SPAM Filter

Petr Vanc

***Abstract*— Designing SPAM filter using different methods. Aim is to classify Emails into SPAM/HAM categories using Sci-kit learn library framework in python. Methods used are Naive Bayes, Multi-Layer Perceptron and Stochastic Gradient Descent. Results are very diverse, reached more than 85% on advanced accuracy depending on data decomposition, model and parameters. Contains algorithm searching most optimal solution.**

I. Assignment

Purpose of the work is to take the knowledge from labs and choose suitable approach to SPAM filter classification. Outline of work is given with enclosed python file `filter_template.py`. We want to get from training data:

$$max_{inputs}|\texttt{modified\_accuracy()}| \leq 1, \qquad (1)$$

where *inputs* are:

$$inputs = \{method, pipelinestructure, parameters, etc...\}$$

Chosen inputs are analyzed in next chapters.

II. Introduction

Besides labs exercises I started by reading few articles about problematic [1] [2]. From there I got a picture of possible structures of classification pipeline.

I used given dataset containing two directories each of 600 records. Which one quarter are useful messages and rest is Spam.

III. Methodology

I chose following methods:

- Naive Bayes
- Multi-Layer Perceptron
- Stochastic Gradient Descent

Used blocks in pipeline:

- *CountVectorizer*
- *If-Idf*
- Relevant classification method block

Program structure is seen in Fig. 1.

A. Analyzer

In block *CountVectorizer*, I used custom function for Email adjustment words into more friendly form:

```
# Lowercase all words
message = message.lower()
# Split sentences into words array
words = word_tokenize(message)
# Don't include short words
words = [w for w in words if len(w) > 2]
# Stop words
```

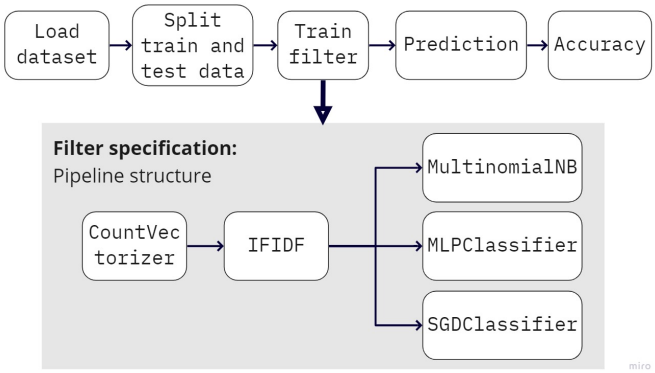

Fig. 1. Program structure.

```
sw = stopwords.words('english')
words = [word for word in words if word not in sw]
# PorterStemmer
stemmer = PorterStemmer()
words = [stemmer.stem(word) for word in words]
```

Listing 1. Analyzer function

I apply this function as callable *analyzer* parameter.

B. Naive bayes

Pipeline structure: *CountVectorizer*, *Ifidf*, *MultinomialNB*

1) CountVectorizer Sklearn block: Converts text of email to matrix of token counts. Parameters of CountVectorizer, that can have influence on result, source from wiki [3]:

`lowercase` - Implemented in analyzer function
`ngram_range` Boundaries of n-values for different word n-grams or char n-grams to be extracted.
`analyzer` - Reference to calling analyzer function
`max_df, min_df` - Ignore words, that have frequency in document higher than threshold.

2) TfidfTransformer Sklearn block: Transform a count matrix to a normalized tf-idf representation. Parameters of TfidfTransformer, that can have influence on result, source from wiki [4]:

`norm={'l2','l1',None}` - Unit norm of output.
`use_idf={True, False}` - I will use as true. Weights inverse document frequency against database.
`smooth_idf={False, True}` - Smoothing idf weights.

