Technical Report: MRDRC Project UM21-14

C-NARS: An Open-Source Tool for Classification of Narratives in Survey Data

Joelle Abramowitz and Jinseok Kim

Abstract

To help researchers and policy makers better understand how different types of self-employments contribute to older adults' income, retirement, and quality of life, this project develops a computational method to classify self-employment narratives in survey data. Among 17,854 job narratives in the Health and Retirement Study between 1994 and 2018, about 4,500 instances are labeled into one of three categories – Owner, Manager, and Independent - by human coders. A variety of machine learning algorithms are trained and tested on the labeled data in which each narrative text is pre-processed (lemmatization, stemming, etc.) and transformed into a vector of word tokens for cosine similarity calculation among narratives. The best performing classification model (Gradient Boosting Trees) is applied to the entire 17,854 instances to produce probability scores of an instance being likely to belong to each of three categories. A total of 14,748 instances with a probability score of 0.9 or above for 'Independent' or with a probability score of 0.6 or above for 'Owner' are filtered as accurately tagged instances because they are highly likely to be assigned correct categories (97.3% for Independent and 99.0% for Owner) when evaluated on 10 random subsets (20% of 4,500 instances each) of the labeled data. The remaining instances are passed to manual inspection and correction before the entire data are to be used for statistical analyses. The classification code sets – **C**lassification of **Nar**ratives in **S**urvey Data (C-NARS) - are made publicly available for researchers to implement machine learning methods for classification of narratives in survey data.

Overview of Algorithmic Classification Method

The project uses machine learning to automate the classification of occupation types – Owner, Manger, and Independent - depicted in narrative responses in the Health and Retirement Study. The machine learning approach uses manual classifications of a subset of the data into self-employment roles to automate classification of the remainder of the data. To do this, the approach performs several tasks: (1) pre-processing the raw data, (2) training candidate machine learning models, (3) developing and validation of the candidate models, and (4) testing and implementation of the optimal model. These tasks follow common procedures conducted in previous similar studies in predicting occupational or demographic classes using text data (Boselli et al, 2018; Ikudo et al, 2019; Lampos et al., 2016; Mac Kim et al., 2017; Preoţiuc-Pietro, Lampos, and Aletras, 2015; Preoţiuc-Pietro and Ungar, 2018).

Our approach first pre-processes all words in the occupation narratives in the Health and Retirement Study data. This pre-processing includes cleaning, standardization, stop-word removal, and stemming which are typical procedures in natural language processing (Berry and Castellanos, 2004). Then, the approach transforms the pre-processed words into a vector (list) of word tokens. Next, to train candidate machine learning models, the vector of tokens are fed into various machine learning algorithms including Logistic Regression, Naïve Bayes, Random Forests, Gradient Boosted Trees, and Support Vector Machine to learn frequencies and combinations of tokens that best associate each narrative to one of the classes using a subset of the data that have been manually coded. To develop and validate the candidate models, the approach identifies the best performing combinations of feature weights and increases generalizability of the machine learning models by implementing various modifications of parameters and data on another subset of the manually coded data. Finally, testing and implementation of the optimal classification model identified from this training and development and validation are applied to the remainder of the unclassified narratives to assign each narrative one of the occupation classes.

Technical Details

This project produces a code set that consists of two execution files with a file extension '.py' (CNARS_main.py and CNARS_training.py) and implements the machine learning procedure described in the Overview. The code is written in a script language ('python' version 3.6.5 or above) using open-source machine learning ('scikit-learn' version 0.21) and natural language processing ('nltk' version 3.6.3) packages. The code set is named 'C-NARS' (Classification of Narratives in Survey Data) and will be shared on public code repositories (GitHub[1] and FigShare[2]) for validation, reuse, and improvement by researchers.

Input Data

The code set accepts two input data sets – training data and test (target) data - in a csv file format. Figure 1 shows a partial preview of input data used for the project. Each input data set consist of three columns: Project id, narrative, and class. In the figure, project id and narrative information are blinded as the data sets used for this project contain IRB-protected information.

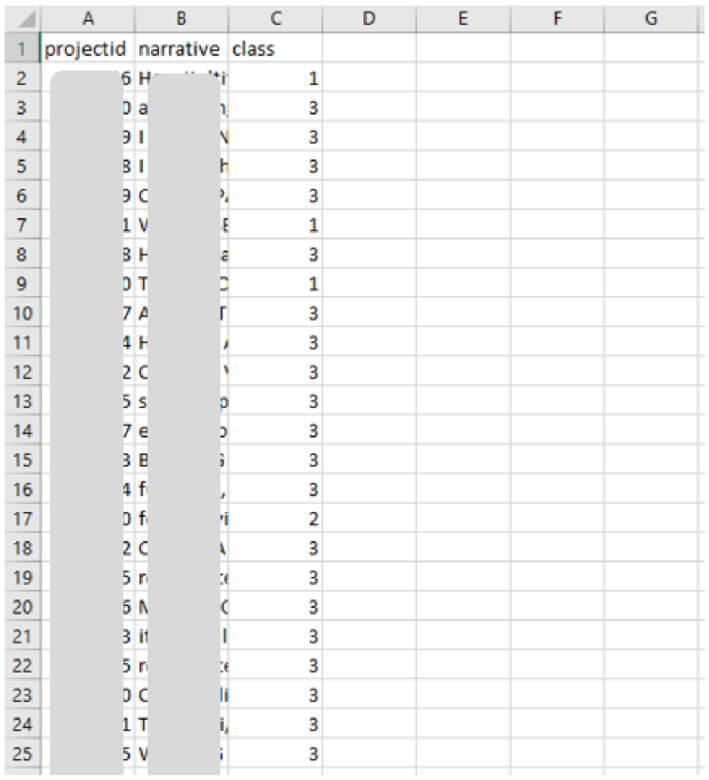
(1) Project id: this is a series of numbers or alphanumeric characters used to refer to a unique instance (case).

(2) Narrative: in this column, a textual narrative of job description is recorded.

(3) Class: One of occupational classes – for this project, Owner (1), Manager (2), or Independent – is assigned to the instance through manual coding. The classes can be recorded in integers or a string of characters (e.g., A, B, or C; or Owner, Manager, or Independent). This column needs to be populated if machine learning algorithms are trained to learn patterns of distinguishing classes (consisting of the training data).

[1] www.github.com
[2] www.figshare.com

Depending on the machine learning stages, however, this column can be empty. Once machine learning algorithms are trained on the training data, they are ready to predict classes in test data where classes for narratives are unknown.



	A	B	C	D	E	F	G
1	projectid	narrative	class				
2	[redacted]	[redacted]	1				
3	[redacted]	[redacted]	3				
4	[redacted]	[redacted]	3				
5	[redacted]	[redacted]	3				
6	[redacted]	[redacted]	3				
7	[redacted]	[redacted]	1				
8	[redacted]	[redacted]	3				
9	[redacted]	[redacted]	1				
10	[redacted]	[redacted]	3				
11	[redacted]	[redacted]	3				
12	[redacted]	[redacted]	3				
13	[redacted]	[redacted]	3				
14	[redacted]	[redacted]	3				
15	[redacted]	[redacted]	3				
16	[redacted]	[redacted]	3				
17	[redacted]	[redacted]	2				
18	[redacted]	[redacted]	3				
19	[redacted]	[redacted]	3				
20	[redacted]	[redacted]	3				
21	[redacted]	[redacted]	3				
22	[redacted]	[redacted]	3				
23	[redacted]	[redacted]	3				
24	[redacted]	[redacted]	3				
25	[redacted]	[redacted]	3				

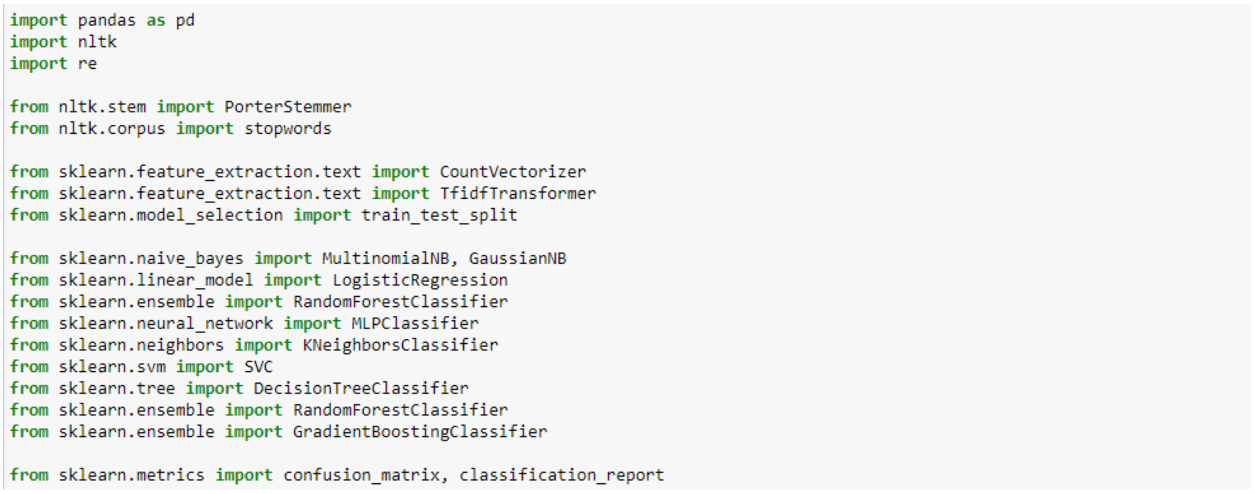
Figure 1: Example of Input Data for C-NARS

Execution Files

Two execution files - CNARS_main.py and CNARS_training.py – can be run on a command prompt by typing ‘python CNARS_main.py’ or ‘CNARS_training.py’ and pushing an Enter button. They are almost identical in data input, pre-processing, machine learning, and prediction steps - except that the CNARS_training file contains steps for random sampling of input data and an evaluation of machine learning performance.

