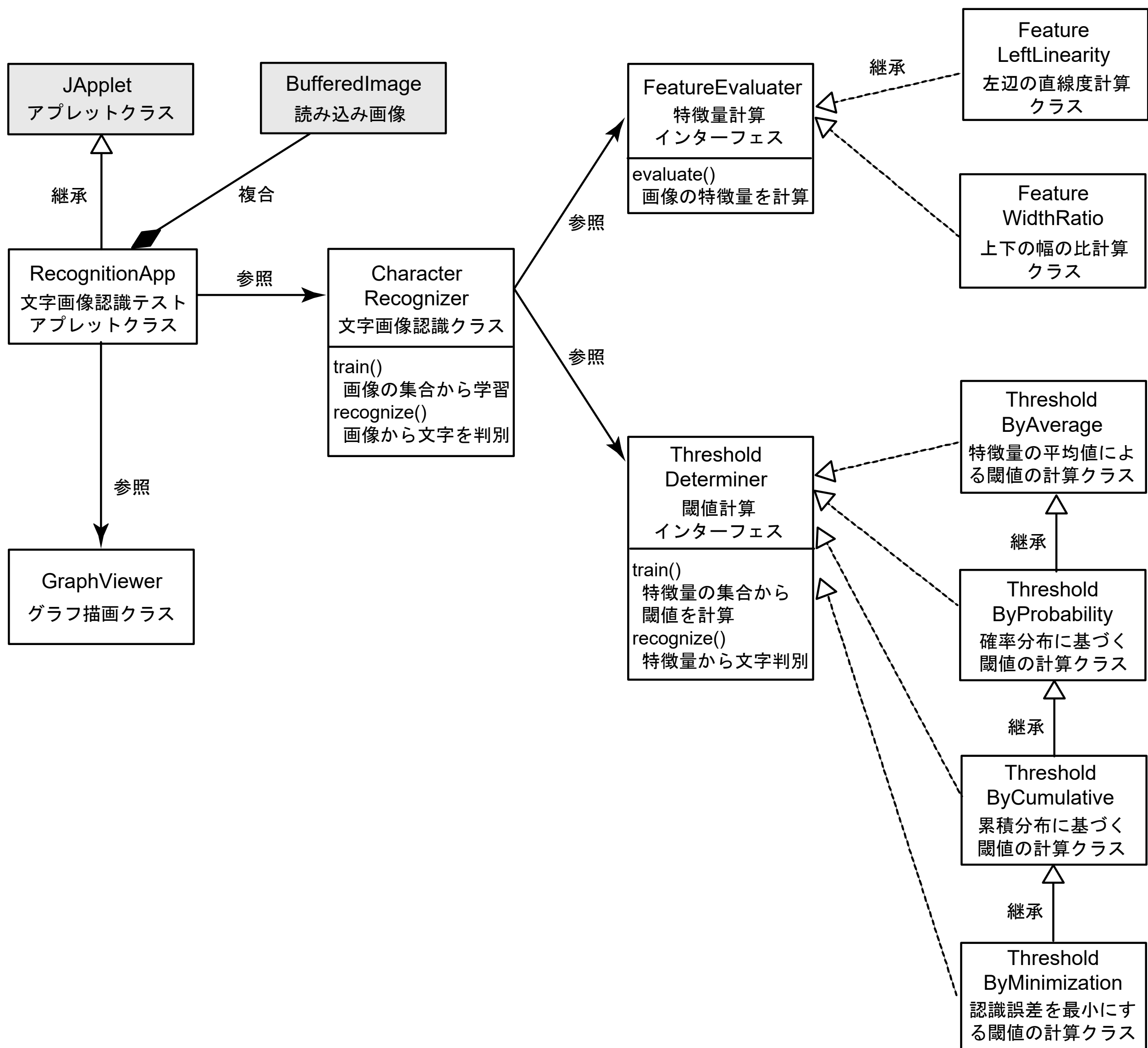

図 7　クラス構成図（クラス図）

3.7.　設計のまとめ

ここまでで、プログラム設計は大体まとまった。ここまでのクラス構造を図 7 に示す。

主なインターフェースとなるメソッドは定義できたので、あとはそれぞれのメソッドの中身を実装すれば、問題なくプログラムが動作するはずである。細部の設計については変更可能なので、必要に応じて、複雑なメソッドは内部で複数のメソッドに分けたり、補助的なメソッドを追加したりすると良い。

ある程度規模の大きいプログラムを開発する時は、十分な設計を行わず、やみくもに実装を行うと後々不整合が出てきて結局大きく修正しなければならなくなったりしてしまう。最初に、少なくとも今回の例程度には全体の設計を行ってから実装に入ることが必要である。

4. プログラムの実装

4.1. スケルトンのコンパイルテスト

本章では、これまでに行った設計に従って、実際に文字画像認識を行うプログラムを作成していく。本来は、プログラム全体を全て自分で作成するのが勉強になり望ましいが、本演習の時間では足りないため、大部分のプログラムについては、こちらで用意したサンプルコードを参考に、以下の説明に従って作業を行っていけば作成できるようになっている。ただし、特徴量に計算など、一部の重要な処理については、各自で実装するようにしている。

まずは、3 章で説明した基本設計に従って、骨組みとなる基本的なメソッド定義のみを記述したソースファイル（スケルトン）を講義のページに用意してある。

http://www.cg.ces.kyutech.ac.jp/lecture/project/CharacterRecognizer_Step1.zip

以下では、これをベースとして実装を行っていく。最初に用意されているスケルトンは、メソッド定義のみで具体的なメソッドの中身の処理は記述されておらず、単に起動して操作できるだけのプログラムとなっている。まずは、ダウンロードした圧縮ファイルを解凍して、ソースファイルを準備すること。

4.2. コンパイルと実行方法

サンプルプログラムのソースファイルが準備できたら、コンパイルのテスト、及び、実行確認を行う。

本演習では、プログラムの開発のために、統合環境 Eclipse を使用する。統合環境とは、ソースファイルの記述、プログラムの実行、デバッグなどの作業を統合して行えるようなソフトウェアである。特に、ステップ実行や変数ウォッチなどのデバッグのための機能を利用することができるため、統合環境を利用することで、プログラムの開発効率は非常に上がる。本演習では、Java の統合環境として広く利用されている、Eclipse を使用する。（今までの講義・演習でやったような、コマンドラインからコンパイル・実行する方法の方が良ければ、今までと同じやり方でやっても構わない。）

Eclipse の使用方法については、下記のページにまとめている（講義のページからもリンクしている）。

http://www.cg.ces.kyutech.ac.jp/lecture/project/eclipse/index.html

上記のページの説明を参考に、下記の手順で、プログラムの実行確認を行う。

1．Eclipse を起動する。（上記のページの 2 節の説明を参照）
2．ダウンロードしたソースファイル一式をプロジェクトとして追加する。（3.2 節の説明を参照）
3．プロジェクトをアプリケーション（またはアプレット）として実行する。（5 節の説明を参照）

RecognitionApp クラスは、JApplet クラスを継承しているので、アプレットとして起動できる。同時に、RecognitionApp クラスは main() 関数を持ち、アプリケーションとしても起動できるように作成されている。基本的には、アプレットとアプリケーションのどちらで起動しても構わない。ただし、アプレットとして起動すると 3.2 節で説明した拡張プラグインがロードさせず、BMP 画像の読み込みができないので、アプリケーションとして起動するようにする（アプレットは、ネットワーク経由でダウンロードして実行するような用途が想定されているので、セキュリティの観点から不用意に外部パッケージを読み込まない仕様になっている）。

4.3. 初期化処理の実装（RecognitionApp クラス）

まずは、システムの初期化処理を行う。この処理は、上でダウンロードしたソースファイルに既に記述されているので、自分で手を加える必要はない。ただし、後で拡張を行うときにはここを修正する必要があるので、何が行われているかをきちんと理解しておくこと。

init() メソッドでは、最初に画像認識クラス（CharacterRecognizer クラス）のオブジェクトを初期化す

る。また、特徴量計算オブジェクト（FeatureEvaluater）と、閾値計算オブジェクト（ThresholdDeterminer）をそれぞれ生成して配列で管理する（現在はまだそれぞれ１つずつだが、後で計算方法を拡張するときにはオブジェクトの数が増える）。次節で作成するユーザインターフェースでは、メニューから計算方法を選択することで、指定されたオブジェクトを利用できるようにする。

初期化処理の実装

```
public class RecognitionApp extends JApplet
{
	// 文字画像認識モジュール
	protected CharacterRecognizer  recognizer;

	// 利用可能な特徴量計算モジュール
	protected FeatureEvaluater   features[];

	// 利用可能な閾値決定モジュール（１次元の特徴量）
	protected ThresholdDeterminer  thresholds[];

	// グラフ描画モジュール
	protected GraphViewer  graph_viewr;

	// 実行モード
	protected int  mode;
	protected final int  RECOGNITION_MODE = 1;
	protected final int  FEATURE_MODE = 2;

	// 初期化処理
	public void  init()
	{
		// 利用可能な特徴量計算モジュールを初期化
		features = new FeatureEvaluater[ 1 ];
		features[ 0 ] = new FeatureLeftLinerity();

		// 利用可能な閾値決定モジュール（１次元の特徴量）を初期化
		thresholds = new ThresholdDeterminer[ 1 ];
		thresholds[ 0 ] = new ThresholdByAverage();

		// 文字画像判別モジュールの生成
		recognizer = new CharacterRecognizer();
		recognizer.setFeatureEvaluater( features[ 0 ] );
		recognizer.setThresholdDeterminer( thresholds[ 0 ] );

		// グラフ描画モジュールの生成
		graph_viewr = new GraphViewer();

		// 開始時の実行モードの設定
		mode = FEATURE_MODE;

		....
	}
}
```

4.4. ユーザインターフェースの実装（RecognitionApp クラス）

この処理についても既に最初のソースコードに記述してあるので、とくに手を加える必要はない。ここでは、3.5 節で説明したユーザインターフェースを実現するためのボタンやコンボボックスなどを配置して、それぞれが動作するようにしている。また、メイン画面の描画を行うメソッドも実装する。ユーザインターフェースについては、3.5 節のインターフェース全部を実現するとかなりプログラムが長くなるので、ここでは詳しい説明やソースコードは省略する。行っている内容自体はそれほど難しいことではない。

