Assignment 1

Christiaan Slabbert 21810354@sun.ac.za
Hendrik Raubenheimer 23919469@sun.ac.za
Pieter Colesky 23584955@sun.ac.za
Kilan Van Loo 27832708@sun.ac.za

I. Belief Propagation

For Part A we were tasked to implement a BPSK (Binary Phase Shift Keying) modulation scheme on an AWGN (Additive White Gaussian Noise communication channel). Using the implementation we needed to plot a simulated BER curve associated with an uncoded BPSK AWGN channel. The theoretical BER curve must also be plotted for an uncoded BPSK channel. Thereafter a Hamming (7,4,3) code should be added to the channel and decoded using belief propagation. This implementation should also be used to plot a BER curve of a coded BPSK AWGN channel.

Binary Phase Shift Keying is a modulation scheme where specific phases represent a 0 or a 1. The two phases normally have a 0-degree and 180-degree phase shift. This method of representing binary states makes it easy to transmit data over an analog channel. This is a very important procedure in digital communication systems.

Communication between a platform (satellite) and a ground station is done in this way. During the signal transmission noise is added and bit flips may occur. This gave the need for an error correction code where parity bits are added to the message to make it more robust against noise-induced errors during transmission.

A. Uncoded BPSK (Binary Phase Shift Keying)

Implementation of Uncoded BPSK is done as follows.

Firstly a a random bit stream was generated with a specific length of N. This bit stream consisted of ones and zeros. The bit stream represents the binary values that digital computers work with. The bit stream is represented as follows.

$$A_n = 10100... \tag{1}$$

This bit stream is then converted to a stream of minus ones and ones, where a zero becomes a minus one and a one, stays a one. This is done because normally it is multiplied with a constant frequency carrier wave. This, minus one, and one will incur phase shifts in order to represent the binary values in analog space. The bit stream is converted by the following equation. In the case of our implementation, only the minus one and one stream will be used.

$$B_n = 2A_n - 1 \tag{2}$$

Normalized signal-to-noise ratio (SNR) values are generated by a range of 300 equally spaced values between 0 and 11. Using these SNR values a standard deviation can be calculated by the following equations.

$$\frac{E_b}{N_0} = 10^{\frac{SNR}{10}} \tag{3}$$

$$\sigma^2 = \left(2R_c\frac{E_b}{N_0}\right)^{-1} \tag{4}$$

where R_c is calculated by.

$$R_c = \frac{k}{n} \tag{5}$$

where n is the code length and k is the message length.

Once the standard deviation of a specific SNR value is calculated noise can be added to the stream. Noise is calculated from a Normal Gaussian distribution with a mean of zero and the standard deviation that was calculated. A random value from the distribution is then added to each minus one or one in the stream.

When the noise is added, meaning the stream of values that is at the receiver, it should be taken out again. Noise is taken out of the stream by setting a negative value to minus one and a positive value to a one. The stream of minus ones and ones are then converted to zeros and ones, by setting a minus one to zero and a one stays a one.

Once the received bit stream is determined, it can be compared to the sent bit stream. Counting bit flips will then be used to calculate the probability of a bit flip for a specific SNR value. The probability of a bit flip is calculated by the following equation.

$$P_e = \frac{\#errors}{\#bits} \tag{6}$$

B. Theoretical Uncoded BPSK (Binary Phase Shift Keying)

The theoretical Uncoded BPSK is calculated as follows. Firstly a range of SNR values should be generated. These SNR values is then used to calculate the probability of a bit flip for a specific SNR value by the following equation.

$$P_e = \frac{1}{2}erfc\left(\sqrt{\frac{E_b}{N_0}}\right) \tag{7}$$

C. Coded BPSK (Binary Phase Shift Keying)

For Coded BPSK the bit stream will be encoded with Hamming (7,4,3). For every set of four message bits, three parity bits are added. Once the bit stream is encoded and sent over the communication stream, the noise will be added to the received stream. Soft-decision decoding Belief Propagation algorithm can be used to correct bit flips that may have occurred.

1) Encoding bit stream by Hamming (7,4,3): Firstly the bit stream was divided up into four bit chunks. Also, the following generator matrix was used in order to represent a Tanner graph used to compute the parity bits.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \tag{8}$$

If $a = \begin{bmatrix} 1011 \end{bmatrix}$ is our code word, it can be encoded using the following equation.

$$x = a \cdot G \tag{9}$$

where x is our encoded bits

Using this method, the whole bit stream can be encoded.

2) Soft-decision decoding Belief Propagation algorithm: Once the bit stream is received, noise has been added. Splitting the bitstream up into chunks of seven bits, 4 bits for the message bits, and 3 bits for the parity bits, belief propagation can be done in order to fix bit flips.

