Debug Information Testing with Machine Learning

Borghini Alessia

November 2021

Abstract

Debug Information Testing is the task of checking whether two instructions, the former expressed in machine language and the latter through a programming language, correspond. In this project, I analyze different Machine Learning methods to solve the task, and compare them by means of the accuracy score.

1 Introduction

Compilers are among the most important software, they are used to generate executable code in machine language from source code in a programming language (e.g. C, C++). Additionally, compilers such as GCC or LLVM also provide an *optimization system* – in order to obtain a code faster to execute and that uses less resources – and a *debugger*, to give developer support to find errors.

Compilers are human-written, so they can be affected by bugs. Bugs can lead to security-sensitive problems for applications and to a more time-consuming process for developers to find coding bugs. Since the compiler software is large and complex, a ML approach can be used to automatically discover incorrect debug information included in optimized binaries.

2 Data Preprocessing

The dataset through which the ML models will be trained consists of a pandas dataframe containing 100k samples. For each sample the following information is provided: (1) **source line** in C language, (2) **assembly instructions** potentially corresponding to the source line, (3) a **binary label** which indicates if there are bugs. The distribution of samples is balanced, indeed, the number of source codes correctly mapped to the corresponding assembly instructions is comparable with the wrongly mapped ones. A blind test set is also provided, i.e. unlabeled, to evaluate the performance of a ML system for the task.

To train the model it's necessary to preprocess the data in order to obtain a vectorized version of them. I start by importing the necessary libraries and loading the dataset, and then applying different feature extraction methods.

2.1 Import Libraries

The imported libraries have different purpose: some are for data processing and to extract features, other to calculate the performance and visualize them. Others, instead, are already-implemented models [2].

```
# utils
import os
from google.colab import drive
from tqdm import tqdm
import time

# process data
import pandas as pd

# plot
```

```
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import graphviz

# counter class
from collections import Counter

# feature extraction
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# metrics
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import PrecisionRecallDisplay
from sklearn.metrics import accuracy_score
from sklearn.metrics import plot_roc_curve

# dataset splitter
from sklearn.model_selection import train_test_split

# models
from sklearn import svm, tree
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier

# preprocessing
from sklearn.preprocessing import StandardScaler
```

2.2 Load Data

After downloading the dataset on Google Drive, the Drive can be mounted on Colab and it's possible to access the dataset folder, in the following way:

```
drive.mount('/content/drive')

path = "drive/My Drive/Homework_dataset"
os.chdir(path)
```

Listing 1: Drive mounting

Then, using the python class `MyDataset` shown in Listing 2.2, the dataset can be loaded and preprocessed. The class takes as parameters the dataset file name, a boolean value which indicates whether the dataset is blind and other feature extraction parameters that will be explained in Section 2.3.

```
class MyDataset(Dataset):
    def __init__(self,
                 data_file:str,
                 words_vocabulary:dict,
                 max_length_instructions=27,
```

```
                max_length_source_lines=37,
                blind=False,
                vectorizer:str = "MyBoW") -> None:
```

Listing 2: MyDataset class definition and instantiation function heading

Using the `__load_data__` function of the class `MyDataset` I read the dataset and I create a DataFrame with `pandas` functions.

```
def __load_data__(self):
    reader = pd.read_csv(self.data_file, delimiter="\t")
    df = pd.DataFrame(reader, columns=["instructions", "source_line", "bug"])

    if not self.blind:
        labels = df.bug
    else:
        labels = []

    return df.instructions, df.source_line, labels
```

Listing 3: Load dataset function from MyDataset class

2.3 Bag-of-words

The samples in the dataset described in Section 2 are composed by two sequence of characters with variable length. However, these raw data can not be fed directly to the Machine Learning models which expect numerical feature vectors with a fixed size.

So, once the dataset is in the form of a data frame, it can be vectorized. The goal is to create for each sample a vector containing features that characterize the sample given.