プログラムを実行したら、適当に操作してメニュー等がきちんと動作していることを確認する。

```
ユーザインターフェースの実装

public class RecognitionApp extends JApplet
{
	....

	// UI 用コンポーネント
	protected JPanel	ui_panel;
	protected JLabel	mode_button_label, threshold_label, feature_label[], image_label;
	protected JRadioButton	mode_r_bottun, mode_f_bottun;
	protected JComboBox	threshold_list, feature_list[], image_list;
	protected MainScreen	screen;

	//コンポーネントの初期化・更新処理（長いので省略、講義のページのリストを参照）
	....

	// メイン画面描画処理（長いので省略、講義のページのリストを参照）
	....
}
```

4.5. サンプル画像の読み込み処理の実装（RecognitionApp クラス）

サンプル画像を読み込む処理が空のままになっているので、まずはこれを実装する。ここからは、各自でプログラムを追加していく必要がある。講義のページに、下記のソースファイルを置いているので、必要であればコピーして使用して良い。

3.2 節で説明したように、画像の読み込みには Image I/O ライブラリを使用する。まず、Image I/O ライブラリを利用して 1 枚の画像を読み込む getBufferedImage() メソッドを作成する。また、実際には連番画像を読み込む必要があるが、画像 1 枚ごとに getBufferedImage() の呼び出しをプログラムに直接記述していると大変なので、自動的に連番の画像を読み込んで配列に格納する loadBufferedImages() メソッドを作成することにする。loadSampleImages() メソッドでは、２種類のサンプル画像の集合を読み込んで、読み込んだ画像を同時にインデックスに記録している。インデックスは、メニューから画像を選択したときに、画像名から画像オブジェクトを取得するために使用する。

```
list1.txt　サンプル画像の読み込み処理の実装

public class RecognitionApp extends JApplet
{
	....

	// サンプル画像
	protected BufferedImage	sample_images0[];
	protected BufferedImage	sample_images1[];

	// 全サンプル画像のインデックス（画像名→画像オブジェクト の参照）
	protected TreeMap	image_index;

	....

	//
	//	サンプル画像の読み込みのための内部メソッド
	//

	// サンプル画像の読み込み
	public void	loadSampleImages()
	{
		// 8 のサンプル画像を読み込み
		//（Samples8Bgif/pic8_001.gif ～ pic8_055.gif の 55 枚）
		sample_images0 = loadBufferedImages( "Samples8Bgif/pic8_", 1, 55, 3, ".gif" );

		// B のサンプル画像を読み込み
		//（Samples8Bgif/picB_001.gif ～ picB_055.gif の 55 枚）
```

```
		sample_images1 = loadBufferedImages( "Samples8Bgif/picB_", 1, 55, 3, ".gif" );

		// 全ての画像をインデックスに記録する
		image_index = new TreeMap();
		String  name;
		for ( int i=0; i<sample_images0.length; i++ )
		{
			name = "" + (i + 1);
			while ( name.length() < 3 )
				name = "0" + name;
			name = "pic8_" + name;
			image_index.put( name, sample_images0[ i ] );
		}
		for ( int i=0; i<sample_images1.length; i++ )
		{
			name = "" + (i + 1);
			while ( name.length() < 3 )
				name = "0" + name;
			name = "picB_" + name;
			image_index.put( name, sample_images1[ i ] );
		}
	}

	// 連番画像の配列への読み込み
	protected BufferedImage[]  loadBufferedImages(
		String prefix, int count_start, int count_end, int count_width, String suffix )
	{
		// 読み込んだ画像を格納する配列のサイズを初期化
		BufferedImage[]  images = new BufferedImage[ count_end - count_start + 1 ];

		// 連番画像を順に読み込み
		for ( int i=count_start; i<=count_end; i++ )
		{
			// ファイル名を作成
			String  filename = "" + i;
			while ( filename.length() < count_width )
				filename = "0" + filename;
			filename = prefix + filename + suffix;

			// 画像を読み込み
			images[ i - count_start ] = getBufferedImage( filename );
		}

		return  images;
	}

	// Image I/O を使った画像の読み込み（BMP に対応）
	protected BufferedImage getBufferedImage( String filename )
	{
		try
		{
			java.io.File  file = new java.io.File( filename );
			BufferedImage image = javax.imageio.ImageIO.read( file );
			return  image;
		}
		catch ( Exception e )
		{
			return  null;
		}
	}
}
```

4.6. 文字認識のテスト処理の実装（RecognitionApp クラス）

読み込んだサンプル画像全てを使って文字認識のテストを行うメソッドを追加する。テスト結果として、

誤認識率を計算して変数に記録しておく。誤認識率は、画像認識モード時にメイン画面に表示される。
　以下の処理を追加して、プログラムを実行して確認してみよ。現在は、特徴量の計算や、文字画像の判定を行う処理が空のままなので、正しく認識されないはずである。ここまでの動作が確認できたら、次節以降で、実際の認識に必要な処理を作成していく。

```
list2.txt　文字認識のテスト処理の実装

public class RecognitionApp extends JApplet
{
	....

	// 学習・文字判別テストの結果
	protected float	error, error0, error1; // 誤認識率

	....

	// サンプル画像を使った文字画像認識のテスト
	public void	recognitionTest()
	{
		// 全てのサンプル画像を使って学習
		recognizer.train( sample_images0, sample_images1 );

		// 全てのサンプル画像を使って誤認識率を計算
		int	error_count[] = { 0, 0 };
		int	char_no;
		error_count[ 0 ] = 0;
		error_count[ 1 ] = 0;
		for ( int i=0; i<sample_images0.length; i++ )
		{
			char_no = recognizer.recognizeCharacter( sample_images0[ i ] );
			if ( char_no != 0 )
				error_count[ 0 ] ++;
		}
		for ( int i=0; i<sample_images1.length; i++ )
		{
			char_no = recognizer.recognizeCharacter( sample_images1[ i ] );
			if ( char_no != 1 )
				error_count[ 1 ] ++;
		}
		error0 = (float) error_count[ 0 ] / sample_images0.length;
		error1 = (float) error_count[ 1 ] / sample_images1.length;
		error = (float) ( error_count[ 0 ] + error_count[ 1 ] ) /
			(float) ( sample_images0.length + sample_images1.length );

		// 特徴空間・閾値などをグラフに設定
		recognizer.drawGraph( graph_viewr );

		// 画面の再描画
		repaint();
	}
}
```

4.7.　特徴量の計算処理の実装（FeatureLeftLinerity クラス）

　ここで、特徴量の計算処理を実装する。まずは、プログラムを起動して、ウィンドウ上部のボタンをクリックして「認識量表示モード」に移り、サンプル画像が表示されることを確認する。ここで、各サンプル画像と同時に、その特徴量の計算結果が画面に表示されるようになっている（図 6 右)。ただし、最初の状態では、まだ特徴量を計算する処理が実装されていないので、正しい特徴量が表示されないはずである。そこで、特徴量を計算する FeatureLeftLinerity クラスのメソッドを実装して、正しく動作するようにする。
　具体的な特徴量の計算アルゴリズムの実装方法については、5 章の説明を参照すること。

4.8.　閾値の計算処理の実装（ThresholdByAverage クラス）

特徴量の計算処理が完成したら、次は、閾値の計算処理の処理を実装する。学習した４通りの計算方法のうち、まずは、6.1 節を参考に、最も簡単な、平均値に基づいて閾値を計算するクラス（ThresholdByAverage クラス）を実装する。

ここまでできれば、全体の認識テストが実行できるはずである。プログラムの「文字画像認識モード」選択して、結果を確認してみよ。図 6 左のように、特徴量の分布が正しく表示され、閾値を表す境界線が適切な位置に描かれて、誤認識率がきちんと表示されれば成功である。もし、特徴量の分布がおかしい場合は、特徴量の計算処理が間違っていると考えられるので、前節の実装を見直す。また、閾値の位置や認識率がおかしい場合は、閾値の計算が間違っていると考えられるので、本節の閾値の計算処理を見直すこと。最終的な認識結果は、特徴量の計算アルゴリズムによって変わるが、5 章の説明通りに実装したら、どんなに悪くとも、誤認識率が３割以下くらいの結果は得られるはずである。

4.9.　閾値の計算処理の改良（ThresholdByProbability, ThresholdByCumulative, ThresholdByMnimization）

ThresholdByAverage クラスを使った１次元の特徴量による画像認識が正しく実行できるようになったら、次は、閾値の計算方法を改良するために、6.2 節～6.5 節の説明に従って残りの３つの計算方法も追加する。

これらのクラスについては、一部のメソッドのみが記述されたサンプルプログラムを用意しているので、講義のページからダウンロードして使用する。具体的な閾値を計算する処理については、6 章の内容を参考に各自で追加する。

http://www.cg.ces.kyutech.ac.jp/lecture/project/CharacterRecognizer_Step2.zip

新しく追加した閾値計算クラスを有効にするためには、最初に初期化した閾値計算オブジェクトの配列に単純に追加すれば良い。以下のように RecognitionApp を修正する。ついでに、起動時に画像認識結果を表示するように実行モードも変更しておく（RECOGNITION_MODE）。

```
list4.txt　閾値計算モジュールの追加

//
//　文字画像認識テスト アプレット
//
public class RecognitionApp extends JApplet
{
		// 初期化処理
		public void　init()
		{
				....

				// 利用可能な閾値決定モジュール（１次元の特徴量）を初期化
				thresholds = new ThresholdDeterminer[ 4 ];
				thresholds[ 0 ] = new ThresholdByAverage();
				thresholds[ 1 ] = new ThresholdByProbability();
				thresholds[ 2 ] = new ThresholdByCumulative();
				thresholds[ 3 ] = new ThresholdByMinimization();

				// 開始時の実行モードの設定
				mode = RECOGNITION_MODE;
				....
		}
}
```

３つの方法を追加したら、それぞれの方法を試して、どの方法が最も低い誤認識率となる確認してみよ。ここまでで、１次元の特徴量を使って画像認識を行う機能については完成である。２次元の特徴量を使ったプログラムの設計と実装については、7 章以降で説明する。

5. 特徴量（左側の辺の直線度）の計算アルゴリズムの実装

最初の講義で学習したように、「８」と「Ｂ」を識別するための特徴量のひとつとして、左側の辺の直線度（辺の長さ／辺の高さ）を使用する。

ここまでのクラス設計に従い、この特徴量を計算する FeatureLeftLinerity.evaluate() メソッドを実装する。以下に、その基本的な方針を示す。

5.1. 画像のピクセルの色の取得方法

BufferedImage クラスには、各ピクセルの色を取得するためのメソッド ing getRGB(int x, int y) が定義されている。getRGB() メソッドは、INT_ARGB 形式でそのピクセルの色を返す。これは、ARGB のそれぞれは各 8 ビットを使用し、1 つのピクセルの色全体を 24 ビット（8 バイト）で表す形式である。A は α チャンネルを意味し、そのピクセルの不透明度である。普通のピクセルは、α チャンネルは最大値（255=0xff）になる。従って、黒のピクセルの色は 0xff000000、白のピクセルの色は 0xffffffff となる。

```
int   color = image.getRGB( x, y );
if ( color == 0xffffffff )
{
        // ピクセルの色が白の場合の処理
}
if ( color == 0xff000000 )
{
        // ピクセルの色が黒の場合の処理
}
```