Loading Modules

Each execution file needs to load python modules to perform routine functions and algorithmic implementations of machine learning procedures. Figure 2 shows the snippet of code that loads modules required for the project.



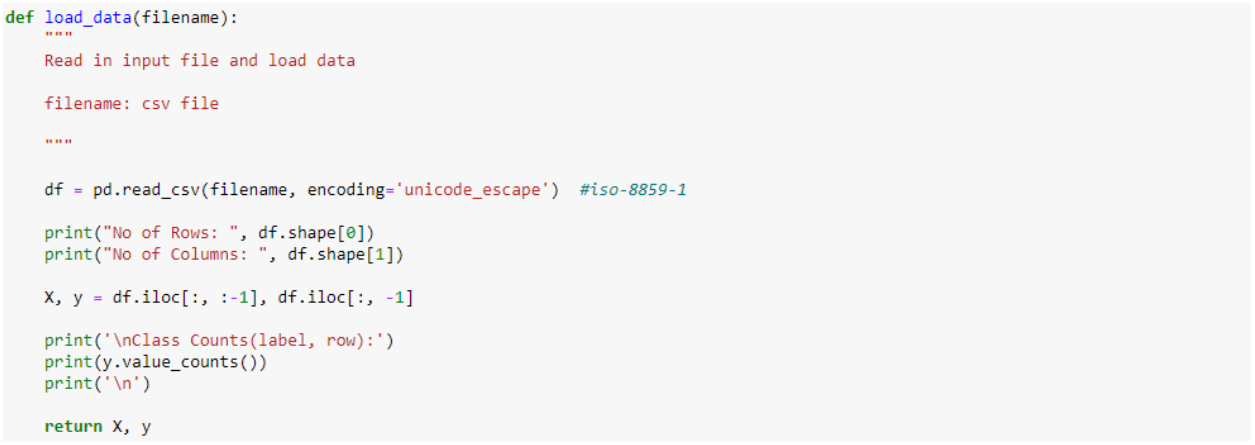
```python
import pandas as pd
import nltk
import re

from nltk.stem import PorterStemmer
from nltk.corpus import stopwords

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import MultinomialNB, GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.metrics import confusion_matrix, classification_report
```

Figure 2: Code Section for Loading Packages and Modules

Each module is imported from that packages that contain it. For example, 'PorterStemmer' is located in 'ntlk,' a python package for natural language processing. The packages required for this project do not come with the installation of python and, accordingly, need to be installed separately[3]

Loading Data

The next part of the execution files directs how input data are loaded. In Figure 3, a process of loading the input data ('filename') is defined ('def'). Although this project uses a CSV file format, files in any other formats such as tab-delimited ('.txt') or MS spreadsheet ('.xls') can be loaded by modifying the 'pd.read_csv' part.[4]



```python
def load_data(filename):
    """
    Read in input file and load data

    filename: csv file

    """

    df = pd.read_csv(filename, encoding='unicode_escape')  #iso-8859-1

    print("No of Rows: ", df.shape[0])
    print("No of Columns: ", df.shape[1])

    X, y = df.iloc[:, :-1], df.iloc[:, -1]

    print('\nClass Counts(label, row):')
    print(y.value_counts())
    print('\n')

    return X, y
```

Figure 3: Code Section for Loading Data

Another important note is that special care needs to be taken for selecting 'encoding' type. If not properly chosen, encoding issues can stop the code execution and produce warnings. For this

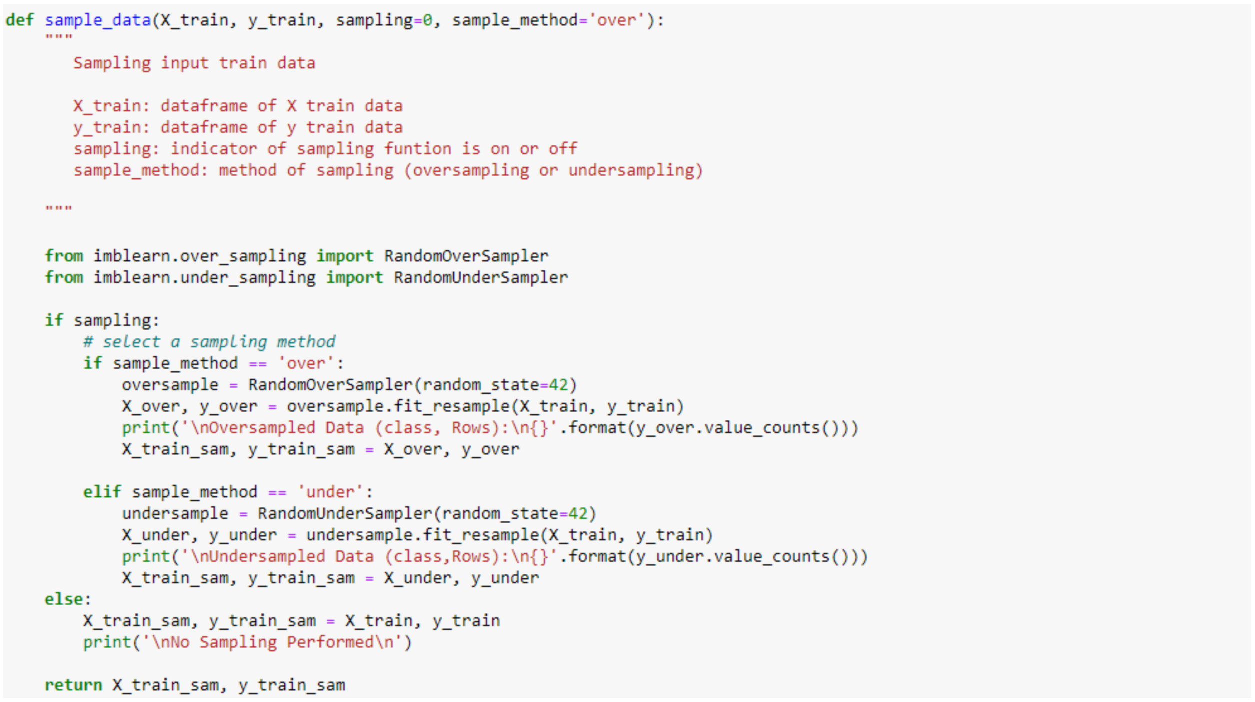
[3] For installing 'scikit-learn,' see https://scikit-learn.org/stable/install.html For installing 'nltk,' see https://www.nltk.org/install.html

[4] For more information, see https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

project, ‘unicode_escape’ was chosen, but depending on how the input file was created and what languages are used, another encoding scheme may need to be selected.[5]

Sampling Data

In much real world data, classes to be predicted can be imbalanced. In a prediction task of cancers from patients’ records, for example, positive cases (patients with cancers) tend to be smaller than negative ones (patients without cancers). Accordingly, labeled data created from target population data tend to have class imbalance. Many machine learning algorithms are sensitive to imbalanced training data, producing inaccurate predictions. To address this issue, this project includes in the code both over-sampling and under-sampling methods to create balanced training data. Figure 4 shows the code snippet for over- and under- sampling. Note that users need to install ‘imblearn’ to run the sampling techniques properly.



```python
def sample_data(X_train, y_train, sampling=0, sample_method='over'):
    """
       Sampling input train data

       X_train: dataframe of X train data
       y_train: dataframe of y train data
       sampling: indicator of sampling funtion is on or off
       sample_method: method of sampling (oversampling or undersampling)

    """

    from imblearn.over_sampling import RandomOverSampler
    from imblearn.under_sampling import RandomUnderSampler

    if sampling:
        # select a sampling method
        if sample_method == 'over':
            oversample = RandomOverSampler(random_state=42)
            X_over, y_over = oversample.fit_resample(X_train, y_train)
            print('\nOversampled Data (class, Rows):\n{}'.format(y_over.value_counts()))
            X_train_sam, y_train_sam = X_over, y_over

        elif sample_method == 'under':
            undersample = RandomUnderSampler(random_state=42)
            X_under, y_under = undersample.fit_resample(X_train, y_train)
            print('\nUndersampled Data (class,Rows):\n{}'.format(y_under.value_counts()))
            X_train_sam, y_train_sam = X_under, y_under
    else:
        X_train_sam, y_train_sam = X_train, y_train
        print('\nNo Sampling Performed\n')

    return X_train_sam, y_train_sam
```

