CSE 454 Data Mining
Homework 4 Report

Ahmed Semih Özmekik
171044039

January 17, 2021

Abstract

Detailed explanation of design choices along with the experimental results in the homework.

1 Implementing the Naive Bayesian Classification

Throughout this section, we will explain our work by giving references from our code, following the implementation steps in the book.

1.1 Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, $X = (x_1, x_2, ..., x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, ..., A_n$.

Let us state their equivalents to show them in the dataset we have chosen.

```
data = load_iris()

X, Y, column_names = data['data'], data['target'], data['feature_names']
X = pd.DataFrame(X, columns=column_names)

print(X)
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 ...               ...                ...               ...
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8
```

Each column represents the feature in an *X* feature set.

$$P(\boldsymbol{X}|C_i) = \prod_{k=1}^{n} P(x_k|C_i)$$
$$= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i).$$

Figure 1

(a) If A_k is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D having the value x_k for A_k, divided by $|C_{i,D}|$, the number of tuples of class C_i in D.

(b) If A_k is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation σ, defined by

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}).$$

Figure 2

1.2 Given data sets with many attributes, it would be extremely computationally expensive to compute P(X—C i). To reduce computation in evaluating P(X—C i), the naıve assumption of class-conditional independence is made. This presumes that the attributes' values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

Naiveness of this naive bayes comes from this assumption. And regarding to this rule, we will implement our algorithm pretty straight forward.

But of course, we need to keep in mind that, calculation will change regarding to our dataset being categorical or continues-valued. This phenomenon is mentioned in the book:

We implemented our algorithm so that it is suitable for calculating data in both cases separately.

Also, we have applied laplace (add-1) smoothing to count values. Thus, the results were more successful and appropriate.

1.3 Naive Bayes Algorithm

```
def nb(X_train, Y_train, X_test, categorical=False):
    if categorical:
        # count dist.
        df = X_train.groupby(Y_train)
        count = [df[c].value_counts() for c in X_train.columns]

        distinct_cols = [len(X_train[col].unique()) for col in X_train.columns]
    else:
        # collect means and standart deviations for gaussian distribution.
        means = X_train.groupby(Y_train).apply(np.mean)
        stds = X_train.groupby(Y_train).apply(np.std)

    # class distribution
    PCi = X_train.groupby(Y_train).apply(lambda x: len(x)) / X_train.shape[0]

    Y_pred = []
    for i in range(X_test.shape[0]):  # row iterate
        p = {}

        for Ci in np.unique(Y_train):  # class iterate
            p[Ci] = PCi.iloc[Ci]
            for j, val in enumerate(X_test.iloc[i]):  # column iterate
                if categorical:
                    # applying laplace smooth 1.
                    V = count[j][Ci].sum() + distinct_cols[j]
                    p[Ci] *= ((count[j][Ci, val] + 1)
                              if (Ci, val) in count[j] else 1) / V
                else:
                    p[Ci] *= g(val, means.iloc[Ci, j], stds.iloc[Ci, j])

        Y_pred.append(pd.Series(p).values.argmax())
    return Y_pred
```

2 Answers

F1 Scores

We will show that our algorithm works for both categorical and numerical datasets. Below are the demo functions we have prepared for demonstration.

For numerical we will use the *iris* dataset in the *scikit* library. For Categorical, we will use the following *tennis* dataset, which we found on the internet:

```
outlook,temp,humidity,windy,play
sunny,hot,high,false,no
sunny,hot,high,true,no
overcast,hot,high,false,yes
rainy,mild,high,false,yes
rainy,cool,normal,false,yes
rainy,cool,normal,true,no
```

```
overcast,cool,normal,true,yes
sunny,mild,high,false,no
sunny,cool,normal,false,yes
rainy,mild,normal,false,yes
sunny,mild,normal,true,yes
overcast,mild,high,true,yes
overcast,hot,normal,false,yes
rainy,mild,high,true,no
```

