Download Data

```
!wget http://www2.aueb.gr/users/ion/data/lingspam_public.tar.gz
!tar -zxf lingspam_public.tar.gz
```

```
--2021-11-01 19:27:09--  http://www2.aueb.gr/users/ion/data/lingspam_public.tar.gz
Resolving www2.aueb.gr (www2.aueb.gr)... 195.251.255.138
Connecting to www2.aueb.gr (www2.aueb.gr)|195.251.255.138|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11564714 (11M) [application/x-gzip]
Saving to: ‘lingspam_public.tar.gz’

lingspam_public.tar 100%[===================>]  11.03M  1.73MB/s    in 7.0s

2021-11-01 19:27:17 (1.59 MB/s) - ‘lingspam_public.tar.gz’ saved [11564714/11564714]
```

Feature selection using information gain(IG) matrix

```
import os
import re
import math
import numpy as np

from collections import Counter

# Copy from readme.txt
# Each one of the 10 subdirectories contains both spam and legitimate
# messages, one message in each file. Files whose names have the form
# spmsg*.txt are spam messages. All other files are legitimate messages.

path = "lingspam_public/lemm_stop"
word_filter = re.compile(r'^[a-z]+-?[a-z]+[0-9]*$')

spam_words = []
ham_words = []
train_X = []
train_Y = []
spam_num = 0
ham_num = 0

# Build IG and training dataset
for dir in [os.path.join(path, 'part' + str(i)) for i in range(1, 10)]:
    mails = [(os.path.join(dir, file_name), 'spmsg' in file_name) for file_name in os.li

    for file_path, spam in mails:
        with open(file_path) as f:
            content = f.readlines()[2]
            words = content.split()
            filtered_words = list(set([w for w in words if (len(w) > 1 and re.match(word
            # Add words to training set
            train_X.append(filtered_words)
            # Label spam as 1, ham as 0
            if spam:
                train_Y.append(1)
                spam_words += filtered_words
                spam_num += 1
            else:
                train_Y.append(0)
                ham_words += filtered_words
                ham_num += 1


train_all_counter = Counter(spam_words + ham_words)
train_spam_counter = Counter(spam_words)
train_ham_counter = Counter(ham_words)

print(f'Loading {spam_num + ham_num} emails, {spam_num} of them are spam, \
and {ham_num} of them are ham.')
print(f'Within the total of {len(train_all_counter)} words, \
getting {len(train_spam_counter)} words occur in spam, \
and {len(train_ham_counter)} words occur in ham email.')
```

```
Loading 2602 emails, 432 of them are spam, and 2170 of them are ham.
Within the total of 48583 words, getting 8657 words occur in spam, and 45091 words occu
```

```
print(train_X[0])
```

```
['letter', 'establish', 'convert', 'participate', 'start', 'customer', 'simple', 'inter
```

```
# Build test data set
test_X = []
test_Y = []

test_path = 'lingspam_public/lemm_stop/part10'
test_mail_path = [(os.path.join(test_path, file_name), 'spmsg' in file_name) for file_name
for file_path, spam in test_mail_path:
    with open(file_path) as f:
        content = f.readlines()[2]
        words = content.split()
        filtered_words = list(set([w for w in words if (len(w) > 1 and re.match(word_filte
        # Add words to test set
        test_X.append(filtered_words)
        # Label spam as 1, ham as 0
        if spam:
            test_Y.append(1)
        else:
            test_Y.append(0)
```

```
# Helper parameter and function for IG
def Log(i):
    return np.log(i) if i != 0 else 0


total_num = spam_num + ham_num
P_SPAM = spam_num / total_num
P_HAM = 1 - P_SPAM
HC = -P_SPAM * Log(P_SPAM) -P_HAM * Log(P_HAM)
```

```
# Generating IG for all words

IG = {}

for word in train_all_counter:
    all_occur = train_all_counter[word]
    spam_occur = train_spam_counter[word]
    ham_occur = train_ham_counter[word]

    # X=1, C = ham
    P1h = (ham_occur / ham_num) * P_HAM
    # X=0, C = ham
    P0h = (1 - (ham_occur / ham_num)) * P_HAM
    # X=1, C = spam
    P1s = (spam_occur / spam_num) * P_SPAM
    # X=0, C = spam
    P0s = (1 - (spam_occur / spam_num)) * P_SPAM
    # P(X=1)
    P1 = all_occur / total_num
    # P(X=0)
    P0 = 1 - P1

    # Note: H(C|X) = - sum(P(X,C) log(C|X))
    # But it is easier to do addition operation here :)
    HCX = P1h*Log(P1h/P1) + P0h*Log(P0h/P0) + P1s*Log(P1s/P1) + P0s*Log(P0s/P0)
    IG[word] = HC + HCX

top_words = [k for k, v in sorted(IG.items(), key=lambda item: -item[1])]
top_10 = top_words[:10]
top_100 = top_words[:100]
top_1000 = top_words[:1000]
```

```
print(top_10)
```

```
['language', 'remove', 'free', 'linguistic', 'university', 'money', 'click', 'market',
```

