Training Classification Models with Rosette

Version 2.0.9 and later

Publication date 2021-07-27

About Basis Technology

Verifying identity, understanding customers, anticipating world events, uncovering crime. For over twenty years, Basis Technology has provided the underlying analytical components enabling businesses and governments to tackle some of their toughest problems. Our Rosette™ text analytics platform employs a hybrid of classical machine learning and deep neural nets to extract meaningful information from unstructured data. Autopsy, our digital forensics platform, and Cyber Triage, our incident response tool, serve the needs of law enforcement, national security, and legal technologists with over 5,000 downloads every week. KonaSearch delivers natural language search of every field, object, and file in Salesforce all from the same index.

Company headquarters are in Somerville, Massachusetts, with branch offices in Washington, London, Tel Aviv, and Tokyo. For more information, visit www.basistech.com.

Basis Technology Corp.
1060 Broadway
Somerville, MA 02144
T 617.386.2000
F 617.386.2020
E support@rosette.com
http://support.rosette.com

Table of Contents

1. Introduction

This document guides users of Rosette through the process of training custom classification models, for categorization or sentiment analysis, using the Rosette Classification Field Training Kit (FTK). The resulting models may be used with the categories or sentiment endpoints of Rosette.

Reasons for training a custom model include:

- Supporting a language that Rosette does not currently support
- Increasing accuracy on your particular input data
- Supporting a specific categorization taxonomy for your data or task.

The Rosette Classification Field Training Kit (FTK) allows users to train their own classification models. If you have training data, you can build a machine-learned model. If you don't have training data, you can build a keyword-based model [31] by supplying a few keywords for each class. The keyword-based model requires significantly less labor, but will not work well for categories without clear Wikipedia-based concepts or very fine-grained categories like iPhone6 vs. iPhone7. At this time, the keyword-based model only supports English models.

While the FTK includes command line programs to run the classifiers for quick testing and evaluation purposes, no API is exposed. To integrate the models into a real application, users must configure the Rosette Server runtime API. The FTK can be used to create custom categorization models to use with the `categories` endpoint or to develop sentiment analysis models to use with the sentiment endpoint.

This document starts out by describing how to build a corpus to develop a customized document classifier for categories. We then describe best practices for testing the categorization models you've developed. Since sentiment is a very specific type of classification, we also include a section on developing custom sentiment analysis models. Feel free to jump to the sections describing the task you need to accomplish.

2. Overview of the MATTER Cycle

If you are new to training models for an NLP solution, we highly recommend reading *Natural Language Annotation for Machine Learning: A guide to corpus-building for applications*[1] (NLAML). Throughout this guide, we will assume a development cycle based on Pustejovsky and Stubbs' MATTER cycle:

1. **M**odel: Determine the set of categories of interest.
2. **A**nnotate: Manually assign category labels to a set of documents.
3. **T**rain: Train a classifier using a training set of manually labeled documents.
4. **T**est: Test the classifier on a held-out development set of the manually labeled documents that does not overlap with the training set and perform error analysis.

[1]Pustejovsky, J., & Stubbs, A. (2012). *Natural Language Annotation for Machine Learning: A guide to corpus-building for applications* ,O'Reilly Media, Inc.

5. **E**valuate: Evaluate the classifier on another held out subset of your manually labeled documents.

TIP

The evaluation set is for measuring the performance of the classifier, not for determining what it got wrong. To preserve the experimental utility of your evaluation set, do ***not*** perform error analysis on your evaluation set.

6. **R**evise: Revise your model or consider augmenting your corpus by annotating additional documents.

Don't get discouraged if your results after a single cycle are unsatisfactory. Attaining highly accurate models requires multiple iterations. Be prepared to invest a significant amount of labor collecting raw data, cleaning your data, manually annotating your data, and performing error analysis.

3. Creating a Document Categorization Corpus

When training classification models, the first and most critical step, is collecting and annotating a training and evaluation corpus. The size and quality of your training data will be the key factor in influencing the quality of the resulting model.

3.1. Modeling the Problem

The first step in the MATTER [1] cycle is to define the desired set of categories. Scoping the problem is important. Before starting a MATTER cycle, you need to concretely define the desired set of categories—obviously you can revise your model later, but you need to choose a concrete model as a target. There are several considerations when drawing up your category list:

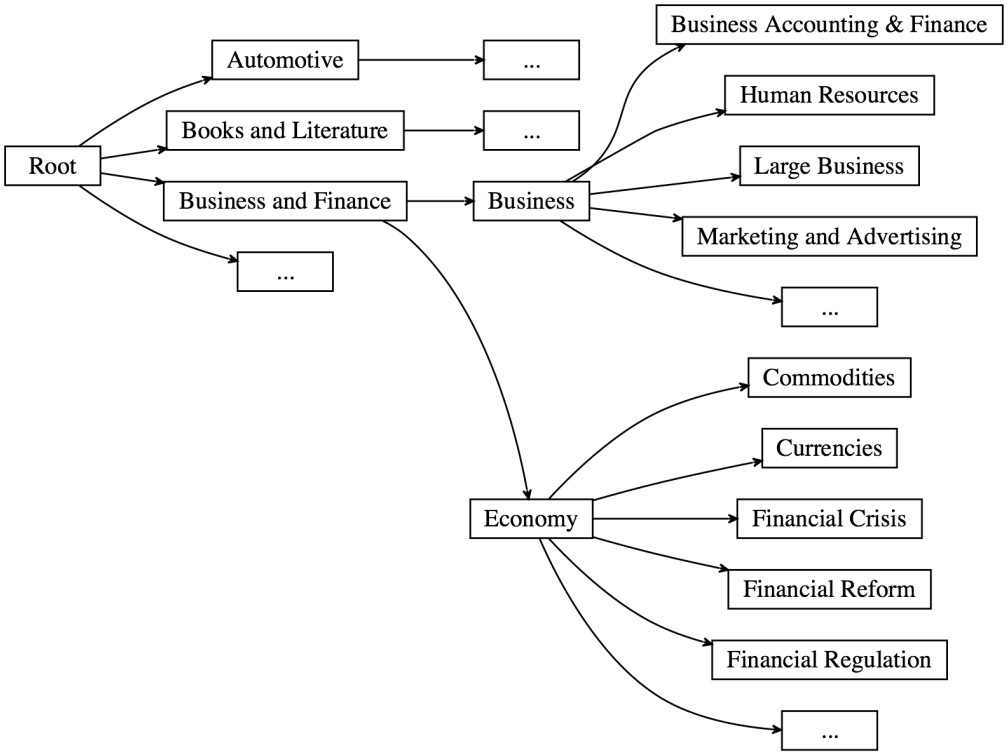
1. The correct number of categories
2. Broad versus fine-grained categories
3. Handling overlapping and mutually exclusive categories

3.1.1. The correct number of categories

Note that the number of defined categories will determine the scope of your effort later on. For instance, if you decide to include thousands of categories in your model, it will be cumbersome to sort through all the possible labels when it comes time to manually annotate documents or perform error analysis. We recommend keeping your model as small as possible while still capturing the distinctions you need within your problem domain. If you discover similar categories, consider combining them. Conversely, If your initial categories are too broad, you may need to split them up. You should aim for enough categories in your model to all possibilities within your problem domain. While it may seem convenient, we don't recommend including a catch-all "misc" category within your model.

3.1.2. Broad versus fine-grained categories

For some domains, a classification problem may lend itself to a taxonomic model with a hierarchy of categories. It is important to consider such hierarchical relationships when specifying your model. Given the hierarchical relationships, do you care about the upper levels of the hierarchy, or only the more specific categories? For instance, one standard taxonomy developed by the Interactive Advertising Bureau is the IAB Tech Lab Taxonomy. The following is a small subset of the IAB taxonomy:



Depending on the depth of your taxonomy, you need to determine at what level of granularity to cut off your model. At a certain point, the distinctions between categories may be too subtle to reliably determine. For instance, considering the subset of the IAB taxonomy above, if you are interested in the domain of Business and Finance, would it be sufficient to distinguish Business from Economy? Or, is it important to descend to more specific subcategories? For a human readable layout of a hierarchical model, it can be extremely helpful to lay out a taxonomy as a graph (such as above), or in a spreadsheet or similar tabular layout (which is how the IAB Tech Lab Taxonomy is distributed). For a machine-readable format, a tabular format is suitable, or consider a data format such as XML or JSON (the IAB Tech Lab distributes a machine-readable JSON version of their taxonomy as well).

3.1.3. Handling overlapping and mutually exclusive categories

At the extremes of being overly specific or overly general, your classification task becomes more difficult. We recommend experimenting with more general categories before creating many fine-grained distinctions. As you create categories, try to keep your categories mutually exclusive and be aware of categories that overlap.

If you find yourself in a situation where you have overlapping categories, consider adding a category to cover the overlap. Imagine you are building a model for classifying cooking recipes, with the categories Baking and Desserts which turn out to overlap considerably. In that case, consider splitting Desserts into Baked Desserts and Non-Baked Desserts. Alternatively, make Baked Desserts a subcategory of Baking. While intuition can be helpful, it is important to make these decisions based on the data instead of what you *think* the categories should look like. Sometimes the data will surprise you, and a model based on preconceptions rather than real data will suffer accordingly. Note that in the case of overlapping labels you may be interested in multi-label classification, where each document may be assigned to multiple categories.

If you discover mutually exclusive categories, you may be interested in creating multiple classifiers and combining their outputs. For instance, you may want one model to classify document topics, but another model to classify emotions expressed in documents. These are mutually exclusive because any given

emotion can be expressed along with any given topic. Separating these categories with individual models may be the cleanest approach. It still may be worth experimenting to see if a single, multi-label model can be trained to classify documents across independent dimensions. This latter case is worth considering if you are limited in terms of time and resources, or you have specific linguistic features in mind that you hypothesize will be informative for one part of the classification task, and not harmful to the rest. Determining the correct approach is the “art” in model training, and requires designing and running experiments to inform the decision-making process.

3.2. Data Collection: Guidelines and Requirements

A statistical classifier makes predictions based on what it learned from training data. Thus the data sources should reflect the real-world data you intend to process.

The training data should have certain characteristics, as described below.

