An Integrated Development Environment for Speech-Based Classification

Michael Feld, Christian Müller

German Research Center for Artificial Intelligence
Saarbrücken, Germany
{michael.feld,christian.mueller}@dfki.de

Abstract

This paper presents a new machine learning framework for speech-based classification tasks that was developed in conjunction with the Agender project (age and gender recognition for telephone applications). The main goal of this framework is to provide a completely integrated development environment supporting all processes from design over evaluation to deployment of classification systems. It is intended for both researchers as well as application developers and specializes in audio signals as the resource to be classified. We show that the proposed framework outperforms other tools in several aspects.

1. Introduction

In the last couple of years, speech-based classification methods have increasingly reached a state of maturity where they are commonly employed in large-scale application scenarios, such as in mobile phones or call centers. Therefore, there is also an increasing need for frameworks that allow both machine learning (ML) engineers and application developers to design, test, and deploy their technologies in a way that fulfills their requirements on classification performance, architecture and runtime behavior. This paper introduces AGENDERIDE, a new ML framework specialized in speech data processing that was created to meet this demand and to expand in areas where other tools are lacking.

AGENDERIDE (AGENDER *Integrated Development Environment*) was started and put forward in conjunction with the AGENDER speaker classification approach [1], where it served as a platform to build classification modules for integration in existing applications. In AGENDER, a person's recorded voice is classified according to several characteristics, originally being restricted to age and gender (hence the name), but meanwhile extended to language, noise context, and to even further aspects in the future. It has been found a great way to support user modeling in situations where no or little explicit information about the user is available [2]. Analyses of several large corpora of labeled speakers such as *Timit* and *GlobalPhone* have revealed that there are indeed speech features like *pitch*, *jitter* and *shimmer*, which convey sufficient information to discriminate between seven classes (four age groups with two genders, one age group, children, with no gender discrimination) with a promising accuracy of 63.5%. Feature extraction was performed with the tool *Praat* [3], while initially several algorithms from the *WEKA* [4] ML library were evaluated.

During the evolution of AGENDER and its integration into applications, it quickly became apparent that the prototype implementation used for initial testing and demonstration would not be able to satisfy all of our requirements. To begin with, all major processes from design to deployment of classification modules should be accessible from a single tool. The multiplicity of tools we used before was difficult to link together, had diverse requirements and could not be easily ported to other machines. Moreover, for AGENDER, it should be possible for the user of a module – in this case the developer of another application – to customize the module's parameters and retrain with own data, or even create new modules from scratch. Without the IDE, this was rather unfeasible, in particular because users were expected to have in-depth knowledge of machine learning to use the existing tools. Further goals of the IDE were the ability to build classification modules, the focus on speech data, and support for high data volumes, which was not given with some of the tools we used before.

The remainder of this paper is organized as follows: Section 2 provides an overview over the IDE system; Section 3 then focusses on the corpus management and analysis components; Section 4 details how feature extraction and the representation of features in done; Section 5 and Section 6 then describe a central component of the system: the design of individual classifiers and their compilation into embedded classification modules; Section 7 details the job execution engine. Finally, Section 8 outlines recent extensions of the system as well as future work.

2. IDE Overview

The main graphical GUI component is a *view*. A view contains a set of controls that operate on the same function or a related set of functions. Views fall into several larger categories that help in structuring the development tasks: The *Corpus* category describes functions that have to do with managing the speech databases, including meta information such as timestamp, class labels or segments. The *Features* category is all about computing various types of features from the speech data, storing and managing it. The category *Classifiers* contains the tools for creating and testing classifiers, which can work on corpus file lists or features. Then there is the *Module* category, which hosts the Embedded Module designer and compiler. The *Project* category is made up of general settings and configuration tools like database connection setup. The *Jobs* category is used to create and manage background tasks from within the IDE, including network cluster set-up and monitoring. This paper picks up several of these categories in explaining the IDE's features in detail in the following sections.

AGENDERIDE uses *Multiple Document Interface (MDI)* layout for its GUI, i.e. there is a parent window representing the whole workspace, and a child window fully contained within for each specific view. This allows the user to organize the workspace to provide the best productivity for his or her needs with respect to the current task. A main menu bar provides access to each of the views grouped by category. Fig. 1 displays AGENDERIDE's parent window with some views opened.

