SeqAnalyzer

Maintenance Guide

Daniel Glickman

Project Overview

The project provides a framework for analyzing sequential data.
It offers modules for processing data and extracting statistically significant features from it, as well as modules for selecting, from these features, those that can differentiate between different sets of sequential data. We show how this is easily extended to perform classification of new data by having the project integrate with a popular machine learning library.
For extraction, It effectively uses a highly optimized java program, which allows working with very large quantities of data, but can also work as a standalone by having a simpler extractor written in python.

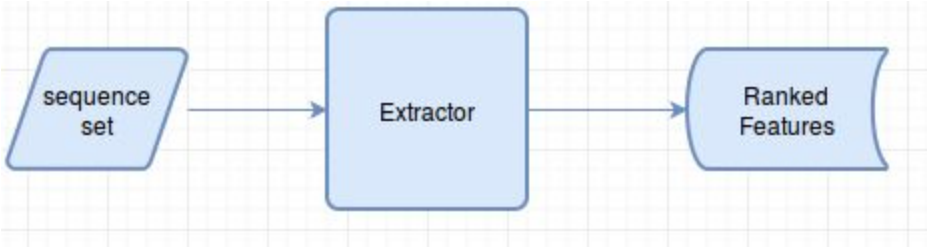
Glossary

Sequence - the basic input unit.
e.g a DNA sequence or an english sentence
Set - a group of sequences who belong to the same one class.
E.g one or many files with DNA sequences who share a common trait , a list of sentences from the same text.
Feature(of a sequence) - an interesting subsequence.each feature has a score.
E.g dna motif or english word.

Introduction

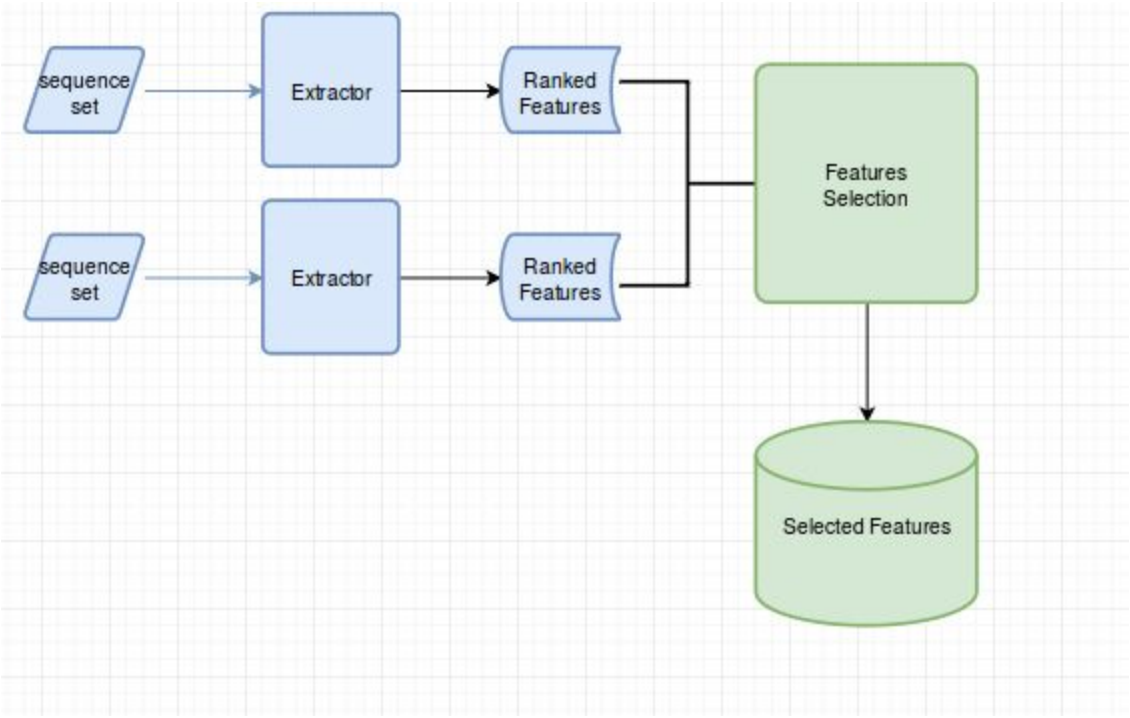
There are 3 tasks that can be performed, each builds upon the previous one and each one is in itself useful.