In particular, there are some scikit learn Bag-of-Words models already implemented. A Bag-of-Words model describes texts by word occurrences while completely ignoring the relative position information of the words in the text. Additionally, I have implemented another simple Bag-of-words model. It's possible to choose the vectorizer to use setting the parameter `vectorizer_name` in Listing 2.2.

The scikit learn models are:

- **Tfidf Vectorizer**: which creates inverse document-frequency vectors, emphasizing the rare words;
- **Count Vectorizer**: which creates token counts vectors;
- **Hashing Vectorizer**: which convert each string token to an integer feature index.

First of all, the vectorizer must be chosen and each row of the dataset must be represented by the concatenation of the `instructions` and the `source_line`. Then, the vectorizer will be initialized and it will learn the vocabulary dictionary returning a vectorized version of the data. For example, in Listing 2.3 is shown the vectorization done using the Tfidf Vecotrizer.

```
if self.vectorizer_name == "TfidfVectorizer":
    self.vectorizer = TfidfVectorizer()
        ...
self.X = self.vectorizer.fit_transform(self.data)
```

Listing 4: Vectorizing the dataset

Note that in some experiments the parameter `ngram_range` is set `(1,2)` – this means that both unigram

and bigram are considered features – and `min_df` is set to 200, i.e. are considered only the tokens which appears in the dataset at least 200 times. For the Hashing Vectorizer also the following parameters are set: `decode_error="ignore", n_features=2 ** 12, alternate_sign=False`.

With the other Bag-of-Words model, MyBoW, every word correspond to an integer index which indicates the ranking position of the number of occurrences in the entire dataset of the word itself. The ranking is made by a sorted counter, where the key is the word and the value is the corresponding number of occurrences. Using the parameter `min_freq` it is possible to specify the minimum number of times that the words must be found in the dataset to become a feature. After that, is created a vector for every sample composed from `max_length_instructions` elements representing the instructions and `max_length_source_line` elements representing the source line. These two values have been chosen after an analysis of the average length of the samples and some trial. A zoom of the obtained histogram of the frequencies of the length of the source lines is reported in Figure 1. Considering the average length there will be some samples with a shorter length for which padding is needed.

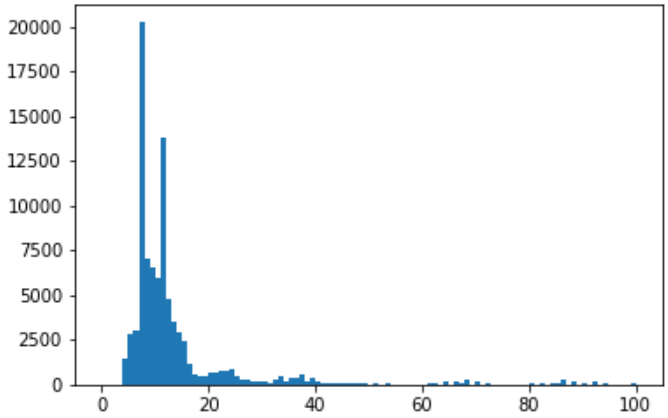

Figure 1: Histogram of the frequency of the length of the source lines up to length 100.

