Identification of Instruments Through Sound Characteristics

Avery Cameron: 200254563

Raymond Knorr: 200302685

Mason Lane: 200376573

Kegan Lavoy: 200378170

Table of Contents

List of Figures:

List of Tables:

Project Summary:

Using the NSynth dataset, a large, high-quality dataset of annotated musical notes, our aim is to train a model to classify a note played on a violin against a note played on a piano. We extract audio features from WAV files using the LibROSA python package, and use these features to train supervised learning models. We also recorded and labelled a smaller set of our own WAV files for violin and piano to compare against the model produced from the NSynth dataset. Algorithms being used to train models include Binary Decision Trees, AdaBoost, K-NN, and Support Vector Machines.

Description of Problem:

The classification of musical sounds has widespread potential. The ability for a computer to accurately distinguish different instruments is the underpinning of any automated transcription software. This would involve computers processing a full song and writing out all of the various instrument parts, and would fundamentally change the way that musical property is managed, among other things. Our project does not aim to tackle such an overarching issue as full-scale musical transcription. However, developing individual classifiers to distinguish notes played on different instruments is vital for any larger applications.

The NSynth dataset is made up of sounds from various sampling libraries, and the sounds are free from additional noise. We are interested to see if models trained on pristine audio samples will still perform well on our own gathered samples. Our project extends the training using the NSynth dataset to classify our violin and piano dataset named the RACK dataset. The RACK dataset is discussed more in depth in the Dataset section.

Project Objectives:

The objective of our project is to accurately predict the instrument being played based off of an input audio file. Various classifiers implemented in Scikit-learn such as AdaBoost, Random Forests and SVM will be used (Pedregosa et al. 2011). The extracted features will be used to train the classifiers and predict on new test sets. The metrics of success for our project are an accuracy of 97.5%, precision of 95% and a recall of 95%. These metrics are based on academic papers of Seipel(2018) and Xu et al. (2004). Precision is a good indicator when false positives have a high cost and recall is a good indicator of false negatives (Shung, 2018). We focused on accuracy which is sensitive to correct classifications as false classifications do not have a high cost and we already balanced the datasets (Huilgol, 2019). These metrics will be gathered for each implementation and compared against each other to determine the most successful result from the project. These results from the classifiers are entered into a confusion matrix which

is of the form: [[TP, FP],[FN, TN]]. The confusion matrices for train, validation and test are used to calculate the accuracy, prediction and recall of the classifier.

This project also seeks to identify audio features that are good indicators of an instrument, specifically piano and violin. These features will be positive indicators of which instrument the audio file is from and can be investigated as a focal point of multi class problems.

Dataset:

We are using the NSynth dataset to train our models. The NSynth dataset consists of 305,979 musical notes, each with a unique pitch, timbre, and envelope (Jesse et al., 2017). The full NSynth dataset consists of individual train, validation, and test datasets. We used all three of these datasets in the development of our models. Each note is a four second long WAV file taken at a sampling rate of 16kHz. All of the samples were synthetically generated using commercial sampling libraries.

The RACK dataset, named after the initials of Avery Cameron and Raymond Knorr, contains piano and violin notes ranging from A#3 to G5 in pitch. We tried to make the audio as similar as possible to the NSynth dataset. The audio clips are approximately four seconds long, three seconds of the instrument playing and one second for the instrument to die off. The sampling rate of these files is 16KHz and is a mono output. These audio files are accompanied with a JSON file for annotations. However, the RACK JSON files do not contain the annotated qualities that the NSynth dataset provides.

We are also using the LibROSA python library to extract several other audio characteristics from the WAV files. These characteristics are: *harmonic*, *percussive*, *Chroma Energy Normalized*, *Mel Frequency Cepstrum Coefficients*, *Mel Spectrum* and *Spectral Contrast*. The *harmonic* and *percussive* components of the sound are extracted and saved separately. *Chroma Energy Normalized* is the sound separated by pitch and height (chromatic quality and the octave of the pitch) and is the short time energy spread over the twelve chroma bands (Müller, Ewert, 2011). The *Mel-Frequency Cepstrum Coefficients* (MFCC) is the fast fourier transform (FFT) of the power spectrum, used in audio recognition and modeling (Xu, et al., 2004)(Xu et al., 2003). In addition to MFCC, the *Mel Spectrum* is an intensity plot of a short-time fourier transform (Smith, 2007). *Spectral Contrast* measures the difference between the valleys and peaks of the spectrum in the sub-bands (Jiang, Lu, Zhang, Tao, & Cai, 2002).

Methodologies & Implementation:

The state-of-the-art approaches used are based partially on research by Patil & Sanjekar (2017) who suggest the use of SVM, and AdaBoost. Additionally, research by Nagawade and Ratnaparkhe (2017) used *MFCC* to accurately represent unique qualities and features of the sound. Random forests were also implemented as a successful solution for instrument classification by Owers (2016).

The audio features mentioned in the description of the dataset were determined by Avery through academic review from the works of Seipel (2018), Xu et al. (2004) and Kawwa (2019). The feature extraction demonstrated in the progress report was modified to extract more information from the features chosen. Avery changed the return values from a single mean of all values to an array of separate means. This change provides a mean over the sections of time captures in the audio clip. The mean of coefficients was removed to preserve separate coefficients for each feature. As an example, the feature MFCC now returns an array of thirteen means of the samples over time determined by the thirteen coefficients chosen for extraction instead of a single value. Constant Q Chroma was analyzed but did not add meaningfully to results and had no apparent impact on accuracy. The qualities within the NSynth JSON file - included in the NSynth download - were also added to the feature extraction implementation, Which allowed the dataset to be split/organized by it's target, the instrument. The extracted features were analyzed using XGBoost which identified the *sixth coefficient of spectral contrast*, and *MFCC coefficient zero* as key features. After determining the best method of extracting features, the features were saved to CSV files, preserving the built dataset while avoiding unnecessary and lengthy WAV read times.

Random Forests:

The first algorithm used was Random Forests implemented by Raymond Knorr using Scikit-learn. Scikit-learn defines a random forest to be "a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting" (Random Forest Classifier).

The Scikit-learn RandomForestClassifier function trains a random forest on a dataset and allows for the manipulation of several hyperparameters. The hyperparameters varied during our testing are as follows: *criterion, max_depth, max_features, min_samples_leaf, min_samples_split,* and *n_estimators*.

All of these hyperparameters were optimized using a Scikit-learn RandomizedSearchCV. This function performs a randomized search on hyperparameters by using random values for each within a given range (RandomizedSearchCV). The model is

trained with every random subset of hyperparameters and the best performing model is chosen. Unlike a grid search, only a fraction of the possible combinations from the value ranges are used. This is a faster, less computationally expensive way of optimizing hyperparameters. Due to the implementation of RandomizedSearchCV, the model is fit to the training dataset, and the hyperparameters are also fit to the training dataset, therefore the NSynth validation dataset functioned as another test dataset for this algorithm.

After tuning hyperparameters and testing on the validation and test sets, the random forest model performed very well and surpassed our success criteria of 97.5% accuracy, 95% precision and 95% recall. The exact values of our final random forest model are in the table below:

Table 1: Random Forest Final Results

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Train	100	100	100	100
Validation	99.51	100	99.03	99.51
Test	99.35	100	98.71	99.35

AdaBoost:

The second algorithm used was AdaBoost which was implemented with Scikit-learn by Avery Cameron. Scikit-learn allows you to define a base estimator which defaults to a DecisionTreeClassifier with a max depth of 1 (Pedregosa et al. 2011). An *estimator* is a simple model that can give a classification. Scikit-learn implements the AdaBoost algorithm to apply heavier weights to specific training estimators with the goal of correcting wrong predictions (AdaBoost). AdaBoost was trained using the train set and the hyperparameter of the number of estimators was optimized using the validation set. The tuning uses a simple loop to identify the optimal *estimator* value. Other implementations such as bayesian model-based optimization or grid search could have been used for tuning as well.