5.2. 左側の辺の検出と描画

左側の辺の長さや高さを計算するためには、まず画像ピクセルデータを走査して左側の辺を取り出す必要がある。

以下に、左側の辺を抽出するプログラムの例を示す。ここで、左側の辺の情報をどのような形で記録するかが重要になる。以下の例では、各行ごとに最も左にあるピクセルの X 座標を、配列 left_x[] 記録するという形式にしている。ただし、ピクセルが 1 つもない行には、X 座標として-1 を入れることにする。

また、きちんと左側の辺が見つかったかどうか確認するため、左側の辺を赤い線で描画する処理も追加する（paintImageFeature() メソッド）。

```
list3.txt   文字画像の左側の辺を抽出する

//
//   文字画像の左辺の直線度を特徴量として計算するクラス
//
class   FeatureLeftLinerity implements FeatureEvaluater
{
        // 左辺の長さと文字の高さ
        protected float   left_length;
        protected float   char_height;

        // 左側の辺の情報
        //（画像の各行の最も左側にあるドットのＸ座標、行にひとつもドットがなければ -1）
        protected int   left_x[];

        // 最後に特徴量計算を行った画像（描画用）
        protected BufferedImage   last_image;
```

```
// 特徴量の名前を返す
public String   getFeatureName()
{
        return  "左辺の直線度（文字の高さ / 左側の辺の長さ）";
}

// 文字画像から１次元の特徴量を計算する
public float   evaluate( BufferedImage image )
{
        int   height = image.getHeight();
        int   width = image.getWidth();

        // 左側の辺を取り出す（各行の最も左側のドットのＸ座標を調べる）
        left_x = new int[ height ];
        for ( int y=0; y<height; y++ )
        {
                // 最初は黒ピクセルが１つもないものとして-1 で初期化
                left_x[ y ] = -1;

                // 左側から順番にピクセルを走査
                for ( int x=0; x<width; x++ )
                {
                        // ピクセルの色を取得
                        int   color = image.getRGB( x, y );

                        // ピクセルの色が黒であれば最も左側のドットとして
                        // X 座標を記録
                        if ( color == 0xff000000 )
                        {
                                left_x[ y ] = x;
                                break;
                        }
                }
        }

        // 文字の高さを計算
        char_height = 1.0f; // 要実装

        // 文字の左側の辺の長さを計算
        left_length = 1.0f; // 要実装

        // 文字の高さ / 左側の辺の長さ の比を計算
        float   left_linearity;
        left_linearity = char_height / left_length;

        // 画像を記録（描画用）
        last_image = image;

        return   left_linearity;
}

// 最後に行った特徴量計算の結果を描画する
public void   paintImageFeature( Graphics g )
{
        if ( last_image == null )
                return;

        int   ox = 0, oy = 0;
        g.drawImage( last_image, ox, oy, null );

        int   x0, y0, x1, y1;
        for ( int y=0; y<left_x.length-1; y++ )
        {
                y0 = y;
                y1 = y+1;
                x0 = left_x[ y0 ];
                x1 = left_x[ y1 ];
```

```
                        if ( ( x0 != -1 ) && ( x1 != -1 ) )
                        {
                                // 左側の辺のピクセルを描画
                                g.setColor( Color.RED );
                                g.drawLine( ox + x0, oy + y0, ox + x1, oy + y1 );
                        }
                }

                String  message;
                g.setColor( Color.RED );
                message = "左辺の長さ: " + left_length;
                g.drawString( message, ox, oy + 16 );
                message = "文字の高さ: " + char_height;
                g.drawString( message, ox, oy + 32 );
                message = "特徴量(文字の高さ / 左辺の長さ): " + char_height / left_length;
                g.drawString( message, ox, oy + 48 );
        }
}
```

5.3. 左側の辺の長さと辺の高さの計算

左側の辺が見つかったら、各行の値を調べて、文字画像の最も上のピクセルがある行（左側の辺の X 座標が -1 でない最初の行）の Y 座標と、最も下のピクセルがある行（左側の辺の X 座標が -1 でない最初の行）の Y 座標の差を取れば、辺の高さが求まる（図 8 の例では、緑色の線が文字の上端・下端を表す）。また、辺を上から下にたどりながら、1 行分ずつ辺の長さを加算していくことで、辺の長さが計算できる。平方根は、java.lang.Math.sqrt( ) メソッドを使えば計算できる。

辺の長さと高さが求まれば、特徴量が計算できるので、文字画像認識処理のテストが行える。ここまでできたら、ひとまずの 6.1 節の処理も実装して、実際に文字画像認識のテストを行い、現在の特徴量計算でどの程度正確に認識できるかを確認してみよ。

5.4. 左側の辺から適切な範囲のみを取り出す処理の追加

上記の処理で、一応左側の辺を取り出し、その直線度を計算することができるようになった。しかし、実際にいくつか画像を確認してみると、左側の辺がかけているところで右側の辺が抽出されてしまい、結果的に辺の長さが長くなってしまうエラーが出ている画像が出ることが分かる（図 8 の例を参照）。

このような問題を解決するための方法としては、最初の処理で見つかった左側の辺を全て使用するのではなく、何らかの方法で左辺に相当する範囲のみを取り出すことが考えられる。

このための方法としては、いくつかのやり方が考えられる。1 つの方法としては、上記の例のように大きなギャップが出ているところを見つけて、ギャップのあるところで複数の区間に分割し、最も長い区間のみを左側の辺として使用する、といったやり方がある。（図 8 はこの方法で左辺を抽出した例。青色＋赤色の部分がもとの左辺で、赤色の部分が抽出された左辺）。あるいは、ギャップがなくなるように、左側の辺の X 座標をなめらかに平滑化してやる方法も考えられる。

このような方法のどれかを試してみて、より正確に特徴量が検出できるかどうかを確認せよ。

また、左側の辺から一部のみを取り出して辺の長さを計算する場合は、図 8 のように、取り出した範囲のみ色を変えて描画するような機能も付け加えると、動作確認ができて良い。

なお、本処理については、実現できなくとも、認識精度が落ちるだけで、一通りの処理は可能なので、難しいようであれば、後回しにして、次の作業に移っても構わない。

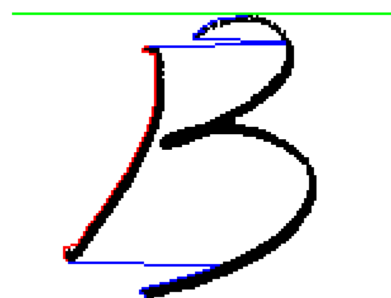

図 8　左辺の高さと長さの計算

6. 1 次元の閾値の計算アルゴリズムの実装

6.1. 平均値に基づき閾値を計算（ThresholdByAverage クラス）

平均値に基づいて閾値を計算する ThresholdByAverage クラスを作成する。用意されているサンプルプログラムでは、棒グラフ描画のためのヒストグラムの計算、棒グラフと平均値・閾値の描画処理、閾値にもとづいた判定処理などはすでに実装されており、閾値を計算する処理（determine() メソッド）のみが空になっている。そこで、この処理を実装する。

具体的には、グループ 0 の全特徴量の平均値（average0）、グループ 1 の全特徴量の平均値（average1）を計算し、両者の中央値を閾値（threshold）とする。ここで重要になるのが、閾値と両グループの平均値の関係である。以下の二通りがあり得る。

- グループ 0 の特徴量 ＜ 閾値 ＜ グループ 1 の特徴量
- グループ 1 の特徴量 ＜ 閾値 ＜ グループ 0 の特徴量

そのため、閾値と両グループの関係が上記のどちらに当たるという関係を表すフラグ is_fist_smaller を用意して、前者の場合は is_first_smaller＝true、後者の場合は false としておく。そして、determine() メソッドの判定処理では、このフラグに応じて判定を行うことにする。従って、閾値（threshold）の計算時に、同時に閾値と両グループの関係を表すフラグ（is_first_smaller）を計算する必要がある。

平均値に基づく閾値の計算クラス

```
//
//  平均値に基づく閾値の計算クラス
//
class   ThresholdByAverage implements ThresholdDeterminer
{
	// 閾値と符号（グループ 0 の方が特徴量が閾値よりも小さければ真）
	protected float   threshold;
	protected boolean   is_first_smaller;

	// 特徴量データ（グラフ描画用）
	protected float   features0[];
	protected float   features1[];

	// 各グループの特徴量の平均値（グラフ描画用）
	protected float   average0;
	protected float   average1;


	// 度数分布のデフォルトの区間幅
	float   default_histogram_delta = 0.02f;


	// 両グループの特徴量から閾値を決定する
	public void   determine( float[] features0, float[] features1 )
	{
		// 各グループの平均値を計算
		average0 = 0.0f; // 要実装
		average1 = 0.0f; // 要実装

		// ２つの平均値の中央値を計算
		threshold = 0.0f; // 要実装

		// 符号を計算
		is_first_smaller = true; // 要実装

		// 特徴量データを記録（グラフ描画用）
		this.features0 = features0;
		this.features1 = features1;
```

```
        }

        // 閾値をもとに特徴量から文字を判定する
        public int   recognize( float feature )
        {
                // グループ 0 の特徴量 < 閾値 < グループ 1 の特徴量
                if ( is_first_smaller )
                {
                        if ( feature < threshold )
                                return   0;
                        else
                                return   1;
                }
                // グループ 1 の特徴量 < 閾値 < グループ 0 の特徴量
                else
                {
                        if ( feature < threshold )
                                return   1;
                        else
                                return   0;
                }
        }
}
```

6.2. ヒストグラムの計算

次に説明する別の閾値計算方法では、特徴量の度数分布（ヒストグラム）が必要になる。また、ヒストグラムは、値の分布状況を分かりやすくグラフ表示するためにも有効である。そこで、ヒストグラムを計算するための処理を ThresholdByAverage クラスに実装する。ヒストグラムの計算処理は、サンプルプログラムに含まれているので、それを利用して良い。ただし、以降の閾値計算を実装するためには、ここで説明するヒストグラムのデータ表現を理解しておく必要がある。

ヒストグラムを作成するためには、空間を固定幅の区間に区切り、各区間でのデータの出現回数を計算する。ヒストグラムを作成するメソッドの例は、次のリストの makeHistogram() のようになる。ヒストグラムの区間の情報（範囲の最小値・最大値、各区間の幅）と特徴量の配列を入力として受け取り、各区間のデータの出現回数をカウントして、int 型の配列として返す。

