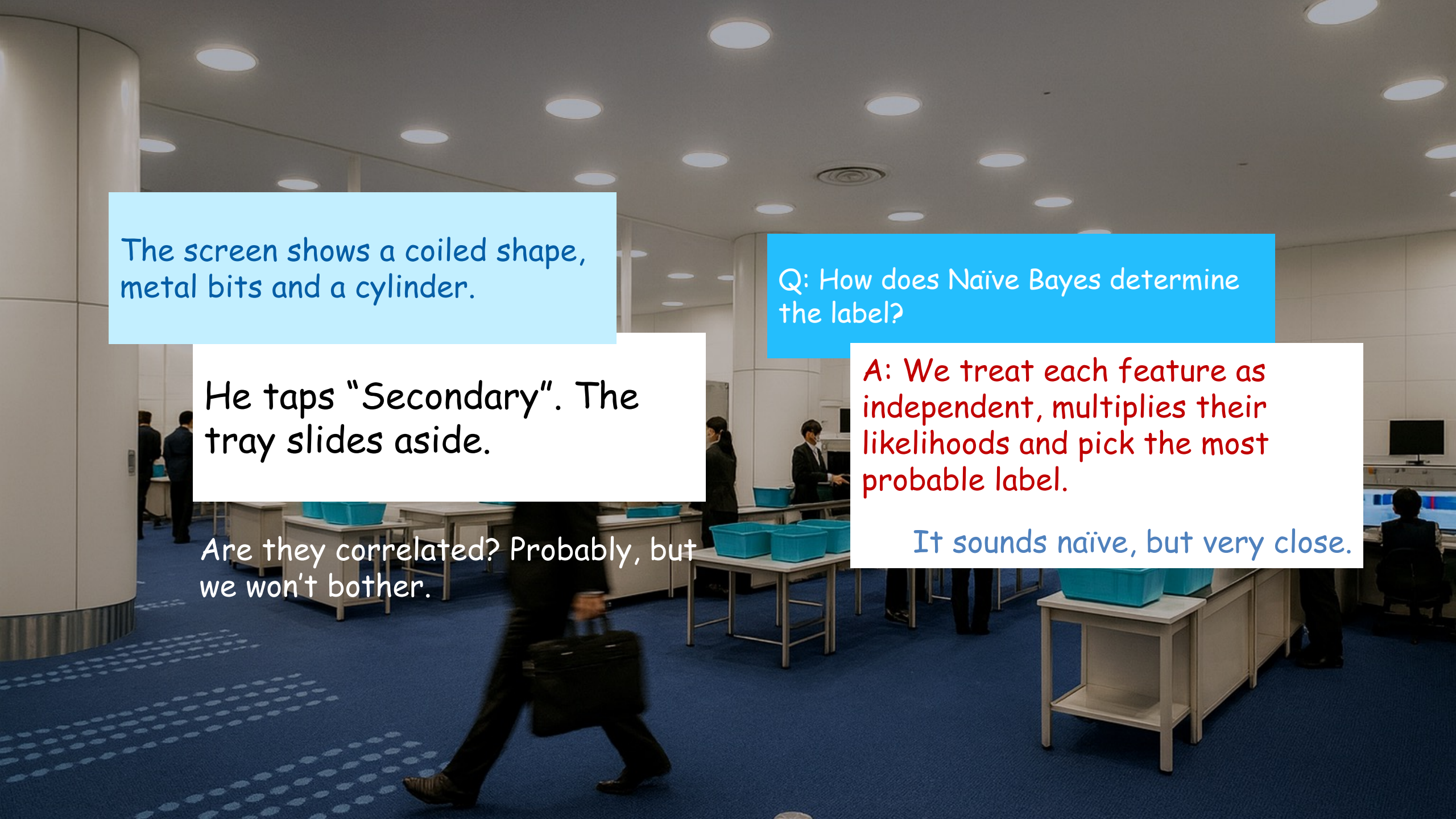
Machine Learning

Naïve Bayes

Tarapong Sreenuch

8 February 2024

克明峻德，格物致知



The screen shows a coiled shape, metal bits and a cylinder.

He taps "Secondary". The tray slides aside.

Are they correlated? Probably, but we won't bother.