3.2.1. Balance

A balanced dataset avoids creating a biased model. Each category should have roughly the same number of examples. Scrutinize the dataset's available metadata to ensure that the metadata values are evenly represented across different dimensions. For example, if you are drawing on multiple data sources, there shouldn’t be disproportionately more data from one source than another. If the data comes from different time periods, try to get a balanced distribution across time. Depending on the variables and the task, you may need to downsample your data in order to maintain balance, even if it reduces the size of your dataset. Don't hesitate to remove documents from your data-set in order to maintain balance as long as you have enough data. The overall size requirements differs from task to task; see the discussion of size below.

3.2.2. Representativeness

The training dataset must include examples of all the categories your model needs to classify. The data must also match the profile of the input data to be processed. If you are interested in classifying tweets, then you should collect tweets for your corpus (rather than, Wikipedia articles, instruction manuals, or sports commentary transcripts). Similarly, if the goal is classifying news articles by topic, your corpus should consist of news articles (not tweets or novels). Language (English, German), domain (financial, scientific, legal), genre (novel, sports news, tweets), register (formality of speech), and document size (tweets, blogs, magazine articles) are all aspects of the content that you should consider when assessing representativeness.

Remove "noisy documents" from the dataset, including documents that are not relevant for the task or do not contain sufficient lexical content. For example, a document which consists of a URL (and nothing else) is not a useful sample (unless you are analyzing URLs).

Note that you can use Rosette’s language identification functionality to select only documents in the target language. If you are interested in multiple languages, you will need to create a separate corpus for each language. Document classification models trained on one language will not function across other languages.

3.2.3. Size

Generally, the more data you train with, the better your results. With more training data, a model can make more confident predictions. However, be careful not to pad your corpus with non-representative data only to increase the size of the corpus. Balance *and* representativeness are both important. One strategy to assess if you have enough data is to plot a learning curve. Since you cannot plot a learning curve until you get to the Test part of the MATTER cycle, when you are first collecting your dataset, collect more data than you think you need. As a rule of thumb, a minimal eval or development dataset would include hundreds of gold-standard documents per-category. Considering that these may only represent 10-20% of all the data you collect and use, you can’t collect too much data. You can always leave data unused, but it can be difficult to restart a data collection effort later on (or impossible if a data source becomes unavailable).

3.2.4. Intellectual Property

If you are using your data for a personal or academic project, there may be some intellectual property concerns, but there are greater issues when using a dataset for commercial purposes. Investigate who owns the data, and any licensing restrictions that exist on the data (if any). If you own the data, you may consider putting the data in the public domain, or creating an explicit license to allow or prevent others from deriving value from your data. You can find some licenses that are friendly for commercial purposes here. If you have concerns about intellectual property, always get legal advice.

3.3. Preparing Data for Annotation

An ideal corpus includes properly formatted and selected documents whose contents are clean and in plain-text. In this section we list how to prepare your data before starting annotating.

3.3.1. De-duplicating Data

Avoid duplicate documents or near-duplicate documents in your corpus. Duplicate data skews the frequencies of different vocabulary terms. Duplicates or near duplicates can also create "data leakage" if they occur across your training corpus -testing corpus partition. Data in your testing set should never overlap with your training set. There are various tools for finding duplicate documents in a corpus before you partition your dataset. We recommend:

1. Dedup - A Python script for finding duplicate text files.
2. Onion - (ONe Instance ONly) is a de-duplicator for large collections of texts. It can measure the similarity of paragraphs or whole documents and drop duplicate ones based on the threshold you set.

3.3.2. Converting Documents to Unicode

Most plain-text documents are encoded as Unicode UTF-8. If you find data in other encodings, we recommend converting all your data to UTF-8. Here are some recommended tools for detecting and converting character encodings:

1. chardet
2. file
3. iconv

3.3.3. Extracting Plain Text from Markup

If the text in your documents includes markup tags (such as HTML, XML, Markdown, etc.), then you will need to extract the plain-text. Some suggested tools that automatically extract text from marked up documents are:

1. Boilerpipe (see thee web-app wrapper here)
2. Apache Tika

For documents in a particularly cumbersome markup language, consider converting it to a friendlier format using a tool such as Pandoc.

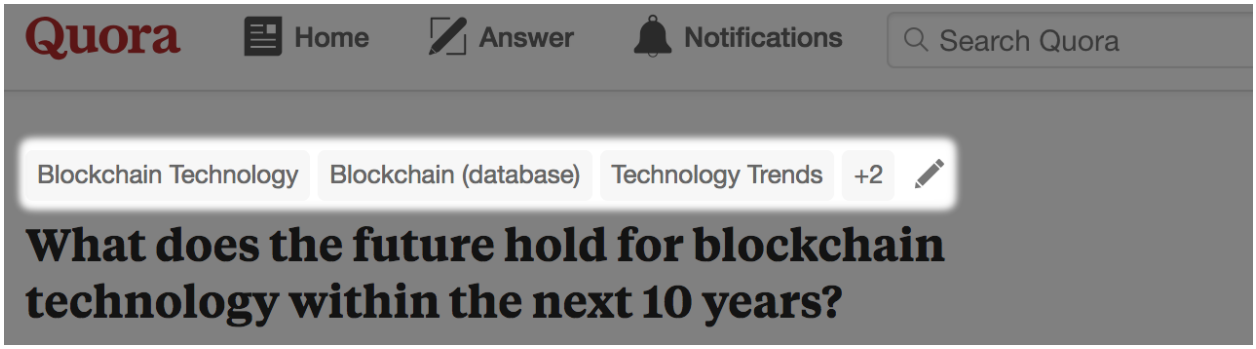
If you need to write your own software to extract text, start by researching what tools your go-to programming language offers; most programming languages will have libraries for parsing markup of various kinds. For example, Python's standard library offers several tools for working with HTML, XML, and other similar markup languages. Some useful non-standard Python tools for working with various markup formats include:

1. Html2text
2. Beautiful Soup

3.3.4. Dangers of not Removing Metadata

Collected documents - especially from structured websites - may include metadata that is not part of the document body (i.e., titles, headers, footers, site map information). At best, metadata is noise, artificially over-representing the frequency of certain terms in your corpus. At worst, the metadata teaches your classifier to classify documents based on the metadata, rather than the document content. For example, if you collect documents from a website where the category label of the document is present in the metadata, then your classifier will learn from this signal since it will be a perfect predictor of the category.

On the website Quora, documents are tagged with categories. If you collected a corpus from Quora, you could use these tags in the metadata as if they were category annotations (which might save you from having to annotate the documents yourself). However, make sure that you don't include these category tags in the document text! For example, here's a Quora page about Blockchain technology that has been tagged:



If every document in your corpus that belongs to the "Blockchain Technology" category has the phrase "Blockchain Technology" in it, then your classifier is simply going to learn that any document with those terms belongs to the "Blockchain Technology" category. This result is called "overfitting" - your classifier ends up fitting exactly to the training data and it won't generalize to real world data. If a document body happens to mention "block chain" (two words), or "cryptographically hashed, distributed transaction ledgers", these terms are relevant to the category "Blockchain Technology". But, if your classifier was trained explicitly on the category tags, and not the document body, these signals would be ignored. If your classifier is ever asked to classify a document that hasn't been pre-tagged with Quora tags, it will not have that signal to rely on, so it will be at a major disadvantage. Additionally, if the documents in your test set have these tags, then the difficulty of classifying those documents will be artificially reduced to the point of triviality—all the classifier will need to do is emit the labels for the tags in the test documents. Your training will not be a simulation of a real-world classification problem (most documents in the real world will not have been annotated with category information like Quora), so your evaluation scores will be artificially high, overestimating the performance of your classifier. All this is to say, make sure that you take care to clean your documents and carefully consider what is actually part of the document body text - all document metadata should be separated in a preprocessing step.

3.4. Annotation

When annotating your data, careful consideration of three items upfront will help you achieve high quality:

1. Annotation tools
2. Annotation guidelines

3. Measuring annotator reliability

The results of this process is a gold-standard corpus, where everything is tagged correctly according to the most recent guidelines. This is the corpus that you will use for machine learning.

3.4.1. Annotation Tools

Remember when selecting an annotation tool that for document categorization, annotators must assign labels (categories) to documents. Many annotation tools are intended for specific NLP tasks that require assigning labels to linguistic units (words, sentences, etc.) within documents rather than to whole documents. Make sure the annotation tool you choose has a suitable interface.

To keep track of a large collection of documents, we recommend assigning each document a Universally Unique Identifier (UUID) – if your documents don't already have an identification scheme. A UUID will give you and your annotators a consistent, convenient way to refer to specific documents during the annotation process.

A popular open source annotation tool like brat is designed for tasks that require assigning labels to individual words, phrases, or other linguistic units within documents. For document categorization, however, it may be simpler to use a spreadsheet application, such as Google Sheets. Whatever tool you decide to use, keep in mind that it will be most convenient for your annotators if they can easily access the full text of the documents and choose a label from your model to assign to each document.

If there are inherent constraints on your category labels, best practice is to enforce those constraints in the annotation tool rather than rely on your annotators. For example, if you are using Google Sheets, set up data validation so that annotators' inputs will be rejected if they fall outside your model's defined set of categories. Any extra cognitive load or room for error that you add to the task has the potential to reduce the quality of the annotations (not to mention, making your annotators unhappy). Consider the size of the corpus, too. Some annotation environments may not be able to support a corpus beyond a certain size.

Finally, when compiling a project schedule, factor in the time needed to set up the annotation environment (software) for the annotators and the time to train them.

Here are some annotation tools for NLP applications:

Software	Use	Language	URL
brat	Manual Annotation	Language-independent	*http://brat.nlplab.org/*
BAT (Brandeis Annotation Tool)	Corpus annotation	Language-independent	*http://timeml.org/site/bat/*
Djangology: A lightweight web-based tool for distributed collaborative text annotation	Distributed collaborative text annotation	Language-independent	*http://sourceforge.net/projects/djangology/*
Ellogon	Manual annotation, machine annotation, project management	Language-independent	*http://www.ellogon.org/*
Knowtator	Manual annotation	Language-independent	*http://knowtator.sourceforge.net/index.shtml*
MAE (Multipurpose Annotation Environment)	Manual annotation	Language-independent	*http://code.google.com/p/mae-annotation/*
SAPIENT: Semantic Annotation of Papers Interface and Enrichment Tool	Manual semantic annotation of scientific concepts	Document Classification Text categorization English	*http://www.aber.ac.uk/en/cs/research/cb/projects/sapienta/software/*

3.4.2. Annotation Guidelines

When drawing up annotation guidelines, it is often helpful to run a pilot annotation effort with a small subset of annotators - who will also work on the broader annotation effort - or with just the author of the

guidelines. The author(s) should do some annotations as a part of developing the guidelines. It's very difficult to guide others through a task you have not performed yourself. The pilot annotation effort will reveal examples or edge cases that haven't been considered, and these will guide the authors in determining how to handle those cases (or to revise the model to account for them).

