Spam e-mail classifiers using spambase dataset

Alexander Björnsson
alexanderb13@ru.is

Sara Sabrina Zemljič
sara.zemljic@gmail.com

School of Computer Science
Reykjavik University, Iceland

October 10, 2017

Abstract

We can all agree that the amount of spam we get every day by e-mail is almost limitless. The annoying spam messages are also getting more and more dangerous since they may contain viruses or other threats. Therefore it is of no surprise that spam filters have been studied quite intesively with various methods from machine learning.

We present our models for spam filtering on the dataset spambase using methods like k nearest-neighbors, Naïve Bayes and neural networks.

Keywords: e-mail, spam, false positives, k nearest-neighbors (kNN), Naïve Bayes (NB), artificial neural network (ANN).

1 Introduction

Cormack [3, p.2] defined spam as *unwanted communication intended to be delivered to an indiscriminate target, directly or indirectly, notwithstanding measures to prevent its delivery.* Very commonly people would also denote not-spam e-mails as *ham*. Spam nowadays can have different purposes but it generally presents a threat for receiver so researchers have been looking for ways to prevent spam e-mails being opened but such that ham messages would not be effected by them. Here it is very important to note that classifying a spam message wrongly as a ham is not as bad as classifying a ham message as a spam. So when comparing models we are mainly just interested in whether a model gives us lower values for false positives.

Let us first list a few models that have been trained for e-mail spam detection. Even though it is not really up-to-date anymore, Cormack [3] gives a very extensive overview of most important spam filter techniques both, hand-crafted and machine-learning ones. Among most popular are kNN, ANN, support vector machines (SVM), and also Bayesian methods. The latter ones have a lot of possibilities and they often depend on distribution of particular attributes in datasets. Most of the options on Naïve Bayesian classifiers were compared and studied in [1], as well as recently in [5] and [11], where NB was compared to SVM. ANN for spam e-mail filtering has been studied in [9], [6] and has been compared to different other classifiers in [4]. There were also several papers focusing on feature selection and parameters optimization [7] or on content-based classification [10]. An intriguing add-on to the very extensive research on spam filters is another recent paper [2] studying unsupervised neural network methods which were then applied on several datasets including our spambase.

The rest of our paper is organized as follows. In the next section we introduce our dataset and what preprocesing we performed on it. In Section 3 we discuss the classifiers we trained and their performance is presented in Section 4. We conclude the paper with final thoughts about the filters and present our code in the appendix.

2 Dataset

The dataset we are using for this research is the *spambase*, a SPAM E-mail Database [8] donated by George Forman (gforman at nospam hpl.hp.com, 650-857-7835) from Hewlett-Packard Labs and was generated in June/July 1999. The spam e-mails in the collection include advertisements for products or web sites, make money fast schemes, chain letters, etc. The collection of non-spam e-mails in the database came from filed work and personal e-mails, therefore the dataset is very specific. For example words like 'george' or the code '650' are very strong indicators that an e-mail is not spam.

There are 4601 instances out of which 1813 (about 39.4%) are spam. Each instance is represented as a vector with 58 entries, so 57 + 1 columns, out of which

– the last one gives us the class information, it is either 1 (= spam) or 0 (= ham);

– first 48 columns are continuous real attributes in the range $[0, 100]$ of type `word_freq_WORD` (e.g. `word_freq_make`, `word_freq_address`, `word_freq_all`, etc.). These attributes present the ration of the number of times the `WORD` appears in the e-mail) over the total number of words in e-mail. A "word" in this case is any string of alphanumeric characters bounded by non-alphanumeric characters or end-of-string.

– next 6 columns are continuous real attributes in the range $[0, 100]$ of type `char_freq_CHAR` (e.g. `char_freq_;`, `char_freq_!`, `char_freq_$`), which present the percentage of characters in the e-mail that match `CHAR`.

– last 3 attributes are continuous real in the range $[1, \ldots]$ and are a bit special. They count occurences with capital letters. For example, `capital_run_length_average` counts average length of *uninterrupted sequences of capital letters*.

This dataset is preprocesed, which means the attributes were chosen this way to classify spam the best when they introduced the dataset in 1999. It also has no missing values. The data is not available in the raw format, so it is imposible to experiment with other attributes that could be extracted from their e-mails. Despite that this dataset was used in various studies for testing different classifiers for recognizing spam e-mails.