ただし、実際には、特徴量の分布はデータによって異なるため、ヒストグラムの区間の情報をあらかじめ定義しておくことは難しい。そこで、データの範囲から自動的に計算するメソッドを用意することにする。ここでは、区間の数を指定すると区間の幅を自動的に決定する方法（makeHistogramsBySize() メソッド）と、区間の幅を指定すると区間の数を自動的に決定して区間の幅を再調整する方法の 2 つの方法（makeHistogramsByWidth() メソッド）を用意している。なお、サンプルプログラムでは、区間の数を指定する方法を呼び出すようになっている（変更しても構わない）。そのため、特徴量の分布に対して極端に小さな最小値や極端に大きな最大値があると、適切なヒストグラムが計算されずに、後の認識で問題が生じてしまうため、注意する必要がある。

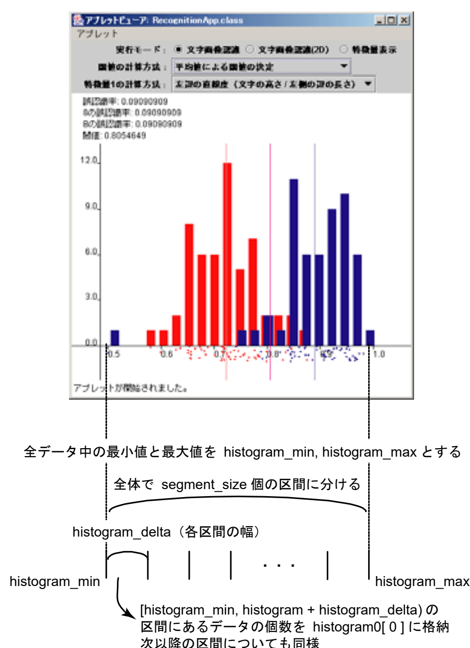

図 9　ヒストグラムの表示と計算処理（makeHistogramBySize() メソッド）

計算したヒストグラムを記録するために、ThresholdByAverage クラスの変数として、ヒストグラムの情報（範囲の最小値 histogram_min_f、範囲の最大値 histogram_max_f、各区間の幅 histogram_delta）と、計算された 2 つのグループの特

徴量のヒストグラム（histogram0, histogram1）を追加している。また、drawHistogram() メソッド（リストは省略）には、これらの変数の値をもとに、ヒストグラムを棒グラフとして描画する処理を追加している。

```
ヒストグラムの計算処理

class   ThresholdByAverage implements ThresholdDeterminer
{
        // 度数分布（ヒストグラム）
        protected float   histogram_min_f, histogram_max_f, histogram_delta_f;
        protected int[]   histogram0;
        protected int[]   histogram1;

        // 指定された範囲・区間で特徴量の度数分布（ヒストグラム）を計算
        //（min_f, maxf はヒストグラムを作成する範囲、delta_f は各区間ごとの幅）
        protected int[]   makeHistogram( float[] features, float min_f, float max_f, float delta_f )
        {
                // ヒストグラムの区間数を計算して配列を初期化
                int   histogram_size = ( max_f - min_f ) / delta_f;
                int[]   histogram = new int[ histogram_size ];

                // ヒストグラムの各区間におけるデータの出現回数をカウント
                for ( int i=0; i<features.length; i++ )
                {
                        int   seg_no = ( features[ i ] - min_f ) / delta_f );
                        histogram[ seg_no ] ++;
                }

                return   histogram;
        }

        // 特徴量の度数分布（ヒストグラム）を計算
        //（全体をいくつの区間に分けるかという区間数を指定、区間幅は自動決定）
        protected void   makeHistograms( int segment_size )
        {
                // ２つのグループの特徴量の範囲（最小値と最大値）を調べる
                histogram_min_f = ...;   // 省略
                histogram_max_f = ...;   // 省略

                // 指定された区間数に応じて、区間幅を計算
                histogram_delta_f = ...;   // 省略

                // 両端の区間の分布が０になるように、左右に区間を１つ追加する
                histogram_min_f -= delta_f;
                histogram_max_f += delta_f;

                // ２つのグループのヒストグラムを作成
                histogram0 = makeHistogram( features0,
                                histogram_min_f, histogram_max_f, delta_f );
                histogram1 = makeHistogram( features1,
                                histogram_min_f, histogram_max_f, delta_f );
        }

        // 特徴量の度数分布（ヒストグラム）を計算
        //（各区間の区間幅を指定、区間数は自動決定）
        protected void   makeHistograms( float delta_f )
        {
                // ２つのグループの特徴量の範囲（最小値と最大値）を調べる
                histogram_min_f = ...;   // 省略
                histogram_max_f = ...;   // 省略

                // 区間幅に応じて範囲を調整（区間の両端値が区間幅の整数倍になるよう調整）
                histogram_delta_f = delta_f;   // 省略

                // ２つのグループのヒストグラムを作成
                histogram0 = makeHistogram( features0,
                                histogram_min_f, histogram_max_f, delta_f );
                histogram1 = makeHistogram( features1,
```

```
                                        histogram_min_f, histogram_max_f, delta_f );
        }
}
```

6.3. 出現確率が等しくなる閾値を計算（ThresholdByProbability クラス）

２つ目の閾値計算の方法として、出現確率が等しくなるような閾値を計算する。まずは、ヒストグラムのそれぞれの区間ごとのデータの出現率を計算する。出現率＝その区間のデータ数／全データ数 なので、これは簡単に計算できる（makeProblility() メソッドを参照)。また、計算結果の出現確率の配列を記録する変数（probability0、problility1）と、この配列を出現確率の折れ線グラフとして描画するための drawProbability() メソッド（リストは省略）を ThresholdByProbability クラスに追加する。ここまでの処理はサンプルプログラムに実装済みなので、計算された出現確率をもとに、出現確率が等しくなるような閾値を計算する処理を各自で追加する。

基本的な考え方としては、ヒストグラムの隣り合う各区間の同士の２つのグループの出現確率を見ていき、左右の区間で出現確率の高い方のグループが異なる場合は、必ず、その２つの区間の間に出現確率が等しくなる値（２つの出現確率の折れ線が交差する点）が存在することに注目する（図 10 右)。ここで、隣接する区間での出現確率の変化が線形であると仮定すると、２つの線分の交点から、閾値を求めることができる。

具体的には、区間を順番に見ていき、seg_no 番目の区間での、特徴量（図 10 右での横軸）を feature_left、グループ 0 の画像の出現率（図 10 右での縦軸）を left_probability0、グループ 1 の画像の出現率を left_probability1 とする。同じく、隣の seg_no+1 番目の区間についても、特徴量を feature_right、グループ 0 の画像の出現率を right_probability0、グループ 1 の画像の出現率を right_probability1 とする。後は、線分（feature_left, left_probability0）－（feature_right, right_probability0）と、線分（feature_left, left_probability1）－(feature_right, right_probability1）の交点（x, y）を計算すれば、その交点の横軸の値 x が、求める閾値（threshold）となる。

ただし、注意しなければならないのは、確率分布によっては、２つのグラフが２箇所以外で交わる可能性があることである。このような場合（交点が複数個見つかった場合）は、何らかの方法で、どちらかの交点を閾値として使用する。例えば、6.1 節の方法で計算した閾値に最も近い値を用いる、といった方法が考えられる。ThresholdByProbability クラスは ThresholdByAverage クラスを継承しているので、ThresholdByProbability クラスの determine()メソッドから、基底クラス ThresholdByAverage の determine()メソッドを呼び出すことで、平均値にもとづく閾値を計算することができる。下記のサンプルプログラムでは、determine()メソッドの１行目でこの処理を行っている。この結果、平均値にもとづいて計算した閾値があらかじめ threshold に代入されるので、最終的な閾値を決めるときにこの初期値を参考にすれば良い。

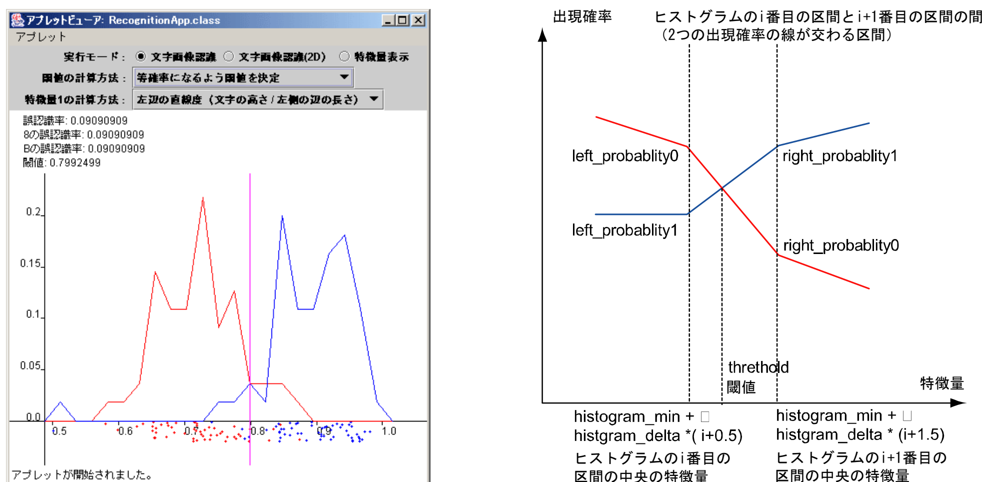

図 10　出現確率が等しくなるような閾値（左）と出現確率の折れ線同士の交点（閾値）の計算（右）

```
//
//  確率分布に基づく閾値の計算クラス
//
class   ThresholdByProbability extends ThresholdByAverage
{
        // 確率分布
        float[]   probability0;
        float[]   probability1;

        // 両グループの特徴量から閾値を決定する
        public void   determine( float[] features0, float[] features1 )
        {
                // 基底クラス（ThresholdByAverage）の計算処理を実行（初期値として使用）
                super.determine( features0, features1 );

                // ２つの特徴量の度数分布（ヒストグラム）を計算
                makeHistogramsBySize( default_histogram_size );

                // 累積度数分布から確率分布を計算
                makeProblility();

                // 各隣接区間（i番目の区間とi+1番目の区間の間の区間）ごとに、
                // 出現確率が等しくなる点があるかどうかを調べる
                for ( int seg_no=0; seg_no<histogram0.length-1; seg_no++ )
                {
                        // 区間の右端・左端の特徴量の値を計算する
                        float   feature_left, feature_right;
                        feature_left = histogram_min_f + histogram_delta_f * ( seg_no + 0.5f );
                        feature_right = histogram_min_f + histogram_delta_f * ( seg_no +
1.5f );

                        // 区間の右端・左端での各グループの出現確率を取得する
                        float   prob0_left, prob1_left, prob0_right, prob1_right;
                        prob0_left = probability0[ seg_no ];
                        prob1_left = probability1[ seg_no ];
                        prob0_right = probability0[ seg_no + 1 ];
                        prob1_right = probability1[ seg_no + 1 ];

                        // 右端・左端で出現確率の高いグループが異なっている、
                        // もしくはどちらかで出現確率が等しければ、
                        // その区間で必ず出現確率が等しい点が存在する
                        if ( /*  要実装  */ )
                        {
                                // 区間内の出現確率が等しい点を計算する
                                // 要実装
                                threshold = ...;
                        }
                }
        }

        // 累積度数分布から確率分布を計算
        protected void   makeProblility()
        {
                probability0 = new float[ histogram0.length ];
                for ( int i=0; i<probability0.length; i++ )
                        probability0[ i ] = (float) histogram0[ i ] / features0.length;

                probability1 = new float[ histogram1.length ];
                for ( int i=0; i<probability0.length; i++ )
                        probability1[ i ] = (float) histogram1[ i ] / features1.length;
        }
}
```

