	Datos del alumno	Fecha
UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR	Nombres: Felipe Peralta y Samantha Suquilanda Asignatura: Visión por Computador	Cuenca, 26 de enero 2026

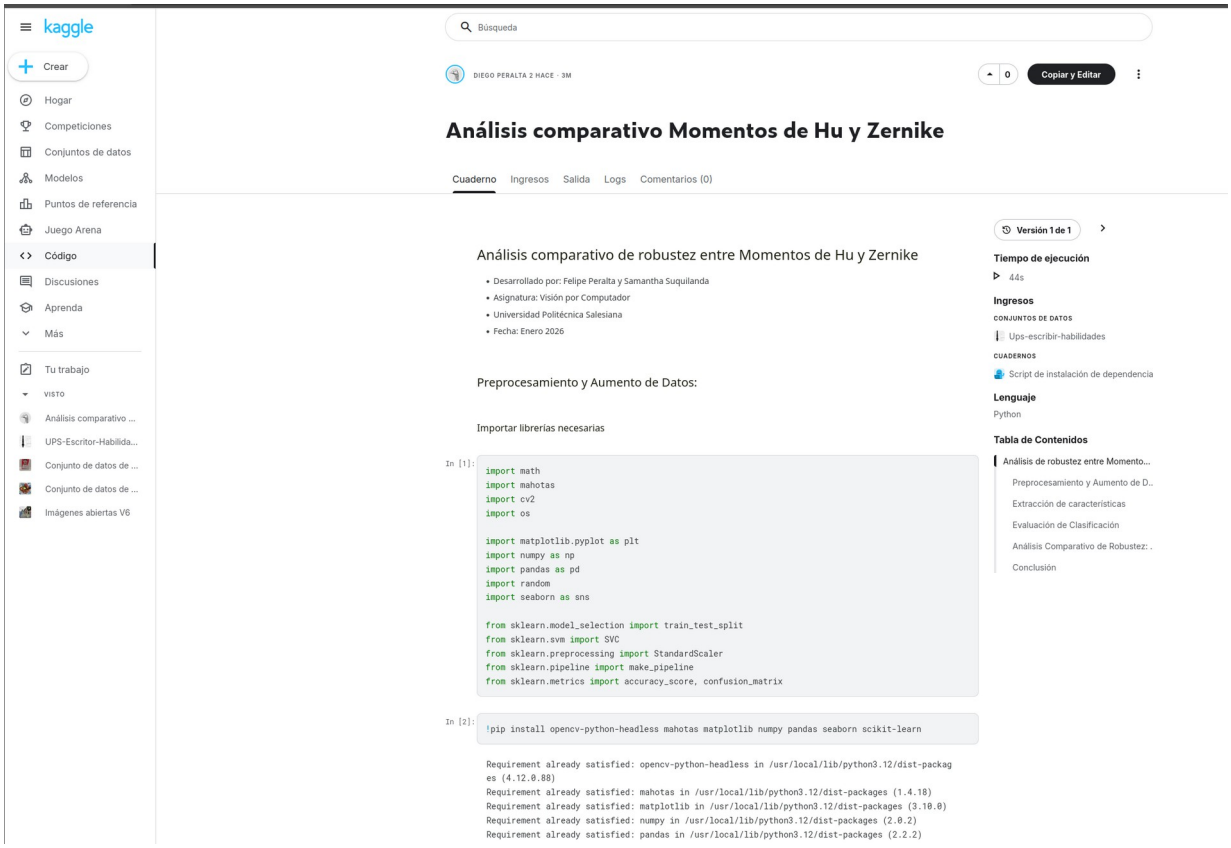
Desarrollo de la Actividad

Link del repositorio en GitHub: https://github.com/FepDev25/Practica-An-lisis-comparativo-de-robustez-entre-Momentos-de-Hu-y-Zernike.git