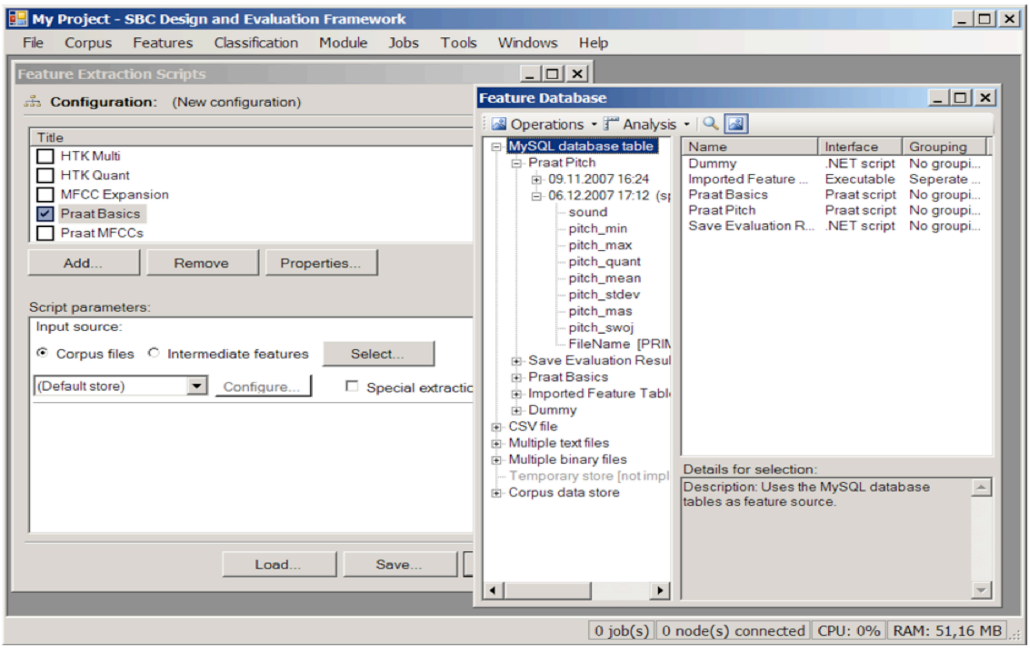

Figure 1: Main window of AGENDERIDE with two views opened.

AGENDERIDE heavily relies on the concept of *scripts* to facilitate its extensibility. Many kinds of operations allow different implementations from which the user may choose one, and which should be extensible without modifying the IDE's source code. At runtime, the user will be able to select the actual script to be used from all available classes that implement the corresponding interface. There are many scripts already included in the application, but the prominent ability of AGENDERIDE is that scripts can also be written using a code editor from within the application in either C# or Visual Basic .NET.

Results of analysis functions or logs of lengthy operations are made available to the user as *reports*. They can be plain text or HTML, and optionally contain images and other attached files. The IDE has an internal HTML viewer that allows the user to read reports, navigate between them and export them.

3. Corpus Management and Analysis

Before any analysis can be done, the speech needs to be available. A *speech corpus* is a database of related wave samples with annotations. For example, one might create corpora based on the origin of the data or the recording conditions.

A deficiency encountered prior to the introduction of the AGENDERIDE was the absence of a common file format for the speech corpora, so that the audio files often had to be converted manually between different formats multiple times. The IDE imports and converts the files from several popular formats such as raw LPCM, *AU*, *a-law*, and *NIST Sphere*. If a tool requires audio data in a different format, the corpus system will handle the conversion automatically.

Meta data is the information associated with a single speech sample beyond the actual digitized PCM signal. *Basic* meta data are fields like a unique ID, timestamp, and length. This information is present for all files. *Static* meta data are annotations or class labels provided by an external database and do never change. *Dynamic* meta data differs from Static meta data only in the way it is computed, namely using a script function that may be based on one or more other fields. For example, the static field *BirthDate* may contain the exact date of birth of the speaker and *RecordingDate* the date when the sample was recorded. To obtain a representation more suitable for classification, a dynamic meta field *Age* is created and computed from the other two fields using a simple script.

