Biometric Liveness and Anomaly Detection

Breanna Seitz
Arizona State University, Tempe, USA
bdseitz1@asu.edu

Shawn Deason
Arizona State University, Tempe, USA
smdeason@asu.edu

Justin Colyar
Arizona State University, Tempe, USA
jcolyar@asu.edu

Prakhar Bhartiya
Arizona State University, Tempe, USA
pbhartiy@asu.edu

***Abstract* - An android application to detect the liveness of EEG signals. This application takes in user input and selects data from a provided data set to predict its liveness. The app communicates to a server via the Internet to retrieve the data and machine learning models.**

I. Introduction

In recent years we have made profound technological advancements within the medical field that allow us to accurately measure biometrics with the use of sensors. With the rise in use of these new technologies, issues are arising in the trustworthiness and reliability of data as sensors are prone to cyber attacks such as false data injection. This can be especially dangerous in situations where these sensors are being used to prescribe proper doses of medication. In this project, we will attempt to use machine learning to detect the liveness of EEG data through an android application.

II. Author Keywords

Biometric Liveness, Brain Signals, EEG, Machine Learning

III. Project Setup & Permissions

The architecture is made up of 2 systems. The system configuration and functions are:

1. Android Device (Emulated)
    a. OS: Android 9.0
    b. API Level: 28
    c. Libraries: Chaquopy
2. Fog Server (Local Server)
    a. Launched on same local area network as android device
    b. Port: 5000
    c. OS: Windows 10

IV. Fog Server Setup

1. Uses Flask RESTful api hosted locally.
2. Models and attack data are trained and stored locally.
3. Android device downloads the trained files.

V. Implementation

1. Data, Preprocessing, and Feature Extraction

We were provided two datasets for this project: live data and attack data. The live data consisted of 2 minute samples for 106 subjects, with 3 runs for each subject. With a sampling rate of 160 Hz, this provided us with 19200 samples per run. Similarly, the attack data consisted of 30 second samples for 106 subjects with 3 runs per subject over 6 attack types. With the same sampling rate, this results in 4800 samples per run. We segmented all of this data into 30 second segments and created labels for each segment, with 0 being live data and 1 being attack data.

Next, we performed frequency filtering to filter out frequencies below 0.1Hz and above 60Hz. Then we performed scaling to normalize the data.

Following this was feature extraction, in which we have selected six features: alpha band, beta band, delta band, power spectral density, PCA, and coiflets. The frequency bands and power spectral density were extracted using the Fourier transform. For PCA, we arbitrarily selected 20 components. This extracts variance between the data which is valuable in brain signals. For coiflets, which is a discrete wavelet family, we extract the approximation coefficients.

2. Training and Testing the Models

For this project, we developed our models using Python scikit-learn. In order to train our models, we first split the dataset into a training set (70% of the data) and a testing set (30% of the data). For our four models, we chose Logistic Regression, SVM, KNN for our distance model, and K-Means as an unsupervised learning technique. To avoid the curse of dimensionality and prevent overfitting, we have chosen to run one feature per model, resulting in 24 total models.

These models are trained on the server and then downloaded onto the emulator. To save these trained models, we have used Python Pickle.

Results

Alpha	Logistic Regression	K-Means	SVM	KNN
Accuracy	0.99	1	1	1
TPR	1	1	1	1
TNR	0.97	1	1	1
Precision	0.98	1	1	1
FNR	0	0	0	0
FPR	0.02	0	0	0
Error	0.01	0	0	0
F1	0.99	1	1	1
Half-Total Error	0.01	1	1	1

Figure 1: Table populated with aggregate results of testing the models using the alpha bands

Beta	Logistic Regression	K-Means	SVM	KNN
Accuracy	0.99	1	0.99	1
TPR	0.99	1	1	1
TNR	0.98	1	0.98	1
Precision	0.98	1	0.98	1
FNR	0.001	0	0	0
FPR	0.01	0	0.1	0
Error	0.01	0	0.01	0
F1	0.99	1	0.99	1
Half-Total Error	0.01	1	0.01	1

Figure 2: Table populated with aggregate results of testing the models using the beta bands

Delta	Logistic Regression	K-Means	SVM	KNN
Accuracy	0.98	1	0.99	1
TPR	0.98	1	1	1
TNR	0.98	1	0.99	1
Precision	0.98	1	0.99	1
FNR	0.01	0	0	0
FPR	0.01	0	0.002	0
Error	0.01	0	0.001	0
F1	0.98	1	0.99	1
Half-Total Error	0.01	1	0.001	1

Figure 3: Table populated with aggregate results of testing the models using the delta bands