6.4. 誤認識率が等しくなる閾値を計算（ThresholdByCumulative クラス）

３つ目の閾値計算の方法として、２つのグループの出現確率の累積の和が１となるような閾値を計算する。この点を閾値とすると、閾値の左右にある誤りのデータの数（２つのグループの誤認識率）が等しいことになるので、誤認識率が低くなることが期待される。

これまでの例と同様、出現確率の累積を計算する makeCumulative() メソッド、計算結果を描画するためのメソッド、両グループの計算結果を記録する変数（cumulative0、cumulative1）はサンプルプログラムに実装済みなので、閾値を計算する処理を追加する（リストは省略しているので、サンプルプログラムを参照のこと）。

ここで、6.3 節と同様に、ある区間の左端点で出現確率の累積の和の値が１以下であり、右端点での値が１以上であれば、その区間内に求める点が存在することが分かるので、同様のやり方で閾値は計算できる。6.3 節の出現確率の場合とは異なり、累積の和が１になるのは必ず１回のみなので、複数の閾値が見つかる可能性は考慮に入れる必要はない。

6.5. 誤認識率が最小になる閾値を計算（ThresholdByMnimization クラス）

最後の閾値計算の方法では、２つのグループの出現確率の累積の差の絶対値が最大となるような閾値を計算する。この点を閾値とすると、閾値の左右にある誤りのデータの数（２つのグループの誤認識率）の和が最小になるになるので、誤認識率が低くなることが期待される。ここでは、単純に計算した値が最大になるような区間の点を選択する。

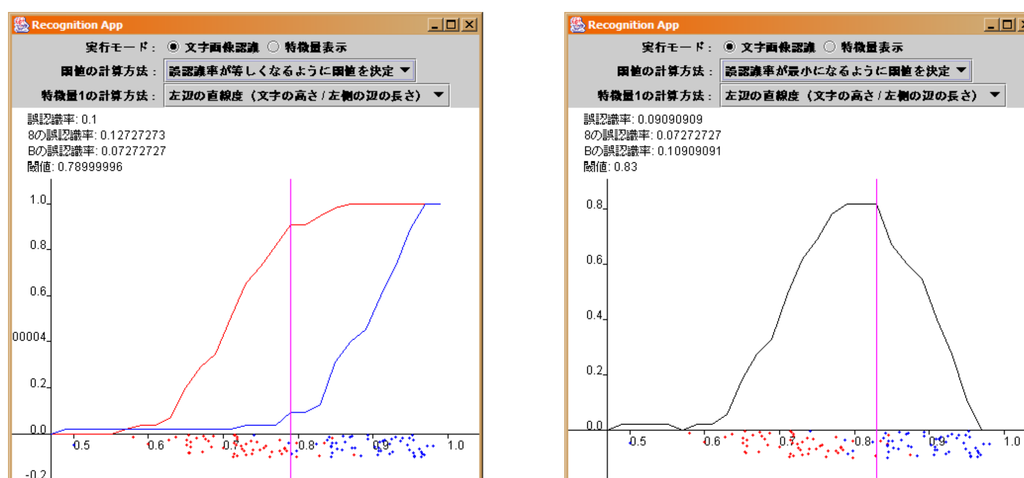

図 11　閾値の計算方法（左：誤認識率が等しくなる値、右：認識誤差が最小になる値）

7. プログラムの設計（2 次元の特徴量を使った認識への拡張）

今までの説明・実装では、１次元の特徴量を使った認識処理を実装した。ここでは、もうひとつ別の特徴量を導入して、２次元の特徴量を使った認識を行えるように改良する。２次元の特徴量を利用することで、より精度の高い認識の実現が期待できる。

ここでまずポイントになるのは、これまでに実装した各処理について、１次元用のクラスやインターフェースを拡張するか、それとも、新たに２次元用に新たなインターフェースを作成するかを決めることである。２次元になることで最も異なる点は、特徴量のデータの受け渡しである。特徴量が２次元になるので、単純にこれを配列として扱うと（画像×特徴量）の２次元の配列になり、メソッドの引数を変更する必要がある。

そこで、特徴量の閾値の計算インターフェース ThresholdDeterminer については、従来のインターフェース・から派生するのではなく、新たなインターフェースを定義することにする。その理由は、そもそも１次元の場合の閾値の計算方法と、２次元の場合の計算方法では全く異なり、無理して１次元用のクラスを継承する意味がほとんどないためである。従来のインターフェースに、新たに２次元配列を引数とするメソッドを追加して拡張することも可能であるが、この場合は、むしろ別のインターフェースとした方が、クラス構成がすっきりして分かりやすい。そこで、２次元用の閾値（境界）計算は新たなインターフェース ThresholdDeterminer2D を定義し、各計算方法はこのインターフェースを実装（継承）することにする。

```
//
//  １次元の特徴量の閾値計算クラスのインターフェース
//
interface  ThresholdDeterminer
{
        // 閾値の決定方法の名前を返す
        public String  getThresholdName();

        // 両グループの特徴量から閾値を決定
        public void  determine( float[] features0, float[] features1 );

        // 与えられた特徴量からどちらの文字かを判定
        public int  recognize( float feature );

        // 閾値を返す
        public float  getThreshold();

        // 特徴空間のデータをグラフに描画（グラフオブジェクトに図形データを設定）
        public void  drawGraph( GraphViewer gv );
}


//
//  ２次元の特徴量の閾値計算クラスのインターフェース
//
interface  ThresholdDeterminer2D
{
        // 閾値の決定方法の名前を返す
        public String  getThresholdName();

        // 両グループの特徴量から閾値を決定
        public void  determine( float[][] features0, float[][] features1 );

        // 与えられた特徴量からどちらの文字かを判定
        public int  recognize( float[] feature );

        // 閾値を返す
        public float  getThreshold( float f0 );

        // 特徴空間のデータをグラフに描画（グラフオブジェクトに図形データを設定）
        public void  drawGraph( GraphViewer gv );
}
```

一方、特徴量の計算処理については、判定は２次元になっても特徴量は次元ごとに独立に計算するだけなので、インターフェースやクラスは変更する必要はない。そのため、従来のクラスをそのまま利用する。ただし、画像認識処理を行う CharacterRecognizer クラスはこれまで１つの FeatureEvaluater オブジェクトのみを持っていたが、２次元の特徴量に対応するためには、次元ごとに FeatureEvaluater オブジェクトを持たせる必要がある。

また、CharacterRecognizer クラスについては、入力としては画像データを扱うので、特徴量が２次元になってもメソッド定義は変更する必要はない。しかし、学習と画像判別の両方で、１次元の特徴量ではなく、２次元の特徴量を利用するように内部処理を拡張する必要がある。そこで、CharacterRecognizer クラスを継承した新たな CharacterRecognizer2D クラスを作成し、従来のメソッドをオーバーライドする。

```
//
//   文字画像認識クラス
//
class   CharacterRecognizer
{
            // 特徴量の評価用オブジェクト
            protected FeatureEvaluater      feature_evaluater;

            // 閾値の決定用オブジェクト
            protected ThresholdDeterminer   threshold_determiner;

            // 学習に使用した画像の特徴量
            protected float   features0[];
            protected float   features1[];


            // 与えられた２つのグループの画像データを判別するような特徴量の閾値を計算
            public void   train( BufferedImage[] images0, BufferedImage[] images1 );

            // 学習結果に基づいて与えられた画像を判別（判別した画像の種類 0 or 1 を返す）
            public int   recognizeCharacter( BufferedImage image );

            // 特徴空間のデータを描画（グラフオブジェクトにデータを設定）
            public void   drawGraph( GraphViewer gv );
}

//
//   文字画像認識クラス（２次元の特徴量に対応した拡張版）
//
class   CharacterRecognizer2D extends CharacterRecognizer
{
            // １次元・２次元のどちらの特徴量を認識に使用するかの設定
            protected int     dimension = 1;

            // 特徴量の評価用オブジェクト
            protected FeatureEvaluater      feature_evaluater2;

            // 閾値の決定用オブジェクト（２次元の特徴量への対応版）
            protected ThresholdDeterminer2D   threshold_determiner_2d;

            // 学習に使用した画像の特徴量（２次元の特徴量）
            protected float   features2d0[][];
            protected float   features2d1[][];


            // 与えられた２つのグループの画像データを判別するような特徴量の閾値を計算
            public void   train( BufferedImage[] images0, BufferedImage[] images1 );

            // 学習結果に基づいて与えられた画像を判別（判別した画像の種類 0 or 1 を返す）
            public int   recognizeCharacter( BufferedImage image );

            // 特徴空間のデータを描画（グラフオブジェクトにデータを設定）
            public void   drawGraph( GraphViewer gv );
}
```

以上を踏まえて、新たなクラス構成を設計する。結論としては、閾値の計算処理については派生を行わず、全く新しいインターフェース ThresholdDeterminer2D を定義し、特徴量の計算 FeatureEvaluater は従来のクラスをそのまま使用して、画像認識クラス CharacterRecognizer については従来のクラスを派生させた CharacterRecognizer2D を定義する。このように、すでにある機能を拡張する場合は、従来の方法との引数・内部処理がどの程度で変更されるかに応じて、継承を利用するかどうかを判断することが必要である。