Extraction:



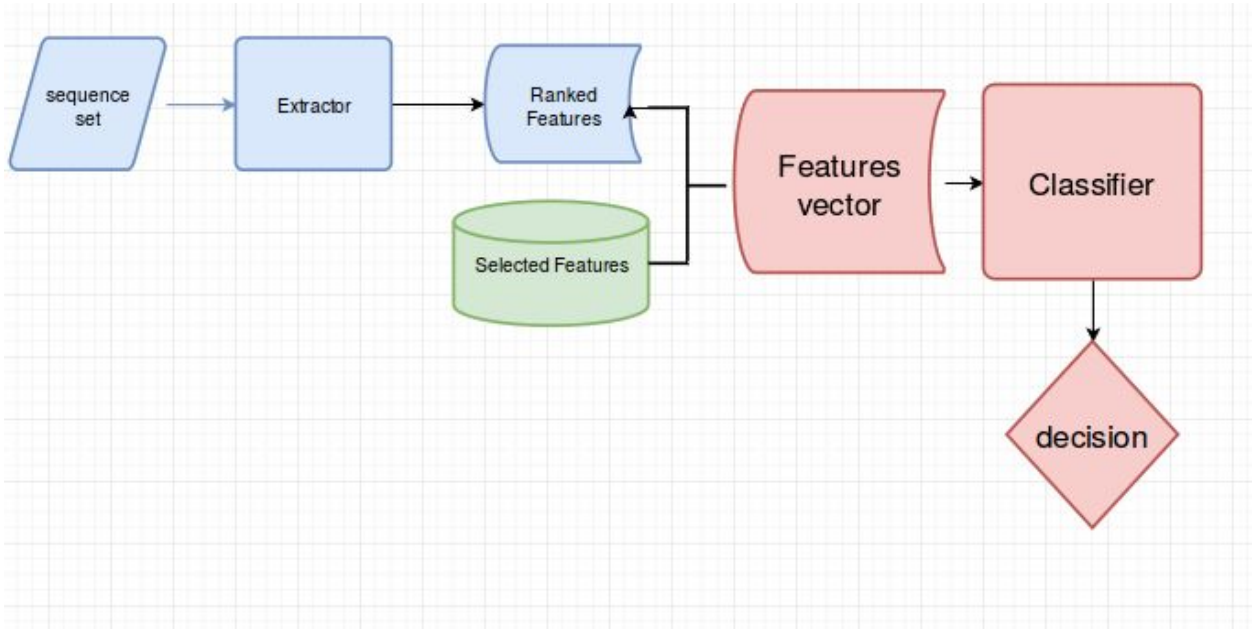
Given a set of sequences, the extractors extract initial features from the sequences and score each one according to some scoring function, ranking its statistical significant.

Selection:



With the extractor having limited filtering capabilities, the number of subsequences(features) it outputs can reach many millions.
In order to be understandable by humans or allow for a reasonable runtime of analysis, the number of features extracted must be efficiently filtered in a way that helps our goal.
Feature selection tries to find only those features that help differentiate between the 2 input sets.

Classification**:**



A dense feature vector is created for each sequence set by extracting features and then taking only these that were chosen during selection. Such vectors can be used for training a classifier on a reference set and predict the class of unknown sets.

Core Program Structure

This section covers the important and not obvious parts of the core code and how they interact together.

Preprocessors

Preprocessors are meant to transform the original, raw input data into data in a format that can be processed by the extractors layer.
For example, if the extractor expects sequences to be separated by a new line, and the original input file contains DNA sequences alongside other data such as tags and meta data, the preprocessor will create a new file with only the the sequences themselves for the extractor to read.

Pre_Processor - is the base preprocessor class.
It takes the original input and calls each of it's processing functions on it sequentially.
Text_PreProcessor - defines and uses functions for processing english text.

To work with DNA files , a custom preprocessor should be made. The implementation will depend on the file format.

Extractors

The extractors package extracts features from the input data. These are initial features, which are usually of very large volume and have been through minimal filtering. For example an extractor may provide all the subsequence in a set of sequences whose count exceeds a certain threshold.

The output of all extractors should be an iterable of (feature-name , feature-score) tuples, where a feature name is the subsequence itself and the score is a number indicating its significance.
the z-score of a feature count is used as its score.

Java_Extractor

This extractor extracts features from a given file using the suffix array java program.
As (the only) point of interaction between the 2 programs, this is a critical part of the code.
The code calls the java jar file with the appropriate parameters, so changes in the java program such as different input parameters or different directory path will lead to changes in this python class. To make changes easier, values that may change are read from the file Consts.py.
The extractor returns an iterable of (feature-name , feature-score) tuples, sorted by feature name.