The outcome of the parameter tuning was a value of 98 estimators, out of a range of 1 to 100. The results of tuning are shown in Appendix B. The training time with the 100 *estimators* was quite a bit longer. With more computing power a more accurate *estimator* value may be usable, although overfitting may occur. Although training with 100 *estimators* is computationally expensive, the prediction is quick after an *estimator* is chosen.

The results for AdaBoost using 98 *estimators* were 99.02% for accuracy, 100% for precision and 98.08% for recall on our test set. These values are above our success metrics we defined. Compared to our initial findings, this is a 3.38% increase in accuracy on the

validation set and a 7.03% increase in the test set accuracy. This is due to improved feature selection, training set and hyper parameter tuning. Table 2 below shows the accuracy, precision and recall of the train, validation and test set results.

Table 2: AdaBoost Final Results:

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Train	99.84	99.82	99.86	99.84
Validation	99.69	100	99.39	99.69
Test	99.02	100	98.08	99.03

Support Vector Machines:

A Support Vector Machine (SVM) is a robust classifier, which Mason Lane prepared using Scikit-learn. With Scikit-learn's methods, we planned to fit and test the SVM with our dataset (Support Vector Machines). However, the SVM could not read our extracted dataframe, which needed to be rearranged to adequately fit the SVM. Before fixing the fitting algorithm's input, the classifier would predict 1 for all samples - resulting in a 50% error.

After reorganizing the dataset for the SVM, the fitting method ran 64 different versions of itself, with varying parameters for the *rbf* kernel. It's results would reveal the combination of parameters which best predicted the instruments. We chose the *rbf* kernel, as it did not hurt performance enough to justify using the *linear* kernel - which trains faster at a cost to accuracy (RBF SVM parameters). *Gamma* and *C* were the main parameters adjusted. *Gamma*, which contains the model at smaller numbers; and *C*, which accepts a smaller margin at larger numbers (Navlani, 2019). A smaller margin generates a more complex decision function at the cost of performance. Our goal was to find the values of *gamma* and *C* which did not overly constrain or loosen the model, while maintaining adequate levels of performance. *Gamma* and *C* were adjusted logarithmically from 100 to 0.00001. We found that a *C* of 100, and a *gamma* of 0.00001 delivered the best results.The SVM predicts strings and keyboards with an accuracy of 99.85% using the Nsynth dataset as shown in Table 3. Though its prediction has a decreased accuracy of ~73.15% when using the RACK dataset. These results are a remarkable improvement compared to the first iteration, which reached a minimum error of ~25%.

Table 3: SVM Final Results:

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Train	99.93	99.97	99.89	99.93
Validation	99.88	100	99.75	99.87
Test	99.84	99.67	100	99.83

KNN:

The last algorithm that we trained was a KNN Classifier implemented by Kegan Lavoy using Scikit-learn. The Scikit-learn KNN Classifier uses a ball tree algorithm along with the *minkowski metric* to calculate the distances between the neighbours. Other hyperparameters that can be set are the *weights*, the *leaf size* for the ball tree, the power parameter *P* for the *minkowski metric*, and *n_neighbours*. For our program the hyperparameters were set as follows: *n_neighbour*s(*K*) = 27 for NSynth dataset, 17 for RACK dataset, *weights* = uniform, *leaf size* = 30, and *P* = 2 (equal to euclidean distance when minkowski power = 2).

The hyperparameter *n_neighbours* or *K* was optimized manually and converged on the best accuracy when *K* = 27 and 17 for the NSynth and RACK datasets respectively. This model was the only one to not meet our success criteria, most likely due to it being the most simplistic of the models used. However, it still gave good results of 94.44% accuracy, 93.46% precision, and 95.33% recall.

Table 4: KNN Final Results:

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Train	94.48	96.32	92.90	94.58
Validation	94.59	93.86	95.26	94.55
Test	94.44	93.46	95.33	94.39

Results:

Of our classifiers used, KNN and Binary Decision Tree were the only classifiers to not meet our expectations for accuracy, precision, recall and F1 score. Of the other classifiers used, SVM was the most successful looking at accuracy, with a 99.83%. SVM exceeded our expectations for accuracy, precision and recall and provided results that are a 2.33% improvement over our expected metrics. Random Forests and AdaBoost were the next most successful with an accuracy of 99.35% and 99.03% respectively. The results shown below in

Table 5 compare the accuracy, precision, recall and F1 score of the results using the test set. Figure 1 provides a graphical view of the accuracy of the different classifiers.

Table 5: Classifier Final Results

Classifier	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Binary Decision Tree	97.22	99.35	95.30	97.28
Random Forests	99.35	100	98.71	99.35
AdaBoost	99.02	100	98.08	99.03
KNN	94.44	93.46	95.33	94.39
SVM	99.84	99.67	100	99.83

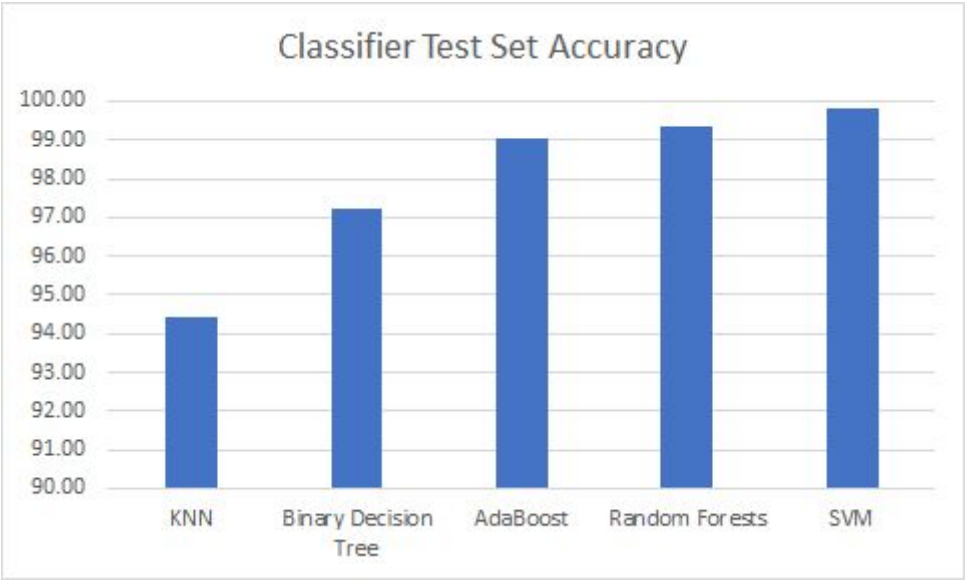


Figure 1: Test Set Accuracy Graph

The classifiers were all used to predict on the RACK dataset as well. The RACK dataset's independently recorded piano and violin sound files provided interesting results. The classifiers were trained on the NSynth dataset and applied to the RACK dataset. The accuracy for the predictions was 73.15% using SVM, AdaBoost and Binary Decision Tree classifiers with similar results for Random Forests and KNN. The dataset contains both short and long violin note audio files. The short violin notes were misclassified as piano which provided the consistent classification error mentioned above. The misclassification is interesting to look at spectral contrast which had previously been a good indicator but had similar values for both piano and violin causing the error. With the removal of the short violin notes, results improved to approximately 95% accuracy overall.