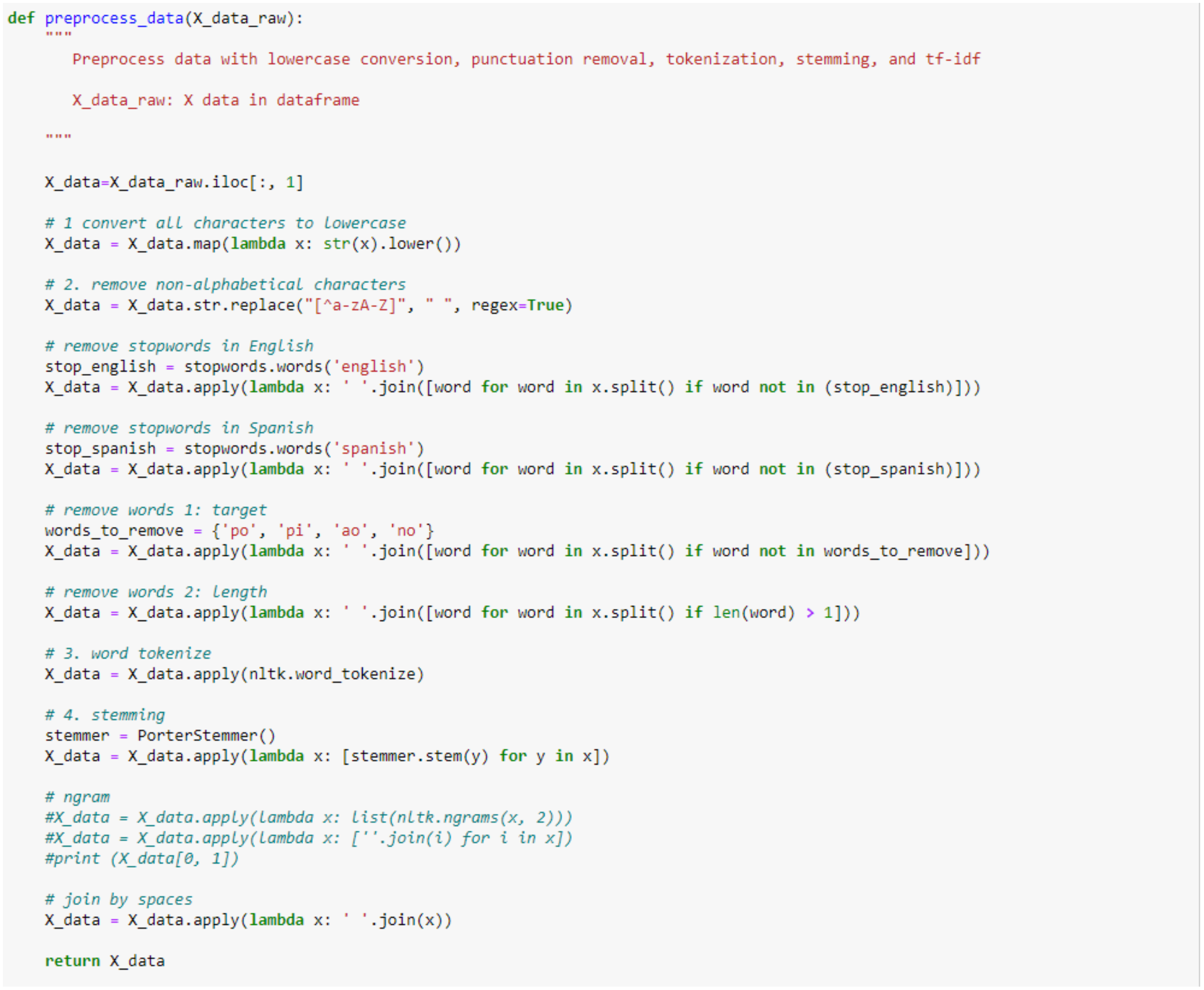
Figure 4: Code Section for Over- and Under-Sampling

Pre-processing Data

After input data are loaded, texts in Narrative go through a series of steps before they are used for machine learning and prediction. Figure 5 shows the code section for data pre-processing. First, all characters in narratives are lowercased. Next, non-alphabetical characters (e.g., numbers and special characters such as question marks) are removed. Then, words that are commonly used for everyday (‘the,’ ‘and,’ ‘to,’ etc.) are deleted. The deleted words are called ‘stop-words.’ The list of stop-words are listed in a dictionary that comes with the installed natural language processing toolkit (‘nltk’). As some narratives are recorded in Spanish, the project also uses the Spanish stop-words list in the nltk package. Another step to clean the narrative text is to remove

[5] https://docs.python.org/3/library/codecs.html#standard-encodings

words that have no meaning such as ‘po,’ and ‘pi’ (used for embedding a question in the narrative). Users can expand the list of words to be removed by editing the items in ‘words_to_remove = { ‘po’ … }’ in the code section below. In addition, this project removes words with a single letter (length < 2).



```
def preprocess_data(X_data_raw):
    """
       Preprocess data with lowercase conversion, punctuation removal, tokenization, stemming, and tf-idf

       X_data_raw: X data in dataframe

    """

    X_data=X_data_raw.iloc[:, 1]

    # 1 convert all characters to lowercase
    X_data = X_data.map(lambda x: str(x).lower())

    # 2. remove non-alphabetical characters
    X_data = X_data.str.replace("[^a-zA-Z]", " ", regex=True)

    # remove stopwords in English
    stop_english = stopwords.words('english')
    X_data = X_data.apply(lambda x: ' '.join([word for word in x.split() if word not in (stop_english)]))

    # remove stopwords in Spanish
    stop_spanish = stopwords.words('spanish')
    X_data = X_data.apply(lambda x: ' '.join([word for word in x.split() if word not in (stop_spanish)]))

    # remove words 1: target
    words_to_remove = {'po', 'pi', 'ao', 'no'}
    X_data = X_data.apply(lambda x: ' '.join([word for word in x.split() if word not in words_to_remove]))

    # remove words 2: length
    X_data = X_data.apply(lambda x: ' '.join([word for word in x.split() if len(word) > 1]))

    # 3. word tokenize
    X_data = X_data.apply(nltk.word_tokenize)

    # 4. stemming
    stemmer = PorterStemmer()
    X_data = X_data.apply(lambda x: [stemmer.stem(y) for y in x])

    # ngram
    #X_data = X_data.apply(lambda x: list(nltk.ngrams(x, 2)))
    #X_data = X_data.apply(lambda x: [''.join(i) for i in x])
    #print (X_data[0, 1])

    # join by spaces
    X_data = X_data.apply(lambda x: ' '.join(x))

    return X_data
```

Figure 5: Code Section for Data Pre-processing

After the aforementioned pre-processing steps are completed, each narrative text is dissected into a list of word tokens using the nltk word tokenizer.[6] For example, a sentence ‘I owned a shop’ is transformed into a bag of words (‘owned’ and ‘shop’; ‘I’ and ‘a’ are deleted through the stop-words listing and single-character word removal). Next, the tokens are trimmed into shorter and simplified word forms. This process is called ‘stemming’ (e.g., ‘owned’ → ‘own’). This project