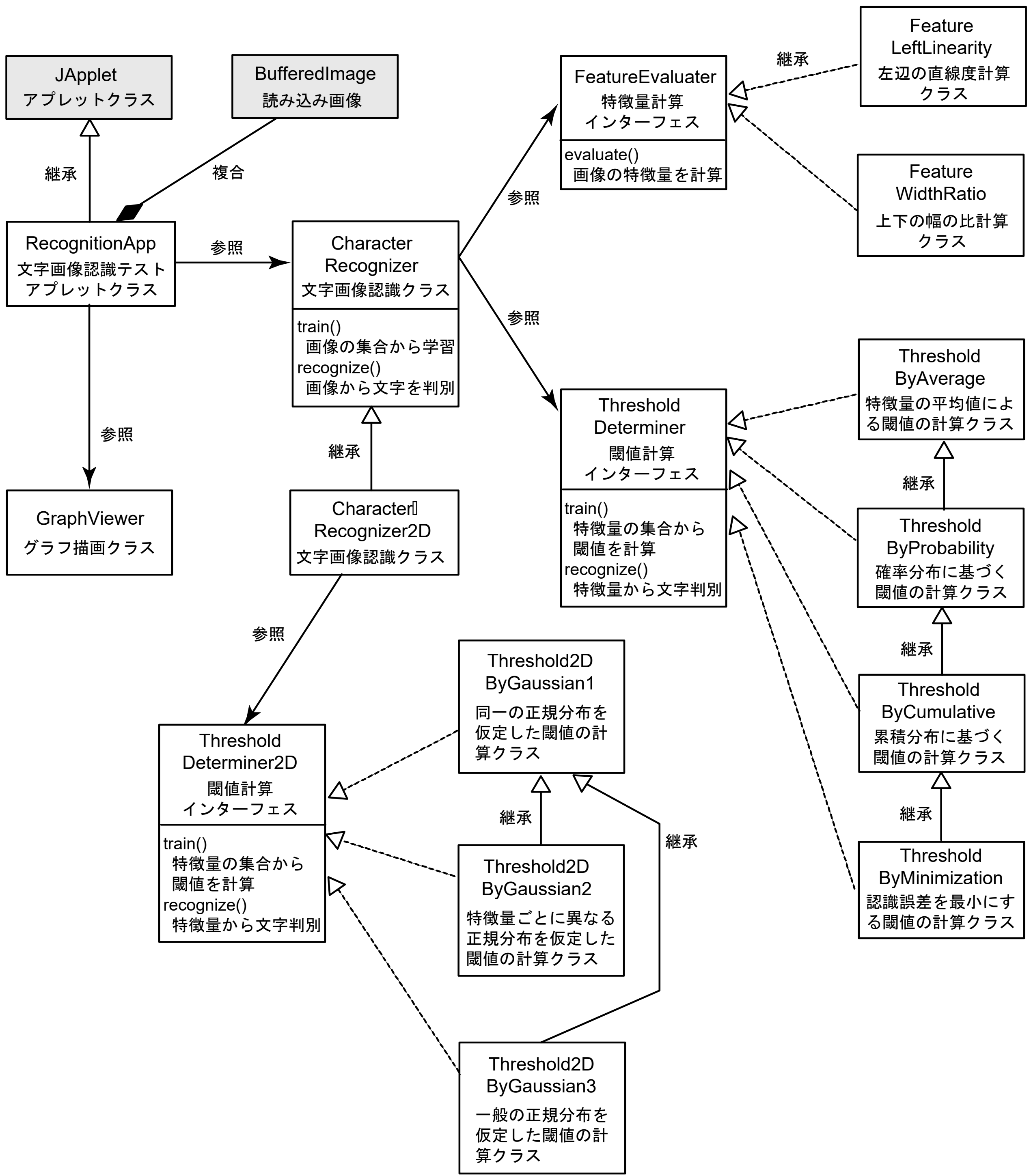

図 12　２次元の特徴量を使った文字画像認識に対応したクラス構成図（クラス図）

8.　特徴量（上下の幅の比）の計算アルゴリズムの実装

２次元の特徴量を利用するためには、これまでの特徴量（左側の辺の長さ）に加えて、新たな特徴量を追加する必要がある。そこで、２次元の実装に移る前に、まず、２つ目の特徴量を実装する。

新たな特徴量として、上部と下部の幅の比率を利用する。この特徴量を計算するクラスを、FeatureWidthRatio クラスとして定義する。このクラスについては、前の特徴量計算クラス FeatureLeftLinerity を継承する必要はないので、新たに FeatureEvaluater インターフェースを実装したクラスとして定義する。クラス定義の書き方は、FeatureLeftLinerity や下記のサンプルを参照。

8.1.　上部と下部の幅の抽出

基本的な考え方としては、以下のような流れで処理をすれば良い。

1. 各ラインの文字の幅を調べる。
2. 画像の中央部で文字の幅が極小になるラインを見つけ、そのラインで文字を上部と下部に分けて考える。
3. 上部で文字の幅が極大になるライン、下部で文字の幅が極大になるラインを見つける。
4. 両ラインにおける文字の幅を、上部の幅・下部の幅として、特徴量（上部の幅 ／ 下部の幅）を計算する。

下記のサンプルコードには、上の処理の流れと、見つかったラインを描画するためのプログラムが記述してある。この流れに従って、実際の処理を記述すれば、特徴量は計算できる。

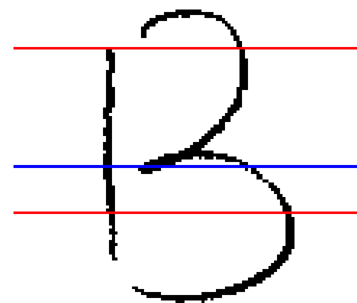

図 13　上部と下部の幅の計算

```
list5.txt　FeatureWidthRatio.java

import java.awt.image.BufferedImage;
import java.awt.Graphics;
import java.awt.Color;

//
//	文字画像の上部と下部の幅の比率を特徴量として計算するクラス
//
class　FeatureWidthRatio implements FeatureEvaluater
{
	// 上部の幅と下部の幅
	float　upper_width;
	float　lower_width;

	// 各行の幅（画像の各行の両端にあるドットのＸ座標の差、行にドットがなければ 0）
	protected int　line_width[];

	// 上部・中央部・下部の行番号
	protected int　upper_line;
	protected int　middle_line;
	protected int　lower_line;

	// 最後に特徴量計算を行った画像（描画用）
	protected BufferedImage　last_image;


	// 特徴量の名前を返す
	public String　getFeatureName()
	{
```

```
		return	"上下の幅の比（上部の幅 / 下部の幅）";
	}

	// 文字画像から 1 次元の特徴量を計算する
	public float	evaluate( BufferedImage image )
	{
		int	height = image.getHeight();
		int	width = image.getWidth();

		// 画像を記録（描画用）
		last_image = image;

		// 各行の幅を計算（各行の両端のドットの X 座標を調べる）
		....

		// 中央部（幅が最小になる行）を探索
		....

		// 上部（幅が最大になる行）を探索
		....

		// 下部（幅が最大になる行）を探索
		....

		// 上部・下部の幅を取得
		....

		// 特徴量（上部の幅 / 下部の幅）を計算
		return	upper_width / lower_width;
	}

	// 最後に行った特徴量計算の結果を描画する
	public void	paintImageFeature( Graphics g )
	{
		if ( last_image == null )
			return;

		// 文字画像を描画
		int	ox = 0, oy = 0;
		g.drawImage( last_image, ox, oy, null );

		// 上部・中央部・下部にラインを描画
		g.setColor( Color.RED );
		g.drawLine( ox, oy + upper_line, ox + last_image.getWidth(), oy + upper_line );
		g.setColor( Color.BLUE );
		g.drawLine( ox, oy + middle_line, ox + last_image.getWidth(), oy + middle_line );
		g.setColor( Color.RED );
		g.drawLine( ox, oy + lower_line, ox + last_image.getWidth(), oy + lower_line );

		// 特徴量を表示
		String	message;
		g.setColor( Color.RED );
		message = "上部の幅: " + upper_width;
		g.drawString( message, ox, oy + 16 );
		message = "下部の幅: " + lower_width;
		g.drawString( message, ox, oy + 32 );
		message = "特徴量(上部の幅 / 下部の幅): " + upper_width / lower_width;
		g.drawString( message, ox, oy + 48 );
	}
}
```

9. ２次元の特徴量の閾値（識別境界）の計算アルゴリズムの実装

２次元の特徴空間での境界を決めるための方法として、最初の講義で、正規分布を用いた３通りの方法を学習した。これらの計算方法を実現するクラスを作成する。以下に、それぞれの概要を説明する。

３種類の方法は、どれも、２つの文字の特徴量が正規分布に従って分布すると仮定する点で同じである。１つ目の方法（Threshold2DByGaussian1 クラス）では、２つの文字の２次元の特徴量が、全て同一の正規分布に従って（同一の分散で）分布すると仮定する。２つ目の方法は（Threshold2DByGaussian2 クラス）、特徴量ごとに異なる分散を考えるが、同じ特徴量であればどちらの文字も分散は同じであると仮定する。３つ目の方法（Threshold2DByGaussian3 クラス）では、そのような仮定は用いず、一般的な正規分布に従って分布するものとする。

ここで、最初の２つの方法では２つの文字の識別境界が直線になるが、３つ目の方法では２次曲線になることに注意する。当然ながら、後の方法になるほど仮定が少なくなるので、特に実際の特徴量の分布状況がこれらの仮定と異なる場合は、後の方法になるほど認識精度が高くなることが期待される。

