Due: Monday, 20 October 2025, 11:59 PM

- Task 1: Data preprocessing
- Task 2: Build a naive Bayes classifier
- Task 3: Improve the classifier with Laplace (additive) smoothing
- Think further

Task 1: Data preprocessing

Here you have to download data and prepare them for being used with your program.

- Small dataset for program development: "weather"(14 observations, 4 categorical variables + 2 classes). Check the weather and decide whether to play tennis or not. Download the Weather data set and the Weather data description.
- Larger dataset for training and testing: "Breast Cancer" (286 observations, 9 categorical variables + 2 classes). Check

Remark: For use with Python, you can read the nominal values as strings. Python has a data type called dictionary that accepts any type (including strings) as keys and as values.

You could work with the numpy library alone. However many operations, like reading data files or splitting data into subsets, are easier if you use the Pandas library.

Remark: The Weather data set is so small that you can do the calculations by hand to compare with your code, and check if it is correct.. So it is advised to use this while you are writing the code. The Breast cancer dataset is larger in all dimensions (number of rows, number of columns, number of values per attribute). So it is better suited for checking the training-testing workflow when the code is working. NOTE THAT THERE ARE MISSING VALUES. They are few (5% of the observations have missing values in 2 columns only).

Remark: In statistical jargon, nominal or categorical variables don't have "values" but "levels".

Task 2: Build a naive Bayes classifier

Here, following the conventions from Scikit-learn, you have to create a Python class as in the following template:

```
import numpy as np
...other imports if needed...

class nbayes:
    trained = False

    def fit(self, X_train, y_train):
         ...your code here...

    def predict(self, X_test):
        ...your code here...

    def test(self, X_test, y_test):
        if not self.trained:
            raise valueError
        y_predict = self.predict(ytest)
        return (np.array(y_test) == np.array(y_predict).sum()/len(y_test)
```

The **fit** method takes the following parameters:

1. **X_train**: a set of categorical data, as a *n x d* matrix, to be used as the training set
2. **y_train**: a set of corresponding class labels, as a *n* vector

The function should train a Naive Bayes classifier on the training set (first input argument), using the second argument as the target. It should save the information obtained during training (probabilities) in class variables, i.e. *self.something*, so that they are retained after the function ends. It should set **self.trained** to True once finished, before returning.

The **predict** method takes the following parameter:

?

1. **X_test**: a set of categorical data, as a *n x d* matrix, to be used for inference (predicting the class of each observation in X_test)

The function should use the information saved during training, and return the predicted class labels as a vector **y_predict**. If the model is not trained, it should do:

```
raise valueError
```

The **test** method is given.

Write a script to test the software on the provided data.

Hint: From the slides, you see that probability = frequency = number/totalnumber.

What may not be immediately clear from the formulas (because it is implicit in the definition of a probability mass function) is that for each variable you have to compute a probability for each possible value. This is the *probability mass function* that is used in the formulae. It gives the probability for each value (conditioned to the class, i.e. a likelihood; but this does not matter for the calculations, conditional probabilities are just probabilities).

Explicitly in pseudocode, what you have to compute (and store in a suitable data structure) is:

```
for each class c
  for each variable x
    for each possible value v for variable x
        the number of instances of class c that have variable x == value v...
        ...divided by the number of instances of class c
```

and this will be our estimate of the likelihood $P(x=v \mid c)$

Remarks: The classifier should be programmed in such a way to be suitable for the following situations:

- other data sets of different cardinalities (n, m) and dimensionality (d)
- test sets including attribute values that **were not in the training set**.

In the last case the program should issue an error and discard the corresponding pattern, because if you use that value to index a matrix it will likely go out of bounds (e.g., if you computed probabilities for values 1, 2 and 3 and you get value 4 in the test).

Hint: Use the slides (ML 2) to implement the classifier. Note that all attributes are categorical, but some are non-binary having more than 2 levels.

Task 3: Make the classifier robust to missing data with Laplace (additive) smoothing

You will observe that, with a small data set, some combinations that appear in the test set were not encountered in the training set, so their probability is assumed to be zero. When you multiply many terms, just one zero will make the overall result be zero.

In general, it may be the case that some values of some attribute do not appear in the training set, however **you know** the number of levels in advance.

An example is binary attributes (e.g., true/false, present/absent...). You know that attribute x can be either true or false, but in your training set you only have observations with x = false. **This does not mean that the probability of x = true is zero!!!**

To deal with this case, you should introduce a kind of prior information, which is called **additive smoothing** or **Laplace smoothing**.

Laplace smoothing -- Suppose you have random variable x taking values in { 1, 2, ..., v } (v possible discrete values). What is the probability of observing a specific value i ?

We perform N experiments and value i occurs n_i times. Then:

- Without smoothing (simple frequency) the probability of observing value i is given by $P(x=i)=\frac{n_i}{N}$
- With Laplace smoothing, the probability of observing value i is given by $P(x=i)=\frac{n_i+a}{N+av}$ where a is a parameter (see below)

NOTE: Smoothing is NOT a way to improve the performance of an existing classifier. It is a way to solve a problem of missing d... that would prevent you from building a classifier. You should not expect a better performance from a classifier with smoothing if

the version without smoothing already works!

You have to change your code in 2 ways:

1) In the data preparation step, add the information about the number of levels. This means that for each data column you should add the number of possible different values for that column.

In the case of the weather data, this can be a list (or vector) [3, 3, 2, 2]. You can add this list as a first row in all the data sets (training and test), to be interpreted with this special meaning, and store it in a class variable.

2) When you compute probabilities, you introduce Laplace smoothing in your formulas by adding some terms that take into account your prior belief. Since **you don't know anything**, your prior belief is that all values are **equally probable**. In this case, Laplace smoothing gives probability = 1/2 (for binary variables) or more generally probability = 1/v (for variables with v possible values). This is why the number of possible values must be known (modification 1).

In formulae, you should replace:

```
P(attribute x = value y | class z) =
   (number of observations in class z where attribute x has value y) / (number of observations in class z)
```

with the following:

```
P(attribute x = value y | class z) =
   ((number of observations in class z where attribute x has value y) +a) / ((number of observations in class z) +av)
```

where:

- v is the number of values of attribute x, as above;
- you can use a = 1. As a further refinement, you can experiment with a > 1 (which means "I trust my prior belief more than the data") or with a < 1 (which means "I trust my prior belief less than the data")

Think further

- If you have large data sets, you may incur in numerical errors while multiplying a lot of frequencies together. A solution is to work with log probabilities, by transforming logarithmically all probabilities and turning all multiplications into sums.
- How would you proceed if the input variables were continuous? (Hint: variable values are used to compute probabilities by counting, but theory tells us that probabilities may be obtained by probability mass functions directly if they are known analytically.)
- You can experiment with the continuous variable case using the Iris data set (with its description). The four features can be made binary by (1) computing the average of each and (2) replacing each individual value with True or 1 if it is above the mean and False or 0 if it is below. You can use the median in place of the average.

Add submission

Submission status

Attempt number	This is attempt 1.
Submission status	No submissions have been made yet
Grading status	Not graded
Time remaining	13 days 7 hours remaining

?