First we split our data into train and test sets, using 20% of the data for testing our models at the end. For all calculations presented in Section 3 we thus only use train part of the data. Furthermore, to ensure that we are always using the same train set we included the same `random_state` in all calculations. Our classes (spam-ham) are in about $2 : 3$ ratio, so we checked that the train-test split has a similar ratio of spam and ham e-mails as well.

For some of our models we required normalized data. We assume some outliers in our data therefore we use standard score normalization (in scikit-learn this corresponds to `norm='l2'` when using normalize `function`).

3 Models

After studying the literature of spam e-mail filters, in particular what has been studied on the spambase dataset we have decided to find a spam filter classifiers for our dataset with three different methods. First we used k nearest-neighbour classifier, then we studied Naïve Bayes methods on our dataset and finally we have built an artificial neural network classifier. In Section 4 we will discuss which of the models fits best for our dataset.

Let us once again mention that we are mainly interested in minimizing the false positives because we thing classifying a regular e-mail as spam is much larger sin than having one or two spam e-mails in our mailbox. Therefore we merely focus on precision rather than accuracy of our models.

k nearest-neighbour classifier

Our data has 57 attributes, that means we are dealing with 57-dimensional space and with "only" 4601 instances, which makes them very sparse in the space. Therefore we have decided to first reduce dimensionality of the dataset using Principle Component Analysis (PCA). In general, the more we reduce the dimensionality the better, but on the other hand using just a single component of the dataset might hide some important characteristics of data. First we looked at what number of

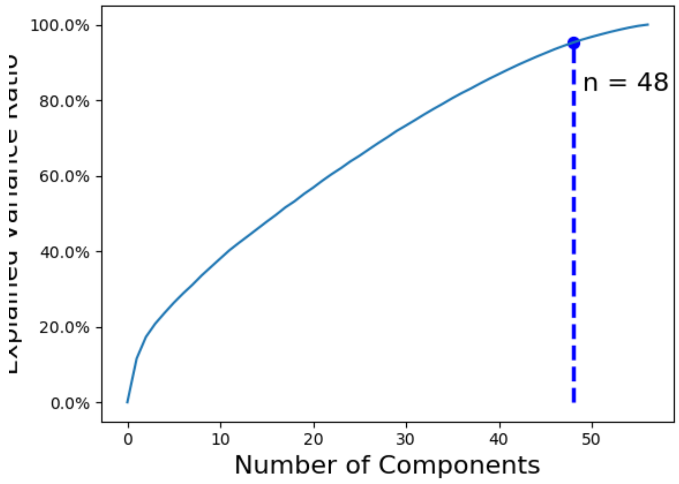

Figure 1: Variance of data represented by components in PCA

components we would still have 95% of variance of our data. This gave us 48 components (see Figure 1), but since this is still quite high number for dimensionality for kNN model we decided to research which number of components would give us best results for a simple (standard[1]) kNN model. Using 10-fold cross-validation and precision we deducted that n_components should be 14. These results are presented in Figure 2 (For more details see the code in the Appendix.)

