George Mason University

Learning From Data   Fall 2024

Computer Exercise #1

Assigned: August 22, 2024
Due Date: September 05, 2024 (Midnight)

This computer exercise is concerned with generative models in general, and kernel density estimation in particular. Most of the code needed for these exercises is available in a Jupyter notebook that is distributed with this assignment. Your lab report should be submitted in a **separate** Jupyter notebook with full documentation of what you have done (and why) along with answers to or discussion about the questions that are given.

Since this may be your first time working with python and scikit-*learn*, this write-up will be more detailed than those in the future in order to get you up to speed with scikit-*learn* and Jupyter notebooks.

The due date for this is Thursday, September 05 at midnight. Absolutely **no late submissions** will be accepted after this date for any reason or any excuse. It is important for you to keep you and finish this assignment and move on to the next. If you have not finished by the deadline, submit the work you have done for partial credit. The purpose of this lab is to help you learn how to put theory into practice, to see how KDE works, and to explore its strengths and understand its limitations.

Purpose

The purpose of this and all future computer exercises is to gain experience in working with data, to learn how to design and train machine learning systems, and to discover and understand the properties and limitations of each ML algorithm. You will find that what you learn in class does not always translate easily into practice, and you will discover the importance of understanding what each hyperparameter does and the importance of setting each one correctly. You should view each exercise as a playground to put theory into practice and to explore new ideas. You should consider these exercises as open-ended projects, and avoid using them as scripted assignments. It is highly recommended that you read the documentation on each class you import and each function or method that you use. Understand what the hyperparameters are, and what the default values are. Be aware that the default values are not necessarily the ones that you will want to use.

In previous years, the course TA spent many hours grading these exercises, and I spent many hours and sometimes days writing up solutions that often exceeded twenty pages in length. While I myself continue to learn as I write these experiments, explore new ideas, ask myself questions and document my findings, many indicators have shown that this was not a shared experience and that many students did not even spend time reading the solutions. As a result, again this year the solutions will be concise, indicating a few of the things you should have discovered, and possibly highlighting some interesting things that I discovered as I worked through the exercises.

Your opportunity to gain a good, solid understanding of ML is through these exercises.

Grading

For this computer exercise, you are provided with four files. The first one is the assignement that you are now reading. The second is a Jupyter notebook that contains much of the code needed to complete the assignment. The third is a dataset that you will use to learn a naive Bayes classifier, and the fourth is a Jupyter notebook that you will submit for grading. This notebook will replicate the questions and assignments that you are to write-up. In your submission, you will be graded based on several factors, including:

1. The completeness of your work - did you answer all questions and do all of the experiments?

2. The quality of the write-up. Is your work clearly explained and any code in your report fully documented?

3. Was your report concise with just the right amount of code and figures? Points will be deducted if there is unnecessary code in your notebook that serves no purpose to illustrate a point or explain a result, and points will be deducted for padding the notebook with too many unnecessary plots or figures. You should plan on having your report be no more than 2-3 pages when exported to a pdf file (this is not to say that you are limited to 3 pages)

4. Extra points will be awarded for any experiments you performed beyond what was asked. An example would be exploring other kernels and comparing what them to what you find with a Gaussian kernel.

Computer Exercise 1.1 (Generative Models):

Kernel Density Estimation (KDE) is a non-parametric density estimation technique that can be used to design a classifier that approximates a **Bayes classifier**. The approach is to estimate the conditional densities $p(\boldsymbol{x}|y=k)$ from a set of data samples from class k, and then use these conditional densities along with estimates of the prior probabilities, $\Pr\{y=k\}$, in the Bayes classification rule,

$$k = \arg\max_i p(\boldsymbol{x}|y=i)\Pr\{y=i\}$$

Once these conditional densities have been estimated, they may also be used to **generate** new data by random sampling of the density $p(\boldsymbol{x}|y=k)$. In this exercise, you will be using either the `digits` dataset that consists of 1,797 images of handwritten digits with grayscale values between 0 and 16, were each image is 8×8 p ixels in size, or the `MNIST` dataset of 7,000 eight-bit images that are 24×24 pixels in size.

