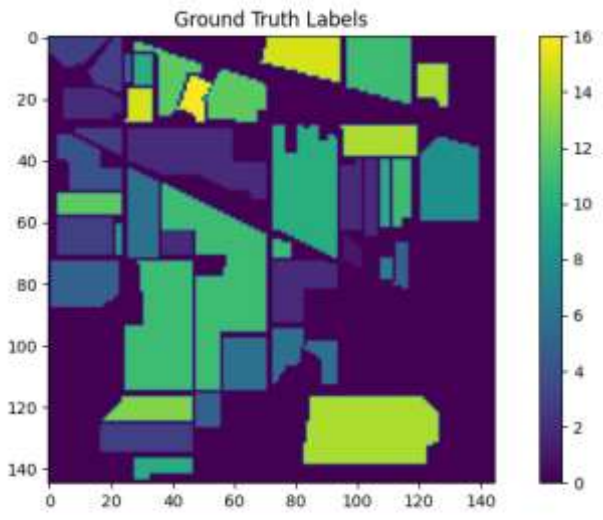
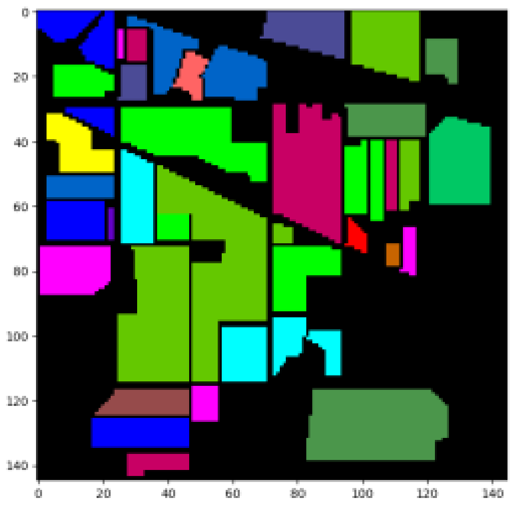
Hyperspectral Image Classification

Enhancing Hyperspectral Image Classification Through Advanced Preprocessing

And Implementing / Tuning Different Models





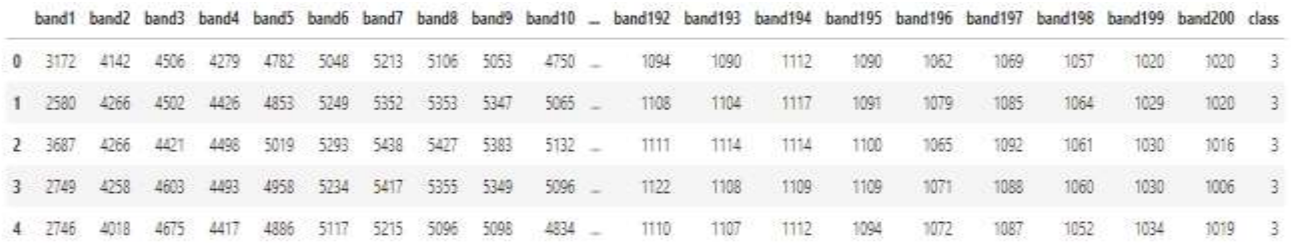
Work done by:

Adam Fendri

1-Understanding the data:

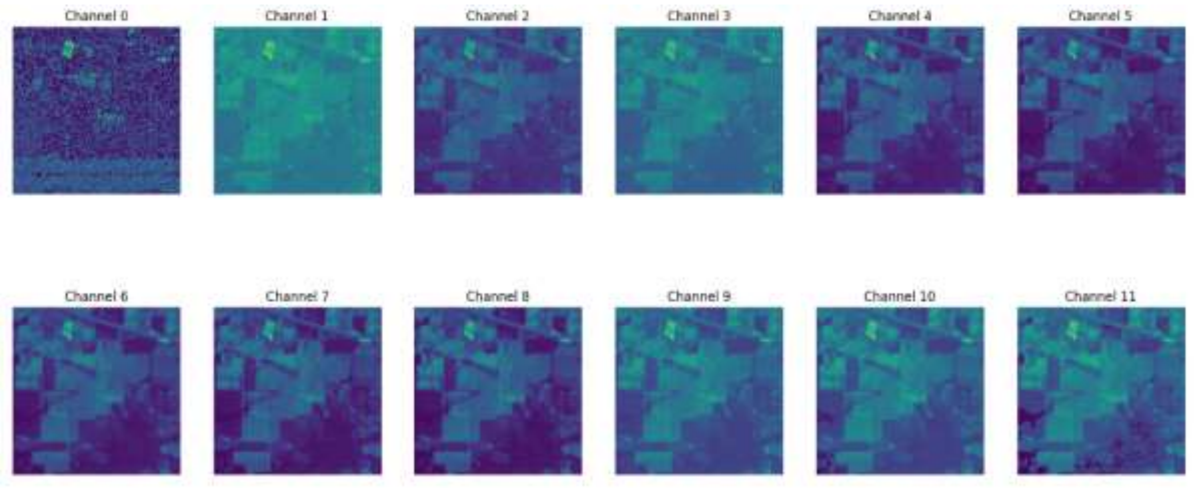
We chose the Indian Pines dataset which is a very famous Hyperspectral image segmentation dataset. The input data consists of hyperspectral bands over a single landscape in Indiana, US, (Indian Pines data set) with 145×145 pixels. For each pixel, the data set contains 200 spectral reflectance bands which represent different portions of the electromagnetic spectrum in the wavelength range 0.4−2.5^10−6.

