Machine Learning Approach for Classifying Natural and Anthropogenic Seismic Events

Saeid Zarifikoliaee, Mark Grmek Supervised by Prof. Dr.-Ing. Rupert Klein

December 2024

Abstract

The distinction between natural and anthropogenic seismic events represents an important task within the field of geological observation and surveying. Historically, this task demanded considerable time and expertise, necessitating the trained eye of experienced professionals. However, with the rapid advancement of Artificial Intelligence (AI), there emerges a promising prospect of possibly transferring this task to computers entirely which could potentially enable better insight into natural seismic activity, thereby facilitating more accurate predictions and proactive warning systems for future events. In this study, we explore and compare two machine learning methodologies for the classification of seismic events: Convolutional Neural Networks (CNNs) and Principal Component Analysis combined with Support Vector Machines (PCA-SVM). Although CNNs demonstrate superior accuracy and robustness, their computational demands are significantly higher. In contrast, PCA-SVM offers a computationally efficient solution with competitive performance, making it a viable option for real-time applications. By analyzing these approaches, this work provides a comprehensive framework for balancing accuracy and efficiency in seismic event classification, laying the groundwork for future advancements in automated geological monitoring systems.

Keywords: seismic filtering, convolutional neural network, support vector machine, principal component analysis, machine learning, time series analysis.

1 Introduction

Seismic waves manifest across a spectrum of origins, ranging from the natural dynamics of tectonic plate movements to volcanic eruptions, landslides, and human-induced activities such as mining, construction, transportation of heavy loads, and demolition of structures. Categorizing these instances hinges upon discerning the underlying cause – natural phenomena or human activity. This classification is fundamental for both scientific understanding and practical applications, such as risk assessment and mitigation in regions vulnerable to seismic hazards.

The ability to distinguish between natural and anthropological seismic events is of significant importance. In regions prone to tectonic or volcanic activity, increased seismic activity often serves as a vital precursor to major geological events, such as earthquakes, tsunamis, or volcanic eruptions. Detecting and interpreting these signals accurately can provide early warnings, enabling communities and authorities to take proactive measures to minimize damage and loss of life. In contrast, in densely populated urban or industrial areas, human-induced seismic events, ranging from mining explosions to construction vibrations, can mask or mimic natural seismic disturbances, complicating the interpretation of underlying geological activity. Thus, effectively separating these two categories is crucial for accurate monitoring and analysis.

Traditional approaches to seismic event classification rely on manual inspection by experienced seismologists, a process that is time-consuming, resource-intensive, and prone to human error. With the exponential growth of seismic data, especially in areas with dense networks of seismometers, this manual approach is increasingly impractical. Automated systems powered by machine learning offer a promising alternative, enabling rapid and large-scale analysis of seismic data. These systems not only reduce the burden on human analysts, but also improve accuracy and consistency by leveraging patterns in data that may not be immediately apparent to the human eye.

This study aims to address the challenges of seismic event classification by comparing two distinct machine learning approaches. Convolutional Neural Networks (CNNs) and Principal Component Analysis combined with Support Vector Machines (PCA-SVM). CNNs are well-suited for complex, high-dimensional data, offering state-of-the-art performance in various classification tasks. However, their computational demands may limit their applicability in resource-constrained settings. However, PCA, as part of the PCA-SVM pipeline, provides a computationally efficient method for dimensionality reduction. This simplifies the classification process while maintaining competitive accuracy when combined with the SVM classifier. By examining the trade-offs between these approaches, this work offers insight into selecting an appropriate methodology based on key factors such as accuracy, computational efficiency, and real-time applicability[1],[2].

2 Data description and definitions

To gain the most detailed understanding of seismic activity, particularly individual seismic events, we analyze the waveforms recorded by seismographs. These waveforms provide crucial information on the characteristics of seismic waves, such as amplitude, frequency, phase, and event duration. Waveform analysis also enables the identification of different types of seismic waves, such as P-waves and S-waves which can have valuable insight into the origin of the seismic event [1]. Thus, examining waveforms is an essential part of any kind of predictive model for seismic events.

2.1 Events dataset

In this study, we utilized the labeled seismic events data compiled in the "Catalogue of Earthquake Hypocenters for Northern Chile from 2007-2021 using IPOC (plus auxiliary) seismic stations" provided by GFZ Data Services [3] (Figure 1).

The dataset contains detailed information for each seismic event, namely: time of the event (YYYY-MM-DD-HH:MM:SS.SSS), the position of the event (Latitude, Longitude, Depth [km]), Magnitude (ML), Magnitude standard deviation (MA), and the Event classification; each event in the dataset is pre-classified based on its origin or cause into the following event classes [4]:

- P1 - Plate Interface
- P2 - Double Seismic Zone - Upper Plane
- P3 - Double Seismic Zone - Lower Plane
- ID - Intermediate Depth
- UP - Upper Plate
- MI - Mining-related
- NN - Not Classified

To tailor the dataset for our research, we first removed the events labeled with the NN class marker. We then reclassified the events with the P1, P2, P3, ID, and UP class markers into a single 'Natural' class (class marker 0). Events labeled with the MI class marker were reclassified into an 'Anthropogenic' class (class marker 1). This process resulted in a coarsened database that was more suitable for conducting our research (visualized in Figure 1).

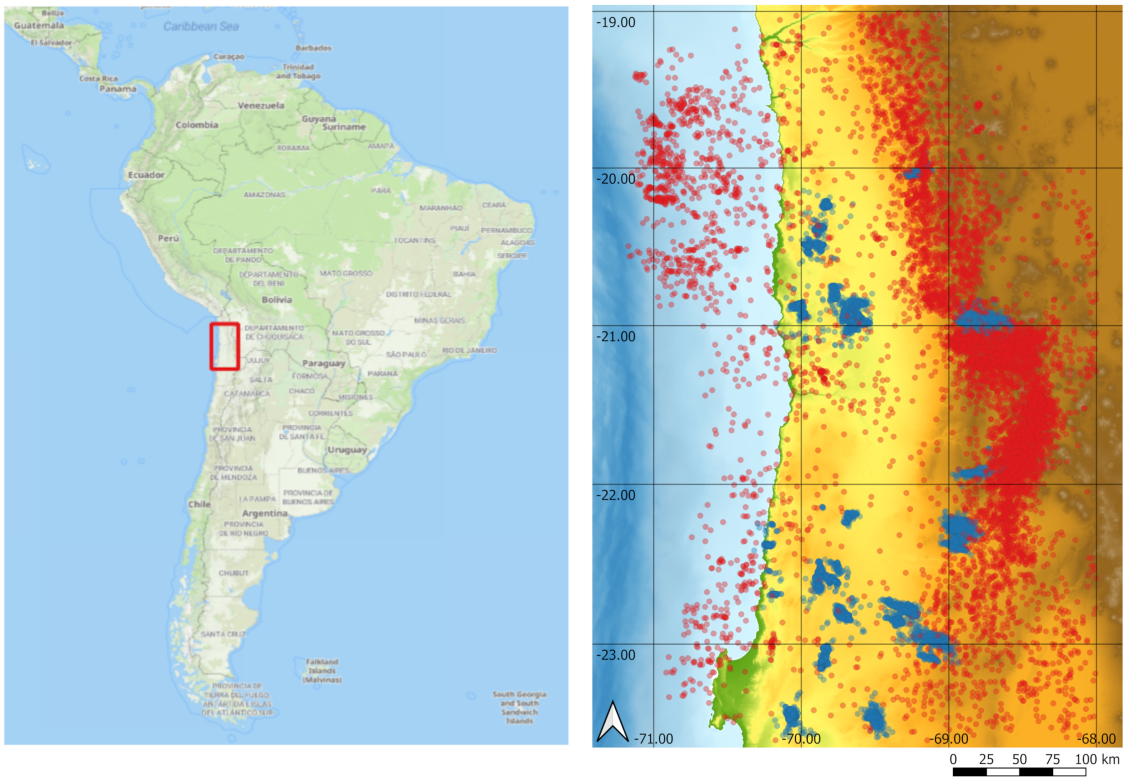

Figure 1: Natural seismic events greater than 3 magnitude (red) and anthropogenic seismic events (blue)

2.2 OBSPY

ObsPy is an open-source project dedicated to providing a Python framework for processing seismological data. It provides parsers for common file formats, clients to access data centers, and seismological signal processing routines that allow the manipulation of seismological time series [5].

