# INF519 Machine Learning 2 / Lab 1

# Discriminative and Generative Models

Jae Yun JUN KIM*

October 6, 2021

**Remark**:

- Only groups of two or three people accepted (preferably three). Forbidden groups of fewer or larger number of people.
- No plagiarism. If plagiarism happens, both the "lender" and the "borrower" will have a zero.
- Do thoroughly all the demanded tasks.

## 1 Naive Bayes

**Sources:** Scikit-learn

### 1.1 Example 1: Bernoulli Naive Bayes

```
import numpy as np
from sklearn.naive_bayes import BernoulliNB

X = np.random.randint(2, size=(6, 100))
y = np.array([1, 2, 3, 4, 4, 5])
clf = BernoulliNB()
clf.fit(X, y)
for i in range(0,6):
    print(clf.predict(X[i:(i+1)]))
```

### 1.2 Example 2: Multinomial Naive Bayes

```
import numpy as np
from sklearn.naive_bayes import MultinomialNB

X = np.random.randint(5, size=(6, 100))
y = np.array([1, 2, 3, 4, 5, 6])
clf = MultinomialNB()
clf.fit(X, y)
```

*ECE Paris Graduate School of Engineering, 37 quai de Grenelle 75015 Paris, France; jae-yun.jun-kim@ece.fr

```
for i in range(0, 6):
    print(clf.predict(X[i:i+1]))
```

## 1.3 Example 3: Gaussian Naive Bayes

```
import numpy as np
from sklearn.naive_bayes import GaussianNB

X = np.array([[-1, -1],[-2, -1],[-3, -2],[1, 1],[2, 1],[3, 2]])
y = np.array([1, 1, 1, 2, 2, 2])

clf = GaussianNB()
clf.fit(X, y)

print(clf.predict([[-1, -0.8]]))

clf_pf = GaussianNB()
clf_pf.partial_fit(X, y, np.unique(y))
print(clf_pf.predict([[-1, -0.8]]))
```

## 1.4 Example 4: Filtering spam emails

Source: **https://appliedmachinelearning.wordpress.com/2017/01/23/email-spam-filter-python-scikit-learn/**

Download and unzip `ling-spam.zip` file from the course website. Then, type the following script and execute it.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 27 22:53:50 2017
@author: Abhijeet Singh
"""

import os
import numpy as np
from collections import Counter
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import confusion_matrix

def make_Dictionary(train_dir):
    emails = [os.path.join(train_dir, f) for f in os.listdir(train_dir)]
    all_words = []
    for mail in emails:
        with open(mail) as m:
            for i, line in enumerate(m):
                if i == 2:
                    words = line.split()
                    all_words += words

    dictionary = Counter(all_words)

    # list_to_remove = dictionary.keys()
```

```
    # for item in list_to_remove: # this works with python 2.x version
    for item in list(dictionary): # this works with python 3.x version
        if item.isalpha() == False:
            del dictionary[item]
        elif len(item) == 1:
            del dictionary[item]
    dictionary = dictionary.most_common(3000)
    return dictionary


def extract_features(mail_dir):
    files = [os.path.join(mail_dir, fi) for fi in os.listdir(mail_dir)]
    features_matrix = np.zeros((len(files), 3000))
    docID = 0
    for fil in files:
        with open(fil) as fi:
            for i, line in enumerate(fi):
                if i == 2:
                    words = line.split()
                    for word in words:
                        wordID = 0
                        for i, d in enumerate(dictionary):
                            if d[0] == word:
                                wordID = i
                                features_matrix[docID, wordID] = words.count(word)
            docID = docID + 1
    return features_matrix


# Create a dictionary of words with its frequency

train_dir = 'ling-spam\\train-mails'
dictionary = make_Dictionary(train_dir)

# Prepare feature vectors per training mail and its labels

train_labels = np.zeros(702) # y=0, ham
train_labels[351:701] = 1 # y=1, spam
train_matrix = extract_features(train_dir)

# Training Naive bayes classifier and its variants

model = MultinomialNB()

model.fit(train_matrix, train_labels)

# Test the unseen mails for Spam
```

```
test_dir = 'ling-spam\\test-mails'
test_matrix = extract_features(test_dir)
test_labels = np.zeros(260)
test_labels[130:260] = 1

result = model.predict(test_matrix)

print(confusion_matrix(test_labels, result))
```

## 1.5 sklearn.naive_bayes.BernoulliNB

```
class sklearn.naive_bayes.BernoulliNB(alpha=1.0, binarize=0.0,
fit_prior=True, class_prior=None)
```

Naive Bayes classifier for multivariate Bernoulli models.

Like MultinomialNB, this classifier is suitable for discrete data. The difference is that while MultinomialNB works with occurrence counts, BernoulliNB is designed for binary/boolean features.

### 1.5.1 Parameters

**alpha**: float, optional (default:1.0)

Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

**binarize**: float or None, optional (default=0.0)

Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.

**fit_prior**: boolean, optional (default=True)

Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

**class_prior**: array-like, size=[n_classes,], optional (default=None)

Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

### 1.5.2 Attributes

**class_log_prior_**: array, shape = [n_classes]

Log probability of each class (smoothed).

**feature_log_prob_**: array, shape = [n_classes, n_features]

Empirical log probability of features given a class, P(x_i—y).

**class_count_** : array, shape = [n_classes]

Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

**feature_count_**: array, shape = [n_classes, n_features]

Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

### 1.5.3 Methods