Annotation guidelines should describe each category and what it is intended to capture. While descriptions are useful, examples are essential. For each category, include positive examples—cases where the label in question applies. Negative examples - where the label in question does not apply - can be equally instructive. For examples that are not straightforward (such as those that annotators disagreed on), be sure to to discuss them and explain the reasoning that led to the final decision.

3.4.3. Measuring Annotator Reliability

We measure annotator reliability because inconsistent annotation or lack of adherence to the annotation guidelines will lead to a less accurate classifier. Especially when starting a new project or on-boarding new human annotators, we check for reliable annotations by having a subset of data annotated in parallel by multiple human annotators.

To measure inter-annotator reliability, we recommend using Krippendorff's alpha, a statistical inter-rater reliability metric. Krippendorff's alpha scores range from -1.0 to 1.0 with 1.0 indicating perfect agreement between annotators. A score of 0.0 indicates agreement no better than random chance (as if your annotators picked their labels randomly out of a hat). A reliability score of 0.80 or greater is generally considered sufficient (though, the higher the better). Lower scores may indicate potential issues with your data, your annotation guidelines, or your annotators' understanding of the task. A low level of inter-annotator agreement will ultimately lead to a less accurate classifier, so we recommend repeatedly measuring the reliability of your annotators until they achieve a satisfactory level of agreement. The cases where annotators disagree are usually good examples to include in your annotation guidelines. It can be useful to have a discussion about points of disagreement with your annotators as a group to reach a consensus.

3.5. Guidelines for Training and Evaluating your Custom Model

The Classification Field Training Kit (FTK) [10] is used to train and evaluate your model. But first you need to set up your human-annotated corpus to enable you to train, experiment, and evalutate the resulting machine-learned models.

3.5.1. Experimental Setup

The standard practice for supervised learning (where a statistical model learns from the gold-standard, human annotated training data) is to partition your annotated data into the following parts:

1. **Training set**: used to train the model
2. Testing corpora
    a. **Development set**: used to test the model and then tune it to increase accuracy
    b. **Evaluation set**: used to score the model on precision, recall, and F-score. This data is never reviewed in order to preserve its impartiality and utility for evaluating models.

The motivation for splitting the data into training and testing sets is that if you evaluate a model on documents that it has seen during training, you are not measuring its true performance on new, unseen data that the model will encounter in the real world. As a quick check, you can evaluate a model against the data it was trained on. In this case, the results should be near perfect, and highly inflated compared to real-world performance.

Ideally, each of the training and testing partitions maintains a balanced distribution of categories. So, assuming your corpus was balanced to begin with, you could maintain balance by dividing the data labeled

with each category into three parts and then putting each third into the training, development, and evaluation set.

For example, imagine a small corpus with 300 documents annotated with three food-related categories:

Category	Number of Documents
DESSERTS-AND-BAKING	100
DINING-OUT	100
FOOD-MOVEMENTS	100

You might partition your corpus as follows:

Category	Total	Training	Development	Evaluation
DESSERTS-AND-BAKING	100	60	20	20
DINING-OUT	100	60	20	20
FOOD-MOVEMENTS	100	60	20	20

The development cycle is as follows:

1. Train a classifier using a training set.
2. Test the classifier against the development set.
3. Experiment with different features, model parameter, or learning strategies to determine what produces the best results.
4. Perform error analysis by inspecting the documents in the development set where your model is making incorrect predictions.
5. If unsatisfied with the performance on the development set, repeat from step #1.
6. If satisfied with the performance on the development set, evaluation your model against the evaluation set and note the scores. You may need to add more data to the training set if results indicate that your training data is insufficient.

The process of trying different strategies to optimize your model (step #3) is called tuning—and specifically, you are tuning your model to perform well on the *development set*. Note, however, that you do *not* want to tune your classifier to the *evaluation set*.

IMPORTANT

Do not look at the documents in the evaluation set once you've begun tuning. Looking at the evaluation set may influence the changes you make to your model that would improve its performance on the evaluation set. Tuning your model to the evaluation set invalidates it because it will end up overestimating your classifier's real-world performance. Once the evaluation data is invalidated, you will need to manually annotate new gold-standard evaluation data (which is expensive and time-consuming). Do not look at the evaluation data to ensure you maintain experimentally valid data to measure of your model's accuracy on unseen documents.

3.6. Troubleshooting Training Data Issues

3.6.1. Assessing Whether You Have Sufficient Training Data

When you are developing your corpus, it is not possible to know, a priori, how much data will be sufficient. Sometimes you will compile a corpus, perform careful manual annotation, and train a system only to discover that your classifier does not perform well even after tuning. One way to assess if your classifier is hampered by a lack of data is to plot a so-called learning curve.

1. Divide your training dataset into parts.
2. Train your model on the first part and plot its performance with respect to size (number of documents) trained on, evaluating against a held out test set.
3. Add the next part of the training set and plot the performance again, evaluating against the same held out test set.

Generally, the trend of this plot should show an increase in performance as more data is added. If this is not the case, it usually indicates a problem with the annotations. If the trend of this plot increases, but levels off after a certain amount of training data has been added, then adding additional data is unlikely to help.

If, however, the trend of the plot continues to increase without leveling off, we recommend adding additional data to your training set until you reach a point where adding more data achieves diminishing returns.

3.6.2. Model Performs Poorly in Certain Categories: Imbalanced Data

If your training set is imbalanced, the categories that have few examples relative to other classes may perform poorly. In this case, there are potentially two remedies:

1. Add more data to the categories until they match the more plentiful categories.
2. Downsample the more plentiful categories to match the category which has the smallest number of training documents.

Option 2 is only recommended if you have ruled out the possibility of having insufficient data.

4. The Classification Field Training Kit (FTK)

Once you've gathered and annotated your data, the next step is to train and test your classification model using the Rosette Classification Field Training Kit (FTK). This section describes how to install the FTK and the command line tools provided by the FTK. To integrate the models into a real application, you must configure Rosette Server [33] to use the new models with the Rosette Server runtime API.

4.1. Installing the FTK

Unzip the FTK zip file:

```
$ unzip classifier-field-training-kit-<version>.zip
```

Install the license:

```
$ cd classifier-field-training-kit-<version>
$ cp rlp-license.xml rbl-je/licenses/rlp-license.xml
```

4.2. Command Line Interface

To build a machine-learned model from labeled training data, you need a directory of training files, arranged by category. The format is:

```
dataset_root/
  category1/
    file1
    file2
    ...
  category2/
    file1
    file2
    ...
  ...
```

The directories under the root directory are the category labels. The files under each category are the training examples. These files should be UTF-8-encoded plain text files. They should contain "clean" data. For best results, the input to the classifier at run time should be cleaned in the same way as the training data.

The command line tools are called `Train`, `TCatCLI`, `MeasurementCLI`, and `Evaluate`.

`Train` is the only command needed to build models and run cross-validation. `TCatCLI` is useful for quick testing of a model, `MeasurementCLI` is useful if you prepare your own train/test splits, and `Evaluate` can measure your model's performance on an annotated, gold-standard data set.

4.2.1. Train

The `Train` command is the only command needed to build models. It can also run cross-validation for a quick and simple way to validate a model once it has been created. `Train` has the following four subcommands:

- `create`: Prepare the data files and create the model directory.
- `append`: Add more training data. The model must be retrained after adding the data.
- `train`: Train the model.
- `xval`: Train the model and run cross-validation from a single dataset, without partitioning data into training and evaluation sets. This function is a quick check that you have your dataset prepared and your model configured correctly, though it will tend to overestimate the performance of the model. This step helps you determine if you have enough correct data to train the model. Does not save the model.

The following parameters are used by the `Train` subcommends

- `lang` is the 3 letter ISO-693-3 language code for the model language.
- `config` is the name of the config file containing the hyperparameters [13] for your model.
- `dataRootDir` is the directory containing the training data files.
- `modelDir` is the name of the trained model directory.
- `n-folds` an integer indicating the number of folds to be used in cross-validation.

```
$ bin/Train
usage: Train [options]... create lang config dataRootDir modelDir
```

```
       Train [options]... append dataRootDir modelDir
       Train [options]... train modelDir
       Train [options]... xval n-folds modelDir
usage: options
    -c,--cost <arg>              one or more (comma separated) cost params
                                 (C); default 0.01 - for train and xval. Only
                                 xval supports multiple cost values.
   --negationWords <arg>         path to negation word list (create only)
   --negativeLexicon <arg>       path to negative lexicon (create only)
   --positiveLexicon <arg>       path to positive lexicon (create only)
   --stop words <arg>             path to stop words list (create only)
   --train                       train model after adding examples
```

4.2.2. TCatCLI for Quickly Testing Models

`TCatCLI` runs the classifier that you trained with the `Train` command. It is a command-line version of the `categories` endpoint, allowing you to run the function using the trained model before integrating into Rosette Server.

```
$ bin/TCatCLI

usage: TCatCLI [options] file [...]
    --adm                              print out results in ADM JSON format
 -ct,--confidenceThreshold <arg>       show confidences above threshold
 -et,--elbowThresholding               use elbow thresholding (invalidates
                                       -st)
    --explanationSet <arg>             show top N positive features
                                       (requires -v)
 -f,--format <arg>                     (line|file|file-list), default=file
    --featureMatrixCategories <arg>    show matrix view of top N categories
 -m,--model <arg>                      model directory
    --maxResults <arg>                 show top N categories, default 1
 -o,--output <arg>                     output file or directory. Output
                                       directory required for -f file-list
 -st,--scoreThreshold <arg>            show scores above threshold
 -v,--verbose                          verbose results
```

The command requires the `-m` (model) and `-o` (output) options, in addition to the input documents. The documents can be a line of text, a file, or a file-list. The default is file. If running with a line of text or file-list, the `-f` (format) option is required. `TCatCLI` should be run on documents not seen at training time.

When providing a file list to `-f`, the file list must contain one input file path per line. Additionally, your output argument (`-o`) must be a directory and not a single file, as the command will create one output file per input file in the list.