Classifiers Implementation

Generate Feature Matrix

```python
def get_feature(N, dataset, term_frequency=False):
    """Generate feature matrix based on top N words."""
    # print(term_frequency)
    top_n = {v:i for i,v in enumerate(top_words[:N])}
    feature_matrix = np.zeros((len(dataset), N))
    for j, content in enumerate(dataset):
        for word in content:
            if word in top_n:
                if term_frequency:
                    feature_matrix[j, top_n[word]] += 1
                else:
                    feature_matrix[j, top_n[word]] = 1

    return feature_matrix
```

Bernoulli NB Classifier

```python
class BNB:
    """Naive Bayes classifier for multivariate Bernoulli models."""
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Actually, no need in this homework :)
        self.spam_num = 0
        self.ham_num = 0
        self.P_X_spam = []
        self.P_X_ham = []


    def fit(self, train_x, train_y):
        """Fit Naive Bayes classifier according to train_x, train_y."""
        self.spam_num = sum(train_y)
        self.ham_num = len(train_y) - sum(train_y)
        rows, feature_num = train_x.shape

        self.P_X_spam = np.zeros(feature_num)
        self.P_X_ham = np.zeros(feature_num)
        for i in range(feature_num):
            x_spam = 0
            x_ham = 0
            for j in range(rows):
                if train_x[j,i]:
                    if train_y[j]:
                        x_spam += 1
                    else:
                        x_ham += 1
            self.P_X_spam[i] = (1 + x_spam) / (self.spam_num + 2)
            self.P_X_ham[i] = (1 + x_ham) / (self.ham_num + 2)


    def _row_predict(self, one_sample):
        """Predict one record of the data set."""
        prob = 1
        for i, feature in enumerate(one_sample):
            p0 = self.P_X_ham[i]
            p1 = self.P_X_spam[i]
            prob *= (p1 / p0) if feature == 1 else (1 - p1) / (1 - p0)
        return prob * self.spam_num / self.ham_num


    def predict(self, test_x):
        """Perform classification on an array of test vectors test_x."""
        pred = np.zeros((test_x.shape[0]))
        for i in range(test_x.shape[0]):
            pred[i] = int(self._row_predict(test_x[i]) > 1)
        return pred


    def score(self, test_x, test_y):
        """Return the mean accuracy on the given test data and labels."""
        y_pred = self.predict(test_x)
        count = 0
        for i in range(len(test_y)):
            count += 1 if y_pred[i] == test_y[i] else 0
        return count / len(test_y)

```

Multinominal NB Classifier

```
class MNB:
    """Naive Bayes classifier for multinomial models."""
    def __init__(self, alpha=1.0):
        self.alpha = alpha
        self.spam_num = 0
        self.ham_num = 0
        self.P_X_spam = []
        self.P_X_ham = []


    def fit(self, train_x, train_y):
        """Fit Naive Bayes classifier according to train_x, train_y."""
        self.spam_num = sum(train_y)
        self.ham_num = len(train_y) - sum(train_y)
        rows, feature_num = train_x.shape

        self.P_X_spam = np.zeros(feature_num)
        self.P_X_ham = np.zeros(feature_num)
        for i in range(feature_num):
            x_spam = 0
            x_ham = 0
            for j in range(rows):
                if train_x[j,i]:
                    if train_y[j]:
                        x_spam += 1
                    else:
                        x_ham += 1
            self.P_X_spam[i] = (1 + x_spam) / (self.spam_num + 2)
            self.P_X_ham[i] = (1 + x_ham) / (self.ham_num + 2)


    def _row_predict(self, one_sample):
        """Predict one record of the data set."""
        prob = 1
        for i, feature in enumerate(one_sample):
            p0 = self.P_X_ham[i]
            p1 = self.P_X_spam[i]
            prob += (Log(p1) - Log(p0)) if feature == 1 else 0
        return prob + Log(self.spam_num) - Log(self.ham_num)


    def predict(self, test_x):
        """Perform classification on an array of test vectors test_x."""
        pred = np.zeros((test_x.shape[0]))
        for i in range(test_x.shape[0]):
            pred[i] = int(self._row_predict(test_x[i]) > 0)
        return pred


    def score(self, test_x, test_y):
        """Return the mean accuracy on the given test data and labels."""
        y_pred = self.predict(test_x)
        count = 0
        for i in range(len(test_y)):
            count += 1 if y_pred[i] == test_y[i] else 0
        return count / len(test_y)
```