Using the previously filtered and re-classified catalog, we could fetch the waveforms for each event from the OBSPY API. We obtained the waveforms for a 60-second window before and after the specified event time in the catalog. Data was retrieved from a single station (LVC station) in the GEOFON - GE network, and waveforms were collected in all three directions. Broad-band weak motion single-component sensors recorded the waveforms [6](BHZ, BHN, BHE, or BH1, BH2). In total, we utilized 50,000 three-component waveforms, each consisting of 4801 time steps or data points. Finally, we converted the waveforms into spectrograms, following the methodology [7] demonstrated by Linville et al., who showed that using spectrograms in Convolutional Neural Networks (CNNs) yields significantly better results compared to using raw waveforms. In contrast, for the PCA-SVM we used the raw waveforms.

2.3 Waveform Data Normalization and Processing

Seismic waveforms are typically represented as matrices $\mathbf{X}_i \in \mathbb{R}^{N \times C}$, where N denotes the number of time steps and C represents the number of channels or components for the event i. In this study, each waveform corresponds to a single seismic event and consists of three components ($C = 3$) representing ground motion recorded in three orthogonal directions: Vertical (Z), North-South (N) and East-West (E) or Vertical (Z), 1st orthogonal (1) and 2nd orthogonal (2). These components are crucial for capturing the spatial dynamics of seismic waves.

The normalization of these waveforms is essential to mitigate the effects of varying amplitude scales caused by differences in event magnitude, instrument sensitivity, and environmental noise. This ensures that the learning process leading to the model is not dominated by large amplitudes in certain features. The normalization process is defined as:

$$\mathbf{X}'_i = \frac{\mathbf{X}_i - \mu}{\sigma + \epsilon}, \tag{1}$$

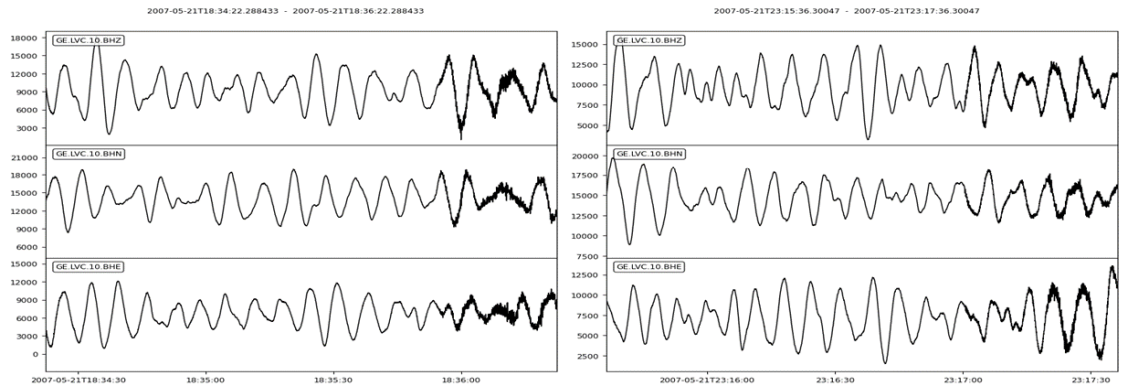

Figure 2: Example of natural (right) and anthropogenic (left) waveforms in three orthogonal directions

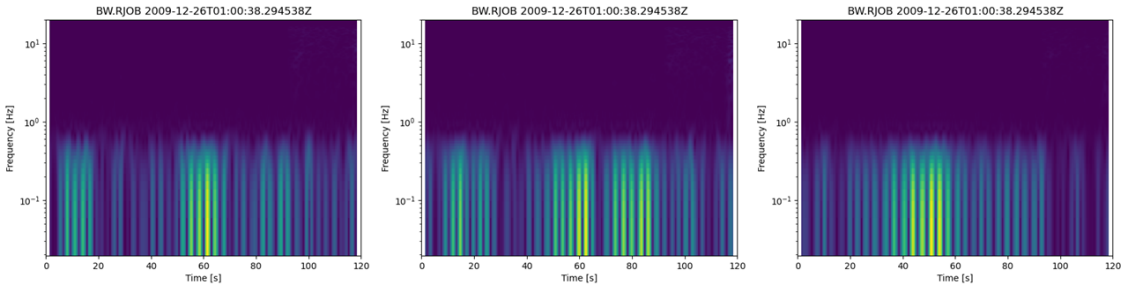

Figure 3: Example of spectrogram in three orthogonal directions

where μ and σ are the mean and standard deviation of $\mathbf{X}_i$, and ϵ is a small constant to prevent division by zero.

The values of μ and σ are computed over the individual waveform $\mathbf{X}_i$ (across all time steps and channels), ensuring that each event is normalized independently. This standardization ensures that all channels of a single waveform contribute equally to the learning process, reducing biases that might arise from differences in amplitude across events or components.

2.4 Data Representation: Raw Waveforms and Spectrograms

A raw waveform is a time-series signal representing ground motion recorded by seismographs (Figure 2). Mathematically, a raw waveform for an event i is represented as:

$$\mathbf{w}(t) : [t_0, t_f] \to \mathbb{R},$$

where t_0 and t_f are the start and end times of the observation window. The waveform is sampled at discrete intervals, resulting in a vector representation:

$$\mathbf{w} = [w_1, w_2, \ldots, w_N] \in \mathbb{R}^N.$$

The three components (N, E, Z) are then concatenated to form a unified representation:

$$\mathbf{X}_i = \begin{bmatrix} \mathbf{w}_\mathrm{N} & \mathbf{w}_\mathrm{E} & \mathbf{w}_\mathrm{Z} \end{bmatrix} \in \mathbb{R}^{N \times C}.$$

A spectrogram is a transformation of the raw waveform into the time-frequency domain, providing insights into the frequency content of the signal as it evolves over time (Figure 3). It is computed by applying a Short-Time Fourier Transform (STFT) to overlapping windows of the waveform. Mathematically, the spectrogram $\mathbf{S}(t, f)$ is given by:

$$\mathbf{S}(t, f) = \left| \mathcal{F}\{\mathbf{w}(t)\} \right|^2,$$

where $\mathcal{F}\{\cdot\}$ denotes the Fourier Transform, t represents the time window, and f is the frequency. The resulting spectrogram is a 2D matrix:

$$\mathbf{S} \in \mathbb{R}^{T \times F},$$

where T is the number of time windows, and F is the number of frequency bins. This representation is particularly useful for ML models, as it captures both temporal and frequency information, essential for distinguishing between natural and anthropogenic seismic events.

2.5 Machine Learning Objective

The machine learning model is designed to learn a classification function f that maps seismic events to their respective categories:

$$f : \mathcal{X} \to \mathcal{Y},$$

where: $\mathcal{X} \subset \mathbb{R}^d$ is the feature space, and d depends on the representation (raw waveform or spectrogram):

- For raw waveforms, $d = N \cdot C$.
- For spectrograms, $d = T \cdot F$.

and $\mathcal{Y} = \{0, 1\}$, where 0 represents natural events and 1 represents anthropogenic events.

The function f aims to map the input data to the correct label by learning patterns inherent to each category. The domain of definition of f is constrained to the subset of $\mathcal{X}$ corresponding to feasible seismic signals, characterized by the expected range of amplitudes, frequencies, and durations encountered in real-world seismic data.