To illustrate the basic concept of how it is possible to generate new digits, note that the 8×8 array of grayscale values of an image represent a **feature vector** of length 64. It is assumed that the handwritten digits, zero through nine, have feature vectors that are generally different from each other, and that the ten digits will be clustered in different regions of the 64-dimensional feature space. In other words, a zero looks different than a nine and a two looks different from a seven, but all zeros are similar to each other. Therefore, if a single probability density function $p(\boldsymbol{x})$ is estimated using the entire digit dataset (all ten digits), then there should be ten (not necessarily well-defined) clusters that are scattered throughout the 64-dimensional feature space, one for each digit. A two-dimensional version of this concept is shown in Figure 1 where data samples from ten classes (digits) are seen to fall roughly into ten distinct clusters.

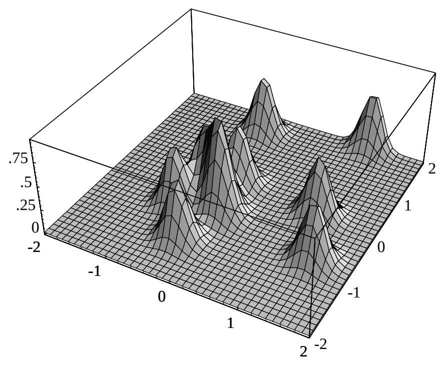


Figure 1: Illustration of how a probability density function may be used to generate random digits.

Alternatively, the dataset could be divided into ten subsets, one for each digit, and a probability density $p(\boldsymbol{x}|y=k)$ estimated for each digit. In this case, sampling $p(\boldsymbol{x}|y=k)$ will generate a new digit belonging to class $y=k$. Recall that the relationship between $p(\boldsymbol{x})$ and $p(\boldsymbol{x}|y=k)$ is given by

$$p(\boldsymbol{x}) = \sum_{k=1}^{10} p(\boldsymbol{x}|y=k)\Pr\{y=k\} \tag{1}$$

where $\Pr\{y=k\}$ are the priors that can be estimated from the relative frequencies

$$\Pr\{y=k\} = \frac{n_k}{N}$$

where n_k is the number of digits with the label k in the dataset and N is the total number of samples in the dataset.

Question for Discussion
If the goal is to randomly generate new digits, which approach is preferable:

(a) Estimate $p(\boldsymbol{x})$ from the entire dataset and then randomly select samples from this density, or

(b) Estimate $p(\boldsymbol{x}|y = k)$ for each digit, and then select an integer k at random and draw a sample from $p(\boldsymbol{x}|y = k)$?

If it does not make any difference which approach is used, explain why.

In the following, we are going to use KDE to estimate $p(\boldsymbol{x})$ using the entire dataset and generate new digits by randomly drawing samples from $p(\boldsymbol{x})$.

(a) To begin, load `NumPy`, a library that adds support for multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions on these arrays. To make calls to the functions in `NumPy` functions a bit more concise, we will load this library and call it `np`,

```
import numpy as np
```

This library will be used in most, if not all, computer exercises that you will be given.

The next library to load is `matplotlib`, which is a library of functions for plotting and visualization with plotting functions that have a syntax similar to MATLAB. The specific function we want for plotting in this exercise is `pyplot`, which will be referenced by `plt`,

```
import matplotlib.pyplot as plt
```

This plotting function will be used frequently.

Now you will need to import some functions from several different libraries from scikit-*learn*. The first is `load_digits` in the `datasets` library, the second is `KernelDensity` from the `neighbors` library, and the third is `PCA` from the `decomposition` library:

```
from sklearn.datasets import load_digits
from sklearn.neighbors import KernelDensity
from sklearn.decomposition import PCA
```

What these functions are and how they are used are discussed below, but it is important that you look at the documentation to see what the various parameters, attributes, and methods are for each of them.