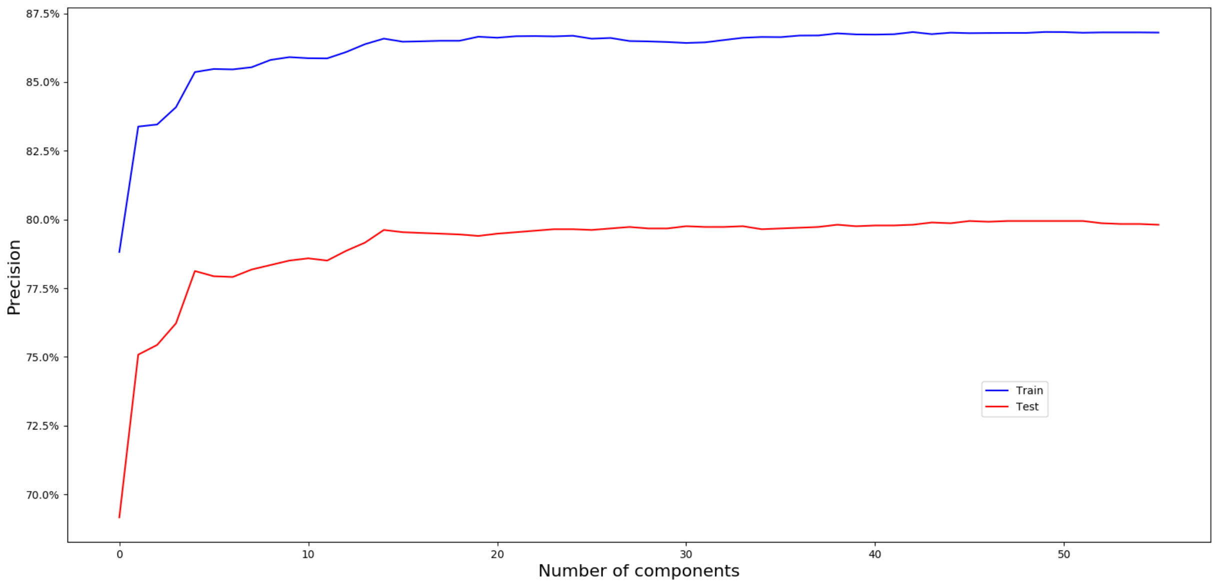

Figure 2: Precision in relation to number of components in PCA

[1]Standard is refering to default value suggested by scikit learn, which is $k = 5$.

Now that we set the number of components in PCA to 14 we can focus on kNN and try to find the best parameters for kNN for our data. To determine what number of neighbors to look at we applied yet another cross-validation: we look at precision for $k \in [1, 100]$, see Figure 3. The best precision on test data we get with $k = 4$.

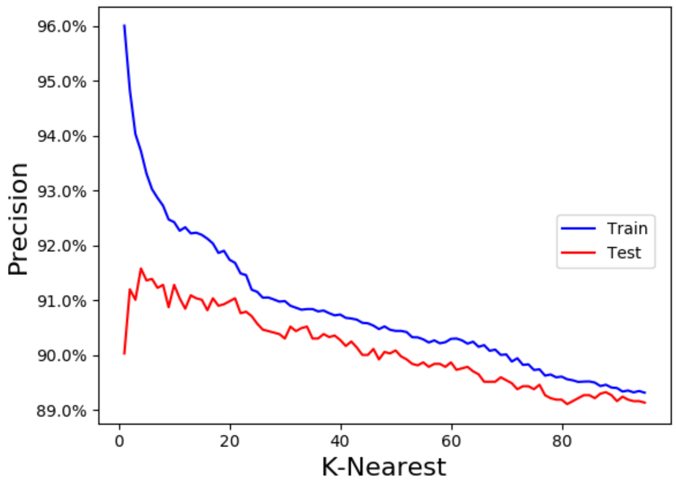

Figure 3: Precision in relation to number of components in PCA

Finally, let us check that all the preprocesing and hypertuning really gave us a better model by comparing our latest kNN classifier trained on 14 principal components with $k = 4$ to the standard kNN model (that is with all of 57 attributes and $k = 5$. The comparison can be found in Figure 4 and we can deduct that hypertuned model clearly gives us better precision.

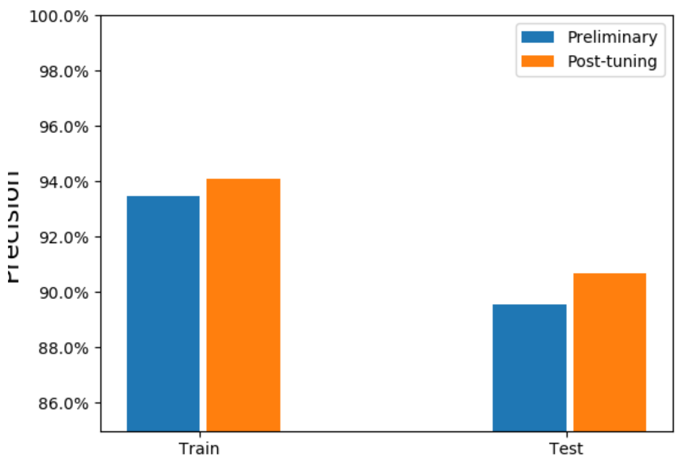

Figure 4: Comparisson of kNN models

NB classifier

Using a Naïve Bayes method we would in general need discrete probabilities, but since we have continuous attributes in our dataset we will instead use distributions of these. There are several options for this, one of the most common ones is Gaussian Naïve Bayes, which assumes Gaussian (so normal) distribution of each attribute. Looking at distributions of a few attributes they all looked very similar (see Figure 5) and close to multinomial distribution so we decided to use Multinomial Naïve Bayes instead.