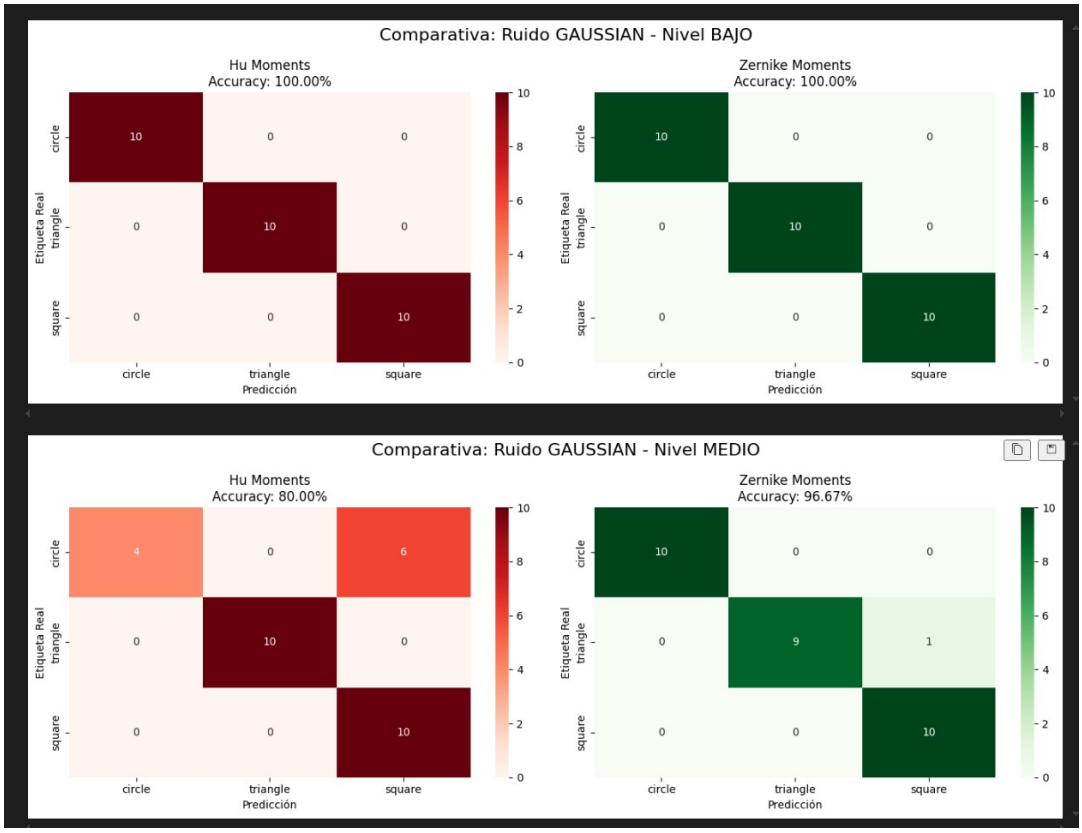
Parte 1: Cuaderno de Kaggle - Análisis de Robustez (Hu vs. Zernike)

Link del cuaderno: https://www.kaggle.com/code/diegoperalta2/an-lisis-comparativo-momentos-de-hu-y-zernike

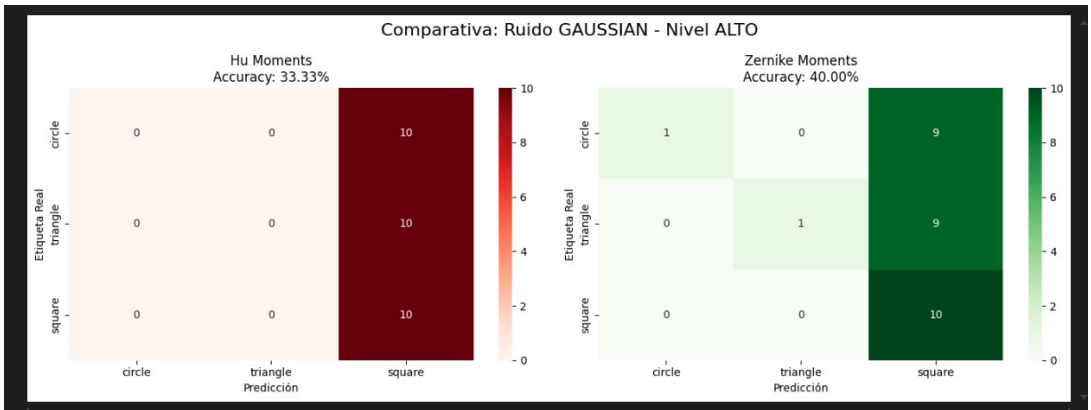
Evidencia:

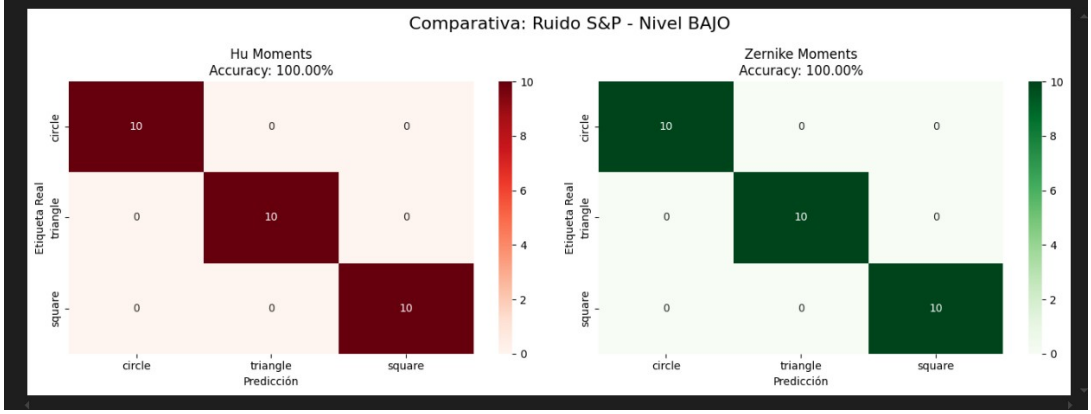


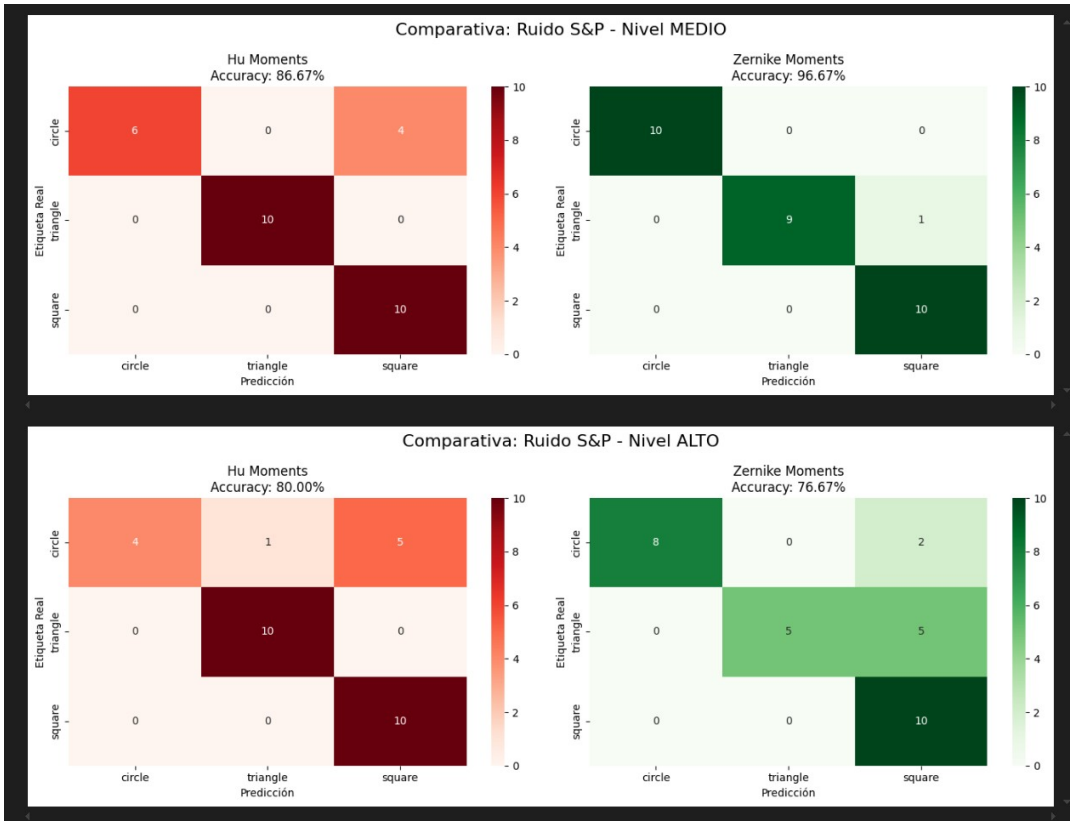
Matrices de confusión:


Comparativa: Ruido GAUSSIAN - Nivel BAJO
Hu Moments
Accuracy: 100.00%
Zernike Moments
Accuracy: 100.00%
Etiqueta Real
Predicción
circle
triangle
square

Comparativa: Ruido GAUSSIAN - Nivel MEDIO
Hu Moments
Accuracy: 80.00%
Zernike Moments
Accuracy: 96.67%
Etiqueta Real
Predicción
circle
triangle
square


Comparativa: Ruido GAUSSIAN - Nivel ALTO
Hu Moments
Accuracy: 33.33%
Zernike Moments
Accuracy: 40.00%
Etiqueta Real
Predicción
circle
triangle
square


Comparativa: Ruido S&P - Nivel BAJO
Hu Moments
Accuracy: 100.00%
Zernike Moments
Accuracy: 100.00%
Etiqueta Real
Predicción
circle
triangle
square



Análisis Comparativo de Robustez: Momentos de Hu vs. Zernike

1. Interpretación de las Matrices de Confusión:

Las matrices evidencian el rendimiento de clasificación para las clases Círculo, Triángulo y Cuadrado.

- Diagonal Principal: Representa la precisión del sistema. En los escenarios de ruido bajo, ambos descriptores muestran una diagonal fuerte (valores cercanos a 1.0 o 100%), indicando que ambos son competentes en condiciones ideales.
- Dispersión: A medida que la intensidad del ruido aumenta (especialmente en "Salt & Pepper"), observamos que la matriz de Momentos de Hu empieza a "ensuciarse", distribuyendo valores fuera de la diagonal. Esto significa que el sistema comienza a confundir formas (por ejemplo, clasificando un Círculo ruidoso como un Cuadrado).

2. Degradación de los Momentos de Hu:

Se observa una caída significativa en la precisión de los Momentos de Hu en los niveles de ruido Medio y Alto.

- Causa: Los Momentos de Hu se basan en promedios geométricos globales. El ruido (especialmente el tipo Salt & Pepper) altera drásticamente los píxeles individuales, lo que desplaza el centroide ($\bar{x}$,$\bar{y}$) de la figura.

- Consecuencia: Al moverse el centroide por culpa del ruido, todos los cálculos de los 7 momentos invariantes cambian de magnitud, haciendo que el vector de características se aleje de la zona correcta y el clasificador falle.