To show that the algorithm works correctly, we use a naive bayes model from the *scikit* library again. And we show the results together, you can also see that we are k-cross validating.

```
def demo_numerical(pca=False, process=None):
    data = load_iris()

    X, Y, column_names = data['data'], data['target'], data['feature_names']
    X = pd.DataFrame(X, columns=column_names)

    if process == "pca":
        print(X.shape)
        pca = PCA(n_components=4)
        X = pd.DataFrame(data=pca.fit_transform(X))
        print(X.shape)
    elif process == "lda":
        print(X.shape)
        lda = LinearDiscriminantAnalysis()
        X = pd.DataFrame(data=lda.fit_transform(X, Y))
        print(X.shape)
    elif process == "filter":
        X = filter_fs(X, Y)
    elif process == "wrapper":
        X = wrapper_fs(X, Y)


    cv = KFold(n_splits=5, shuffle=True, random_state=0)

    my_score = []
    true_score = []
    for train_index, test_index in cv.split(X):
        X_train, X_test, Y_train, Y_test = X.iloc[train_index], X.iloc[
                                                test_index], Y[train_index], Y[
                                                test_index]


        Y_pred = nb(X_train, Y_train, X_test)
        my_score.append(f1_score(Y_test, Y_pred))
        true_score.append(test(X_train, Y_train, X_test, Y_test))

    print(np.mean(my_score))
    print(np.mean(true_score))


def demo_categorical():
```

```
    df = pd.read_csv('data/tennis.csv')

    for col in df.columns:
        df[col] = df[col].astype('category').cat.codes

    X = df.drop('play', axis=1)
    Y = df['play']
    Y = np.array(Y)

    cv = KFold(n_splits=5, shuffle=True, random_state=0)

    my_score = []
    true_score = []
    for train_index, test_index in cv.split(X):
        X_train, X_test, Y_train, Y_test = X.iloc[train_index], X.iloc[
                                                test_index], Y[train_index], Y[
                                                test_index]

        Y_pred = nb(X_train, Y_train, X_test, categorical=True)
        my_score.append(f1_score(Y_test, Y_pred))
        true_score.append(
            test(X_train, Y_train, X_test, Y_test, categorical=True))

    print(np.mean(my_score))
    print(np.mean(true_score))
```

Here is the f1 score output after calling this two function:

```
0.4766666666666667
0.4766666666666667
0.2833333333333333
0.2833333333333333
```

Now let's go back to the numerical dataset only to answer other questions.

We will obtain test results in four different ways for the same dataset, then see and interpret the results for four.

```
demo_numerical()
demo_numerical(process="filter")
demo_numerical(process="wrapper")
demo_numerical(process="pca")
demo_numerical(process="lda")
```

```
0.4766666666666667
(150, 4)
(150, 2)
0.4833333333333333
(150, 4)
(150, 3)
```

```
0.4833333333333333
(150, 4)
(150, 2)
0.44333333333333336
(150, 4)
(150, 2)
0.49000000000000005
```

When we look at the outputs respectively, we can list the success as follows, $F1_{lda} > F1_{filter} > F1_{wrapper} > F1_{normal} > F1_{pca}$

2.1 Which technique has given better results in terms of f1 score? (filter feature selection or wrapper feature selection) Was it expected?

The results look the same in this dataset, let's try it on a data set with a larger feature. We try this by switching from *iris* data set to *breast cancer* data set.

```
0.4657584226051855
(569, 30)
(569, 14)
0.4666278528178854
(569, 30)
(569, 14)
0.46399627387051695
```

$F1_{filter} > F1_{normal} > F1_{wrapper}$
The filter gave better results. But they are quite similar, and this was the expected result, since they almost selected the same features from dataset.

2.2 Which technique has given better results in terms of f1 score? (PCA or LDA)? Was it expected?

```
0.4657584226051855
(569, 30)
(569, 2)
0.4508150908244062
(569, 30)
(569, 1)
0.4868265797236454
```

$F1_{lda} > F1_{normal} > F1_{pca}$
LDA was better. Because PCA performs better in case where number of samples per class is less. Whereas LDA works better with large dataset having multiple classes[1]

[1]Class separability is an important factor while reducing dimensionality.

2.3 Have the filter feature selection and wrapper feature selection technique given similar set of features? Which attributes are different?