Firstly for a seven-bit chunk, the log-likelihood for each soft value needs to be calculated. Log-likelihood is calculated with the following equation.

$$\Lambda(m_n) = \frac{-4r_i}{2\sigma^2} \tag{10}$$

where r_i is the soft value and $\Lambda(m_n)$ is the log-likelihood for a given soft value r_i.

Using the above equation we can calculate the log-likelihood for each soft value and represent it as the following.

$$\mathbf{r} = (r_1, r_2, ..., r_7) \tag{11}$$

Belief Propagation has the following steps.

- Initialize Code Nodes
- Update Check Node
- Update Code Node
- Final Update

For the Initialization step, the code nodes will be initialized with the log-likelihoods that have been calculated. The values are then propagated down each edge connected to the specific code node.

The initialized values are then used to update the check nodes with the following equation.

$$\Lambda(m_o) = 2tanh^{-1}\left[\prod_{i\neq 0} tanh\left(\frac{\Lambda(m_i)}{2}\right)\right] \tag{12}$$

Using the updated check nodes, the code nodes need to be updated with the following equation.

$$\Lambda(m_o) = \sum_{i\neq o} \Lambda(m_i) + \Lambda(m_n) \tag{13}$$

Doing the check node and code node update multiple times, in this case, ten times would suffice, the final update can be done with the following equation.

$$\Lambda(m_f) = \sum_{i} \Lambda(m_i) + \Lambda(m_n) \tag{14}$$

Using the values from the final update, the decoded codeword y can be obtained. If the value is negative, y_i is set to one and when the value is positive, y_i is set to zero. Extracting the message bits from y and concatenating every chunk's message bits, can be compared to the sent bit stream. Getting a bit flip probability for the specific SNR value with Equation 6.

D. Final Result

Simulating all these algorithms and plotting them, will give the following result.

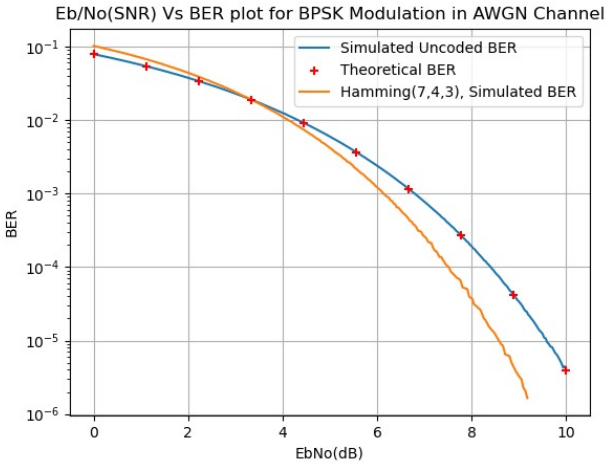


Fig. 1

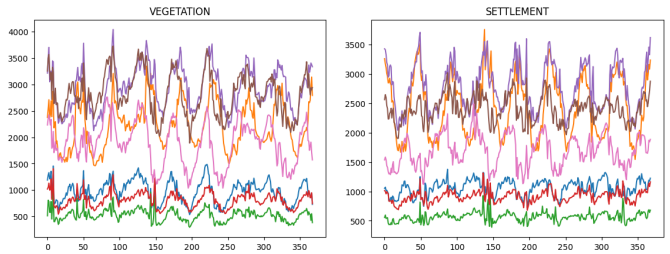
II. Machine Learning

A. Dataset

The dataset used for this part of the assignment includes land surface data for 925 pixels sampled from the Gauteng province in South Africa. The land surface data was collected from the MODIS MCD43A4 BRDF (Bidirectional Reflectance Distribution Function) corrected dataset, which has a spatial resolution of 500 meters and a temporal resolution of 45 observations a year (one every 8 days).

The pixels in this dataset belong to either one of two land cover classes, namely vegetation and settlement. The dataset consists of 333 settlement pixels and 592 vegetation pixels. Each pixel consist of eight time-series containing 368 samples. The eight time-series correspond to the first seven MODIS bands and the Normalized Difference Vegetation Index (NDVI).

Figure 2 plots the time-series for a random MODIS vegetation and settlement pixel.



(a) Vegetation pixel (b) Settlement pixel

Fig. 2: Random time-series plots

B. Feature extraction

Before training a classifier, we need features. We will use the means and the seasonal amplitudes of each pixel as features as suggested by paper [1]. These features correspond to the two largest harmonic components associated with the time-series of each pixel, which can be extracted for each band via the Fast Fourier Transform (FFT):

$$[\mathbf{Y}_2]_{ij} = \begin{cases} |\mathcal{F}[x_i(t)][0]|, & j=1 \\ |2\mathcal{F}[x_i(t)][f_s]|, & j=2 \end{cases} \quad (15)$$

Where:

$x_i(t)$: denotes the time series of MODIS pixel i for band b

$\mathcal{F}$: denotes the Fourier Transform

$f_s : \frac{1}{45}\text{Hz}$

We apply numpy.fft.fft to the MODIS data and extract the features using Equation 15. Additionally we use sklearn.preprocessing.MinMaxScaler to scale each feature between 0 and 1[1]. This results in the reduced feature space obtained in figure 3, which shows the FFT feature vectors per band.