(b) Having imported all of the libraries that are needed, the next thing to do is to load the dataset:[1]

```
# load the data
digits = load_digits()
```

The data in `digits` is a `bunch`, which is a dictionary-like object with a number of **attributes** (think of these as dictionary names given to some data). Some of the attributes in this bunch are:

(a) `data`: A flattened data matrix for each image, containing 1,797 arrays of length 64.

[1]Note that a hash, #, indicates that everything that follows until the next new line is a comment. Documentation of your code is highly recommended.

(b) `target`: The classification target values for each image.

(c) `target_names`: The names of the target classes, i.e., $0, 1, 2, \ldots, 9$.

(d) `images`: The image data, 1797 images of size 8×8.

(e) `DESCR`: The full description of the dataset.

The data in these attributes may be accessed by name. For example, the images may be accessed and put into a $1797 \times 8 \times 8$ array named `images` as follows,

```
images = digits.images
```

Typing `images.shape` will give the shape (dimension) of the array:

```
(1797, 8, 8)
```

i.e., 1,797 images of size 8×8, and with the command

```
print(images[1])
```

the 8×8 array of pixel values for the second image in the dataset,[2] which is the number 1, will be printed:

```
[[ 0.  0.  0. 12. 13.  5.  0.  0.]
 [ 0.  0.  0. 11. 16.  9.  0.  0.]
 [ 0.  0.  3. 15. 16.  6.  0.  0.]
 [ 0.  7. 15. 16. 16.  2.  0.  0.]
 [ 0.  0.  1. 16. 16.  3.  0.  0.]
 [ 0.  0.  1. 16. 16.  6.  0.  0.]
 [ 0.  0.  1. 16. 16.  6.  0.  0.]
 [ 0.  0.  0. 11. 16. 10.  0.  0.]]
```

If you type `digits` you will see the structure of the bunch.

MNIST

Unless you are limited in computation resources, it is recommended that you work with the `MNIST` dataset, instead of the `digits` dataset, since they are of much higher resolution. This dataset consists of 6,000 8-bit images of size 28×28, so there are 784 features compared to 64 in the `digits` dataset. Some sample images are shown in the following figure.

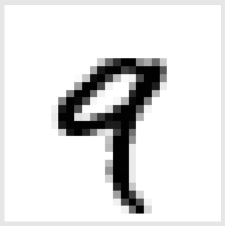
The `MNIST` dataset can be loaded as follows:

```
from sklearn.datasets import fetch_openml
# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
# Extract features and labels
X, y = mnist["data"], mnist["target"]
# Convert the DataFrame to a NumPy array
X = X.to_numpy()
```

[2]Why is it the second image and not the first?

The images stored in `X` are vectors (not arrays) of length 784, and the target values (classes) of each image are stored in the vector `y`. To view the images, the vectors in `X` must be reshaped into 28×28 arrays,

```
Image = X.reshape(-1, 28, 28)
plt.imshow(Image[4],cmap='binary')   # Show the fifth image in the dataset
```



(c) To create a generative model for images, it is necessary for them to be represented in vector form and not as a matrix. For the `MNIST` dataset, the data `X` is read in as vector so it may be used directly (do not reshape it), but if working with the `digits` dataset, it will be necessary to use the flattened data matrix

```
X = digits.data
```

or load `digits.images` and flatten the images. With the command `X.shape` the output should be `(1797, 64)` for the `digits` data and with the command `print(X[1])` a vector of 64 integers between 0 and 16 will be printed,

```
[ 0.  0.  0. 12. 13.  5.  0.  0.  0.  0.  0. 11. 16.  9.  0.  0.  0.  0.
  3. 15. 16.  6.  0.  0.  0.  7. 15. 16. 16.  2.  0.  0.  0.  0.  1. 16.
 16.  3.  0.  0.  0.  0.  1. 16. 16.  6.  0.  0.  0.  0.  1. 16. 16.  6.
  0.  0.  0.  0.  0. 11. 16. 10.  0.  0.]
```

If using the `MNIST` dataset, `X.shape` will give `(70000, 784)` and printing the values for one of the images will list 784 integers with values between 0 and 255.