2.6 Accuracy Definition

In this study, accuracy is used as the primary metric to evaluate model performance. Based on the definition provided in the scikit-learn library's `accuracy_score` function [8], accuracy is computed as the fraction of correct predictions made by the model. Formally, for a dataset with n_{samples}, accuracy is defined as:

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i),$$

where:

- y_i represents the true label of the i-th sample,
- $\hat{y}_i$ is the predicted label for the i-th sample,
- $1(x)$ is the indicator function that equals 1 if x is true, and 0 otherwise.

This metric evaluates the proportion of samples for which the predicted labels match the true labels. It is particularly useful for binary classification tasks such as the seismic event classification undertaken in this study.

2.7 Computational Environment

All computations were conducted on the following hardware configuration:

- **Processor**: Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz (6 cores, 12 threads)
- **Memory**: 64 GB RAM
- **GPU**: NVIDIA GeForce MX250
- **System Architecture**: 64-bit operating system, x64-based processor

The CNN model training and inference tasks were performed on the GPU to leverage the parallel processing capabilities for faster computations, while the PCA-SVM pipeline, including dimensionality reduction and classification, was executed on the CPU. This division reflects typical resource allocation strategies for deep learning and classical machine learning tasks, respectively.

3 CNN

Convolutional Neural Networks (CNNs) are a class of deep neural networks renowned for their effectiveness in capturing spatial dependencies within data. Using convolutional layers, CNNs can detect patterns or features in input data through filters or kernels. These networks excel in tasks requiring hierarchical feature extraction, such as image classification and object detection, which makes them suitable for seismogram classification.

In general, deeper CNN models tend to perform better than shallower ones; however, this improvement often comes at the cost of longer training times and increased computational complexity [9]. In this study, we explored various model architectures to balance performance metrics such as accuracy and training speed

3.1 Model Architecture and Results

All of our CNNs were composed of pairs of 2D convolutional layers activated by the ReLU function, followed by a 2D Max-Pooling layer. These layers process spectrograms, which are 2D representations of waveforms with time on one axis and frequency on the other. Mathematically, a 2D convolutional layer applies a convolution operation to the input spectrogram $\mathbf{S} \in \mathbb{R}^{F \times T}$, where F is the number of frequency bins and T is the number of time steps. The convolution operation for a single filter $\mathbf{K} \in \mathbb{R}^{f \times t}$ is defined as:

$$\mathbf{Z}(i,j) = \sum_{p=0}^{f-1} \sum_{q=0}^{t-1} \mathbf{S}(i+p, j+q) \cdot \mathbf{K}(p,q) + b, \tag{2}$$

where $\mathbf{Z}$ is the output feature map, and b is the bias term. The filter slides across the spectrogram, capturing local features such as frequency patterns and temporal changes.

Max-Pooling layers are used to reduce the spatial dimensions of the feature maps while retaining the most salient information. For a pooling window of size $k \times k$, the Max-Pooling operation is defined as:

$$\mathbf{P}(i,j) = \max_{p=0}^{k-1} \max_{q=0}^{k-1} \mathbf{Z}(i+p, j+q), \tag{3}$$

where $\mathbf{P}$ is the pooled output. Max-Pooling reduces computational complexity, mitigates overfitting, and ensures translational invariance by focusing on the most prominent features within local regions.

The processed feature maps are flattened into a 1D tensor and passed through fully connected layers for classification. The ReLU activation function is defined as:

$$\text{ReLU}(x) = \max(0, x), \tag{4}$$

and the final layer uses the Sigmoid activation function:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}, \tag{5}$$

which maps the output into the range $[0, 1]$, making it suitable for binary classification.

For optimization, we employed the Adam optimizer, which adapts the learning rate during training, and the Binary Cross Entropy (BCE) with Logits loss function, defined as:

$$\text{BCE} = -\frac{1}{m} \sum_{i=1}^{m} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right], \tag{6}$$

where y_i is the true label, $\hat{y}_i$ is the predicted probability, and m is the batch size [10].

3.1.1 Rationale for Architectural Choices

The 2D convolutional layers were chosen because spectrograms are 2D representations of waveforms, where convolution can effectively capture spatial correlations in both time and frequency domains. Max-Pooling was selected for its ability to reduce the spatial dimensions of feature maps while retaining the

most important features, thereby reducing computational cost and overfitting. Alternatives such as average pooling or global pooling could also have been considered, but Max-Pooling was preferred due to its robustness in preserving sharp features.

The specific model architecture was determined through experimentation, with a key focus on balancing accuracy and training speed. The experimentation process and results are presented in the following section.

3.1.2 Results

We began with a basic model consisting of three convolutional layers with Max-Pooling, followed by a flattening layer and two fully connected linear layers (Figure 4). Using this model, we achieved an accuracy of 51.7% on the dataset, demonstrating the feasibility of using CNNs for seismic event classification. Further refinements to the architecture, including modifications to layer dimensions and activation functions, led to improved performance, as detailed subsequently.

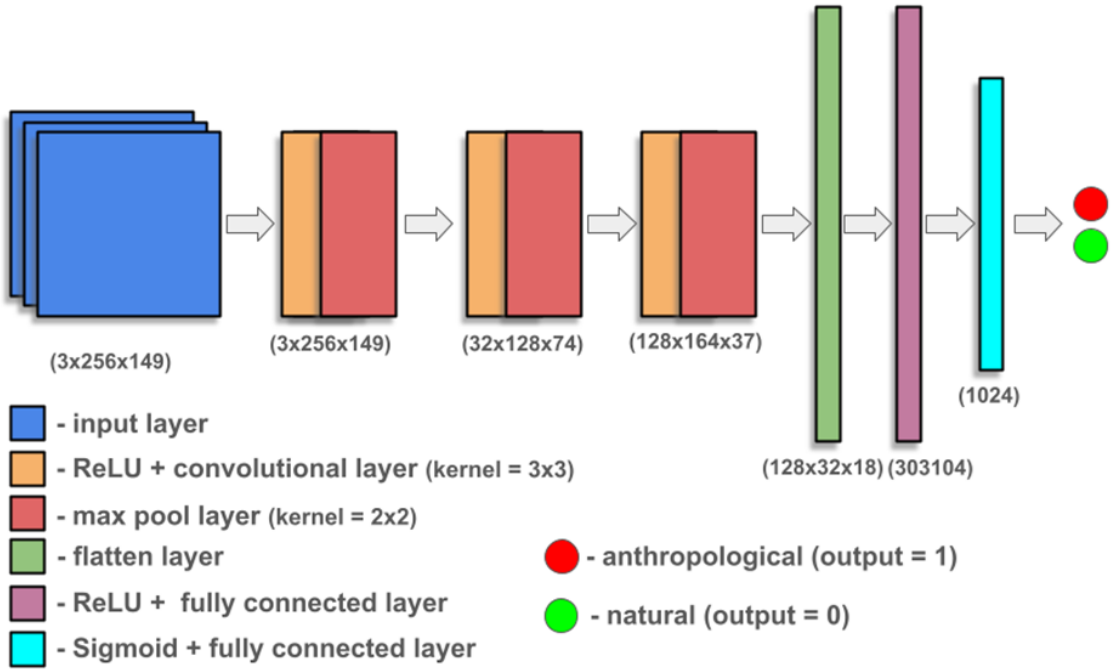

Figure 4: Schema of the test CNN

Expanding the output of the last convolutional layer resulted in a slight increase in the accuracy of the model, approximately 5%, when increasing the output tensor size from $128 \times 32 \times 18$ to $512 \times 30 \times 16$. These specific dimensions were determined by the size of the convolutional kernels, the strides, and the padding used in the convolutional layers. The output tensor size after a convolution operation is calculated as:

$$H_{\text{out}} = \frac{H_{\text{in}} - K + 2P}{S} + 1, \quad W_{\text{out}} = \frac{W_{\text{in}} - K + 2P}{S} + 1, \tag{7}$$

where H_{in} and W_{in} are the height and width of the input tensor, K is the kernel size, P is the padding, and S is the stride. The depth of the output tensor corresponds to the number of filters used in the convolutional layer. For instance, increasing the number of filters from 128 to 512 increases the representational capacity of the layer, enabling the model to capture more complex patterns in the data.