After the words vocabulary has been created, the function `__vectorize_data_MyBoW__` of the class MyDataset is used to generate the vectors, as shown in the Listing 5. In particular, this function maps the tokens in the sample to the indices of the words vocabulary for creating two vectors. Then, the vectors are padded, truncated and concatenated to obtain all the resulting vector of the same desired length.

```
def __vectorize_data_MyBoW__(self):
    X = []

    for (ins, sl) in tqdm(zip(self.instructions, self.source_lines)):

        instruction = ins.strip().split()
        source_line = sl.strip().split()

        temp_instructions = []
        temp_source_lines = []

        # take the index that corresponds to each word from the words vocabulary
        for elem in instruction:
            elem = elem.replace('"', "")
            if elem in self.words_vocabulary:
                temp_instructions.append(self.words_vocabulary[elem])
            else:
                temp_instructions.append(self.words_vocabulary["<unk>"])
```

```
        for elem in source_line:
            elem = elem.replace('"', "")
            if elem in self.words_vocabulary:
                temp_source_lines.append(self.words_vocabulary[elem])
            else:
                temp_source_lines.append(self.words_vocabulary["<unk>"])

        # padding vectors
        temp_instructions += [0]*(self.max_lenght_instructions -
        ↪  len(temp_instructions))
        temp_source_lines += [0]*(self.max_lenght_source_lines -
        ↪  len(temp_source_lines))

        # trunc vectors
        temp_instructions = temp_instructions[:self.max_lenght_instructions]
        temp_source_lines = temp_source_lines[:self.max_lenght_source_lines]

        X.append(temp_instructions + temp_source_lines)

    return X
```

Listing 5: MyBoW Vectorizer.

2.4 Split Dataset

Lastly, the dataset must be splitted into training and a testing set if the dataset is not the blind one. When the `random_state` parameter is set the resulting dataset will be always the same, by this way all the results are based on the same data and reproducible. The size of the test set is 20% of the total. The following code shows the scikit learn function that splits the dataset obtaining 20000 samples for the test and 80000 for the training.

```
if not self.blind:
    self.X_train, self.X_test, self.labels_train, self.labels_test =
    ↪  train_test_split(self.X, self.labels, test_size=0.2, random_state=0)
else:
    self.X_test = self.X
```

Listing 6: Split dataset if not blind.

3 Models

In this Section will be made a comparison between different Machine Learning models [1, 3], those shown in the Listing 3. For each model will be reported the configuration which obtain the best results. For this task will be considered the accuracy score because the dataset is balanced, indeed when the parameter `class_weight` exists is set to `"balanced"`. Furthermore, in order to make the results reproducible the parameter `random_state` is set to `0`.

```
models = {
    "tree": tree.DecisionTreeClassifier(min_samples_split=12, random_state=0,
    ↪  class_weight="balanced"),
    "forest": RandomForestClassifier(n_estimators = 100, min_samples_split=10,
    ↪  n_jobs=-1, verbose=1, random_state=0, class_weight="balanced"),
    "svm": svm.SVC(max_iter = 10000),
    "linear-svm": svm.LinearSVC(max_iter = 10000, random_state=0, class_weight =
    ↪  "balanced"),
    "gaussian": GaussianNB(),
```

```
    "bernoulli": BernoulliNB(),
    "multinomial": MultinomialNB(alpha=0.1),
    "kneighbors": KNeighborsClassifier(n_neighbors=5, n_jobs=-1),
    "logistic-regression": LogisticRegression(max_iter=10000, verbose = 1, n_jobs =
    ↪  -1, penalty="elasticnet", solver="saga", l1_ratio=0.5, C=1e5, random_state=0,
    ↪  class_weight="balanced"),
    "perceptron": Perceptron(early_stopping = True, verbose=3, max_iter=200,
    ↪  n_iter_no_change=5, random_state=0, class_weight="balanced"),
    "mlp": MLPClassifier(hidden_layer_sizes = (1), early_stopping = True, max_iter =
    ↪  200, verbose=3, n_iter_no_change=5, random_state=0, class_weight="balanced")
}
```

Listing 7: List of models to choosen from.

After choosing the model, it will be trained in the following way and then it's possible to compute the performances as shown in Listing 3.6. For some experiment has been used the `StandardScaler()` tool in order to standardize the features before doing the training and for other has been used `toarray()` method to convert the sparse matrix to an array. For all of this scikit learn functions are used and for further detail consult the Colab Notebook.

```
model.fit(dataset.X_train, dataset.labels_train)

y_pred = model.predict(dataset.X_test)

print(classification_report(dataset.labels_test, y_pred, labels=None, target_names =
↪  ["correct", "bugged"], digits=4))
```

Listing 8: Training and testing the model.