Q: How does Naïve Bayes determine the label?

A: We treat each feature as independent, multiplies their likelihoods and pick the most probable label.

It sounds naïve, but very close.

2015 Gallup Poll: Online Dating Sites

		Age				
		18-29	30-49	50-64	65+	Total
Used online dating site	Yes	60	86	58	21	225
	No	255	426	450	382	1513
	Total	315	512	508	403	1738

% of 30-49 year olds using online dating sites = 86/512 ≈ 0.17.

Conditional Probability

Definition: The conditional probability of event A given an event B happened is

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

where we assume $P(B) \neq 0$).

An equivalent and useful formula is

$$P(A \cap B) = P(A|B)P(B)$$

Conditional Probability (cont.)

		Age				
		18-29	30-49	50-64	65+	Total
Used online dating site	Yes	60	86	58	21	225
	No	255	426	450	382	1513
	Total	315	512	508	403	1738

$$P(\text{Used}|30-49) = \frac{P(30-49 \cap \text{Used})}{P(30-49)}$$

$$= \frac{\frac{86}{1738}}{\frac{512}{1738}} = \frac{86}{512} \approx 0.17$$

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Bayes Theorem

Theorem (Bayes Rule): For events A and B, where $P(A), P(B) > 0$,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

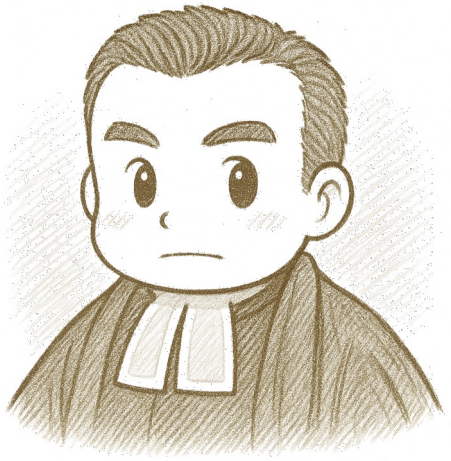
$$P(A \cap B) = P(B \cap A)$$

$$P(A|B)P(B) = P(B|A)P(A)$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$P(A), P(B) > 0$, is called the prior (our belief without knowing anything).

$P(A|B)$ is called the posterior (our belief after learning B).



Bayes Theorem (cont.)

		Age				
		18-29	30-49	50-64	65+	Total
Used online dating site	Yes	60	86	58	21	225
	No	255	426	450	382	1513
	Total	315	512	508	403	1738

$$P(30{-}49|\text{Used}) = \frac{P(\text{Used}|30-49)P(30-49)}{P(\text{Used})}$$

$$= \frac{\frac{86}{512} \times \frac{512}{1738}}{\frac{225}{1738}} = \frac{86}{225} \approx 0.38$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Recall: Joint Probability

Q: What is the probability (or likelihood) of rolling a six on both dice?

A: $\frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$

$$P(A,B) = P(A) \times P(B)$$

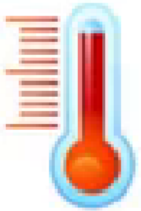
Probability of Rolling a Six on Dice A

Probability of Rolling a Six on Dice B

Joint Probability: Counter Example

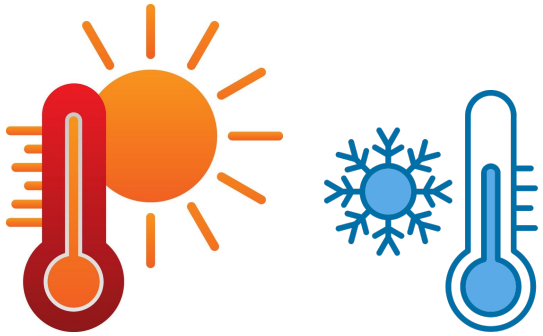
$$P(A, B) = P(A) \times P(B)$$

$= 0$



> 0

> 0



Events often exhibit correlations. It is unlikely to hold true in most real-world scenarios

Naïve Assumption

The naïve assumption refers to the simplifying assumption of conditional independence. It assumes that all the features (variables) in the dataset are independent of each other, given a target class label.

$$P\left(x^{(1)}, x^{(2)}, \ldots, x^{(m)} \middle| Y\right) \approx P(x^{(1)}|Y)P(x^{(2)}|Y)\cdots P(x^{(m)}|Y)$$

$$\approx \prod_{i=1}^{m} P(x^{(i)}|Y)$$

Why Is It Called "Naïve"?