[1]This significantly simplified plotting decision boundaries and did not negatively affect the classifier accuracies.

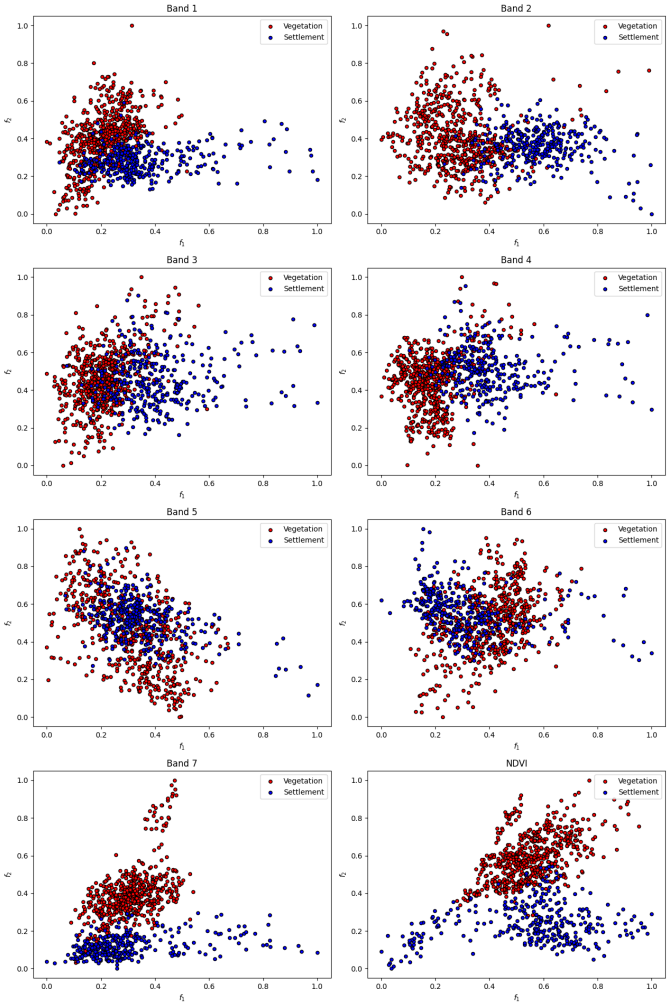


Fig. 3: FFT features

The mean of each pixel is associated with the x-axis (f_1), while the seasonal amplitude of each pixel is associated with the y-axis (f_2). We can see that the two land cover classes are the most separable for band 7 and NDVI, the least separable for band 5 and 6 and somewhat separable for bands 1 to 4.

C. Classification Procedure

1) Logistic Regression and Naive Bayes: We randomly split our obtained features into two equal parts: 50% for training and 50% for testing according to a chosen random seed using sklearn.model_selection.train_test_split. To ensure a stratified train/test split, we pass the class labels to the `stratify` parameter of `train_test_split` as well. This ensures that the class distributions are the same for both the training and test sets. Figure 4 shows the train/test set splits for all the bands.

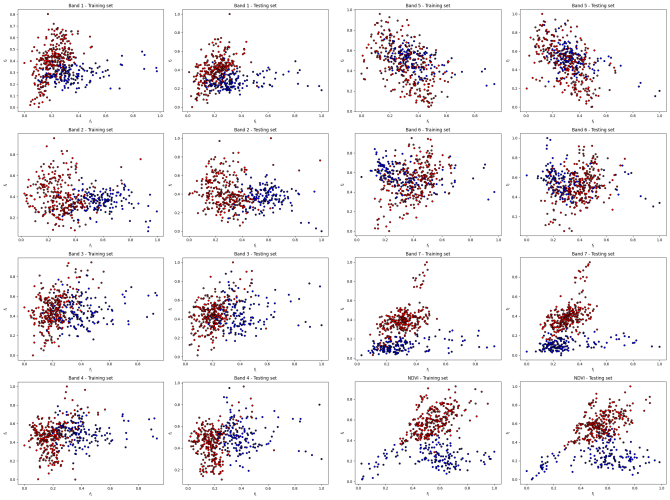


Fig. 4: Train/test sets

Next we use sklearn.model_selection.StratifiedKFold to randomly split the training set for each band into 10 different training and validation sets. For each fold about 90% is used for training and 10% is used for validation. We pass `n_splits=10`, `shuffle=True`[2] and `random_state` equal to our chosen random seed. Once again the folds are stratified by preserving the percentage of samples for each class. Figure 5 shows the 10 different training and validation sets for band 1.