Multi-Label Testing

By default, `TCatCLI` returns a single category. The options `maxResults`, `scoreThreshold`, and `confidenceThreshold` allow you to evaluate the multilabel performance of your model along with the default single-label behavior. To return a set number of categories, simply set the `maxResults` option to a value greater than 1. Alternatively, set the threshold for a result's raw score or confidence score (or both) with the `scoreThreshold` and `confidenceThreshold` options, respectively. Note that if the `maxResults` is set alongside either threshold option, it will act as a cap on the number of returned results and not an exact count. In this situation, `TCatCLI` will return up to `<maxResults>` results as long as their score exceeds the specified threshold value.

4.2.3. MeasurementCLI Command: Measurement

`MeasurementCLI` generates the same statistics as `Train xval`, but instead of using cross-validation, it compares two lists of data, actual versus predicted. The first list is the gold-standard, annotated category

each item belongs to, and the second list is the trained model's predicted category for each item. Each list contains one category per line. It is useful if you prepare your own train/test splits, as described in Evaluating on a Train/Test Partition [26].

```
$ bin/MeasurementCLI

 usage: MeasurementCLI [options]... actual predicted
  -m,--mode <arg>    (stats|matrix|list), default=stats, matrix supports up
                     to 26 categories
```

Mode `--mode` arguments:

- **stats**: Displays the per-category statistics [18]
- **matrix**: Displays the confusion matrix [18]
- **list**: Displays the confusion list [19]

4.2.4. Evaluate

If you have a gold set of annotated documents in ADM format, you can score your model's performance using `Evaluate`. By default, this evaluation will calculate per-category and overall (micro and macro) precision (P), recall (F), and F1 scores based on the model's performance in a single-label context. By specifying the `--multilabel` flag, the evaluation can also be performed in a multilabel context. In this case, the same P/R/F metrics will be calculated, as well as two additional metrics, hamming-loss and subset-loss. Note that by default, the multi-label evaluation will use a default raw score threshold of -0.25f. Only categories with a score greater than the default raw score threshold are returned. If you find that this value is suboptimal for your dataset and the not all expected categories are being returned, it can be updated by using the `--score-threshold` option. Additionally, the `--include-categories` and `--exclude-categories` options can be used to include or exclude specific categories from the evaluation.

```
$ bin/Evaluate

usage: Evaluate modelDir dataDir [options]
 -et,--elbow-thresholding        use elbow thresholding (only applicable
                                 in multilabel mode)
 -i,--include-categories <arg>   categories to include in evaluation
 -m,--multilabel                 multilabel evaluation
 -mr,--max-results <arg>         max number of results to return
 -s,--score-threshold <arg>      raw score threshold (only applicable in
                                 multilabel mode)
 -t,--tokenize                   Ignore gold tokens
```

5. Configuring Your Model with Hyperparameters

The `config.yaml` file provided to `bin/Train create` determines the features with which your model will be trained. These hyperparameters are set in the configuration file prior to training, and can be modified in subsequent training runs until they are optimal, as measured against your development dataset. Parameters are values that the model learned from your training data. Hyperparameters instead of being learned from data, are predefined before training begins.

The `config.yaml` file has three main components, `features`, `featureSelectors`, and `tokenFilters`.

- `features` lists the specific features to be used at training time and byy the classifier at run-time. General categorization models will use one or both of the features `TOKEN_UNIGRAM` and `TOKEN_BIGRAM`, although there are a few additional features available to test.
- `featureSelectors` has only one available member, `FREQUENCY_THRESHOLD`, which filters out features that occur less than five times in the training data. we recommend leaving this selector active as it generally reduces a model's training time and memory footprint without sacrificing accuracy.
- `tokenFilters`, controls the tokens that will be ignored from the input before feature generation. There are two options here: `STOP_WORD`, filters based on a specified list of stop words; `ASCII_NUMERIC`, filters out numeric tokens.

5.1. Stop Words

Stop words are words that are blacklisted by the classifier so that they are ignored at training time and at prediction time. In English, it is often useful to ignore certain high frequency, but uninformative words such as "a", "an", "the, "but", "or, "of", "to", "if", etc. Certain classes of words, including determiners, prepositions, particles, and conjunctions are generally not informative for document classification, so they are good candidates for stop words. These are good stop words, intuitively, because there is no reason to believe that the presence or absence of these words in a document will tell you much, if anything, about the content of the document. In fact, most English language documents of any significant length are likely to include these words. You can use the default list of stop words provided by the FTK (`classifier-field-training-kit-<version>/etc/stopwords-default.txt`), or create your own. Stop words lists are one example of a hyperparameter because they are predefined, before training your classifier, and the effects of varying stop words lists on classifier accuracy can be measured at evaluation time.

5.2. Feature Selection

Feature selection is the process of selecting functions that will extract information from the data that will be informative for classification. An example of a feature in the sample configuration is the `TERM_UNIGRAM` feature. This feature simply takes the set of words that occur in a document, individually. In NLP, this is commonly referred to as a "bag-of-words" the sequence in which the words occur in the document is ignored. While sometimes a "bag-of-words" is sufficient to help a classifier perform well, sometimes more complex features are required.

The `TERM_BIGRAM` feature take pairs of words in sequence from the document, but ignores the overall structure of the document. Unigrams, bigrams and so on are instances of the generalized term "n-gram" which is a subsequence of n words extracted from a longer sequence. For example, for the following sentence, the table lists the unigrams and bigrams:

Budgerigars are popular pets around the world due to their small size, low cost and ability to mimic human speech.

n-gram	sequence
unigram	`['Budgerigars', 'are', 'popular', 'pets', 'around', 'the', 'world', 'due', 'to', 'their', 'small', 'size', ',', 'low', 'cost', 'and', 'ability', 'to', 'mimic', 'human', 'speech', '.']`
bigram	`['Budgerigars are', 'are popular', 'popular pets', 'pets around', 'around the', 'the world', 'world due', 'due to', 'to their', 'their small', 'small size', 'size ,', ', low', 'low cost', 'cost and', 'and ability', 'ability to', 'to mimic', 'mimic human', 'human speech', 'speech .']`

Selecting appropriate features for your task involves some combination of linguistic intuition and experimentation. For this reason, feature selection is sometimes described as an art.

5.3. Available Features

- TOKEN_UNIGRAM - each individual token is used as features.
- TOKEN_BIGRAM - all token pairs are used as features.
- TF_IDF_TOKEN_UNIGRAM - all tokens are used as features with weights representing their TF-IDF (term frequency-inverse document frequecy) score. IDF scores are based off of the training corpus.
- NEGATION_BIGRAM - all token pairs where the first token is a "negation word" are used as features. Custom negation words can be provided as an argument to `bin/Train create`. This feature is for English only, and negation words should NOT also be included in the stop words list.

6. Building a Model from Start to Finish: A Trivial Example

In this example we'll train an English model to distinguish different types of pets: birds, cats, and dogs.

6.1. Prepare Your Dataset

The first step is to make sure you prepare your data. The FTK expects your corpus to contain documents as plain text, UTF-8 encoded files. Each file should be located in a directory where the name of the directory is the category label for all documents within that directory. We will consider a corpus organized in this way to be a “dataset”.

6.1.1. Dataset Example

Consider the following pets dataset example with three categories: bird, cat, and dog. This is a toy dataset of Wikipedia documents for different animal breeds. It is too small for a real dataset, but is used here for demonstration purposes. You can find it in the distribution at `classifier-field-training-kit-<version>/samples/data`.

TIP

To help view your corpus from the command-line, we recommend the tree program, which can recursively list directories and summarize their contents.

```
$ tree -d .
.
└── pets
    ├── bird
    ├── cat
    └── dog

4 directories
# sort documents by frequency per category
$ find pets -name "*.txt" | cut -f2 -d'/' | sort | uniq -c | sort -nr
  10 bird
  10 cat
  10 dog
```

6.2. Prepare the Model

NOTE
The example code is written assuming you are in the `classifier-field-training-kit-<version>` directory.

To prepare a model, use the `Train` command with the `create` option. Supply a language code, model config file, your dataset directory, and the desired output directory. This command prepares the data files and creates a directory for the model. In the example below, we provide the following arguments to `Train create`:

1. `eng`: the three-letter language ISO code for English (because the dataset documents are in English)
2. `etc/config.sample`: location of sample config (you can use the sample config provided with the FTK and modify it to suit your needs later)
3. `samples/data/pets`: the dataset directory
4. `pets.model`: a directory where the model resources will be created (it's OK if this directory doesn't exist as it will be created for you)

NOTE
As a convention, we use the `.model` suffix for all directories containing a classification model. Although the suffix is not required, it helps to keep track of your models and distinguish them from your datasets.

```
$ bin/Train create eng etc/config.sample samples/data/pets pets.model
0     [main] INFO Stopwords - Loaded 0 stop words
629   [main] INFO MutableLexicon - allowLookupByOrdinal: true
643   [main] INFO DatasetWalker - processing: bird
1300  [main] INFO DatasetWalker - processing: cat
1396  [main] INFO DatasetWalker - processing: dog
$ ls pets.model
categories   config   examples    lexicon
```

6.3. Cross-Validation: Quick Check Your Training Data

Cross-validation is a quick and simple way to validate your training data once the model directory has been created, even if you have a small amount of data. The model created to perform cross-validation is not saved.

TIP

Cross-validation can be used as a stop-gap when you have a small amount of data because it artificially inflates the size of your dataset by reusing documents for training within different "folds". Cross-validation can lead to an overestimation of real-world performance of a classifier. We recommend using cross-validation as a quick check to make sure that you have your dataset prepared and your model configured correctly. When you are ready to begin tuning a classifier, you should create a well-defined training, development, and evaluation partition of your dataset rather than rely on cross-validation.

