Daxmod: A Toolbox for Text Classification

Ahmed Tchagnaou
University of Tours, France
ahmed.tchagnaou@etu.univ-tours.fr

Dominique H. Li
LIFAT Laboratory, University of Tours, France
dominique.li@univ-tours.fr

Abstract

Daxmod is a Python toolbox designed for, but not limited-vto, the simplification of text classification tasks. **Daxmod** includes modules for data loading, feature extraction, feature selection, model building, validation, and object persistence. It has a consistent API that enables developers and researchers to conduct experiments and perform various tasks efficiently. **Daxmod** provides an extensible pipeline for text classification and the ability to reuse trained objects and conduct reproducible research work. The source code of **Daxmod** is available at https://github.com/Authentic10/daxmod.

CCS Concepts: • **Computing methodologies** → **Machine learning**.

Keywords: python, text classification, feature extraction, feature selection

ACM Reference Format:
Ahmed Tchagnaou and Dominique H. Li. 2022. Daxmod: A Toolbox for Text Classification. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Text classification is one of the most common tasks in data mining and machine learning. As textual data is generated continuously, appropriate methods for gaining insights into the data are required to aid research and business applications in understanding user needs and improving services. Many techniques have emerged to handle text classification problems since their inception. Among numerous approaches, the most effective ones are machine learning, particularly deep learning based [7, 12]. Because of the proliferation of libraries and toolboxes, learning methods are now easily accessible and widely applied in research and application domains [1, 4, 8, 11].

CIKM '22, October 17–22, 2022, Atlanta, GA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X...$15.00
https://doi.org/XXXXXXX.XXXXXXX

Despite the availability of algorithms, in general, users have to define their problem pipeline and write custom functions to perform specific tasks. Furthermore, machine learning necessitates to extract features that will feed the models. Therefore, many steps are required to build a text classification model. The workflows and algorithms are determined by the type of classification tasks, the size of the dataset, and other factors. It is hard to find a tool that supports multiple workflows and solutions. Therefore, managing the pipeline becomes complex because not all modules follow the same API structure. Furthermore, as user code base expand, the maintenance of experiment code is a time-costing job.

In this paper, we present an efficient pipeline for text classification-related workflows. Our work resulted in the development of **Daxmod** (**DA**ta, e**X**traction, and **MOD**el)[1], a Python-based toolbox, with consistent APIs for performing text classification tasks (but not limited to). We note that **Daxmod** also provides API interface to algorithm developers in order to make focus on the algorithm itself but not to concentrate the attention on creating test pipelines.

The rest of this paper is organized as follows. Section 2 briefly introduces related work. With examples, We detail the structure of **Daxmod** in Section 3 and the workflow of **Daxmod** in Section 4. Finally, we conclude in Section 5.

2 Related work

Daxmod was motivated by the need to automate tasks for problems encountered while working on text classification. In this section, we introduce the most common methods, workflows, and existing tools in text classification.

Many solutions have achieved effective results using word representation methods such as Bag of Words, TF-IDF, and N-grams [2, 13]. In particular, there are several methods for feature selection: dimensionality reduction, which creates new features from existing ones, or feature selection, which reduces the number of features by selecting the best ones [5, 10]. Regarding the classifier algorithms, the most common ones are SVM and Naive Bayes [5, 7, 10, 13]. They are frequently considered as baselines. It is mentioned that neural network-based algorithms outperform classical machine learning algorithms in text classification [7].

Several libraries and toolboxes are available for text classification [4, 8, 14]. However, the complexity they assist with requires a specific workflow or the dataset to be in a specific

[1] https://github.com/Authentic10/daxmod

format to be processed. General machine learning frameworks such as **scikit-learn** [11] and **Tensorflow** [1] include modules for text-related tasks. These modules have numerous features and can be tuned to solve specific text classification tasks.

3 Daxmod

The purpose of **Daxmod** is to provide a simple and easy-to-use toolbox for text classification. We built **Daxmod** with different modules: (1) Access; (2) Extraction; (3) Selection; (4) Classifiers; and (5) Persistence. Figure 1 shows the structure of **Daxmod**.

Daxmod allows customizing the existing implementations and extending the features by adding custom functions and other algorithms. It mainly uses the structure of the **scikit-learn** API. We focus on providing the features required to conduct experiments and build models rapidly. The toolbox is designed to perform the various tasks involved in building text classification models and reusing them. The module access is used to load data, the module extraction to extract features from the data, the module selection for feature selection, the module classifiers to build or use pre-trained classifiers, and the module persistence to save and load models.