After we loaded the data, we tried to understand it, so we did some plots:



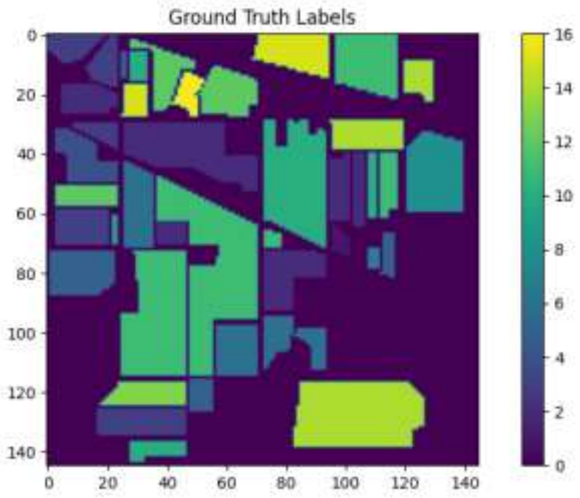
	band1	band2	band3	band4	band5	band6	band7	band8	band9	band10	...	band192	band193	band194	band195	band196	band197	band198	band199	band200	class
0	3172	4142	4506	4279	4782	5048	5213	5106	5053	4750	...	1094	1090	1112	1090	1062	1069	1057	1020	1020	3
1	2580	4266	4502	4426	4853	5249	5352	5353	5347	5065	...	1108	1104	1117	1091	1079	1085	1064	1029	1020	3
2	3687	4266	4421	4498	5019	5293	5438	5427	5383	5132	...	1111	1114	1114	1100	1065	1092	1061	1030	1016	3
3	2749	4258	4603	4493	4958	5234	5417	5355	5349	5096	...	1122	1108	1109	1109	1071	1088	1060	1030	1006	3
4	2746	4018	4675	4417	4886	5117	5215	5096	5098	4834	...	1110	1107	1112	1094	1072	1087	1052	1034	1019	3

This how actually the dataset is presented, each pixel has value for those 200 bands which will be considered as the features afterwards.



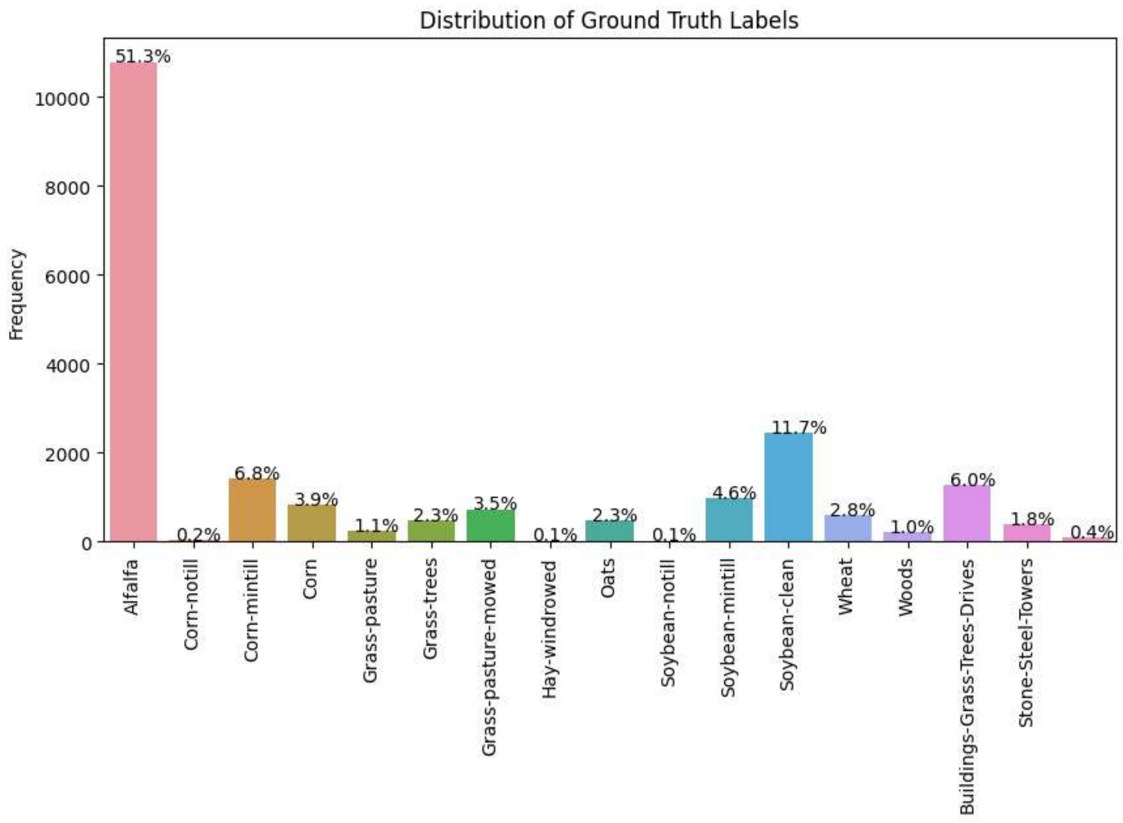
Here we just applied a filter to see each band separately just to get an overall idea.

Once the we display all 200 bands we get the full image like this:

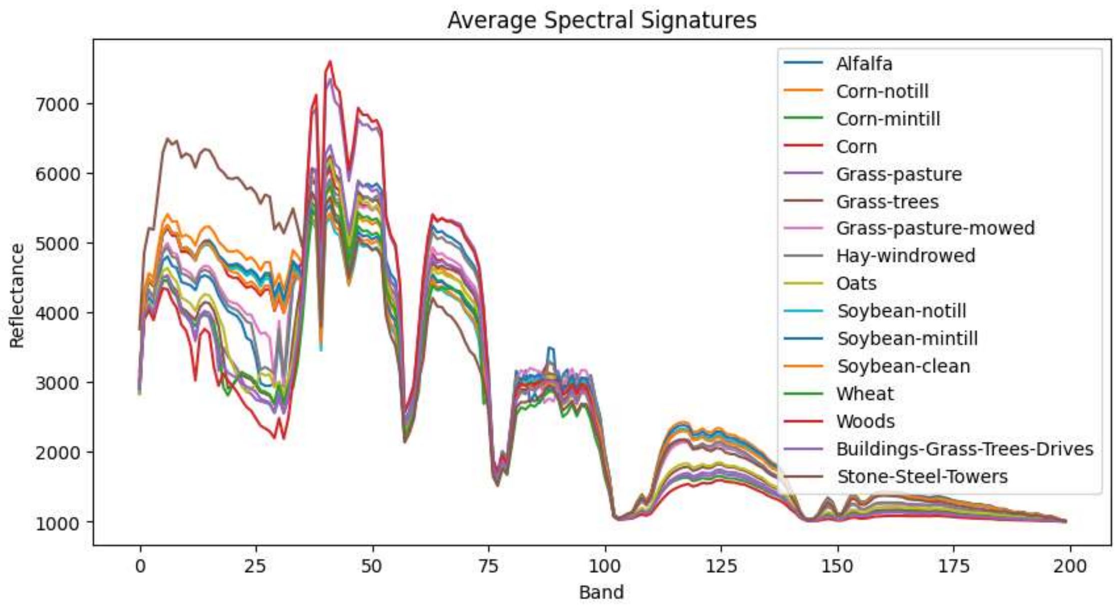


As we can see the dataset has 16 classes which will be considered as our target for our classification.