This method doesn't require partitioning your data into a training subset and held out evaluation subset. Instead, this method partitions the dataset into subsets for training and evaluation, automatically and multiple times. Separate classifiers are trained and evaluated for each partition (each partition is called a "fold", hence the term "N-fold cross-validation"). The dataset is divided in such a way that the evaluation subset does not overlap with any other—the evaluation "folds", as such, are mutually exclusive and exhaustive. The training subsets, however, will overlap with one another. The evaluation metrics for each "fold" are ultimately averaged together to give an overall metric.

```
# run "10-fold" cross-validation
bin/Train xval 10 pets.model

0    [main] INFO Stopwords - Loaded 0 stop words
630  [main] INFO MutableLexicon - allowLookupByOrdinal: true
632  [main] INFO Dataset - reading examples...
632  [main] INFO Dataset - skipping feature selection
650  [main] INFO Dataset - starting cross-validation...

optimization finished, #iter = 4
Objective value = -0.0823930
nSV = 27
... # many optimization lines omitted
optimization finished, #iter = 4
Objective value = -0.0981776
nSV = 27
682 [main] INFO Dataset - elapsed for cross-validation: 32 ms
COST: 0.01
Per-category statistics:
                                 size     tp     fn     fp         P         R        F1
bird                               10     10      0      0     1.000     1.000     1.000
cat                                10      9      1      0     1.000     0.900     0.947
dog                                10     10      0      1     0.909     1.000     0.952

Total counts:                      30     29      1      1         -         -         -
Macro Average:                      -      -      -      -     0.970     0.967     0.968
Micro Average:                      -      -      -      -     0.967     0.967     0.967

Confusion matrix:
      a       b       c <-- Predicted as
     10       0       0 |         10 a=bird
      0       9       1 |         10 b=cat
      0       0      10 |         10 c=dog

Confusion list:
actual                           predicted              errors
cat                              dog                         1
```

6.4. Interpreting Results

You'll notice `Train xval` prints out several statistics. These are the same statistics printed by `MeasurementCLI`. This section reviews these statistics and discusses how to interpret them. These statistics allow you to objectively measure the accuracy of your custom classifiers and compare different configuration parameters to see which changes improved your classifier and which ones didn't.

6.4.1. Per-Category Statistics

The per-category statistics table displays the accuracy of your classifier for each category as well as averaged accuracy across all categories. The table columns are as follows:

1. **size**: This is the number of gold instances (cumulative over all "folds" in the case of cross-validation) for each category in the test subset.
2. **tp**: This is the number of true positives for each category. (Higher is more accurate.)
3. **fn**: This is the number of false negatives for each category. (Lower is more accurate.)
4. **fp**: This is the number of false positives for each category. (Lower is more accurate.)
5. **P**: This is the precision score for each category (an accuracy metric ranging from 0.0 to 1.0 that is sensitive to false positives—higher is more precise).
6. **R**: This is the recall score for each category (an accuracy metric ranging from 0.0 to 1.0 that is sensitive to false negatives—higher is better recall).
7. **F1**: This is the F1 score for each category (the harmonic mean of precision and recall, ranging from 0.0 to 1.0, thus sensitive to both false positives and false negatives—higher is better accuracy).

The statistics are displayed for each category individually (per row) and there are additional average metrics at the foot of the table. These averages are explained as follows:

1. **Macro Average:** The macro average for precision and recall are the means of the per-category precision and recall scores. The macro F1 is the harmonic mean of the macro precision and macro recall scores.
2. **Micro Average**: This score accounts for the total false positives and false negatives. The micro scores are sensitive to imbalanced datasets—if one category is overly represented (i.e., there are more gold instances of this category in the evaluation subset relative to the others), it will have a bigger impact on the micro averaged precision, recall, and F scores.

```
Per-category statistics:
                               size     tp     fn     fp         P         R        F1
bird                             10     10      0      0     1.000     1.000     1.000
cat                              10      9      1      0     1.000     0.900     0.947
dog                              10     10      0      1     0.909     1.000     0.952

Total counts:                    30     29      1      1         -         -         -
Macro Average:                    -      -      -      -     0.970     0.967     0.968
Micro Average:                    -      -      -      -     0.967     0.967     0.967
```

6.4.2. Confusion Matrix

A confusion matrix is a representation of the true positives, false positives, and false negatives for a given evaluation run. In this case, the column headers of the matrix represent the category label that the classifier predicted, and the row labels (shown as the rightmost column) represent the gold standard category label that was annotated. The cells of the matrix contain the frequency of error types.

- True positives are the cells along the diagonal, where the row and column category labels match (shown as from upper left to lower right), that is, cases where the classifier prediction matched the gold label.
- False positives are the cells along each column that are off the diagonal for that row. For example, the "1" in the middle cell of the "c" column, represents a false positive for category "c", as it was really a category "b" item.
- False negatives are the cells along each row that are off the diagonal. For example, the "1" in the middle of the "c" column is in the row for "gold standard category b" and thus represents a false negative for category "b", as it was predicted as a category "c" item.

If all predictions were correct, all values would be 0 except on the diagonal.

In the following example, there are 10 total examples for cat. Nine were correctly identified, while One (column 3. row 2) was mis-predicted as dog ("c").

```
Confusion matrix:
      a       b       c <-- Predicted as
     10       0       0 |         10 a=bird
      0       9       1 |         10 b=cat
      0       0      10 |         10 c=dog
```

6.4.3. Confusion List

A confusion list is a representation of the false positives and false negatives for a given evaluation run. It contains essentially the same information as the confusion matrix, however, the true positives are omitted (these are the values that occur along the diagonal in the confusion matrix). This view of the statistics can help to suss out what are the most common error types and which categories the classifier is confusing most often. It is also useful when you have so many categories that a confusion matrix becomes unwieldy (e.g., more than 10 categories results in a larger than 10 by 10 matrix which can make it difficult to read).

```
Confusion list:
actual                          predicted               errors
cat                             dog                          1
```

6.5. Train the Model

Now that the training data has been validated, we can train the model. Use the `Train train` command to train the actual model on the training data subset. This model is saved in the model directory.

```
bin/Train train pets.model
```

To test the model, use the `bin/Train TCatCLI` command on multiple files from a different dataset.

7. A Real Example

The previous example was on a tiny dataset. In this section we'll use the command line tools to build an English categorizer for the 20 Newsgroups dataset, which is a popular dataset in research circles.

The 20 Newsgroups data is already arranged in the correct format. There are a few files that are not strictly UTF-8, but that can be ignored for our purposes. The version of the data we show below has all newsgroup headers stripped except "From" and "Subject".

```
$ wget -q http://qwone.com/~jason/20Newsgroups/20news-18828.tar.gz -O - | tar xzf -
```

```
$ ls -1F 20news-18828
alt.atheism/
comp.graphics/
comp.os.ms-windows.misc/
comp.sys.ibm.pc.hardware/
comp.sys.mac.hardware/
comp.windows.x/
misc.forsale/
rec.autos/
rec.motorcycles/
rec.sport.baseball/
rec.sport.hockey/
sci.crypt/
sci.electronics/
sci.med/
sci.space/
soc.religion.christian/
talk.politics.guns/
talk.politics.mideast/
talk.politics.misc/
talk.religion.misc/
```

First we create the dataset and generate the examples. The `Train` command below specifies stop words [14] (words that are to be ignored by the classifier) and uses a non-default config file (`config.term_unigram`).

```
$ rm -rf news.model
$ time bin/Train create eng etc/config.term_unigram 20news-18828 news.model \
-stopwords etc/stopwords-default.txt
real    0m33.242s
user    0m35.516s
sys     0m0.920s
```

Next, we run 10-fold cross-validation:

```
$ time bin/Train xval 10 news.model
```

The `Train xval` outputs the per-category statistics:

```
Per-category statistics:
                        size      tp      fn      fp         P         R        F1
alt.atheism              799     722      77      53     0.932     0.904     0.917
comp.graphics            973     838     135     189     0.816     0.861     0.838
comp.os.ms-windows.misc  985     853     132     143     0.856     0.866     0.861
comp.sys.ibm.pc.hardware 982     784     198     177     0.816     0.798     0.807
comp.sys.mac.hardware    961     858     103     100     0.896     0.893     0.894
comp.windows.x           980     887      93     101     0.898     0.905     0.901
misc.forsale             972     900      72     175     0.837     0.926     0.879
rec.autos                990     907      83      76     0.923     0.916     0.919
rec.motorcycles          994     938      56      24     0.975     0.944     0.959
rec.sport.baseball       994     960      34      44     0.956     0.966     0.961
rec.sport.hockey         999     978      21      19     0.981     0.979     0.980
sci.crypt                991     944      47      25     0.974     0.953     0.963
sci.electronics          981     853     128     117     0.879     0.870     0.874
sci.med                  990     935      55      66     0.934     0.944     0.939
sci.space                987     953      34      36     0.964     0.966     0.965
soc.religion.christian   997     937      60      67     0.933     0.940     0.937
talk.politics.guns       910     864      46      61     0.934     0.949     0.942
talk.politics.mideast    940     922      18      32     0.966     0.981     0.974
```

```
talk.politics.misc        775     685      90      35     0.951     0.884     0.916
talk.religion.misc        628     511     117      59     0.896     0.814     0.853

Total counts:           18828   17223    1605    1605         -         -         -
Macro Average:              -       -       -       -     0.915     0.912     0.914
Micro Average:              -       -       -       -     0.915     0.915     0.915
```

And the confusion matrix:

```
Confusion matrix:
  a     b     c     d     e     f     g     h     i     j     k     l     m     n     o     p     q     r     s     t <-- Predicted as
722     4     0     0     1     0     1     2     0     0     0     1     1     8     1    15     2     2     3    36 |  799 a=alt.atheism
  1   838    32    25    11    34    15     0     1     2     0     1     5     2     2     0     1     2     0     1 |  973 b=comp.graphics
  0    32   853    51     9    22     8     0     0     2     0     0     4     1     2     0     1     0     0     0 |  985 c=comp.os.ms-windows.misc
  1    33    60   784    33    10    27     4     1     0     0     1    22     6     0     0     0     0     0     0 |  982 d=comp.sys.ibm.pc.hardwar
  0    13    12    26   858     5    25     1     1     0     1     0    15     3     0     1     0     0     0     0 |  961 e=comp.sys.mac.hardware
  0    35    24     8     4   887     5     0     0     3     1     0     6     1     3     1     1     0     0     1 |  980 f=comp.windows.x
  0     5     4    16     8     0   900    13     0     1     1     3    12     3     1     2     2     0     0     1 |  972 g=misc.forsale
  0     5     1     9     3     4    24   907    10     3     0     0    17     2     3     0     1     0     1     0 |  990 h=rec.autos
  0     2     0     2     2     2    18    18   938     2     1     1     2     2     2     0     2     0     0     0 |  994 i=rec.motorcycles
  0     2     0     1     1     3     3     3     1   960    11     0     2     2     0     2     1     1     1     0 |  994 j=rec.sport.baseball
  0     0     0     2     2     0     2     2     0     9   978     0     2     0     1     0     0     0     1     0 |  999 k=rec.sport.hockey
  2    13     1     3     3     5     3     0     1     2     0   944     7     2     1     1     2     0     1     0 |  991 l=sci.crypt
  0    12     4    27    15     5    26    14     1     4     2     4   853     5     5     1     2     0     1     0 |  981 m=sci.electronics
  1     7     2     3     4     2     5     5     2     2     0     3    10   935     3     1     3     0     2     0 |  990 n=sci.med
  1     9     0     1     1     2     4     0     1     0     0     0     4     4   953     3     2     0     1     1 |  987 o=sci.space
  3     9     0     1     2     1     3     2     1     4     0     0     1     9     3   937     3     7     2     9 |  997 p=soc.religion.christian
  1     1     1     1     0     3     3     0     2     5     0     6     0     1     0     0   864     1    16     5 |  910 q=talk.politics.guns
  1     2     0     0     1     1     1     2     0     2     0     2     1     1     0     0     0   922     2     2 |  940 r=talk.politics.mideast
  6     2     1     0     0     1     0     6     1     1     2     3     2     8     6     4    31    13   685     3 |  775 s=talk.politics.misc
 36     3     1     1     0     1     2     4     1     2     0     0     4     6     3    36     7     6     4   511 |  628 t=talk.religion.misc
```