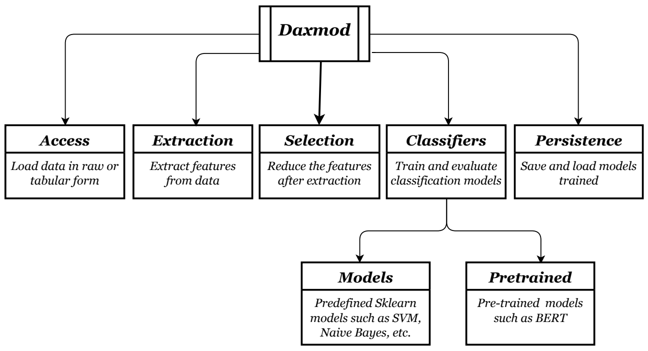
Daxmod
Access
Load data in raw or tabular form
Extraction
Extract features from data
Selection
Reduce the features after extraction
Classifiers
Train and evaluate classification models
Persistence
Save and load models trained
Models
Predefined Sklearn models such as SVM, Naive Bayes, etc.
Pretrained
Pre-trained models such as BERT

Figure 1. Daxmod package structure.

3.1 Access

Loading data is the first step in creating models. Text data can be presented in CSV-like or raw format; **Daxmod** provides methods for loading data in both cases. In the latter, as shown in Figure 2, files are stored in a folder named by their class. We assume that a user provides a folder that contains two or more sub-folders. The sub-folders are a train set, a test set, and eventually a validation set. Each sub-folder contains folders corresponding to the classes in the dataset. For instance, the folder names for binary sentiment analysis will be positive and negative. This file organization allows us to load different subsets of data with this module. The data loading functions return a pandas dataframe[9] to permit users to perform data analysis and cleaning.

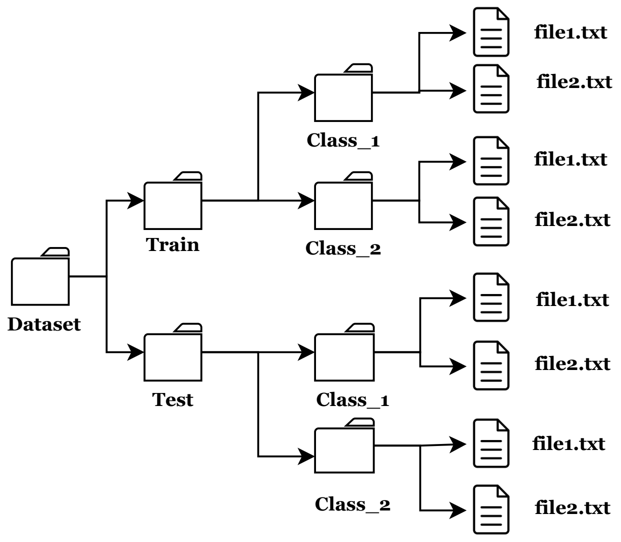

Figure 2. Binary classification folder structure

```
# Load data in raw format
from daxmod.access import load_from_folder

train_data = load_from_folder(dataset_folder='imdb')

# Load data in CSV format
from daxmod.access import load_from_file

data = load_from_file(filepath='imdb.csv')
```

Figure 3. Load data with the access module

Figure 3 shows how to load data with the module **Access**. In this example, we loaded a dataset corresponding to the folder structure in Figure 2 and a CSV file.

3.2 Extraction

It is necessary to transform the text into a form the algorithms can process. This step is crucial in text classification because it can impact the result of the task. In this module, we provide different feature extraction methods.

The predefined extraction methods in this module include Bag of Words, Bigrams, Trigrams, TF, and TF-IDF. We also provide classes to support various scenarios, such as using characters instead of words as tokens and creating custom N-gram extractions.