Features often exhibit correlations. It is unlikely to hold true in most real-world scenarios

Naïve Bayes Classifier

The Naïve Bayes classifier relies on Bayes' Theorem:

Conditional Probability (or Likelihood) *Prior Probability*

$$P\left(y \mid x^{(1)}, x^{(2)}, \ldots, x^{(m)}\right) = \frac{P\left(x^{(1)}, x^{(2)}, \ldots, x^{(m)} \mid y\right) P(y)}{P\left(x^{(1)}, x^{(2)}, \ldots, x^{(m)}\right)}$$

Posterior Probability *Evidence*

Bayes' Theorem: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

$$\approx \frac{P(x^{(1)}|y)P(x^{(2)}|y)\cdots P(x^{(m)}|y)P(y)}{P\left(x^{(1)}, x^{(2)}, \ldots, x^{(m)}\right)}$$

Naïve Assumption:
$P\left(x^{(1)}, x^{(2)}, \ldots, x^{(m)} \middle| Y\right) \approx P(x^{(1)}|y)P(x^{(2)}|y)\cdots P(x^{(m)}|y)$

$$\propto P(x^{(1)}|y)P(x^{(2)}|y)\cdots P(x^{(m)}|y)P(y)$$

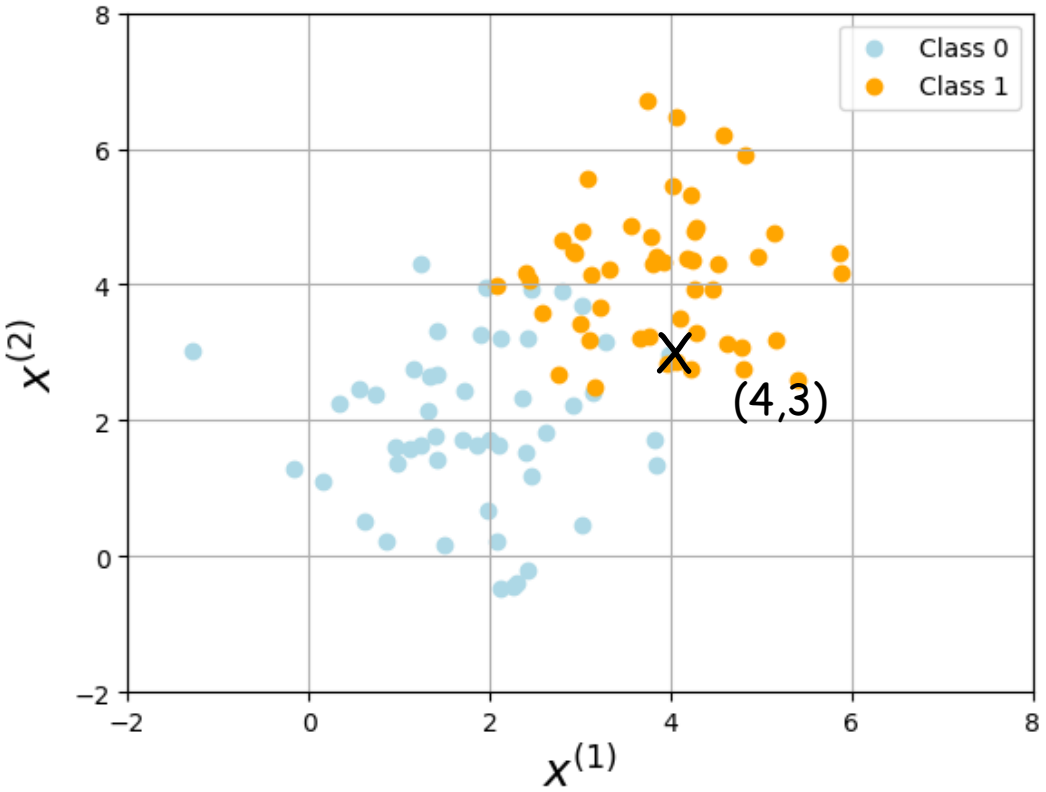
Ignoring the Evidence:
$P\left(x^{(1)}, x^{(2)}, \ldots, x^{(m)}\right)$ *is constant for all classes.*