And the confusion list:

```
Confusion list:
actual                          predicted                       errors
comp.sys.ibm.pc.hardware        comp.os.ms-windows.misc             60
comp.os.ms-windows.misc         comp.sys.ibm.pc.hardware            51
alt.atheism                     talk.religion.misc                  36
...
```

```
real    0m50.709s
user    0m54.128s
sys     0m0.761s
```

The cross-validation results show the per-category precision, recall, and F1, as well as a confusion matrix and a confusion list. Each row header in the matrix represents the actual results. The column headers represent the predicted results. If all predictions were correct, all values would be 0 except on the diagonal.

The last row, t, at column u, shows there were 628 total examples with actual category talk.religion.misc. 36 of these (column a) were mis-predicted as alt.atheism. 36 (column p) were mis-predicted as soc.religion.christian. 511 (column t) were predicted correctly as talk.religion.misc.

The confusion list shows the actual/predicted errors sorted by error count in descending order. The third worst offender is the confusion between alt.atheism and talk.religion.misc. The list form can be easier to read when the matrix becomes too large. Indeed, if there are more than 26 categories, the matrix will not be printed, just the list.

Now that we've validated the training data, we can train our model:

```
$ bin/Train train news.model
```

Finally, we can use our model on new documents we did not see at training time:

```
$ bin/TCatCLI -m news.model -o /dev/stdout \
<(echo "The Red Sox won at Fenway yesterday.") 2>/dev/null
rec.sport.baseball    0.501416    0.084238
```

```
$ bin/TCatCLI -m news.model -o /dev/stdout \
<(echo "The Bruins won at the garden") 2>/dev/null
rec.sport.hockey    0.300458    0.068404
```

8. Evaluating Your Classifier

Once you've trained and improved your classifier, you should evaluate it on a completely new set of data, the evaluation set.

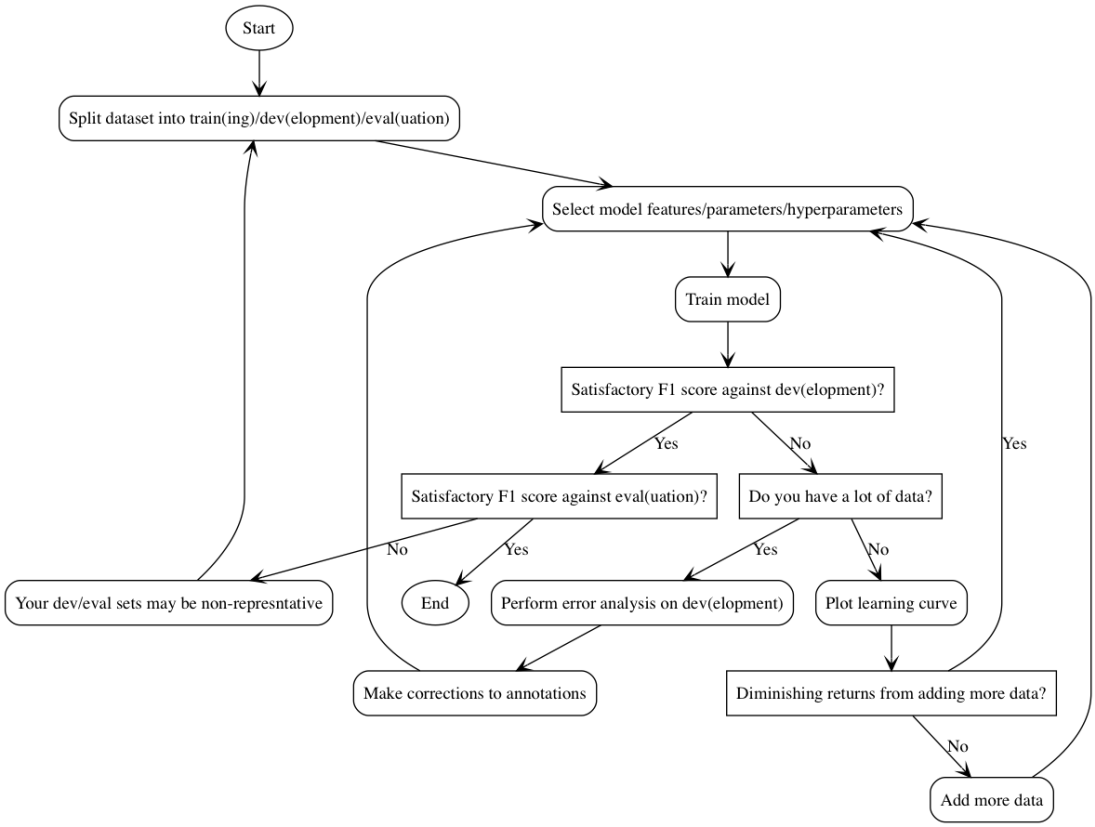
8.1. Iterating on a Model

Recall the MATTER cycle[2]:

1. **M**odel: Determine the set of categories of interest.
2. **A**nnotate: Manually assign category labels to a set of documents.
3. **T**rain: Train a classifier using a training set of the manually labeled documents.
4. **T**est: Test the classifier on a held-out development set of the manually labeled documents that does not overlap with the training set and perform error analysis.
5. **E**valuate: You can evaluate the classifier on yet another held-out subset of your manually labeled documents. (To preserve the experimental utility of your evaluation set, do ***not*** perform error analysis on your evaluation set).
6. **R**evise: Revise your model or consider augmenting your corpus by annotating additional documents.

The following flow diagram shows a more detailed process for iterating through the TTER part of the cycle. This part of the cycle, at a more granular level, involves configuring, training, evaluating, and tuning a classifier to iteratively improve its accuracy. The steps of selecting features and hyperparameters [13] is the heart of improving a classifier.

[2]Pustejovsky, J., & Stubbs, A. (2012), *Natural Language Annotation for Machine Learning: A guide to corpus-building for applications.* O'Reilly Media, Inc.



8.2. Partition Your Dataset

When you have run cross-validation as a quick check, you are ready to set up experiments to evaluate your classifier in a more controlled manner. To do so, first partition your dataset into subsets:

1. Training subset: Your classifier will learn from this subset.

2. Development subset: This subset was held out from training by you. Your classifier will never be trained on this dataset—only evaluated against it. During the tuning process, you can inspect this subset and make decisions about how to improve your classifier by evaluating against this subset.

3. Evaluation subset: This subset (sometimes called a "hold-out" or "validation" set) will never be inspected by you nor your classifier. It is imperative that this dataset is left uninspected in order to experimentally validate that tuning your classifier against the development subset has resulted in a classifier that can perform at an acceptable level of accuracy on new, unseen data. This dataset, thus, is simulating new, unseen data. If you accidentally inspect this subset, or accidentally train your classifier on this subset, your experiments that use that information will become invalid—you will no longer be estimating your classifier's ability to generalize.

NOTE
When you split your dataset into training/development/evaluation subsets, try to ensure that each subset maintains the desired properties of balance and representativeness. Each category should have approximately equal numbers of examples in each subset. Additionally, if there are metadata in your corpus that represent different dimensions of your data, it is ideal to maintain the distributions of those attributes within each subset. That is to say, if you collected your data from different sources, from different time periods, etc., each subset should represent a similar distribution of sources, time periods, etc. If these properties can be maintained, it will help mitigate skewing your classifier.

Consider this partitioned version of our toy pets dataset as an example.

```
$ tree pets
pets
├── dev
│   ├── bird
│   │   ├── Lories_and_lorikeets.txt
│   │   └── Palm_cockatoo.txt
│   ├── cat
│   │   ├── Siamese_cat.txt
│   │   └── York_Chocolate.txt
│   └── dog
│       ├── Irish_Terrier.txt
│       └── Weimaraner.txt
├── eval
│   ├── bird
│   │   ├── Cockatiel.txt
│   │   └── Lovebird.txt
│   ├── cat
│   │   ├── Persian_cat.txt
│   │   └── Sphynx_cat.txt
│   └── dog
│       ├── Chihuahua_(dog).txt
│       └── German_Shepherd.txt
└── train
    ├── bird
    │   ├── Budgerigar.txt
    │   ├── Cockatoo.txt
    │   ├── Grey_parrot.txt
    │   ├── Kakapo.txt
    │   ├── New_Zealand_parrot.txt
    │   └── Psittacoidea.txt
    ├── cat
    │   ├── Arabian_Mau.txt
    │   ├── British_Longhair.txt
    │   ├── California_Spangled.txt
    │   ├── Maine_Coon.txt
    │   ├── Snowshoe_cat.txt
    │   └── Turkish_Angora.txt
    └── dog
        ├── Akita_(dog).txt
        ├── Dachshund.txt
        ├── Poodle.txt
        ├── Rottweiler.txt
        ├── Samoyed_dog.txt
        └── Yorkshire_Terrier.txt

12 directories, 30 files
$ find pets -name "*.txt" | cut -f2 -d'/' | sort | uniq -c | sort -nr
  18 train
   6 eval
   6 dev
$ find pets -name "*.txt" | cut -f2,3 -d'/' | sort | uniq -c | sort -nr
   6 train/dog
   6 train/cat
   6 train/bird
   2 eval/dog
   2 eval/cat
   2 eval/bird
   2 dev/dog
   2 dev/cat
   2 dev/bird
```

In this example, we've done a 60%:20%:20%::train:dev:eval partition. It's ideal to give your classifier as much training data as you can while maintaining sufficiently representative dev and eval subsets. The ratios of your partition will be somewhat dependent on how much data you have in total. An 80%:10%:10% partition, or

even a 90%:5%:5% could potentially work, presuming 5% of your total documents are still enough to provide a balanced, representative evaluation subset.

