Machine Learning **Emanuele Giona**
A.Y. 2019/2020

Homework 1: Compiler provenance

1 Introduction

In the scope of malware analysis, and cybersecurity in general, being able to reverse engineer a malicious script heavily depends on recognizing which compiler and compiler settings (such as the optimization level) used to produce the aforementioned script.
This task is commonly referred to as the compiler provenance problem: given a function as a list of low-level instructions, the goal consists in determining which compiler and optimization level have been used during the compilation phase.
This problem can be encoded as two distinct classification tasks:

- target function for the compiler sub-task:
$$f : X \rightarrow C = \{c_1, ..., c_k\} \tag{1}$$
with $\forall i \in [1, k]$ c_i refers to a compiler label;
- target function for the optimization level sub-task:
$$g : X \rightarrow O = \{o_1, ..., o_j\} \tag{2}$$
with $\forall i \in [1, j]$ o_i refers to an optimization level label;

while the dataset can be formalized as per the following equation:
$$D = \{(x_i, c_i, o_i)_{i=1}^{N}\} \tag{3}$$
with $\forall i \in [1, N]$ $x_i \in X,\ c_i \in C,\ o_i \in O$; X represents to the input space for samples (same meaning in Equations 1 and 2), C represents the set of compiler labels as defined in Equation 1, and O represents the set of optimization level labels as defined in Equation 2.

2 Dataset

The provided dataset is composed of 30 000 functions, being produced by three compilers (namely `gcc`, `icc`, and `clang`) and two optimization levels (low - `L` - and high - `H`).
The dataset is balanced with respect to compilers: for each compiler there are 10 000 functions each; however, it is not balanced with respect to the optimization level: indeed, `L`-labelled functions are way more common than the `H`-labelled ones, with 17 924 and 12 076 occurrences respectively[1].
The dataset doesn't appear to be subject to noisy or malformed data, as all the required features are present, with no missing labels, and therefore it doesn't require further processing in this direction.

[1]More statistics about the dataset can be obtained via the `stats()` function implemented in the `preprocess.py` file

3 Pre-processing and feature engineering

First of all, the dataset is provided in a JSONL-formatted file: it is a textual file containing new-line separated JSON objects in the form of strings; a first pre-processing step[2] reads the dataset file line by line, creating Function objects[3] holding the following fields:

- instructions: list of instructions, obtained by blankspace-splitting the JSON object `instructions` field, in which the first component is the mnemonic and the rest of the list is composed by its arguments;
- compiler: compiler label, obtained by copying the JSON object `compiler` field; it can be `None` in case the dataset refers to a blind test set;
- optimization: optimization level label, obtained by copying the JSON object `opt` field; it can be `None` in case the dataset refers to a blind test set.

As a second phase, the following dictionaries are built from the training set:

- mnemonics dictionary: each mnemonic is mapped to an integer value;
- compilers dictionary: each compiler is mapped to an integer value; this dictionary also serves as output dictionary in the prediction phase;
- optimization levels dictionary: each optimization level is mapped to an integer value; this dictionary also serves as output dictionary in the prediction phase.

These dictionaries will be used to encode a sample (either coming from the training set or the test set) in the input space and the appropriate output space given which sub-task is being considered (as defined in Equations 1 and 2), and during the prediction phase in order to produce an output file as specified in the homework requirements.
Additionally, each dictionary is initialized with a special token to prevent the Out-of-Vocabulary (OOV) problem, in case functions in the test set present mnemonics never seen before.

The feature engineering phase considers two different representations for samples: a simple Bag-of-Words (BoW) model in which dimensions in the vector representation of a sample correspond to mnemonics; another vector representation instead relies on the concept of *ngrams*, which is still a BoW model, but mnemonics in a function are considered using a sliding window of size n, in order to retain the order of the instructions.
Both representations also consider the number of instructions contained in a given function as an additional feature.

3.1 BoW model

Assuming M being the size of the input dictionary containing all the different mnemonics[4] found in the training set and the special token for unknown mnemonics; considering the additional feature representing the number of instructions of which a given function consists of, a sample in the dataset D can be seen as a vector of size $M + 1$, over integer values.
For each dimension among the first M ones, the vector representation of a sample contains the number of occurrences of the mnemonic encoded with the integer value corresponding to that dimension.
In our particular case, the training set contains around 400 different mnemonics.

[2]See `take()` function in the `preprocess.py` file

[3]Actually not full Python objects, but instances of `namedtuple` for a more lightweight operation

[4]See `build_dictionaries()` function in the `preprocess.py` file

3.2 *ngrams* model

The *ngrams* model is able to retain information about instruction ordering in a function by considering mnemonics with a sliding window of size n[5]. This allows being able to encode a function via the number of occurrences of sequences of n mnemonics.
The *ngrams* dictionary is still built out of the training set, and considering that increasing the size n of the sliding window greatly increases the dictionary size, in our particular case n has been chosen equal to 2, resulting in around 10 000 bi-grams.
The same considerations applied in the BoW model are applied for the *ngrams* model as well, which, in this case, the special token for unknown bi-grams results being much more useful than the first case.