However, this improvement in accuracy came at the cost of significantly slower training speeds due to the increased computational complexity introduced by larger tensor dimensions and more filters.

As a test, we implemented a simpler CNN model with only one convolutional layer followed by max pooling, then a flattening layer, and a single fully connected linear layer (Figure 5). This simpler model had a significantly reduced number of parameters and was faster to train. Despite its simplicity, the model achieved an accuracy of 83.2%, demonstrating the feasibility of using shallow architectures for this classification task.

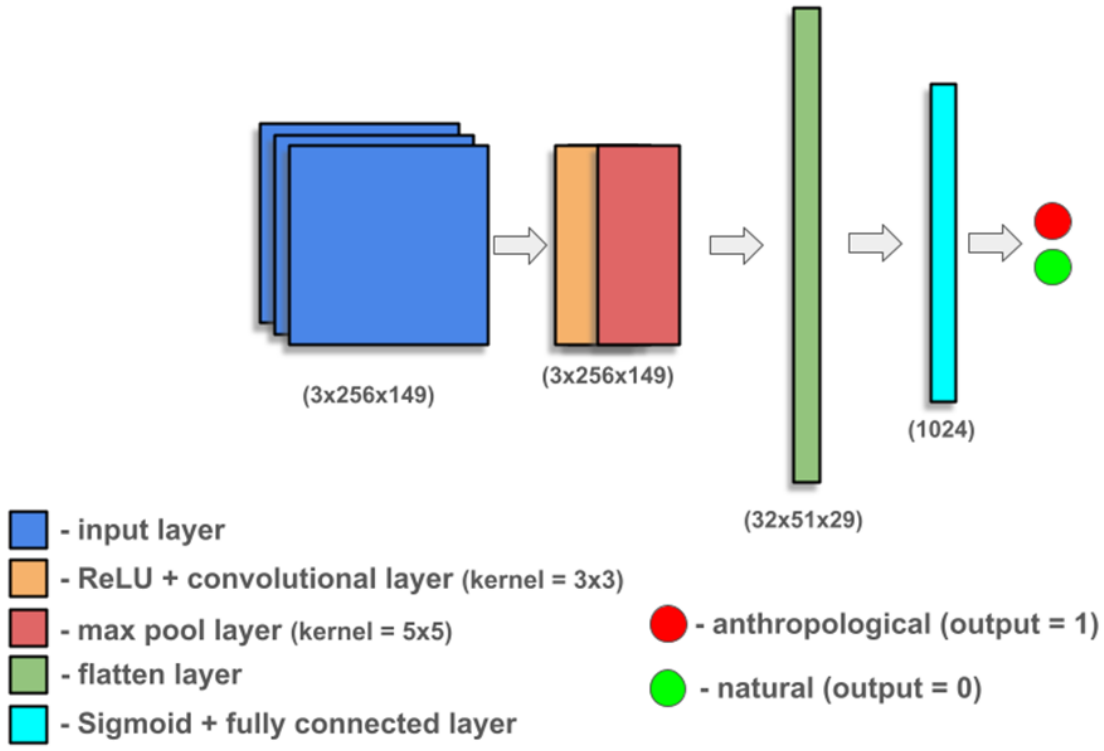

Figure 5: Schema of the single convolutional layer CNN

Given the drastic improvement in overall accuracy and training speed of the test model, we incorporated some features of this simpler model into our initial design. We ended up with a refined model consisting of three convolutional layers with max pooling, followed by a flattening layer and a single fully connected linear layer (Figure 6). The specific dimensions for each layer were carefully chosen based on empirical testing and the trade-off between accuracy and computational cost.

This refined model achieved an overall accuracy of 97.1%, significantly enhancing the model's functionality while maintaining a moderate training speed.

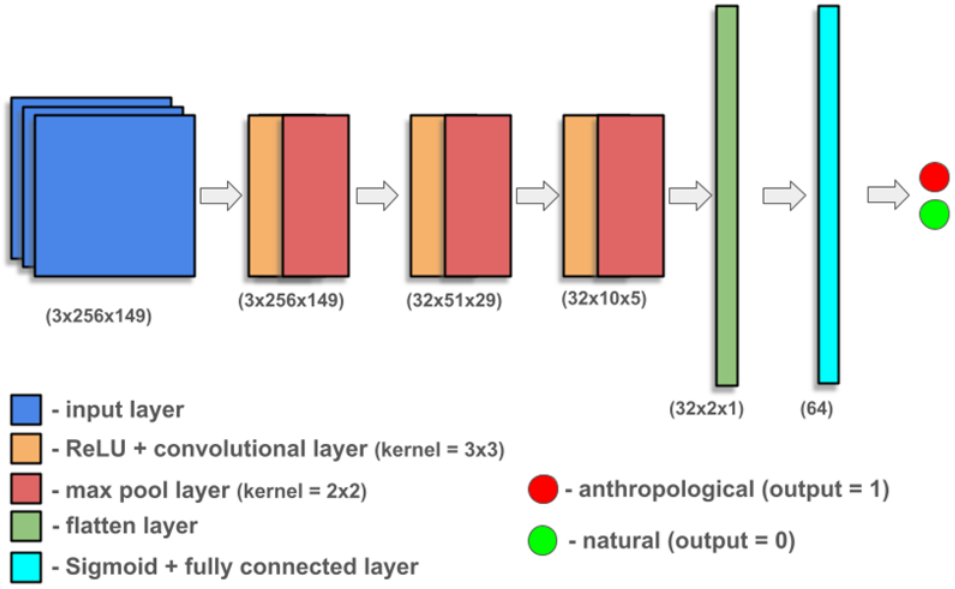

Figure 6: Schema of the final CNN version

4 PCA-SVM

In the context of seismic event classification, the combination of Principal Component Analysis (PCA) [11],[12] and Support Vector Machines (SVM) [13], optimized via Stochastic Gradient Descent (SGD) [14],[15], provides a structured approach for addressing the challenges of high-dimensional data. PCA reduces the dimensionality by retaining the most relevant features, thereby simplifying the feature space and potentially improving computational efficiency. However, the success of this framework depends on the ability of PCA to capture patterns critical for classification and the appropriateness of SVM as a classifier for the transformed data.

4.1 Feature Extraction

After normalization by using (1), the waveform data across all channels are concatenated to form a single feature vector for each event:

$$\mathbf{z}_i = \text{vec}(\mathbf{X}'_i) \in \mathbb{R}^{NC}. \tag{8}$$

This vectorization consolidates the temporal and spatial information from the seismic event into a unified representation, facilitating efficient processing in subsequent analysis stages.

4.2 Principal Component Analysis (PCA)

Given the high dimensionality of the seismic data, PCA is used to reduce the feature space while preserving the most significant variance in the data. For a dataset $\mathbf{Z} \in \mathbb{R}^{m \times d}$, where m is the number of samples and d is the dimensionality of each sample, PCA projects the data onto a lower-dimensional subspace:

$$\mathbf{Z}_{\text{PCA}} = \mathbf{Z}\mathbf{U}, \tag{9}$$

where $\mathbf{U} \in \mathbb{R}^{d \times k}$ contains the top k eigenvectors of the covariance matrix $\mathbf{C} = \frac{1}{m}\mathbf{Z}^T\mathbf{Z}$. The selection of k is critical; the goal is to choose k as small as possible while retaining sufficient variance and ensuring that classification accuracy is not compromised. The explained variance ratio, which quantifies the proportion

of total variance captured by the selected components, is calculated as:

$$\text{Explained Variance Ratio} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{j=1}^{d} \lambda_j}, \tag{10}$$

where λ_i are the eigenvalues corresponding to the eigenvectors in $\mathbf{U}$. By retaining components that capture a high explained variance ratio, PCA reduces dimensionality, which can help mitigate some challenges associated with high-dimensional data. However, the success of this approach depends on the assumption that the set of functions to be learned aligns well with the second-order statistics of the data, such as covariance. In practice, this alignment is not guaranteed, and the effectiveness of PCA in reducing dimensionality without sacrificing performance varies depending on the specific application.