Conclusion:

Classification of piano versus violin was completed using the extracted audio features and various classification techniques. The updated extraction of audio features for Music Information Retrieval (MIR) provided more accurate results. It improved accuracy from an average of 83.30% to 97.97% and a best accuracy of 91.99% to 99.84%. The audio features of spectral contrast and MFCC provided additional information that improved our overall results. Our best performing model prior to the updated feature extraction and training was our AdaBoost model at about an 8% error rate, and the Binary Decision Tree was comparable second at 11%. These rates were improved using the features identified for MIR and tuning of hyperparameters.

The difficulty with optimizing the SVM implementation is a potential downside of SVMs although when tuned and implemented the results can be positive. SVM's reliance on kerneling, such as the use of the *rbf* kernel, increases computational requirements as dimensionality is added. The relative ease of AdaBoost and Random Forests, especially when using the Scikit-learn library function, provide similar results with a faster implementation time. The strengths of AdaBoost and Random Forests provide an advantage over SVM in terms of usability. Extending the classification problem to multiclass classification could identify the more appropriate classifier for the broader, more useful problem of general instrument classification.

Future Direction:

Future work could be used to extend the model to multi-class classification problems, as well as optimize audio feature extraction more. Feature extraction could be updated to remove features that do not increase the results of the classifiers to help optimize and clean the extracted data. Adding a track splitting feature to separate instruments from audio tracks could be helpful for implementation with instrument identification systems for songs. The results of our metrics with SVM and the train and prediction time are accurate and extending this to multiclass classification would be interesting to determine if the high accuracy is maintainable with more classes.

The RACK dataset could be cleaned and expanded to include more samples and have samples be more consistent. This would provide a smaller alternative for sound classification datasets. Annotations could also be updated and improved upon to ease use of the dataset.

References:

AdaBoost. (n.d.). Retrieved March 13, 2020, from https://scikit-learn.org/stable/modules/ensemble.html#adaboost

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Eck, D., Simonyan, K., & Norouzi, M. (2017). Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders. Retrieved March 24, 2020, from https://arxiv.org/abs/1704.01279v1

Huilgol, P. (2019) Accuracy vs. F1-Score Retrieved April 1, 2020, from https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2

Jiang, D.-N., Lu, L., Zhang, H.-J., Tao, J.-H., & Cai, L.-H. (2002). Music type classification by spectral contrast feature. *Proceedings. IEEE International Conference on Multimedia and Expo*. doi: 10.1109/icme.2002.1035731

Kawwa, N. (2019, April 29). Can We Guess Musical Instruments With Machine Learning? Retrieved March 13, 2020, from https://medium.com/@nadimkawwa/can-we-guess-musical-instruments-with-machine-learning-afc8790590b8

KNeighbors Classifier. (n.d.). Retrieved March 12, 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Koehrsen, W. (2018, August 31). An Implementation and Explanation of the Random Forest in Python. Retrieved from https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76

Müller, M., Ewert, S. (2011) “Chroma Toolbox: MATLAB implementations for extracting variants of chroma-based audio features” International Society for Music Information Retrieval Conference (ISMIR) Retrieved March 12, 2020, from https://ismir2011.ismir.net/papers/PS2-8.pdf

Nagawade, M. S., & Ratnaparkhe, V. R. (2017). Musical instrument identification using MFCC. *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. doi: 10.1109/rteict.2017.8256990

Navlani, A. (2019, December 27). (Tutorial) Support Vector Machines (SVM) in Scikit-learn. Retrieved March 9, 2020, from https://www.datacamp.com/community/tutorials/svm-classification-scikit-learn-python

Owers, J. (2016). *An Exploration into the Application of Convolutional Neural Networks for Instrument Classification on Monophonic Note Samples* (Master's thesis, University of Edinburgh, Edinburgh, United Kingdom). Retrieved from https://jamesowers.github.io/files/thesis.pdf

Patil, S. D., & Sanjekar, P. S. (2017). Musical instrument identification using SVM, MLP& AdaBoost with formal concept analysis. *2017 1st International Conference on Intelligent Systems and Information Management (ICISIM)*. doi: 10.1109/icisim.2017.8122157

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python Journal of Machine Learning Research, 12, 2825–2830.

Random Forest Classifier. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

RandomizedSearchCV (n.d) Retrieved April 2, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

RBF SVM parameters. (n.d.). Retrieved April 2, 2020, from https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

Shung, K (2018) Accuracy, Precision, Recall or F1? Retrieved April 1, 2020, from https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

Smith, J. O. (2007). Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications. Retrieved March 12, 2020, from https://ccrma.stanford.edu/~jos/st/Spectrograms.html

Seipel, F. (2018). Music Instrument Identification using Convolutional Neural Networks. Retrieved March 20, 2020, from

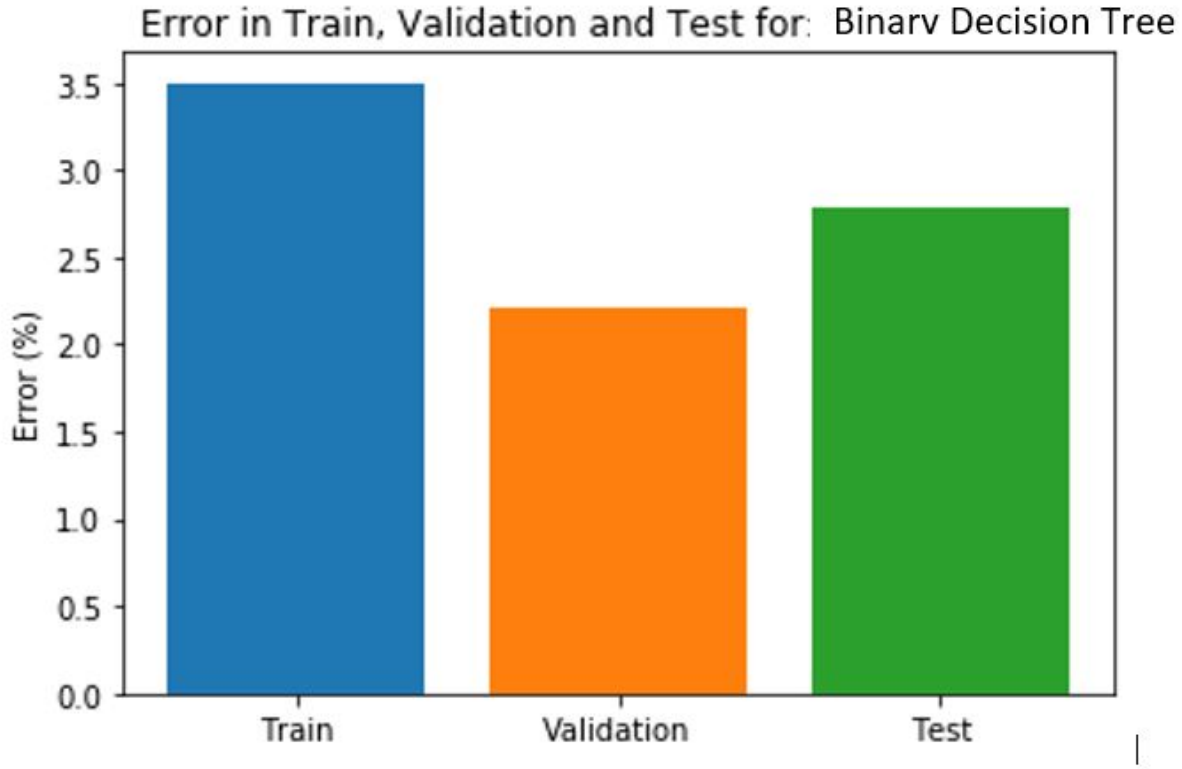
https://www2.ak.tu-berlin.de/~akgroup/ak_pub/abschlussarbeiten/2018/Seipel_MasA.pdf