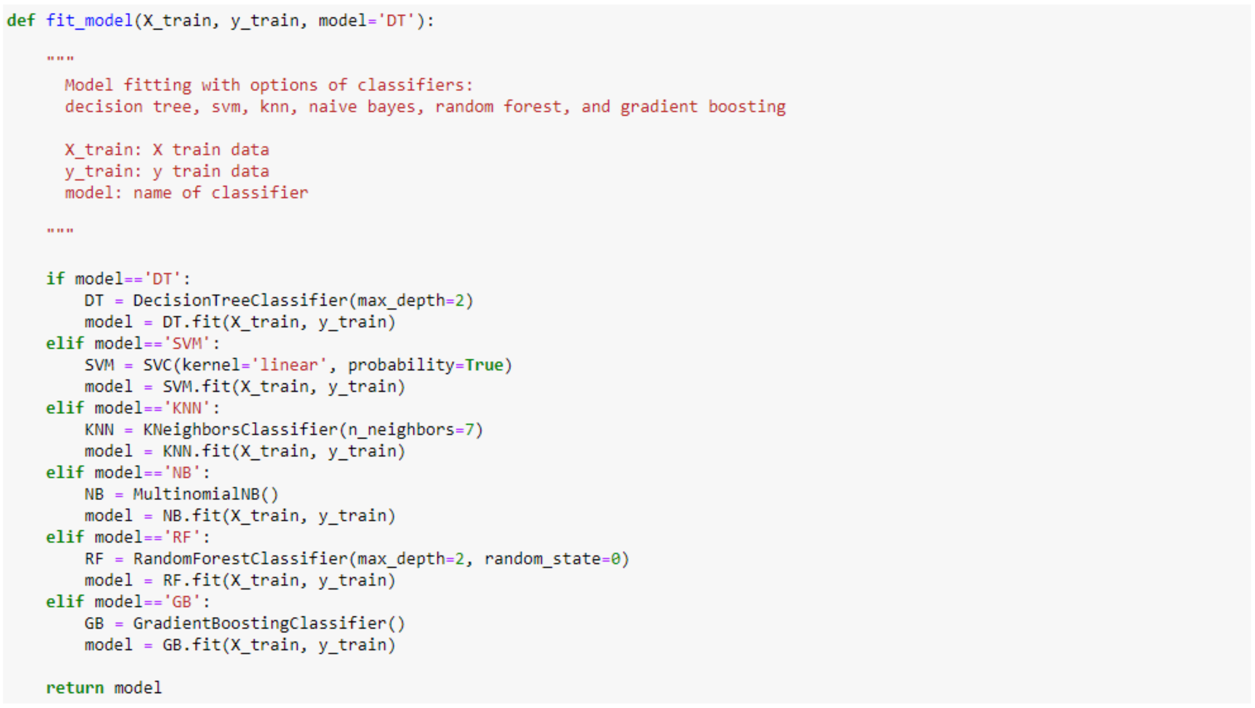
[6] For more information about word tokenizing, see https://www.guru99.com/tokenize-words-sentences-nltk.html

uses the most commonly used stemmer, ‘Porter’s Stemmer,’ as implemented in the nltk package.[7]

For future use, this code section contains a function to convert a narrative text into a list of *n*-gram tokens. For example, a sentence ‘I love you’ is converted into a list of word pairs (‘I love’ - ‘love you’) if 2-gram is chosen. Users can un-commentize the code part below ‘#ngram’ to include it in pre-processing.

Fitting Models

This stage trains machine learning algorithms to learn patterns of classification from training data. Figure 6 describes how the training is conducted. Various machine learning algorithms are included in the code so that users can try each to find the best performers. In the released code, a total of six algorithms are included: Decision Tree, Support Vector Machine, K-Nearest Neighbors, Naïve Bayes, Random Forests, and Gradient Boosting Trees. They have been widely used in text classification tasks as baselines or best performers.



```python
def fit_model(X_train, y_train, model='DT'):

    """
      Model fitting with options of classifiers:
      decision tree, svm, knn, naive bayes, random forest, and gradient boosting

      X_train: X train data
      y_train: y train data
      model: name of classifier

    """

    if model=='DT':
        DT = DecisionTreeClassifier(max_depth=2)
        model = DT.fit(X_train, y_train)
    elif model=='SVM':
        SVM = SVC(kernel='linear', probability=True)
        model = SVM.fit(X_train, y_train)
    elif model=='KNN':
        KNN = KNeighborsClassifier(n_neighbors=7)
        model = KNN.fit(X_train, y_train)
    elif model=='NB':
        NB = MultinomialNB()
        model = NB.fit(X_train, y_train)
    elif model=='RF':
        RF = RandomForestClassifier(max_depth=2, random_state=0)
        model = RF.fit(X_train, y_train)
    elif model=='GB':
        GB = GradientBoostingClassifier()
        model = GB.fit(X_train, y_train)

    return model
```

Figure 6: Code Section for Fitting Model

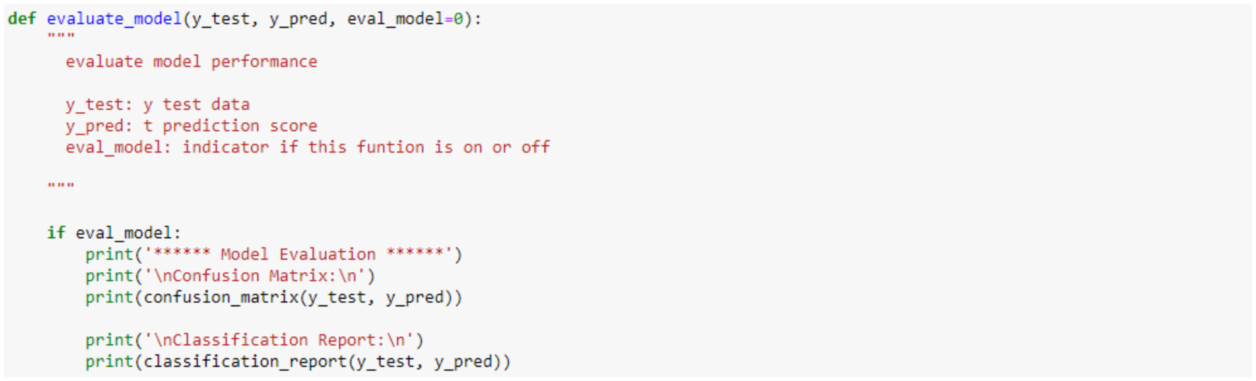
Each algorithm is implemented using its baseline hyper-parameters (e.g., max_depth in Random Forests) as described in the documentation for scikit-learn packages.[8] For more information

[7] https://tartarus.org/martin/PorterStemmer/
[8] https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html; https://scikit-learn.org/stable/user_guide.html

about the hyper-parameter options available for each algorithm, please refer to the documentation.

Evaluating Models

Given labeled data in which each instance is assigned a tag (label decided by human coders), the performance of machine learning models can be assessed by counting how many correct predictions are made by a model. In Figure 7, two evaluation methods are implemented: a confusion matrix and a classification report.[9] Both the Confusion Matrix and the Classification Report are generated by innate functions in the code.



```
def evaluate_model(y_test, y_pred, eval_model=0):
    """
      evaluate model performance

      y_test: y test data
      y_pred: t prediction score
      eval_model: indicator if this funtion is on or off

    """

    if eval_model:
        print('****** Model Evaluation ******')
        print('\nConfusion Matrix:\n')
        print(confusion_matrix(y_test, y_pred))

        print('\nClassification Report:\n')
        print(classification_report(y_test, y_pred))
```

Figure 7: Code Section for Model Evaluation

(1) Confusion Matrix: This visualizes the performance of a model in a table format. Table 1 illustrates how a confusion matrix is created.

Table 1: An Example of a Confusion Matrix

	Predicted A	Predicted B
True A	15	5
True B	3	7

In Table 1, two labels – A and B – are predicted for a total of 30 cases (= 15+ 5 + 3 + 7). Among 20 cases labeled A (true), 15 cases are correctly predicted as A while the remaining five are predicted as B. In contrast, three out of 10 B-labeled cases are predicted as A and seven of them are predicted as B. As such, the confusion matrix succinctly summarizes the prediction results of a model per label.

(2) Classification Report: The model performance can be quantified as precision and recall based on the confusion matrix. Recall measures how many cases are correctly

[9] For more detailed explanation about confusion matrix and classification report, please refer to https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

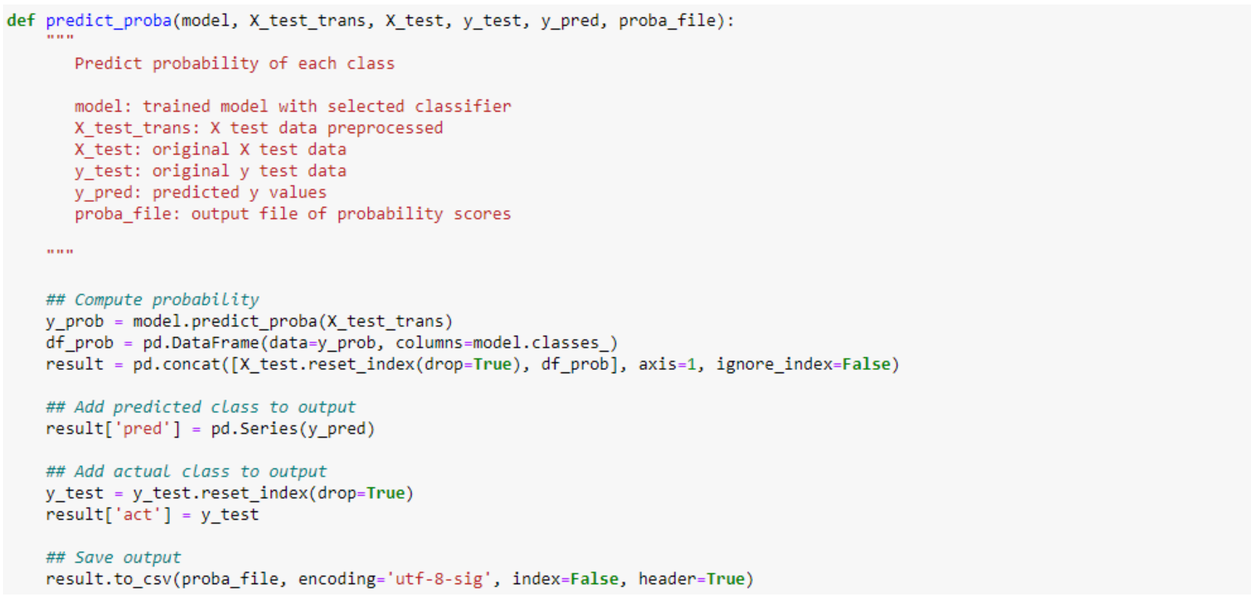
predicted by a model for their labels (= Predicted A among True A/True A = 15/20 or Predicted B among True B/True B = 7/10). Precision measures how many predicted cases are actually true (= True A among Predicted A/Predicted A = 15/18 or True B among Predicted B/Predicted B = 7/12). The f1 score is a harmonic mean of recall and precision (= 2*Recall*Precision/(Recall + Precision)) weighing both equally.