$$\propto P(y)\prod_{i=1}^{m} P(x^{(i)}|y)$$

To predict the class y for a given input $\vec{x}$, the Naïve Bayes classifier selects the class with the highest posterior probability:

$$\hat{y} = \arg\max_{y}\left(P(y)\prod_{i=1}^{m} P(x^{(i)}|y)\right)$$

Ignoring Evidence



$$P(0\ |4,3) \propto \frac{P(4|0)P(3|0)}{P(4,3)}$$

$$P\ (1\ |4,3) \propto \frac{P(4|1)P(3|1)}{P(4,3)}$$

Evidence

Think About It: (Example)

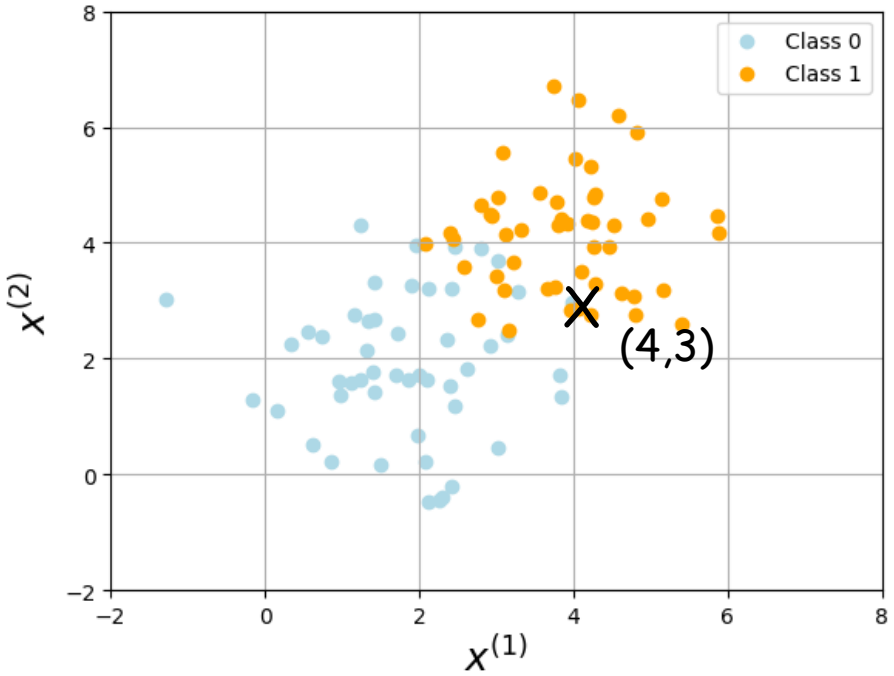
0.7 > 0.3 → 0.7/x > 0.3/x regardless the value of x (>0) is.

To predict the class y, we selects the class based on which of the twos, i.e. $P(0\ |4,3)$ and $P(1\ |4,3)$, is the highest. Both are sharing the same evidence, i.e. $P(4,3)$, and hence we can drop it from the posterior calculations. This reduces the calculations to $P(0\ |4,3) \propto P(4|0)P(3|0)$ and $P(1\ |4,3) \propto P(4|1)P(3|1)$.

Gaussian Naïve Bayes

To predict the class $\mathbf{y}$, we calculate the likelihood of a data point $\vec{\mathbf{x}}$ being class $\mathbf{y}$, and then select the class with the highest posterior probability.