Corpus file filters based on meta data represent an efficient way of selecting subsets of corpus files, e.g. for feature extraction or classification. They are implemented as filter rules that restrict the choice of corpus files by certain conditions. For example, if we want to consider only adult speakers between 25 and 60 years, a meta-based filter rule could be written as:

$$age >= 25 \; AND \; age <= 60$$

4. Feature Extraction and Storage

Features are the pieces of information on the basis of which a classifier makes its decisions, hence choosing the right features is a key in creating a good classifier [5]. The basic storage unit for features both in-memory and on disk is called a *feature table*. A feature table has one column for the primary key and one column for each feature that it contains. The type of primary key varies depending on the semantics of the features extracted, and can for instance be the file ID, class label or segment ID. Feature tables also support `null` values in their records, which denote a missing feature value.

As there can be many files and feature tables, the amount of information that has to be processed and stored can become very large. To handle this volume, AGENDERIDE supports multiple feature storage providers. A storage provider handles the following aspects of feature access: storage of table descriptions, storage of the actual feature data, listing and retrieval of feature tables, and caching of feature data. There are several advantages to multiple storage providers. First, the performance of a storage provider depends on the structure and amount of feature data and the physical platform used for storage. For example,

some storage engines work well with many features but are not suitable for storing a lot of records. Also, some feature extractions like n-gram builders do not produce a fixed vector, so they cannot easily be stored in a normal database table.

The feature extraction is done by running scripts. AGENDERIDE can run executables, integrated and custom scripts. For executables, command line parameters can be defined, which may also be feature values, script-generated values, temporary files, input files or output files. Besides that, an output parser can be specified, which parses the information returned by the tool into a table. AGENDERIDE is able to handle different script call semantics, e.g. whether the script is called for each record or handles all records in one call. This affects the input sent to the script and how the output is treated.

A feature extraction configuration consists of one or more predefined feature extraction scripts and parameters for each script. The input to a script can either be a set of audio files on disk, an existing feature table or the output of another script. When using files on disk, the file format, audio format and additional transformations can be specified. For the output, a storage provider can be chosen by the user. The order in which scripts are executed is derived from dependencies between scripts.

Large feature tables do not only pose challenges to storage, but also to retrieval of their contents. Some popular ML frameworks like *WEKA* require all features of a table to be kept in memory for processing. This can limit the type of experiments that can be performed with these tools considerably. Nevertheless, it usually is the fastest way of accessing features. To combine both aspects, feature tables in AGENDERIDE support two ways of data retrieval: table-based and enumerator-based. The table-based method reads all records into a single table structure. Traversal of its rows is extremely fast, especially because data is kept in RAM. The uni-directional enumerator-based approach maintains a pointer to the only record that is kept in memory. Although it is up to several times slower because the data structures need to be updated for each record, a number of optimizations have been made to counter this.

When working with features, it is often not sufficient to use the tables in the form they were created. For this reason, there is a special *compound* feature table, which is accessed like any other feature table, but is dynamically created from other tables. It does not contain any actual data, but rather retrieves and parses the data from the underlying tables on-the-fly while it is being read. It is also possible to build compound tables on top of other compound tables, which is similar to the concept of operator chaining known from *YALE* [6].

AGENDERIDE supports horizontal (or feature) joining and vertical (or instance) joining of feature tables. Horizontal joining is performed when multiple tables contain different features of the same instances, while vertical joining occurs when several tables store identical features, but differ in primary keys. It is also possible to cope with advanced combinations of both, which may require specification of some rules for conflict resolution. Joining of feature tables is done incrementally when using the enumerator-based access and with complexity $O(n)$. This is possible because all feature tables are sorted by their primary keys.

A topic that is sometimes neglected in other tools is that of data archival. Doing this manually is both error-prone and time-consuming. Even if done carefully, a feature table contains much meta information that is essential for describing its contents that a single label such as the filename cannot hold it all, and all too often the user has to look at the actual data to determine – or sometimes guess – its attributes. As we consider data management an important subject, AGENDERIDE has ample built-in support for it. First of all, when a feature table is created, a cache of meta information, e.g. creation date, source corpora and files, and audio format, is stored with it. To browse features, the Feature Explorer depicted in Fig. 1 provides an explorer-like UI where features can be scanned and selected. There is also a search function for looking for features with specific criteria across all storage providers.

