1. Turn a question into records

Work individually in Python. The latest lecture introduced Bayes classification; Lab 9 practised dice and two-bag simulations. Here you will build a new classifier from recorded data. Write your own counting, fitting, prediction and evaluation functions; use ready-made classification functions only in Task 9. Keep code, small result tables/plots and explanations. All delivery records are fictional.

Understand the concept

Our question is: will a delivery arrive late? Each row is one distinct delivery. Two binary inputs are known at dispatch: rain (1=yes) and heavy_traffic (1=yes). The label late is learned after arrival: 1=late and 0=on time. Repeated feature values can be valid different deliveries. Training estimates model probabilities; validation checks choices; the final test evaluates the chosen procedure.

Small worked example

The group (0,0,0,7) means seven different deliveries with no rain, no heavy traffic and an on-time outcome. Expand it into seven rows. It is not one observation with a fourth numerical feature. The counts summarize a small teaching dataset, not a true population law.

Implement and experiment

1. Expand all eight groups in the code below. Print the 20 rows with row numbers and count on-time and late records using loops.
2. Write a data check: every row must contain three binary values, and both label classes must be present.
3. Separate each row into two input values and its label. Explain why actual arrival time or the late label must never be an input to this prediction.
4. Print the number of rainy, heavy-traffic and late deliveries, stating the denominator for each proportion.

Coding hint - complete the missing work

```
# Each tuple: (rain, heavy_traffic, late, number_of_records)
grouped = [(0,0,0,7), (0,1,0,2), (1,0,0,2), (1,1,0,1),
           (0,0,1,1), (0,1,1,1), (1,0,1,1), (1,1,1,5)]
train = []
for rain, traffic, late, count in grouped:
    # TODO: append (rain, traffic, late) exactly count times
    pass
# Use row positions as separate delivery IDs; do not use ID as a feature.
```

Check and debug

There should be 20 training deliveries: 12 on time and 8 late. IDs are for identifying rows only. Repeated binary patterns should not be removed as duplicates without evidence that they represent the same original delivery.

Explain what you learned

What does one row represent? Which information is available at prediction time? Explain why the model may make different predictions for two rows with different inputs but cannot see their future labels.

2. Describe a new random variable

Understand the concept

Let K be the number of warning flags on a delivery: K=rain+heavy_traffic. It can be 0, 1 or 2. A probability mass function assigns a probability to each possible K. Here we use observed training proportions to form an empirical distribution. Its mean is a weighted average; its variance measures squared distance from that mean. These summaries concern warning flags, not lateness.

Small worked example

The records with rain=0 and traffic=0 have K=0. There are 7 on-time and 1 late records with this input, so the empirical probability P(K=0) is 8/20=0.4. To calculate E[K], multiply each possible K by its probability and add; do not average only the distinct values [0,1,2].

Implement and experiment

1. Write a loop that calculates K for each of the 20 records. Count K=0,1,2 and print counts, probabilities and cumulative probabilities.
2. Compute mean=sum(k*p[k]) and variance=sum((k-mean)**2*p[k]) using your own loops.
3. Independently average the 20 recorded K values and check that it matches the weighted mean.
4. Plot the distribution for all training rows and, separately, within late and on-time rows. Label the group used as each denominator.

Coding hint - complete the missing work

```
counts = [0, 0, 0]
for rain, traffic, late in train:
    k = rain + traffic
    # TODO: increment counts[k]
# TODO: probabilities = counts divided by len(train)
# TODO: weighted mean, variance, cumulative sum and bar plot
# In a class-specific plot, divide by that class's row count.
```

Check and debug

The probabilities must sum to 1. Your all-row mean should be 0.9; variance should be non-negative. K=2 is one input pattern, but can occur with either label. Do not interpret E[K] as a probability of late delivery.

Explain what you learned

Why can the expected number of flags lie between possible integer outcomes? What do the class-specific distributions suggest, and why do they not establish that rain causes late arrival?

3. Learn conditional probabilities

Understand the concept

A conditional probability changes the reference group. P(rain=1 | late=1) asks about rainy deliveries within the late group. P(late=1 | rain=1) asks about late outcomes within the rainy group. The same joint count appears in both, but the denominators differ. Bayes rule relates the two directions using the class prior and total evidence probability.

Small worked example

Six of the eight late records have rain=1, so P(rain=1 | late=1)=6/8=0.75. There are nine rainy deliveries in total, so P(late=1 | rain=1)=6/9. Bayes gives the same result: (6/8)*(8/20)/(9/20). These are empirical estimates from the training table.