```
__init__(alpha=1.0, binarize=0.0, fit_prior=True, class_prior=None)
```

```
fit(X, y, sample_weight=None)
```

Fit Naive Bayes classifier according to X, y.

**Parameters**:

**X**: {array-like}, shape=[n_samples, n_features]

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

**y**: array-like, shape=[n_samples]

Target values.

**sample_weight**: array-like, shape=[n_samples], (default=None)

Weights applied to individual samples (1. for unweighted). Higher weights force the classifier to put more emphasis on these points.

**Returns**: **self**: object

Returns `self`.

```
get_params(deep=True)
```

Get parameters for this estimator.
**Params**:

**deep**: boolean, optional

If True, will return the parameters for this estimator and contained `subobjects` that are estimators.

**Returns**:

**params**: mapping of string to any

Parameter names mapped to their values.

```
partial_fit(X, y, classes=None, sample_weight=None)
```

Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.
**Parameters**:

**X**: {array-like, sparse matrix}, shape = [n_samples, n_features]

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

**y**: array-like, shape=[n_samples]

Target values.

**classes**: array-like, shape=[n_classes] (default=None)

List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

**sample_weight**: array-like, shape=[n_samples], (default=None)

Weights applied to individual samples (1. for unweighted). Higher weights force the classifier to put more emphasis on these points.

**Returns**: **self**: object

Returns `self`.

```
predict(X)
```

Perform classification on samples in $X$.

**Parameters**:

**X**: array-like, sparse matrix, shape=[n_samples, n_features]

**Returns**:

**C**: array, shape=[n_samples]

Class labels for samples in X.

```
predict_log_proba(X)
```

Return log-probability estimates for the test vector X.

**Parameters**:

**X**: array-like, shape=[n_samples, n_features]

**Returns**:

**C**: array-like, shape=[n_samples,n_classes]

Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
predict_proba(X)
```

Return probability estimates for the test vector X.

**Parameters**:

**X**: array-like, shape=[n_samples, n_features]

**Returns**:

**C**: array-like, shape=[n_samples,n_classes]

Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
score(X,y,sample_weight=None)
```

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**:

**X**: array-like, shape = [n_samples, n_features]

Test samples

**y**: array-like, shape = [n_samples] or , [n_samples,n_features]

True labels for $X$.

**sample_weight**: array-like, shape = [n_samples], optional

Sample weights

**Returns**:

**score**: float

Mean accuracy of `self.predict(X)` w.r.t. $y$.

```
set_params(**params)
```

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form $< component > \_\_ < parameter >$ so that it is possible to update each component of a nested object.

**Returns**:

**self**: object

Returns self.

## 1.6 sklearn.naive_bayes.MultinomialNB

```
class sklearn.naive_bayes.MultinomialNB(alpha=1.0, binarize=0.0,
fit_prior=True, class_prior=None)
```

Naive Bayes classifier for multinomial models.

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf (term frequency-inverse document frequency) may also work.

### 1.6.1 Parameters

**alpha**: float, optional (default:1.0)

Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

**binarize** : float or None, optional (default=0.0)

Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.

**fit_prior**: boolean, optional (default=True)

Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

**class_prior**: array-like, size=[n_classes,], optional (default=None)

Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

### 1.6.2 Attributes

**class_log_prior_**: array, shape = [n_classes]

Log probability of each class (smoothed).

**intercept_**: property

Mirrors class_log_prior_ for interpreting MultinomialNB as a linear model.

**feature_log_prob_**: array, shape = [n_classes, n_features]

Empirical log probability of features given a class, P(x_i—y).

**coef_**: property

Mirrors feature_log_prob_ for interpreting MultinomialNB as a linear model.

**class_count_** : array, shape = [n_classes]

Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

**feature_count_**: array, shape = [n_classes, n_features]

Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

### 1.6.3 Methods

```
__init__(alpha=1.0, binarize=0.0, fit_prior=True, class_prior=None)
```

```
fit(X, y, sample_weight=None)
```

Fit Naive Bayes classifier according to X, y.

**Parameters**:

**X**: {array-like, sparse matrix}, shape =[n_samples, n_features]

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

**y**: array-like, shape=[n_samples]

Target values.

**sample_weight**: array-like, shape=[n_samples], (default=None)

Weights applied to individual samples (1. for unweighted). Higher weights force the classifier to put more emphasis on these points.

**Returns**: **self**: object

Returns `self`.

```
get_params(deep=True)
```

Get parameters for this estimator.
**Params**:

**deep**: boolean, optional

If True, will return the parameters for this estimator and contained `subobjects` that are estimators.

**Returns**:

**params**: mapping of string to any

Parameter names mapped to their values.