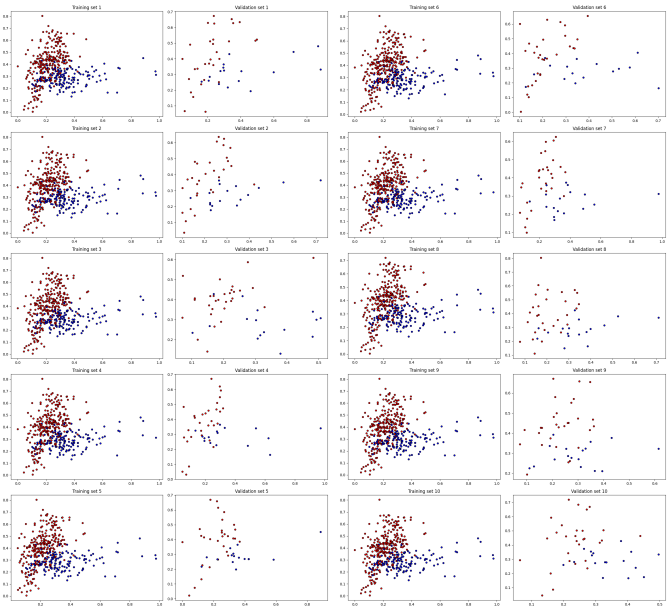


Fig. 5: K-fold splits for band 1

Next we fine tune the hyper-parameters of logistic regression and naive bayes classifiers for each band, using our different validation sets. For logistic regression we use sklearn.linear_model.LogisticRegression and fine tune the `C`[3] parameter. We train 3 separate logistic regression classifiers using different solvers, namely `lbfgs` (default), `newton-cholesky`[4] and `liblinear`[5]. For naive bayes we use sklearn.naive_bayes.GaussianNB and fine tune the `var_smoothing`[6] parameter. For fine-tuning hyper-parameters we use sklearn.model_selection.GridSearchCV and pass the following arguments:

- `estimator` equal to an instance of a `SKLEARN` classifier.
- `param_grid` equal to a dictionary containing an experimental range of hyper-parameter values:
  - **Logistic Regression**:
    `{"C": np.logspace(-1,3,1000)}`
  - **Naive Bayes**:
    `{"var_smoothing": np.logspace(0,-3,1000)}`
- `scoring="accuracy"`
- `n_jobs=-1`[7]
- `cv` equal to our `StratifiedKFold` instance.
- `return_train_score=True`[8]

For each classifier, we create a `GridSearchCV` instance for each band and call `fit` on the training set of the band. This will do the following for each of the hyper-parameter values in `param_grid` for each of the train/validation splits of the current band:

1) Call `fit` on the training set.
2) Call `predict` on the validation set and log the accuracy.

`GridSearchCV` then retrains the classifier on the entire training set of the current band using the hyper-parameter value that resulted in the highest average validation accuracy. We then evaluate the performance of the fine-tuned classifier on the hold-out test set of the current band.

2) Time-varying sequential model and Sequential Probability Ratio Test: The data is grouped together again and split into a training set containing 50% of the data and a test set containing the other 50% of the data. The data is divided in such a way that all data with respect to individual pixels belong to either the training or test set so that a time-varying sequential model as described in paper [2] and the Sequential Probability Ratio Test (SPRT) test can be applied to the test data. A random seed was chosen and numpy.random.choice was used to allocate 50% of all vegetation pixels to the training set and the rest to the test set, and 50% of all settlement pixels to the training set and the rest to the test set.

[2]To shuffle each class's samples before splitting into batches.

[3]Inverse of regularization strength; must be a positive float. Smaller values specify stronger regularization.

[4]Good choice for $n_{samples} > n_{features}$, especially with one-hot encoded categorical features with rare categories.