Predicting Probability

In classification tasks, machine algorithms learn patterns of classification based on training data and produce probability scores for instances in test data to refer to target classes. If the probability score is 0.5 or above for an instance to belong to a target class, the class is assigned to the instance, while if below 0.5, the class is not assigned (→ binary decision). This threshold (= 0.5) is set by default in most machine learning procedures.

Unlike those tasks, the code set used for this project is set to produce the probability scores used by machine learning algorithms to assign labels for target data. This allows users to modify the levels of probability scores that are decided to refer to a target class. For example, users can set the probability score for a target class as 0.75 or higher, increasing the accuracy of prediction results.

Figure 8 shows the procedure of producing probability scores for instances in test data. The output file is formatted as a CSV file with the utf8 encoding by default.

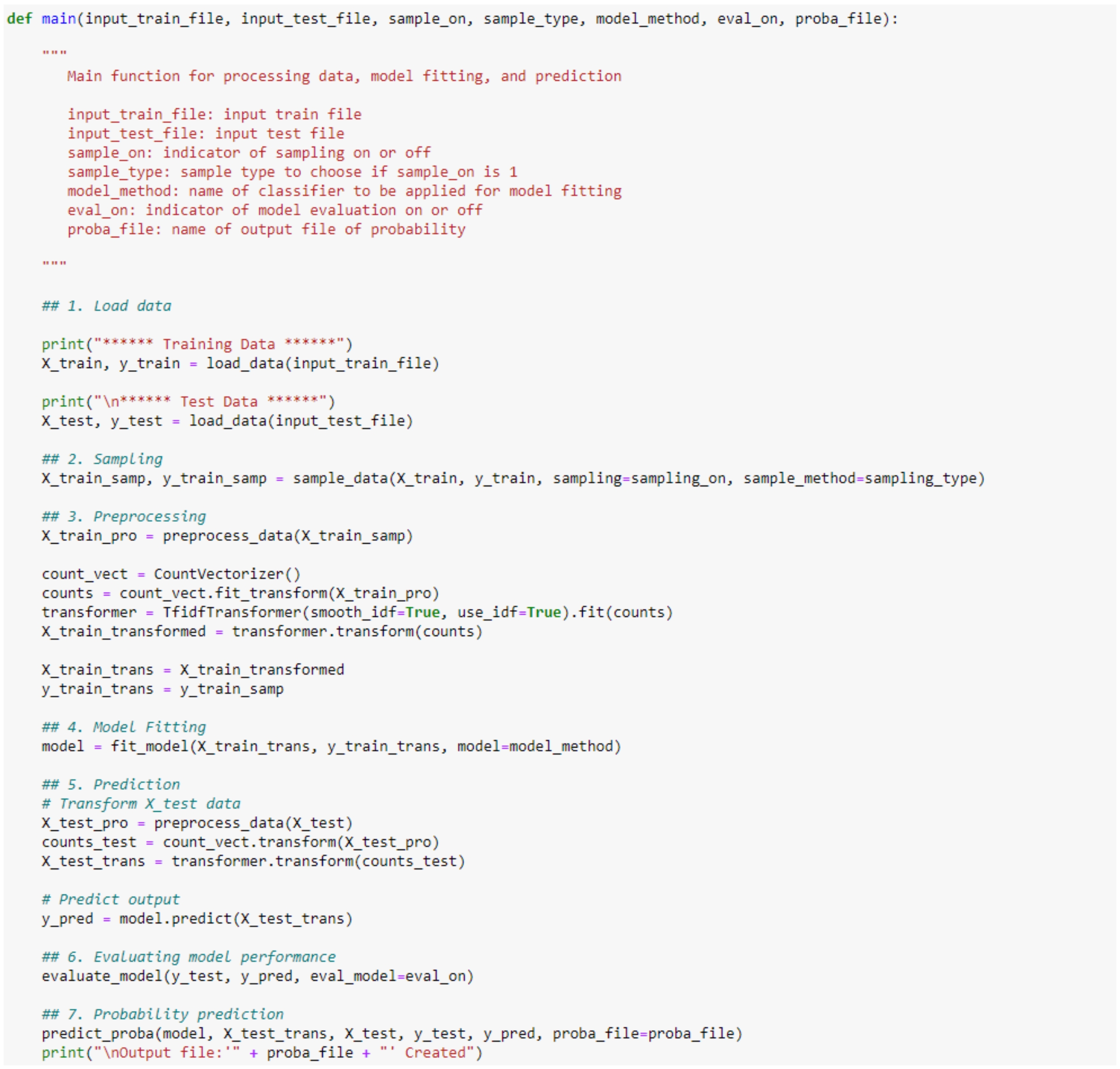


```
def predict_proba(model, X_test_trans, X_test, y_test, y_pred, proba_file):
    """
       Predict probability of each class

       model: trained model with selected classifier
       X_test_trans: X test data preprocessed
       X_test: original X test data
       y_test: original y test data
       y_pred: predicted y values
       proba_file: output file of probability scores

    """

    ## Compute probability
    y_prob = model.predict_proba(X_test_trans)
    df_prob = pd.DataFrame(data=y_prob, columns=model.classes_)
    result = pd.concat([X_test.reset_index(drop=True), df_prob], axis=1, ignore_index=False)

    ## Add predicted class to output
    result['pred'] = pd.Series(y_pred)

    ## Add actual class to output
    y_test = y_test.reset_index(drop=True)
    result['act'] = y_test

    ## Save output
    result.to_csv(proba_file, encoding='utf-8-sig', index=False, header=True)
```

Figure 8: Code Section for Predicting Probability Scores

Main Function

So far, several functions are defined for loading and sampling data, pre-processing features (variables), fitting models, and predicting probability scores. These steps need to be executed one by one to produce a final output. A main execution function as defined in Figure 9 controls the order of execution.



```python
def main(input_train_file, input_test_file, sample_on, sample_type, model_method, eval_on, proba_file):

    """
       Main function for processing data, model fitting, and prediction

       input_train_file: input train file
       input_test_file: input test file
       sample_on: indicator of sampling on or off
       sample_type: sample type to choose if sample_on is 1
       model_method: name of classifier to be applied for model fitting
       eval_on: indicator of model evaluation on or off
       proba_file: name of output file of probability

    """

    ## 1. Load data

    print("****** Training Data ******")
    X_train, y_train = load_data(input_train_file)

    print("\n****** Test Data ******")
    X_test, y_test = load_data(input_test_file)

    ## 2. Sampling
    X_train_samp, y_train_samp = sample_data(X_train, y_train, sampling=sampling_on, sample_method=sampling_type)

    ## 3. Preprocessing
    X_train_pro = preprocess_data(X_train_samp)

    count_vect = CountVectorizer()
    counts = count_vect.fit_transform(X_train_pro)
    transformer = TfidfTransformer(smooth_idf=True, use_idf=True).fit(counts)
    X_train_transformed = transformer.transform(counts)

    X_train_trans = X_train_transformed
    y_train_trans = y_train_samp

    ## 4. Model Fitting
    model = fit_model(X_train_trans, y_train_trans, model=model_method)

    ## 5. Prediction
    # Transform X_test data
    X_test_pro = preprocess_data(X_test)
    counts_test = count_vect.transform(X_test_pro)
    X_test_trans = transformer.transform(counts_test)

    # Predict output
    y_pred = model.predict(X_test_trans)

    ## 6. Evaluating model performance
    evaluate_model(y_test, y_pred, eval_model=eval_on)

    ## 7. Probability prediction
    predict_proba(model, X_test_trans, X_test, y_test, y_pred, proba_file=proba_file)
    print("\nOutput file:'" + proba_file + "' Created")
```

Figure 9: Code Section for Main Function

(1) Load Data: Training and test data are loaded to be pre-processed through various steps of natural language processing techniques.