In the following, the `MNIST` dataset will be used, so the results may be different if using the `digits` dataset.

(d) The model that is used with KDE is

$$p(\boldsymbol{x}) = \sum_{k=1}^{N} K(\boldsymbol{x} - \boldsymbol{x}_k; \boldsymbol{H})$$

where $\boldsymbol{H}$ is a bandwidth matrix, $K(\cdot)$ is the kernel, and N is the number of samples in the dataset. If a Gaussian kernel is used, which is often the kernel of choice, the bandwidth parameter is the variance of the Gaussian (in the case of 1-d kernel density estimation), or the covariance matrix $\boldsymbol{\Sigma}$ in the case of multivariate density estimation. For the `digits` dataset, $\boldsymbol{x}$ is a d-dimensional vector where $d = 64$ and for the `MNIST` dataset $d = 784$. There are several ways that the bandwidth may be defined. The first is to assume that the features are independent with the same variance, so

$$\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I}$$

where $\boldsymbol{I}$ is a $d \times d$ identity matrix. In this case, the bandwidth is specified by a single parameter, σ^2, and the Gaussian density is isotropic, meaning that it has the same spread in all directions. Another option would be to allow for a different variance for each feature,

$$\boldsymbol{\Sigma} = \text{diag}[\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2]$$

This requires that d bandwidth parameters be defined. The most general case would be for $\boldsymbol{\Sigma}$ to be an arbitrary symmetric positive definite matrix, but now the number of bandwidth parameters is $\frac{1}{2}d(d+1)$, and when $d \gg 1$ determining the best set of values for this matrix is a challenging problem.

Scikit-*Learn* only supports an isotropic bandwidth, $\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I}$, so the Gaussian kernel has the form

$$p(\boldsymbol{x}) = \frac{1}{N}\sum_{k=1}^{N} K(\boldsymbol{x}-\boldsymbol{x}_k, \sigma^2) = \frac{1}{N}\sum_{k=1}^{N} \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(\frac{1}{2\sigma^2}\|\boldsymbol{x}-\boldsymbol{x}_k\|^2\right)$$

Choosing the best bandwidth parameter σ^2 is a difficult problem and, as we will see later, there are some principled ways to find the best parameter using **cross-validation**, but here we will just use simple trial and error.[3]

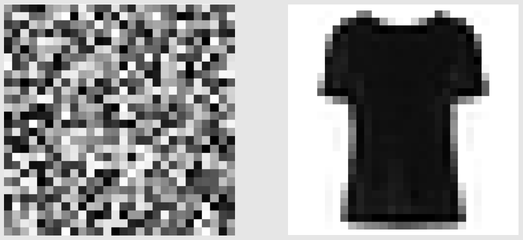
Before jumping into the problem of estimating $p(\boldsymbol{x})$, it is important to remember the **curse of dimensionality**. With only 1,797 data samples to occupy 17 bins in a 64-dimensional feature space in the case of the `digits` dataset, and only 70,000 data samples with 256 bins in a 784-dimensional feature space for the `MNIST` dataset, most of the bins will be empty. Although kernel density estimation can be performed in any number of dimensions, in practice the curse of dimensionality makes density esimation more challenging as the dimension of the feature space increases.

Questions for Discussion

(a) For the images in the `digits` dataset, the pixel values have 17 different grayscale values. If the 64-dimensional feature space is partitioned into 64-dimensional unit cubes, with 17 bins for each feature, how many cubes (bins) will there be?

(b) Repeat for the `MNIST` dataset where there are 256 different grayscale values and the dimension of the feature space is 784.

(c) Most of the d-dimensional feature space of pixels do not correspond to handwritten digits. Some may just look like random noise such as the image on the left in the figure below while others may be of some other object as shown on the right.