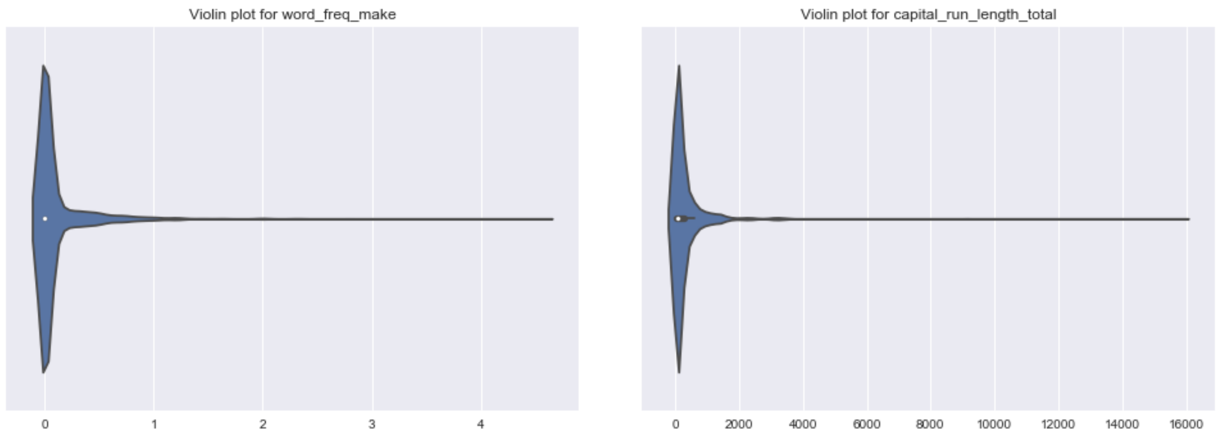

Figure 5: Distributions of attributes `word_freq_make` and `capital_run_length_total`.

There are a few restrictions to multinomial distributions though. For example, our values have to be non-negative so using normalized data from the previous model does not work, but it may work if we scale those values to a non-negative range then. We have therefore tried out a few preprocesing options to see which ones would give us best results. In Figure 6 for comparison of NB models built after different preprocessings. The preprocessings we test are no preprocessing, normalized data that is shifted to non-negative range, and finally normalized data that is then scaled on the interval between $[0, 1]$. The best results we achieve with **** normalized dataset which is then shifted to non-**

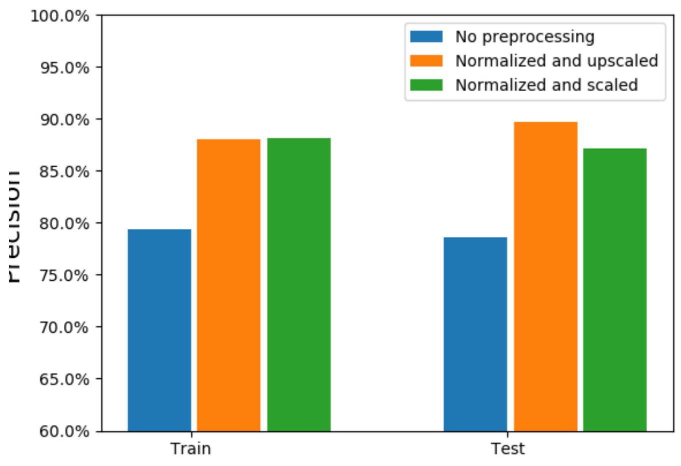

Figure 6: Comparison of precision from different preprocessings in MultinomialNB.

negative range **, so we use this dataset for hypertuning our model. 7. The only hypertuning we can do with MultinomialNB classifier is adjusting α value, which is introduced, similarly as m-estimate for regular NB, for handling 0s in probabilities that we get. If any of the probabilities is 0 we would get the whole product $= 0$ and this would hide all the information of the other probabilities. Hypertuning