4.3 Support Vector Machine (SVM) Classifier

4.3.1 Objective Function

SVMs are supervised learning models commonly used for classification tasks, particularly when the dataset is of small to medium size and the classes are linearly or non-linearly separable. They are valued for their ability to find an optimal hyperplane that maximizes the margin between classes, making them effective in various applications. However, their performance can be limited in very high-dimensional spaces or with large, noisy datasets, where selecting an appropriate kernel becomes critical.

In this framework, a linear SVM is trained on the PCA-transformed data to classify seismic events. The objective function of the SVM aims to find the optimal hyperplane that separates the classes with the maximum margin, formulated as:

$$\min_{\mathbf{w},b} [\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} \max(0, 1 - y_i(\mathbf{w}^T\mathbf{z}_i + b))], \tag{11}$$

where $\mathbf{w}$ is the weight vector, b is the bias term, $y_i \in \{-1, 1\}$ are the class labels, $\mathbf{z}_i$ is the feature vectors, and C is the regularization parameter that controls the trade-off between maximizing the margin and minimizing classification errors.

In cases where the data points cannot be linearly separated in the original feature space (e.g., if one class occupies a subvolume of the feature space while the other lies outside it), SVMs employ the kernel trick. This technique maps the original feature space to a higher-dimensional space, enabling linear separation in the transformed space. The kernel function, such as a radial basis function (RBF) or polynomial kernel, implicitly computes this mapping without explicitly transforming the data, preserving computational efficiency.

By leveraging the kernel trick, SVMs are robust in handling non-linear relationships, provided that the choice of kernel aligns with the underlying data structure. This flexibility makes them particularly well-suited for tasks where linear separation in the original space is not feasible. However, it is important to note that the success of this approach relies on the relationship between the features and the separability of the classes in the transformed space.

4.3.2 Stochastic Gradient Descent (SGD)

To optimize the SVM's objective function, Stochastic Gradient Descent (SGD) is employed due to its computational efficiency when handling large datasets. Unlike traditional gradient descent, which calculates the gradient over the entire dataset at each iteration, SGD updates the model parameters using the gradient of a single sample (or a small batch of samples). This approach significantly reduces memory requirements and accelerates computations.

The update rules for the weight vector $\mathbf{w}$ and the bias term b at each iteration t are given by:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\nabla_{\mathbf{w}}\ell(\mathbf{w}_t, b_t, \mathbf{z}_i, y_i), \tag{12}$$

$$b_{t+1} = b_t - \eta\nabla_b\ell(\mathbf{w}_t, b_t, \mathbf{z}_i, y_i), \tag{13}$$

where:

- η is the learning rate, controlling the step size for the update.
- $\ell(\mathbf{w}, b, \mathbf{z}_i, y_i) = \max(0, 1 - y_i(\mathbf{w}^T \mathbf{z}_i + b))$ is the hinge loss function for a single data point $(\mathbf{z}_i, y_i)$.
- $\nabla_{\mathbf{w}}$ and ∇_b represent the gradients of the loss function with respect to $\mathbf{w}$ and b, respectively.

This iterative process introduces stochastic noise into the optimization, which can lead to faster convergence in practice, although it may cause fluctuations around the optimal solution. Despite this, the lightweight computation per iteration makes SGD a practical choice for large-scale datasets, particularly when computational resources are constrained.

4.4 Model Selection and Parameter Tuning

Model selection involves optimizing hyper-parameters to maximize the model's performance while maintaining computational efficiency. For the PCA-SVM framework, the two primary hyper-parameters are:

1. **Number of PCA Components (k):** This determines the dimensionality of the PCA-transformed feature space. The goal is to select k such that at least 80% of the data's variance is retained, while keeping k as small as possible to reduce computational costs. This trade-off ensures that the reduced feature space is both representative and computationally efficient.
2. **Regularization Parameter (C or α):** For the linear SVM classifier, the regularization parameter C controls the trade-off between maximizing the margin and minimizing classification errors. The equivalent parameter $\alpha = \frac{1}{C}$ for the SGD classifier is optimized to maximize classification accuracy while avoiding overfitting.

4.4.1 Number of Components Optimization

The number of PCA components k is determined first. We evaluate values of k in the range $[30, 50]$, ensuring that each configuration covers at least 80% of the explained variance. Among configurations meeting this threshold, the optimal k is chosen to minimize the number of components without sacrificing classification accuracy or variance representation. This approach balances computational efficiency and model performance.

4.4.2 Alpha Optimization

Once k is fixed, the regularization parameter α is optimized. A grid search is performed over values of α logarithmically spaced between 10^{-5} and 10^{3}. This ensures a wide exploration of the trade-off between regularization strength and model complexity. The optimal α is selected to maximize accuracy on the validation set, while maintaining generalizability to unseen data.

4.4.3 Sequential Optimization Approach

The optimization of k and α is performed sequentially, not jointly. First, k is selected based on the explained variance and computational efficiency criteria. Then, for the fixed value of k, the parameter α is optimized. This approach avoids the computational overhead of a two-dimensional grid search in the $k - C$ space while maintaining a clear hierarchy in the optimization process.

This sequential strategy ensures that the PCA-SVM framework remains computationally tractable while delivering competitive performance in classifying seismic events.

4.5 Results

With our dataset and the mentioned setup for PCA-SVM, we could achieve 91.0% accuracy for both the train and also the test phase while the best value for α was 0.000183 and the optimal number of components was 43 for 50000 waveforms (Figure 7).

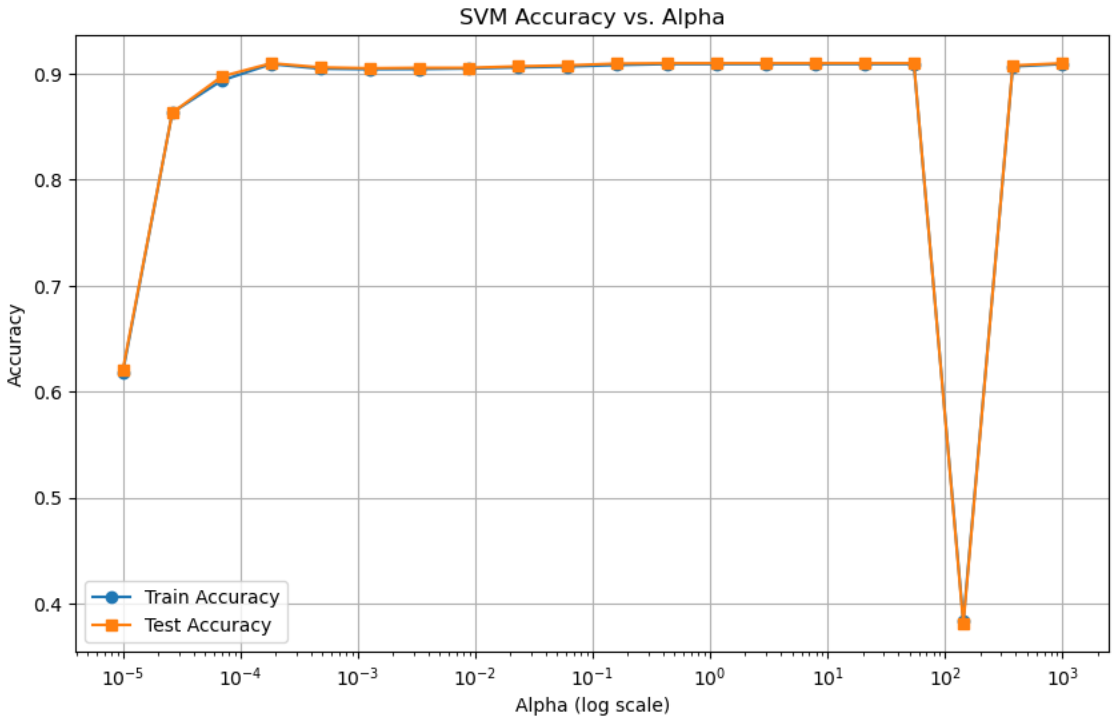

Figure 7: PCA-SVM with the optimal number of components accuracy vs alpha for 50000 waveforms.

