STT 3795: Final Project Report

Bio Samir Gbian[1], Kamen Damov[1], and Simon Langlois[1]

[1]Department of Mathematics and Statistics
[1]University of Montreal

April 28, 2024

GitHub Code Repository

1 Introduction

Language classification is a crucial task in many fields, from natural language technology to speech recognition and machine translation. The rise of big data and the diversity of languages spoken around the world have amplified the importance of having effective classification methods to process and analyze these data meaningfully.

In this context, evaluating and comparing the performance of classification algorithms is essential to identify the most suitable approaches for this complex task. This project aims to examine and compare the quality of some of the most commonly used classification algorithms in language classification. In particular, we focus on evaluating the performance of Support Vector Classifiers (SVC) and Random Forests Classifiers.

2 Objectives

The main objective of this project is to identify the strengths and weaknesses of each classification algorithm in the specific context of language classification. To achieve this, we will use representative datasets containing voice samples from different people (with varying genders and age groups) covering a variety of linguistic structures and features. We will assess the performance of each algorithm in terms of accuracy, precision, recall, and F1-score.

3 Description of Analyzed Data

3.1 Source

The data used in the project comes from the *Hugging Face* website, a reputable online platform that hosts datasets and shares pre-trained AI models. This allows us to have a reference to compare our model to, which is one of the main reasons why this dataset was chosen. The dataset, named *Common Language*, contains several audio files in WAV format (specifically 34045 files). These files are separated into three types: training, testing, and validation. Each audio represents a person of a certain gender (Male/Female) quoting a phrase or repeating words in a language. Forty-five different languages are spoken in these audios (See source in reference for more details on all the languages spoken).

3.2 Data Cleaning

The initial raw data attributes are: Client id, Path (Link to the audio file), Age (Speaker's age), Gender(Speaker's gender) and the language spoken in the audio.

Given that our project was solely focused on the classification of the spoken languages, We removed all the columns except the paths to the Language (our target label), and Path (the paths to the raw wav files containing the spoken languages). After listening to some audio files and plotting the spectral representation, we realized that most wav files had a few seconds of silence or some background sounds that weren't spoken language. In order to have data that is exclusively spoken language sound waves, and have higher quality data to feed to our classification models, we have to remove the non-spoken language sounds (see Algorithm 1).

As we can see in Figure 1 and Figure 2, only relevant data has been kept, as the white noise (at the beginning and at the end) has been removed. The new shape obtained contains less noise than the first shape. This allows for more reliable data.

Algorithm 1 Audio Cleaning Process

Initialize paths for raw and cleaned audio files
Prepare empty list for errors
function CLEAN_SOUND(audio)
    Define threshold to identify significant audio (e.g., 1%)
    Find the start and end of significant audio using the threshold
    Trim the audio outside the significant range
    return the trimmed audio
end function
for each audio file in the dataset **do**
    Read the audio file to obtain waveform data
    Apply noise reduction to the waveform
    Clean the audio using the clean_sound function
    Save the cleaned audio to the designated output path
    if An exception occurs **then**
        Log error with file details
    end if
end for

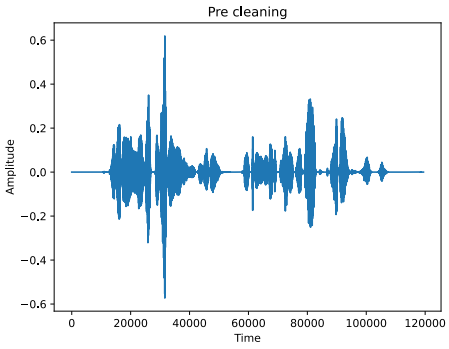


Figure 1: Pre cleaning shape

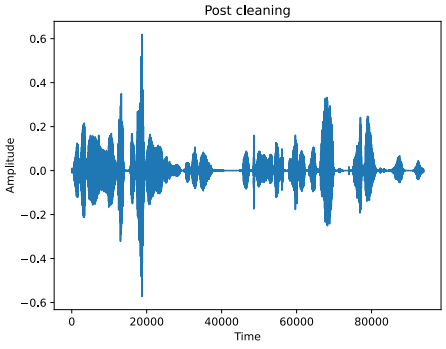
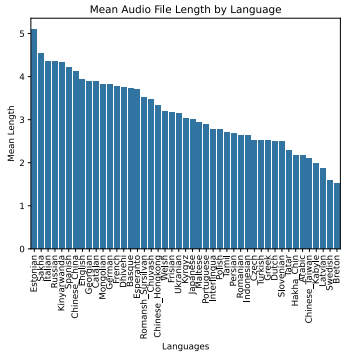


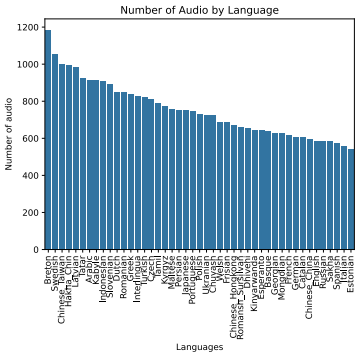
Figure 2: Post cleaning shape

3.3 Statistics

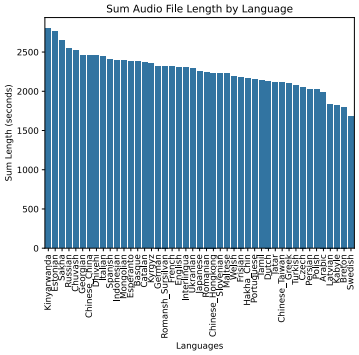
Before preprocessing our audio data into features for model training, we need to ensure that the number of samples per label is uniformly distributed, and that the audio lengths are also uniform. We looked at three statistics on the data: the number of audio files by language (Figure 3 (a)), the average length of audios by language (Figure 3 (b)), and the total length of audios by language (Figure 3 (c)).