5. Classifier Design

Based on features, classifiers can be trained and evaluated from within the GUI. AGENDERIDE distinguishes between *Design-Time classifiers* and *Runtime classifiers*. Design-time classifiers are only part of the development framework and implemented as .NET scripts, while runtime classifiers are the ones that are embedded into the final application as C++ code and controlled via their API. Runtime classifiers are created from existing design-time classifiers and use the stored model. Design-time classifiers support a more sophisticated interface, including conversion into runtime classifiers and serialization.

For every classification problem, a design-time classifier is created. Before it can be trained, the features have to be selected from all available data sources using the Feature Explorer. It is also possible to apply dynamic post-processing steps such as normalization and custom scripts, and to filter files based on corpus and meta properties. There are also some filters which work purely on the file list, e.g. to select random sets for cross-evaluation. Then, the class property is picked from the list of corpus meta properties. Finally, the classification algorithm is chosen.

AGENDERIDE allows custom classification algorithms to be written in the integrated editor. There are two types of classifiers: Binary classifiers, which can only decide on the class membership of an instance for a single class label, and multi-label classifiers, which decide between multiple labels of class. The difference between the two is mostly a semantic one, but nonetheless important, especially for evaluation and visualization. However, a multi-label wrapper can be created for a set of binary classifiers. The interface for a classifier is depicted in Fig. 2. As can be seen from this UML chart, the classification method returns an array of arrays of floating-point numbers. The inner array stores scores for the individual classes, with binary classifiers only using a single element, and the outer array encompasses the test samples for use with batch classification.

The IDE features a basic set of evaluation functions for classifiers. Both design-time classifiers as well as runtime classifiers can be evaluated. The latter is especially useful because it enables the user to benchmark the classification performance of the final application.

Evaluation results are archived in the same way as feature data. From the results, reports in different formats (e.g. text and HTML) can be created, which include a detailed confusion matrix, precision, recall, error rate, ROC curves [7], and other statistics.

During evaluation, a so-called score normalization can be performed. This is useful when classifier scores need to be post-processed in an interoperable manner, i.e. using the normalized range [0; 1].

The evaluation results can also be written to a new feature table. This offers the possibility to create n-th order meta-classifiers, which are trained on the results of other classifiers. For example, to support the gender-dependent aging concept from AGENDER, one could create two classifiers for age, one

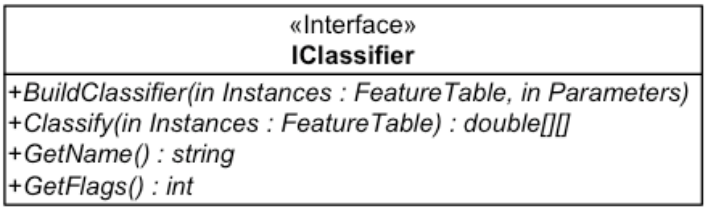

Figure 2: Interface which is implemented by all classification algorithms.

for gender, and then train a third "final" age classifier on the evaluation results of the other three classifiers, using the true age as class label.

6. Embedded Module Development

The idea of Embedded Classification Modules was already described in related work [8]. It is born from the need for a fast, compact and portable solution for classification modules. Not only are the modules embedded in the sense that one file contains all program code and models, but they are also optimized for use in embedded scenarios such as mobile devices an automotive scenarios.

Classification modules support several interfaces, most of which are optional. The "core" interface, which is the most compact version of the code, can be statically compiled into C or C++ programs and is 100% embedded into the host application. The DLL interface is made up of a single .dll file that can be called from C, C++ or any other language supporting DLL calls. In addition, wrappers for .NET and Java are available.

A module consists of components, which are created and plugged together in the IDE. Components can be classifiers, feature extractors, pre- and post-processing components and so on. The dependencies that are introduced by connecting the modules will determine how the final classification pipeline looks. Once the module design is complete, a fully automated build process generates and compiles the code and the desired interfaces.