5 Conclusion

This study highlights the trade-offs between CNNs and PCA-SVM in seismic event classification. CNN is recommended when accuracy is the primary concern and computational resources are readily available - increased accuracy comes at the cost of significantly higher computational cost for training and longer processing times. On the other hand, PCA-SVM is more suitable for scenarios where computational resources are limited or when rapid model deployment is required. It is particularly advantageous for datasets with more than 25,000 waveforms, where the running time is substantially lower compared to CNN, often by an order of magnitude (Figure 8). For applications requiring real-time or near-real-time seismic event classification, PCA-SVM provides a practical and efficient solution.

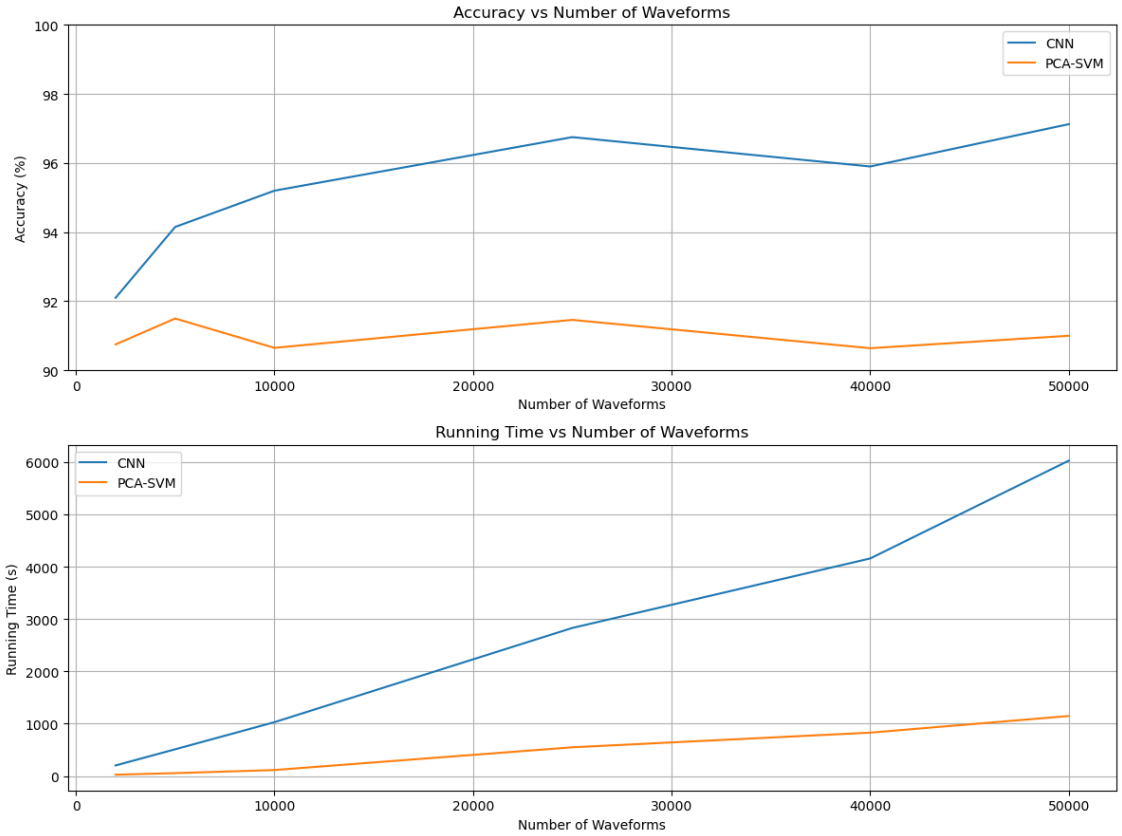

Figure 8: Performace comparison between CNN and PCA-SVM.

5.1 PCA-SVM Pipeline Integration

The PCA-SVM pipeline consists of the following steps:

1. Normalize waveform data.
2. Apply PCA to reduce dimensionality with the optimal number of components k.
3. Train an SVM classifier using SGD with the optimal regularization parameter α.
4. Evaluate the trained model on the test data.
5. Save the trained model for deployment in real-time seismic event classification tasks.

This pipeline ensures a robust and efficient approach to classifying seismic events, combining the strengths of PCA for dimensionality reduction and SVM for robust classification under high-dimensional settings. With these carefully designed steps, the framework achieves a balance between computational efficiency and classification accuracy, making it a valuable tool for seismic monitoring and analysis.

Code and Data Availability

The dataset and code in Python3 used in this study are openly available and can be accessed through the following GitHub repository: `https://github.com/markgrmek/project_c`. The repository includes the preprocessed seismic event dataset, the complete implementation of both the Convolutional Neural Network (CNN) and Principal Component Analysis-Support Vector Machine (PCA-SVM) pipelines, and their trained models.

Acknowledgments

Special thanks to our supervisor, Prof. Dr.-Ing. Rupert Klein, as well as the kind and supportive members of GFZ-Potsdam.

References

[1] Michaela Schwardt et al. “Natural and Anthropogenic Sources of Seismic, Hydroacoustic, and Infrasonic Waves: Waveforms and Spectral Characteristics (and Their Applicability for Sensor Calibration)”. In: *Surveys in Geophysics* 43 (2022), pp. 1265–1361.

[2] Jari Kortström, Marja Uski, and Timo Tiira. “Automatic classification of seismic events within a regional seismograph network”. In: *Computers and Geosciences* 87 (2016), pp. 22–30. ISSN: 0098-3004. DOI: https://doi.org/10.1016/j.cageo.2015.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0098300415300832.

[3] Christian Sippl et al. *Catalogue of Earthquake Hypocenters for Northern Chile from 2007-2021 using IPOC (plus auxiliary) seismic stations. GFZ Data Services.* 2023. URL: https://doi.org/10.5880/GFZ.4.1.2023.004 (visited on 11/22/2024).

[4] Christian Sippl et al. “The Northern Chile forearc constrained by 15 years of permanent seismic monitoring”. In: *Journal of South American Earth Sciences* 126 (2023).

[5] *ObsPy Documentation (1.4.1)*. 2024. URL: https://docs.obspy.org/ (visited on 11/22/2024).

[6] Scott Halbert. *SEED Format definitions*. 2012.286. EarthScope Consortium. 2012.

[7] Lisa Linville, Kristine Pankow, and Timothy Draelos. “Deep learning models augment analyst decisions for event discrimination.” In: *Geophysical Research Letters* 46 (2019). DOI: 10.1029/2018GL081119.

[8] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems* (2012).

[10] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[11] Jinkun Cheng, Ke Chen, and Mauricio D. Sacchi. “Robust principle component analysis (RPCA) for seismic data denoising”. In: *GeoConvention 2015: New Horizons*. 2015.

[12] Jeffrey F. Tan, Robert R. Stewart, and Joe Wong. “Classification of microseismic events via principal component analysis of trace statistics”. In: *CSEG RECORDER* (2010).

[13] Tingting Wang et al. “Using Artificial Intelligence Methods to Classify Different Seismic Events”. In: *Seismological Research Letters* 94.1 (2022), pp. 1–16. DOI: 10.1785/0220220055.