Discuss the importance of reducing the dimension of the feature space, i.e., is the curse of dimensionality a serious issue for this problem? If so, speculate on what a reasonable dimension might be.

Assuming that there may be an issue with the curse of dimensionality, or to look at how many *features* are actually needed to represent digits, we will use **principal components analysis** (PCA) to reduce the dimension of the feature space. As discussed later in the course, PCA is a method that can be used to drastically reduce the dimension of a feature space in such a way that the *most important information* (in some sense) is retained. With PCA, the idea is simple—reduce the dimension of a dataset while preserving as much *variability* (statistical information) as possible. To convince yourself that dimension reduction should be possible, note that most of the pixels in the images are equal to zero, particularly around the perimeter

[3]As a note, there are also optimality criteria that may be used in selecting the bandwidth.

of the image and are not needed. PCA is more sophisticated than this, but this simple example illustrates the basic idea that dimension reduction may be possible without losing too much information.

To reduce the dimension of a dataset X of vectors using PCA is done in scikit-*learn* as follows,

```
# Reduced dimension: n_components
n_components = ???   # put in a number
pca = PCA(n_components)
X_pc = pca.fit_transform(X)
```

where `n_components` is the number of **principal components** to use, i.e., the dimension that the feature space of X should be reduced to.

(e) After reducing the dimension of the feature space (or keeping it the same), the next step is to instantiate the kernel density estimator, and use the `fit` method to form an estimate of the density, $p(\boldsymbol{x})$. This may be done in a single line as follows:

```
kde = KernelDensity().fit(X_pc)
```

The two most important parameters in KDE are `kernel` and `bandwidth`. Jere the default values are used, which are `kernel='Gaussian'` and `bandwidth=1.0`. This produces an instance of the KernelDensity class that has been fitted to `X_pc`.

(f) It is now possible to generate new digits (hence the name **generative model**) by sampling `kde` using the `sample` method,

```
# Generate a random sample from the KDE
new_data = kde.sample()
```

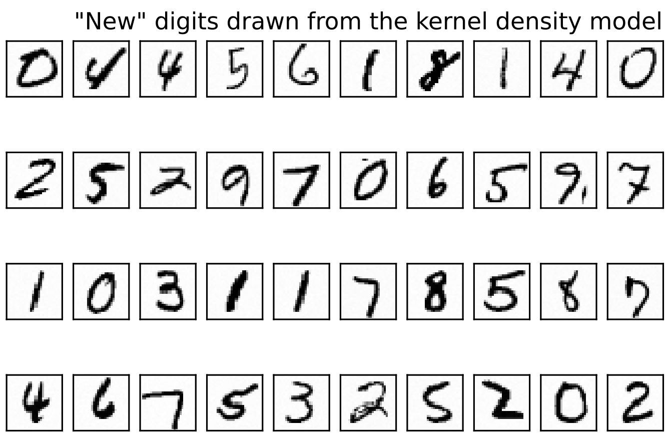
There are two parameters of interest in the `sample` method. The first is the number of random samples to generate (the default is one) and the second is `random_state` that determines the random number generation used to generate random samples. If this is set to an integer, reproducible results will be generated across multiple function calls when the same number is used again. If it is not set, then each time the function is called, different samples will be generated. Here is an example on how to generate ten random samples, and each time this command is issued, the same ten samples will be generated.

```
new_data = kde.sample(10, random_state=0)
```

(g) Once one or more random samples of $p(\boldsymbol{x})$ have been created, the vectors are transformed back into vectors of length d using the inverse PCA class, and then reshaped into an image of size 8×8 or 24×24,

```
ndim = 24   #  Set ndim to 8 for the digits dataset
new_data = pca.inverse_transform(new_data)
new_data = new_data.reshape((ndim,ndim))
```

Shown in the figure below are forty synthesized digits generated by sampling the estimated density function, $p(\boldsymbol{x})$, for the `MNIST` dataset with no reduction in the dimension of the images with a Gaussian kernel and `bandwidth=1`.