Now, let's talk about the models in more detail.

3.1 Naive Bayes Classifiers

The Naive Bayes Classifiers are Machine Learning models based on the Bayes theorem. That Classifiers compute the probabilities that a sample belongs to each class and choose the higher one. There are three main models:

- **Gaussian Naive Bayes**: whose parameters used to predict are taken from a gaussian distributions;
- **Bernoulli Naive Bayes**: which check whether some words are in the considered sample;
- **Multinomial Naive Bayes**: which is focused on the frequencies of the words present in the sample. This works better with integer feature counts. In the experiments I fine tuned the smoothing parameter obtaining the best results setting `alpha` to `0.1`.

The best result obtained using the **Gaussian Naive Bayes** is when is combined with **Tfidf Vectorizer**, where `ngram_range=(1,2)` and `min_df=100` (this reduces the dimension of the vector and make the system generalize better). Using **Bernoulli Naive Bayes** and **Multinomial**, instead, the best combination in with the **Count Vectorizer** model where `ngram_range` is set to `(1,2)`. The performances are all reported in Table 1.

	GaussianNB			BernoulliNB			MulinomialNB			
	precision	recall	f1-score	precision	recall	f1-score	precision	recall	f1-score	support
correct	0.634382	0.636043	0.635211	0.665977	0.864169	0.752238	0.605994	0.857560	0.710156	9927
bugged	0.640390	0.638737	0.639563	0.803054	0.560990	0.660544	0.762433	0.450511	0.566365	10073
accuracy			0.637400			0.713550			0.652550	20000
macro avg	0.717356	0.697173	0.689172	0.734516	0.712580	0.706391	0.684213	0.654036	0.638261	20000
weighted avg	0.637408	0.637400	0.637403	0.734077	0.713550	0.706684	0.684784	0.652550	0.637736	20000

Table 1: Naive Bayes Classifiers performances.

3.2 Logistic Regression

The **Logistic Regression Binary** model is used when the labels classify a sample as belonging to a category or not, like in this case where each sample can contains bugs or not. In Logistic Regression is applied a linear function to the features of the sample, then the result is compressed using the sigmoid function and finally the threshold value decides at which class the sample belongs to.

The higher performances are reported in Table 2 and are obtained with **Tfidf Vectorizer**.

	precision	recall	f1-score	support
correct	0.784535	0.832981	0.808032	9927.00000
bugged	0.824736	0.774546	0.798853	10073.00000
accuracy	0.803550	0.803550	0.803550	0.80355
macro avg	0.804635	0.803763	0.803443	20000.00000
weighted avg	0.804782	0.803550	0.803409	20000.00000

Table 2: Logistic Regression

3.3 Support Vector Machine

The **Support Vector Machine (SVM)** model is a powerful algorithm for learning complex non linear functions. It creates a decision boundary, i.e. an hyperplane if the number of features is greater than 2, which separates data into classes. SVM embed the data into a higher-dimensional space, using the so-called *kernel trick* in order to create a decision boundary that is linearly separated and that is the one with the biggest margin between the data of the classes. It works better with unstructured and semi-structured data (e.g. text and images) than logistic regression – that prefers already identified independent variables – when the kernel is not linear, otherwise they are almost similar.

The results with the SVM using both a linear kernel and rbf kernel are reported in Table 3. They are in both cases obtained using **Tfidf Vectorized** with `ngram_range=(1,2)` and `min_df=200`.

	LinearSVM			SVM			
	precision	recall	f1-score	precision	recall	f1-score	support
correct	0.775231	0.868943	0.819417	0.940630	0.978342	0.959115	9927
bugged	0.853375	0.751712	0.799324	0.977778	0.939144	0.958072	10073
accuracy			0.809900			0.958600	20000
macro avg	0.814303	0.810328	0.809371	0.959204	0.958743	0.958593	20000
weighted avg	0.814589	0.809900	0.809297	0.959339	0.958600	0.958590	20000

Table 3: Support Vector Machine

3.4 K-nearest Neighbors Classifier

The **K-nearest Neighbors Classifier** is an instance-based learning technique which classifies computing a majority vote of the nearest neighbors of each sample. A sample, indeed, is assigned to the most

common data class of the neighbors.