[14] Jahongir Azimjonov and Taehong Kim. “Stochastic gradient descent classifier-based lightweight intrusion detection systems using the efficient feature subsets of datasets”. In: *Expert Systems with Applications* 237 (2024), p. 121493. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2023.121493.

[15] P Netrapalli. “Stochastic Gradient Descent and Its Variants in Machine Learning”. In: *J Indian Inst Sci* 99 (2019), pp. 201–213. DOI: https://doi.org/10.1007/s41745-019-0098-4.

[16] Céline Hourcade, Mickael Bonnin, and Éric Beucler. “New CNN based tool to discriminate anthropogenic from natural low magnitude seismic events”. In: *Geophysical Journal International* 339 (2023), pp. 22–34.

[17] Hyeongki Ahn et al. “Imbalanced Seismic Event Discrimination Using Supervised Machine Learning”. In: *Sensors* 22 (2022).

[18] Noah Luna. *Identifying Seismic Waves with Convolutional Neural Networks [Part I]*. 2019. URL: https://ngrayluna.github.io/post/p-phase-picker-tutorial_pi/ (visited on 11/22/2024).

Appendix

CNN Code

```
# Imports and Setup
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import numpy as np
import pandas as pd
import random
import os
from obspy.core import read
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from spectrogram_to_array import spectrogram
import time


# Loading Event Data

# Path setup
main_path = os.path.abspath("")
all_events_file_path = os.path.join(main_path, 'earthquakes_filtered.txt')  # all events
all_events = pd.read_csv(all_events_file_path, sep=',')

try:
    all_events.drop(columns=['Unnamed: 0'], inplace=True)  # automatically created column
except:
    pass

all_events.head()


# Prepare Dataset

# Preparing file list from geofon_waveforms folder
dataset_size = 50000
file_list = os.listdir(os.path.join(main_path, "geofon_waveforms"))
file_list = [int(file[:-6]) for file in file_list if file.endswith(".mseed")]

# Select random sample of N events from all files
file_list = random.sample(file_list, dataset_size)

# Train-test split
train_events, test_events = train_test_split(file_list, test_size=0.2, random_state=42)

# Merge with events data
train_events = pd.DataFrame(train_events, columns =['event_id']).merge(all_events,
↪  on='event_id')
test_events = pd.DataFrame(test_events, columns =['event_id']).merge(all_events,
↪  on='event_id')
```

```python
# Dataset Class

class event_dataset(Dataset):
    def __init__(self, dataset_type: str, transform=None):
        if dataset_type not in ['train', 'test']:
            raise KeyError("dataset_type has to be 'train' or 'test'")

        self.dataframe = train_events if dataset_type == "train" else test_events
        self.data_directory = "geofon_waveforms"
        self.transform = transform

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        row = self.dataframe.iloc[idx]
        event_id, label = int(row['event_id']), int(row['category'])

        # Waveform fetch
        file_name = f"{event_id}.mseed"
        waveform = read(os.path.join(main_path, self.data_directory, file_name))

        # Spectrogram preparation
        spec_data = [spectrogram(data=trace.data, samp_rate=40.0, log=True, wlen=2,
        ↪  per_lap= 0.5, dbscale=False)[0] for trace in waveform]
        spec_data = np.stack(spec_data, axis=0, dtype=np.float32)

        # Convert to torch tensors
        label = torch.tensor(label, dtype=torch.int64)
        spec_data = torch.from_numpy(spec_data)

        sample = {'label': label, 'data': spec_data}
        if self.transform:
            sample['data'] = self.transform(sample['data'])

        return sample


# Normalization Function

def normalize_tensor(tensor):
    return (tensor - tensor.mean()) / tensor.std()


# CNN Model

class Simple_CNN_v2(nn.Module):
    def __init__(self):
        super(Simple_CNN_v2, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1,
        ↪  padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1,
        ↪  padding=1)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1,
        ↪  padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(33728, 1024)
        self.fc2 = nn.Linear(1024, 1)

    def forward(self, x):
```

```
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(-1, 33728)
        x = F.relu(self.fc1(x))

        return self.fc2(x)  # Returning logits directly


# Data Loaders

batch_size = 4
train_loader = DataLoader(
    dataset=event_dataset(dataset_type='train', transform=normalize_tensor),
    batch_size=batch_size,
    shuffle=True,
    num_workers=0
    )

test_loader = DataLoader(
    dataset=event_dataset(dataset_type='test', transform=normalize_tensor),
    batch_size=batch_size,
    shuffle=True,
    num_workers=0
    )


# Training Setup

# Model, loss, optimizer setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Simple_CNN_v2().to(device)

loss_function = nn.BCEWithLogitsLoss().to(device)
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)


# Training and Validation Functions

def train_one_epoch(N_batch_stats: int):
    model.train(True)
    running_loss = 0.0
    running_accuracy = 0.0

    start_time = time.time()  # Start time

    for batch_idx, sample in enumerate(train_loader):
        labels = sample['label'].to(device)
        data = sample['data'].to(device)

        optimizer.zero_grad()

        labels_logit = model(data).squeeze()
        labels_pred = torch.round(torch.sigmoid(labels_logit))

        correct = torch.sum(labels == labels_pred).item()
        running_accuracy += correct / batch_size

        loss = loss_function(labels_logit, labels.float())
        running_loss += loss.item()
```

```
            loss.backward()
            optimizer.step()

            if batch_idx % N_batch_stats == N_batch_stats - 1:
                avg_loss = running_loss / N_batch_stats
                avg_acc = (running_accuracy / N_batch_stats) * 100
                print(f'Batch {batch_idx + 1}, Loss: {avg_loss:.3f}, Accuracy:
                ↪  {avg_acc:.1f}%')

                running_loss = 0.0
                running_accuracy = 0.0

        end_time = time.time()  # End time
        elapsed_time = end_time - start_time
        print(f"Training Time for one epoch: {elapsed_time:.2f} seconds")


    def validate_one_epoch():
        model.train(False)
        running_loss = 0.0
        running_accuracy = 0.0

        start_time = time.time()  # Start time

        for i, sample in enumerate(test_loader):
            true_labels = sample['label'].to(device)
            inputs = sample['data'].to(device)

             with torch.no_grad():
                labels_logit = model(inputs).squeeze()
                labels_pred = torch.round(torch.sigmoid(labels_logit))

                correct = torch.sum(true_labels == labels_pred).item()
                running_accuracy += correct / batch_size
                loss = loss_function(labels_logit, true_labels.float())
                running_loss += loss.item()

        avg_loss = running_loss / len(test_loader)
        avg_acc = (running_accuracy / len(test_loader)) * 100
        print(f'Val Loss: {avg_loss:.3f}, Val Accuracy: {avg_acc:.1f}%')
        print('**************************************************')

        end_time = time.time()  # End time
        elapsed_time = end_time - start_time
        print(f"Validation Time for one epoch: {elapsed_time:.2f} seconds")


    # Training Loop

    # Initialize model, optimizer, and loss function
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = Simple_CNN_v2().to(device)

    loss_function = nn.BCEWithLogitsLoss().to(device)
    optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)

    num_epochs = 4
    N_batch_print_stats = 100

    for epoch_idx in range(num_epochs):
```

```
    print(f'Epoch: {epoch_idx + 1}\n')
    train_one_epoch(N_batch_stats=N_batch_print_stats)
    validate_one_epoch()

print('Finished training')


# Saving the Model

PATH = './simple_cnn.pth'
torch.save(model.state_dict(), PATH)
```