[5]Good choice for smaller datasets

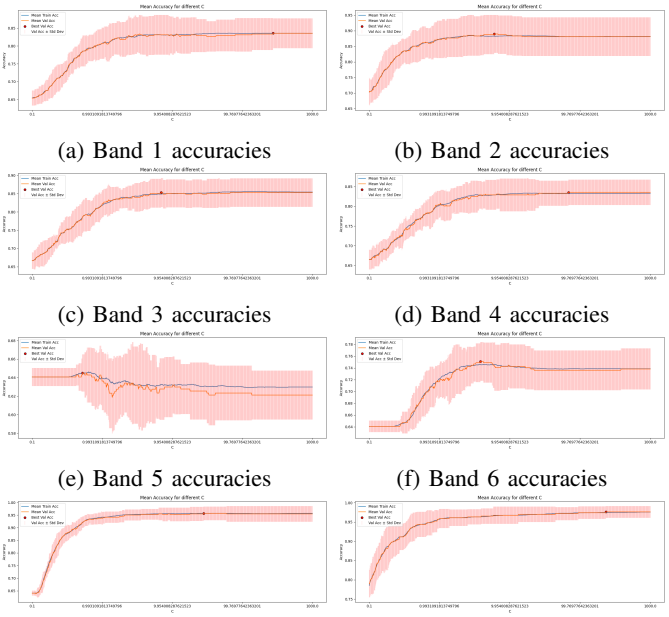
[6]Portion of the largest variance of all features that is added to variances for calculation stability.

[7]So that all processors are used.

[8]So that training scores are also returned in the results.

D. Results

Figure 6 shows the average training and validation accuracies as well as the standard deviations of the validation accuracies and the best validation accuracy for the logistic regression classifiers per band using the default `lbfgs` solver for different `C` values.



(a) Band 1 accuracies (b) Band 2 accuracies

(c) Band 3 accuracies (d) Band 4 accuracies

(e) Band 5 accuracies (f) Band 6 accuracies

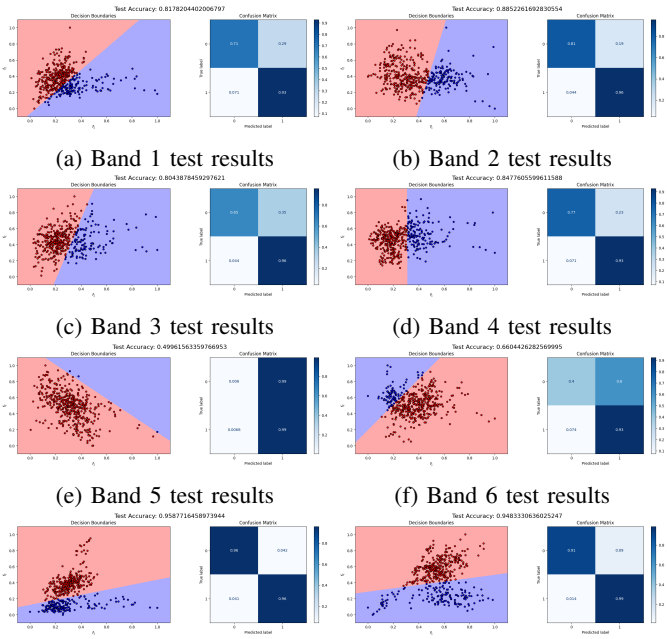
(g) Band 7 accuracies (h) NDVI accuracies

Fig. 6: Logistic Regression accuracies for `lbfgs` solver

We can observe that the average training and validation accuracies steadily increase and then converge around a certain accuracy for all the bands, except for band 5 since introducing more or less regularization in the model yields no significant improvements when the classes are inseparable. We can also observe that there seems to be an inverse correlation between the separability of classes for a band and the standard deviation of the validation accuracies. This suggests that bands with less separable classes, e.g. band 5, provides less stable results across different validation sets and vice versa.

Figure 7 shows the decision boundaries and confusion matrices[9] after running the fine-tuned logistic regression classifiers on the hold-out test sets of each band. We can observe from the confusion matrices that the classifiers perform better on the vegetation class for each band. This is most likely due to the fact that there is more vegetation pixels than settlement pixels in our dataset. We can also observe that there is a direct correlation between the separability of classes for a band and the accuracy of the classifier.

[9]Note, label 0 refers to the settlement class and label 1 refers to the vegetation class in the confusion matrices.



(a) Band 1 test results (b) Band 2 test results

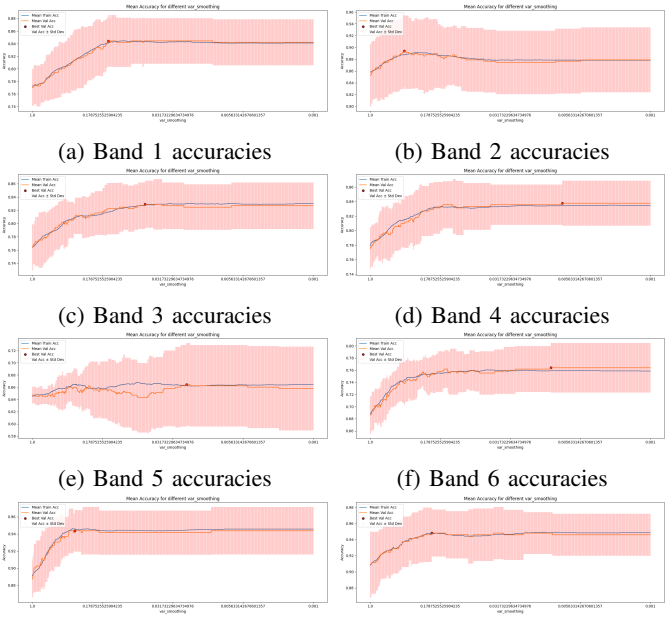
(c) Band 3 test results (d) Band 4 test results