IMPORTANT

It is important to stress that the pets dataset above is a toy dataset. Having only a couple instances per-category in the eval and dev subsets is far too small to ensure that these subsets are representative. At minimum, in the real world, a dataset would include hundreds of gold-standard documents per-category.

8.3. Evaluating on a Train/Test Partition

Once you've created a partition of your dataset, you can run an evaluation against a specific, held-out subset as follows:

1. Train a classifier using your training subset
2. Write out the gold labels for the test subset to a file (one document label per line)
3. Write out the predicted labels for the test subset to a file (one document label per line)

The files generated in steps 2 and 3 are used to evaluate the classifier accuracy using the `MeasurementCLI` program. The following example demonstrates training a classifier on `pets/train` and evaluating against `pets/dev`.

NOTE

We make use of the jq program below to query into the Annotated Data Model JSON responses.

8.4. Train the Classifier

1. Create a model directory with model resources, using the training subset (`pets/train`)

NOTE

This example assumes you created a partitioned pets dataset in `classifier-field-training-kit-<version>`.

```
$ bin/Train create eng etc/config.sample pets/train pets.model

0    [main] INFO Stopwords - Loaded 0 stop words
916  [main] INFO MutableLexicon - allowLookupByOrdinal: true
933  [main] INFO DatasetWalker - processing: bird
1642 [main] INFO DatasetWalker - processing: cat
1719 [main] INFO DatasetWalker - processing: dog
```

2. Train the classifier using the model resources

```
$ bin/Train train pets.model

0 [main] INFO Stopwords - Loaded 0 stop words
876  [main] INFO MutableLexicon - allowLookupByOrdinal: true
880  [main] INFO Dataset - reading examples...
880  [main] INFO Dataset - skipping feature selection
895  [main] INFO Dataset - training model...

optimization finished, #iter = 4
Objective value = -0.0575345
nSV = 18

optimization finished, #iter = 4
Objective value = -0.0635544
nSV = 18

optimization finished, #iter = 5
Objective value = -0.0677941
nSV = 18
905  [main] INFO Dataset - elapsed for training model: 9 ms
905  [main] INFO Dataset - writing model header...
907  [main] INFO Dataset - writing model weights...
917  [main] INFO Dataset - elapsed for writing model.weights: 9 ms
```

8.5. Evaluate the Classifier

In this section we evaluate the classifier accuracy by comparing gold labels, the "correct" labels, with the predicted labels generated by the classifier. The generated files are compared using the `MeasurementCLI` program.

Here we use the data in `pets/dev` to evaluate the classifier that we just trained on `pets/train`.

1. Create a list of the files in the dev subset.

```
$ find pets/dev -name "*.txt" > dev-filenames.txt
```

2. Create a directory to hold the results.

```
$ mkdir dev-results
```

3. Run `TCatCLI` on the file list in the dev subset (`dev-filenames.txt`) using the `pets.model` model, and putting the results in `dev-results`. These are the predicted results.

```
bin/TCatCLI --format 'file-list' --adm -m pets.model -o dev-results dev-filenames.txt

0    [main] INFO Stopwords - Loaded 0 stop words 8 [main] INFO MutableLexicon - allowLookupByOrdinal: true
8    [main] INFO CategorizerModel - time loading example/pets.model/lexicon_filtered: 6 ms
9    [main] INFO CategorizerModel - confidenceStrategy: SIMPLE
19   [main] INFO CategorizerModel - time loading example/pets.model/model.weights: 9 ms
```

4. Inspect the prediction for a sample result file. The program `jq` helps us query the response that is in `ADM JSON` format.

```
$ jq .attributes.categorizerResults dev-results/Irish_Terrier.txt
{
  "type": "list",
  "itemType": "categorizerResults",
  "items": [
    {
      "label": "dog",
      "score": -0.01803384,
      "confidence": 0.40695784
    }
  ]
}
```

5. Create empty files where gold labels and predictions will be recorded

```
$ touch dev-gold
$ touch dev-predicted
```

6. Append the gold labels and the predicted labels. The gold labels come from the directory name; the predicted labels are from the results generated by `TCatTCLI` and put in `dev-results`.

```
$ find pets/dev -name "*.txt" | while read f;\
  do basename $(dirname "$f") >> dev-gold;\
  jq -r .attributes.categorizerResults.items[].label\
  dev-results/"$(basename $f)" >> dev-predicted; done
```

7. Look at the gold and predicted labels side by side

```
$ paste dev-filenames.txt  dev-gold dev-predicted

pets/dev/cat/York_Chocolate.txt   cat      cat
pets/dev/dog/Weimaraner.txt       dog      dog
pets/dev/dog/Irish_Terrier.txt    dog      dog
pets/dev/bird/Lories_and_lorikeets.txt     bird     bird
pets/dev/bird/Palm_cockatoo.txt   bird     bird
```

8. Evaluate the classifier using `MeasurementCLI`. `MeasurementCLI` generates the same statistics as `Train eval`, but compares the values in the gold data with the predicted data.

```
$ bin/MeasurementCLI -m stats dev-gold dev-predicted

                    size   tp  fn  fp      P       R       F1
   bird                2    2   0   0  1.000   1.000    1.000
   cat                 2    2   0   0  1.000   1.000    1.000
   dog                 2    2   0   0  1.000   1.000    1.000
   Total counts:       6    6   0   0      -       -        -
   Macro Average:      -    -   -   -  1.000   1.000    1.000
   Micro Average:      -    -   -   -  1.000   1.000    1.000
```

9. Show confusion matrix

```
$ bin/MeasurementCLI -m matrix dev-gold dev-predicted

      a       b       c <-- Predicted as
      2       0       0 |          2 a=bird
      0       2       0 |          2 b=cat
      0       0       2 |          2 c=dog
```

Because this is a toy dataset, with a small number of categories that are relatively easy to classify, this classifier makes perfect predictions on all six documents in the dev subset.

If we were using real datasets, we would refine our model, perhaps adding more training data or modifying the hyperparameters [13] by editing the config file. Once the model is trained to our satisfaction, we would repeat the steps above with the eval set.

9. Training a Sentiment Model

Sentiment can be thought of as a specific case of text classification, where the categories represent sentiment labels (`pos`, `neu`, or `neg` for our default models). As a result, this FTK can be used to train a sentiment model in addition to a more general categorizer classification model. The training process is identical to the one detailed above for building a categorizer, with one additional feature available - `SENTIMENT_LEXICON`. This feature will, by default, rely on our stock English positive and negative lexicon files, although custom lists can be provided via the `--negativeLexicon` and `--positiveLexicon` options to `bin/Train create`.

9.1. What is a Sentiment Lexicon?

A sentiment lexicon, in its most simplistic form, is a list of words that have been compiled with the express purpose that all words in the list are exemplars for indicating a particular sentiment. While a purely lexical approach to sentiment analysis assumes that the presence of certain words or phrases in a document are enough to determine the sentiment, we use sentiment lexicons as features in combination with other linguistic characteristics in a statistical model. Using lexicons as features is a common method of augmenting sentiment models, but such lexicons must be developed with consideration to the nuances of natural language, especially lexical ambiguity.

In an ideal world, if you had an extensive training corpus that covered the entire vocabulary of the domain in question, defining explicit sentiment lexicons would not be necessary. All of the statistics about how lexical items contribute to sentiment could be learned from the training corpus. In the real world, however, the available resources to create labeled training data are finite, so when the training data does not cover the lexicon sufficiently, there may be terms that the model will encounter that are very low frequency or out-of-vocabulary (OOV), i.e., they don't occur at all in the training corpus. In such cases, a sentiment lexicon may help to make up for a smaller training corpus. Even if a lexical item in a document was not represented in the training corpus, if it exists in a sentiment lexicon, the model may still have a chance to classify the sentiment correctly.

Let's take a look at some samples from Basis Technology's English sentiment lexicons. We have two sentiment lexicons for English, one for the pos (positive) sentiment label, and one for the neg (negative) sentiment label. We've sampled some lexical items from each lexicon:

neg	pos
allegation	ample
bullshit	considerate
cataclysm	convienient
costly	eloquently
darkness	enthral
embroilment	excitement
phobia	graciously

neg	pos
sacrificed	impressed
tense	influential
unorthodox	wise

9.1.1. Misspellings and Variation

Note the misspelling of "convenient" as "convienient". Misspellings are intentionally included as they may be sparser than correct spellings, so a finite training corpus may lack examples of the misspellings ("convenient" with the correct spelling is also included in the lexicon). Also note the British spelling variation of "enthrall" as "enthral" ("enthrall" with two "l"s is also included in the lexicon). These lexicons were developed for the purpose of analyzing sentiment in informal, social media communications, specifically tweets. As such, common misspellings or variations are helpful to capture along with correct spellings. If misspellings are excluded from your lexicons, their signals may be lost to the model, and may reduce the model's ability to learn.

9.1.2. Pitfalls

When developing sentiment lexicon resources, it is important to consider an inherent property of natural language: lexical ambiguity. Many words are polysemous (have multiple meanings), and it is through context, or world knowledge, that humans interpret the intended meaning. As such, it is important to avoid, as much as possible, words that are relatively ambiguous across different sentiment categories.

Ambiguity

Let's consider some purely lexical examples:

A. The movie's plot was **unpredictable**.

B. The network's performance was **unpredictable**.

In A., unpredictability in the plot is probably intended as praise, indicating a positive sentiment. In contrast, in B., unpredictability in a network is probably an undesirable quality, indicating a negative sentiment. One might do a corpus survey and determine which sense of unpredictable is more common. If you found that "unpredictable" significantly and more commonly indicates a negative sentiment, then you might add it to your negative sentiment lexicon. If it, more frequently, contributes to a significantly positive sentiment, then add it to your positive sentiment lexicon. If it is so ambiguous that it doesn't clearly lean towards one or the other, it would not be a good candidate to include in either a "pos" or "neg" lexicon as it would not be an informative feature for the model to learn from. When considering lexical items to include or exclude, the level of ambiguity is essential to consider, and it may require significant effort to build high quality, unambiguous sentiment lexicons. (One might consider this effort an annotation task in itself, where instead of labeling documents, one is labeling lexical items with sentiment labels.)