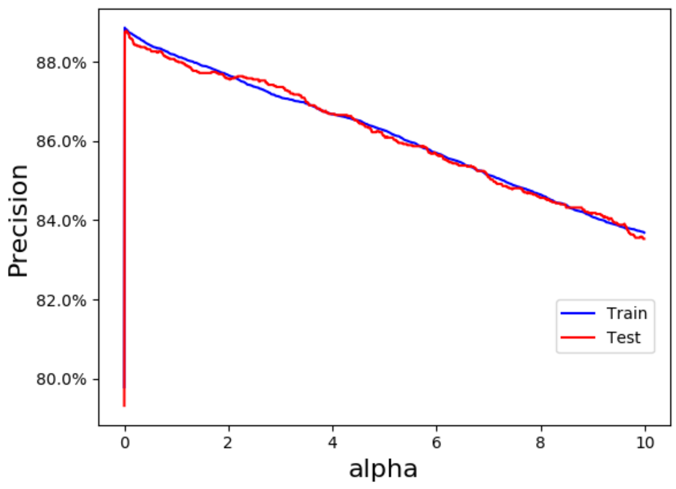

Figure 7: Hypertuning of α in MultinomialNB.

using 10-fold cross-validation gives us the results presented on Figure We figured *lidstone smoothing*[2] with α as small as possible gives best results, so we decided to make a final NB model with $\alpha = 0.01$.

Finally we compare the hypertuned NB model with a NB model that we would have gotten on no-preproced data with no hypertuning. We significantly increased precision both on train and test data.

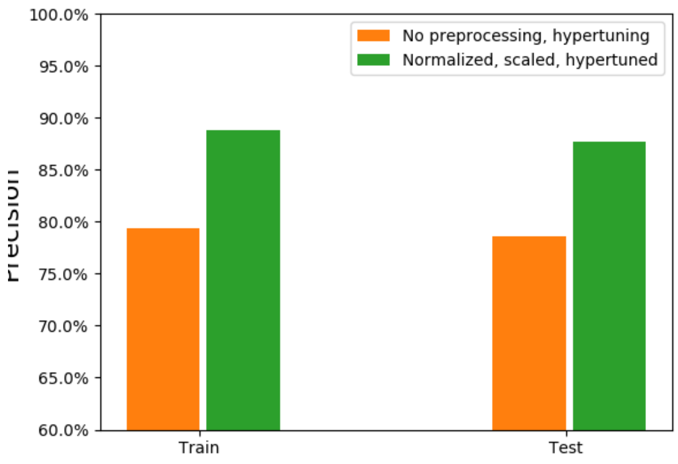

Figure 8: Comparisson of MultinomialNB models.

[2]Lidstone smoothing means $\alpha < 1$, see [12, 1.9.] for more information.

ANN classifier

– what is with ANN?

4 Evaluation of models

bla bla introduction text :D

– confusion matrix for each of them – precision for each of them – accuracy for each of them

5 Conclusions

– future topics: build a super model out of all three of ours

References

[1] I. Androutsopoulos, V. Metsis, G. Paliouras, Spam filtering with Naive Bayes – Which Naive Bayes?, in: Proceedings of the Third Conference on Email and AntiSpam (2006).

[2] R. Asadi, S. Abdul Kareem, M. Asadi, S. Asadi, An unsupervised feed forward neural network method for efficient clustering, The International Arab Journal of Information Technology 14(4) (2017) 436–441.

[3] G. V. Cormack, Email spam filtering: A systematic review, Foundation and Trends in Information Retrieval (2006) 1(4) 335–455.

[4] R. Deepa Lakshmi, N. Radha, Supervised learning approach for spam classification analysis using data mining tools, International Journal on Computer Science and Engineering 2(9) (2010) 2783–2789.

[5] J. Eberhardt, Bayesian Spam Detection, Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal 2(1) (2015).

[6] I. Idris, E-mail spam classification with artificial neural network and negative selection algorithm, International Journal of Computer Science & Communication Networks 1(3) (2011) 227–231.

[7] S. M. Lee, D. S. Kim, J. H. Kim, J. S. Park, Spam detection using feature selection and parameters optimization, in: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS) (2010).