PSD	Logistic Regression	K-Means	SVM	KNN
Accuracy	1	1	1	1
TPR	1	1	1	1
TNR	1	1	1	1
Precision	1	1	1	1
FNR	0	0	0	0
FPR	0	0	0	0
Error	0	0	0	0
F1	1	1	1	1
Half-Total Error	1	1	1	1

Figure 4: Table populated with aggregate results of testing the models using the power spectral density

PCA	Logistic Regression	K-Means	SVM	KNN
Accuracy	1	0.74	1	1
TPR	1	1	1	1
TNR	1	0.39	1	1
Precision	1	0.69	1	1
FNR	0	0	0	0
FPR	0	0.6	0	0
Error	0	0.25	0	0
F1	1	0.82	1	1
Half-Total Error	1	0.3	1	1

Figure 5: Table populated with aggregate results of testing the models using PCA

Coiflets	Logistic Regression	K-Means	SVM	KNN
Accuracy	1	0.73	1	1
TPR	1	1	1	1
TNR	1	0.36	1	1
Precision	1	0.68	1	1
FNR	0	0	0	0
FPR	0	0.63	0	0
Error	0	0.26	0	0
F1	1	0.81	1	1
Half-Total Error	1	0.31	1	1

Figure 6: Table populated with aggregate results of testing the models using the coiflets

Training	Logistic Regression	K-Means	SVM	KNN
Alpha band	0.0435	0.03696	5.849	0.00062
Beta band	0.14662	0.08036	0.33786	0.001008
Delta band	0.03989	0.0333	4.4318	0.00065
Power Spectral Density	0.24569	0.33628	0.3936	0.00339
PCA	0.017	0.12883	0.00286	0.000507
Coiflets	0.1763	0.84626	0.275	0.00335

Figure 7: Table populated with training times (in seconds) for each of the 24 models

Testing	Logistic Regression	K-Means	SVM	KNN
Alpha band	0.000222	0.05488	0.000309	0.07981
Beta band	0.00053	0.00284	0.00062	0.09383
Delta band	0.00024	0.00192	0.00037	0.06265
Power Spectral Density	0.00259	0.00758	0.0022	0.17814
PCA	0.00011	0.00167	0.00018	0.08239
Coiflets	0.00232	0.00749	0.0022	0.1834

Figure 8: Table populated with training times (in seconds) for each of the 24 models

3. Attack Signal Generation

For creating the generative adversarial network (GAN), we first looked at the sample models that used MNIST digits [1]. We adapted these models to instead generate brain signals. For the generative model, we decided to use a sequential model with leaky redacted unit layers with a dropout rate of 0.4. Based on experimentation, these layers seemed to perform the best and the dropout rate helps to ensure that the model is not memorizing the training data. Moreover, the activation function for the generative model was softsign, and this too was chosen based on the best experimental results. Finally, this data is reshaped to reproduce a sample brain signal sample.

The discriminator part of the generative adversarial network is very similar to the generative model but in reverse. It is a sequential model with leaky redacted units decreasing from 512 to 1 before finally being fed to a sigmoid activation function.

For training in the GAN model, the general process is to generate some noise to be fed into the generator. This creates a new signal given the noise and the trained model which can then be checked against the discriminator to compute the loss values and ultimately fed back into the generator to train. Moreover, it's worth noting that the brain training data is changed to be in the relative range of -1 to 1 to improve the classification using the sigmoid function.

The variational autoencoder is composed of the decoder and encoder. Due to the relative success of the layers in the model for the generative

adversarial network, similar layers were applied to the encoder and decoder following the example specified in [2]. More specifically, for the encoder, we increase the units for each layer from 32 to128 (multiplying by 2) while using the redacted linear unit. The mean, log variance, and sampling of this last layer are then taken to encode the result.

For the decoder, the layers are reduced in units from 128 to 1, using the sigmoid activation function and reshaping to reproduce a brain signal sample. With both the decoder and encoder, the model can be trained by first resizing the data to be in the range -1 to 1 (for the sigmoid activation function). The training steps then consist of encoding a sample data with the encoder, reconstructing it using the decoder, and then calculating the losses (total loss, reconstruction loss, etc). The gradient from the loss is then fed into the optimizer.