3) MultinomialNB Sklearn block: Naive Bayes classifier for multinomial models. Suitable for classification with discrete features e.g. word count. Parameters of MultinomialNB, that can have influence on result, source from wiki [5]:

`alpha` - Additive smoothing parameter, zero for no smoothing.
`fit_prior={True, False}` - Learn class prior probabilities

C. Multi-Layer Perceptron

Pipeline structure: *CountVectorizer*, *Ifidf*, *MLPClassifier*

1) MLPClassifier Sklearn block: Multi-layer Perceptron classifier. Log-loss function using LBFGS or stochastic gradient descent. Parameters of MultinomialNB, that can have influence on result, source from wiki [6]:

`hidden_layer_sizes` - Number of neurons in the i-th hidden layer
`activation={'identity','logistic','tanh','relu'}` - Activation function for the hidden layer.
`solver={'lbfgs','sgd','adam'}` - Configure weight optimization.
`alpha` - L2 penalty parameter.
`learning_rate={'constant','invscaling','adaptive'}` - Learning rate schedule for weight updates.
`learning_rate_init` - The initial learning rate.

D. Stochastic gradient descent

Linear classifiers (SVM) with SGD training. Regularized linear models with stochastic gradient descent learning. The gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule.

Pipeline structure: *CountVectorizer*, *Ifidf*, *SGDClassifier*

1) SGDClassifier Sklearn block: Parameters of MultinomialNB, that can have influence on result, source from wiki [7]:

`loss={'hinge', 'modified_huber', 'perceptron'}` - The loss function to be used.
`penalty={'l2', 'l1', 'elasticnet'}` - The penalty/regularization term.
`alpha` - Constant that multiplies the regularization term.
`l1_ratio` - The Elastic Net mixing parameter.
`learning_rate={'optimal','invscaling','adaptive'}` - The learning rate schedule.
`eta0` - The initial learning rate.

IV. Experiments

Reading few articles, I started to gather some information about possible structure. [1] [2] and review every labs and lecture. I started with edited Naive Bayes filter from given template.

I tried to change parameters myself, to get optimal result, but then I thaught it would be better to run program with every possible combination, so I created algorithm:

```
bestParameters = None
bestAccuracy = 0.0
Dict = Dict[METHODS.index(METHOD)]
keys, values = zip(*Dict.items())
params = [dict(zip(keys, v)) for v in itertools.
    product(*values)]
for param in params:
    if checkDepedences(param):
        continue
    print("Parameters this iteration: ", param)
    filter1 = my_train_filter(X_train, y_train,
    param)
    if filter1 == None:
        continue
    # Compute predictions for training data and
    report our accuracy
    y_tr_pred = predict(filter1, X_train)
    print('Modified accuracy on training data: ',
          modified_accuracy(y_train, y_tr_pred))
    # Compute predictions for testing data and
    report our accuracy
    y_tst_pred = predict(filter1, X_test)
    accuracyNow = modified_accuracy(y_test,
    y_tst_pred)
    print('Modified accuracy on testing data: ',
          modified_accuracy(y_test, y_tst_pred))
    if (accuracyNow > accuracyBefore):
        bestParameters = param
        bestAccuracy = accuracyNow
```

Listing 2. Classification for every combination

Where `checkDepedency` function checks if parameter combination is not invalid, for example 'max_df' parameter must be greater than 'min_df'. From my created parameters dictionary it execute every combination.

Because time dependency, I made better finding solution algorithm. It is based on finding extreme within every parameter, therefore it is not executing every solution but converge towards only one. Unfortunately, it is not optimal, because it can get stuck on local extreme. Useful for number parameters.

```
keys, values = zip(*Dict.items())
params = [dict(zip(keys, v)) for v in itertools.
    product(*values)]

best_param_assesment = []
for opt in values:
    best_param_assesment.append(opt[int(len(opt)/2)
    ])

isBetter = True
accuracyOpt = 0.0
for n_p, param in enumerate(params):
    accuracyBefore = 0.0
    state = 0
    while True:
        n_p_val, n_param_val, quitting =
    moveInParams(param, isBetter)
        if quitting:
            break

        # Searching for extreme in one variable
        tmp_prm = best_param_assesment

        # Replace tmp_prm for current improving
    value
        tmp_prm[n_p] = round(n_param_val, 2)

        # Run training
        filter1 = my_train_filter(X,y,isBetter)

        # Check improvement, predict
        y_tst_pred = predict(filter1, X_test)
        acc = modified_accuracy(y_test, y_tst_pred)
        if acc > accuracyBefore:
            isBetter = True
            if acc > accuracyOpt:
                print("New best solution: ",tmp_prm,
                      "with acc: ", round(acc, 2))
                accuracyOpt = acc
                best_param_assesment = tmp_prm
        else:
            isBetter = False

        accuracyBefore = acc
```

Listing 3. Direction Parameter Search

V. Discussion

My results were:

TABLE I
RESULTS TABLE WITH *If-Idf* BLOCK

Method	Advanced accuracy	
	Train data	Test data
MLP	1.0	0.719
SGD	1.0	0.704
Bayes	1.0	0.663

All results were not good, so I deleted *Ifidf* unit and all results got better. I wonder why deleting this block helped, when it should be worse.