With **MyBoW** and the parameter `n_neighbors` set at value 6 the highest results are reached. With this parameter is possible to specify how many neighbors votes for the considered sample class. The results are shown in Table 4.

	precision	recall	f1-score	support
correct	0.870070	0.944394	0.905710	9927
bugged	0.940163	0.861015	0.898850	10073
accuracy			0.902400	20000
macro avg	0.905116	0.902704	0.902280	20000
weighted avg	0.905372	0.902400	0.902255	20000

Table 4: K-Neighbors Classifier

3.5 Perceptron

The **Perceptron** implemented by scikit learn learns the weight used to create a linear combination of the input features and then uses a sign function – which maps the negative values to -1 and the positive ones to +1 – to assign the class to the sample. The **Multi Layer Perceptron Classifier**, instead, other non-linear activation functions such as ReLU are applied to the linear combination. The ReLU activation function – which set to zero the negative values and reproduce linearly the other values – is more powerful than the sign function which can takes only two values. Using the **MLPClassifier** implemented by scikit learn a huge increment is achieved respecting to the Perceptron.

The results with both **Perceptron** and **MLPClassifier** are obtained with **Tifdf Vectorizer** setting `ngram_range=(1,2)` and `min_df=200` and are shown in Table 5.

	Perceptron			MLPClassifier			
	precision	recall	f1-score	precision	recall	f1-score	support
correct	0.705817	0.700413	0.703104	0.821398	0.965951	0.887829	9927
bugged	0.706966	0.712300	0.709623	0.959404	0.793011	0.868308	10073
accuracy			0.706400			0.878850	20000
macro avg	0.706391	0.706357	0.706364	0.890401	0.879481	0.878069	20000
weighted avg	0.706396	0.706400	0.706388	0.890905	0.878850	0.877997	20000

Table 5: Perceptron

3.6 Decision Tree Classifier

The **Decision Tree Classifier** is a supervised Machine Learning algorithm that uses a set of rules built on the training data to make decisions. In particular, the algorithm uses the data features to create boolean questions that lead to splitting the dataset until reaches the leaves which represent the samples belonging to a class.

In order make the model generalize better the parameter `min_samples_split` is set to `12`, i.e. the minimum number of samples required to split an internal node. The best perfomances are obtained with **MyBoW** when the best splits are computed (`splitter = "best"`). These are reported in Table 6.

	precision	recall	f1-score	support
correct	0.941165	0.958799	0.949900	9927
bugged	0.958633	0.940931	0.949699	10073
accuracy			0.949800	20000
macro avg	0.949899	0.949865	0.949800	20000
weighted avg	0.949962	0.949800	0.949799	20000

Table 6: Decision Tree

Using the following lines it is possible to obtain the tree generated from the training of the model shown in Figure 2. It can be seen that the tree is not too deep but is mostly expanded in width.

```
# tree plot

dot_data = tree.export_graphviz(model, out_file=None,
                                class_names=["correct", "incorrect"],
                                filled=True, rounded=True,
                                special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("my_tree")
```

Listing 9: Training and testing the model.

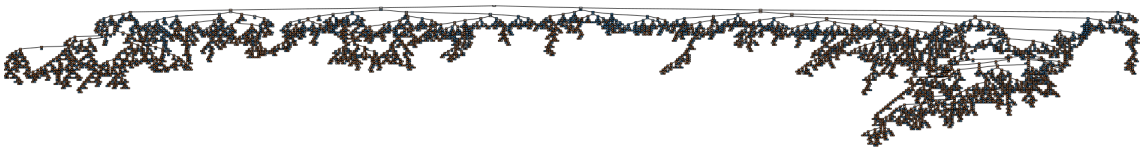

Figure 2: Decision Tree

It's also provided a zoom of the tree in Figure 3 in which can be seen how the dataset is splitted until the leaf. In particular, the blue nodes represent the *bugged* samples while the orange ones represent the *correct* compiled samples. The whole tree is attached in the file `my_tree.pdf`.