The results for the timings of the GAN and VAE attack models as well as for the signal generation are listed in the following table:

	GAN	VAE
Model Training	52.78s	164.93s
Attack Signal Generation	0.33s	0.14s

The accuracy of the generated signals is listed in the following output. A classification of zero means the attack vector is considered live, a 1 means that it is considered fake. The first screenshot is the classification results of the GAN signal, the second is the result of the VAE signal. The results are of the classification outputs from the models in the form (logReg model, K-means model, SVM model, KNN model).

```
(logReg, Kmeans, SVM, KNN)
0.0 = classified as liveness, 1.0 = classified as fake
feature extraction: PCA
(1.0, 1.0, 1.0, 1.0)
feature extraction: alpha
(1.0, 1.0, 1.0, 1.0)
feature extraction: delta
(1.0, 1.0, 1.0, 1.0)
feature extraction: beta
(0.0, 0.0, 0.0, 0.0)
feature extraction: PD
(1.0, 1.0, 1.0, 1.0)
feature extraction: coif
(1.0, 1.0, 1.0, 1.0)
```

Figure 9: Classification of 1 GAN attack signal

```
(logReg, Kmeans, SVM, KNN)
0.0 = classified as liveness, 1.0 = classified as fake
feature extraction: PCA
(1.0, 1.0, 1.0, 1.0)
feature extraction: alpha
(1.0, 1.0, 1.0, 1.0)
feature extraction: delta
(1.0, 1.0, 1.0, 1.0)
feature extraction: beta
(0.0, 0.0, 0.0, 0.0)
feature extraction: PD
(1.0, 1.0, 1.0, 1.0)
feature extraction: coif
(1.0, 1.0, 1.0, 1.0)
```

Figure 10: Classification of 1 VAE attack signal

4. *Creating the Application/Server*

The UI needed to support running single and multiple samples, selecting attack data, selecting a feature, and running models. Single and multiple samples are differentiated by an input to the text field for selecting users. An input in the form of a single number, #, results in a single user test, while an input in the form of, #-#, results in a multi-user test from the start to end range and it also ignores attack data so it disables that selector. The attack data and features are selected through dropdowns and once all 3 select buttons are pressed the models can be run. If a multi-user test is run, the aggregate results are instead outputted with a scrollbar.

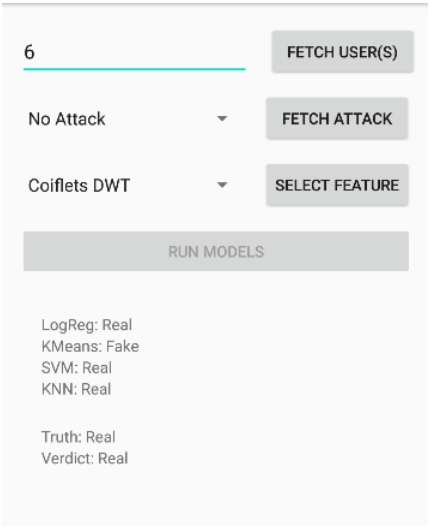

Figure 10: Liveness Detection UI

Chaquopy was used in the app as it allowed for our app to run python natively. This allowed for us to not have to use any other library besides what the models were trained in. That being said, it is a licensed library, however, we are able to use it in limited capacity unlicensed for 5 minutes at a time. Once the time is up the app shuts down.

Communication with the server is done through making POST http requests to server routes. The routes are used to download user data files, attack data files, and trained model files. The app will attempt to download models once the run models button is clicked, if the files are already downloaded they get ignored. The attack is downloaded once the fetch attack button is pressed. All downloads happen on async threads.

The server was written using Flask as it allowed for the python code to be shared between Chaquopy and the server easing development. On initial startup, the server will attempt to train all 4 models on the 6 features one feature at a time.

Currently, the application, if run inside of the Android Studio AVD emulator, cannot download .pkl or .mat files greater than 15KB. The download downloads 99% of the bytes then it errors out and thus breaks those files and crashes the app later. To deal with this the files need to be copied into the /sdcard/livenessApp/ folder on the emulated device.

5. *Using the Application*

a) *Single Sample*

To test a single sample, the user will be asked to input a user ID within the range 0-105. This corresponds to the 106 subjects in the provided data sets. The user will also select an attack type, with -1 being no attack, 0-5 corresponding to the 6 attack types provided, and 6 and 7 corresponding to GAN and VAE attacks respectively. The user will also be able to select one of the six features to test on. Once this is submitted, the app will grab the respective sample, perform feature extraction, and use all 4 models to predict its liveness. The prediction for each model is displayed, along with a voting model which is a majority vote that gives the final verdict.

b) *Multiple Samples*

To test multiple samples, the user will be asked to input a range of user IDs within the range 0-105. This corresponds to the 106 subjects in the provided data sets. The user will also select one of the six features to test on. Once this is submitted, the app will grab the data for each subject within the range from both the live dataset and all the attack sets. The aggregate results of the predictions will then be calculated for each model and displayed on the screen.