To understand more about those classes we decided to plot the distribution of those classes compared to frequency(values of actual bands.)



As we can see we have some classes that represent less than 1% in our dataset which hints the use of a complex model for classification afterwords.



Now here we plotted the average spectral signatures of each class compared to each bands which gave us a rough idea about how some bands actually form each class.

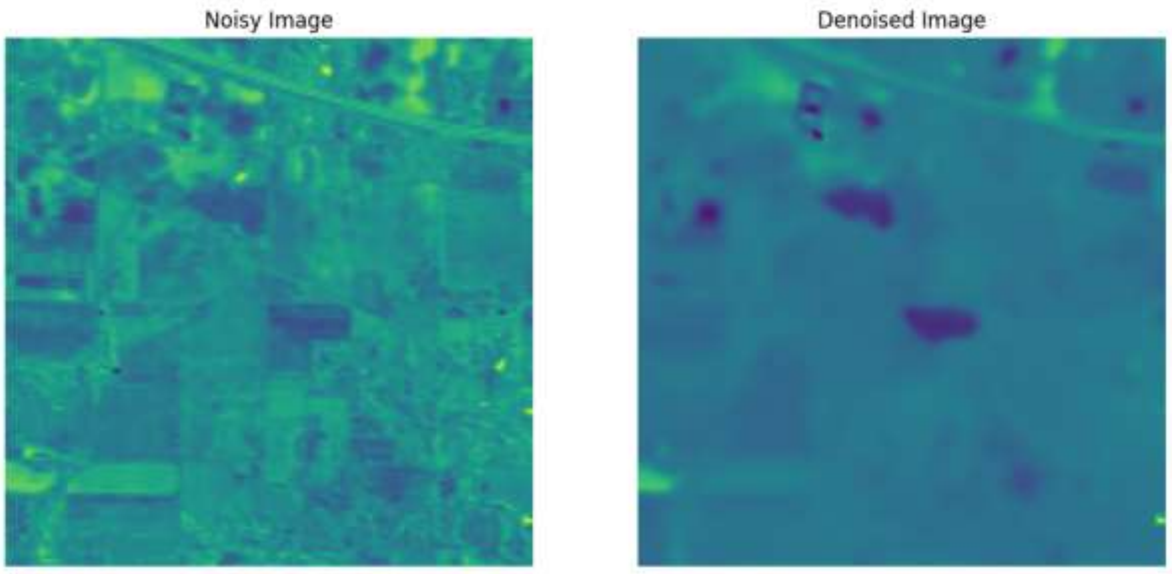
2-Pre-Processing:

2.1: Non Local Mean Denoising Filter:

As we discussed earlier, we discovered that some of our classes are a minority, so to help overcome this problem we thought using a non local mean denoising filter can actually help us differentiate those classes.

For context the NLM filter works by averaging similar patches of pixels in an image to reduce noise. Unlike local filters that only consider nearby pixels, NLM compares patches from different parts of the image.
For each pixel in the image, the filter calculates a weighted average of all pixels in the image, where the weights are determined by the similarity between the patch centred at the pixel of interest and patches centred at all other pixels.

But this was a rookie mistake, our image is actually pretty clear and we could've saved ourselves the hustle if we plotted those minority classes at first and checked them.



As we said above the filter made things worse and we didn't use it.

2.2: Dimension Reduction:

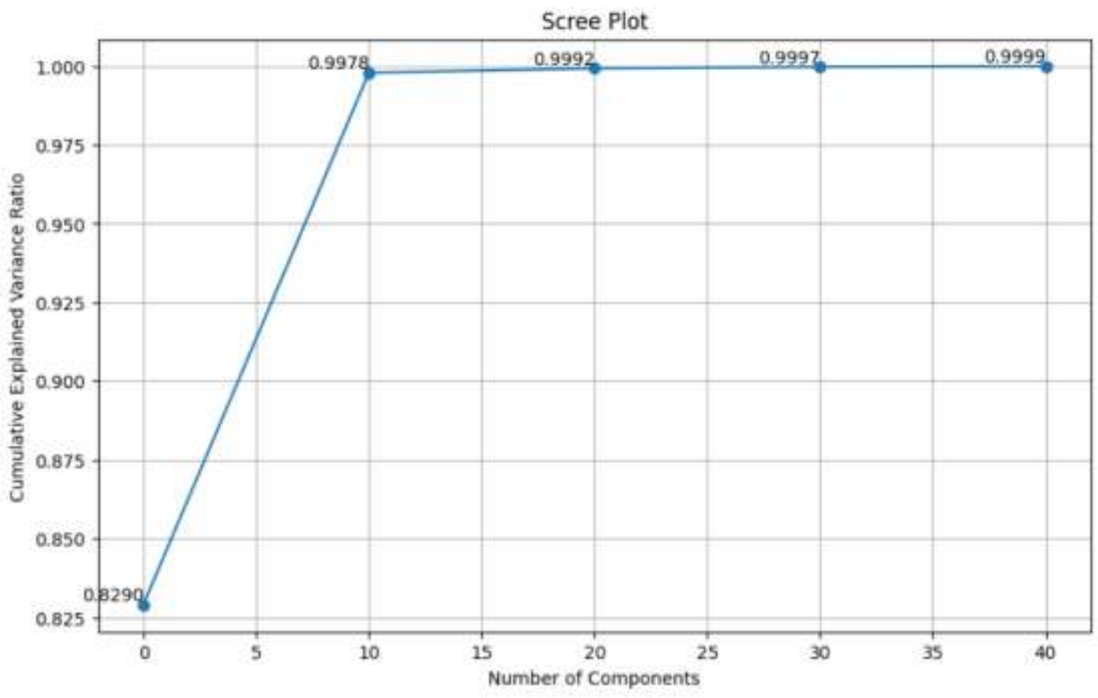
When we plotted our data to understand it, we realised that we have 200 bands which is quite complex for most Machine/Deep Learning Models, so

To fix this problem we decided to use dimension reduction algorithms that we learned last year in our statistics course:

Principal Component Analysis (PCA) is a widely used dimensionality reduction technique that aims to transform high-dimensional data into a lower-dimensional space while preserving most of the important information. The core idea behind PCA is to find a new set of variables (called principal components) that are linear combinations of the original variables and capture the maximum variance in the data.