Implement and experiment

1. Build a two-way count table: late/on-time versus rain/no rain. Derive class priors, both rain likelihoods and both reversed conditionals from counts.
2. Write a Bayes calculation for a new rainy delivery and verify it against the fraction of late rows within the rainy group.
3. Repeat the calculation for a non-rainy delivery, using 1-P(rain=1 | class). Then repeat for the traffic feature.
4. For each result, print numerator and denominator counts before the decimal probability.

Coding hint - complete the missing work

```
def count_rows(rows, condition):
    total = 0
    for row in rows:
        # TODO: if condition(row), increment total
        pass
    return total
# Bayes numerator = P(rain|late) * P(late)
# Evidence denominator = sum of the rainy routes from BOTH classes.
# TODO: verify with direct conditional counts.
```

Check and debug

P(late) should be 8/20. P(rain) should be 9/20. Reverse conditionals need not match. If a conditioning group is empty, report the estimate as undefined rather than divide by zero.

Explain what you learned

Why is P(rain | late) different from P(late | rain)? How does Bayes turn a probability about observed evidence into a probability about an unknown class?

4. Combine two pieces of evidence

Understand the concept

One feature ignores some available information. Bernoulli Naive Bayes combines binary features by assuming they are independent given the class. This is a modelling assumption, not a fact guaranteed by the dataset. For an input x=(rain,traffic), multiply its class prior by the probability of each observed feature value within that class, then normalize the two class scores.

Small worked example

For a rainy, heavy-traffic input, the unsmoothed late score is (8/20)*(6/8)*(6/8)=0.225. Its on-time score is (12/20)*(3/12)*(3/12)=0.0375. Divide each by their sum to obtain two posterior probabilities. These scores are not yet normalized probabilities.

Implement and experiment

1. Use your class counts to calculate all four feature likelihoods: rain and traffic, each within late and on-time.
2. Write score_one_class(x,class_id,prior,theta). If a feature is 1 use theta; if it is 0 use 1-theta. Include both features.
3. Write predict_proba(x,model) and calculate probabilities for [0,0], [0,1], [1,0] and [1,1]. Predict late at probability >=0.5.
4. Compare the [1,1] Naive Bayes posterior with the empirical joint-table answer: among all [1,1] records, what fraction are late? Explain why they need not be equal.

Coding hint - complete the missing work

```
# theta[c][j] means P(feature j = 1 | class c)
def score_one_class(x, c, prior, theta):
    score = prior[c]
    for j in range(len(x)):
        # TODO: factor = theta[c][j] if x[j]==1 else 1-theta[c][j]
        # TODO: multiply score by factor
        pass
    return score
# TODO: normalize scores[0] and scores[1] by their sum.
```

Check and debug

Each posterior pair must sum to 1. For [1,1], the empirical joint fraction is 5/(5+1); the Naive Bayes calculation uses products of marginal feature likelihoods within each class. Distinguish the assumption-based approximation from that joint-table estimate.

Explain what you learned

What does conditional independence mean here? Why must a feature value of zero contribute evidence too? Can a high model probability guarantee the delivery outcome?

5. Build reusable fit and predict code

Understand the concept

Fitting this model means counting training records and saving probabilities. It does not require gradients, a learning rate or backpropagation. A reusable fit function should work for any fixed number of binary input features. Keep the label in the last position and keep all other entries as features. Prediction must reuse the saved model instead of recomputing counts from the new example.

Small worked example

For the original two-feature model, a late-class row count is 8 and its rain-one count is 6, so its unsmoothed rain likelihood is 6/8. The class prior uses all training rows: 8/20. Different quantities require different denominators.

Implement and experiment

1. Implement fit_nb(rows,alpha=0.0), returning classes, priors and per-class feature-one probabilities. Derive the feature count from len(rows[0])-1.
2. Implement predict_proba for any number of binary features, and predict_label at threshold 0.5. Return probabilities in a clearly stated class order.
3. Reproduce every result from Task 4 using the saved model. Print the learned priors and likelihoods before predictions.
4. Change the order of training rows and refit. Confirm that counts and probabilities stay the same. Check that predictions do not modify the model.

Coding hint - complete the missing work

```
def fit_nb(rows, alpha=0.0):
    d = len(rows[0]) - 1
    priors, theta = {}, {}
    for c in [0, 1]:
        # TODO: collect rows whose last value is c
        # TODO: prior = class count / total count
        # TODO: for each j, count feature-one values within class
        pass
    # Return a dictionary with priors, theta and classes [0,1].
    raise NotImplementedError

# Tasks 1-5 use alpha=0: each theta is count/class_count.
# The smoothing extension on the next page changes this formula.
```

Check and debug

Keep labels out of x. Require binary input values and the expected feature count. If a class has no training rows, stop with an explanatory error for this exercise. Do not silently invent a class distribution.