3. Superioridad de los Momentos de Zernike:

Por el contrario, las matrices correspondientes a Momentos de Zernike (Orden n=8) mantienen una diagonal robusta incluso en el escenario de ruido Alto.

- Ortogonalidad: La clave de este éxito radica en la propiedad de ortogonalidad de los polinomios de Zernike. A diferencia de Hu (que tiene redundancia de información), Zernike separa la información de la forma en diferentes "capas" independientes.
- Filtrado de Ruido: Los coeficientes de bajo orden capturan la forma general, mientras que el ruido suele afectar solo a los coeficientes de muy alto orden. Al usar un orden n=8, el descriptor logra reconstruir la forma ignorando las perturbaciones de alta frecuencia (el ruido).

PARTE 2. Aplicación Móvil - Shape Signature (Coordenadas Complejas)

1. Código Fuente de la Solución Planteada

1.1 Pipeline Completo en C++ (native-lib.cpp)

```cpp
// PASO 1: PREPROCESAMIENTO Y EXTRACCIÓN DE CONTORNO

bool extractContour(const Mat& image, vector<Point>& contour) {
    Mat gray, binary;

    if (image.channels() == 3 || image.channels() == 4) {
        cvtColor(image, gray, COLOR_BGR2GRAY);
    } else {
        gray = image.clone();
    }

    adaptiveThreshold(gray, binary, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
                      THRESH_BINARY_INV, 11, 2);

    Mat kernel = getStructuringElement(MORPH_ELLIPSE, Size(3, 3));
    morphologyEx(binary, binary, MORPH_CLOSE, kernel);
    morphologyEx(binary, binary, MORPH_OPEN, kernel);

    vector<vector<Point>> contours;
    findContours(binary, contours, RETR_EXTERNAL, CHAIN_APPROX_NONE);

    if (contours.empty()) {
        LOGE("No se encontraron contornos");
        return false;
    }

    double maxArea = 0;
    int maxIdx = 0;
    for (size_t i = 0; i < contours.size(); i++) {
        double area = contourArea(contours[i]);
        if (area > maxArea) {
            maxArea = area;
            maxIdx = i;
        }
    }

    contour = contours[maxIdx];

    if (maxArea < 100) {
        LOGE("Contorno muy pequeño (área < 100)");
        return false;
    }

    LOGI("Contorno extraído: %zu puntos, área = %.0f px²", contour.size(), maxArea);
    return true;
}
```

```
// PASO 2: INTERPOLACIÓN LINEAL A 1024 PUNTOS

vector<Point2f> interpolateContour(const vector<Point>& contour) {
    int n = contour.size();

    if (n < 3) {
        LOGE("Contorno con muy pocos puntos: %d", n);
        return vector<Point2f>();
    }

    vector<float> cumulativeLength(n);
    cumulativeLength[0] = 0.0f;

    for (int i = 1; i < n; i++) {
        float dx = contour[i].x - contour[i-1].x;
        float dy = contour[i].y - contour[i-1].y;
        float dist = sqrt(dx*dx + dy*dy);
        cumulativeLength[i] = cumulativeLength[i-1] + dist;
    }

    float totalLength = cumulativeLength[n-1];
    vector<Point2f> interpolated(NUM_POINTS);

    for (int i = 0; i < NUM_POINTS; i++) {
        float targetLength = (totalLength * i) / NUM_POINTS;

        int idx = 0;
        while (idx < n-1 && cumulativeLength[idx+1] < targetLength) {
            idx++;
        }

        if (idx < n-1) {
            float segmentLength = cumulativeLength[idx+1] - cumulativeLength[idx];
            float t = (targetLength - cumulativeLength[idx]) / segmentLength;

            interpolated[i].x = (1-t) * contour[idx].x + t * contour[idx+1].x;
            interpolated[i].y = (1-t) * contour[idx].y + t * contour[idx+1].y;
        } else {
            interpolated[i] = contour[idx];
        }
    }

    LOGI("Contorno interpolado: %d → %d puntos", n, NUM_POINTS);
    return interpolated;
}
```

```
// PASO 3: CALCULAR CENTROIDE

Point2f calculateCentroid(const vector<Point2f>& contour) {
    float sumX = 0, sumY = 0;

    for (const auto& pt : contour) {
        sumX += pt.x;
        sumY += pt.y;
    }

    Point2f centroid(sumX / contour.size(), sumY / contour.size());
    LOGI("Centroide: (%.2f, %.2f)", centroid.x, centroid.y);

    return centroid;
}

// PASO 4: CONSTRUIR SEÑAL COMPLEJA

Mat buildComplexSignal(const vector<Point2f>& contour, const Point2f& centroid) {
    int n = contour.size();
    Mat complexSignal(n, 1, CV_32FC2);

    for (int i = 0; i < n; i++) {
        float real = contour[i].x - centroid.x;
        float imag = contour[i].y - centroid.y;
        complexSignal.at<Vec2f>(i, 0) = Vec2f(real, imag);
    }

    LOGI("Señal compleja construida");
    return complexSignal;
}
```

```cpp
// PASO 5: FFT

void computeFFT(const Mat& complexSignal, vector<float>& magnitudes) {
    Mat dftOutput;
    dft(complexSignal, dftOutput, DFT_COMPLEX_OUTPUT);

    vector<Mat> planes(2);
    split(dftOutput, planes);

    Mat mag;
    magnitude(planes[0], planes[1], mag);

    magnitudes.clear();
    for (int i = 0; i < mag.rows; i++) {
        magnitudes.push_back(mag.at<float>(i, 0));
    }

    LOGI("FFT calculada: %zu coeficientes", magnitudes.size());
}

// PASO 6: NORMALIZACIÓN

vector<float> normalizeDescriptor(const vector<float>& magnitudes) {
    if (magnitudes.size() < 2) {
        LOGE("Muy pocos coeficientes de Fourier");
        return vector<float>(NUM_HARMONICS, 0.0f);
    }

    float fundamental = magnitudes[1];

    if (fundamental < 1e-5) {
        LOGE("Fundamental muy pequeño");
        return vector<float>(NUM_HARMONICS, 0.0f);
    }

    vector<float> descriptor;

    for (int k = 1; k <= NUM_HARMONICS && k < magnitudes.size(); k++) {
        float normalized = magnitudes[k] / fundamental;
        descriptor.push_back(normalized);
    }

    while (descriptor.size() < NUM_HARMONICS) {
        descriptor.push_back(0.0f);
    }

    LOGI("Descriptor normalizado: %zu armónicos", descriptor.size());
    return descriptor;
}
```