(e) Band 5 test results (f) Band 6 test results

(g) Band 7 test results (h) NDVI test results

Fig. 7: Logistic Regression test results for `lbfgs` solver

Figure 8 shows the average training and validation accuracies for the naive bayes classifiers per band for different `var_smoothing` values as well as the standard deviations of the validation accuracies and the best validation accuracy.



(a) Band 1 accuracies (b) Band 2 accuracies

(c) Band 3 accuracies (d) Band 4 accuracies

(e) Band 5 accuracies (f) Band 6 accuracies

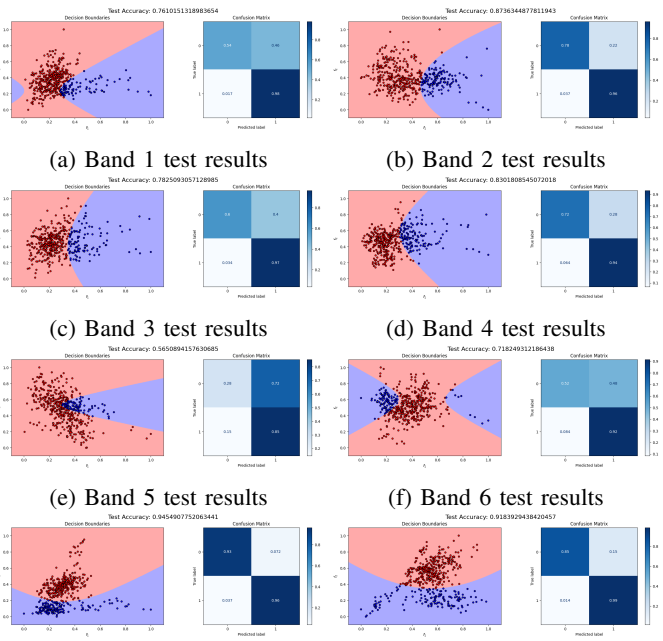
(g) Band 7 accuracies (h) NDVI accuracies

Fig. 8: Naive Bayes accuracies

Similar to logistic regression, we once again observe that the average training and validation accuracies steadily increase and then converge around a certain accuracy for all the bands

except for band 5. We can also see that the inverse correlation between the separability of classes for a band and the standard deviation of the validation accuracies seem to hold for the naive bayes classifiers as well. It is worth noting, however, that the naive bayes classifiers appear to exhibit more deviation in the validation accuracies compared to the logistic regression classifiers, suggesting less stable results across validation sets.

Figure 9 shows the decision boundaries and confusion matrices after running the fine-tuned naive bayes classifiers on the hold-out test sets of each band.



(a) Band 1 test results (b) Band 2 test results

(c) Band 3 test results (d) Band 4 test results

(e) Band 5 test results (f) Band 6 test results

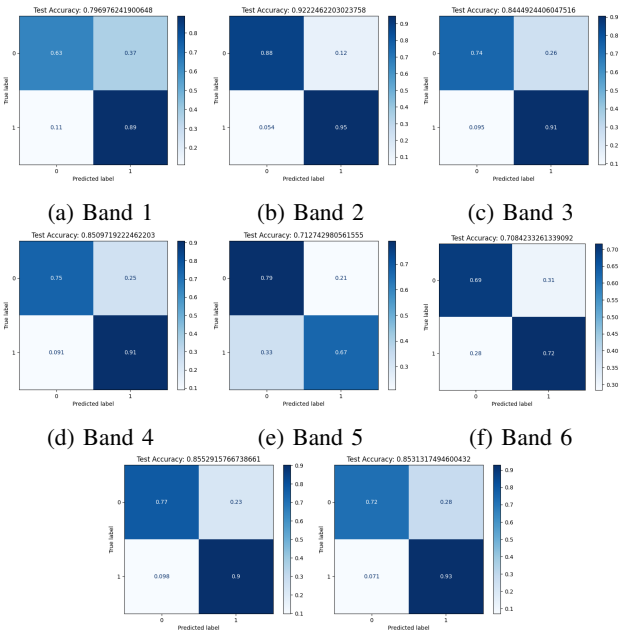
(g) Band 7 test results (h) NDVI test results

Fig. 9: Naive Bayes test results

Similar to logistic regression, we once again observe from the confusion matrices that the naive bayes classifiers perform better on the vegetation class for each band. We can also observe that the correlation between the separability of classes and accuracy holds true for the naive bayes classifiers as well. The most notable difference is that the naive bayes classifiers produce non-linear decision boundaries, which in fact helps it to achieve better results for bands (5 and 6) with less separable classes than the logistic regression classifiers.