(2) Sampling: If sampling is chosen (‘sample_on’), instances in training data are over- or under-sampled to create balanced labeled data.

(3) The narratives in pre-processed training data are converted into bags of words for the calculation of TF-IDF scores. TF-IDF is an acronym of Term Frequency – Inversed Document Frequency. This score counts first how often a token appears in a document and then divides it by how often the token appears across documents. In this way, a token is assigned a score that represents its importance in distinguishing a document that contains it from others. In our project, specifically, a token (e.g., ‘own’) can be helpful to

decide whether a narrative refers to a specific class (e.g., Owner) if it frequently (→ high TF) and almost uniquely (→ low DF) appears in the narratives of respondents who are labeled 'Owner.'

(4) Model Fitting: Machine learning algorithms learn that specific words are indicative of specific classes in training data through the distribution of TF-IDF scores of tokens in training data.

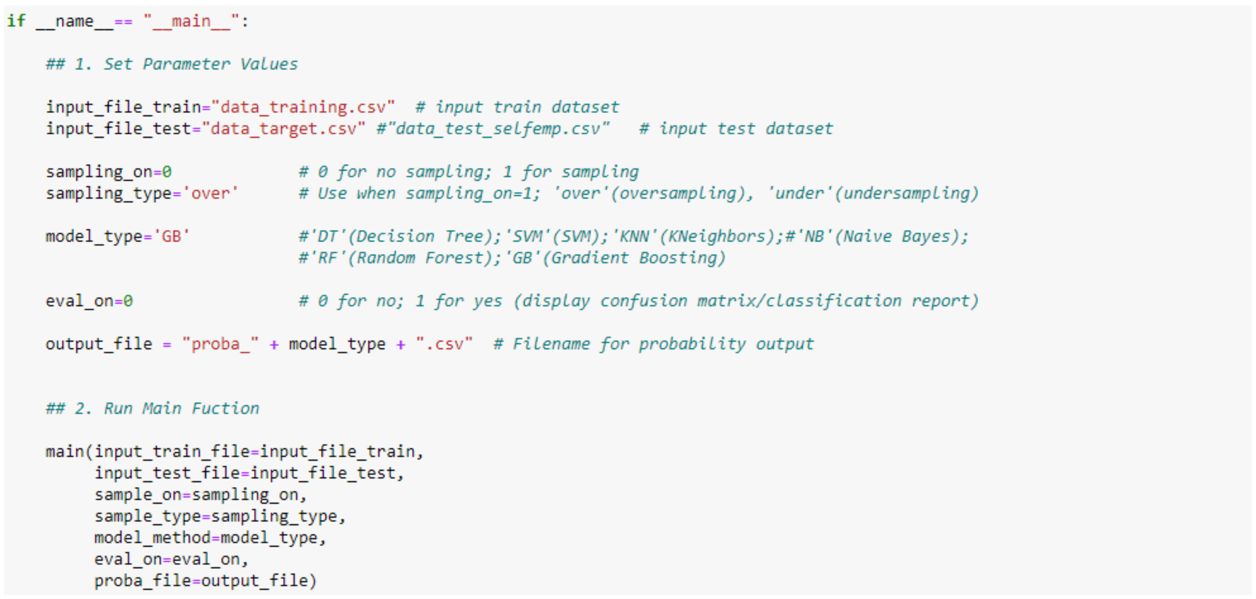
(5) Predict Output: Based on the models learned from model fitting, machine learning algorithms predict labels for instances in test data by looking into the TF-IDF scores of tokens in narratives of those instances.

(6) Evaluating Model Performance: If test data are provided with labels and the evaluation mode is chosen ('eval_on'), the code produces both a confusion matrix and a classification report that contain evaluation results of the machine learning models that produce probability scores for instances in test data.

(7) Probability Prediction: The code produces an output file in which instances in test data are assigned probability scores to belong to each of the target classes and a label assigned by the machine learning models.

Execution Command

In this section, users decide which dataset is used for training and data, whether over- or under-sampling is used, which machine learning algorithm is used, whether evaluation is conducted, and how the output file is named. Figure 10 shows the types of choices that need to be made by users and what options are available for each choice. The selected choices are reflected in running the main execution function explained above.



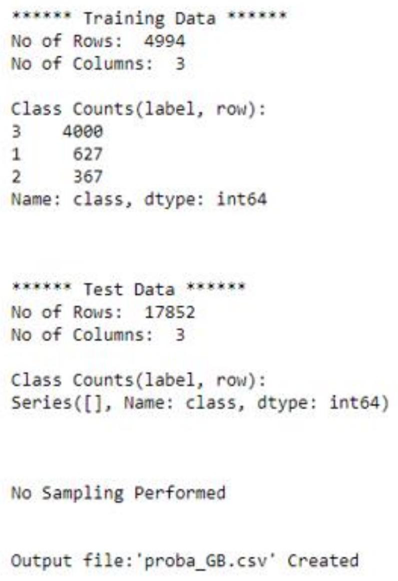
```
if __name__== "__main__":

    ## 1. Set Parameter Values

    input_file_train="data_training.csv"  # input train dataset
    input_file_test="data_target.csv" #"data_test_selfemp.csv"   # input test dataset

    sampling_on=0              # 0 for no sampling; 1 for sampling
    sampling_type='over'       # Use when sampling_on=1; 'over'(oversampling), 'under'(undersampling)

    model_type='GB'            #'DT'(Decision Tree);'SVM'(SVM);'KNN'(KNeighbors);#'NB'(Naive Bayes);
                               #'RF'(Random Forest);'GB'(Gradient Boosting)

    eval_on=0                  # 0 for no; 1 for yes (display confusion matrix/classification report)

    output_file = "proba_" + model_type + ".csv"  # Filename for probability output


    ## 2. Run Main Fuction

    main(input_train_file=input_file_train,
         input_test_file=input_file_test,
         sample_on=sampling_on,
         sample_type=sampling_type,
         model_method=model_type,
         eval_on=eval_on,
         proba_file=output_file)
```

Figure 10: Code Section for Execution Command

For this project, over- or under-sampling is not selected (‘sampling_on = 0’) as they are found not to improve the performance of algorithmic models. After trying several algorithms, this project uses the Gradient Boosting Trees algorithm for producing final outputs as it outperforms others in precision and overall f1 scores). The Gradient Boosting Trees algorithm outperformed the other models mainly because the algorithm works well in classification tasks for imbalanced data.

Output Screen

Once the code set is run on a command prompt, it produces a progress report as illustrated below in Figure 11. It shows (1) the numbers of rows and columns (for training and test data) and (2) class counts (for training data) in loaded data. It also reports whether sampling is chosen and whether an output file is created. Note that the numbers shown in the screenshot are different from those reported for this project.

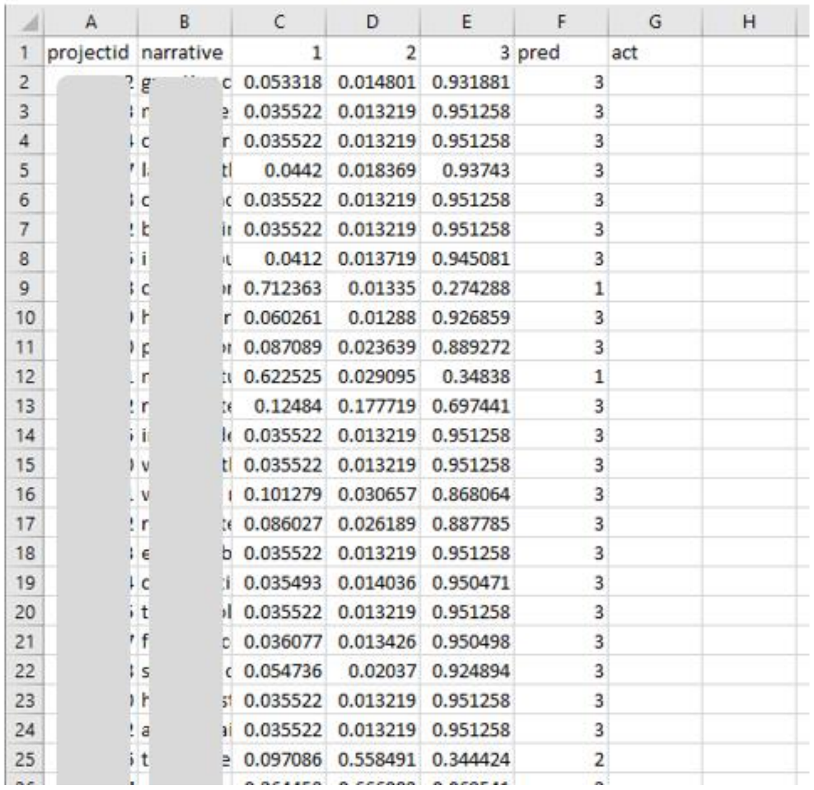


```
****** Training Data ******
No of Rows:  4994
No of Columns:  3

Class Counts(label, row):
3    4000
1     627
2     367
Name: class, dtype: int64


****** Test Data ******
No of Rows:  17852
No of Columns:  3

Class Counts(label, row):
Series([], Name: class, dtype: int64)


No Sampling Performed


Output file:'proba_GB.csv' Created
```

Figure 11: Example of Code Implementation Output Screen

Output File

Once the execution of the code set is completed, an output file with a CSV file extension is created and saved on the folder where the code set is located. Figure 12 shows part of the output file as an example.