Classifiers Comparison

```
from sklearn.naive_bayes import MultinomialNB, BernoulliNB
from sklearn.metrics import classification_report

top_N = [10, 100, 1000]
result_accuracy = []

for N in top_N:
    Xprint = get_feature(N, train_X)
    # X label with term frequency
    X_tf = get_feature(N, train_X, term_frequency=True)
    Y = np.array(train_Y)

    X_test = get_feature(N, test_X)
    # X label with term frequency
    X_test_tf = get_feature(N, test_X, term_frequency=True)
    Y_test = np.array(test_Y)

    model1 = BNB()
    model2 = MNB()
    model3 = MNB()

    # train models
    model1.fit(X, Y)
    model2.fit(X, Y)
    model3.fit(X_tf, Y)

    # predict
    y_1 = model1.predict(X_test)
    y_2 = model2.predict(X_test)
    y_3 = model3.predict(X_test_tf)

    print(f'--------------- Using top {N} features --------------')
    print( '--------------- Bernoulli Naive Bayes ---------------')
    print(classification_report(Y_test,y_1, target_names = ["ham", "spam"]))
    print('--- Multinominal Naive Bayes with binary features ---')
    print(classification_report(Y_test,y_2, target_names = ["ham", "spam"]))
    print('---- Multinominal Naive Bayes with term frequency ---')
    print(classification_report(Y_test,y_3, target_names = ["ham", "spam"]))

    N_accuray = [model1.score(X_test,Y_test),
                 model2.score(X_test,Y_test),
                 model3.score(X_test_tf,Y_test)]
    result_accuracy.append(N_accuray)
```

```
--------------- Using top 10 features --------------
--------------- Bernoulli Naive Bayes ---------------
              precision    recall  f1-score   support

         ham       0.96      0.98      0.97       242
```

```
        spam       0.89      0.82      0.85        49

    accuracy                           0.95       291
   macro avg       0.93      0.90      0.91       291
weighted avg       0.95      0.95      0.95       291

--- Multinominal Naive Bayes with binary features ---
              precision    recall  f1-score   support

         ham       0.98      0.95      0.97       242
        spam       0.80      0.92      0.86        49

    accuracy                           0.95       291
   macro avg       0.89      0.94      0.91       291
weighted avg       0.95      0.95      0.95       291

---- Multinominal Naive Bayes with term frequency ---
              precision    recall  f1-score   support

         ham       0.98      0.95      0.97       242
        spam       0.80      0.92      0.86        49

    accuracy                           0.95       291
   macro avg       0.89      0.94      0.91       291
weighted avg       0.95      0.95      0.95       291

--------------- Using top 100 features --------------
--------------- Bernoulli Naive Bayes ---------------
              precision    recall  f1-score   support

         ham       0.94      1.00      0.97       242
        spam       1.00      0.67      0.80        49

    accuracy                           0.95       291
   macro avg       0.97      0.84      0.89       291
weighted avg       0.95      0.95      0.94       291

--- Multinominal Naive Bayes with binary features ---
              precision    recall  f1-score   support

         ham       1.00      0.98      0.99       242
        spam       0.89      1.00      0.94        49

    accuracy                           0.98       291
   macro avg       0.95      0.99      0.96       291
weighted avg       0.98      0.98      0.98       291

---- Multinominal Naive Bayes with term frequency ---
              precision    recall  f1-score   support

         ham       1.00      0.98      0.99       242
        spam       0.89      1.00      0.94        49

    accuracy                           0.98       291
```