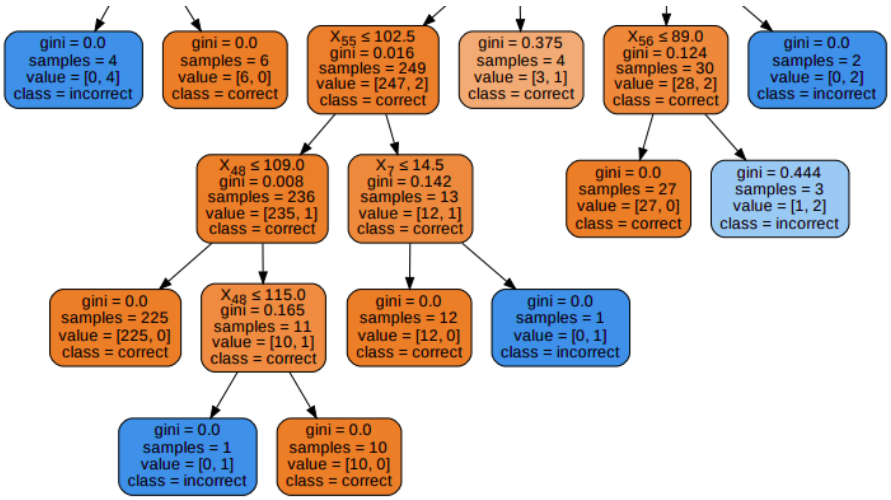

Figure 3: Zoom of the Decision Tree

3.7 Random Forest Classifier

The **Random Forest Classifier** is obtained combining Decision Trees and acts like an ensemble. After `n_estimators` decision trees are built the new sample is classified through majority voting. Note that in this case the Decision Trees are created randomly splitting the nodes.

Also this model works better with **MyBoW Vectorizer** with 100 trees and `min_samples_split=10`. The performnces are reported in Table 7.

	precision	recall	f1-score	support
correct	0.949521	0.989121	0.968917	9927
bugged	0.988819	0.948178	0.968072	1007
accuracy			0.968500	20000
macro avg	0.969170	0.968649	0.968494	20000
weighted avg	0.969313	0.968500	0.968491	20000

Table 7: Random Forest

The confusion matrix is also reported in Figure 4. This shows that the most errors relate to the *bugged* samples, in particular some *bugged* samples are predicted by the model as *correct*.

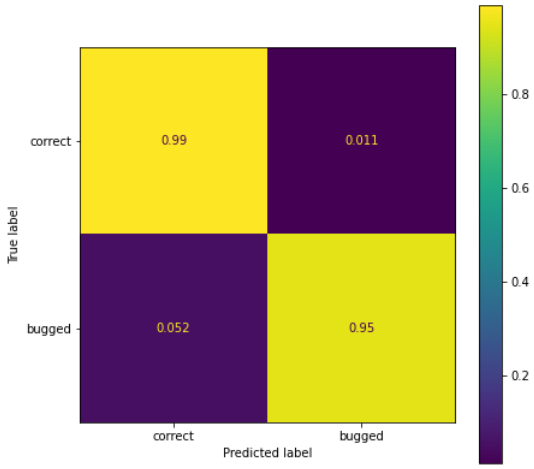

Figure 4: Random Forest Classifier Confusion Matrix.

4 Further analyses

Lastly, in Figure 5 are shown the times taken from the models for the training process and in Figure 6 are reported the accuracy score achieved with the Tfidf Vectorizer. The times with the different vectorizers are almost similar. For further details about the implementation of this plot I refer to the python notebook.