To efficiently sort by name, which is required for selection, I used linux sort. To do that i had to first convert the java output file from utf-16 to utf-8. Such overhead can be avoided if the output is written in utf-8 in the first place.

Python extractors

Parts of the java program were taking a longer time to develop so in order to be able to test code and ideas and be independent of my partner schedule and changes to the program i developed an extractor written in python.

The implementation is memory inefficient as it uses hash tables and does not scale well. However it is still able to work with reasonably sized data and may be useful when having many sets of sequences that are small, for example DNA of a single person, since using the python extractor can save the overhead of calling another program and handling the I/O every time. This makes the python extractor a good choice for classification, when there are many small individual sets.

A good rule of thumb would be that if the python extractor can handle a single set it will handle many sets of the same size, since after extracting features from a single set, the number of features saved for that set will be greatly reduced by filtering out features that were not selected during feature selection.

Hash_Collection - features statistics and scores and computed independently for each possible subsequence length. To help doing so, Hash_Collection keeps a separate hash for each length while also implementing methods that allow accessing and iterating the data like a default python dict.

Suffixhash - extracts features from sequences by iterating their subsequences , counting them and scoring them using hash tables.

Z-Extractor - is the python extractor that is mostly used. It uses z as score. Note that the input for the java extractor is a directory name while here it is a list of strings.

Py_extractor - Z-Extractor wrapper for working with directories as input.

Iterators

To better separate the extractors from other layers, extractors should always return an iterable of (feature-name , feature-score) tuples as output. In practice the data extracted by different extractors is stored in different ways that have different requirement. While with the java extractor it is stored in a sorted text file that we wish to avoid loading all to memory, in python's case, it is loaded to a hash collection and is not sorted.
For dealing with this issue, the iterators file defines decorator functions that wrap the data and change access to it according to the needs.

The file defines generators objects that reads from a file line by line , without loading it to memory. It also defines parsers that transform the read lines to tuples according to the file format and wrappers that allow iteration.

Another issue was that a library i was using required the data object to implement the iteritems method. I solved the issue, for the java output, by wrapping it with the class *iteritemswrapper* and for python's extractor output, which is a *hash_collection*, i implemented iteritems as a class method.

Features Selection

After extracting initial features from 2 sets, feature selection selects a subsets of the features who are the most important, reducing the amount of features by many factors.

Selection functions

selection_functions.py defines the different functions that can be used for selection. These functions take scores of a feature, according to both sets, and return a number indicating how good the feature is.

Note that with some functions, by mathematical definition, the best fit is when the function value is lowest, while for others the higher values are those of interest. There is also the fact that some functions are symmetric while others are not.

To make the selection process compatible with all functions, it is required of selection functions implementation to be:

1) Asymmetrical - f(a,b) = -f(b,a)
2) Optimal when **absolute** value of f is **highest**

This way the features chosen as characteristic of set A are the ones with the highest selection function values and those chosen for set B are with the lowest values.
This also means that the functions implementations may be different from their mathematical definition so that they fit the requirements.

Defining new functions in selection_functions.py will automatically add them to the selector factory and make them useable from Params.py

Merge Feature Selector - which implements the selector interface, is the class that does the selection given outputs of 2 extractors. The selector is loaded with a selection function and uses merge with a custom heap data structure.

Difference Selector - this is an old file that works only with hash tables as input. It is replaced by the more generic Merge Feature Selector but I left it anyway as it may be faster for some cases.

Merge and Heap:

It is important to realize that this is a critical point of the program and a potential bottleneck.
Up until now dealing with massive amounts of features was either done by the java program, which was highly optimized to be memory efficient, or in some cases, such as sorting, with linux OS calls.Now at this point, we go from many millions features to selecting just a much smaller portion of them, but to do so we must pass through our python code.

To do that, Merge_selector uses the heap class and merge function who take memory limitations to account.

Merge - merge works similarly to the merge known from merge sort but is designed to avoid loading all of the data to memory. To do so it receives 2 tuples iterables, for example iterables that return lines from the java extractor output file sorted by name, and manually iterates both at the same time while keeping items in order.

Merge currently uses next on the iterable.It should be changed to iterating by first calling the iteritems method, this will create better separation of layers.

Heap is a binary heap that is limited in size by the number of features we wish to select, allowing getting the top k features in O(k) and lower overhead compared to the default heap.

Still, if run time needs to be improved, merge and Heap are good places to start looking at for reducing some overhead.