To generate these 40 images from the MNIST dataset, only two lines of code are needed,[4]

```
kde = KernelDensity().fit(X)
new_data = kde.sample(10, random_state=0)
```

Experiments
Now that you have a working environment for your **generative model**, there are a number of interesting experiments that can be performed. In the following, there are two parameters that you will be experimenting with:

1. `k`: The number of principal components, and
2. `bandwidth`: The variance of the multidimensional Gaussian.

In the following, `d` is the dimension of the digits in the dataset, which is 64 for the `digits` dataset and 784 for the `MNIST` dataset.

(a) Beginning with `bandwidth=1` and `k=d/2`, generate a set of 40 images by sampling the KDE of the image dataset.

(b) With two parameters to play with, the number of PCA components and the bandwidth of the KDE kernel, it is difficult to decide on the best pair of numbers since there are many possible combinations. One approach is to first focus on the number of PCA components, and then find the best bandwidth.

One of the attributes of the `PCA` class is `explained_variance_`. A simple explanation of what this is and why it might be useful can be described as follows. PCA begins by finding the one feature (first principal component) that accounts for the largest amount of variance in the dataset (the variance is taken to represent the amount of information that feature contains). This principal component is not generally any one of the original features, but some linear combination of them. After the first principal component is found, PCA then finds a second one that accounts for the largest amount of remaining variance. The process continues until the number of principal components is equal to the dimension of the original dataset. The attribute `explained_variance_` is the amount of variance captured by each principal component. If this variance is plotted versus the principal component number, it will be a monotonically non-increasing function, and it can be used to visually determine where or not there is a knee in the curve beyond which the addition of more principal components does not capture much additional information about the dataset.

[4]The code to display an array of images is given in your Jupyter notebook.

Make a plot of `explained_variance_` and determine what you think would be a good number to use for the dimension of the reduced feature space, and use this number in a principal components analysis of the dataset, and then generate a kernel density estimate of the data and generate 40 digits by sampling the KDE. The plot of explained variance may be generated as follows:

```
data = pca.fit_transform(digits.data)
v = pca.explained_variance_
plt.plot(v)
plt.show
```

(c) Discuss what you found in part (b). Are the images of high quality? What kind of distortions do you observe?

(d) Experiment with generating new digits using different values of `k`. What do you find when `k=1`? Would a digit classifier perform well working with images of digits that have only 5 principal components? What about 10?

Note: When `k=10`, this means is that an image is represented by only 10 real numbers.

(e) Use KDE to estimate the density function $p(\boldsymbol{x})$ of the images in your dataset without reducing the dimension of the images. Use a Gaussian kernel with different bandwidths and then generate 40 new digits by sampling the density. Describe the effect of the bandwidth parameter of the quality of the generated images? Is there some limit, large or small, beyond which the images begin to degrade significantly? Is there a way to describe analytically what you observe? What happens in the limit as the bandwidth goes to zero?

Conclusions

Discuss what you have learned concerning the generation of random digits using KDE. How sensitive is the generative model to the Gaussian bandwidth and the dimension of the feature space?

Computer Exercise 1.2 (Naive Bayes Classification):

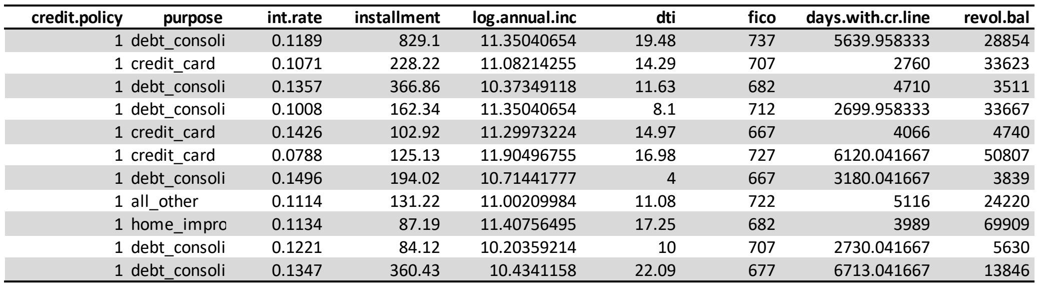
In this second part of the computer exercise you will be designing a Naive Bayes (NB) classifier to analyze loan data. In finance, making informed decisions regarding loans is extremely important for mitigating risks and ensuring a profitable portfolio. Here, you use a dataset to help understanding the factors influencing loan performance and borrower behavior. The dataset has a number of diverse features such as credit policies, interest rates, borrower income levels, debt-to-income ratios, and credit scores. The primary learning objective is to predict whether a borrower will fully repay or default on the loan.