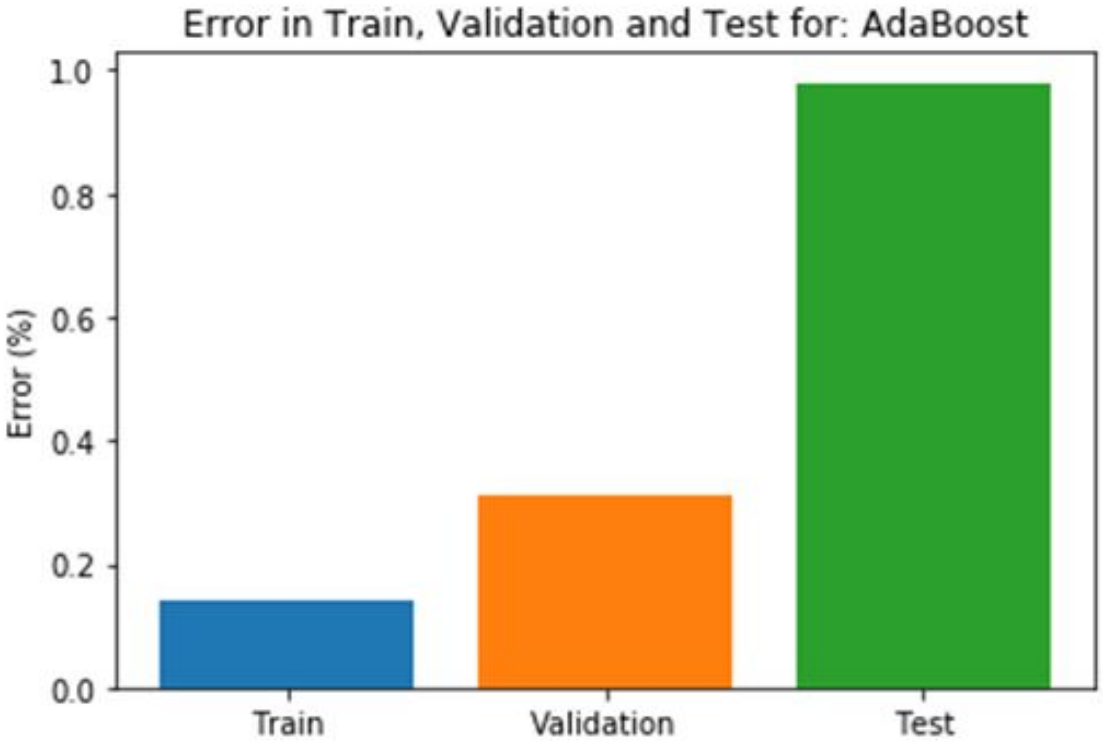
Support Vector Machines. (n.d). Retrieved April 2, 2020, from https://scikit-learn.org/stable/modules/svm.html#svm-classification

Xu, M., Duan, L.-Y., Cai, J., Chia, L.-T., Xu, C., & Tian, Q. (2004). HMM-Based Audio Keyword Generation. *Advances in Multimedia Information Processing - PCM 2004 Lecture Notes in Computer Science*, 566–574. doi: 10.1007/978-3-540-30543-9_71

Xu, M., Maddage, N., Xu, C., Kankanhalli, M., & Tian, Q. (2003). Creating audio keywords for event detection in soccer video. *2003 International Conference on Multimedia and Expo. ICME 03. Proceedings (Cat. No.03TH8698)*. doi: 10.1109/icme.2003.1221608

Appendix A: Classifier Error Graphs

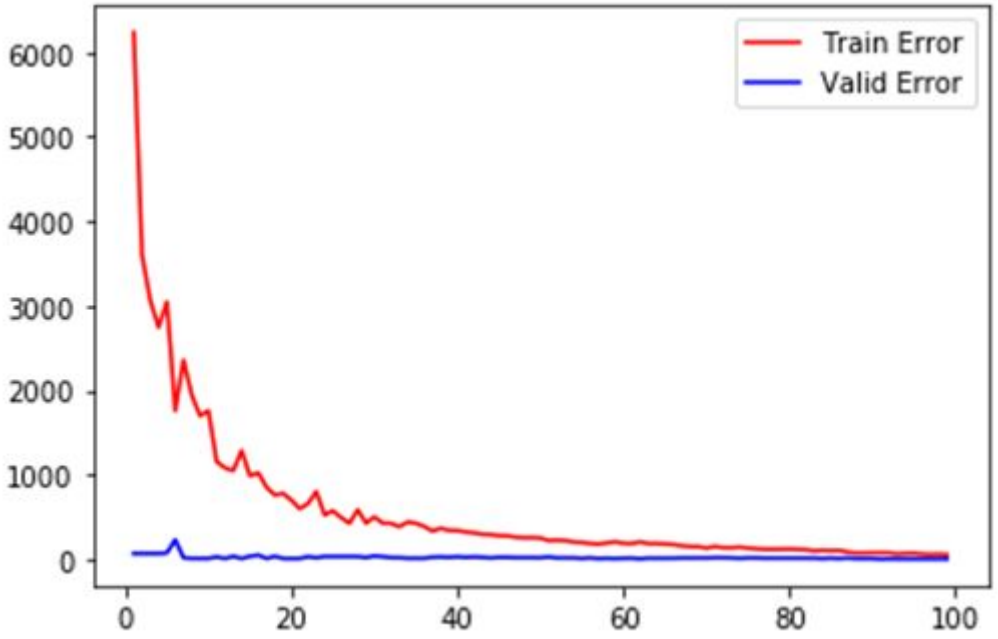



Error in Train, Validation and Test for: AdaBoost
Error (%)
1.0
0.8
0.6
0.4
0.2
0.0
Train
Validation
Test


Error in Train, Validation and Test for: KNN
Error (%)
5
4
3
2
1
0
Train
Validation
Test


Error in Train, Validation and Test for: SVM
Error (%)
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Train
Validation
Test

Appendix B: AdaBoost Hyperparameter Tuning



Prediction errors versus AdaBoost Estimator value, range 1 to 100.

Appendix C: Classifier Implementations

Prediction Models

April 3, 2020

0.1 Prediction Models

Predict the instrument based on the input csv files from build dataset. Uses various sklearn classification models and plots error%

[8]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import tree
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import cross_val_score
from sklearn import svm
from sklearn import metrics
```

0.1.1 Load the Datasets

Load the data from the created csv files.

[9]:
```
df_train = pd.read_csv('./Extracted CSVs/train.csv')
df_valid = pd.read_csv('./Extracted CSVs/valid.csv')
df_test = pd.read_csv('./Extracted CSVs/test.csv')
```

0.1.2 Prepare the Datasets

Balances the data so that both classes have equal entries, to help balance the classification of our implementations.

Seperates the target and data of the classes.

[10]:
```
def equalize_data(class1, class2):
    class1_copy = class1
    class2_copy = class2

    if (class1_copy['y_harmonic'].count() < class2_copy['y_harmonic'].count()):
```

```
        while (class1_copy['y_harmonic'].count()<class2_copy['y_harmonic'].
 ↪count()):
            temp = [class1_copy, class1]
            class1_copy = pd.concat(temp)
        class1_copy = class1_copy[:class2_copy['y_harmonic'].count()]
    else:
        while (class2_copy['y_harmonic'].count()<class1_copy['y_harmonic'].
 ↪count()):
            temp = [class2_copy, class2]
            class2_copy = pd.concat(temp)
        class2_copy = class2_copy[:class1_copy['y_harmonic'].count()]
    return (pd.concat([class1_copy, class2_copy]))

def count_errors(predictions):
    count = 0;
    for pred in predictions:
        if (pred[0] != pred[1]):
            count += 1
    return count;