PCA-SVM Code

```
# Imports and Setup
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import numpy as np
import pandas as pd
import random
import os
from obspy.core import read
from sklearn.model_selection import train_test_split
from spectrogram_to_array import spectrogram
import time
from tqdm import tqdm
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA
from sklearn.linear_model import SGDClassifier
import matplotlib.pyplot as plt


# Loading Event Data

# Path setup
main_path = os.path.abspath("")
all_events_file_path = os.path.join(main_path, 'earthquakes_filtered.txt')  # all events
all_events = pd.read_csv(all_events_file_path, sep=',')

try:
    all_events.drop(columns=['Unnamed: 0'], inplace=True)  # automatically created column
    ↪  (idk why)
except:
    pass

all_events.head()


# Prepare Dataset

# Preparing file list from geofon_waveforms folder
dataset_size = 2000
file_list = os.listdir(os.path.join(main_path, "geofon_waveforms"))
file_list = [int(file[:-6]) for file in file_list if file.endswith(".mseed")]
```

```
# Select random sample of N events from all files
file_list = random.sample(file_list, dataset_size)

# Train-test split
train_events, test_events = train_test_split(file_list, test_size=0.2, random_state=42)

# Merge with events data
train_events = pd.DataFrame(train_events, columns=['event_id']).merge(all_events,
↪  on='event_id')
test_events = pd.DataFrame(test_events, columns=['event_id']).merge(all_events,
↪  on='event_id')


# Helper Functions

def normalize_tensor(tensor):
    mean = tensor.mean()
    std = tensor.std()
    return (tensor - mean) / (std + 1e-6)  # Add epsilon to avoid division by zero

class EventDataset(Dataset):
    def __init__(self, dataframe, data_directory, transform=None):
        self.dataframe = dataframe  # The DataFrame containing event IDs and labels
        self.data_directory = data_directory  # Directory containing waveform files
        self.transform = transform  # Optional preprocessing function

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        row = self.dataframe.iloc[idx]
        event_id, label = int(row['event_id']), int(row['category'])

        # Load waveform
        file_name = f"{event_id}.mseed"
        waveform_path = os.path.join(self.data_directory, file_name)
        waveform = read(waveform_path)

        # Concatenate waveform data from all channels
        waveform_data = np.concatenate([trace.data for trace in waveform],
        ↪  axis=0).astype(np.float32)

        # Apply transformation (e.g., normalization) if provided
        if self.transform:
            waveform_data = self.transform(waveform_data)

        return {'label': label, 'data': waveform_data}


# Define directories
data_directory = os.path.join(main_path, "geofon_waveforms")

# Define batch size
batch_size = 4  # Adjust this value as needed

# Create datasets and loaders
train_dataset = EventDataset(train_events, "geofon_waveforms",
                             transform=normalize_tensor)
test_dataset = EventDataset(test_events, "geofon_waveforms",
                            transform=normalize_tensor)
```

```
train_loader = DataLoader(train_dataset, batch_size=batch_size,
                              shuffle=True, num_workers=0, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
                             shuffle=True, num_workers=0, pin_memory=True)


# Feature Extraction
start_time = time.time()

print("Extracting features...")
train_features, train_labels = [], []
test_features, test_labels = [], []

for sample in tqdm(train_loader, desc="Processing Training Data"):
    data = sample['data']
    labels = sample['label']
    train_features.append(data.numpy())
    train_labels.append(labels.numpy())

train_features = np.concatenate(train_features)
train_labels = np.concatenate(train_labels)

for sample in tqdm(test_loader, desc="Processing Test Data"):
    data = sample['data']
    labels = sample['label']
    test_features.append(data.numpy())
    test_labels.append(labels.numpy())

test_features = np.concatenate(test_features)
test_labels = np.concatenate(test_labels)

end_time = time.time()

print(f"Feature extraction completed in: {end_time - start_time:.2f}
     seconds")


# Optimize PCA Components
results = []
min_variance_ratio = 0.80

print("\nOptimizing PCA components...")
start_time_pca = time.time()  # Start timing PCA component optimization
for n_components in range(30, 50):
    step_start = time.time()  # Start timing for each iteration

    # Apply PCA
    pca = PCA(n_components=n_components)
    train_features_pca = pca.fit_transform(train_features)
    test_features_pca = pca.transform(test_features)
    explained_variance_ratio = np.sum(pca.explained_variance_ratio_)

    # Train SVM
    svm = SGDClassifier(loss='hinge', max_iter=1000000000, tol=1e-3, warm_start=False,
    ↪  n_jobs=-1)
    svm.fit(train_features_pca, train_labels)

    # Evaluate Accuracy
    accuracy = accuracy_score(test_labels, svm.predict(test_features_pca))
    results.append((n_components, accuracy, explained_variance_ratio))
```

```
    step_end = time.time()  # End timing for this iteration
    print(f"Components: {n_components}, Accuracy: {accuracy:.2f}, Variance Ratio:
    ↪  {explained_variance_ratio:.2f}, Time: {step_end - step_start:.2f} seconds")

end_time_pca = time.time()  # End timing PCA component optimization
print(f"PCA components optimization completed in: {end_time_pca - start_time_pca:.2f}
↪  seconds")

results_df = pd.DataFrame(results, columns=['n_components', 'accuracy',
↪  'explained_variance_ratio'])
filtered_results = results_df[results_df['explained_variance_ratio'] >=
↪  min_variance_ratio]

if not filtered_results.empty:
    optimal_row = filtered_results.loc[filtered_results['accuracy'].idxmax()]
else:
    print("No configurations met the minimum variance threshold.")
    optimal_row = results_df.loc[results_df['accuracy'].idxmax()]

optimal_components = int(optimal_row['n_components'])
print(f"Optimal components: {optimal_components}")


# Final PCA and Alpha Optimization
pca = PCA(n_components=optimal_components)
train_features_pca = pca.fit_transform(train_features)
test_features_pca = pca.transform(test_features)

alpha_values = np.logspace(-5, 3, 20)
train_accuracies, test_accuracies = [], []
best_alpha, best_test_accuracy = None, 0

print("\nOptimizing SGDClassifier alpha parameter...")
start_time_alpha = time.time()  # Start timing alpha optimization
for idx, alpha in enumerate(alpha_values):
    step_start = time.time()  # Start timing for each alpha iteration

    # Train SVM
    svm = SGDClassifier(loss='hinge', alpha=alpha, max_iter=1000000000, tol=1e-3,
    ↪  warm_start=False, n_jobs=-1)
    svm.fit(train_features_pca, train_labels)

    # Evaluate Accuracy
    train_acc = accuracy_score(train_labels, svm.predict(train_features_pca))
    test_acc = accuracy_score(test_labels, svm.predict(test_features_pca))
    train_accuracies.append(train_acc)
    test_accuracies.append(test_acc)

    # Update best alpha
    if test_acc > best_test_accuracy:
        best_test_accuracy = test_acc
        best_alpha = alpha

    step_end = time.time()  # End timing for this alpha iteration
    print(f"Alpha: {alpha:.6f}, Train Accuracy: {train_acc:.2f}, Test Accuracy:
    ↪  {test_acc:.2f}, Time: {step_end - step_start:.2f} seconds")

end_time_alpha = time.time()  # End timing alpha optimization
print(f"Alpha optimization completed in: {end_time_alpha - start_time_alpha:.2f}
↪  seconds")
```

```python
# Print Best Alpha
print(f"\nBest alpha: {best_alpha:.6f} with Test Accuracy: {best_test_accuracy *
↪  100:.2f}%")

# Plot Accuracy vs Alpha
plt.figure(figsize=(10, 6))
plt.plot(alpha_values, train_accuracies, label='Train Accuracy', marker='o')
plt.plot(alpha_values, test_accuracies, label='Test Accuracy', marker='s')
plt.xscale('log')
plt.xlabel('Alpha (log scale)')
plt.ylabel('Accuracy')
plt.title('SVM Accuracy vs. Alpha')
plt.legend()
plt.grid(True)
plt.show()


# Save Final Model
final_model = SGDClassifier(loss='hinge', alpha=best_alpha,
                            max_iter=1000000000, tol=1e-3, warm_start=False, n_jobs=-1)
final_model.fit(train_features_pca, train_labels)
torch.save(final_model, "pca_svm_model.pth")
print("Final model saved.")
```