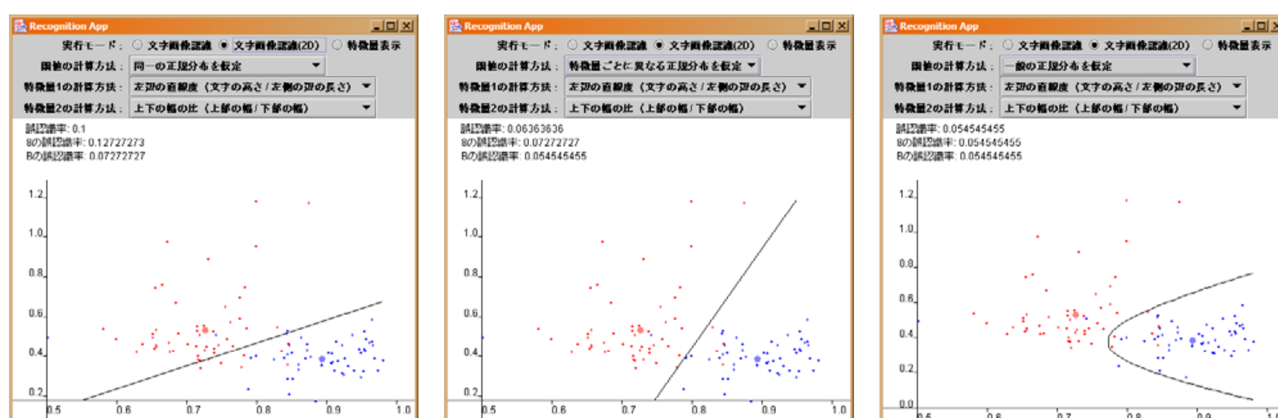

図 14　２次元の特徴量の識別境界の計算方法（左：同一の正規分布を仮定、中央：特徴量ごとに異なる正規分布を仮定、右：一般の正規分布を仮定）

9.1. 同一の正規分布に従うと仮定（Threshold2DByGaussian1）

同一の正規分布に従う場合、２つのグループの境界線は、２つのグループの平均点からの距離が等しい直線である。従って、境界線は、２つの平均点の中央を通る。また、境界線の傾きは、２つの平均点を結ぶ直線と直交に交わる。これらの条件から、境界線の方程式が計算できる（図 14 左は、縦軸と横軸の縮尺が異なるので、一見、平均点を結ぶ線分と境界線が直交しているように見えないが、実際には直交していることに注意）。

サンプルプログラムでは、境界線の方程式は、境界線を通る１つの点（２つのグループの平均点の中央点）と、境界線の傾きによって表現している。従って、determine() メソッドでは、入力された特徴量から、これらの係数を計算してやれば良い。ただし、単純に１本の境界線だけでは、境界線の上側と下側のどちらがグループ０なのかが分らない。そこで、6 章で説明した１次元の閾値と同様に、どちらが上かを表すフラグ変数 is_first_smaller を用意する。determine() メソッドでは、このフラグも正しく設定する必要がある。なお、今回作成するクラスでは、１つ目の特徴量を x 軸、２つの特徴量を y 軸として扱うものとする。

これらの変数をもとに、recognize() メソッドでは、与えられた特徴量を持つ画像がどちらのグループに含まれるかを判定する。また、計算した境界線を描画する処理 drawBorderLine() も追加する。どちらもサンプルプログラムで実装済みなので、実装する必要はない。

同一の正規分布に従うと仮定した２次元の特徴の境界の計算の追加

```
//
//  同一の正規分布に基づく２次元の特徴量の閾値の計算クラス
//
class  Threshold2DByGaussian1 implements ThresholdDeterminer2D
{
	// 特徴量の平均値
	protected float  mean0x, mean0y;
	protected float  mean1x, mean1y;

	// 閾値（境界）の方程式
	protected float  border_ox, border_oy; // 境界線の中心点
	protected float  border_dy; // 境界線の傾き

	// 閾値の符号（グループ０の方が特徴量が閾値よりも小さければ真）
	protected boolean  is_first_smaller;

	// 両グループの特徴量から閾値を決定
	public void  determine( float[][] features0, float[][] features1 )
	{
		// 特徴量の平均値（mean0x, mean0y, mean1x, mean1y）を計算
		// 要実装

		// 境界の方程式を計算（２つの平均値からの距離が等しくなる直線を境界とする）
		// border_ox, border_oy, border_dy を求める
		// 要実装

		// 境界の符号を判定
		// （グループ０が特徴量が境界よりも下にあれば is_first_smaller を真にする）
		// 要実装

	}

	// 閾値をもとに特徴量から文字を判定する
	public int  recognize( float[] feature )
	{
		// グループ０の特徴量 y < 閾値 y < グループ１の特徴量 y
		if ( is_first_smaller )
		{
			if ( feature[ 1 ] < getThreshold( feature[ 0 ] ) )
				return  0;
			else
				return  1;
		}
		// グループ１の特徴量 y < 閾値 < グループ０の特徴量 y
		else
		{
			if ( feature[ 1 ] < getThreshold( feature[ 0 ] ) )
				return  1;
			else
				return  0;
		}
	}

	// 閾値を返す
	public float  getThreshold( float x )
	{
		float  y;
		y = ( x - border_ox ) * border_dy + border_oy;
		return  y;
	}
}
```

9.2.　特徴量ごとに分散の異なる正規分布に従うと仮定（Threshold2DByGaussian2）

特徴量ごとに分散の異なる正規分布に従う場合、境界線が平均点の中央点を通る点では同じだが、各軸の特徴量の分散に応じて、傾きが変化する。分散が大きい特徴量については、平均値からの距離が離れても、分布からは大きくは外れないと考えられる。逆に、分散が小さい特徴量については、平均値からの距離が少し離れただけでも、分布から大きく外れていると考えられる。そこで、この方法では、前の同一の正規分布を仮定する方法で求めた境界線の傾きに、y 軸方向の分散の大きさ／x 軸方向の分散の大きさをかけることで、傾きを計算する。

9.3.　一般の正規分布に従うと仮定（Threshold2DByGaussian3）

一般の正規分布の場合は、講義で学習したように、各特徴量同士の共分散行列、及び、その逆行列を計算し、マハラノビス距離によってどちらのグループに属するかを判定すれば良い。一般の正規分布については、サンプルプログラムで既に一通り実装されているので、確認すること。

境界線についても、マハラノビス距離が一致するという条件のもとに双曲線の方程式を求めることで、描画することができる。こちらについては、詳しい説明は省略する。

10. プログラムの実装（2 次元の特徴量を利用するように拡張）

10.1. 特徴量（上下の幅の比）の計算（FeatureWidthRatio クラス）

9 章の説明に従って、新たな特徴量（上下の幅の比）を計算する FeatureWidthRatio クラスを定義・実装する。講義のページに FeatureWidthRatio クラスのサンプルリストを用意しているので、とりあえずはこれをそのまま使用して構わない。

次に、以下のように、メインアプリケーションの初期化処理で、新たな特徴量を利用できるようリストに追加する。ここまで追加を行った時点で、従来の 1 次元での文字画像認識で、新たに今回追加した特徴量が使えるようになる。プログラムを実行して、今回の特徴量が期待通りに計算できていることを確認すること。

特徴量計算モジュールの追加

```
//
//   文字画像認識テスト アプレット
//
public class RecognitionApp extends JApplet
{
        // 初期化処理
        public void   init()
        {
                ....
                // 利用可能な特徴量計算モジュールを初期化
                features = new FeatureEvaluater[ 2 ];
                features[ 0 ] = new FeatureLeftLinerity();
                features[ 1 ] = new FeatureWidthRatio();
                ....
        }
}
```

10.2. 2 次元の特徴量を使った認識への拡張（ThresholdDeterminer2D、CharacterRecognizer2D）

7 章の設計に従い、2 次元の特徴量を使った認識に対応するための拡張を行う。講義のページに、拡張のためのクラスのソースファイルをまとめたアーカイブを置いてあるので、それをコピーして使用する。

http://www.cg.ces.kyutech.ac.jp/lecture/project/CharacterRecognizer_Step3.zip

具体的には、ThresholdDeterminer2D インターフェース、CharacterRecognizer2D クラスを新たに追加する。また、メインアプレットである RecognitionApp クラスを、CharacterRecognizer2D を使うように変更し、ユーザインターフェースとして２次元での文字画像認識を選べるように拡張する。ファイルを追加したら、プログラムが問題なく起動できることを確認せよ。

10.3. 2 次元の特徴量を使った閾値（識別境界）の計算クラスを作成（Threshold2DByGaussian1 クラス）

２次元の特徴空間での識別境界を計算するための３つのクラス（Threshold2DByGaussian1、Threshold2DByGaussian2、Threshold2DByGaussian3）を作成する。10.2 節のアーカイブにこれらのファイルも含まれているので、一緒にコピーする。このうち、Threshold2DByGaussian1 クラス、及び、Threshold 2DByGaussian2 クラスについては、識別処理を行う determine()メソッドが作成されていないので、各自で作成する必要がある。

まずは、9.1 節の説明に従って、Threshold2DByGaussian1 クラスの determine()メソッドを作成する。実装が終わったら、プログラムを実行して、２次元の特徴量を使った認識結果を確認すること。

10.4. 2 次元の特徴量を使った閾値（識別境界）の計算クラスを追加（Threshold2DByGaussian2~3 クラス）

Threshold2DByGaussian1 クラスを使った識別が問題なく動作したら、引き続き、9.2 節の説明に従って Threshold2DByGaussian2 の determin()メソッドを作成する。

なお、Threshold2DByGaussian3 クラスについては、9.3 節の説明の通り、サンプルプログラムをそのまま利用して構わない。３通りの識別境界の計算クラスの組み込みが終わったら、プログラムを実行して、各手法を使ったときの識別結果を比較してみる。

11. 異なる文字画像に対応したプログラムへの拡張

ここまでの内容で、「８」と「Ｂ」の手書き画像を使用して画像認識を行うプログラムを作成した。次は、各グループで識別対象とする２つの文字を選んで、その２つの文字を認識できるようなプログラムに変更する。必ずしも文字に限定する必要はなく、記号や図などでも良い。

現在のプログラムは、基本的にはどのような文字にも対応できるようになっている。しかし、現在の特徴量は「８」と「Ｂ」の認識を意識したものになっているので、別の文字の認識を行う場合は、その文字に応じた新たな特徴量の計算を追加してやる必要がある。

以下、プログラムを変更する上で必要な作業をまとめる。

11.1. 手書き文字のスキャンと画像ファイルの作成

画像認識のテストを行うためには、テストに使用する画像ファイルが必要である。そこで、グループで決めた２種類の文字を手書きしたものをなるべく多く（１種類あたり 100 サンプル以上）用意する。手書き文字は講義ホームページで用意されている画像スキャン用フレームを利用する。このとき、なるべく友人などに頼み、いろんなタイプの文字が混じるようにする。ただし、同じ人物が書いた文字が多くあるとデータの偏りが生じるため、同一人物からは１種類の文字あたり最大 10 サンプルとする。

手書き文字が用意できたら、スキャナで読み込む。スキャン画像から個々の文字を切り出して２値化を行うプログラムを講義ホームページで用意されているため、必要であればこれを利用する。

11.2. 読み込む画像ファイルの設定変更

ReactionApp クラスのメンバ変数に設定されている読み込み画像のファイル名の情報を、上で用意したファイル名に変更する。例えば、下記のようなサンプルプログラムの記述であれば、pic8_001.gif～pic8_055.gif をグループ 0（文字 0）の画像として、また、picB_001.gif～picB_055.gif をグループ 1（文字 1）の画像とし