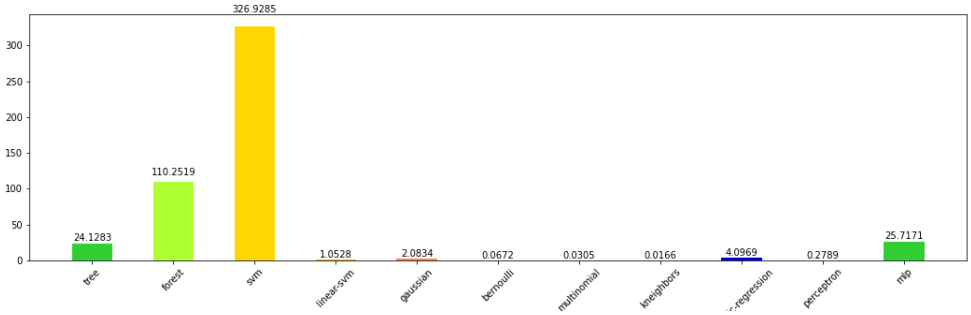

Figure 5: Times taken from the fit with the models and Tfidf Vectorizer.

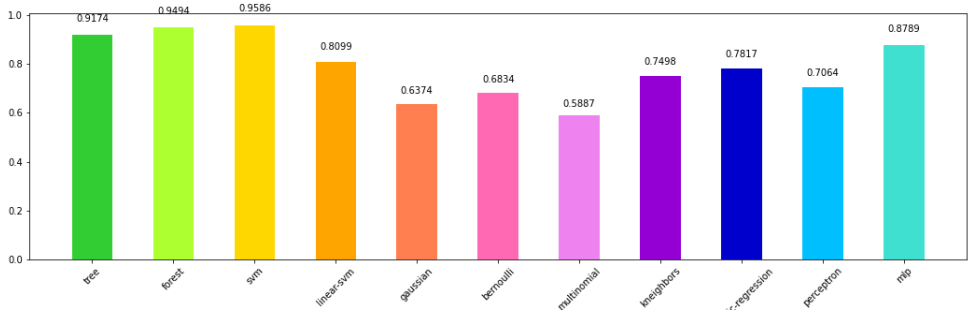

Figure 6: Accuracy scores with the models and Tfidf Vectorizer.

5 Blind Test Classification

Now, I classify with the best model obtained with the new instances of the Blind Test Set and write the result in a txt file. In each row there will be a label relative to one sample. The best model is achieved with the Random Forest Classifier and MyBoW Vectorizer combination. For creating the prediction I need to preprocess the dataset using the MyDataset class. In this case the word vocabulary is the same used for the training phase and the parameter `blind` is set to `True`. The `blind` parameter allows to skip some actions like the extraction of the labels and the dataset split. The code in the Listing 5 shows the steps to take.

```
blind_test_file = "blind_test.csv"

# create the dataset
blind_dataset = MyDataset(blind_test_file, words_vocabulary = words_vocab, bow =
↪  "MyBoW", blind = True)

# predict the blind test
y_pred_blind = model.predict(blind_dataset.X_test)

# save prediction on file
with open("1834906.txt", "w+") as f:
    for p in y_pred_blind:
        f.write(p + "\n")
```

Listing 10: Classification of the blind test.

6 Conclusions

Summarizing, the worst models are the Naive Bayes probably this is caused by the features extracted from the dataset which are not totally indipendent to each other. Also the perceptron with the sign function does not achieves very high results, but using a more powerful activation function there is a good increment. K-nearest Neighbors, Logistic Regression and Linear Support Vector Machine reach almost the same results, probably this is due to the boundary-approach of the algorithms. The best results are achieved by the combination of Decision Trees which creates a Random Forest. This could be a consequence of the fact that the Decision Trees used by the Random Forest are randomly generated, so they does not depend highly on any specific set of features. Therefore, the Random Forest can generalize over the data in a better way and can be more accurate.

References

[1] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[2] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[3] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.