When we examine some features, for example the 4th column in the filter and the 1st column in the wrapper are similar, or the 5th column in the filter and the 5th column in the wrapper are exactly the same. It is possible to see that some similar features like this one are selected for both algorithms in a similar way. The answer to the question is yes.

Afterwards, we see that the remaining feature selections are made differently.

```
        0      1       2       3        4       5      6       7       8      9      10      11       12      13
0    17.99  10.38  122.80  1001.0  0.30010  1.0950  8.589  153.40  25.380  17.33  184.60  2019.0  0.66560  0.7119
1    20.57  17.77  132.90  1326.0  0.08690  0.5435  3.398   74.08  24.990  23.41  158.80  1956.0  0.18660  0.2416
2    19.69  21.25  130.00  1203.0  0.19740  0.7456  4.585   94.03  23.570  25.53  152.50  1709.0  0.42450  0.4504
3    11.42  20.38   77.58   386.1  0.24140  0.4956  3.445   27.23  14.910  26.50   98.87   567.7  0.86630  0.6869
4    20.29  14.34  135.10  1297.0  0.19800  0.7572  5.438   94.44  22.540  16.67  152.20  1575.0  0.20500  0.4000
..     ...    ...     ...     ...      ...     ...    ...     ...     ...    ...     ...     ...      ...     ...
564  21.56  22.39  142.00  1479.0  0.24390  1.1760  7.673  158.70  25.450  26.40  166.10  2027.0  0.21130  0.4107
565  20.13  28.25  131.20  1261.0  0.14400  0.7655  5.203   99.04  23.690  38.25  155.00  1731.0  0.19220  0.3215
566  16.60  28.08  108.30   858.1  0.09251  0.4564  3.425   48.55  18.980  34.12  126.70  1124.0  0.30940  0.3403
567  20.60  29.33  140.10  1265.0  0.35140  0.7260  5.772   86.22  25.740  39.42  184.60  1821.0  0.86810  0.9387
568   7.76  24.54   47.92   181.0  0.00000  0.3857  2.548   19.15   9.456  30.37   59.16   268.6  0.06444  0.0000
[569 rows x 14 columns]
     mean compactness  mean concave points  radius error  ...  worst concave points  worst symmetry  worst fractal dimension
0             0.27760              0.14710        1.0950  ...                0.2654          0.4601                  0.11890
1             0.07864              0.07017        0.5435  ...                0.1860          0.2750                  0.08902
2             0.15990              0.12790        0.7456  ...                0.2430          0.3613                  0.08758
3             0.28390              0.10520        0.4956  ...                0.2575          0.6638                  0.17300
4             0.13280              0.10430        0.7572  ...                0.1625          0.2364                  0.07678
..                ...                  ...           ...  ...                   ...             ...                      ...
564           0.11590              0.13890        1.1760  ...                0.2216          0.2060                  0.07115
565           0.10340              0.09791        0.7655  ...                0.1628          0.2572                  0.06637
566           0.10230              0.05302        0.4564  ...                0.1418          0.2218                  0.07820
567           0.27700              0.15200        0.7260  ...                0.2650          0.4087                  0.12400
568           0.04362              0.00000        0.3857  ...                0.0000          0.2871                  0.07039
[569 rows x 14 columns]
```

When we examine some features, for example the 4th column in the filter and the 1st column in the wrapper are similar, or the 5th column in the filter and the 5th column in the wrapper are exactly the same. It is possible to see that some similar features like this one are selected for both algorithms in a similar way. The answer to the question is yes.

Afterwards, we see that the remaining feature selections are made differently.

2.4 Which technique has given better results? (feature selection or dimension reduction)? Was it expected?

I got the better results in LDA, which is a dimension reduction technique.

Feature selection removes features from our data while dimensionality reduction performs a linear combination to generate a new space with few features.

As a result we can say that: we were dealing with a classification problem. We should always first perform feature selection to discard irrelevant variables, and then, if necessary, run PCA in order to further decrease the number of variables. In this case, regarding to our dataset with small features (in both of them), decreasing the number of features performed better results.