(a) Figure 1: Audios mean



(b) Figure 2: Audios count



(c) Figure 3: Sum of audio length

Figure 3: Three aligned images with individual captions

By looking at the count, some languages are overrepresented in our data, which is not ideal because it can lead to a model that is biased towards these languages, potentially reducing its performance on underrepresented languages. Additionally, we observe that the average length of audio samples for each label varies significantly, ranging from 2 to 5 seconds. This variation needs to be considered when generating features from the soundwaves, ensuring that the windowing properly accommodates the length of each file. That being said, by looking at the total length of audios, which is the sum of seconds spoken for each language, we see that the difference between languages is slightly smaller. Overall, we see some unbalancing in our labels. Knowing this, we will need to stratify the dataset when doing the train/test splits so that each language is proportionately represented in the train and test sets, ensuring a more balanced and accurate evaluation of the model's capabilities across all languages.

3.4 Data Preprocessing

For the preprocessing of audio data, we use different features which are presented below. These features were picked to produce a holistic representation of spoken languages, covering a wide range of language features. We applied the windowing method to all our audio samples which is particularly important when dealing with audio data because sound characteristics can vary significantly over time. By breaking the audio into smaller pieces, we can analyze each segment more effectively. In our case, we decomposed our audio data using 512 samples from each audio file. With a sample rate (number of samples of audio carried per second, measured in Hz) of 16000 we get windows of length 0.032 milliseconds. All the following features will be extracted from audio data which was preprocessed on these specs.

MFCCs: The Mel Frequency Cepstral Coefficients (MFCCs) are criticals in the sense that it allows capturing the different aspects that are unique to each spoken language. It is much easier to differentiate languages by using MFCCs. To obtain the MFCCs, we first found the spectrum of power as a function of our audio file's frequencies from a Fast Fourier Transformation (FFT). Then, it was necessary to find the Mel Filter Banks (MFB) of the spectrum from which we obtain the frequency energies (weighted energy) passing through each filter of the bank. A logarithmic function will be applied to these energies. Finally, by applying a Discrete Cosine Transformation on the log-transformed energies, we obtain the MFCCs. We set the number of MFCCs to be 25, as recommended for spoken language. We yield, for each data point, a matrix $M \in \mathbb{R}^{25\times n}$, where n is the number of windows. We then compute a vector $v \in \mathbb{R}^{25}$ from M, where each element is this mean of an MFCC. These 25 features are added to the final attribute matrix.

Spectral Centroid: The spectral centroid indicates the center of mass of the sound's spectrum, providing a way to characterize the brightness of a sound. It is calculated as the weighted mean of the frequencies present in the signal, with their magnitudes as the weights. We yield a vector $v \in \mathbb{R}^n$ where n is the number of windows, and where each element corresponds to the spectral centroid of a specific window. To have a standardized format to add in our attribute matrix, we compute mean, median, standard deviation, $25th$ percentile, $75th$ percentile, maximum value, and minimum value, from v for each data point. This transformation is applied to all the spectral features (spectral rolloff and spectral bandwidth).

Spectral Rolloff: The spectral rolloff point indicates the frequency below which a specified percentage of the total energy of the spectrum is located, typically used to distinguish between harmonic (tonal) and noisy sounds. It serves as a way to capture the shape of the sound's spectrum, particularly its high-frequency content.

Spectral Bandwidth: Spectral bandwidth quantifies the width of the frequency band where most of the sound's energy is concentrated, reflecting the sound's perceived texture. It measures the spread of the spectrum around the spectral centroid, indicating the range of significant frequencies that contribute to the sound.

Pitch and Intonation Patterns: Pitch refers to the perceived frequency of a sound, determining the musical note or tone that it corresponds to. Intonation pattern describes the variation of pitch over time within spoken language, contributing to the expression of emotions, questioning, or emphasis in speech. Each data point produces a matrix $P \in \mathbb{R}^{n\times m}$ of pitch tracks (where n represents the number of windows into which the audio signal is divided, and m represents the number of pitch measurements taken within each window) and a matrix $M \in \mathbb{R}^{n\times m}$ of magnitude windows (where n again represents the number of windows, and m represents the number of magnitude measurements within each window). From these matrices, we produce a vector of size $v \in \mathbb{R}^n$ where each element is the pitch with the most magnitude for each window (windows that only have null pitches are dropped). From v, we compute, the

mean, median, standard deviation, $25th$ percentile, $75th$ percentile, maximum value, and minimum value, which is then added to our attribute matrix.

Formant Frequencies: A formant is a characteristic component of the quality of a speech sound, particularly a vowel. When we speak, our vocal tract acts as a resonator, emphasizing specific frequencies and shaping the sounds we produce. Formants play a crucial role in characterizing vowels and certain consonants, greatly influencing speech recognition and synthesis. We will focus on three types of formants, which will be extracted using the *parselmouth* library. *F1* which is determined primarily by the height of the tongue body, *F2* influenced by the frontness or backness of the tongue body, *F3* which is associated with lip rounding. For each data point, we extract *F1*, *F2*, *F3* values for a set time point. Given that the lengths of our data points aren't the same, we compute the habitual mean, median, standard deviation, $25th$ percentile, $75th$ percentile, maximum value, and minimum value, for each type of formant. We also extract the difference between the means of *F1*, *F2*, *F3*, yielding two additional features. These values are then added to the attributes matrix.

Root Mean Square (RMS): Root Mean Square (RMS) represents the square root of the average power of a signal, providing a measure of its amplitude. It's commonly used to quantify the volume level or energy of an audio signal over time. We compute the RMS energy for each frame, yielding a vector $v \in \mathbb{R}^n$, where n is the number of windows. From v, we compute the habitual central tendency metrics (as cited above), interquartile range, and the energy variability (sum of absolute differences)

Zero Crossing Rate (ZCR): Zero crossing rate (ZCR) is the rate at which an audio signal changes from positive to negative or back, effectively measuring the number of times the amplitude crosses the zero point. This feature is useful for analyzing the noisiness of a signal and distinguishing between voiced and unvoiced speech. We retrieve a vector $v \in \mathbb{R}^n$ where n is the number of windows, and where each element is the fraction of zero crossings in a given window.

Harmonics to Noise Ratio (HNR): Harmonics to noise quantifies the amount of harmonic sound (periodic vibrations) relative to the amount of noise (non-periodic components) in a voice signal. A higher HNR generally indicates a clearer and more harmonic sound, which is often associated with a healthier and more typical voice production. In our case, we compute the HNR mean (represented in decibels) from all the audio clips, using the *parselmouth*. This value is then added to the attribute matrix.