Factor Analysis (FA) is a statistical method used to identify underlying latent variables (factors) that explain patterns of correlations among observed variables. It is commonly used in fields such as psychology, sociology, economics, and market research to understand the structure of complex data sets and reduce the dimensionality of data.

These are the 2 algorithms we tried to experiment with, we ended up choosing PCA since it gave a slightly better result compared to FA.



We used a scee plot to determine the number of components witjout loosing too much information. As seen in the plot 30 was the sweet spot for PCA.

2.3: Dividing the image:

Now our dataset is actually a very big matrix, even with the reduction of the bands it's actually still pretty complex for our models, so we implemented a function that divides the images into smaller patches so the models coul d process the data better. To summarize how the function work we give it t he size of our patch and then it iterates through the image and divide it by t he size of the patch to create equal size patches, this happens by reshapin g the matrix so we end up with a matrix of (10249, 25, 25, 30) instead of (145,145,30).

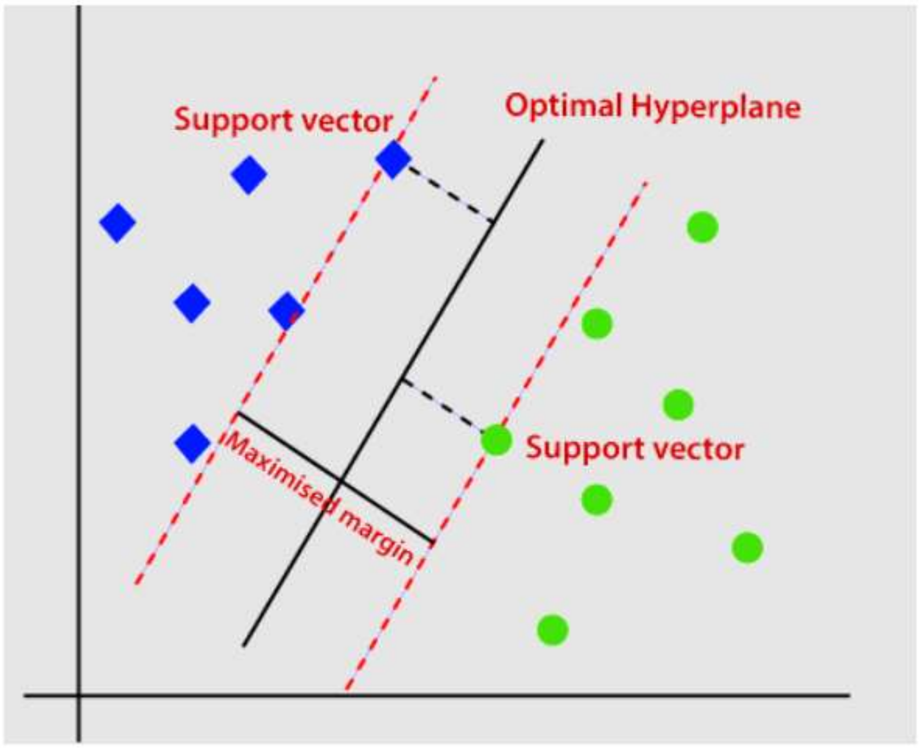
3-Modeling and Validation:

Now as we saw our problem here is very different from other projects, so we can't use any ML/DL Model. We did some research and found out that an SVM Model could be the appropriate one for our task:

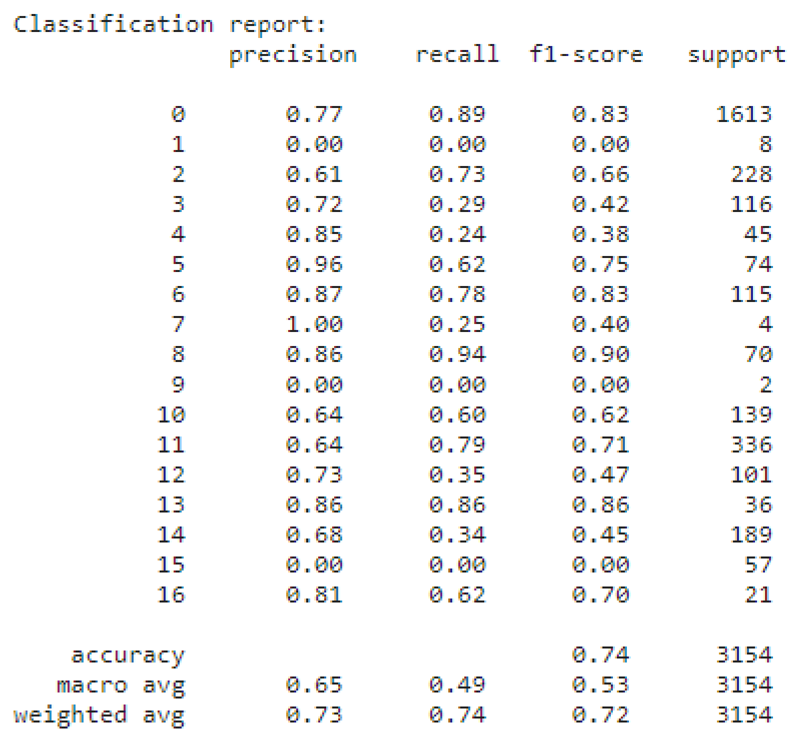
2.1: SVM Model:

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It's particularly effective in high-dimensional spaces and is widely used for classification tasks in various domains, including image recognition, text classification, and bioinformatics.

Key Concepts:
1-Decision Boundary: In SVM, the goal is to find the optimal hyperplane that best separates the classes in the feature space. This hyperplane is called the decision boundary.
2-Support Vectors: These are the data points that are closest to the decision boundary. They play a crucial role in defining the decision boundary.
3-Margin: The margin is the distance between the decision boundary and the nearest data point from each class. SVM aims to maximize this margin, as it leads to better generalization and improves the model's ability to classify unseen data.



