image_processor.py - Image Quality Assessment

Overview

This module provides quality control for satellite imagery before detection. It assesses whether images are suitable for reliable solar panel detection by checking brightness, cloud cover, and image detail.

Logic

Class: ImageQualityChecker

Method	Purpose
__init__()	Configure quality thresholds
check_quality()	Assess image suitability for detection

Quality Checks Performed

Check	Threshold	Failure Reason
Minimum brightness	brightness_low (30)	"Image too dark"
Maximum bright pixels	cloud_threshold (70%)	"Heavy cloud cover"
Image variance	min_variance (100)	"Low image detail"

How It Works

1. Image Loading & Preprocessing

```
img = cv2.imread(str(image_path))
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Grayscale conversion simplifies brightness analysis (single intensity channel).

2. Darkness Check

```
┌─────────────────────────────────────┐
│  Check: Is the image too dark?       │
│                                      │
│  mean_brightness = np.mean(gray)     │
│                                      │
│  if mean_brightness < 30:            │
│    → REJECT: "shadows/poor lighting" │
└─────────────────────────────────────┘
```

Why this threshold?

- Pixel values range 0-255
- Mean < 30 indicates nighttime, heavy shadows, or extremely poor lighting
- Solar panels become invisible in such conditions

3. Cloud Cover Detection

```
┌──────────────────────────────────────────────────────────────┐
│  Check: Is there heavy cloud cover?   │
│                                       │
│  bright_pixels = count(gray > 225)    │
│  bright_ratio = bright_pixels / total │
│                                       │
│  if bright_ratio > 0.7:               │
│    → REJECT: "Heavy cloud cover"      │
└──────────────────────────────────────────────────────────────┘
```

Why this works:

- Clouds appear as very bright (near-white) areas
- 225+ pixel intensity is typically cloud/overexposure
- 70% threshold allows partial cloud while rejecting unusable images

4. Detail/Variance Check

```
┌──────────────────────────────────────────────────────────────┐
│  Check: Does the image have detail?   │
│                                       │
│  variance = np.var(gray)              │
│                                       │
│  if variance < 100:                   │
│    → REJECT: "possibly occluded"      │
└──────────────────────────────────────────────────────────────┘
```

Why variance matters:

- High variance = diverse pixel values = visible details
- Low variance = uniform/flat = fog, blank imagery, or data errors
- Solar panels require clear structural detail for detection

Complete Decision Flow

```
┌─────────────────────┐
│   Load Image        │
└──────────┬──────────┘
           │
           ▼
┌──────────────────────┐
│ Convert to Grayscale │
└──────────┬───────────┘
           │
           ▼
┌──────────────────────┐     Fail
│ Mean brightness < 30? ├──────────► "Too dark"
└──────────┬───────────┘
           │ Pass
           ▼
┌──────────────────────┐     Fail
│ Bright ratio > 70%?  ├──────────► "Cloud cover"
└──────────┬───────────┘
           │ Pass
           ▼
┌──────────────────────┐     Fail
│ Variance < 100?      ├──────────► "Low detail"
└──────────┬───────────┘
           │ Pass
           ▼
┌──────────────────────┐
│ Return: "Good quality" │
└──────────────────────┘
```

Why It Works

Grayscale Analysis

Color images have 3 channels (BGR). Grayscale:

- Reduces complexity to 1 channel
- Captures luminance (perceived brightness)
- Is sufficient for quality metrics

Thresholds Are Empirically Tuned

These values were calibrated on satellite imagery:

Threshold	Too Low	Just Right	Too High
Brightness: 30	Miss dark images	Catch shadows	Reject valid twilight
Cloud: 0.7	Miss cloudy	Balance	Reject snowy/sandy areas
Variance: 100	Miss fog	Catch blur	Reject valid uniform areas

Error Resilience

```
try:
    img = cv2.imread(str(image_path))
    if img is None:
        return False, "Failed to load image"
    ...
except Exception as e:
    return False, f"Quality check error: {str(e)}"
```

Corrupted or unreadable images are gracefully handled.

Usage in Main Pipeline

Initialization

```
# In pipeline.py
self.quality_checker = ImageQualityChecker(
    brightness_low=Config.BRIGHTNESS_THRESHOLD_LOW,    # 30
    brightness_high=Config.BRIGHTNESS_THRESHOLD_HIGH,  # 225
    cloud_threshold=Config.CLOUD_THRESHOLD,            # 0.7
    min_variance=Config.MIN_IMAGE_VARIANCE             # 100
)
```

Per-Sample Quality Check

```
def _process_sample(self, row, ...):
    ...
    image_path = self.maps_client.download_satellite_image(...)

    # Quality assessment
    is_verifiable, quality_reason = self.quality_checker.check_quality(image_path)

    # Detection runs regardless (for comparison)
    all_polygons = self.detector.detect(image_path)

    # Quality status recorded in output
    output_record = {
        ...
        "qc_status": "VERIFIABLE" if is_verifiable else "NOT_VERIFIABLE",
        "image_metadata": {
            ...
            "quality_check": quality_reason
        }
    }
```

Pipeline Flow

```
Download Image
      │
      ▼
┌──────────────────┐
│  Quality Check   │
│  check_quality() │
└────────┬─────────┘
         │
         ├──────── is_verifiable=True ──────► Detection results are reliable
         │
         └──────── is_verifiable=False ─────► Results flagged as "NOT_VERIFIABLE"
                                    (detection still runs for data collection)
```

Why Detection Still Runs

Even for low-quality images, detection provides:

- Baseline data for analysis
- False positive/negative study material
- Training data for future models

The qc_status flag allows downstream filtering.