Development of an Automated Image Classification and Compression System for a Small Satellite

GIDI RABI[1], ROI BRUCHIM[1]
[1]School of Computer Science, Ariel University, Israel.
Corresponding author: Gidi Rabi (e-mail: Gidirabi111@gmail.com). Roi Bruchim (e-mail: Roibr23@gmail.com).

ABSTRACT In this project, we built a full pipeline to automatically analyze and compress satellite images. The main goal was to classify whether a given image is usable or should be rejected based on visual quality, sky visibility, and star clarity. We combined various image processing techniques such as Laplacian sharpness detection, star counting, flicker analysis, and horizon detection to make a decision for each frame. The pipeline was implemented using OpenCV, PyWavelets, and Streamlit for an interactive interface. We tested our solution on a dataset of 175 images of stars and achieved an accuracy of approximately (85%) compared to manual evaluation.

INDEX TERMS Satellite image analysis, image classification, star detection, horizon detection, wavelet compression, computer vision, OpenCV, Streamlit.

I. INTRODUCTION

When working with satellite images, not every picture we get is useful. Some might be blurry, have too much noise, or just not show enough stars or sky. Since there's usually a huge amount of images, it's really hard to check them all manually.

In this project, we built a tool that automatically checks each image and decides if it's good enough to use or if it should be rejected. We used different image processing techniques like checking how sharp the image is, counting stars, detecting the horizon, and checking for issues like flicker or overexposure. We also made sure to reject images that show a large number of bright spots that aren't actual stars- like reflections or sensor noise- which can mislead the system.

We added a compression feature too, which reduces the image size using wavelet transforms while keeping the important visual details. Everything is available in a simple web interface using Streamlit. The user just uploads an image, and the system shows the analysis and lets them download a compressed version if they want. We tested the system on 175 images and compared the results to manual review. It reached about 84.6% accuracy, which shows that it can really help speed up the filtering process and reduce manual work.

-

II. EXPERIMENTAL PREPARATION

To build and test our system, we created a full pipeline that handles both image analysis and compression. The project includes several Python scripts we wrote and connected together to process satellite images from start to finish. Most of the processing is based on OpenCV, and the interface is built using Streamlit.

We started by writing modules that each focus on one part of the problem. The `earth.py` file detects whether the image shows Earth or just the sky. It also separates the sky from the ground using thresholding and morphological filters. This is important because we only want to detect stars in the sky region. The `star_finder.py` script looks for bright circular patterns (stars) inside the sky mask and filters out things that are too small, too big, or not bright enough.

The main script that brings everything together is `Algorithms.py`. It runs a full analysis on each image and checks for different quality indicators: sharpness using the Laplacian variance, noise level based on residuals, number of stars, horizon detection using Sobel and Hough transforms, overexposed pixels, and flickering based on row brightness variation. It then decides whether the image should pass or be rejected based on thresholds we defined.

We also wrote a separate module, `image_compress.py`, that compresses the images using wavelet decomposition. It keeps the low-frequency components and aggressively reduces the high-frequency details, which helps shrink the

file size without losing too much visual quality.

Finally, all of this is wrapped into a user interface using `gui_app.py`. This script lets the user upload an image, see the analysis results, and optionally download a compressed version of the image. All processed images and info files are saved in organized folders for convenience.

-

A. DATASET AND PREPROCESSING

For this project, we used images from the "Stars" folder of the *Space Images Category* dataset available on Kaggle[1]. These images mostly show outer space with visible stars, although some contain noise, motion blur, overexposure, or artifacts that make them less usable.

Before running analysis on the images, we resized them to a fixed resolution of 1280x720 to keep the input consistent across the pipeline. Each image was then converted to grayscale for further processing. To avoid detecting stars over non-sky regions like Earth or bright blobs, we used the mask generated by `earth.py` to isolate just the sky area. This helped reduce false star detections and improved classification reliability.

B. EVALUATION PIPELINE

The main image analysis flow was written in `Algorithms.py`, which acts as the core evaluation script. For every image, it runs a series of checks:

- Calculates image sharpness using Laplacian variance to flag blurry frames.
- Measures noise based on the standard deviation of residuals between the original and median-blurred image.
- Uses the `earth.py` module to separate the sky and earth regions.
- Runs custom star detection from `star_finder.py` on the sky mask.
- Applies Sobel and Hough transforms to check if a clear horizon is present.
- Looks for overexposed pixels (glitches) and detects flicker using row-based brightness analysis.

Based on these checks, the image is either marked as `PASSED` or `REJECTED`. For example, if the image is blurry, has low star count, or shows too much flicker, it's automatically rejected. The results for all images are printed with explanations and can also be saved to a text file.