The result of the feature preprocessing is an attributes matrix, $A \in \mathbb{R}^{34045 \times 161}$, consisting of features that are correctly formatted for use in machine learning models.

4 Methodology

4.1 Why supervised Learning ?

Supervised learning algorithms (SVC and Random Forest Classifiers) are used in this project because unlike unsupervised learning algorithms, they learn the specific characteristics of each language's phonetic, intonational, and rhythmic properties, leading to highly accurate classification once the model is well-tuned. Unsupervised methods like clustering algorithms (DBscan, K-means or hierarchical clustering) identify groups based on feature similarity but do not inherently know what these groups represent (which language each group corresponds to) which can lead to an ambiguity in cluster interpretation.

Unsupervised algorithms suffer more from the curse of dimensionality than supervised models. In fact, SVCs are more effective even in high dimensions due to the fact that they primarily care about the points closest to the decision boundary (support vectors) rather than the dimensionality of the space. Furthermore, the maximization of the distance between the support vectors and the boundary ensures that the model focuses on the most informative features for classification. This property allows SVM to be less prone to overfitting, a common issue in high-dimensional spaces. Due to the kernel trick, SVMs can also handle non linear decisions which allows them to operate in higher dimensions without directly computing the coordinates in that space unlike DBscan or k-means for example (more description about SVC on the section **4.3**).

Additionally, Random Forest Classifiers are based on a set of decision trees which are robust against overfitting because each tree in the forest is bult from a random sample of features, reducing the variance of the set without substantially increasing the biais. Furthermore, the classification for Random Forests is more efficient in hight dimensionality compared to unsupervised algorithms because it randomly selects subsets of features at each split, which means it can manage high-dimensional data by focusing on the most informative features for making splits.

4.2 Dimensionality Reduction

Before applying PCA (Principal Component Analysis) or MDS (Multi-Dimensional Scaling), we wanted to see the relationship between features i.e. how strongly they are correlated. This correlation study would guide us in the dimensionality reduction algorithm we choose to apply to our data. If some features are strongly correlated, PCA would be pertinent to be applied, as some features are linearly related. Even though our correlation study (see the correlation matrix in the appendix) gave us strong evidence of non-linear relationship between our features (meaning that MDS might be a better suit for our dataset), we opted to reduce dimensionality using PCA due to insufficient computing and memory (see 5.1 Rejected method).

Principal Component Analysis

The PCA method searches for an orthogonal projection of the original data from a space of dimension D into a subspace of lower dimension $k < D$, that could stand as an approximation of the original dataset. We consider the optimal projection as the one minimizing the mean squared error over the data,

$$\arg\min_{P:\mathbb{R}^{d\times d}} \left\{\mathbb{E}_x\left[||\vec{x}_i - P\vec{x}_i||^2\right]\right\}$$
$$s.t.\ P = P^2 = P^T; rank(P) = k$$

The error term can be seen as the residual of the projection, so it minimizes $||P^{\perp}\vec{x}_i||^2$. This projection matrix can be defined from an orthonormal basis, as $P = UU^T$, whose components maximise their product with the covariance matrix of our training set.

$$\arg\max_{U:\mathbb{R}^{k\times d}} \left\{\sum_{j=1}^{k} \vec{u}_j^T \mathbb{C}ov_x(\vec{x_i})\vec{u}_j\right\} s.t.\ U^TU = I$$

As we found during class, this basis is made up of eigenvectors of the covariance matrix, $\Sigma\vec{u}_j = \lambda_j \cdot \vec{u}_j$. The k-dimensional projection is built from the k eigenvectors (normalized) with largest eigenvalues.

For the purpose of dimensionality reduction, we could just perform an orthogonal projection as optimized above. But the PCA process will also transform the coordinate basis from our chosen feature space into the eigenspace of $\Sigma = U\Lambda U^T$, via:

$$\vec{z} = U^TP\vec{x} = U^TUU^T\vec{x} = U^T\vec{x}$$

The vectors are now embedded in a lower dimensional space where the covariance matrix is diagonal, cutting off the original (eigen)dimensions responsible for the least variance. We can show that the vector norm is preserved by the change of basis, same as the simply projected vectors, $||U^T\vec{x}||^2 = ||P\vec{x}||^2$.

The method described above, maximising variance in the data, is biased towards features that are numerically bigger, with results that could vary with the choice of units of measurement (e.g. meters vs kilometers) that should not be significant. To avoid this problem, we standardize the data. This means we center the data to $\vec{0}$ mean, then scale each feature independently so that their variances σ_k^2 are all equal to 1. The net effect is that PCA uses the correlation matrix of our original dataset, not the covariance matrix.

$$\hat{x}_k = \frac{x_k - \mu_k}{\sigma_j},\ \hat{\Sigma}_{i,j} = \frac{\Sigma_{i,j}}{\sigma_i \cdot \sigma_j} = \rho_{i,j} \in [-1, 1]$$

While the elements of the covariance matrix in feature space are all standardized to 1, and the total variance is just the number of dimensions, the diagonalization computed by PCA will tease out a new basis where variances can differ, so asking for the k largest directions still makes sense.

Algorithm 2 Dimensionality Reduction with PCA

Initialize dataset $X \in \mathbb{R}^{N\times D}$
Compute all feature means $\vec{\mu}$ and variances $\vec{\sigma}^2$
Center and scale the data:
for each vector feature $x_{i,t}$ in X **do**
    $z_{i,t} \leftarrow \frac{1}{\sigma_t}(x_{i,t} - \mu_t)$
end for
Compute the covariance matrix:
$\Sigma = Z^TZ/N \in \mathbb{R}^{D\times D}$:
Take the singular value decomposition:
$svd(\Sigma) = U \cdot \Lambda \cdot U^T$
Keep the largest components of the basis:
$\hat{u}_{1\ldots k} = \arg^k \max\left\{\vec{u}_t^T\Lambda\vec{u}_t\right\} \in \mathbb{R}^{k\times D}$
Transform data vectors into $\hat{U}$ coordinates:
for each data vector $\vec{z}_i$ in Z **do**
    $\vec{y}_i \leftarrow \hat{U}^T\vec{z}_i \in \mathbb{R}^k$
end for
return the new dataset $Y \in \mathbb{R}^{N\times k}$

After applying PCA on our dataset, with a target of maintaining 95% of the variance, we keep 90 features from the 161 initial features. This means that the first 90 principal components of the correlation matrix $\hat{\Sigma} \in \mathbb{R}^{161\times 161}$, capture 95% of the variance in the (standardized) training set.