NOTE

After developing any lexicons, it is vital to double-check that no entries overlap across the lexicons for different labels. If entries overlap, they are not unambiguous, and should be discarded from all lexicons they appear in.

10. Overview of Keyword-Based Classification Training

If you have sufficient training data, you can use the techniques described earlier to build and evaluate a classifier. If you don't have any data, you might consider a keyword-based classifier which is very quick and easy to create, since you select your own keywords, and you don't need to collect, clean, and annotate data. Currently, a keyword-based classifier can only be trained for English using the FTK.

First, create a directory to hold the keyword model, and copy in the "semantic index":

```
$ mkdir keyword_model
$ cp -r data/esa/index keyword_model
```

Next, create a taxonomy file with your keywords and phrases. We ship a sample one based on the IAB QAG Taxonomy:

```
[~/tmp/tcat-<version>]
$ head -20 etc/iab_taxonomy.txt
ARTS_AND_ENTERTAINMENT
  "Arts & Entertainment"
  "Books & Literature"
  "Celebrity Fan/Gossip"
  "Fine Art"
  Humor
  Movies
  Music
  Television
AUTOMOTIVE
  Automotive
  "Auto Parts"
  "Auto Repair"
  "Buying/Selling Cars"
  "Car Culture"
  "Certified Pre-Owned"
  Convertible
  Coupe
  Crossover
  Diesel
```

Note that keyphrases are enclosed in quotes, while keywords are not. Without the quotes, each token in a keyphrase will be considered individually, as if each token appeared on its own line. Thus, you may often want to have overlapping keyphrases and keywords, e.g. include both "Auto Parts" and Auto for AUTOMOTIVE.

Next, train your model by building a concept vector for each category. Below, we choose the best 1000 concepts per category:

```
$ bin/ConceptVectorsBuilder 1000 keyword_model/index \
etc/iab_taxonomy.txt keyword_model/concepts
```

This command turns your keywords into a list of Wikipedia pages related to your categories. We treat each Wikipedia page as a separate concept. You can look at the resulting concepts file to make sure things make sense:

```
$ grep AUTOMOTIVE keyword_model/concepts | head
AUTOMOTIVE      Sport_utility_vehicle
```

```
AUTOMOTIVE      Trumpchi
AUTOMOTIVE      Luxury_vehicle
AUTOMOTIVE      Motor_vehicle
AUTOMOTIVE      BYD_Auto
AUTOMOTIVE      Crossover_(automobile)
AUTOMOTIVE      Aftermarket_(automotive)
AUTOMOTIVE      Lifan_Group
AUTOMOTIVE      Honda_Accord
AUTOMOTIVE      Automotive_industry_in_Japan
```

You can remove keywords that don't look useful or add ones you feel are missing, either by adding/removing keywords, or by dealing directly with the generated concepts file. These activities fine-tune your classifier.

Now you are ready to classify new documents:

```
$ bin/TCatCLI -m keyword_model -o /dev/stdout \
<(echo "the red sox won at fenway") \
<(echo "i drive a ford focus") | cut -f1 2>/dev/null
SPORTS
AUTOMOTIVE
```

10.1. How the Keyword-Based Classifier Works

The keyword-based classifier works by pivoting through Wikipedia. We have processed Wikipedia into a local artifact so there is no need for an internet connection. Our version of Wikipedia has been pruned to remove uninteresting pages. We call this our *semantic index*.

Consider each page in this index to be a concept. We represent a document as a concept vector. A concept vector is a mathematical representation of the meaning (concept) of a document. Documents about similar concepts will have vectors that are closer together.

For example, the sentence "The Red Sox play at Fenway Park" might have a vector that looks like this:

```
$ bin/SemanticInterpreterCLI --max-concepts 10000 eng keyword_model/index \
<(echo "The Red Sox play at Fenway Park.")
Fenway_Sports_Group     36.049507
Fenway_Par              34.954315
Boston_Red_Sox          33.44084
Portland_Sea_Dog        32.253838
Huntington_Avenue_Grounds    30.448547
Green_Monster           28.773073
...
Brooklyn_Cyclones       7.8293853
Edgeley_Park            7.8293853
Motherwell_F.C.         7.8293853
Newcastle_Falcons       7.8293853
Rochester_Rhinos        7.8293853
Ellis_Park_Stadium      7.8200245
...
Roland_Winters          7.5505514
...
Yu_(wind_instrument)    4.427081
```

You can imagine this vector as having one slot for every page in Wikipedia, but most slots will be fairly low. For example, Roland Winters (an actor) is ranked 1000 on this list and is only vaguely related to the Red Sox - he was born in Boston. Item 10000 on this list is Yu (wind instrument), with score a of 4.427081, and it's not clear how it's related to the Red Sox at all.

When you train a category using keywords, you are combining weighted concept vectors. The overall vector for your category should end up having clearly related concepts at the top.

At runtime, the input document is also turned into a concept vector by querying our local copy of Wikipedia. This input concept vector is then compared against the concept vectors for each category your classifier has been trained on. We choose the category that has the closest vector distance.

Because this works by pivoting through Wikipedia, keyword-based classifiers will not work well for categories without clear Wikipedia-based concepts. For example, a sentiment model would not be a good fit. Also, fine-grained categories like iPhone6 vs. iPhone7 may also not be a good fit.

11. Integrating Your Custom Model with Rosette Server

To deploy your custom-trained model, integrate it into Rosette Server as follows:

- Ensure that, for the language your are targeting, the following directory exists: `${tcat-root}/models/<lang>/combined-iab-qag`
- Move any existing model files in the target directory to an `unused` directory, e.g.

```
> mkdir ${tcat-root}/models/<lang>/unused
> mv ${tcat-root}/models/<lang>/combined-iab-qag/* ${tcat-root}/models/<lang>/unused
```

- Copy all the model files from your newly trained model into your target directory, `${tcat-root}/models/<lang>/combined-iab-qag`
- Relaunch the Rosette Server server

After relaunching the Rosette Server, the `categorization` endpoint will use the models in the `combined-iab-qag` directory, therefore using your new model for the language of the newmodel.

For sentiment model integration, place your model files into `${sentiment-root}/data/svm/<lang>/` and use the `sentiment` endpoint. Similarly, move all existing files for that model to a backup directory before copying over the new files.

NOTE

Note that depending on your specific FTK version, your newly created model may have a `lexicon_filtered` file while the existing model has `lexicon.filtered` instead. Rosette supports both naming schemes for backwards compatibility. Regardless of which naming scheme you see, you should remove the existing filtered lexicon file before adding the one from your new model. If both `lexicon.filtered` and `lexicon_filtered` files are in the same model directory, `lexicon.filtered` will take precedence.

11.1. Adding New Language Models

Out of the box, the `/sentiment` and `/categories` endpoints only support the languages of the models that ship with the distribution. Once you have trained a model in a new language, you must add the new languages to the `transport-rules.tsv` and `worker-config.yaml` files in Rosette Enteprise.

- For both endpoints, edit the `transport-rules.tsv` file. Each endpoint is listed, with a `lang=` statement listing the supported languages for the endpoint. Add the three letter ISO 693-3 language code for the new model languages.

```
/categories lang=eng
/sentiment  lang=ara|eng|fas|fra|jpn|spa
```

- For the `/sentiment` endpoint only, edit the `worker-config.yaml` file. Go to the section labeled `textPipelines`. Each endpoint is listed with a `languages:` statement listing the supported languages for the endpoint. Add the three letter ISO 693-3 language code for the new model languages.

```
# sentiment
- endpoint: /sentiment
  languages: [ 'ara', 'eng', 'fas', 'fra', 'jpn', 'spa' ]
  steps:
  - componentName: entity-extraction
  - componentName: sentiment
```

12. Glossary

While these specific terms may have broader definitions in other contexts, within the scope of this document, we will use the following definitions.

Annotation	The process of assigning labels to documents
Category	a class or division of documents
Classification	The process of automatically annotating documents using a classifier (by machine)
Classifier	A system that assigns labels to documents
Confusion List	A representation of the false positives and false negatives for a given evaluation run
Confusion Matrix	A representation of the true positives, false positives, and false negatives for a given evaluation run
Corpus	a collection (literally "body of") documents
Cross-Validation	A quick and simple way to validate a model once it's been created, even if you have a small amount of data
Data	Written prose (we are only concerned with natural language data)
Development Set	A subset of data held out from training, but used during the tuning process to improve the classifier
Document	A discrete, cohesive unit of data (e.g. a tweet, a letter, a news article etc. - this is intentionally loosely defined.

Evaluation Set	A subset of data used to evaluate the quality of the classifier. This subset should never be inspected or reviewed.
F1 Score	A weighted average of precision and recall, where an F1 score reaches its best value at 1 and worst at 0. The harmonic mean of precision and recall.
False Negative	Cases where the classifier prediction says it does not match a label, but it really does match the label
False Positive	Cases where the classifier prediction says it matches a label, but it does not.
Gold Label	The correct label
Label	The name of a particular category
Manual annotation	The process of annotating by hand (by humans)
Model	A set of categories or labels (and possibly relationships between categories); this sense of the term 'model' is more specifically a 'data model' which we will use in this context[3]
N-Fold Cross Validation	A validation methodology where the original sample is randomly partitioned into n equal sized subsamples. Of the n subsamples, a single subsample is retained as the validation data for testing the model, and the remaining n − 1 subsamples are used as training data.
NLP	Natural language processing, referring to a set of technologies to analyze text written by people
Precision	The proportion of the positive identifications which are correct
Recall	The proportion of the positives that were correctly identified
Taxonomy	A model exhibiting a hierarchy or tree-like structure branching from most general to most specific
Testing Corpora	The data used for testing, as opposed to, training the classifier. It is comprised of the development set and the evaluation set.
Training Set	The data used to train the classifier
True Positive	Cases where the classifier prediction correctly matches a label

[3]Note that another sense of the term "model" is a "machine learning model": a collection of parameters and statistics learned from sample data. We will refer to "machine learning models" in this context as "classifiers" to avoid overloading the term "model".