(a) **The Dataset**

The first thing to do is to load the dataset and become familiar with its properties. The dataset is stored in an excel sheet and can be read into Scikit-*Learn* and stored in a Pandas dataframe as follows:

```
import pandas as pd
df = pd.read_csv('loan_data.csv')
```

It is assumed that you have `loan_data.csv'` saved in the same directory as your notebook, but if you wish to change the location, modify the read statement accordingly. A Pandas DataFrame is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). It is one of the most commonly used data structures in the Pandas library, which is a powerful data manipulation and analysis tool in Python. To get an understanding of the structure of the dataframe, type `df.head()`,



credit.policy	purpose	int.rate	installment	log.annual.inc	dti	fico	days.with.cr.line	revol.bal
1	debt_consoli	0.1189	829.1	11.35040654	19.48	737	5639.958333	28854
1	credit_card	0.1071	228.22	11.08214255	14.29	707	2760	33623
1	debt_consoli	0.1357	366.86	10.37349118	11.63	682	4710	3511
1	debt_consoli	0.1008	162.34	11.35040654	8.1	712	2699.958333	33667
1	credit_card	0.1426	102.92	11.29973224	14.97	667	4066	4740
1	credit_card	0.0788	125.13	11.90496755	16.98	727	6120.041667	50807
1	debt_consoli	0.1496	194.02	10.71441777	4	667	3180.041667	3839
1	all_other	0.1114	131.22	11.00209984	11.08	722	5116	24220
1	home_impro	0.1134	87.19	11.40756495	17.25	682	3989	69909
1	debt_consoli	0.1221	84.12	10.20359214	10	707	2730.041667	5630
1	debt_consoli	0.1347	360.43	10.4341158	22.09	677	6713.041667	13846

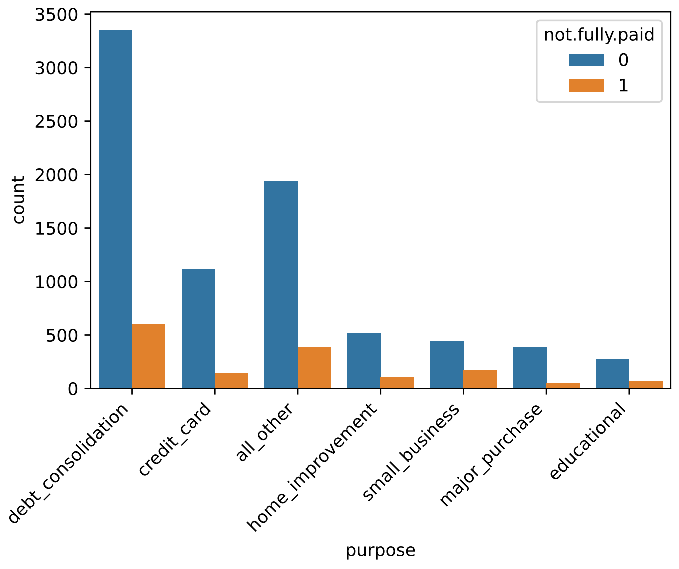
and to get a better understanding of the contents of the dataset, type `df.info()`, and you will see:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9578 entries, 0 to 9577
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   credit.policy      9578 non-null   int64
 1   purpose            9578 non-null   object
 2   int.rate           9578 non-null   float64
 3   installment        9578 non-null   float64
 4   log.annual.inc     9578 non-null   float64
 5   dti                9578 non-null   float64
 6   fico               9578 non-null   int64
 7   days.with.cr.line  9578 non-null   float64
 8   revol.bal          9578 non-null   int64
 9   revol.util         9578 non-null   float64
 10  inq.last.6mths     9578 non-null   int64
 11  delinq.2yrs        9578 non-null   int64
 12  pub.rec            9578 non-null   int64
 13  not.fully.paid     9578 non-null   int64
dtypes: float64(6), int64(7), object(1)
memory usage: 1.0+ MB
```

As shown above, the dataset consists of 14 columns and 9578 rows and all entries in the dataframe are either floating point numbers or integers, with the exception of the `purpose` column. The target variable, the one we will be trying to predict (classify from a data record) is `not.fully.paid`. To see what information the `purpose` column provides by using seaborn's countplot.

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.countplot(data=df,x='purpose',hue='not.fully.paid')
plt.xticks(rotation=45, ha='right');
```