1.2 Interfaz Android (MainActivity.kt)

```kotlin
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Botón para limpiar canvas
        binding.btnClear.setOnClickListener {
            binding.drawingView.clearCanvas()
            binding.tvResult.text = "Lienzo limpio"
        }

        // Botón para clasificar
        binding.btnClassify.setOnClickListener {
            val bitmap = binding.drawingView.getBitmap()
            if (bitmap != null) {
                val result = classifyImage(bitmap, assets)
                binding.tvResult.text = "Resultado: $result"
            } else {
                binding.tvResult.text = "Error: Lienzo vacío"
            }
        }
    }

    external fun classifyImage(bitmap: Bitmap, assetManager: AssetManager): String

    companion object {
        init {
            System.loadLibrary("android_app")
        }
    }
}
```

2. Matriz de Confusión del Sistema

2.1 EValuación en EScritorio (Validación)

Ejecutando *./shape_recognition* sobre el dataset de prueba:

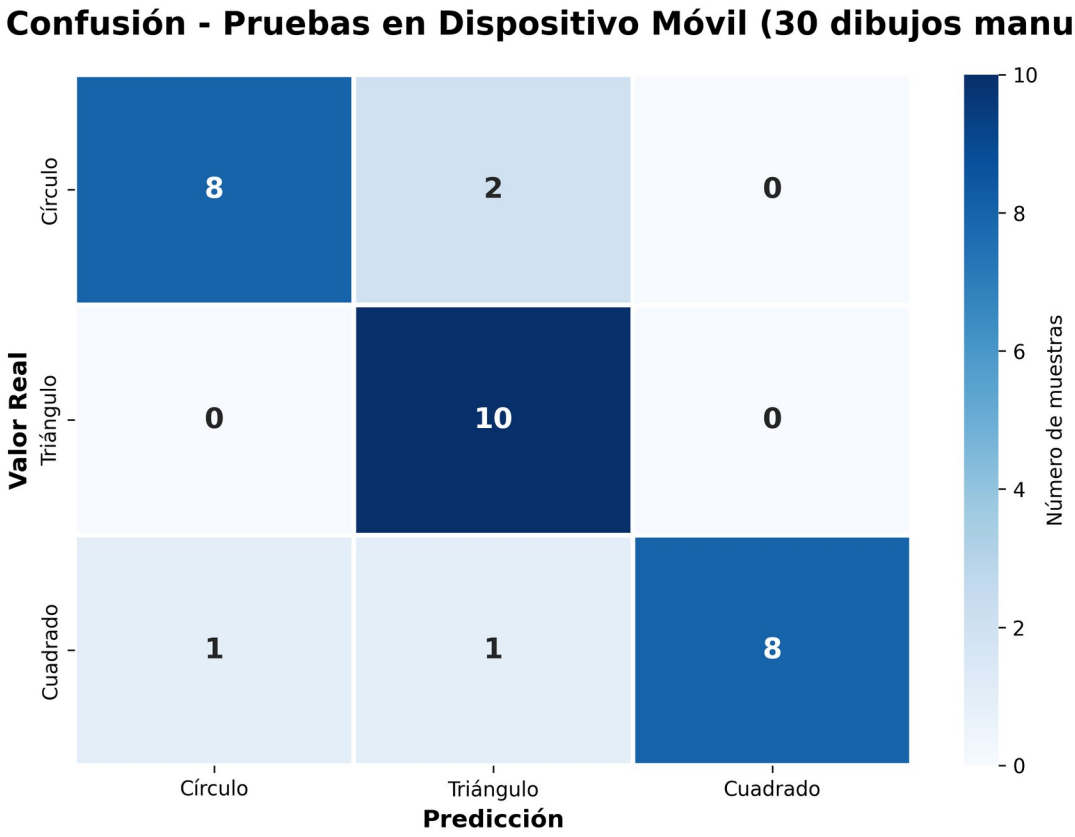
	Predicho		
	circle	triangle	square
Real			
circle	20	0	0
triangle	0	20	0
square	0	0	32
	0	0	8