Spam precision and spam recall

Params	Spam Precision	Spam Recall
Top 10 Words, Bernoulli Naive Bayes	0.89	0.82
Top 10 Words, Multinominal Naive Bayes with binary features	0.80	0.92
Top 10 Words, Multinominal Naive Bayes with term frequency	0.80	0.92
Top 100 Words, Bernoulli Naive Bayes	1.00	0.67
Top 100 Words, Multinominal Naive Bayes with binary features	0.89	1.00
Top 100 Words, Multinominal Naive Bayes with term frequency	0.89	1.00
Top 1000 Words, Bernoulli Naive Bayes	1.00	0.61
Top 1000 Words, Multinominal Naive Bayes with binary features	1.00	1.00
Top 1000 Words, Multinominal Naive Bayes with term frequency	1.00	1.00

Compare with sklearn

```
result_accuracy_sk = []

for N in top_N:
    model1_sk = BernoulliNB()
    model2_sk = MultinomialNB()
    model3_sk = MultinomialNB()

    X = get_feature(N, train_X)
    X_tf = get_feature(N, train_X, term_frequency=True)
    Y = np.array(train_Y)

    X_test = get_feature(N, test_X)
    X_test_tf = get_feature(N, test_X, term_frequency=True)
    Y_test = np.array(test_Y)

    # train models
    model1_sk.fit(X, Y)
    model2_sk.fit(X, Y)
    model3_sk.fit(X_tf, Y)

    # predict
    N_accuray = [model1_sk.score(X_test,Y_test),
                 model2_sk.score(X_test,Y_test),
                 model3_sk.score(X_test_tf,Y_test)]
    result_accuracy_sk.append(N_accuray)


print(result_accuracy)
print(result_accuracy_sk)
```

```
[[0.9518900343642611, 0.9484536082474226, 0.9484536082474226], [0.9450171821305842, 0.9
[[0.9518900343642611, 0.9518900343642611, 0.9518900343642611], [0.9450171821305842, 0.9
```

SVM based spam filter

```
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import MinMaxScaler
from sklearn.base import clone


feature_sizes = [10, 100, 200]
modes = ["bf", "tf"]
gamma = [1, 0.1, 0.01, 0.001]
C = [0.1, 1, 10, 100]
degree = [2,3,10]
tuned_parameters = [{'kernel': ['rbf','sigmoid'], 'gamma': gamma,'C': C},
                    {'kernel': ['linear'], 'C': C},
                    {'kernel': ['poly'], 'gamma': gamma, "degree" : degree, 'C':C}
                    ]
now_best = {}
now_score = 0
best_clf = SVC()
for N in feature_sizes:
    for mode in modes:
        X = get_feature(N, train_X, term_frequency=(mode=='tf'))
        X_test = get_feature(N, test_X, term_frequency=(mode=='tf'))
        Y = np.array(train_Y)
        if mode == 'tf':
            ss = MinMaxScaler()
            X = ss.fit_transform(X)
            X_test = ss.transform(X_test)
        clf = GridSearchCV(SVC(), tuned_parameters, cv = 5, scoring='recall_macro', verbos
        clf.fit(X, Y)

        if clf.best_score_ > now_score:
            now_score = clf.best_score_
            now_best = {"size":N, "mode":mode, "score":clf.best_score_, "params":clf.best_
            best_clf = clone(clf.best_estimator_)
            print (now_best)

N = now_best["size"]
mode = now_best["mode"]
X = get_feature(N, train_X, term_frequency=(mode=='tf'))
X_test = get_feature(N, test_X, term_frequency=(mode=='tf'))
Y = np.array(train_Y)
Y_test = np.array(test_Y)

if mode == "tf":
    ss = MinMaxScaler()
    X = ss.fit_transform(X)
    X_test = ss.transform(X_test)

best_clf.fit(X, Y)
pred_y = best_clf.predict(X_test)
print (now_best)
print (classification_report(Y_test, pred_y, target_names = ["ham", "spam"]))
```

```
{'size': 10, 'mode': 'bf', 'score': 0.9333773098447026, 'params': {'C': 1, 'gamma': 1,
{'size': 100, 'mode': 'bf', 'score': 0.9611211300362037, 'params': {'C': 1, 'gamma': 0.
{'size': 200, 'mode': 'bf', 'score': 0.9699466122688307, 'params': {'C': 10, 'gamma': 0
{'size': 200, 'mode': 'bf', 'score': 0.9699466122688307, 'params': {'C': 10, 'gamma': 0
              precision    recall  f1-score   support

         ham       0.98      1.00      0.99       242
        spam       0.98      0.88      0.92        49

    accuracy                           0.98       291
   macro avg       0.98      0.94      0.96       291
weighted avg       0.98      0.98      0.98       291
```

Methodology

I use `GridSearchCV` to help me find the best parameters for SVM. Also, we've see that higher Information Gain feature matrix can have a better result in prediction. So I compare the score between "BF" and "TF" and the score among "N={10, 100, 200}". The result shows that using **top 200** words, **binary features**, and `{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}` as the SVM parameter can get the best result.