At first we used the default values and we got these results:



```
Classification report:
              precision    recall  f1-score   support

           0       0.77      0.89      0.83      1613
           1       0.00      0.00      0.00         8
           2       0.61      0.73      0.66       228
           3       0.72      0.29      0.42       116
           4       0.85      0.24      0.38        45
           5       0.96      0.62      0.75        74
           6       0.87      0.78      0.83       115
           7       1.00      0.25      0.40         4
           8       0.86      0.94      0.90        70
           9       0.00      0.00      0.00         2
          10       0.64      0.60      0.62       139
          11       0.64      0.79      0.71       336
          12       0.73      0.35      0.47       101
          13       0.86      0.86      0.86        36
          14       0.68      0.34      0.45       189
          15       0.00      0.00      0.00        57
          16       0.81      0.62      0.70        21

    accuracy                           0.74      3154
   macro avg       0.65      0.49      0.53      3154
weighted avg       0.73      0.74      0.72      3154
```

For default values, an accuracy of 74% isn't that bad but not enough, so we decided to tune the parameters of our model which are:

1- **kernel:** This parameter specifies the type of kernel used in the SVM algorithm. The kernel function computes the inner product between two points in a higher-dimensional space, allowing the SVM to operate in that higher-dimensional space without explicitly calculating the coordinates of the data points. Common kernel types include:

'linear': Linear kernel, which creates linear decision boundaries.

'rbf' (Radial Basis Function): Gaussian kernel, which is commonly used and can handle non-linear decision boundaries.

'poly': Polynomial kernel, which uses a polynomial function to create non-linear decision boundaries.

'sigmoid': Sigmoid kernel, which uses a hyperbolic tangent function.

2- degree: This parameter is specific to the polynomial kernel (kernel='poly'). It defines the degree of the polynomial kernel function. For example, if degree=3, the polynomial kernel function used is (1 + gamma · ⟨X ,X'⟩)3

3- gamma: This parameter defines the kernel coefficient for 'rbf', 'poly', and 'sigmoid' kernels. Intuitively, a higher value of gamma defines how far the influence of a single training example reaches. It can be seen as the inverse of the radius of influence of samples selected by the model. Typical values for gamma are 'scale' (default), 'auto', or a float value.

4- cache_size: This parameter specifies the size of the kernel cache in MB. The kernel cache is used to store the kernel matrix, which can speed up the training process. Larger values can lead to faster execution, but may also consume more memory. In the example, 1024*7 specifies a cache size of 7 GB.

For tuning our parameters we used a grid search function:

Grid search is a hyperparameter tuning technique used to find the optimal combination of hyperparameters for a machine learning model. It systematically searches through a predefined grid of hyperparameter values and evaluates the model's performance using cross-validation.

Here's a general overview of how grid search works:

1-Define Hyperparameter Grid: You specify a grid of hyperparameters and their corresponding values to search over. Each hyperparameter can take on multiple values, creating a multidimensional grid.

2-Split Data: The dataset is divided into training and validation sets. The training set is used to train the models, while the validation set is used to evaluate their performance.

3-Select Performance Metric: You choose an appropriate performance metric to evaluate the models. This could be accuracy, precision, recall, F1-score, or any other metric relevant to your specific problem.

4-Perform Grid Search: The grid search algorithm iterates through every combination of hyperparameters in the grid. For each combination, it trains a new model on the training data and evaluates its performance on the validation set using the chosen performance metric.

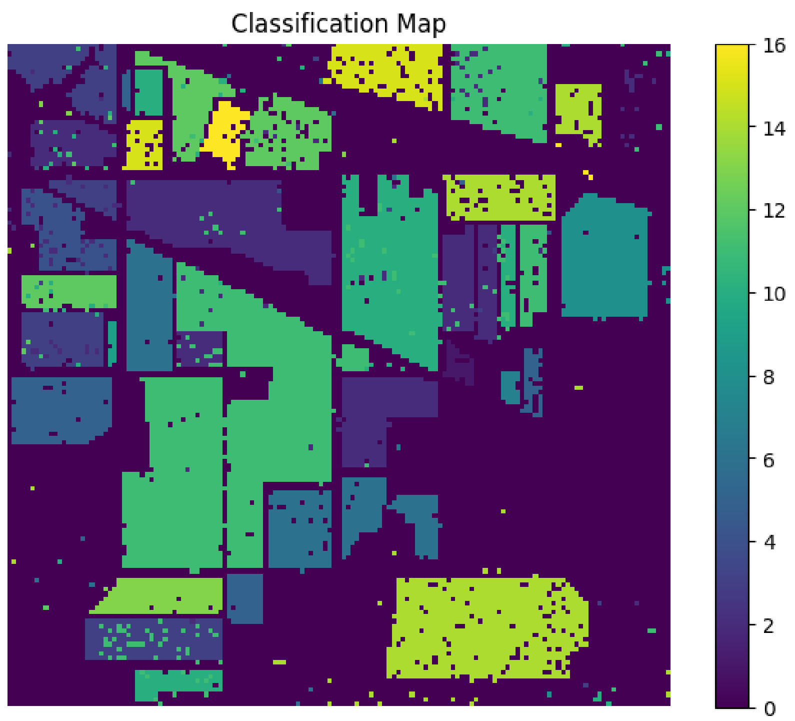
4-Select Best Model: After evaluating all combinations, the grid search identifies the combination of hyperparameters that maximizes (or minimizes, depending on the metric) the chosen performance metric on the validation set. This combination is considered the best model configuration.

5-Evaluate on Test Set: Finally, the performance of the best model configuration is evaluated on an independent test set to obtain an unbiased estimate of its generalization performance.

By exhaustively searching through a predefined grid of hyperparameters, grid search helps you identify the optimal configuration for your model, ultimately leading to better performance and generalization. However, it can be computationally expensive, especially when dealing with large datasets or complex models with many hyperparameters. Nonetheless, it remains a widely used technique for hyperparameter tuning due to its simplicity and effectiveness.

Now after using the grid search function and defining the range for the various parameters our accuracy jumped to 84% from 74%.