Explain what you learned

Which numbers are learned during fitting? Which calculations happen when predicting a new delivery? Why does row order not affect this count-based fit?

6. Handle small counts with smoothing

Understand the concept

Not observing an event does not prove that it is impossible. An unsmoothed zero feature likelihood can make an entire class score zero. Additive smoothing adds pseudo-counts for both possible binary outcomes. With alpha>0, estimate theta=(ones+alpha)/(class_count+2*alpha). The denominator has 2*alpha because the feature has two possible values, 0 and 1.

Small worked example

If a feature is 1 zero times among eight class records, its unsmoothed probability is 0/8=0. With alpha=1 it becomes (0+1)/(8+2)=0.1. Its feature-zero probability is then 0.9. In our delivery table, late-class rain with alpha=1 becomes (6+1)/(8+2)=0.7.

Implement and experiment

1. Extend fit_nb with the smoothing formula. Keep class priors equal to observed class fractions; smooth feature likelihoods only.
2. Fit the original data at alpha=0 and alpha=1. Print likelihoods and all four input-pattern posterior probabilities side by side.
3. On a copy of the data, set the traffic feature to zero in every late row. Refit at alpha=0 and alpha=1 and compare the late-class score for [1,1]. Restore the original data afterward.
4. Predict before running: which score becomes exactly zero without smoothing, and why? In all later baseline tasks, use the ORIGINAL data and alpha=1.

Coding hint - complete the missing work

```
# In fit_nb, replace ones/class_count with:
# (ones + alpha) / (class_count + 2*alpha)
# Keep P(feature=0 | class) = 1 - theta.

# Make a new list of tuples for the deliberate zero-count case.
# Do not edit the original train list in place.
# TODO: compare fitted likelihoods and posterior probabilities.
```

Check and debug

Smoothing is applied separately within each class and feature. It is not division by the number of input features. Do not add alpha to the priors for the library comparison. A zero score for one class can be normalized if the other class score is positive; both-zero evidence must be handled explicitly.

Explain what you learned

Why does alpha change predictions even when every original count is non-zero? What assumption does smoothing add, and why is a small observed count less certain than a large one?

7. Test the model assumptions

Understand the concept

The Naive Bayes independence assumption is about features within each class. Duplicating a feature adds no new information, yet a naive product can count the same evidence again. Separately, the class prior describes how common each outcome is before observing inputs. Changing that prior while holding likelihoods fixed is a controlled scenario, not proof that the new prior is correct.

Small worked example

For a late record, the empirical joint probability of rain=1 and traffic=1 is 5/8. The product of its unsmoothed separate likelihoods is (6/8)*(6/8). They differ, so the features are not exactly conditionally independent in this training table. Copying rain into a third column creates an even clearer dependence.

Implement and experiment

1. For each class, compare the empirical joint probability of [1,1] with the product of its separate unsmoothed likelihoods. Print both.
2. Create duplicate-feature rows (rain,traffic,rain,label). Fit your generic model at alpha=1 and compare its posterior for [1,1,1] with the original model's posterior for [1,1].
3. Using a fresh copy of the ORIGINAL alpha=1 model, change only the late prior from 0.4 to 0.1 and the on-time prior to 0.9. Keep every feature likelihood unchanged. Recompute all four posteriors.
4. Plot original, duplicate-feature and changed-prior probabilities for matching input patterns. Restore the baseline before evaluation.

Coding hint - complete the missing work

```
duplicate_rows = []
for rain, traffic, late in train:
    # TODO: append (rain, traffic, rain, late)
    pass
# Fit three binary features using the same generic fit function.
# For the prior experiment, copy model dictionaries/lists first.
# TODO: change priors only; do not refit likelihoods.
```

Check and debug

In duplicate-feature data the last element remains the label. Never treat the copied rain value as a new independent measurement. A posterior change does not establish a quality improvement. The changed-prior scenario assumes the class-conditional feature distributions are still appropriate.

Explain what you learned

Did the duplicate make the model more confident without adding information? Why can prior frequencies change the predicted class? Which assumptions would you inspect before trusting a real classifier?

8. Evaluate predictions and decisions

Understand the concept

A posterior probability and a class decision are different outputs. A threshold converts P(late | inputs) into an action: predict late when it is at least the threshold. Validation labels allow us to compare choices after fitting on training data. Accuracy treats all mistakes equally; a decision rule may care more about one kind of mistake. We use a fictional cost to make that distinction visible.