```
partial_fit(X, y, classes=None, sample_weight=None)
```

Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

**Parameters**:

**X**: {array-like, sparse matrix}, shape = [n_samples, n_features]

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

**y**: array-like, shape=[n_samples]

Target values.

**classes**: array-like, shape=[n_classes] (default=None)

List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

**sample_weight**: array-like, shape=[n_samples], (default=None)

Weights applied to individual samples (1. for unweighted). Higher weights force the classifier to put more emphasis on these points.

**Returns**: **self**: object

Returns `self`.

```
predict(X)
```

Perform classification on samples in $X$.

**Parameters**:

**X**: array-like, sparse matrix, shape=[n_samples, n_features]

**Returns**:

**C**: array, shape=[n_samples]

Class labels for samples in X.

```
predict_log_proba(X)
```

Return log-probability estimates for the test vector X.

**Parameters**:

**X**: array-like, shape=[n_samples, n_features]

**Returns**:

**C**: array-like, shape=[n_samples,n_classes]

Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
predict_proba(X)
```

Return probability estimates for the test vector X.

**Parameters**:

**X**: array-like, shape=[n_samples, n_features]

**Returns**:

**C**: array-like, shape=[n_samples,n_classes]

Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
score(X,y,sample_weight=None)
```

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**:

**X**: array-like, shape = [n_samples, n_features]

Test samples

**y**: array-like, shape = [n_samples] or , [n_samples,n_features]

True labels for $X$.

**sample_weight**: array-like, shape = [n_samples], optional

Sample weights

**Returns**:

**score**: float

Mean accuracy of `self.predict(X)` w.r.t. $y$.

```
set_params(**params)
```

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form $< component > \_\_ < parameter >$ so that it is possible to update each component of a nested object.

**Returns**:

**self**: object

Returns self.

## 1.7 sklearn.naive_bayes.GaussianNB

```
class sklearn.naive_bayes.GaussianNB(priors=None)
```

Gaussian Naive Bayes (GaussianNB).

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf (term frequency-inverse document frequency) may also work.

### 1.7.1 Parameters

**priors**: array-like, shape=[n_classes]

Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

### 1.7.2 Attributes

**class_prior_**: array, shape = [n_classes]

Probability of each class (smoothed).

**class_count_** : array, shape = [n_classes]

Number of training samples observed in each class.

**theta_**: array, shape = [n_classes, n_features]

Mean of each feature per class.

**sigma_**: array, shape = [n_classes, n_features]

Variance of each feature per class.

### 1.7.3 Methods

```
__init__(priors=None)
```

```
fit(X, y, sample_weight=None)
```

Fit Gaussian Naive Bayes according to X and y.

**Parameters**:

**X**: {array-like, sparse matrix}, shape =[n_samples, n_features]

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

**y**: array-like, shape=[n_samples]

Target values.

**sample_weight**: array-like, shape=[n_samples], (default=None)

Weights applied to individual samples (1. for unweighted). Higher weights force the classifier to put more emphasis on these points.

**Returns**: **self**: object

Returns `self`.

```
get_params(deep=True)
```

Get parameters for this estimator.
**Params**:

**deep**: boolean, optional

If True, will return the parameters for this estimator and contained `subobjects` that are estimators.

**Returns**:

**params**: mapping of string to any

Parameter names mapped to their values.

```
partial_fit(X, y, classes=None, sample_weight=None)
```

Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.
**Parameters**:

**X**: {array-like, sparse matrix}, shape = [n_samples, n_features]

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

**y**: array-like, shape=[n_samples]

Target values.

**classes**: array-like, shape=[n_classes] (default=None)

List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

**sample_weight**: array-like, shape=[n_samples], (default=None)

Weights applied to individual samples (1. for unweighted). Higher weights force the classifier to put more emphasis on these points.

**Returns**: **self**: object

Returns `self`.

```
predict(X)
```

Perform classification on samples in $X$.

**Parameters**:

**X**: array-like, sparse matrix, shape=[n_samples, n_features]

**Returns**:

**C**: array, shape=[n_samples]

Class labels for samples in X.

```
predict_log_proba(X)
```

Return log-probability estimates for the test vector X.

**Parameters**:

**X**: array-like, shape=[n_samples, n_features]

**Returns**:

**C**: array-like, shape=[n_samples,n_classes]

Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
predict_proba(X)
```

Return probability estimates for the test vector X.

**Parameters**:

**X**: array-like, shape=[n_samples, n_features]

**Returns**:

**C**: array-like, shape=[n_samples,n_classes]

Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
score(X,y,sample_weight=None)
```

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**:

**X**: array-like, shape = [n_samples, n_features]

Test samples

**y**: array-like, shape = [n_samples] or , [n_samples,n_features]

True labels for $X$.

**sample_weight**: array-like, shape = [n_samples], optional

Sample weights

**Returns**:

**score**: float

Mean accuracy of `self.predict(X)` w.r.t. $y$.

```
set_params(**params)
```

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form $<component>\_\_<parameter>$ so that it is possible to update each component of a nested object.

**Returns**:

**self**: object

Returns self.