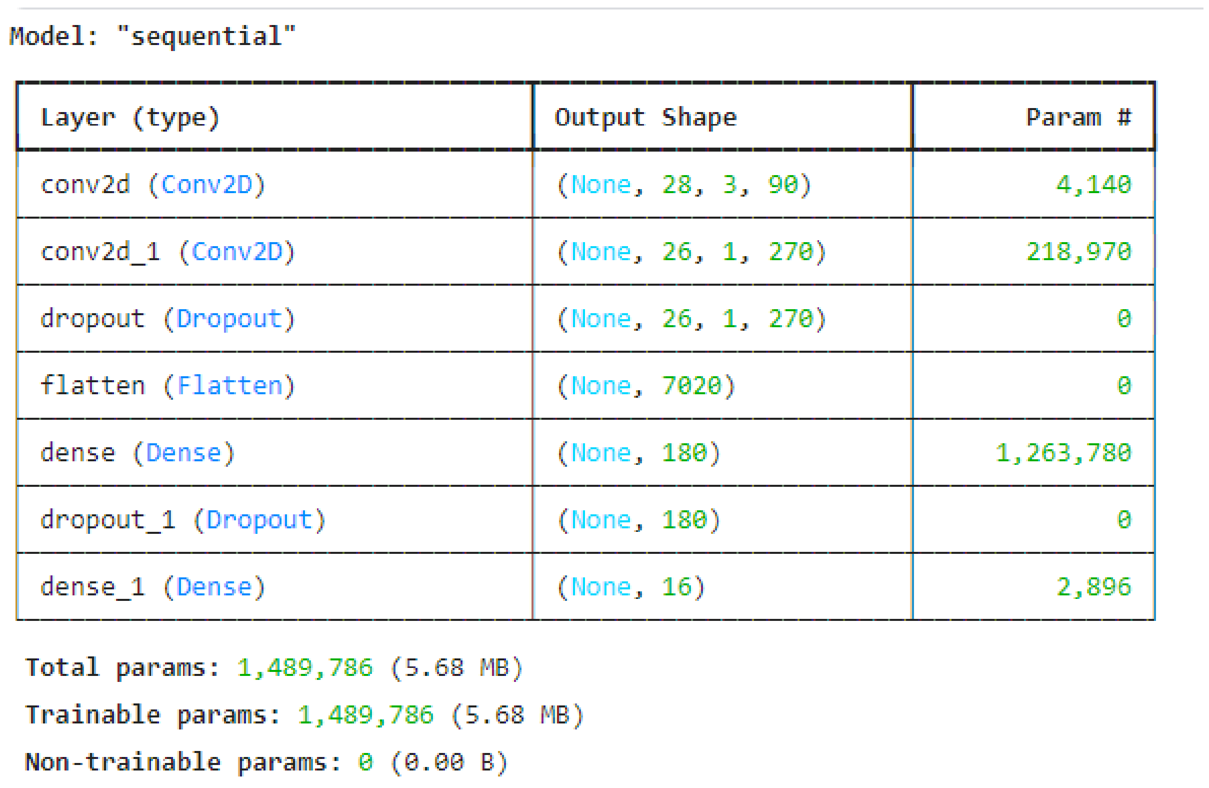
Here's the predicted output by our model:



That’s great but we wanted to experiment more and create our proper Deep Learning model since this task is quite complex. After some research 2D CNN models could work for classifying hyperspectral images.

2.2: 2D CNN Model:

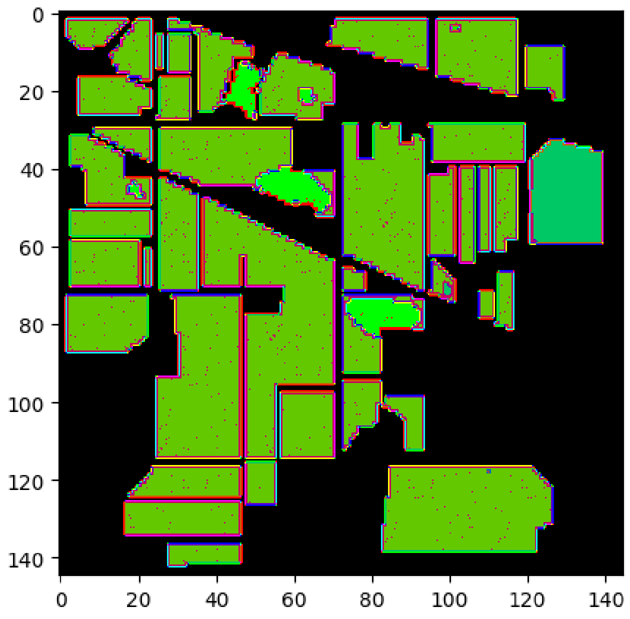
Now our image is actually 3D so we modified our patch function to reshape the image into 2D by choosing a patch size that can work with our 2D CNN Model. After a lot of thinking we decided to make this architecture since it respects an input of a 2D image and it’s quite simple not complex come up with for students who are creating an architecture for the first time:



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                  │ (None, 28, 3, 90)      │         4,140 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_1 (Conv2D)                │ (None, 26, 1, 270)     │       218,970 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)                │ (None, 26, 1, 270)     │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)                │ (None, 7020)           │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                    │ (None, 180)            │     1,263,780 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)              │ (None, 180)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                  │ (None, 16)             │         2,896 │
└──────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 1,489,786 (5.68 MB)
 Trainable params: 1,489,786 (5.68 MB)
 Non-trainable params: 0 (0.00 B)
```

This is a summary of our architecture we have 2 2d convolution layers, followed by a dropout to eliminate overfitting, 1 flatten layer to make the output 1D, then a dense layer(fully connected) followed by a dropout layer and finally the output layer which matches the size of our desired output(16 classes in our case)
We used at first some random parameters and we suffered from underfitting which was quite expected.

Here’s the predicted output:



As we can see it only predicted 4 correct classes out of the 16 with a lot of noise. It's obvious that the problem is because of our bad pramaters, so we modified the grid search function to take the patch size in consideration of the patch function and the different parameters of our model like the learning rate, momentum, optimizer and loss types.

We were able to boost the accuracy from a very bad 69% to 92%

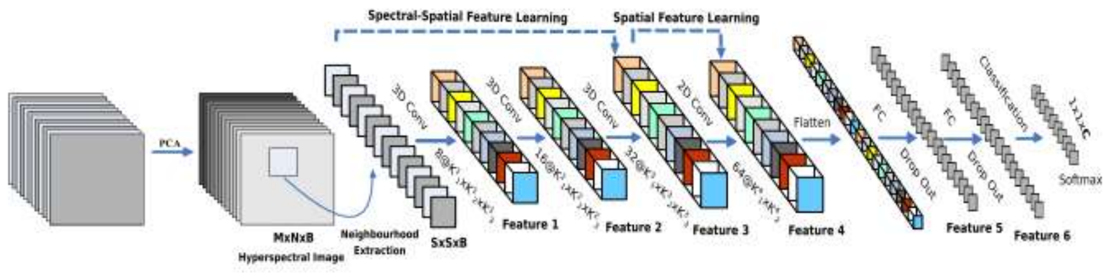
Now this wasn't enough for us so rather than experimenting with the model architecture itself, we did some research and found out that actually this was expected since 2D CNN models aren't quite complex to handle 3D images but 3D CNN on the other hand take way to much resources and time to run properly, 2 things we don't have actually.

Enter HybridSN models, a model combining a 3D into a 2D architecture.

2.3: HybridSN Model:

Hybrid Spectral-Spatial Convolutional Neural Network (HybridSN) for Hyperspectral Image Classification The HybridSN model is a deep learning architecture designed for hyperspectral image classification. It leverages the strengths of both 3D and 2D convolutional neural networks (CNNs) to capture both spectral and spatial features from hyperspectral data.

We found a very famous research paper that tackled the problem and found that the best architecture for this model is something like this, we tried to studied and understand it and we ended up implementing from 0.



Here's a breakdown of the HybridSN model:

1. 3D Spectral Convolutional Layer:
This layer processes the input hyperspectral image, which typically has a 3D structure with dimensions of (height, width, channels). The channels represent different spectral bands capturing information at various wavelengths. The 3D convolution filters learn to extract spatial and spectral features jointly.

2. Spatial Pooling:
A pooling layer like MaxPooling is often used after the 3D convolution to reduce the dimensionality and introduce some level of invariance to small spatial shifts.

3. 2D Spatial Convolutional Layers:
One or more 2D convolutional layers are stacked on top of the processed feature maps from the 3D layer. These layers focus on extracting more complex spatial features from the data.

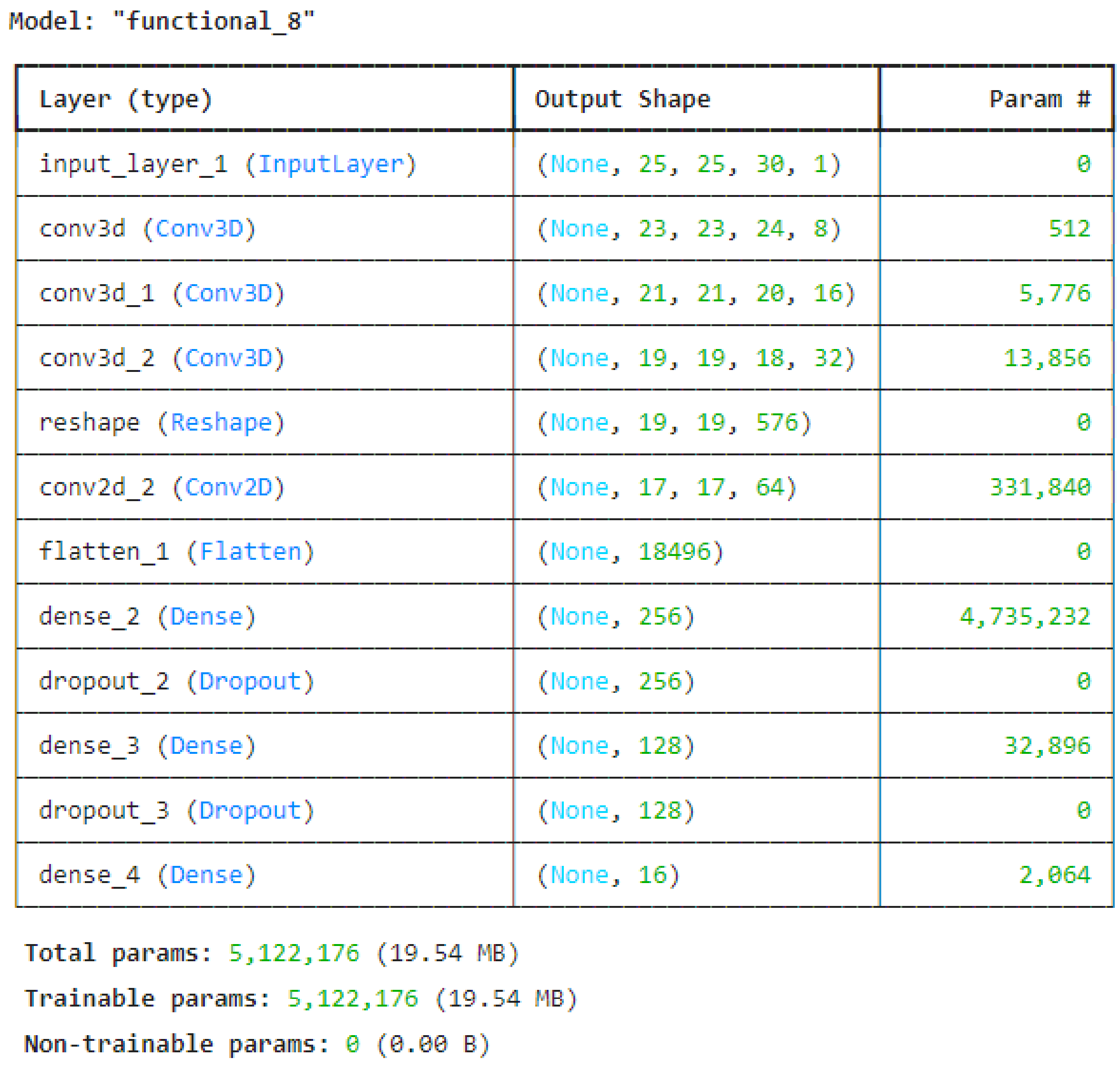
4. Flatten Layer:
The output from the final 2D convolutional layer is flattened into a 1D vector.