Figures 4 and 5 show how to extract features from data with **Daxmod**. We used the bag of words method to extract features from data in Figure 4, and showed how to create a custom N-grams extractor in Figure 5.

```python
# Import Extractors class from the extractor module
from daxmod.extraction import Extractors

# Instantiate a bag of words extractor
extractor = Extractors(extractor='bow')

# Fit the extractor and transform the data
extractor.fit(X, y)
X_transformed = extractor.transform(X)
```

Figure 4. Extract features with bag of words

```python
# Import Ngrams to define custom extractor
from daxmod.predefined.extractors import Ngrams

# Create custom N-grams with unigrams and bigrams
extractor = Ngrams(ngram_range=(1,2))

# Fit and transform the data with the created Ngrams
extractor.fit(X, y)
X_transformed = extractor.transform(X)
```

Figure 5. Extract features with custom N-grams

3.3 Selection

When we have a large corpus and use extraction methods such as N-grams, the extraction step returns a high-dimensional feature set. Depending on the task and the available resources, we may not require all the features generated. A solution is to reduce the number of features with the several feature selection algorithms available[3]. This module contains a class called **SelectTopK**, which select the best features by analyzing the data.

Daxmod currently provides two methods for determining the best features from data: analysis of variance and chi-squared test. Both are available via the **SelectTopK** class.

```python
# Import the SelectTopK class from the module selection
from daxmod.selection import SelectTopK

# Use ANOVA as selection method and 10K as the number of features to keep
topk = SelectTopK(score_func='anova', k=10000)

# Fit and select the best features
topk.fit(X_transformed, y)
X_top = topk.transform(X_transformed)
```

Figure 6. Selection of features with ANOVA

Figure 6 shows the usage of the module **Selection** to reduce the features from data. In this example, we used the analysis of variance method to select the best 10,000 features from the data.

3.4 Classifiers

The next step after feature extraction and selection in a text classification task is to train and evaluate a classifier that will work well on an unknown dataset. Users typically test different features, algorithms, and hyperparameters to build classifiers. It is important to follow general practices, such as maintaining a test set for evaluating the model or performing cross-validation. The module classifier contains two sub-modules: **Models** for **scikit-learn** classifiers and **Pre-trained** for pre-trained models defined with **Tensorflow**.

The module **Models** includes a class that allows using pre-defined classifiers like SVM, Naive Bayes, and Multi-layer Perceptron. Other algorithms from **scikit-learn** can be used as well. To train a model with this module, the user can specify the classifier name when initializing the class.

The module also includes a method for evaluating the model on a dataset with different metrics using the trained classifier. The supported metrics are accuracy, balanced_accuracy, f1, precision, and recall.

The pretrained module has an implementation of a classifier based on **BERT**. We used a light version of **BERT** for users' convenience. However, it is possible to modify some parameters for other versions of **BERT** from the **Tensorflow** hub. For rapid experimentation, we defined a method for auto-compiling the classifier with the appropriate metric, loss, and optimizer. We can also compile the model with other metrics, losses, and optimizers from **Tensorflow**. The pre-trained module also provides **BERTLayer**, which is practical to define custom models with more layers and others.

```python
# Import the Models class from daxmod
from daxmod.classifiers.models import Models

# Instantiate an SVM classifier
model = Models(classifier='svm')

# Train the SVM classifier
model.fit(X_top, y)
```

Figure 7. Build an SVM classifier

Figures 7, 8, and 9 show how to train and evaluate a classifier with the module **Models**. In Figure 7, we created an SVM classifier by fitting it to the data. The Figure 8 shows how to use a classifier imported from **scikit-learn**, and Figure 9 shows how to evaluate a trained model.

```python
# Import RandomForest from scikit-learn
from sklearn.ensemble import RandomForestClassifier

# Pass an instance of the RandomForest as the parameter to Models
model = Models(classifier=RandomForestClassifier())
```

Figure 8. Use RandomForest from **scikit-learn**

3.5 Persistence

Joblib [6] provides model persistence methods. While conducting experiments, users typically test various algorithms.

```
# Load the test set
...............................
# Transform the test set with the extractor and the selector
X_test_transformed = extractor.transform(X_test)
X_test_top = topk.transform(X_test_transformed)

# Evaluate the model on the test set (accuracy)
model.evaluate(X_test_top, y_test, metric='accuracy')
```

Figure 9. Evaluate a classifier

Therefore, we can keep the built models to reuse them later. The main advantages are that the models are not trained again and that the results are reproducible. This module includes methods for saving and loading objects, it makes sharing, transporting, and reusing built models simple.

Many of the modules discussed above provide a way to save objects without importing the persistence class. However, this module is required to load a previously saved object.

```
# Save a model without the persistence module
model.save(name='svm', folder='models')

# Import the method save_object from the persistence module
from daxmod.persistence import save_object

# Save a model in the folder 'models' with 'svm' as name
save_object(obj=model, name='svm', folder='models')

# Import the method load_object from the persistence module
from daxmod.persistence import load_object

# Load a saved model
model = load_object(path='models/svm.model')
```

Figure 10. Save and load a model

Figure 10 shows how to use the module **Persistence**. In this example, we saved and loaded a trained model.