7. Job Execution Engine

In Machine Learning, we often encounter processes like feature extraction, classifier training, and classifier evaluation, that need a considerable amount of time to complete, either because they involve large amounts of data or because they are computationally expensive. In AGENDERIDE, a job execution engine is responsible for executing these so-called jobs and for distributing the commands to different nodes on a cluster, which allows time-consuming computations to be passed to other machines[1]. Moreover, it allows the user to configure and plan several processes in advance and then run them sequentially.

Remote execution is facilitated through the AGENDERIDE Cluster Service, which is essentially a platform-independent TCP server implemented in .NET. It has been successfully employed on a Linux cluster with the *Mono* runtime. When the job execution engine receives new commands, it waits until a node is idle and then sends the command to that node. The result of the command is returned to the engine using a back-channel in the service. As operating system and file paths may differ between the IDE and nodes, command lines cannot be transferred 1:1. Therefore, the IDE applies a command packaging method.

[1]Currently, only executable commands can be run remotely.

8. Recent Extensions and Future Work

Recently, the system has been extended to support the GMM/SVM Supervector approach, which is nowadays commonly applied in speech classification research. Again, as a prototype application, it has been used for speaker age recognition. However, results on that study have not been published yet by the time of writing.

The GMM-SVM supervector approach was proposed by [11] and later adopted for speaker recognition. It was first applied to the problem of speaker age recognition by [12]. The approach combines the strengths of the generative Gaussian Mixture Model with Universal Background Model (GMM-UBM) approach and the discriminative power of large-marginal methods like the Support Vector Machine (SVM).

The version of the GMM-SVM Supervector system supported by IDE is displayed in Figure 3. From training and background data, frame-by-frame features are extracted. For each sample a GMM is trained. A single, large GMM is trained for all background data. The class-specific GMMs are not trained on the relatively short training data alone but derived from the background model using the Maximum A Posteriori (MAP) method. The resulting GMMs are never applied (tested) in the conventional sense. Instead, the stacked means are extracted and used as input features for the backend SVM. In this way, for the every class, one feature vector of dimensionality *number of mixtures* times *number of coefficients* is obtained. To compensate inter-class variability, certain "nuisance" dimensions of the resulting supervector space are projected out. This is done on the basis of a the ratio between the within-class and between-class variance of all dimensions. Features are afterwards normalized. One SVM is trained for every acoustic event in a one-against-all fashion i.e. training vectors from one class (e.g. CHILDREN) is used as positive examples and the training vectors of all other events are used as negative examples. The bias resulting from a larger negative training set is compensated by weighing the training errors on the positive cases higher. The development test set one and two as well as later the blind evaluation data is processed analogously from the feature extraction step until the normalization step. Scores from all SVMs are obtained. The highest score is taken to determine the "winner"-class for the objective function *accuracy*.

In the previous sections we have seen many of the features that comprise AGENDERIDE and make it an excellent platform for speech-based classification. Although all of the described parts are working, the application itself is still in a state of development as more feedback from other projects is needed to improve usability. One such area for improvements is the designer GUI for configurations such as that of Embedded Modules and feature extractions. Early responses indicate that a graphical designer, i.e. one that supports dragging and dropping of components and connecting them via lines, would be