Adversarial Classification

Attacker

```
class Attacker:
    def __init__(self):
        self.Lo = []
        self.Lo0 = []
        self.Lo1 = []


    def cal_Lo(self, train_x, train_y):
        rows, feature_num = train_x.shape
        self.Lo = np.zeros(feature_num)
        self.Lo0 = np.zeros(feature_num)
        self.Lo1 = np.zeros(feature_num)

        for i in range(feature_num):
            x_spam = 0
            x_ham = 0
            for j in range(rows):
                if train_x[j,i]:
                    if train_y[j]:
                        x_spam += 1
                    else:
                        x_ham += 1
            P_X1_ham = x_ham / ham_num
            P_X1_spam = x_spam / spam_num
            P_X0_ham = 1 - P_X1_ham
            P_X0_spam = 1 - P_X1_spam
            self.Lo1[i] = Log(P_X1_spam/P_X1_ham)
            self.Lo0[i] = Log(P_X0_spam/P_X0_ham)
            self.Lo[i] = self.Lo1[i] - self.Lo0[i]


    def _row_attack(self, one_sample):
        sort_Lo = sorted(range(len(self.Lo)), key=lambda i: self.Lo[i])

        cost = 0
        sigmaLoX = 0
        for i,v in enumerate(one_sample):
            sigmaLoX += self.Lo1[i] if v==1 else self.Lo0[i]
        if sigmaLoX < 0:
            return 0, one_sample

        for i in sort_Lo:
            if(self.Lo[i] >= 0):
                return cost, one_sample
            if one_sample[i]:
                continue
            one_sample[i] = 1
            sigmaLoX += self.Lo[i]
            cost += 1
            if (sigmaLoX < 0):
                return cost, one_sample


    def attack(self, test_x, test_y):
        costs = []
        new_test_x = []
        rows, feature_num = test_x.shape
        for i in range(rows):
            if test_y[i]:
                cost, changed = self._row_attack(test_x[i])
                new_test_x.append(changed)
                costs.append(cost)
            else:
                new_test_x.append(test_x[i])

        return np.mean(costs), np.array(new_test_x)

```

Defender

```
class Defender:
    def __init__(self):
        self.P_X1_ham = []
        self.P_X1_spam = []
        self.P_X0_ham = []
        self.P_X0_spam = []
        self.Lo = []
        self.Lo0 = []
        self.Lo1 = []
        self.result = []

    def fit(self, train_x, train_y):
        rows, feature_num = train_x.shape
        self.P_X1_ham = np.zeros(feature_num)
        self.P_X1_spam = np.zeros(feature_num)
        self.P_X0_ham = np.zeros(feature_num)
        self.P_X0_spam = np.zeros(feature_num)
        self.Lo = np.zeros(feature_num)
        self.Lo0 = np.zeros(feature_num)
        self.Lo1 = np.zeros(feature_num)

        for i in range(feature_num):
            x_spam = 0
            x_ham = 0
            for j in range(rows):
                if train_x[j,i]:
                    if train_y[j]:
                        x_spam += 1
                    else:
                        x_ham += 1
            self.P_X1_ham[i] = x_ham / ham_num
            self.P_X1_spam[i] = x_spam / spam_num
            self.P_X0_ham[i] = 1 - self.P_X1_ham[i]
            self.P_X0_spam[i] = 1 - self.P_X1_spam[i]
            self.Lo1[i] = Log(self.P_X1_spam[i] / self.P_X1_ham[i])
            self.Lo0[i] = Log(self.P_X0_spam[i] / self.P_X0_ham[i])
            self.Lo[i] = self.Lo1[i] - self.Lo0[i]


    def _row_attack(self, one_sample):
        sort_Lo = sorted(range(len(self.Lo)), key=lambda i: self.Lo[i])

        cost = 0
        sigmaLoX = 0
        for i,v in enumerate(one_sample):
            sigmaLoX += self.Lo1[i] if v==1 else self.Lo0[i]
        if sigmaLoX < 0:
            return 0, one_sample

        for i in sort_Lo:
            if(self.Lo[i] >= 0):
                return cost, one_sample
            if one_sample[i]:
                continue
            one_sample[i] = 1
            sigmaLoX += self.Lo[i]
            cost += 1
            if (sigmaLoX < 0):
                return cost, one_sample


    def defence(self, model, attacked_x):
        for sample in attacked_x:
            if model._row_predict(sample):
                self.result += [1]
                continue
            new_matrix = [[]]
            for i in sample:
                if i:
                    new_row = [(x + [0]) for x in new_matrix] + [(x + [1]) for x in new_ma
                else:
                    new_matrix_f = [(x + [0]) for x in new_matrix]
                new_matrix = new_matrix_f
            new_matrix = [np.asarray(x) for x in new_matrix]
            realFroms = []
            for origin in new_matrix:
                _, attacked = self._row_attack(origin)
                if np.array_equal(attacked, sample):
                    if model._row_predict(origin):
                        realFroms += [origin]
            if len(realFroms):
                P_spam = 0
                P_ham = 0
                for origin in realFroms:
                    for i in origin:
                        P_spam *= self.P_X1_spam[i] if i else self.P_X0_spam[i]
                        P_ham *= self.P_X1_ham[i] if i else self.P_X0_ham[i]
                    P_spam += 1
                    P_ham += 1
                self.result += [0 if P_spam < P_ham else 1]
            else:
                self.result += [0]

        return np.array(self.result)
```