て読み込む。

```
読み込む画像ファイルの設定変更（RecognitionApp.java）

//
//  文字画像認識テスト アプレット
//
public class RecognitionApp extends JApplet
{

        // 読み込む画像の設定
        protected String    image_dir = "Samples8Bgif/"; // フォルダ名（相対パス）
        protected String    image_ext = ".gif";          // 拡張子
        protected String    image_name0 = "pic8_";       // 頭につける文字列（文字 0）
        protected String    image_name1 = "picB_";       // 頭につける文字列（文字 1）
        protected int       num_images0 = 55;            // ファイル数（文字 0）
        protected int       num_images1 = 55;            // ファイル数（文字 1）
        protected int       image_digits = 3;            // 連番の桁数
        protected String    character0 = "8";            // 文字名（表示用）（文字 0）
        protected String    character1 = "B";            // 文字名（表示用）（文字 0）

}
```

11.3. 特徴量計算クラスの追加

特徴量を計算するクラスを自分で追加する。基本的には、FeatureEvaluater インターフェースを継承した新しいクラスを作成する。もし必要であれば、既存のクラス（FeatureLeftLinerity や FeatureWidthRatio）から派生させても構わない。クラスの名前は、分かりやすいものを自分で決める。

```
新しい特徴量計算クラスの追加（MyNewFeature.java）

//
//  特徴量・・・を計算するクラス
//
class   FeatureWidthRatio implements FeatureEvaluater
{
        //
        //  FeatureLeftLinerity, FeatureWidthRatio を参考に作成
        //
}
```

新しいクラスを作成したら、これまでと同様に、メインプログラムの初期化時に特徴量計算オブジェクトを配列に格納するように修正する。これで、自分の追加した特徴量が選択できるようになる。

```
特徴量計算オブジェクトの追加（RecognitionApp.java）

//
//  文字画像認識テスト アプレット
//
public class RecognitionApp extends JApplet
{
        // 初期化処理
        public void   init()
        {
                // 利用可能な特徴量計算モジュールを初期化
                features = new FeatureEvaluater[ 3 ];
                features[ 0 ] = new FeatureLeftLinerity();
                features[ 1 ] = new FeatureWidthRatio();
                features[ 2 ] = new MyNewFeature();

                // 後は変更なし
        }
}
```

12. Cross Validation 法および Bootstrap 法による分類精度

誤認識率を計算する際に、全ての画像を学習・評価に用いるか、または Cross Validation 法や Bootstrap 法を用いるかを切り替えることのできる機能を追加する。

12.1. Cross Validation 法および Bootstrap 法の実装

まず、学習用画像を保持するための配列と評価用画像を保持するための配列を準備する必要がある。ここでは、それぞれの文字に対して学習用画像と評価用画像の２つの配列を準備する。RecognitionApp クラス（RecognitionApp.java）に以下の配列宣言を追加する。

```
// 学習用画像
protected BufferedImage   training_images0[];
protected BufferedImage   training_images1[];

// 評価用画像
protected BufferedImage   evaluation_images0[];
protected BufferedImage   evaluation_images1[];
```

以下では、列挙型により、学習用画像と評価用画像の決定方法として、学習用画像と評価用画像が同一として分類精度を調べる場合に USE_ALL_SAMPLES、Cross Validation 法を使用する場合に CROSS_VALIDATION、Bootstrap 法を使用する場合に BOOTSTRAP という値を持つとする。列挙型とは、プログラマが任意に作成できるある特定の値のみを持つ型のことであり、定数を表現する専用の型と考えることができる。定数をそのまま記述でき、ソースコードの可読性を高めることができる。

```
// 学習用・評価用画像の決定方法の設定
enum   DistributionMethod
{
        USE_ALL_SAMPLES,
        CROSS_VALIDATION,
        BOOTSTRAP
};

// 学習用・評価用画像の決定方法
protected DistributionMethod   distribution_method =
                DistributionMethod.CROSS_VALIDATION;
```

Cross Validation 法の分割数を変数 cv_number_of_folds で持つことにする。例えば、4-Fold の場合は 4、10-Fold の場合は 10 を設定する。また分割されたどのグループを評価用画像として使用するかを変数 cv_evaluation_fold により設定することにする。これらの値は後で述べる sampleDistribution()メソッドの中で利用されることに注意する。

```
// Cross Validation 法を用いるときのグループ数
protected int   cv_number_of_folds = 4;

// Cross Validation 法を用いるとき、何番目のグループを評価に使用するか設定
protected int   cv_evaluation_fold = 0;
```

cv_evaluation_fold の値を 0, 1, …, (cv_number_of_folds – 1) のいずれかに設定することで、何番目の

グループを評価に用いるかを変更できる。

distribution_method の値に従って学習または評価に用いる画像データの決定を RecogniztionApp.java に sampleDistribution()メソッドを追加することにより行う。ただし、Cross Validation 法、Bootstrap 法については必要な部分を各自で完成させる必要がある。

```
// 学習・評価に用いるサンプル画像の決定
public void  sampleDistribution()
{
	// サンプル画像が読み込まれていなければ終了
	if ( ( sample_images0 == null ) | | ( sample_images1 == null ) )
		return;

	// 学習用画像の配列を削除する
	training_images0 = null;
	training_images1 = null;

	// 評価用画像の配列を削除する
	evaluation_images0 = null;
	evaluation_images1 = null;

	// 全てのサンプル画像を学習と評価に使用
	if ( distribution_method == DistributionMethod.USE_ALL_SAMPLES )
	{
		// 全てのサンプル画像を学習用画像の配列にコピー
		training_images0 = new BufferedImage[ sample_images0.length ];
		for ( int i=0; i<sample_images0.length; i++ )
			training_images0[ i ] = sample_images0[ i ];
		training_images1 = new BufferedImage[ sample_images1.length ];
		for ( int i=0; i<sample_images1.length; i++ )
			training_images1[ i ] = sample_images1[ i ];

		// 全てのサンプル画像を評価用画像の配列にコピー
		evaluation_images0 = new BufferedImage[ sample_images0.length ];
		for ( int i=0; i<sample_images0.length; i++ )
			evaluation_images0[ i ] = sample_images0[ i ];
		evaluation_images1 = new BufferedImage[ sample_images1.length ];
		for ( int i=0; i<sample_images1.length; i++ )
			evaluation_images1[ i ] = sample_images1[ i ];
	}

	// Cross Validation  法を使用
	if ( distribution_method == DistributionMethod.CROSS_VALIDATION )
	{
		// 全サンプル画像の何番目～何番目のデータを評価に使用するかを決定 （文字 0）
		//（cv_number_of_folds, cv_evaluation_fold をもとに決定）
		int evaluation_begin0;  // 評価データの先頭
		int evaluation_end0;    // 評価データの最後尾
		int evaluation_count0;  // 評価データの個数
```

```
			//　ここは各自で完成

			// 全サンプル画像の何番目～何番目のデータを評価に使用するかを決定　（文字 1）
			//（cv_number_of_folds, cv_evaluation_fold をもとに決定）
			int evaluation_begin1;　// 評価データの先頭
			int evaluation_end1;　　// 評価データの最後尾
			int evaluation_count1;　// 評価データの個数

			//　ここは各自で完成

			// 全サンプル画像を評価用画像と学習用画像の配列に分配　（文字 0）

			//　ここは各自で完成

			// 全サンプル画像を評価用画像と学習用画像の配列に分配　（文字 1）

			//　ここは各自で完成

		}

		// BOOTSTRAP 法を使用
		if ( distribution_method == DistributionMethod.BOOTSTRAP )
		{
			// 全てのサンプル画像を学習用画像の配列にコピー
			training_images0 = new BufferedImage[ sample_images0.length ];
			for ( int i=0; i<sample_images0.length; i++ )
				training_images0[ i ] = sample_images0[ i ];
			training_images1 = new BufferedImage[ sample_images1.length ];
			for ( int i=0; i<sample_images1.length; i++ )
				training_images1[ i ] = sample_images1[ i ];

			// 評価用画像を重複を許してランダムに選択

			// ここは各自で完成

			/* 乱数は java.lang.Math.random()メソッドで得ることができる
			  evaluation_images0[ i ] =
				sample_images0[ (int)( java.lang.Math.random() *
							sample_images0.length ) ]; */

		}
}
```

　学習用と評価用に画像を分けるように変更を行ったため RecognitionAPP.java のメイン処理である reconginitionTest() の一部を変更する必要があるので注意する（一部抜粋)。

```
public void　recognitionTest()
{
		// . . .
```

```
        // 学習・評価に用いるサンプル画像の決定
        sampleDistribution();

        // 学習用画像を使って学習
        recognizer.train( training_images0, training_images1 );

        // 評価用画像を使って誤認識率を計算
        int   error_count[] = { 0, 0 };
        int   char_no;
        error_count[ 0 ] = 0;
        error_count[ 1 ] = 0;
        for ( int i=0; i<evaluation_images0.length; i++ )
        {
                char_no = recognizer.recognizeCharacter( evaluation_images0[ i ] );
                if ( char_no != 0 )
                        error_count[ 0 ] ++;
        }
        for ( int i=0; i<evaluation_images1.length; i++ )
        {
                char_no = recognizer.recognizeCharacter( evaluation_images1[ i ] );
                if ( char_no != 1 )
                        error_count[ 1 ] ++;
        }
        error0 = (float) error_count[ 0 ] / evaluation_images0.length;
        error1 = (float) error_count[ 1 ] / evaluation_images1.length;
        error = (float) ( error_count[ 0 ] + error_count[ 1 ] ) / (float)
                    ( evaluation_images0.length + evaluation_images1.length );

        // . . .

}
```

12.2. 分類精度の調査

各グループで決めた２つの文字に対して 10-Fold Cross Validation 法を適用して、誤認識率の変化、分類精度について調べる。10 回それぞれの場合の誤認識率と、全体での平均・標準偏差を求める。

Bootstrap 法は、今回は作成・実験は行わなくとも良い。

付録. コマンドラインからのコンパイルと実行方法

もし、Eclipse を使用しない場合は、いままでの講義・演習で行ったのとおなじように、コマンドラインからコンパイルや実行を行うこともできる。
コンパイルには、以下のように javac コマンドを使用する。

```
コンパイル方法（特定のファイルのみをコンパイル）
> javac ファイル名
コンパイル方法（全ファイルをコンパイル）
> javac *.java
```

プログラムコンパイルをしたら、以下のようにアプリケーションとして起動する。

```
アプリケーションとして起動する方法
> java RecognitionApp
```

また、RecognitionApp.java にはアプレットとして起動するための設定が記述されているので、以下のようにアプレットとしても起動できる（ただし、この場合は、サンプル画像として BMP 画像は読み込めないので注意する）。

```
アプレットとして起動する方法
> appletviewer RecognitionApp.java
```