# balance the datasets
input_df_train = equalize_data(df_train[df_train['target'] == 0],␣
 ↪df_train[df_train['target'] == 1])
input_df_valid = equalize_data(df_valid[df_valid['target'] == 0],␣
 ↪df_valid[df_valid['target'] == 1])
input_df_test = equalize_data(df_test[df_test['target'] == 0],␣
 ↪df_test[df_test['target'] == 1])

# randomize datasets
input_df_train = input_df_train.sample(frac=1).reset_index(drop=True)
input_df_valid = input_df_valid.sample(frac=1).reset_index(drop=True)
input_df_test = input_df_test.sample(frac=1).reset_index(drop=True)

# Separate the target and the columns
y_train = input_df_train['target']
x_train = input_df_train.drop(labels=['target'], axis=1)
#Remove the string column
x_train = x_train.drop(x_train.columns[0], axis=1)

# Separate the target and the columns
y_valid = input_df_valid['target']
x_valid = input_df_valid.drop(labels=['target'], axis=1)
#Remove the string column
x_valid = x_valid.drop(x_valid.columns[0], axis=1)

# Separate the target and the columns
y_test = input_df_test['target']
```

```
x_test = input_df_test.drop(labels=['target'], axis=1)
#Remove the string column
x_test = x_test.drop(x_test.columns[0], axis=1)

data_length_train = len(x_train)
data_length_valid = len(x_valid)
data_length_test = len(x_test)
```

0.1.3 Format Results

These functions provide formatting for results. Using Graphs to display error rate and calculate accuracy, precision and recall of each classifier.

Note: These functions use confusion matrices from SKLearn. These are in the form of: Uses [[True Positives,False Positives], [False Negatives, True Negatives]]

References: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

[11]:
```
"""
Name: display_metrics
Input:
    conf_matrix_train: Confusion Matrix from train set prediction
    conf_matrix_valid: Confusion Matrix from validation set prediction
    conf_matrix_test: Confusion Matrix from test set prediction
Returns: None

Calls functions to print the accuracy,precision and recall of a model


Uses Confusion Matrix in form of:
[[True Positives,False Positives],
[False Negatives, True Negatives]]
"""
def display_metrics(conf_matrix_train, conf_matrix_valid, conf_matrix_test,␣
↪classifier):
    print(classifier + ' Results')
    display_error_graph(conf_matrix_train, conf_matrix_valid, conf_matrix_test,␣
↪classifier)
    print('Accuracy:')
    print_accuracy(conf_matrix_train, conf_matrix_valid, conf_matrix_test)
    print('Precision:')
    print_precision(conf_matrix_train, conf_matrix_valid, conf_matrix_test)
    print('Recall:')
    print_recall(conf_matrix_train, conf_matrix_valid, conf_matrix_test)

"""
Name: display_error_graph
```

```
Input:
    conf_matrix_train: Confusion Matrix from train set prediction
    conf_matrix_valid: Confusion Matrix from validation set prediction
    conf_matrix_test: Confusion Matrix from test set prediction
    classifier: Name of Classifier
    fig_num: Figure number to use
Returns: None

Outputs a graph of the correct / total cases percentage for train, valid and␣
 ↪test
"""
def display_error_graph(conf_matrix_train, conf_matrix_valid, conf_matrix_test,␣
 ↪classifier):
    num_errors_train = conf_matrix_train[0][1] + conf_matrix_train[1][0]
    num_errors_valid = conf_matrix_valid[0][1] + conf_matrix_valid[1][0]
    num_errors_test  = conf_matrix_test[0][1]  + conf_matrix_test[1][0]

    error_percent_train = round(num_errors_train / data_length_train * 100, 2);
    error_percent_valid = round(num_errors_valid / data_length_valid * 100, 2);
    error_percent_test = round(num_errors_test / data_length_test * 100, 2)

    plt.title("Error in Train, Validation and Test for: " + classifier)
    plt.ylabel("Error (%)")
    plt.bar('Train', error_percent_train)
    plt.bar('Validation', error_percent_valid)
    plt.bar('Test', error_percent_test)
    plt.show()

"""
Name: print_accuracy
Input:
    conf_matrix_train: Confusion Matrix from train set prediction
    conf_matrix_valid: Confusion Matrix from validation set prediction
    conf_matrix_test: Confusion Matrix from test set prediction
Returns: None

Calculates the accuracy of the model:
Correct over total predictions
Accuracy = TP + TN / TP+TN+FP+FN
"""
def print_accuracy(conf_matrix_train, conf_matrix_valid, conf_matrix_test):
    accuracy_percent_train = round((conf_matrix_train[0][0] +␣
 ↪conf_matrix_train[1][1]) / data_length_train * 100, 2)
    accuracy_percent_valid = round((conf_matrix_valid[0][0] +␣
 ↪conf_matrix_valid[1][1]) / data_length_valid * 100, 2)
    accuracy_percent_test =  round((conf_matrix_test[0][0]  +␣
 ↪conf_matrix_test[1][1])  / data_length_test  * 100, 2)
```

```
    print('Train set accuracy: ', accuracy_percent_train, '%')
    print('Validation set accuracy: ', accuracy_percent_valid, '%')
    print('Test set accuracy: ', accuracy_percent_test, '%')

"""
Name: print_precision
Input:
    conf_matrix_train: Confusion Matrix from train set prediction
    conf_matrix_valid: Confusion Matrix from validation set prediction
    conf_matrix_test: Confusion Matrix from test set prediction
Returns: None

Calculates the precision of the model:
What proportion of positive identifications was actually correct?
Precision = TP/TP+FP
"""
def print_precision(conf_matrix_train, conf_matrix_valid, conf_matrix_test):
    recall_train = round(conf_matrix_train[0][0] / (conf_matrix_train[0][0] +␣
 ↪conf_matrix_train[0][1]) * 100, 2)
    recall_valid = round(conf_matrix_valid[0][0] / (conf_matrix_valid[0][0] +␣
 ↪conf_matrix_valid[0][1]) * 100, 2)
    recall_test =  round(conf_matrix_test[0][0]  / (conf_matrix_test[0][0]  +␣
 ↪conf_matrix_test[0][1])  * 100, 2)
    print('Train set precision: ', recall_train, '%')
    print('Validation set precision: ', recall_valid, '%')
    print('Test set precision: ', recall_test, '%')

"""
Name: print_recall
Input:
    conf_matrix_train: Confusion Matrix from train set prediction
    conf_matrix_valid: Confusion Matrix from validation set prediction
    conf_matrix_test: Confusion Matrix from test set prediction
Returns: None


Calculates the recall of the model:
What proportion of actual positives was identified correctly?
Recall = TP/ TP+FN
"""
def print_recall(conf_matrix_train, conf_matrix_valid, conf_matrix_test):
    recall_train = round(conf_matrix_train[0][0] / (conf_matrix_train[0][0] +␣
 ↪conf_matrix_train[1][0]) * 100, 2)
    recall_valid = round(conf_matrix_valid[0][0] / (conf_matrix_valid[0][0] +␣
 ↪conf_matrix_valid[1][0]) * 100, 2)
    recall_test =  round(conf_matrix_test[0][0]  / (conf_matrix_test[0][0]  +␣
 ↪conf_matrix_test[1][0])  * 100, 2)
```