Figure 10 shows the confusion matrices after training all necessary distributions of the Time-Varying Model and applying the model on the test set of every band. Like the naive bayes and logistic regression models the vegetation class is classified the most accurately except for the fifth band.

Figure 11 shows the confusion matrices after training all necessary distributions for the SPRT tests and applying the test on the test set for every band. Unlike the previous models, the SPRT tests are not generally most accurate when classifying vegetation. From the confusion matrices alone, it can be seen that the SPRT tests generally misclassify more frequently than the other models.

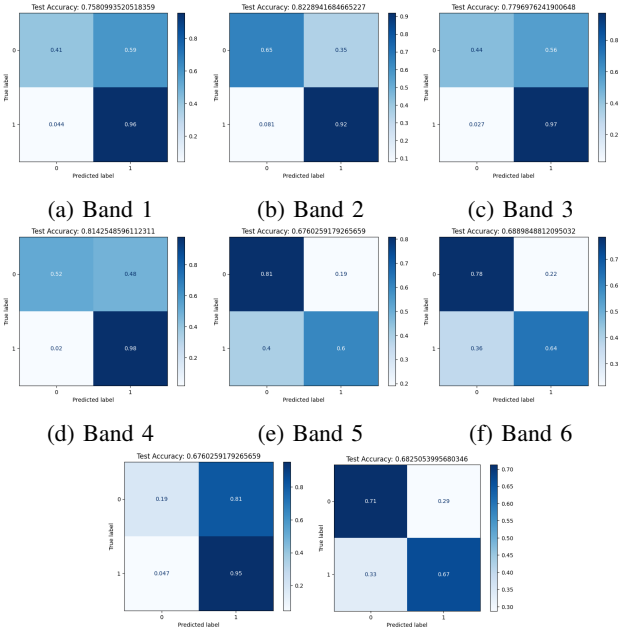


(a) Band 1 (b) Band 2 (c) Band 3

(d) Band 4 (e) Band 5 (f) Band 6

(g) Band 7 (h) NDVI

Fig. 10: Time-Varying Model test confusion matrices per band



(a) Band 1 (b) Band 2 (c) Band 3

(d) Band 4 (e) Band 5 (f) Band 6

(g) Band 7 (h) NDVI

Fig. 11: SPRT test confusion matrices per band

Figure 12 shows the average test accuracies and standard deviations per band for each classifier, after repeating the procedure described in II-C 10 times with a different random seed each time. The accuracies are calculated by taking the average of the diagonal values of the confusion matrices.

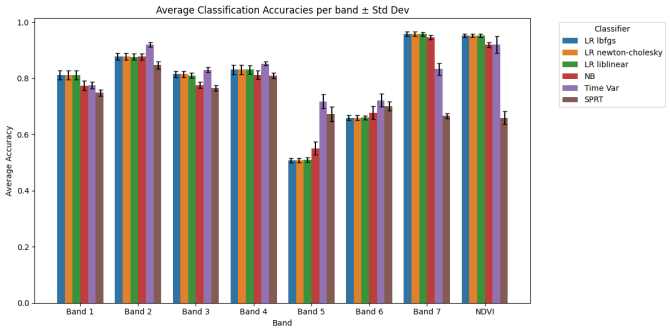


Fig. 12: Average classifier accuracies ± standard deviations

We can observe that using different solvers for logistic regression yielded no significant improvements. Naive bayes performs better than logistic regression for bands (5 and 6) with less separable classes because it makes an assumption of feature independence, which allows it to handle overlapping data points effectively. Conversely, logistic regression performs better than naive bayes for bands (1, 3, 4, 7, and NDVI) with more separable classes because it models linear decision boundaries, making it suitable for distinguishing well-separated classes. Logistic regression and naive bayes achieve similar results for band 2. The time-varying model performs best across all classifiers for bands 2, 3, 4, 5 and 6, which is likely because too much information is lost through dimensionality reduction after applying logistic regression and naive bayes. Logistic regression performs best for the remaining bands 1, 7 and NDVI, which indicates that if dimensionality reduction can result in well separable classes it is still feasible. The SPRT tests consistently perform worse than the time-varying model, which indicates that the tests classify too abruptly.

We can also observe from the standard deviations that there is less deviation from the average accuracies for logistic regression suggesting that it provides more stable and consistent results across various random initializations compared to naive bayes. The time-varying model produces less consistent results for bands 5, 6, 7 and NDVI as observed from the large standard deviations.