4.3 Support Vector Classifier

For this project, we use the library *SVC* from *sklearn* for multi-class classification. It trains an SVM one-vs-one classifier for each pair of classes (meaning that

each classifier is responsible for deciding between two classes), and then builds from them a one-vs-rest classifier for each class (meaning that a separate classifier is trained for each class to distinguish instances of that class from all other classes combined). The single class prediction is obtained from the most confident prediction of the one-vs-rest classifiers.

This library use a non-linear SVM with a soft margin. Non-linear SVM with a soft margin is particularly useful when dealing with data that cannot be linearly separated in the original feature space (which is the case in our situation). The inclusion of slack variables, ξ_i, allows some data points to be on the incorrect side of the margin, providing a balance between the complexity of the model and its error minimization.

Mathematical Description

For a dataset $X \in \mathbb{R}^{n \times p}$ and a vector $y \in \{-1, 1\}^n$, the objective is to find w such that the values returned by the functions $f(X) = sign(w^T \phi(x) + b)$ are correct for most samples.
SVC resolve the following primal problem : $\min_{w,b,\boldsymbol{\xi}} \left(\frac{1}{2} w^T w + \beta \sum_{i=1}^{n} \xi_i \right)$ subject to:
$y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i$, $\xi_i \geq 0, \quad \forall i$. By using Lagrange multipliers, we obtains the dual :
$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - \mathbf{1}^T \alpha$ subject to:
$y^T \alpha = 0$, $0 \leq \alpha_i \leq \beta, i = 1, ..., n$ with Q an n by n positive semidefinite matrix, $Q_{ij} \equiv y_i y_j K(x_i, x_j)$, where $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is the kernel.

After the optimisation is done, the output for the decision function for X is : $\sum_{i \in SV} y_i \alpha_i K(x_i, x) + b$ where SV, represent the support vectors (the points that lies within the margin).

Hyperparameters

When tuning our model, we considered three hyperparameters to tune, the regularization parameter for the soft margin β, different types of kernels, and their scale coefficient. As mentionned previously, slack variables allow some points to be on the wrong side of the margin. We can intuitively see that if there is no constraint on the slack variables, the model will be too lenient, meaning that it will accept too many incorrect classifications. The β regularization parameter will modulate the slackness of the margin. The higher the value of β, the slacker, and the more lenient, is the margin.
For the kernels (γ parameter that is included in the kernel definitions), we test a polynomial kernel, RBF kernel, and a linear kernel. The linear kernel, defined as $K(x, y) = x^T y$ where x and y are feature vectors, assumes that the boundaries between classes is a linear function in the $\mathbb{R}^n$ space. Knowing that our data is most likely non-linear given our correlation study, we didn't have much hope that this kernel would yield good results, but we still added it to our kernel set to compare to the other intuitively more promising kernels. The polynomial kernel defined as, $K(x, y) = (\gamma x^T y + r)^d$, where γ is a scale factor that controls the influence of higher-order terms in the polynomial, r is a constant, and d is the degree of the polynomial. This kernel represents the similarity between vectors in a feature space over polynomials of the original data. The RBF kernel, defined as $K(x, y) = exp(\gamma \|x - y\|^2)$, with γ defines how much influence a single training example has on the fitting of the model. The larger γ is, the closer data points must be to affect the model significantly, and thus, preventing from overfitting. The RBF kernel maps samples into a higher dimensional space using the square of the Euclidean distance between them, and thus it is very effective in cases where the decision boundary is highly irregular and cannot be approximated by a simple hyperplane.
Having briefly defined the γ parameter, let's delve into more detail into how it affects the polynomial and RBF kernels respectively. The γ parameter plays a critical role in defining the behavior of both the polynomial and RBF kernels, each in a distinct manner, tailored to their specific mathematical formulations. For the polynomial kernel, γ serves as a scaling factor for the dot product $x^T y$ in the formula $(K(x, y) = (\gamma x^T y + r)^d)$. By adjusting γ, we control the influence of higher-order terms in the polynomial. Increasing γ emphasizes the higher degrees of the polynomial, thereby capturing more complex patterns in the data, but also increasing the risk of overfitting, especially in noisy datasets or those with outlier points. Conversely, a lower γ reduces the complexity of the model, making it behave more linearly, which could be beneficial for datasets where the underlying patterns are less complex.

In the case of the RBF kernel, γ impacts the width of the Gaussian function used to measure the similarity between points: $(K(x, y) = \exp(-\gamma \|x - y\|^2))$. A high γ value results in a narrow peak, meaning that only data points close to each other are considered similar. This sensitivity allows the model to fit closely to the training data, which can capture intricate data structures but may lead to overfitting if the data has noise. A lower γ produces a wider peak, giving the model a more general perspective of the

data, enhancing its ability to generalize but potentially underfitting complex datasets.