[8] M. Hopkins, E. Reeber, G. Forman, J. Suermondt, SPAM e-mail database, Hewlett-Packard Labs, 1501 Page Mill Rd., Palo Alto, CA 94304, `https://archive.ics.uci.edu/ml/datasets/spambase`.

[9] D. Puniškis, R. Laurutis, R. Dirmeikis, An artificial neural nets for spam email recognition, Elektronika ir Elektrotechnika (Electronics and Electrical Engineering) 5(69) (2006) 73–76.

[10] S. A. Saab, N. Mitri, M. Awad, Ham or spam? A comparative study for some content-based classification Algorithms for email filtering, in: Proceedings of the 17th IEEE Mediterranean Electrotechnical Conference (2014).

[11] U. K. Sah, N. Parmar, An approach for malicious spam detection in email with comparison of different classifiers, International Research Journal of Engineering and Technology 4(8) (2017) 2238–2242.

[12] http://scikit-learn.org/stable/index.html

Appendix: code

STRUCTURE THE CODE into smaller segments, we will only have main training parts here (no prints, no confusion matrices codes etc). Only main models and their preprocesing, for results we will just analyze them with words

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.neighbors import KNeighborsClassifier
from  sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from matplotlib.ticker import FuncFormatter
from sklearn.model_selection import KFold
from  sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer
from sklearn.metrics import precision_score

from sklearn.metrics import confusion_matrix

"""
loading and preprocesing the data
"""

file = open("spambase.data")
data = np.loadtxt(file,delimiter=",")

X = data[:, 0:57]
y = data[:, 57]

dataframe = pd.DataFrame(data=X)

#apply normalization function to every attribute
dataframe_norm = dataframe.apply(lambda x: (x - np.mean(x)) / np.std(x))

"""
kNN model
"""
# explained_variance_ratio_ for n_components
EVC = []
for attribute in dataframe_norm:
    pca = PCA(n_components=attribute)
    pca.fit(dataframe_norm)
    EVC.append(pca.explained_variance_ratio_.sum())

# n_components hyperparameter tuning using KFold cross-validation
splits = 10
kf = KFold(n_splits = splits)
n_components_kFold_train = np.zeros(n_attributes-1)
n_components_kFold_test = np.zeros(n_attributes-1)
kFold_train_precision = np.zeros(n_attributes-1)
kFold_test_precision = np.zeros(n_attributes-1)
for i in range(1, n_attributes):
    X = pca(i, dataframe)
    # Using the same random_state ensures we do not contaminate our training data with
        our test data
    X_train , X_test , y_train , y_test = train_test_split(X, y, test_size=0.20 ,
        random_state=42)
    for cv_train_index, cv_test_index in kf.split(X_train):
        X_cv_train = [X_train_1[i] for i in cv_train_index]
        y_cv_train = [y_train_1[i] for i in cv_train_index]
        X_cv_test = [X_train_1[i] for i in cv_test_index]
        y_cv_test = [y_train_1[i] for i in cv_test_index]
        # k = 5 by defult
        kNN = KNeighborsClassifier()
        kNN.fit(X_cv_train, y_cv_train)
        kFold_train_precision[i-1] += precision_score(y_cv_train,
            kNN.predict(X_cv_train), average = 'micro') / splits
        kFold_test_precision[i-1] += precision_score(y_cv_test, kNN.predict(X_cv_test),
            average = 'micro') / splits

# n_neighbors cross-validation
n_components = 14
X = pca(n_components, dataframe_norm)
X_train , X_test , y_train , y_test = train_test_split(X, y, test_size=0.20 ,
    random_state=42)

k_tests = list(range(1, 100))
k_precision_train = np.zeros(len(k_tests))
k_precision_test = np.zeros(len(k_tests))

kf = KFold(n_splits = splits)
for cv_train_index, cv_test_index in kf.split(X_train):
        X_cv_train = [X_train[i] for i in cv_train_index]
        y_cv_train = [y_train[i] for i in cv_train_index]
        X_cv_test = [X_train[i] for i in cv_test_index]
        y_cv_test = [y_train[i] for i in cv_test_index]
        i=0
        for test in k_tests:
            kNN = KNeighborsClassifier(n_neighbors = test)
            kNN.fit(X_cv_train, y_cv_train)
             k_precision_train [i-1] += precision_score(y_cv_train,
                 kNN.predict(X_cv_train), average = 'micro') / splits
            k_precision_test [i-1] += precision_score(y_cv_test,
                kNN.predict(X_cv_test), average = 'micro') / splits
            i += 1