```
    print('Train set recall: ', recall_train, '%')
    print('Validation set recall: ', recall_valid, '%')
    print('Test set recall: ', recall_test, '%')
```

0.1.4 Binary Decision Tree

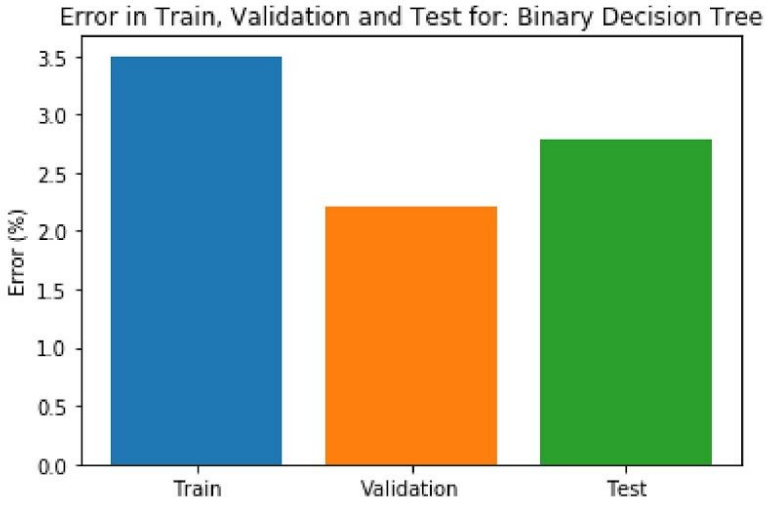
The first algorithm we used was a Binary Decision Tree. The tree was created and fit to the data using a Sci-kit learn decision tree. Sci-kit learn uses an optimised version of the Classification and Regression Trees (CART) algorithm to construct its decision trees. The CART algorithm "constructs binary trees using the feature and threshold that yield the largest information gain at each node."

Reference: https://scikit-learn.org/stable/modules/tree.html

[12]:
```
def simple_decision_tree(max_depth):
    # hyperparameters
    # max_depth of 4 seems to work well for training with validation set

    # create a scikit-learn tree based on validation set
    clf = tree.DecisionTreeClassifier(max_depth=max_depth)
    clf = clf.fit(x_train, y_train)

    # make predictions
    predictions_train = clf.predict(x_train)
    predictions_valid = clf.predict(x_valid)
    predictions_test = clf.predict(x_test)

    # make a confusion matrix
    results_train = confusion_matrix(y_train, predictions_train)
    results_valid = confusion_matrix(y_valid, predictions_valid)
    results_test = confusion_matrix(y_test, predictions_test)
    display_metrics(results_train, results_valid, results_test, "Binary␣
↪Decision Tree")
```

[13]:
```
# fit a tree to our data and plot error
simple_decision_tree(4)
```

```
Binary Decision Tree Results
```




```
Accuracy:
Train set accuracy:  96.5 %
Validation set accuracy:  97.79 %
Test set accuracy:  97.22 %
Precision:
Train set precision:  96.7 %
Validation set precision:  100.0 %
Test set precision:  99.35 %
Recall:
Train set recall:  96.32 %
Validation set recall:  95.76 %
Test set recall:  95.3 %
```

0.1.5 Random Forest

The forest was created and fit to the data using a Sci-kit learn random forest. A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Hyperparameter tuning was done using the Sci-kit learn RandomizedSearchCV function. This takes a range of input values for various hyperparameters and randomly tests different configurations. It saves the best performing model, which is used to make predictions for the test set.

Reference: https://scikit-learn.org/stable/modules/ensemble.html#forest

[23]:
```
# random forest implementation
def random_forest():
    # hyperparameter training. Uses sklearn Random Search
    random_search = {'criterion': ['entropy', 'gini'],
                'max_depth': list(np.linspace(1, 10, 10, dtype = int)) + [None],
                'max_features': ['auto', 'sqrt','log2', None],
                'min_samples_leaf': [1, 2, 3, 4, 10],
                'min_samples_split': [2, 3, 4, 7],
                'n_estimators': list(np.linspace(10, 150, 10, dtype = int))}

    clf = RandomForestClassifier()
    model = RandomizedSearchCV(estimator = clf, param_distributions =␣
 ↪random_search, n_iter = 10,
                                    cv = 4, verbose= 5, random_state= 101,␣
 ↪n_jobs = -1)
    model.fit(x_train,y_train)

    # make predictions
    predictions_train = model.best_estimator_.predict(x_train)
    predictions_valid = model.best_estimator_.predict(x_valid)
    predictions_test = model.best_estimator_.predict(x_test)

    # make a confusion matrix
    results_train = confusion_matrix(y_train, predictions_train)
    results_valid = confusion_matrix(y_valid, predictions_valid)
    results_test = confusion_matrix(y_test, predictions_test)
    display_metrics(results_train, results_valid, results_test, "Random Forest")
```

[24]:
```
# fit a tree to our data and plot error
random_forest()
```

```
Fitting 4 folds for each of 10 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done   6 tasks      | elapsed:   28.2s
[Parallel(n_jobs=-1)]: Done  38 out of  40 | elapsed:  1.3min remaining:    4.1s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  1.3min finished

Random Forest Results
```


```
Accuracy:
Train set accuracy:  100.0 %
Validation set accuracy:  99.51 %
Test set accuracy:  99.35 %
Precision:
Train set precision:  100.0 %
Validation set precision:  100.0 %
Test set precision:  100.0 %
Recall:
Train set recall:  100.0 %
Validation set recall:  99.03 %
Test set recall:  98.71 %
```

0.1.6 AdaBoost

The second algorithm used was AdaBoost which was implemented with Sci-kit learn. Sci-kit learn allows you to define a base estimator which defaults to a DecisionTreeClassifier with a max depth of 1. An estimator is a simple model that can give a classification. Sci-kit learn implements the AdaBoost algorithm to apply heavier weights to specific training estimators with the goal of correcting wrong predictions.

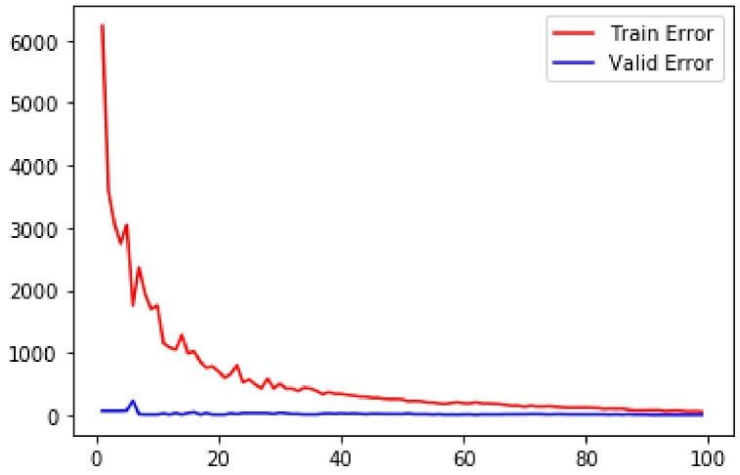
Takes max number of estimators to use, it then uses the train and validation sets to tune the hypeparameters and pick the best outcome.

Reference: https://scikit-learn.org/stable/modules/ensemble.html#adaboost

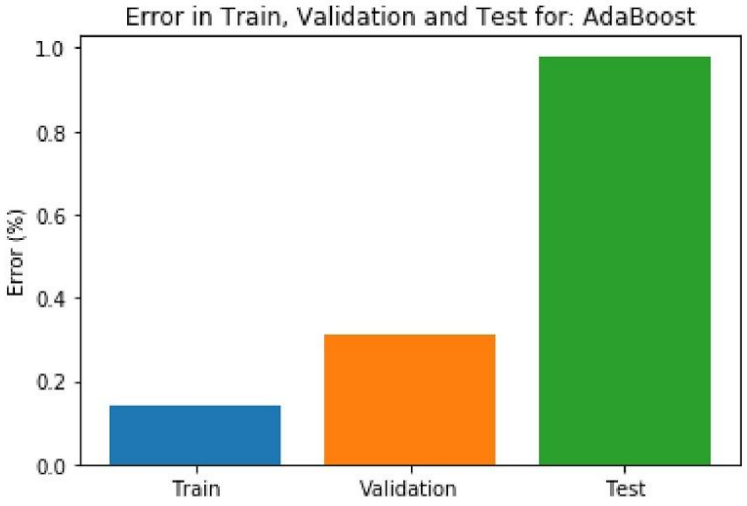
[12]:
```
def ada_boost(max_estimators):
    train_results = []
    valid_results = []
    #Tune the hyperparameter estimators
    for estimators in range(1,  max_estimators):
        classifier = AdaBoostClassifier(
            DecisionTreeClassifier(max_depth=1),
            n_estimators=estimators
        )
        classifier.fit(x_train, y_train)
        predictions_train = classifier.predict(x_train)
        results_train = confusion_matrix(y_train, predictions_train)
        num_errors_train = results_train[0][1] + results_train[1][0]
        train_results.append(num_errors_train)
        predictions_valid = classifier.predict(x_valid)
        results_valid = confusion_matrix(y_valid, predictions_valid)
        num_errors_valid = results_valid[0][1] + results_valid[1][0]
        valid_results.append(num_errors_valid)