Here is a pseudo code for SVC:

Algorithm 3 SVC with Non-linear SVM (Soft Margin) with Kernel

Input: Training data X, labels y where $y \in \{1, 2, \ldots, K\}$, regularization parameter β
Output: Support vectors X_{SV}, coefficients α, bias b, multi-class classifiers C

procedure SVMTrain(X, y, β)
    Choose a kernel function K (e.g., RBF)
    Formulate the dual optimization problem
    Use quadratic programming to solve for α
    Compute b using support vectors (where $0 < \alpha_i < \beta$)
    Train multi-class classifiers using one-vs-one strategy
    Train multi-class classifiers using one-vs-rest strategy
    return X_{SV}, α, b
end procedure

function OneVsOneClassify(X, C)
    for each pair (i, j), where $i, j \in \{1, 2, \ldots, K\}, i \neq j$ **do**
        Compute predictions using C_{ij}
    Aggregate predictions across all classifiers
    return Most frequent class
end function

function OneVsRestClassify(X, C)
    for each class $i \in \{1, 2, \ldots, K\}$ **do**
        Compute predictions using C_i
    Choose the class with the highest confidence from the classifiers
    return Chosen class
end function

function SVMPredict(X, X_{SV}, α, b, K)
    Compute: $f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x, x_i) + b$
    return sign($f(x)$)
end function

4.4 Random Forest

For this project, we use the library *RandomForestClassifier* (RFC) from *sklearn*. Random Forest is an ensemble learning technique that builds multiple decision trees and merges them together to get a more accurate and stable prediction. The algorithm combines the output of multiple (often hundreds) randomly created decision trees, a method known as *bootstrap aggregating*, or *bagging*. Each tree in the forest is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

Algorithm 4 Random Forest Algorithm

Input: Data set D, Number of trees N
Output: A Random Forest F
procedure RandomForest(D, N)
    Initialize forest F as an empty list
    for $i = 1$ to N **do**
        $D_i \leftarrow$ BootstrapSample(D) (sample data)
        $Tree \leftarrow$ BuildTree(D_i)
        Add $Tree$ to F
    end for
    return F
end procedure
function BuildTree(D)
    if Base case condition **then**
        return leaf node
    end if
    $Feature, Threshold \leftarrow$ BestSplit(D)
    Split D into D_{left} and D_{right} based
    on $features$ and $threshold$
    $Node.left \leftarrow$ BuildTree(D_{left})
    $Node.right \leftarrow$ BuildTree(D_{right})
    return $Node$
end function
function BestSplit(D)
    Choose a feature and threshold
    that best splits the data D
    return $Feature, Threshold$
end function

Mathematical Description

Let $\mathcal{D} = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)\}$ be a dataset containing n samples, where each sample consists of a feature vector X_i and a corresponding label Y_i. Random Forest creates a multitude of decision trees using the following steps:

Mathematically, the prediction of a Random Forest classifier can be expressed as:

$$\hat{Y}^p = \frac{1}{N_T} \sum_{b=1}^{N_T} T_b(X) \tag{1}$$

where N_T is the number of trees in the forest and T_b represents the prediction of the b-th tree.

Hyperparameters

The performance of a Random Forest model is highly dependent on the settings of various hyperparameters. Here, we discuss some of the most crucial ones:

- `n_estimators`: It specifies the number of trees in the forest. More trees increase the model's accuracy and stability by reducing the variance in predictions, but they also increase computational demands. Generally, there are diminishing returns in performance improvement beyond a certain number of trees.

- `criterion`: The impurity measure used when splitting tree nodes. Each decision split chooses the predicate condition that minimises the expectation of this measure on the sub-populations divided among sub-trees.
  First, the Gini score $G(p_1 \dots p_k) = 1 - \mathbb{E}_k[p_k]$. In words, that is the average probability of not finding each class.
  Second, the entropy $H(p_1 \dots p_k) = \mathbb{E}_k[-\log_2(p_k)]$ measures the uncertainty in bits These impurity measures are minimised when there is only one class found in a subtree's population. They are maximised when all classes are found in equal amounts.

- `max_depth`: The maximum depth allowed for each tree. A deeper tree can model more complex relationships by creating more specific rules. However, it also makes the model more likely to overfit to the noise in the training data.

- `min_samples_split`: The minimum number of samples required to split an internal node. Higher values prevent the model from learning overly specific patterns, thus lowering the risk of overfitting. Smaller values allow the trees to make more complex decisions, increasing the risk of overfitting.

- `min_samples_leaf`: The minimum number of samples required to be at a leaf node. Setting this parameter can ensure that leaf nodes contain more than one sample, which helps in smoothing the model's predictions and preventing overfitting.

- `max_features`: The number of features to consider when looking for the best split. Increasing 'max_features' generally improves the performance of the model as each node now has a higher number of options to consider. However, this can also increase the correlation between the trees, reducing model variance but potentially increasing bias.

- `bootstrap`: Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree. Using the default bootstrapping adds randomness to the model, by training each tree on a random subset of the data, which helps in reducing model variance and avoiding overfitting.

4.5 Hyperparameter Tuning

Algorithm 5 Random Search with Cross-Validation

```
Initialize dataset D
Define set of hyperparameters H
Initialize best performance as −∞
Initialize best hyperparameters as ∅
for each set of hyperparameters h ∈ H do
    Split D into 5 equal folds D_1, D_2, D_3, D_4, D_5
    Initialize performance list P
    for i ← 1 to 5 do
        Set D_train ← D \ D_i (all folds except the i-th fold)
        Set D_test ← D_i
        Train a RFC model on D_train with hyperparameters h
        Evaluate the model on D_test and store the performance in P
    end for
    Compute mean performance m ← mean(P)
    if m > best performance then
        Update best performance ← m
        Update best hyperparameters ← h
    end if
end for
Output the best hyperparameters
```

Hyperparamter tuning is a crucial step to finding the best model. As data our data is complex and highly dimensional, we will need to apply a hyperparameter tuning to find the best parameters of our models (SVC and RFC). We will use a Random Search algorithm to find the best hyperparameters (see Algorithm 2). A Grid Search would've been another alternative, which would exhaustively search for the best model in a given hyperparameter space.

5 Results

For the sake of consistency and reproducibility, we fixed the "random_state" parameter of all training components:
- The data splitter deciding which vectors are part of the overall training vs test set.
- The dimension reduction operators PCA and MDS.
- The classifier models SVC and RandomForestClassifier.
- The hyper-parameter optimizer RandomizedSearchCV.

5.1 Rejected methods

In this section, we talk about the methods that we tried but didn't work.

First, the *Common Language* dataset we are using came pre-separated into a 'train', 'test' and 'validation' set. Initial training runs with this split gave very poor results ($\approx$ 2x worse), compared to random selections from the whole dataset. So, we decided to use *sklearn*'s stratified splitter on the entire dataset, ignoring the pre-defined split. Stratification ensures that the chosen subsets contain the same proportions of each class label.