X_train , X_test , y_train , y_test = train_test_split(dataframe_norm, y,
    test_size=0.20 , random_state=42)

kNN = KNeighborsClassifier(n_neighbors = k)
kNN.fit(X_train, y_train)

kNN_precision_train_pre = precision_score(y_train, kNN.predict(X_train), average =
    'micro')
kNN_precision_test_pre = precision_score(y_test, kNN.predict(X_test), average =
    'micro')

k = 4
X = pca(n_components, dataframe_norm)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20 ,
    random_state=42)

kNN_precision_train_post = precision_score(y_train, kNN.predict(X_train), average =
    'micro')
kNN_precision_test_post = precision_score(y_test, kNN.predict(X_test), average =
    'micro')
```

```
kNN_precision_train_post = precision_score(y_train, kNN.predict(X_train), average =
    'micro')
precision_test_post = precision_score(y_test, kNN.predict(X_test), average = 'micro')


"""
NB model
"""

# Different preprocessings
dataframe = pd.DataFrame(data=X)
dataframe_norm = dataframe.apply(lambda x: (x - np.mean(x)) / np.std(x))
min_max_scaler = preprocessing.MinMaxScaler()
dataframe_norm_scaled = min_max_scaler.fit_transform(dataframe_norm)
dataframe_norm_upscaled = dataframe_norm.apply(lambda x: (x + abs(min(x))))

# No preprocessing of the data
X_train , X_test , y_train , y_test = train_test_split(dataframe, y, test_size=0.20 ,
    random_state=42)
mNB = GaussianNB()
mNB.fit(X_train, y_train)
precision_train = precision_score(y_train, mNB.predict(X_train), average = 'micro')
precision_test = precision_score(y_test, mNB.predict(X_test), average = 'micro')

# Upscaled normalized data with no negative attributes
X_train , X_test , y_train , y_test = train_test_split(dataframe_norm_upscaled, y,
    test_size=0.20 , random_state=42)
mNB_upscaled = GaussianNB()
mNB_upscaled.fit(X_train, y_train)
norm_upscaled_precision_train = precision_score(y_train,
    mNB_upscaled.predict(X_train), average = 'micro')
norm_upscaled_precision_test = precision_score(y_test, mNB_upscaled.predict(X_test),
    average = 'micro')


# Scaled normalized data
X_train , X_test , y_train , y_test = train_test_split(dataframe_norm_scaled, y,
    test_size=0.20 , random_state=42)
mNB_scaled = GaussianNB()
mNB_scaled.fit(X_train, y_train)
norm_scaled_precision_train = precision_score(y_train, mNB_scaled.predict(X_train),
    average = 'micro')
norm_scaled_precision_test = precision_score(y_test, mNB_scaled.predict(X_test),
    average = 'micro')


# kFold cross-validation for alpha
splits = 10
kf = KFold(n_splits = splits)

X_train , X_test , y_train , y_test = train_test_split(dataframe_norm_scaled, y,
    test_size=0.20 , random_state=42)

alpha_tests = np.arange(0, 10.0, 0.01)
alpha_cv_train = np.zeros(len(alpha_tests))
alpha_cv_test = np.zeros(len(alpha_tests))
for cv_train_index, cv_test_index in kf.split(X_train):
```

```
X_cv_train = [X_train[i] for i in cv_train_index]
y_cv_train = [y_train[i] for i in cv_train_index]
X_cv_test = [X_train[i] for i in cv_test_index]
y_cv_test = [y_train[i] for i in cv_test_index]
i = 0
for test in alpha_tests:
    mNB_cv = MultinomialNB(alpha=test)
    mNB_cv.fit(X_cv_train, y_cv_train)
    alpha_cv_train[i] += precision_score(y_cv_train, mNB_cv.predict(X_cv_train),
        average = 'micro') / splits
    alpha_cv_test[i] += precision_score(y_cv_test, mNB_cv.predict(X_cv_test),
        average = 'micro') / splits
    i += 1
```