$$P\left(y|x^{(1)},x^{(2)}\right) \propto P(x^{(1)}|y)\, P(x^{(2)}|y)\, P(y)$$



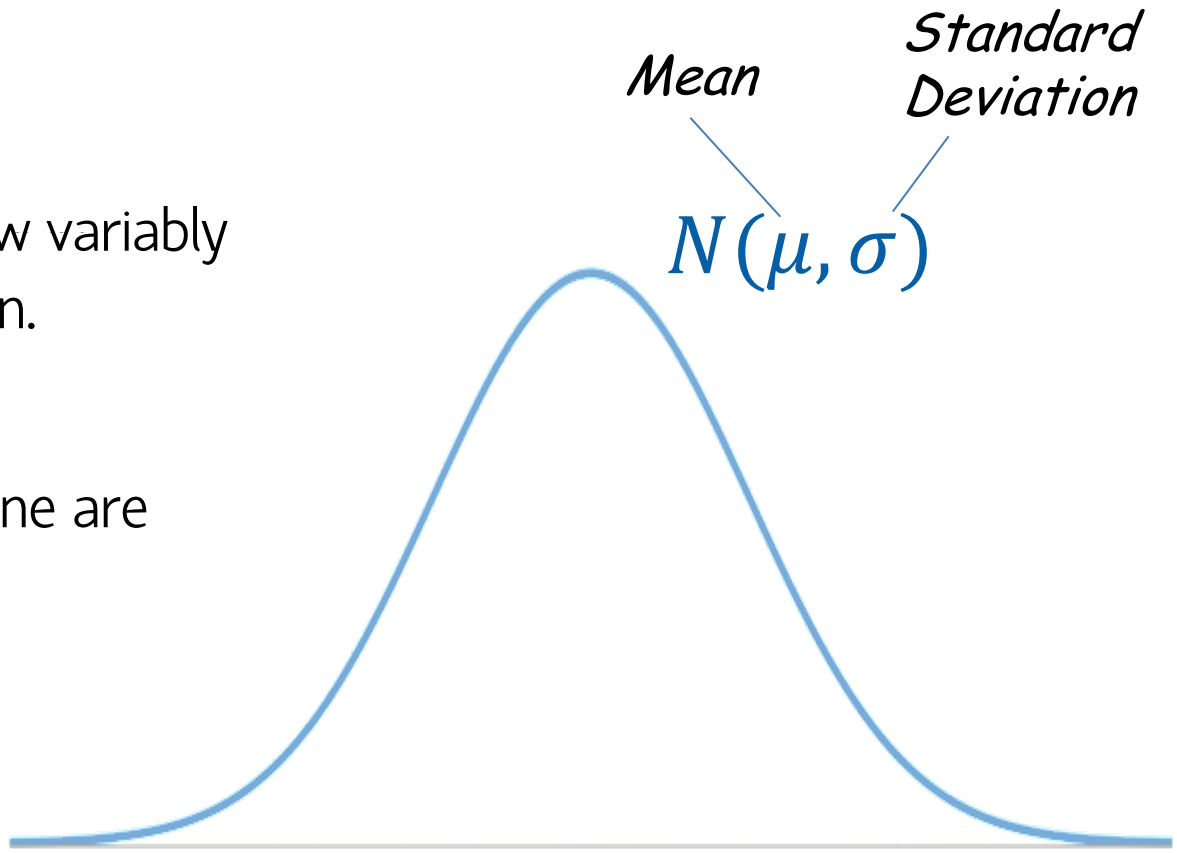
Q: How do we calculate these?

A: We count frequencies for prior probabilities. For conditional probabilities, we fit Normal distributions from the data and then compute the likelihood.