C. IMPLEMENTATION AND TOOLS

The project was developed in Python using several core libraries. We used `OpenCV` for all the image processing tasks, including masking, blurring, edge detection, and thresholding. The star detection algorithm is based on detecting small bright contours in the sky region, filtered by size and brightness. To compress the images, we used the `PyWavelets` library with the `haar` wavelet, along with `imageio` to export images in WebP format.

The user interface was built using `Streamlit`, which allowed us to easily create a web app where users can upload images, run analysis, and download the results. We also created organized folders like `uploads`, `info`, and `Compressed` to keep all files well managed throughout the process.

[1]https://www.kaggle.com/datasets/abhikalpsrivastava15/space-images-category

III. RESEARCH QUESTIONS AND ANSWERS

A. CAN WE AUTOMATICALLY DETECT WHETHER A SATELLITE IMAGE IS USABLE OR NOT?

Yes. The system uses a combination of computer vision techniques to automatically decide if a satellite image is good (i.e., usable) or should be rejected. It checks multiple factors like blurriness, star count, horizon visibility, noise, overexposure, and flicker. Based on these, it gives a final pass or reject decision. In most cases, it correctly matches what we would decide by looking at the image manually.

B. HOW ACCURATE IS THE SYSTEM COMPARED TO MANUAL REVIEW?

We ran the system on 175 images from the "Stars" folder in the Kaggle space dataset. After comparing the results to manual evaluation, we found that the system made 27 mistakes - meaning it either accepted a bad image or rejected a good one. This gives an overall accuracy of around **84.6%**.

Confusion Matrix (based on manual labels):

Prediction vs. Ground Truth	Actual: Good	Actual: Bad
Predicted: Good	127 (True Positive)	13 (False Positive)
Predicted: Bad	8 (False Negative)	27 (True Negative)

TABLE 1: Confusion matrix of system predictions vs. manual labels (175 images total).

C. WHAT ARE COMMON REASONS FOR WRONG DECISIONS?

Most of the wrong decisions were due to borderline cases. For example:

- Some images had too many bright spots that weren't stars, which confused the star count check.
- A few blurry images still passed because they had high sharpness in noisy areas.
- Some images had strong flicker, but the variation wasn't high enough to cross the rejection threshold.

D. IS THE STAR DETECTION RELIABLE?

Yes - in most cases. The system uses a custom masking method to make sure stars are only detected in the sky region. However, in a few cases where there were bright blobs or noise that looked like stars, the count went up and the image was accepted even though it shouldn't have been. We added a rule that flags very high star counts as "possible noise," but that's still an edge case.

E. DOES THE SYSTEM AVOID OVERFITTING TO ONE SPECIFIC TYPE OF ERROR?

Yes. The system doesn't just rely on one feature. For example, even if an image has enough stars, it will still be rejected if it's blurry or has overexposure. This makes the system more general and avoids focusing too much on one type of issue.

F. HOW USEFUL IS THIS SYSTEM IN REAL SCENARIOS?

It's very useful in situations where you have to go over hundreds of satellite images quickly. It can help filter out bad frames automatically and save a lot of time. Even though it's not perfect, getting over 84% right with no human involvement is a big step, and it can definitely support a human operator.

IV. CONCLUSION

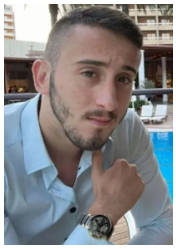
In this project, we built a full pipeline that can analyze and compress satellite images automatically. Our main goal was to filter out low-quality frames and keep only the ones that are usable for space-related tasks, such as star tracking or sky observation. We combined several computer vision techniques, including sharpness detection, noise analysis, star counting, horizon detection, and flicker/glitch checks. The system also included a wavelet-based image compression feature to reduce file size while preserving quality.

We tested the system on 175 images from a public space dataset and compared its results with manual evaluation. Out of all the images, 27 were misclassified, which gave us an overall accuracy of about 84.6%. Most of the errors were from edge cases, like images with high brightness noise or borderline flicker.

ACKNOWLEDGMENT

We would like to thank **Gidi Rabi** and **Roi Bruchim** for their contributions to this research.

GIDI RABI is an undergraduate B.Sc. student in Computer Science at Ariel University, Israel. He developed and implemented machine learning scripts, conducted performance analyses, and contributed to model evaluation. His work included scripting model comparisons, optimizing computational efficiency, and researching relevant literature to support findings.



ROI BRUCHIM is an undergraduate B.Sc. student in Computer Science at Ariel University, Israel. He contributed to implementing classification models, optimizing hyperparameters, and analyzing misclassification patterns. Roi also documented findings, formulated research insights, and sourced academic references to contextualize results.