My results without *If-Idf* block are:

TABLE II
RESULTS TABLE WITHOUT *If-Idf* BLOCK

Method	Advanced accuracy	
	Train data	Test data
MLP	1.0	0.837
SGD	1.0	0.86
Bayes	1.0	0.861

Winning configurations are:

```
'alpha': [0.1],
'ngram_range': [(1, 1)],
'max_df': [1.0],
'min_df': [1]
```

Listing 4. Best Bayes configuration

```
'ngram_range': [(1, 2)],
'hidden_layer_sizes': [(100, )],
'activation': ['tanh'],
'solver': ['adam'],
'alpha': [0.001],
'batch_size': ['auto'],
'learning_rate': ['adaptive'],
'learning_rate_init': [0.001],
'power_t': [0.5],
'max_iter': [200],
'shuffle': [True],
'random_state': [None],
'tol': [0.0001],
'verbose': [False],
'warm_start': [False],
'momentum': [0.9],
'nesterovs_momentum': [True],
'early_stopping': [False],
'validation_fraction': [0.1],
'beta_1': [0.9],
'beta_2': [0.999],
'epsilon': [1e-08],
'n_iter_no_change': [10],
'max_fun': [15000]
```

Listing 5. Best MLP configuration

```
'ngram_range': [(2,2)],
'loss': ['modified_huber'],
'penalty': ['elasticnet'],
'alpha': [0.0001],
'l1_ratio': [0.15],
'fit_intercept': [True],
'max_iter': [1000],
'tol': [0.001],
'shuffle': [True],
'verbose': [0],
'epsilon': [0.1],
'n_jobs': [None],
'random_state': [None],
'learning_rate': ['optimal'],
'eta0': [0.0],
'power_t': [0.5]
```

Listing 6. Best SGD configuration

VI. Conclusion

Summary of the work can be written as trying 3 different methods. Results were better when I remove *If-Idf* block. In my attempts, no method that I tried was significantly better than the other. Results can vary depending on dataset. When I split the dataset, the results were more spread out, so I could get 98% success on testing data and then only 80%. When I didn't split data, the results were more stable.

However when using best methods, I can say, that results never go below 80%.

I never really reached optimal solution, because of reduction of parameters due to huge duration of testing. Unfortunately *Sklearn* library doesn't support running tasks on GPU, so running it on school server would help, but not much in my estimation.

Average time of methods were about $30s$ for Naive Bayes, $100s$ for MLP and $40s$ for SGD.

My absolute best option is using Naive Bayes option here described and use dataset split into 70% training and 30% testing. This way I got over 90% on advanced accuracy test five times in a row, but that depends on how the dataset will be split.

References

[1] Author Dilip Kumar: *Spam/ham detection using Naive bayes Classifier*, published by Kaggle, 2018. link.
[2] Author Tejan Karmali: *Spam Classifier in Python from scratch*, published by Towards data science, 2017. link.
[3] Scikit-learn developers (BSD License): *Sklearn Feature Extraction Text CountVectorizer*, published by Scikit Learn, 2007 - 2019. link.
[4] Scikit-learn developers (BSD License): *Sklearn Feature Extraction Text TfidfTransformer*, published by Scikit Learn, 2007 - 2019. link.
[5] Scikit-learn developers (BSD License): *Sklearn Naive Bayes MultinomialNB*, published by Scikit Learn, 2007 - 2019. link.
[6] Scikit-learn developers (BSD License): *Sklearn Neural Network MLPClassifier*, published by Scikit Learn, 2007 - 2019. link.
[7] Scikit-learn developers (BSD License): *Sklearn Linear model SGDClassifier*, published by Scikit Learn, 2007 - 2019. link.