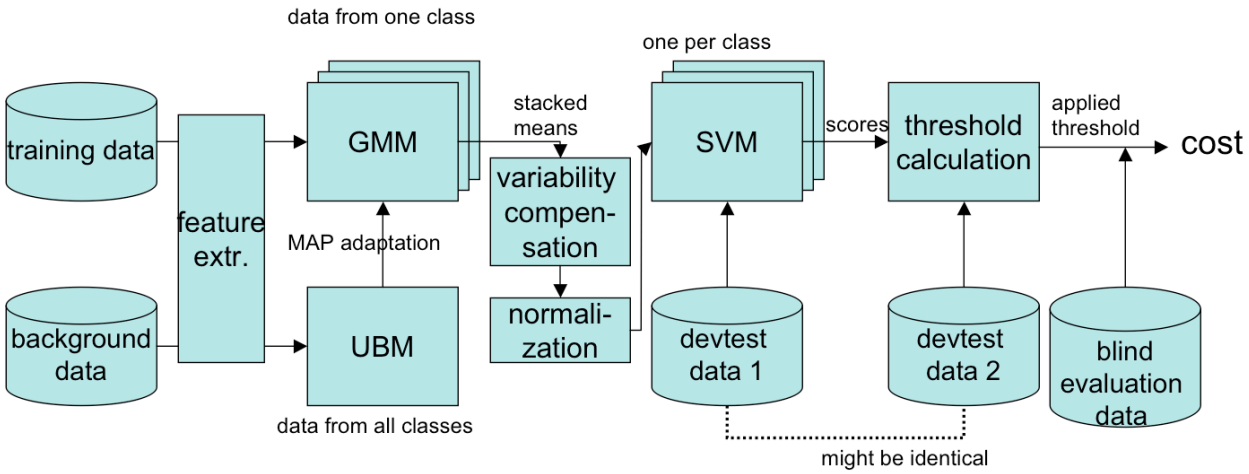

Figure 3: The version of the GMM-SVM supervector approach for speaker age recognition supported by out IDE.

preferred over the current list-and-buttons-style interfaces for ease of use and improved readability. In addition, there are still some technical features which have not been implemented yet, but are closely considered for further enhancement resulting in additional value of the framework. Examples are unsupervised learning methods, meta classifiers, and integration of existing classifier packages like *WEKA*.

9. References

[1] C. Müller, "Zweistufige kontextsensitive Sprecherklassifikation am Beispiel von Alter und Geschlecht [Two-layered Context-Sensitive Speaker Classification on the Example of Age and Gender]," Ph.D. dissertation, Computer Science Institute, University of the Saarland, Germany, 2005.

[2] M. Feld and G. Kahl, "Integrated Speaker Classification for Mobile Shopping Applications," 2008, to appear in "Proceedings of Adaptive Hypermedia 2008", Hannover, Germany.

[3] P. Boersma, "PRAAT, a system for doing phonetics by computer," *Glot International*, vol. 9, no. 5, pp. 341–345, 2001.

[4] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd ed. San Francisco: Morgan Kaufmann, 2005.

[5] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, 2nd ed. New York, USA: Wiley-Interscience, 2000.

[6] O. Ritthoff, R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske, "YALE: Yet Another Machine Learning Environment," in *LLWA 01 – Tagungsband der GI-Workshop-Woche* Lernen – Lehren – Wissen – Adaptivität, ser. Forschungsberichte des Fachbereichs Informatik, Universität Dortmund, R. Klinkenberg, S. Rüping, A. Fick, N. Henze, C. Herzog, R. Molitor, and O. Schröder, Eds., no. 763, Dortmund, Germany, 10 2001, pp. 84–92.

[7] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, June 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1159475

[8] M. Feld, "Embedded Modules for Speaker Classification," submitted to ICSC 2008. Will be removed for camera-ready version if not accepted.

[9] C. Müller and M. Feld, "Towards a Multilingual Approach on Speaker Classification," in *Proceedings of the 11th International Conference on Speech and Computer (SPECOM 2006)*. St. Petersburg, Russia: Anatolya Publishers, 2006, pp. 120 – 124.

[10] M. Feld and C. Müller, "Speaker Classification for Mobile Devices," in *Proceedings of the 2nd IEEE International Interdisciplinary Conference on Portable Information Devices (Portable 2008)*, Garmisch-Partenkirchen, Germany, August 2008.

[11] P. Moreno, P. Ho, and N. Vasconcelos, "A Generative Model Based Kernel for SVM Classication in Multimedia Applications," in *NIPS*, 2003.

[12] T. Bocklet, A. Maier, J. G. Bauer, F. Burkhardt, and E. Nöth, "Age and Gender Recognition for Telephone Applications Based on GMM Supervectors and Support Vector Machines," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Las Vegas, NV, 2008.

[13] F. Metze, J. Ajmera, R. Englert, U. Bub, F. Burkhardt, J. Stegmann, C. Müller, R. Huber, B. Andrassy, J. G. Bauer, and B. Littel, "Comparison of Four Approaches to Age and Gender Recognition for Telephone Applications," in *Proceedings of the 32nd International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2007)*, Honolulu, Hawaii, 2007.