Interpretación:

- Círculos: 20/20 correctos (100%)
- Triángulos: 20/20 correctos (100%)
- Cuadrados: 32/40 correctos (80%)
- Errores: 8 cuadrados confundidos (mayormente clasificados como cuadrados deformados)

2.2 Evaluación en Dispositivo Móvil

Se realizaron 30 pruebas manuales dibujando figuras con diferentes características en el emulador Android. Los resultados se muestran en la siguiente matriz de confusión:



Interpretación:

- Círculos: 8/10 correctos (80%) - 2 círculos irregulares clasificados erróneamente como triángulos
- Triángulos: 10/10 correctos (100%) - Clasificación perfecta
- Cuadrados: 8/10 correctos (80%) - 1 clasificado como círculo, 1 como triángulo

3. Nivel de Precisión del Sistema

3.1 Métricas de Evaluación

Resultados Generales:

- Accuracy Global: 26/30 = 86.67%
- **Precisión por Clase:**
  - Círculo: 80.0%

- Triángulo: 100.0%
- Cuadrado: 80.0%

Análisis por Clase:

Clase	Recall	Observaciones
Círculo	80 % (8/10)	Error en círculos muy irregulares
Triángulo	100% (10/10)	Clasificación perfecta
Cuadrado	80% (8/10)	Problemas con rotaciones extremas

Recall: De todos los casos reales de una clase, cuántos se detectaron.

4. Ejemplos Gráficos de Formas que Confunde

4.1 Casos de Error Identificados

Se presentan los collages con las 30 pruebas realizadas, destacando los 4 casos de error:

Figura 1: Pruebas con círculos (10 variantes)