	A	B	C	D	E	F	G	H
1	projectid	narrative	1	2	3	pred	act	
2	[redacted]	[redacted]	0.053318	0.014801	0.931881	3		
3	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
4	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
5	[redacted]	[redacted]	0.0442	0.018369	0.93743	3		
6	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
7	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
8	[redacted]	[redacted]	0.0412	0.013719	0.945081	3		
9	[redacted]	[redacted]	0.712363	0.01335	0.274288	1		
10	[redacted]	[redacted]	0.060261	0.01288	0.926859	3		
11	[redacted]	[redacted]	0.087089	0.023639	0.889272	3		
12	[redacted]	[redacted]	0.622525	0.029095	0.34838	1		
13	[redacted]	[redacted]	0.12484	0.177719	0.697441	3		
14	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
15	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
16	[redacted]	[redacted]	0.101279	0.030657	0.868064	3		
17	[redacted]	[redacted]	0.086027	0.026189	0.887785	3		
18	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
19	[redacted]	[redacted]	0.035493	0.014036	0.950471	3		
20	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
21	[redacted]	[redacted]	0.036077	0.013426	0.950498	3		
22	[redacted]	[redacted]	0.054736	0.02037	0.924894	3		
23	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
24	[redacted]	[redacted]	0.035522	0.013219	0.951258	3		
25	[redacted]	[redacted]	0.097086	0.558491	0.344424	2		

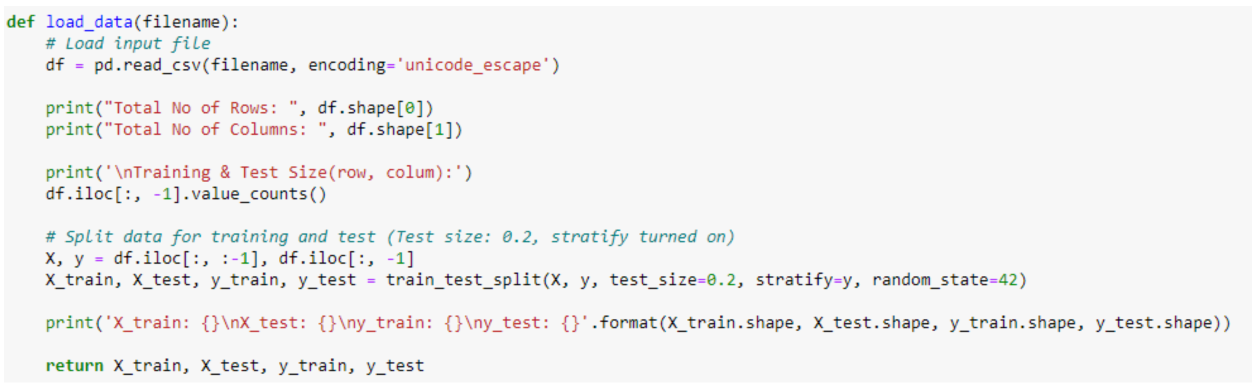
Figure 12: Example of Output File

In the file, an instance (row) is assigned probability scores to refer to each of three classes ('1,' '2,' and '3') and a label ('pred' column) decided by a machine learning model. As the code set run for this example case does not include labels for test data, actual labels ('act' column) are not shown. In the example file, the first instance (Row #2) is assigned '3' for a label because its probability score for the class '3' is 0.981881.

Using the probability scores assigned to instances, this project filters instances that are highly likely to refer to a specific class. Specifically, a total of 17,854 instances were assigned one of three tags – Independent, Manager, and Owner – along with probability scores of an instance belonging to each tag. Using the probability score, a total of 14,748 instances with a probability score of 0.9 or above for 'Independent' or with a probability score of 0.6 or above for 'Owner' were filtered as accurately tagged instances because they are highly likely to be assigned correct tags (on average 97.3% for Independent and 99.0% for Owner) when evaluated on 10 random subsets (20% of 4,500 instances) of hand-coded labeled data used for training models. The probability score thresholds are chosen by trying different probability scores and comparing accuracies of correct tag assignments. The remaining instances are assigned under manual inspection and correction by two human coders before the data are able to be used for analysis.

Evaluating Performance

One of the code sets (CNARS_training.py) has a function to be used to evaluate how well an algorithmic model performs in assigning classes to instances. Figure 13 shows the code section for loading data in the CNARS_traing.py file.

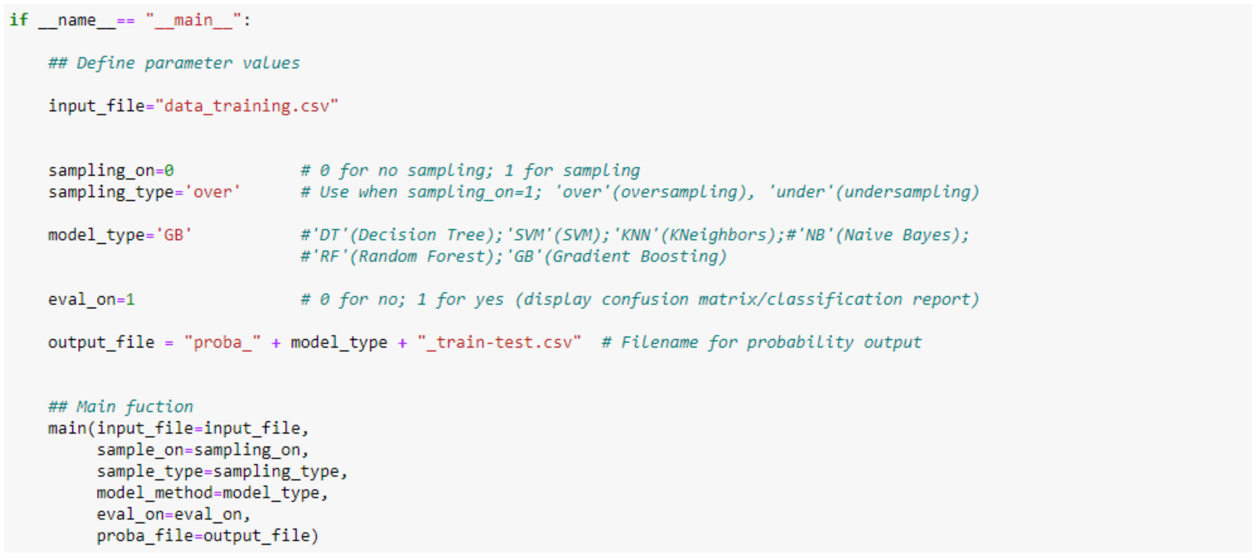


```
def load_data(filename):
    # Load input file
    df = pd.read_csv(filename, encoding='unicode_escape')

    print("Total No of Rows: ", df.shape[0])
    print("Total No of Columns: ", df.shape[1])

    print('\nTraining & Test Size(row, colum):')
    df.iloc[:, -1].value_counts()

    # Split data for training and test (Test size: 0.2, stratify turned on)
    X, y = df.iloc[:, :-1], df.iloc[:, -1]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

    print('X_train: {}\nX_test: {}\ny_train: {}\ny_test: {}'.format(X_train.shape, X_test.shape, y_train.shape, y_test.shape))

    return X_train, X_test, y_train, y_test
```

Figure 13: Code Section for Splitting Data in CNARS_training.py

Unlike the CNARS_main.py file in which separate training and test files are required, the load_data function here divides the loaded data into two subsets: training and test sets. The ratio between training and test sets can be set by users. Following the common practice in machine learning research, the test set size for this project is set to be 0.2 (‘test_size = 0.2’; 20% of the loaded data).