Secondly, based on the correlation matrix in the appendix, we can see that our data are not well correlated. So, it is more relevant to use MDS instead of PCA but MDS demand more computational resources. We tried to reduce the dimensionality with MDS, both with euclidian or malahanobis distance, but our Python environments crashed due to insufficient memory. Maybe the number of data points was too high at $\sim$ 34k. This happened also on the cloud Colab environment. In fact, MDS involves computing the distances between all pairs of points in the dataset and then optimizing the configuration of points in the new space to reflect these distances as closely as possible.

There is also the possibility of "whitening" or "sphering" the embedded data from PCA, that is to standardize the output variances, before the classifier training step. In the end, we found it did not improve classification performance.

Models	Accuracy	f1-score	Precision	Recall
SVC	13.61%	12.64%	12.91%	13.17%
RFC	10.99%	8.75%	9.81%	10.22%

Table 1: With the *Hugging Face* data split

Finaly, in the initial datasets, we have 45 labels. Some languages in the labels are very similar so we reduced the number of labels by grouping them based on the region in which they are spoken. We used GPT-4 to find new labels. Here is the promp we used : `We have a set of labels for a classification task based on speech and want to relable them based on similarity and geographic location. Output 10 new labels.`

Here is the comparaison of the results after running the models (with the best hyperparameters) with 45 labels vs 10 labels:

Models	Accuracy	f1-score	Precision	Recall
SVC	24.60%	23.35%	24.08%	23.61%
RFC	17.91%	16.20%	17.12%	17.06%

Table 2: With 45 labels

Models	Accuracy	f1-score	Precision	Recall
SVC	29.82%	29.66%	29.72%	29.66%
RFC	26.48%	25.36%	28.51%	25.50%

Table 3: With 10 labels

Given that the data has less labels (is less complex), there is less room for misclassification for the models which explain why the metrics are higher. We decided to keep the inital labels do to the fact that a model trained with the initial labels might be better for classifying languages than a model trained based on the region in which the languages are spoken because the first model will be trained with more precise labels.

5.2 Metrics

In order to compare, and determine the best model, we need to set metrics which will guide us to our decision. We will use four metrics, accuracy, F1 scores, precision and recall. These metrics are defined in terms of the number of sample and the four categories. True/False means that the classifier is right or wrong and Positive/Negative means that the classifier predicts that the sample is in the class +/- resp. So a *True Positive* represent the fact that the classifier predicts that a sample is in the + class and he is right. A *False Positive* represent the fact that the classifier predicts that a sample is in the + class but he is wrong. And the same logic goes for *True Negative* and *False Negative*.

Accuracy: The accuracy of a classifier is the ratio between the number of True positives and the total number of predictions ($P(Class == Prediction)$):

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{Total number of predictions}}$$

Precision score: The precision of a classifier represent the probability that the sample is really in the class + knowing that the classifier prediction is + ($P(Class = +|Prediction = +)$). The is the probability that the prediction of the classifier is right:

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

Recall score: The recall is the probablity that the prediction of the classifier is + knowing that the sample is really in the class + ($P(Prediction = +|Class = +)$):

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

F1 Score: This metric is (almost) the Jaccard similarity between the sets of positive classification and positive prediction. That would be the ratio of their intersection over their union.

$$\frac{P(Prediction = + \wedge Class = +)}{P(Prediction = + \vee Class = +)}$$

But instead, F1 score is twice the intersection, divided by the union plus the intersection again.

$$\text{F1} = \frac{2 \times P(Predict = + \wedge Class = +)}{P(Class = +) + P(Predict = +)}$$

It can be calculated as the harmonic mean of Precision and Recall:

$$\text{F1} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

5.3 Support Vector Classification

After cleaning our data and training the models with no hyperparameters tuning, we get those results:

Kernel	Accuracy	f1-score	Precision	Recall
Linear	19.75%	18.42%	18.49%	19.01%
Poly	19.02%	17.91%	26.08%	17.74%
RBF	24.54%	16.20%	23.02%	22.54%

Table 4: Results for SVC before grid search

Then, applying random search (Algorithm 2) gives us those parameters : `C=10, gamma=auto, kernel=RBF` and we get these results after training a SVM with those hyperparameters:

Kernel	Accuracy	f1-score	Precision	Recall
RBF	24.60%	23.35%	24.08%	23.61%

Table 5: Results SVC after grid search

5.4 Random Forest Classification

As an exploratory first step, we fitted our data to an Random Forest Classifier, with sklearn's default hyperparameters. Here are the results:

Accuracy	f1-score	Precision	Recall
17.91%	16.20%	17.12%	17.06%

Table 6: Results RFC before grid search

We applied the Random Search algorithm (Algorithm 5) to find the best hyperparamters in a given hyperparameter space. The output of this search gave us this set of hyperparameters `n_estimators = 200, min_samples_split = 5, min_samples_leaf = 4, max_features = sqrt, max_depth = None, criterion = gini` which yield the best. Results obtained after running the Random Search :

Accuracy	F1-score	Precision	Recall
20.57%	17.90%	20.96%	19.51%

Table 7: Results RFC after grid search

In both SVC and RFC, we see an improvement in all of our metrics after running the Random Search. This suggests that by exhaustively exploring the hyperparamter space, we could potentially find an even better model (even though the performance gain is marginal).

6 Conclusion

This project embarked on the challenge of comparing the efficacy of two widely used machine learning algorithms, Support Vector Machines (SVC) and Random Forests (RFC), in the domain of language classification based on speech. Through extensive data preprocessing, model training, and hyperparameter optimization, several key insights were unearthed.

Firstly, the reduction of labels from 45 to 10, intended to simplify the classification problem, yielded expected results. Both SVC and RFC metrics increased do to the reduce of the complexity of the label set. So, both algorithm demonstrated robustness by improving accuracy when the problem space was

reduced, underscoring its adaptability to variations in data complexity.

Moreover, the process of hyperparameter tuning was pivotal, showcasing significant improvements in model performance. It emphasized the importance of algorithm-specific parameter settings, which tailored each model more closely to the intricacies of the task.