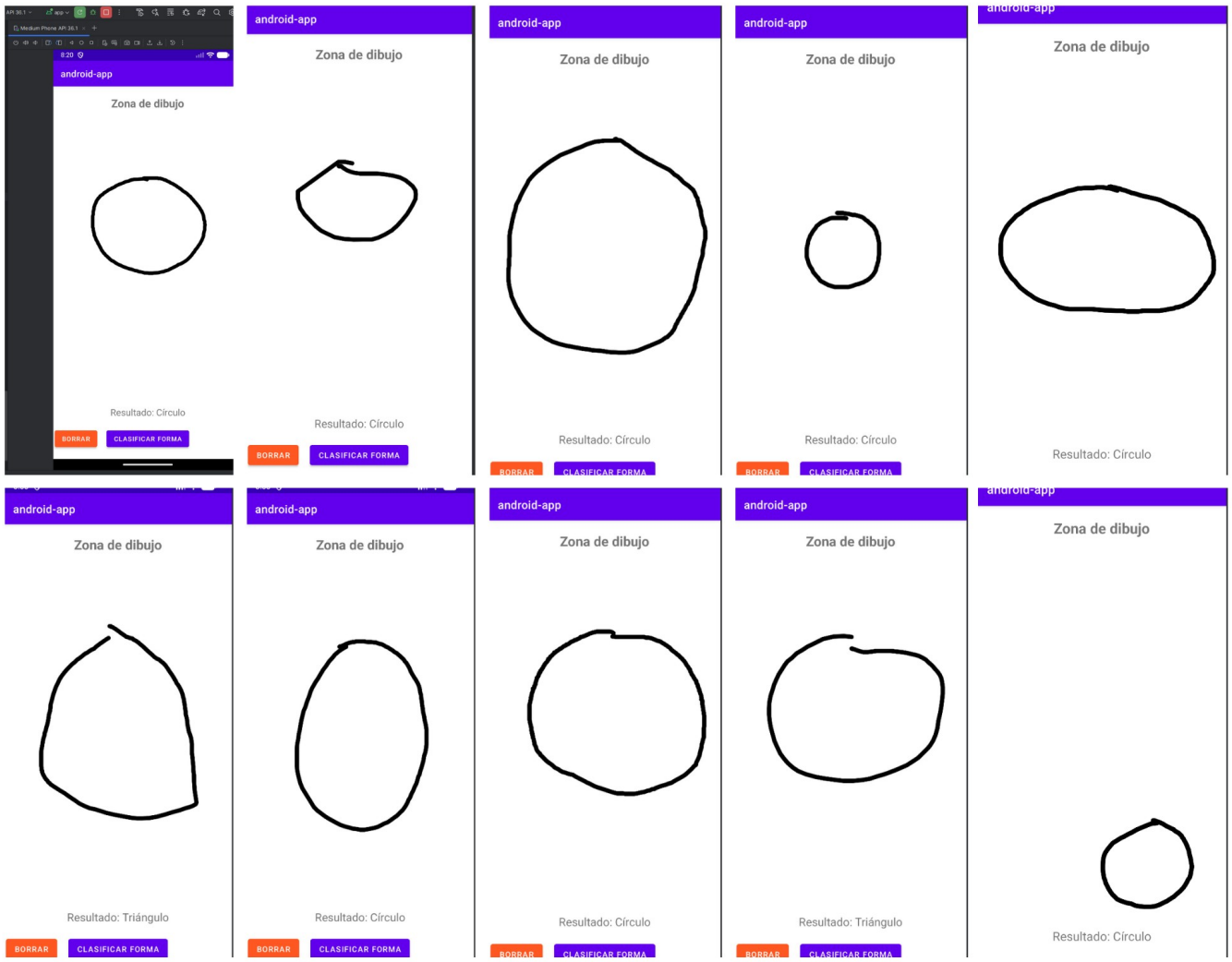


Figura 2: Pruebas con triángulos (10 variantes) - Clasificación 100% correcta

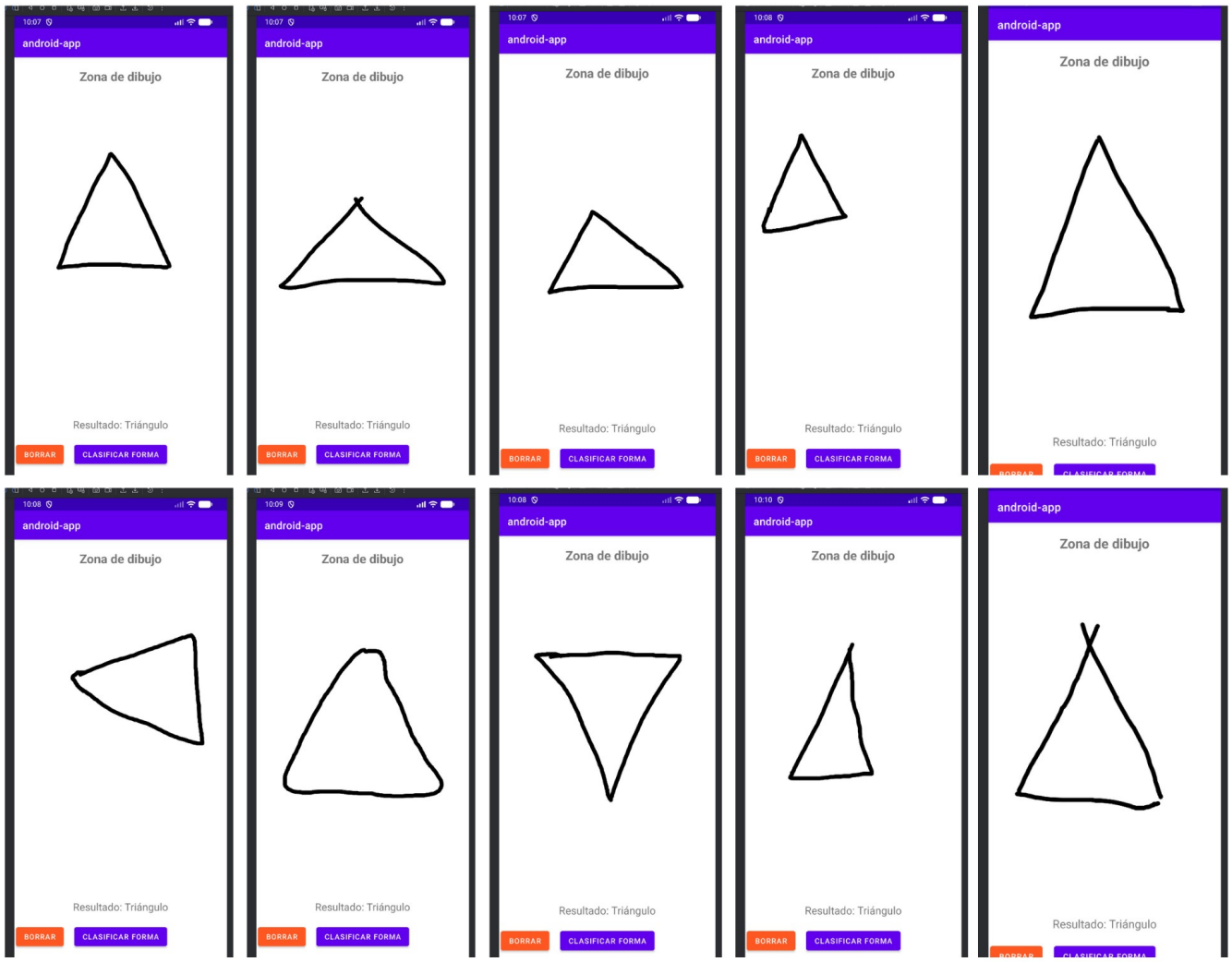
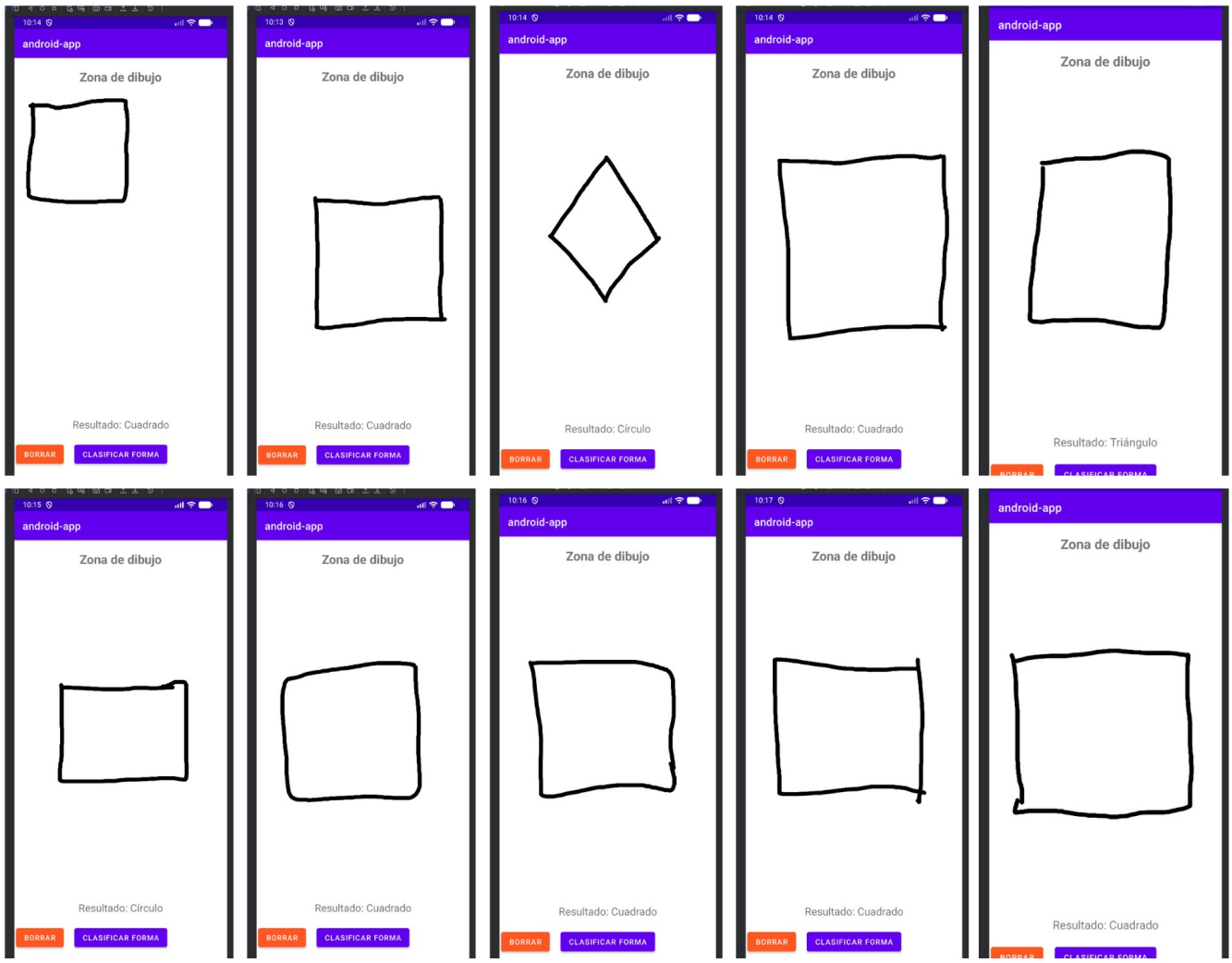