VI. Completion of Tasks

Serial	Task Name	Assignee
1	Create an app capable of receiving data through internet	Shawn
2	Data Preprocessing	Prakhar
3	Feature Extraction	Prakhar, Breanna
4	Have four trained models on the system	Shawn
5	One single input is received and liveness is reported	Shawn
6	A group of inputs is received and aggregate results are reported	Shawn, Breanna
7	Train models on machine	Breanna
8	Two class classification, utilize unsupervised learning	Prakhar
9	Test models beyond testing set	Justin
10	GAN/VAE Signal Generation	Justin
11	Execution Time Analysis for training, testing, attack generation	Breanna, Justin
12	Performance analysis for training and testing of models	Prakhar

VII. Limitations

Our models were trained and tested on a dataset provided consisting of EEG data from 106 subjects and 6 attack types. We were unable to access a brain sensor or obtain new data from other subjects to further test our models. Additionally, we could not test the app using truly live brain signals directly from a brain sensor. Because of this, most of the data being used to detect liveness in the app has already been seen by the models. This results in significantly better outcomes than in a real-world scenario. Moreover, for generating the attack vectors, the limited data would negatively impact the performance and accuracy of GAN and VAE performance.

VIII. Detecting Non-Recorded Liveness

Detecting the liveness of biometric data such as brain signals has two main properties: brain signals belong to humans (rather than being artificially generated), and the collected signal is fresh (not being pre-recorded). The first property is being implemented in the project utilizing the above-mentioned methodology, and for the second property, we have planned two techniques.

1. *Freshness based on predefined brain patterns:*

In this technique, we can use a pre-recognized common brain signal, such as eye blinking, as a token for liveness. We will ask the user to blink for a specified number of times rapidly at random time intervals, and we should see the same amount of blinking brain signal patterns in the input signal. As a result, we can correlate and identify whether the current brain signal is live or stale. Another similar strategy would be to provide the user visual content such as video/images that trigger common human reactions and relevant brain signals. For example, a user may be shown a funny image/video at a random time, and the user's brain activity could be analyzed during that time window, which we could then compare to a common brain pattern during such emotion. Both of these ideas' ideologies are to link an already known brain signal pattern as a token generated at an application-defined random moment.

2. *Verification based on visualization*

In recent years numerous applications of studying brain activity have emerged. Researchers from Russian corporation Neurobotics and the Moscow Institute of Physics and Technology have found a way to visualize a person's brain activity as actual images mimicking what they observe in real-time [3]. A similar technique can be used to evaluate the liveness of a brain signal. The user will be shown a picture and asked to envision it, after which we will rebuild the image using brain signals and compare the two images to see if they are comparable. If the photos are similar to a threshold (important to the outcome of the process), the data is live; otherwise, it is stale.

IX. Conclusion

In conclusion, we have developed an android application to partially detect liveness in biometric data. The application receives data as input via communication with the server, and through the use of trained machine learning models, verifies the realness of the data. Additionally we were able to generate false brain signals using the GAN and VAE signal generation methods. Future work for this project would include taking in real-time data via a brain sensor and implementing a detection method to verify the authenticity of this data.

X. Acknowledgements

We would like to thank our professor Ayan Banerjee for giving us the opportunity to learn and work on many great mobile computing problems and applications. It is in part through these tasks that we are able to gain valuable experience in a variety of fields. We would also like to thank the TA Mohammad Javad Sohankar Esfahani for working with us personally on implementing and helping us understand the intricate details of EEG data, ML models, feature extraction, and attack signal generation.

XI. References

[1] K. Team, "Keras documentation: Variational Autoencoder," *Keras*. [Online]. Available: https://keras.io/examples/generative/vae/. [Accessed: 04-May-2022].

[2] R. Atienza, "Gan by example using Keras on tensorflow backend," *Medium*, 28-Apr-2017. [Online]. Available: https://towardsdatascience.com/gan-by-example-using-keras-on-tensorflow-backend-1a6d515a60d0. [Accessed: 04-May-2022].

[3] G. Rashkov, A. Bobe, D. Fastovets, and M. Komarova, "Natural Image Reconstruction from brain waves: A novel visual BCI system with native feedback," *bioRxiv*, 01-Jan-2019. [Online]. Available: https://www.biorxiv.org/content/10.1101/787101v2. [Accessed: 05-May-2022].

[4] C. Delgado, "How to use Chaquopy to run Python code and obtain its output using java in your Android App," *Our Code World*, 12-Jan-2022. [Online]. Available: https://ourcodeworld.com/articles/read/1656/how-to-use-chaquopy-to-run-python-code-and-obtain-its-output-using-java-in-your-android-app. [Accessed: 05-May-2022].

[5] "Gradle plugin#," *Gradle plugin - Chaquopy 11.0*. [Online]. Available: https://chaquo.com/chaquopy/doc/current/android.html. [Accessed: 05-May-2022].