Small worked example

A false alarm predicts late for an on-time delivery and costs 1 point. A missed late delivery predicts on time for a late delivery and costs 4 points. Two rules can make the same number of mistakes yet have different total costs. These values are teaching assumptions, not a real shipping policy.

Implement and experiment

1. Fit the ORIGINAL 20-row data once with alpha=1. Predict the eight validation probabilities using this unchanged model.
2. Write evaluation code that counts TP, FP, FN and TN. Test thresholds 0.2, 0.5 and 0.8; print the four counts, accuracy and FP+4*FN.
3. Choose the threshold with the lowest validation cost. If tied, choose the larger threshold and state that tie rule. Do not refit on validation rows.
4. Inspect one false alarm or missed late delivery. Explain which probability and threshold produced it. Keep the selected threshold fixed for the final test.

Coding hint - complete the missing work

```
# New deliveries: (rain, traffic, late). Use for validation only.
validation = [(1,1,1), (1,0,1), (0,1,1), (0,0,0),
              (1,0,0), (0,1,0), (0,0,0), (1,1,1)]
# Class 1 means late. For each prediction:
# TP: late -> late; FP: on-time -> late
# FN: late -> on-time; TN: on-time -> on-time
# accuracy = (TP+TN)/number_of_rows
# fictional decision cost = FP + 4*FN
```

Check and debug

The four confusion counts must sum to 8. Every validation probability comes from the same baseline alpha=1 model. Keep full precision when applying thresholds. Identical input patterns can have different observed outcomes because the two features do not determine the label.

Explain what you learned

Did the highest accuracy identify the lowest-cost choice? Why is a threshold change different from changing the fitted probabilities? What does this small validation set fail to guarantee?

9. Compare with a library implementation

Understand the concept

scikit-learn is a Python machine-learning library; sklearn is its import name. Its BernoulliNB class implements Naive Bayes for binary features, including evidence from zeros. Here fit estimates counts and probabilities; predict_proba returns class probabilities. It replaces your manual fitting and prediction code, but does not choose the question, check your assumptions or interpret the result.

Small worked example

For a fair comparison use BernoulliNB(alpha=1.0,binarize=None,fit_prior=True). Match the original training rows, feature order, empirical priors and smoothing formula. Inspect classes_: its order tells you which column of predict_proba refers to the late class. Do not compare only final class labels, which can hide a probability mismatch.

Implement and experiment

1. Only now import BernoulliNB. Fit it on two-column input arrays and the separate labels. If needed, install scikit-learn in your Python environment or use a notebook environment where it is available.
2. Compare your manual and library priors, feature-one likelihoods and posterior probabilities for all four input patterns. Print the largest absolute difference.
3. Apply the SAME validation-selected threshold to both models' late probabilities. Check that their decisions and confusion counts match.
4. Evaluate once on the 12 final test deliveries below. Report accuracy, cost and the confusion counts; explain one error. Do not change alpha, features or threshold after seeing this result.

Coding hint - complete the missing work

```
from sklearn.naive_bayes import BernoulliNB
# X_train = [list(row[:-1]) for row in train]
# y_train = [row[-1] for row in train]
reference = BernoulliNB(alpha=1.0, binarize=None, fit_prior=True)
# TODO: reference.fit(X_train, y_train)
# TODO: predict_proba on [[0,0],[0,1],[1,0],[1,1]]
# Compare exp(class_log_prior_) and exp(feature_log_prob_).
# Find late-column index from reference.classes_.tolist().index(1)
```

Check and debug

Matched probabilities should agree to about 1e-10. Check class order, feature order, absent-feature factors and alpha before blaming floating-point arithmetic. Equal results verify the implementation agreement, not that Naive Bayes assumptions are true for real deliveries.

Explain what you learned

What work did the library replace? What decisions remained yours? Did the selected threshold generalize as expected? Explain why twelve synthetic test rows cannot establish real-world reliability.

Final test data

```
# Open these final test records only after selecting the threshold.
test = [(0,0,0), (0,1,0), (1,0,0), (1,1,1),
        (0,0,0), (0,1,1), (1,0,1), (1,1,1),
        (0,0,1), (0,1,0), (1,0,0), (1,1,1)]
# The first two entries are inputs; the last is the label.
# TODO: evaluate the selected procedure without changing it.
```

Lecture: Week 9, slides 15-18, 25-29, 45-55 and A10-A11. Reference: scikit-learn BernoulliNB documentation.