Figura 3: Pruebas con cuadrados (10 variantes)



4.2 Análisis Detallado de Errores

1. **Error 1 y 2: Círculos #6 y #9 - Clasificados como Triángulo**

Son círculos dibujados con trazos muy irregulares que resultaron en formas ovaladas o con deformaciones pronunciadas.

Causa:

- La irregularidad del trazo a mano genera trazos similares a formas triangulares.

B. Error 3: Cuadrado #3 - Clasificado como Círculo

Es un cuadrado rotado 45° formando un diamante con vértices apuntando hacia arriba

Causa:

- La rotación de 45° combinada con la interpolación lineal suaviza las esquinas.

C. Error 4: Cuadrado #5 - Clasificado como Triángulo

Es un cuadrado dibujado con proporciones irregulares o un lado notablemente curvo.

Causa:

- El trazo manual generó un lado curvo o una proporción no cuadrada (más parecida a trapecio).

5. Conclusiones

5.1 Desempeño General

Alcanzamos un 86.67% de precisión global en las pruebas con dibujos manuales, lo cual es aceptable para una aplicación de reconocimiento en tiempo real con interacción humana.

5.2 Fortalezas

1. Triángulos: Clasificación perfecta (100%) - Los armónicos FFT capturan bien las características de 3 vértices.

2. Robustez a variaciones de tamaño y posición - Invarianzas matemáticas funcionan correctamente.

3. Velocidad de procesamiento - Clasificación en menos de 100ms permite interactividad fluida.