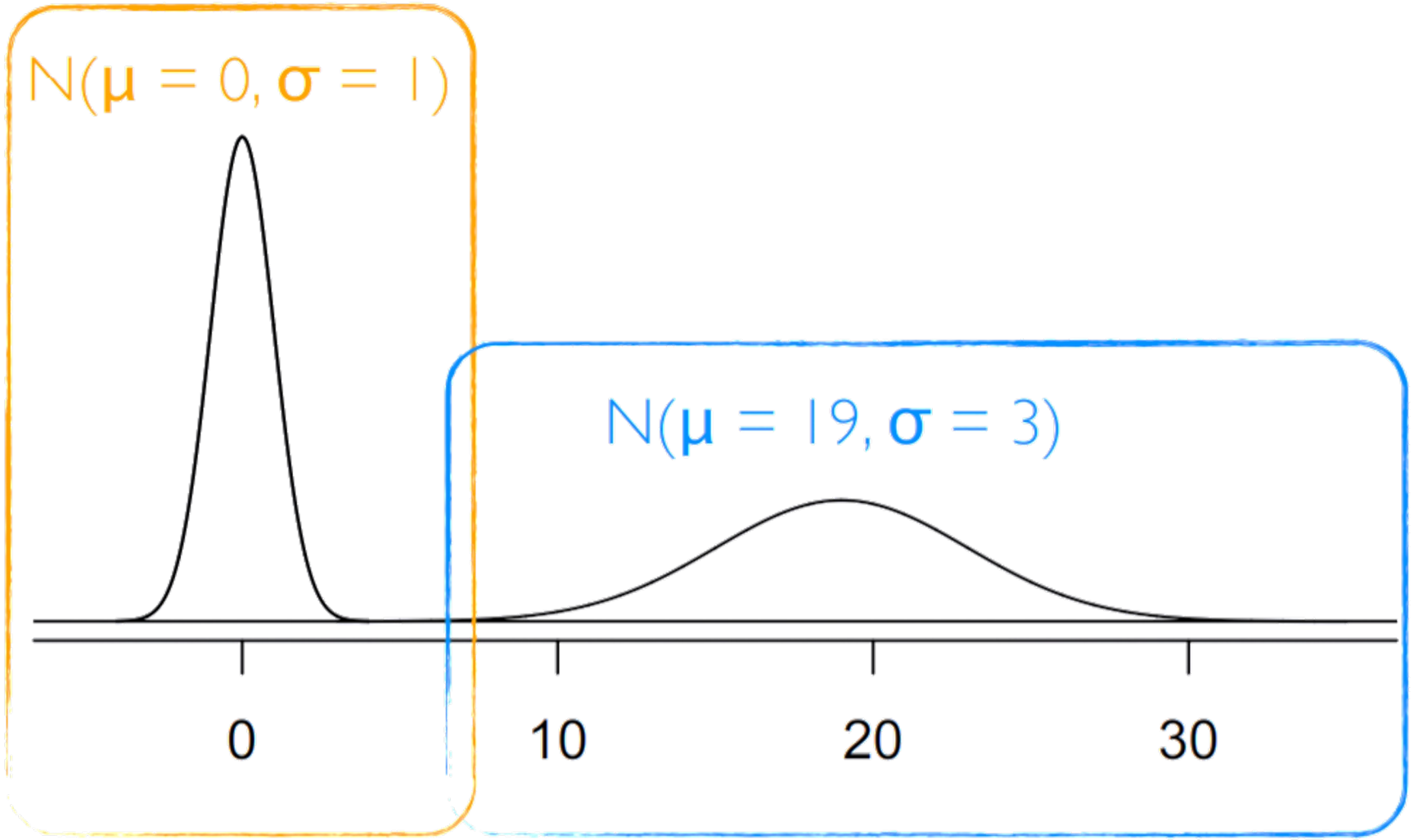
Normal Distribution

- Unimodal and Symmetric
  - Bell Curve
- It follows very strict guidelines about how variably the data are distributed around the mean.
- Many variables are nearly normal, but none are exactly normal.