    estimator_indices = np.arange(1,max_estimators)
    plt.figure(0)
    plt.plot(estimator_indices, train_results, color='r' , label='Train Error')
    plt.plot(estimator_indices, valid_results, color='b' , label='Valid Error')
    plt.legend()
    plt.show()

    #Combine the error for each set
    error_results = np.arange(1,max_estimators)
    for i in range(0, len(train_results)):
        error_results[i] = train_results[i] + valid_results[i]
    print(error_results)

    best_estimator = np.argmin(error_results) + 1
    print('The Best Estimator found is: ', best_estimator)

    classifier = AdaBoostClassifier(
            DecisionTreeClassifier(max_depth=1),
            n_estimators=best_estimator
    )
    classifier.fit(x_train, y_train)

    predictions_train = classifier.predict(x_train)
    results_train = confusion_matrix(y_train, predictions_train)

    predictions_valid = classifier.predict(x_valid)
    results_valid = confusion_matrix(y_valid, predictions_valid)
```

```
    predictions_test = classifier.predict(x_test)
    results_test = confusion_matrix(y_test, predictions_test)
    display_metrics(results_train, results_valid, results_test, "AdaBoost")
```

[13]:
```
ada_boost(50)
```




```
[6304 3668 3130 2814 3123 1991 2376 1952 1710 1772 1183 1093 1085 1298
 1024 1075  866  797  792  711  611  691  821  560  608  527  462  620
  449  543  460  448  408  455  441  404  361  398  367  372  348  343
  321  312  304  302  284  277  279  272  252  245  241  223  213  205
  190  207  218  203  202  214  202  201  196  191  178  168  168  151
  172  159  156  158  152  144  135  134  135  137  133  130  116  117
  119  116  104   85   86   88   82   86   69   76   79   68   64   67
   60]
The Best Estimator found is:  99
AdaBoost Results
```




```
Accuracy:
Train set accuracy:  99.86 %
Validation set accuracy:  99.69 %
Test set accuracy:  99.02 %
Precision:
Train set precision:  99.85 %
Validation set precision:  100.0 %
Test set precision:  100.0 %
Recall:
Train set recall:  99.87 %
Validation set recall:  99.39 %
Test set recall:  98.08 %
```

0.1.7 KNN

K Nearest Neighbor (KNN) Classifier. The classifier was created using the Sci-kit learn KNeighborsClassifier function. This function uses the minkowski distance to compute the distance between points. After some initial tuning of the hyperparameter K, we found that the best value to use for our dataset was K = 27.

[14]:
```
def KNN(k):
    KNN_Classifier = KNeighborsClassifier(n_neighbors = k)
    KNN_Classifier.fit(x_train, y_train)

    predictions_train = KNN_Classifier.predict(x_train)
```

```
    predictions_valid = KNN_Classifier.predict(x_valid)
    predictions_test = KNN_Classifier.predict(x_test)

    results_train = confusion_matrix(y_train, predictions_train)
    results_valid = confusion_matrix(y_valid, predictions_valid)
    results_test = confusion_matrix(y_test, predictions_test)

    display_metrics(results_train, results_valid, results_test, "KNN")
```

[15]:
```
KNN(27)
```

```
KNN Results
```


```
Accuracy:
Train set accuracy:  94.48 %
Validation set accuracy:  94.59 %
Test set accuracy:  94.44 %
Precision:
Train set precision:  96.32 %
Validation set precision:  93.86 %
Test set precision:  93.46 %
Recall:
Train set recall:  92.9 %
Validation set recall:  95.26 %
Test set recall:  95.33 %
```

1 Support Vector Machines

Our final prediction model - Support Vector Machines, implemented with Scikit-Learn. This Support Vector Machine (SVM) uses the rbf kernel, with a gamma of 0.00001 and c of 100. This configuration was chosen after testing 25 combonations of gamma and c, following logarithmic increments of each. The SVM is very sensitive to the gamma parameter. A smaller gamma than used would fail to capture the shape, being too constrictive. In SVMs, smaller values of c lead to simpler decision functions. Larger values are more complex. We found 100 did not sacrifice accuracy for efficiency.

[16]:
```
def runSVM(k, c, g):
    x_train_svm=x_train
    x_test_svm = x_test
    x_valid_svm = x_valid

    x_train_svm = x_train_svm.to_numpy()
    x_test_svm = x_test_svm.to_numpy()
    x_valid_svm= x_valid_svm.to_numpy()

    y_train_svm = y_train
    y_train_svm = y_train_svm.values

    clf = svm.SVC(kernel=k, C=c, gamma=g)
    clf.fit(x_train, y_train)

    predictions_train = clf.predict(x_train)
    predictions_valid = clf.predict(x_valid)
    predictions_test = clf.predict(x_test)

    results_train = confusion_matrix(y_train, predictions_train)
    results_valid = confusion_matrix(y_valid, predictions_valid)
    results_test = confusion_matrix(y_test, predictions_test)