Control

Feature Manager encapsulates extraction and selection to automate the program's workflow.
On initialization the class extracts features from 2 different sources and then uses features selectors to build its selected features database.
Once initialized, the transform method will take new raw input sources and return dense vectors of selected features that can be stored in memory.

For filtering vectors feature manager uses sk-learn's DictVectorizer.
Currently it only works with the python extractor.

A word about classification

Without well defined data format and data to test the classifier specifics with, there was little sense in writing generic code to do the classification process from start to finish.Once these are defined, classification should not be hard to do.
The code is built to integrate with sk-learn, a popular machine learning library.
The examples package provides examples of classification.

Integration with scikit-learn

Our project deals with data in a non standard way. Processing sequences, extracting massive amounts of subsequences and making sense from those using efficient feature selection all required special treatment. However, by the end of the process, we are left with a system that can take the desired data as input and return a reasonably sized vector of selected features.
Such representation is general and can be plugged in easily in standard machine learning algorithms. Integrating with a machine learning library instantly opens new ways to play with our data.

Scikit offers a Pipeline class that assembles several steps in the learning process such as preprocessing, classification and clustering by applying them sequentially.
The “steps” in the pipeline must implement the transform interface , that is they must implement fit and transform methods.
In the case of this project, the classes that implement this interface are *Processor, Extractor and Feature_Manager.*

Processor is first called to prepare the data for extraction and then the Feature Manager returns vectors in a format sklearn can work with. Examples show how integration is done easily.

Other code

Code that handles testing and evaluating the program and helper functions.

Data generation

Was used mostly at the beginning of the project for evaluating the extraction with z score.Able to:

- *create synthetic datasets of randomly generated sequences of a given alphabet*
- *insert random noise at different levels to existing sequences.*
- *automate the creation of datasets in the required format and with different levels of random noise .*
- *recreate to experiment from this paper* [*http://u.cs.biu.ac.il/~galk/Publications/Papers/ida05.pdf*](http://u.cs.biu.ac.il/~galk/Publications/Papers/ida05.pdf)

Utility

Functions to help reading command line input , using system calls and saving and loading data.

Scripts

Scripts to sort and convert files and extract results.

Tests and examples

Includes examples that evaluate extraction , selection and classification, by themselves and how they affect each other.

Future extensions

1) Make Feature Manager work with java extractors too.

2) After feature selection there are a lot of redundant features.
For example if the word “computer” shows as a very good feature, suffixes and infixes , like “compute” and “omputer” will also show high.
Taking care of that should added to the selection process

3)Change the output format :
This is an idea of saving space and time by changing the format that the java outputs and python reads.
Currently the suffix-array results are stored in 1 file in the format:
score(e.g z) count string

e.g:

12.34	43	the
2.5	3	book
2.5	3	blue
2.5	3	door
0.4	2	ball
4.6	3	table
4.6	3	mouse

This table has a lot of redundancy as there are a lot of strings with the same count in a given length.

We can avoid writing their score and count multiple times.

for example by using a format like:
$ LENGTH 4 $
#COUNT 1 : Z=..
.
.
.
#COUNT 2 : Z = 0.4
ball
.
.
#COUNT 3 : Z = 2.5
book
blue
door

$ LENGTH 5 $
#COUNT 1 : Z=..
.
.
.
.

#COUNT 3 : Z=4.5
table
mouse

.
.
.
$ LENGTH N $
.

.

So by using this format, that shares data, we can save about 2/3 of the space(by not writing count and z multiple times).

But we can do better,
for calculating the z score of a given length what we need is a distribution(or a histogram) of the counts in that length. For example: (*1*)

count	unique strings in length 3 with that count
1	99999
2	1234
3	100
4	50

With such format it is easy to calculate the mean and std and calculate the z score of a string in O(1). In fact we can replace Z with any scoring function that is based on the distribution of counts with such format, because the file above is exactly the histogram of a specific length.

So improving on this idea, we can write the histogram/distribution of every length to a separate file.
In addition, each length and count we will have its own file with all strings in that specific length and count.
for example we will have a file called "4.3" this file will look like:
book
blue
door

We are saving space by grouping shared data, just like with the first format example, but with this format we can analyse the data by groups rather than individual strings.
For example what if we want the top 150 strings of length 3(example *1*)? We just need to load the distribution for length 3, that is in a separate distribution file, and look what counts(=files now) contain the top 150.
This is easy because the distribution is very very small compared to the data(features names) itself.
Using this metadata we just read 3.4 and some of 3.3 files, and by doing so, making the top k operation O(1)(or O(k) when actually reading) instead of O(n).
Generalizing the operation strings of all lengths is also easy.