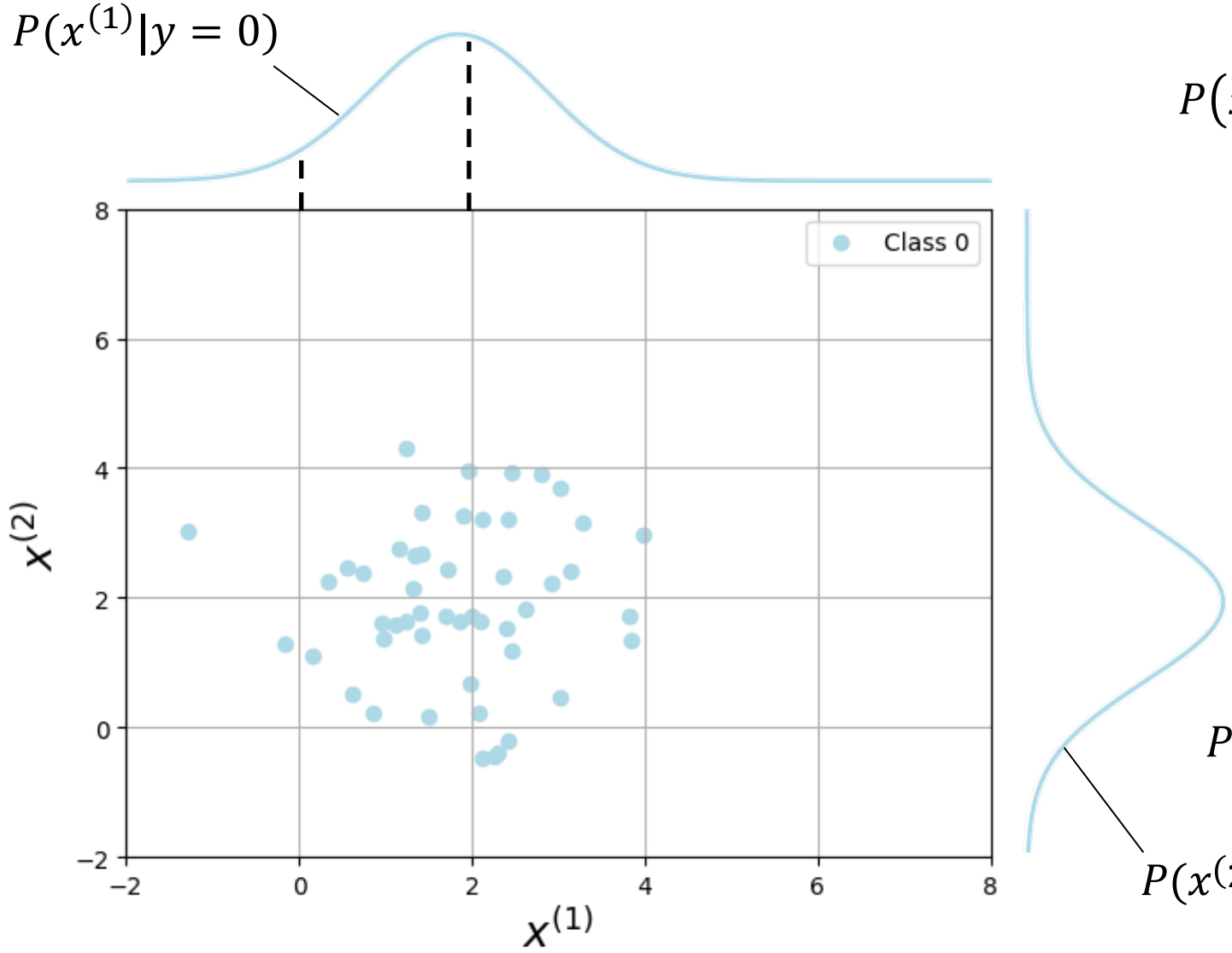
$$N(\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}$$



Normal Distribution (cont.)



Conditional Probability



$$P\left(x^{(1)}\middle|y=0\right) = N(2,1.25)$$

$$= \frac{1}{\sqrt{2\pi \times 1.25^2}} e^{\left(-\frac{(x-2)^2}{2\times 1.25^2}\right)}$$