What we see are the seven `purpose` labels,and the number of loans `fully_ paid`and those that are not for each `purpose`.



One thing to note here is that the dataset is **imbalanced**, i.e., there are many more data samples for clients that fully paid their loans compared to those who defaulted. Later we will consider the impact of an imbalanced dataset on the accuracy of a classifier and how to deal with imbalanced datasets.

(b) **Data Preparation**
There are three things that need to be done before using the dataset to learn a NB classifier. The first is to change the `purpose` feature into a numeric value, since a NB classifier is unable to handle categorical feature. We will learn more about this later, but what is done is to **one-hot encode** the feature,

```
df_prep = pd.get_dummies(df,columns=['purpose'],drop_first=True)
```

The second is to remove the target value from from the dataset to create the feature set `X`, and put it into a target variable `y`,

```
X = df_prep.drop('not.fully.paid', axis=1)
y = df_prep['not.fully.paid']
```

The third and final step is to separate the dataset `(X,y)` into a **training set** and a **test set**. The training set is used to learn the NB classification rule, and the test set is used to evaluate how well it performs on new data.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

(c) **NB Classifier**
Having created a training and test set, the process of learning a NB classifier is simple. The following will instantiate the NB classifier, use the training data `(X_train,y_train)` to learn the classifier, and find the accuracy on the test set:

```
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB()
clf.fit(X_train, y_train);
acc = clf.score(X_test,y_test);
print('Accuracy = ',acc)
```

What we find is that the accuracy is 82.06%. An accuracy of 82.06% may sound good and, depending on a number of other factors, it may be acceptable. However, a little deeper analysis may shed some more light on this classifier.

Evaluation

It should be clear that there is a difference between classifying a high-risk client as low-risk versus classifying a low-risk client as high-risk. The former has real financial implications while the other may only lead to the loss of a client. And given that the dataset is imbalanced, our accuracy may be misleading.

i. One of the performance analysis tools to be discussed later is a **confusion matrix**. that provides a finer-grained analysis of the performance of a classifier. Create a confusion matrix for your NB classifier using the following code,

```
labels = ["Fully Paid", "Not fully Paid"]
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
disp.plot();
```

The result is a 2×2 matrix of numbers where the rows correspond to the samples that are `fully_paid` or `not_fully_paid` and the columns are how many samples in each of these classes are predicted to be `fully_paid` or `not_fully_paid`. For example, the number in the upper left-hand corner is the number of data samples representing loans that are fully paid and the NB classifier predicted that they would be fully paid. Note that the samples along the diagonal are those that are correctly classified whereas those off the diagonal are misclassified.

ii. What is the accuracy of your NB classifier in correctly classifying that a client will default on their loan? How does this comare to the overall accuracy of your NB classifier? Discuss.

iii. What is the accuracy of your NB classifier in correctly classifying that a client will fully pay back the loan?