4 Building with Daxmod

Daxmod provides two main workflows for text classification: one based on building a model from scratch, **Building workflow**, and the other one, based on pretrained models, **Pretrained workflow**.

As shown in Figure 11, the building workflow performs text classification in three steps: feature extraction, feature selection, and model training and evaluation. The feature selection step is optional. Therefore, after extracting features, models can be built and evaluated directly. The pretrained workflow employs pre-trained models via transfer learning. We provide a model based on **BERT** that can be used immediately after loading data.

Figure 12 shows how to use the pre-trained model **BERT** in **Daxmod**. In this example, we built a classifier based on **BERT** for a binary classification task.

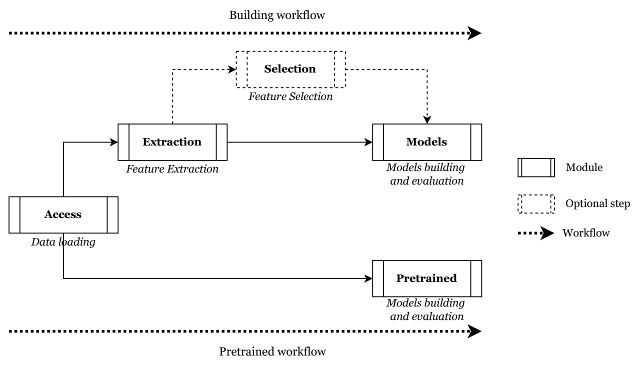

Figure 11. Daxmod workflows.

```
# Import Bert from pretrained module
from daxmod.classifiers.pretrained import Bert

# Instantiate bert for a binary classifier
bert = Bert(n_classes=2)

# Autocompile the model
bert.auto_compile()

# Train the model
bert.fit(X, y)

# Evaluate the model
bert.evaluate(X_test, y_test)
```

Figure 12. Build a classifier with BERT

Daxmod allows developers to extend the toolbox with other feature extraction, selection, and classification algorithms. For instance, to extend the available extractors with a new one, we can create a new class with the necessary methods in a file reserved for extractors. Then, we can add it to the list of the extractors, and it will be available in the **Extraction** module. This feature allows users to focus on the methods rather than the steps used by **Daxmod** to perform tasks.

5 Conclusion

Daxmod is a simple toolbox created for text classification tasks with Python. It provides modules and methods to perform text classification with various workflows. Although **Daxmod** is focused on text classification problems, however, its ability to load data in tabular form makes it possible to work on other types of datasets. **Daxmod** is still in its early stages of development. The code-base will be improved in the future, and new features such as word embeddings, other feature selection and classification algorithms, pre-trained models, and more will be added.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving,

Michael Isard, et al. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Ronen Feldman. 2013. Techniques and applications for sentiment analysis. *Commun. ACM* 56, 4 (2013), 82–89.

[3] George Forman et al. 2003. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.* 3, Mar (2003), 1289–1305.

[4] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[5] M Ikonomakis, Sotiris Kotsiantis, and V Tampakas. 2005. Text classification using machine learning techniques. *WSEAS transactions on computers* 4, 8 (2005), 966–974.

[6] Joblib Development Team. 2011. *Joblib*. https://joblib.readthedocs.io/

[7] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. 2019. Text classification algorithms: A survey. *Information* 10, 4 (2019), 150.

[8] Edward Loper and Steven Bird. 2002. Nltk: The natural language toolkit. *arXiv preprint cs/0205028* (2002).

[9] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

[10] Marcin Michał Mirończuk and Jarosław Protasiewicz. 2018. A recent overview of the state-of-the-art elements of text classification. *Expert Systems with Applications* 106 (2018), 36–54.

[11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[12] Duyu Tang, Bing Qin, and Ting Liu. 2015. Deep learning for sentiment analysis: successful approaches and future challenges. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 5, 6 (2015), 292–303.

[13] Sida I Wang and Christopher D Manning. 2012. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 90–94.

[14] H Yu, C Ho, Y Juan, and C Lin. 2013. Libshorttext: A library for short-text classification and analysis. *Rapport interne, Department of Computer Science, National Taiwan University* (2013).