We can also see that SVC f1-score is 30% higher than RFC. So globaly, based on the results obtained in this project, we can conclude that SVC is a better classifier.

In conclusion, this project not only reinforced the necessity of careful data preprocessing and the strategic choice of model parameters but also illuminated how different algorithms can uniquely respond to changes in data structure. For future work, exploring additional ensemble techniques, and possibly integrating more advanced neural network architectures, could provide further enhancements in the field of language classification. The insights gained here lay a foundation for deeper exploration into automated language recognition, promising to elevate the tools available for global communication and information exchange.

7 Team Contributions

Samir: During the project, i wrote the code for the Fast Fourier tranform for audio visualisation that helped us see the form (wave form) of the audios. I made the exploration of the data to see the global shape and the statistics of the data (mean, sum and count). I wrote the code for the preprocessing of our data to get all the features we needed. I also wrote the code to reduce the demensionality (PCA), train the models and get the result metrics for each trained model.

For the report, i wrote the introduction (*section 1*), objectives of the project (*section 2*). I also described where we found our data (*section 3.1*). I helped Kamen on the explanation of each preprocessing features we used (*section 3.4*). I wrote the *section 4.1* about why we used supervised learning instead of unsupervised learning. I described the maths behind SVC and i wrote the *algorithm 3* with help of Kamen (*section 4.3*). I also wrote the algo *algorithm 4* and contributed on the writing of the *section 4.4*. I explained why we kept the initial labels in the dataset in the *section 5.1*, described the metrics (*section 5.2*), explained the results for the SVC (*section 5.3*) and wrote the conclusion (*section 6*).

Kamen: I developed the initial scripts for retrieving data from Hugging Face and prepared it for data analysis (*algorithm 1*). I recommended relevant metrics to explore and analyzed the data alongside Samir. Additionally, I developed the script to generate the attribute matrix from the raw feature extraction, enabling its use in PCA and model training. I also contributed to creating the initial train/test splits and tested the SVC and Random Forest models with our data. Furthermore, I proposed and implemented Random Search as a method for fine-tuning the models' hyperparameters.

For the report, I wrote the *section 3.2* and the *section 3.3*. I contributed to writing the *section 3.4* with Samir. I described the SVC hyperparamters in *section 4.3* and explained the Random Forest Classifiers' results in *section 5.4*. I also wrote *section 4.5* on hyperparameter tuning.

Simon: I proposed that we work on a language classification task from voice data. I found the *Common Language* dataset from *Hugging Face*. During initial research, I also found tutorials for data analysis specifically on audio and voice data, with explanations of relevant features to extract.

During the project, I kept the code base organized and helped my teammates with Git difficulties. Most of the code was initially written in an exploratory way in Jupiter notebooks, both locally and in Google Colab, and I compiled the relevant sections into libraries and batch Python scripts that could give us reproducible results. That also means reviewing and fixing errors that we would just skip over during exploration. I would often help my teammates to debug code, or clarify difficult concepts and formulas. There is no part of the code base that I started or completed myself.

For example, I debugged the cleaning and feature extraction stages as some of our cleaned voice clips ended up too short and caused exceptions from the library. I also solved the issue where random search would only find NaN scores. I fixed our train/test splitting logic, and a leak of test data into the PCA step. I ran many of our final training runs and provided the optimal parameters.

For the report, I wrote the *section 4.2* on Dimensional Reduction (PCA/MDS). I wrote descriptions for some of the Random Forest hyperparameters, and for result evaluation metrics. I wrote on our difficulties with MDS and the train/test splitting. I coded and printed the feature correlation matrix, and the classification confusion matrices.