    display_metrics(results_train, results_valid, results_test, "SVM")
```

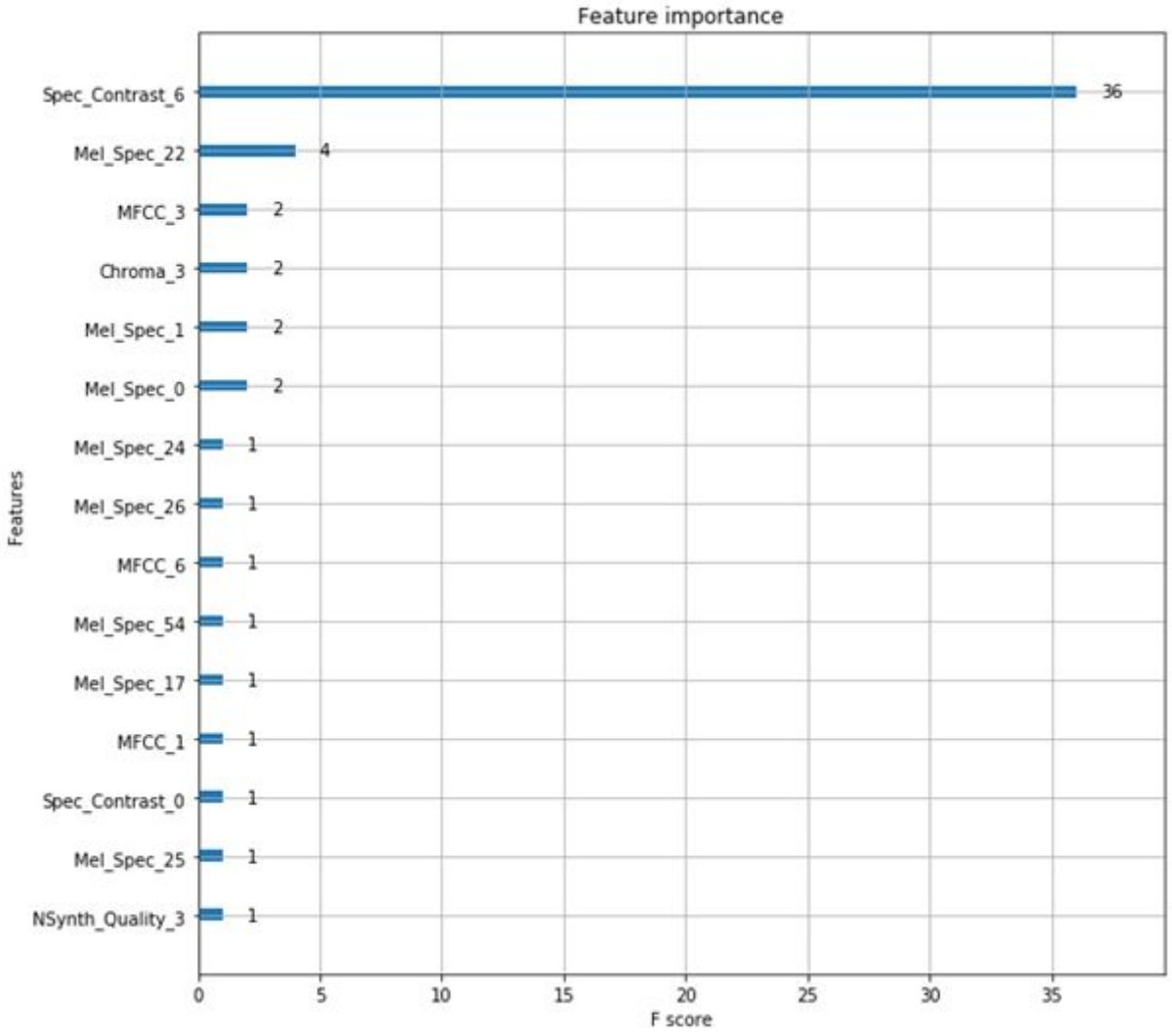
[17]:
```
runSVM('rbf', 100, 0.00001)
```

```
SVM Results
```

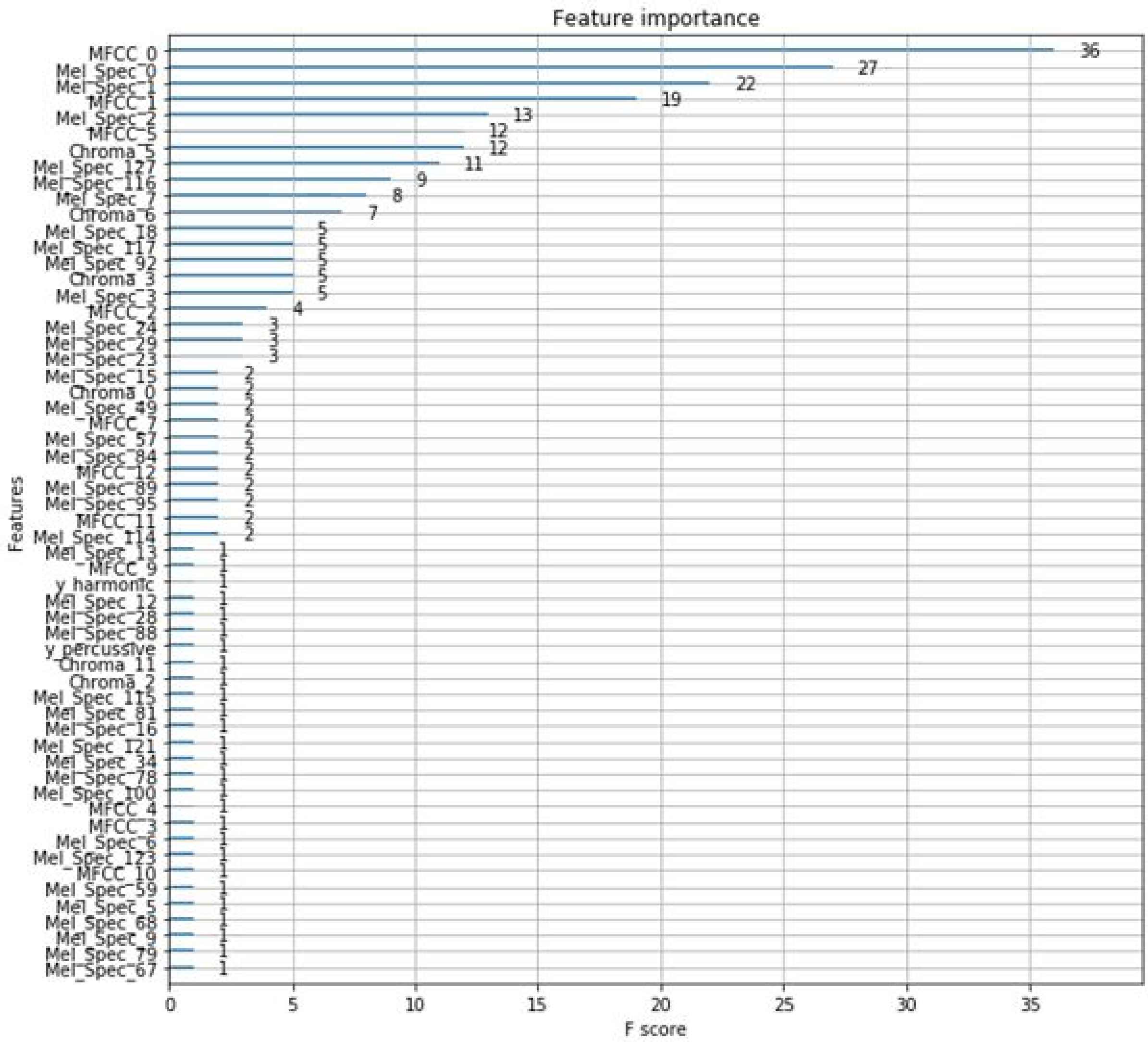

```
Accuracy:
Train set accuracy:  99.93 %
Validation set accuracy:  99.88 %
Test set accuracy:  99.84 %
Precision:
Train set precision:  99.97 %
Validation set precision:  100.0 %
Test set precision:  99.67 %
Recall:
Train set recall:  99.89 %
Validation set recall:  99.75 %
Test set recall:  100.0 %
```

[ ]:

Appendix D: Feature Importance Graphs:



Feature Importance Including Spectral Contrast Feature



Feature Importance Without Spectral Contrast Data