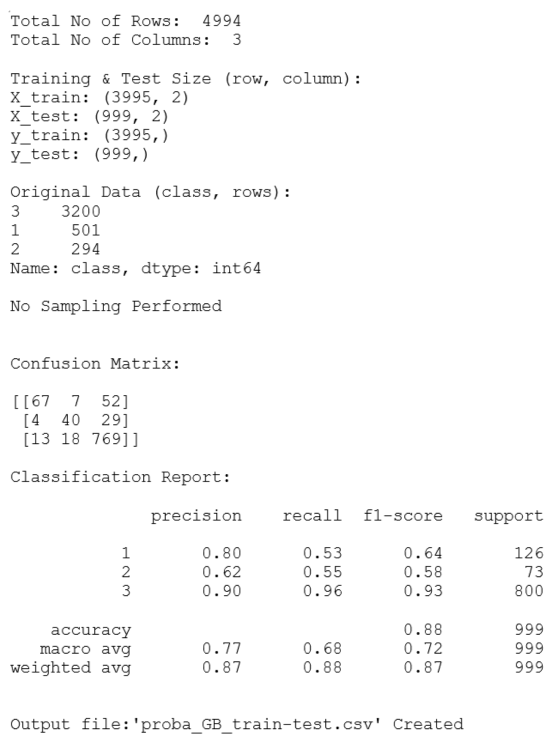
The Execution Command in CNARS_training.py is shown in Figure 14. Unlike the CNARS_main.py, it does not require a test file as a training file as the input file (input_file = “data_training.csv”) is split into training and test sets in the data loading step (→ Figure 13).



```
if __name__== "__main__":

    ## Define parameter values

    input_file="data_training.csv"


    sampling_on=0              # 0 for no sampling; 1 for sampling
    sampling_type='over'       # Use when sampling_on=1; 'over'(oversampling), 'under'(undersampling)

    model_type='GB'            #'DT'(Decision Tree);'SVM'(SVM);'KNN'(KNeighbors);#'NB'(Naive Bayes);
                               #'RF'(Random Forest);'GB'(Gradient Boosting)

    eval_on=1                  # 0 for no; 1 for yes (display confusion matrix/classification report)

    output_file = "proba_" + model_type + "_train-test.csv"  # Filename for probability output


    ## Main fuction
    main(input_file=input_file,
         sample_on=sampling_on,
         sample_type=sampling_type,
         model_method=model_type,
         eval_on=eval_on,
         proba_file=output_file)
```

Figure 14: Code Section for Execution Command in CNARS_training.py

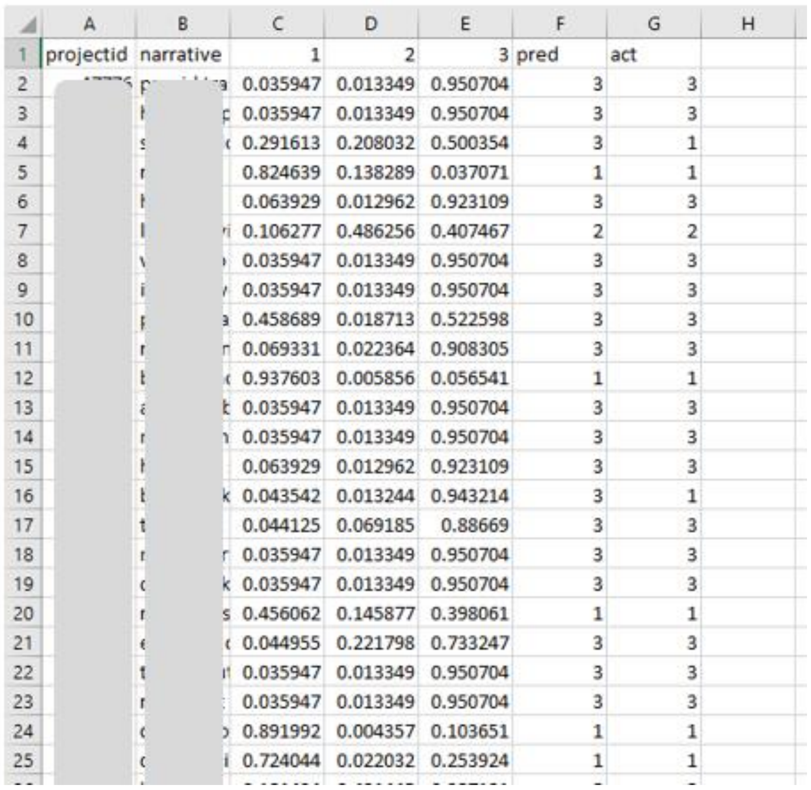
Note that the evaluation option is turned on ('eval_on' = 1). Once the Execution Command is run and completed, it produces an output screen as shown below in Figure 15. The screen reports both the confusion matrix and the classification report.



```
Total No of Rows:  4994
Total No of Columns:  3

Training & Test Size (row, column):
X_train: (3995, 2)
X_test: (999, 2)
y_train: (3995,)
y_test: (999,)

Original Data (class, rows):
3    3200
1     501
2     294
Name: class, dtype: int64

No Sampling Performed


Confusion Matrix:

[[67  7  52]
 [4  40  29]
 [13 18 769]]

Classification Report:

              precision    recall  f1-score   support

           1       0.80      0.53      0.64       126
           2       0.62      0.55      0.58        73
           3       0.90      0.96      0.93       800

    accuracy                           0.88       999
   macro avg       0.77      0.68      0.72       999
weighted avg       0.87      0.88      0.87       999


Output file:'proba_GB_train-test.csv' Created
```

Figure 15: Example of Code Implementation Output Screen for CNARS_training.py

The output file is different from that produced by running the CNARS_main.py file. As shown in Figure 16 below, it contains actual (= true) labels for each instance in a test set. For example, the instance in the fourth row is assigned '3' ('pred' column) by a machine learning algorithm because the probability score for the class '3' is highest (0.500354). But the label should be '1' ('act' column) according to the decision by human coders.



	A	B	C	D	E	F	G	H
1	projectid	narrative	1	2	3	pred	act	
2	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
3	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
4	[redacted]	[redacted]	0.291613	0.208032	0.500354	3	1	
5	[redacted]	[redacted]	0.824639	0.138289	0.037071	1	1	
6	[redacted]	[redacted]	0.063929	0.012962	0.923109	3	3	
7	[redacted]	[redacted]	0.106277	0.486256	0.407467	2	2	
8	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
9	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
10	[redacted]	[redacted]	0.458689	0.018713	0.522598	3	3	
11	[redacted]	[redacted]	0.069331	0.022364	0.908305	3	3	
12	[redacted]	[redacted]	0.937603	0.005856	0.056541	1	1	
13	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
14	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
15	[redacted]	[redacted]	0.063929	0.012962	0.923109	3	3	
16	[redacted]	[redacted]	0.043542	0.013244	0.943214	3	1	
17	[redacted]	[redacted]	0.044125	0.069185	0.88669	3	3	
18	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
19	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
20	[redacted]	[redacted]	0.456062	0.145877	0.398061	1	1	
21	[redacted]	[redacted]	0.044955	0.221798	0.733247	3	3	
22	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
23	[redacted]	[redacted]	0.035947	0.013349	0.950704	3	3	
24	[redacted]	[redacted]	0.891992	0.004357	0.103651	1	1	
25	[redacted]	[redacted]	0.724044	0.022032	0.253924	1	1	

Figure 16: Figure 12: Example of Output File by CNARS_taining.py

Code Maintenance and Update

The code set will be deposited on two code repositories (GitHub and FigShare) that are accessible without restrictions. Acknowledgements of the project sponsor (MRDRC) with the project number (UM21-14) and title ("What We Talk about When We Talk about Self-Employment: Examining Self-Employment and the Transition to Retirement among Older Adults in the United States") will appear in the released code set. Users can report issues and request modifications by leaving comments on GitHub. The code set will be maintained and updated on a regular basis (every other month) for a year (until September 2022) and intermittently after that. Any technical questions or suggestions for the code set will be directed to Dr. Jinseok Kim (jinseokk@umich.edu).

References

Berry, Michael W., & Castellanos, Malu (2004). Survey of text mining. Computing Reviews, 45(9), 548.

Boselli, Roberto, Cesarini, Mirko, Mercorio, Fabio, & Mezzanzanica, Mario (2018). Classifying online job advertisements through machine learning. Future Generation Computer Systems, 86, 319-328.

Ikudo, Akina, Lane, Julia I., Staudt, Joseph, & Weinberg, Bruce A. (2019). Occupational classifications: A machine learning approach. Journal of Economic and Social Measurement, 44(2-3), 57-87.

Lampos, Vasileios, Aletras, Nikolaos, Geyti, Jens K., Zou, Bin, & Cox, Ingemar J. (2016). Inferring the socioeconomic status of social media users based on behaviour and language. Paper presented at the European conference on information retrieval.

Mac Kim, Sunghwan, Xu, Qiongkai, Qu, Lizhen, Wan, Stephen, & Paris, Cécile. (2017). Demographic inference on twitter using recursive neural networks. Paper presented at the Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers).

Preoţiuc-Pietro, Daniel, Lampos, Vasileios, & Aletras, Nikolaos. (2015). An analysis of the user occupational class through Twitter content. Paper presented at the Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).

Preoţiuc-Pietro, Daniel, & Ungar, Lyle. (2018). User-level race and ethnicity predictors from twitter text. Paper presented at the Proceedings of the 27th International Conference on Computational Linguistics.