Adversarial attack analysis

```
from sklearn.metrics import confusion_matrix

N = 10
X = get_feature(N, train_X)
Y = np.array(train_Y)
X_test = get_feature(N, test_X)
Y_test = np.array(test_Y)

# Use original model to predict Y_test
botnet = BNB()
botnet.fit(X, Y)
Y_pred_origin = botnet.predict(X_test)
print("Original classification report before attack")
print(classification_report(Y_test, Y_pred_origin, target_names = ["ham", "spam"]))
print("Confusion Matrix")
print(confusion_matrix(Y_test, Y_pred_origin))

# Assume that Attacker knows
# - original model's result: Y_pred_origin
# - original test words list: test_X
# - oringinal test label: test_Y
# Using 'Add-Words' attack
hacker = Attacker()
hacker.cal_Lo(X, Y)
cost, X_test_attacked = hacker.attack(X_test, Y_test)
Y_pred_attacked = botnet.predict(X_test_attacked)
print("Original classification report after attack")
print(classification_report(Y_test, Y_pred_attacked, target_names = ["ham", "spam"]))
print("Confusion Matrix")
print(confusion_matrix(Y_test, Y_pred_attacked))
print(f"Average cost = {cost}")

# Denfender
antispam = Defender()
antispam.fit(X, Y)
Y_pred_defence = antispam.defence(botnet, X_test_attacked)
print("Original classification report after defence")
print(classification_report(Y_test, Y_pred_defence, target_names = ["ham", "spam"]))
print("Confusion Matrix")
print(confusion_matrix(Y_test, Y_pred_defence))
print(f"Average cost = {cost}")
```

```
Original classification report before attack
              precision    recall  f1-score   support

         ham       0.96      0.98      0.97       242
        spam       0.89      0.82      0.85        49

    accuracy                           0.95       291
   macro avg       0.93      0.90      0.91       291
weighted avg       0.95      0.95      0.95       291

Confusion Matrix
[[237   5]
 [  9  40]]
Original classification report after attack
              precision    recall  f1-score   support

         ham       0.83      0.98      0.90       242
        spam       0.29      0.04      0.07        49

    accuracy                           0.82       291
   macro avg       0.56      0.51      0.49       291
weighted avg       0.74      0.82      0.76       291

Confusion Matrix
[[237   5]
 [ 47   2]]
Average cost = 1.7959183673469388
Original classification report after defence
              precision    recall  f1-score   support

         ham       0.00      0.00      0.00       242
        spam       0.17      1.00      0.29        49

    accuracy                           0.17       291
   macro avg       0.08      0.50      0.14       291
weighted avg       0.03      0.17      0.05       291

Confusion Matrix
[[  0 242]
 [  0  49]]
Average cost = 1.7959183673469388
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1272: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

✓ 0s completed at 5:51 PM ● ×