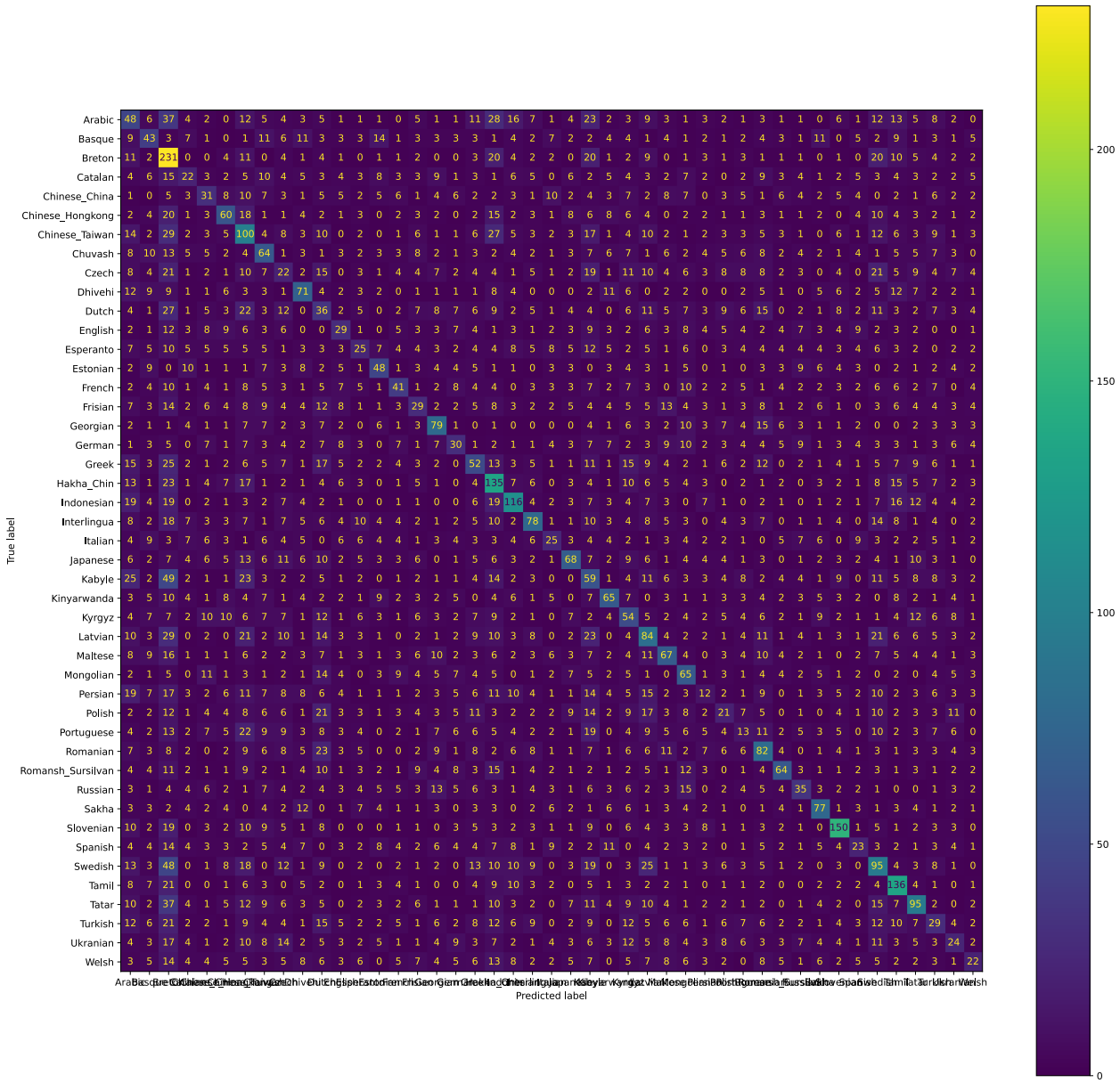
8 Appendix

4.2 Dimensionality Reduction

Correlation Matrix of the original data features:

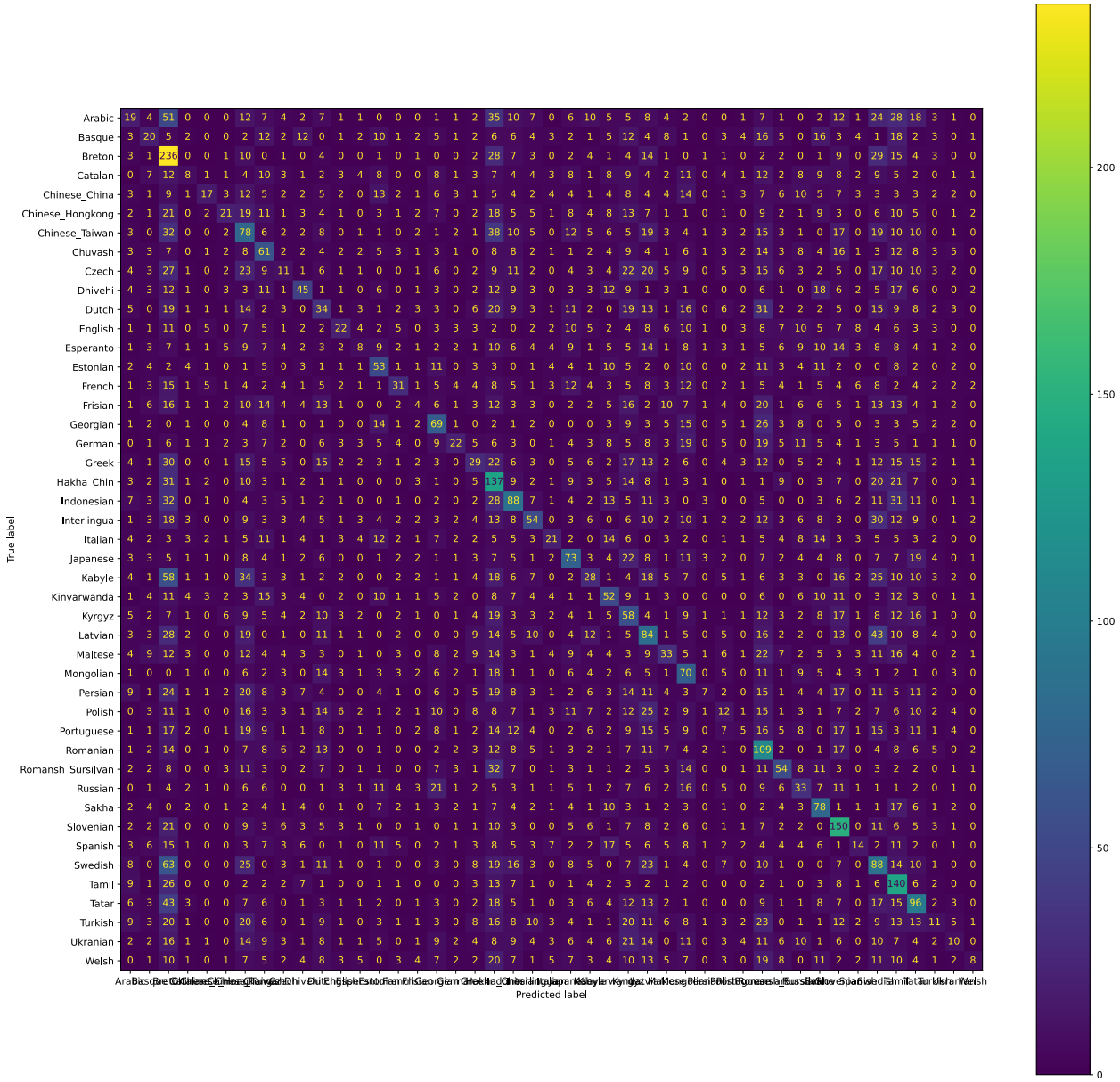
5.3 Support Vector Classification

Confusion Matrix for Support Vector Classification results:



5.4 Random Forest Classification

Confusion Matrix for Random Forest Classification results:



9 References

- "Common Language" Data Source
- Audio Signal Processing for Machine Learning
- SINDy
- Python Package for Music and Audio Analysis
- Number of MFCCs used in the code
- Formants
- Principal Component Analysis
- Randomized Hyperparameter Search
- Support Vector Classifier
- Multi-Dimensional Scaling
- Random Forest Classifier
- K-Fold Cross Validation
- Randomized Hyperparameter Search
- Model Evaluation Metrics[1], Another[2]
- Multiclass Receiver Operating Characteristic (ROC-AUC)
- **The most important algorithm of all time**[1], **Another**[2]