4 Models

For solving this homework, the models that have been used are the following: Support Vector Machines (SVM) with linear kernel, and K-Nearest Neighbors (KNN); both models are implemented in the SciKit-Learn Python library[6].

4.1 SVM with linear kernel

A Support Vector Machines model is a supervised learning technique that approximates the target function using a linear function of the input sample with a learned function of the shape of:

$$y(x) = w^T \cdot x \tag{4}$$

with w being a coefficient vector and x a new instance of the input space.

The particularity of SVM models is that the coefficient vector w is learned in such a way to maximize the margins between the linear solution and the support vectors; support vectors are a subset of the training set, representing instances of classes which are the closest to the linear solution.
The linear kernel implies that there is no other transformation applied to samples either at learning or prediction times.
In the SciKit-Learn package, the classification-specific model of SVM with linear kernel is implemented by the `LinearSVC` class[7].

4.2 KNN

A K-Nearest Neighbors model is a supervised learning technique that does not attempt at approximating the target function, it instead exploits the locality that can be found inside the training set.
A new instance x can be classified with the class achieving the highest probability, defining the probability of each class as per the following equation:

$$p(c|x, D, K) = \frac{1}{K} \sum_{i \in N_k(x,D)} \chi(t_i = c) \tag{5}$$

[5]See `build_ngrams()` function in the `preprocess.py` file
[6]URL: https://scikit-learn.org/
[7]See `get_SVM_model()` function in the `models.py` file

with $N_k(x, D)$ being the K *nearest* neighbors of the new instance x given the training set D and the function χ defined as:

$$\chi(e) = \begin{cases} 1 & if\ e\ is\ True \\ 0 & otherwise \end{cases} \tag{6}$$

KNN allows to weight contributions of the K neighbors either uniformly or based on the inverse of distance between them and the query sample, therefore closest neighbors affect the prediction in a greater way than distant ones.
In the SciKit-Learn package, the classification-specific model of K-Nearest Neighbors is implemented by the `KNeighborsClassifier` class[8].

5 Parameter tuning

The parameter tuning phase has been carried out on a reduced training set: out of the 30 000 functions present in the original training set, the reduced one has been defined as the first 2 500 functions per compiler.
For the `LinearSVC` model, the only parameter to tune has been `C`: the coefficient for balancing the importance of the error function with respect to the regularization term. The following values have been explored:

Parameter	**Set of values**
C	1.0, 1.5, 2.0, 3.0

Table 1: Parameter tuning for the `LinearSVC` model

For the `KNeighborsClassifier` model, the `n_neighbors` (referred to as K in Equation 5) and `weights` parameters have been tuned: they represent the number of neighbors being considered during the prediction phase, and whether using the distance between each neighbor and the query sample to weight each neighbor's contribution, respectively. The following values have been explored:

Parameter	**Set of values**
K	3, 5
weight	uniform, distance

Table 2: Parameter tuning for the `KNeighborsClassifier` model

Both models have undergone the parameter tuning phase via the `GridSearchCV` class provided by SciKit-Learn[9]: this class performs a grid parameter search with the given set of values on a given model, internally performing k-fold cross-validation for more stable performance evaluation. As shown by the `cross_validation()` function in the `train.py` file, cross-validation has been applied with 5 folds.
The parameter tuning phase has also been explored for both representations described in Section 3.

The best performing model regarding compiler classification has been `LinearSVC` with `C` equal

[8]See `get_KNN_model()` function in the `models.py` file
[9]See `cross_validation()` function in the `train.py` file

to 1.5, achieving 77.8% accuracy when using the bi-gram representation.
The best performing model regarding optimization level classification has been `LinearSVC` with `C` equal to 1.5 as well, achieving 76.9% accuracy when using the bi-gram representation.

The bi-gram representation has consistently yielded the best performances: when relying on the simple BoW representation, the best performing models achieved 58.1% and 70.7% accuracies in the compiler classification and the optimization level classification tasks, respectively.

Therefore, the training phase has been carried out the `LinearSVC` model using the bi-gram feature representation and optimal value 1.5 for the `C` parameter.

6 Performance evaluation

The best performing model has undergone the same kind of 5-fold cross-validation phase[10] for both the classification tasks on the whole training set, achieving 82.2% accuracy on compiler classification and 74.2% accuracy on optimization level classification.

7 Prediction phase

Due to the special token to prevent the OOV problem, as described in Section 3, during the prediction phase, an additional concern has been put on the outputs of the model. In fact, in case either the unknown compiler or unknown optimization level would have been predicted, the model applies a very simple fallback strategy: randomly pick a *real* label, excluding the special token[11].
While this may not hurt the compiler classification task, due to the balanced proportions of samples per compiler, it may negatively affect the optimization level classification task. A further development direction might be applying a sampling technique based on frequencies found in the training set.

[10]See `linearSVC_best_tuning_CV()` function in the `train.py` file
[11]See `predict()` function in the `predict.py` file