5. Dense Layers:!download.png One or more fully connected (dense) layers are used to learn higher-level features and perform classification. The final dense layer typically has a softmax activation function to predict the probability distribution over the different classes. Benefits of HybridSN:

Joint Feature Extraction: Captures both spectral and spatial information simultaneously through 3D convolutions.

Improved Classification Performance: Can achieve better accuracy compared to models relying solely on 2D convolutions.

Reduced Model Complexity: Utilizes 2D convolutions in later stages for efficiency compared to using only 3D convolutions throughout.



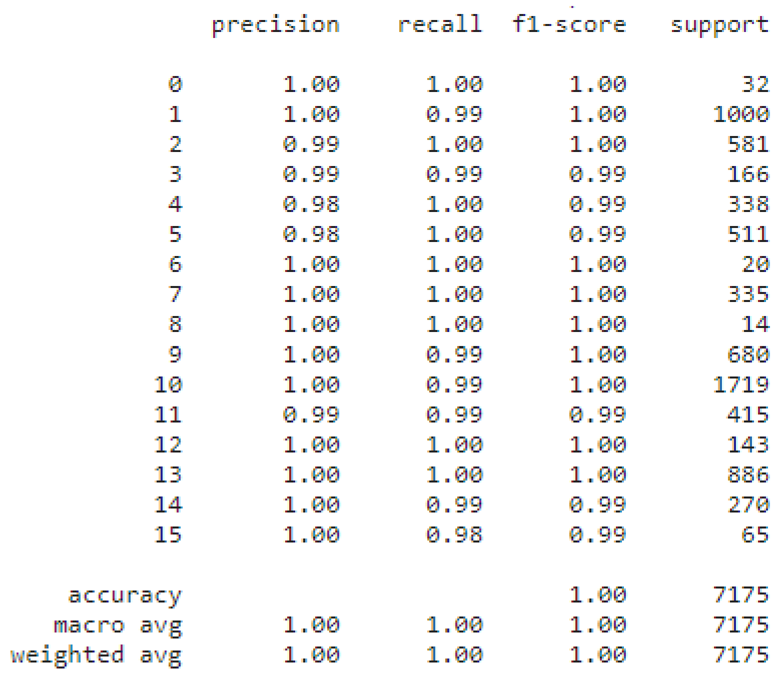
```
Model: "functional_8"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_1 (InputLayer)       │ (None, 25, 25, 30, 1)  │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ conv3d (Conv3D)                  │ (None, 23, 23, 24, 8)  │           512 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ conv3d_1 (Conv3D)                │ (None, 21, 21, 20, 16) │         5,776 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ conv3d_2 (Conv3D)                │ (None, 19, 19, 18, 32) │        13,856 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ reshape (Reshape)                │ (None, 19, 19, 576)    │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_2 (Conv2D)                │ (None, 17, 17, 64)     │       331,840 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_1 (Flatten)              │ (None, 18496)          │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                  │ (None, 256)            │     4,735,232 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_2 (Dropout)              │ (None, 256)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                  │ (None, 128)            │        32,896 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_3 (Dropout)              │ (None, 128)            │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                  │ (None, 16)             │         2,064 │
└──────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 5,122,176 (19.54 MB)
 Trainable params: 5,122,176 (19.54 MB)
 Non-trainable params: 0 (0.00 B)
```

(the implemented architecture)

After understanding the model architecture and implementing it, we made sure this time to use some logical parameters for our model and we ended up with some very impressive result from the first time, we almost matched the paper result without any tuning this time!



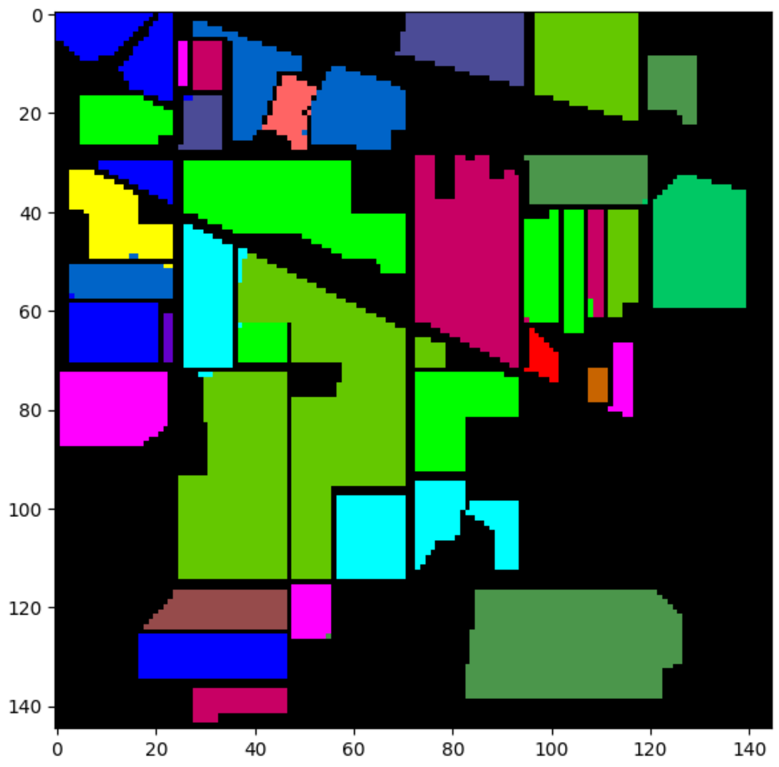
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        32
           1       1.00      0.99      1.00      1000
           2       0.99      1.00      1.00       581
           3       0.99      0.99      0.99       166
           4       0.98      1.00      0.99       338
           5       0.98      1.00      0.99       511
           6       1.00      1.00      1.00        20
           7       1.00      1.00      1.00       335
           8       1.00      1.00      1.00        14
           9       1.00      0.99      1.00       680
          10       1.00      0.99      1.00      1719
          11       0.99      0.99      0.99       415
          12       1.00      1.00      1.00       143
          13       1.00      1.00      1.00       886
          14       1.00      0.99      0.99       270
          15       1.00      0.98      0.99        65

    accuracy                           1.00      7175
   macro avg       1.00      1.00      1.00      7175
weighted avg       1.00      1.00      1.00      7175
```

As we can see we have an accuracy of 1.0 and a very good result for F1 score, for context The F1-score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall. It is useful when the classes are imbalanced because it takes into account both false positives and false negatives.

Here's the predicted Image:


0
20
40
60
80
100
120
140
0
20
40
60
80
100
120
140