III. Fast Fourier Transform

A. Fourier Transform

A Fourier Transform is a mathematical operation that is used to transform signals or functions from the time or spatial domain into the frequency domain. It allows you to decompose a complex waveform or signal into its underlying frequency components. It is defined as:

$$F(f) = \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{-2\pi ift}dt \tag{16}$$

Where:

$F(f)$: Fourier transformed spectrum of signal f
$f(t)$: Time-domain signal
t : Time variable
f : Frequency variable
i : Imaginary unit

B. Discrete Fourier Transform

The Discrete Fourier Transform (DFT) as the name suggests analyses the frequency components of discrete signals. It is computed as:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-(i\frac{2\pi nk}{N})} \tag{17}$$

Where:

$X(k)$: k^{th}element of the Fourier transformed spectrum X
$x(n)$: Input signal in the time domain
N : Number of samples
k : Frequency index
i : Imaginary unit

This formula calculates the sum over n elements of the function x, each multiplied by a complex twiddle factor $e^{-(i\frac{2\pi nk}{N})}$.

C. Fast Fourier Transform

The Fast Fourier Transform (FFT) significantly reduces the computational complexity of computing the Fourier transform - especially for large N - by utilizing recursion, resulting in a computational complexity of approximately $O(N \log_2 N)$. The computational efficiency of the FFT is directly attributable to its recursive design, which capitalizes on a binary tree structure and incorporates generalized twiddle factors. The Radix-2 Decimation in Time (DIT) FFT is a common variant of the FFT, which recursively divides the DFT computation into smaller parts. The formula is given below:

$$
\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(n) e^{-(i\frac{2\pi nk}{N})} \leftarrow DFT \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-(i\frac{2\pi(2n)k}{N})} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-(i\frac{2\pi(2n+1)k}{N})} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-(i\frac{2\pi nk}{\frac{N}{2}})} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-(i\frac{2\pi nk}{\frac{N}{2}})} \\
&= DFT_{\frac{N}{2}} \left[[x(0), x(2), \ldots, x(N-2)]\right] \\
&\quad + W_N^k DFT_{\frac{N}{2}} \left[[x(1), x(3), \ldots, x(N-1)]\right]
\end{aligned}
\tag{18}
$$

Where:

$X(k)$: k^{th}element of the Fourier transformed spectrum X

$x(n)$: Input signal in the time domain

N : Number of samples

k : Frequency index

W_N^k : Twiddle factor $(e^{-(i\frac{2\pi k}{N})})$

D. Results

In this section, we evaluate the performance of our Radix-2 DIT FFT implementation. First, we confirm that our implementation is correct by comparing its output to that of Numpy's FFT using numpy.allclose which checks if the values are the same within a tolerance. Now that we have confirmed that our implementation is correct, we compare it's performance with other methods, including a double for loop DFT, a vectorised DFT, 1D Cooley-Tukey FFT using one layer, and Numpy's FFT. The results are summarized in table I.

Method	Average Time	Standard Deviation
Double Loop DFT	126 ms	12.2 ms
Matrix DFT	9.37 ms	5.17 ms
One Layer FFT + Matrix DFT	6.96 ms	554 µs
Our FFT	3.54 ms	773 µs
Numpy's FFT	7.31 µs	2.18 µs

TABLE I: Computation times for random input of length 256

We can see that Numpy's implementation is the fastest which is due to its low-level implementation. However, our implementation still outperforms the rest of the methods.

Next we introduce a generic version of our implementation that can take input of any length. It first checks if the input length is divisible by 2, if so then it will use our FFT, otherwise it will default to the slower vectorized DFT implementation. We use the same method described earlier to confirm that our implementation is correct and then evaluate its performance on the following input vectors:

- $xTest2$: Random input of length 251.
- $xTest3$: Random input of length 12×32.

Table II shows the computation times of our generic FFT implementation on the input vectors above:

Method	Average Time	Standard Deviation
generalFFT($xTest2$)	5.57 ms	199 µs
generalFFT($xTest3$)	3.26 ms	108 µs

TABLE II: Computation Times for our generic FFT

We can see that even though $xTest3$ is larger than $xTest2$ our generic FFT still runs faster on it since it is divisible by 2 and our algorithm does not default to the slower vectorized DFT.

REFERENCES

[1] T. Grobler, W. Kleynhans, and B. Salmon, "Empirically comparing two dimensionality reduction techniques – pca and fft: A settlement detection case study in the gauteng province of south africa," in *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*, 2019, pp. 3329–3332.

[2] T. Grobler, E. Ackermann, A. van Zyl, W. Kleynhans, B. Salmon, and J. Olivier, "Sequential classification of modis time series," in *2012 IEEE International Geoscience and Remote Sensing Symposium*, 2012, pp. 6236–6239.