Examples: $P\left(x^{(1)}=2|y=0\right)=0.319$

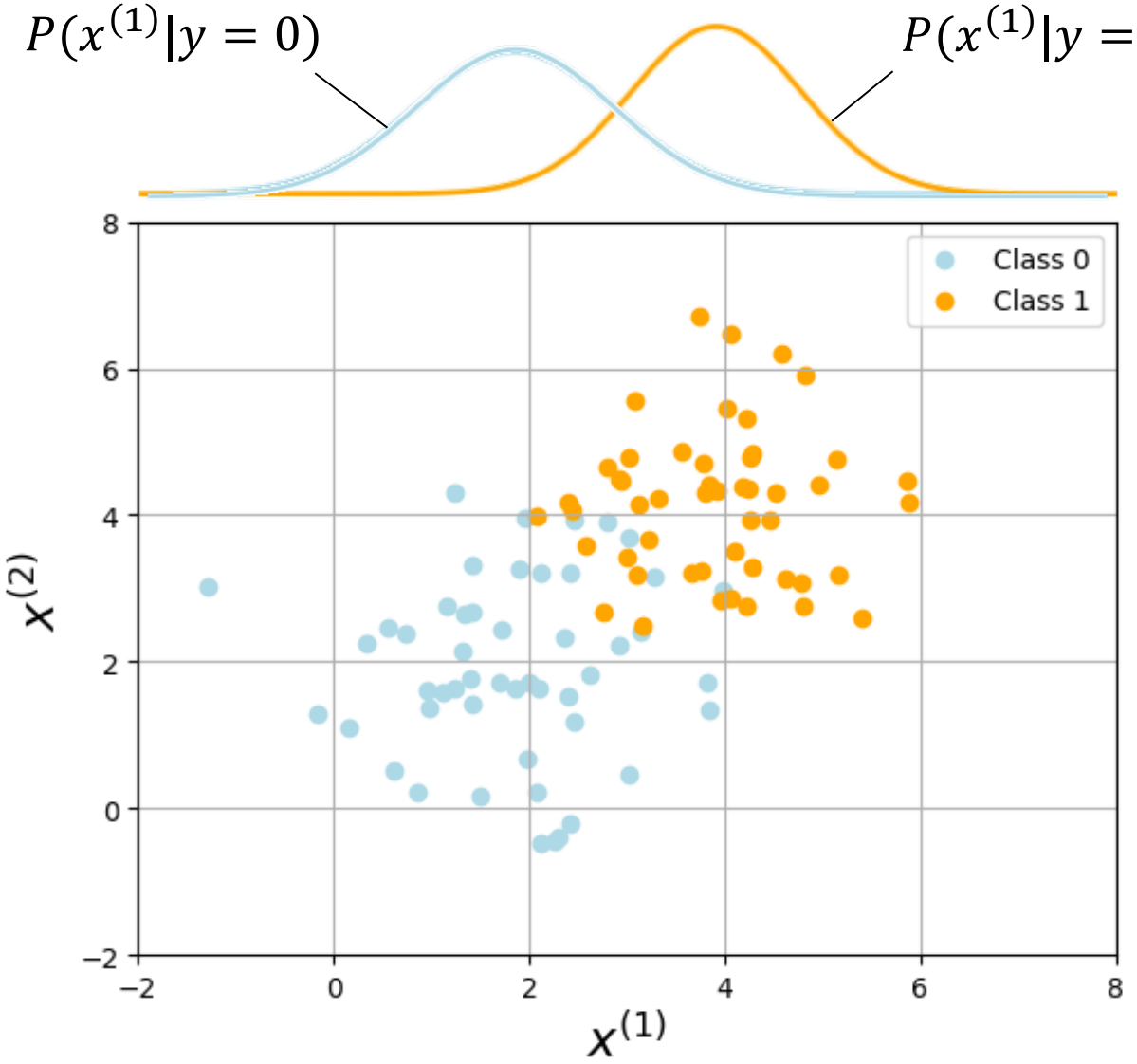
$$P\left(x^{(1)}=0\middle|y=0\right)=0.089$$

$$P\left(x^{(2)}\middle|y=0\right) = N(2,1.25)$$

$P(x^{(2)}|y=0)$

$$= \frac{1}{\sqrt{2\pi \times 1.25^2}} e^{\left(-\frac{(x-2)^2}{2\times 1.25^2}\right)}$$

Conditional Probability (cont.)



$$P\left(x^{(1)}|y=1\right) = N(4,1)$$

$$= \frac{1}{\sqrt{2\pi \times 1^2}} e^{\left(-\frac{(x-4)^2}{2\times 1^2}\right)}$$

$$P\left(x^{(2)}|y=1\right) = N(4,1)$$

$$= \frac{1}{\sqrt{2\pi \times 1^2}} e^{\left(-\frac{(x-4)^2}{2\times 1^2}\right)}$$

$P(x^{(2)}|y=1)$

$P(x^{(2)}|y=0)$

Posterior Probability

$$P\left(y=0|x^{(1)}=4,x^{(2)}=3\right)=P\left(x^{(1)}=4|y=0\right)P\left(x^{(2)}=3|y=0\right)P(y=0)$$

$$P\left(y=0|x^{(1)}=4,x^{(2)}=3\right)=0.087\times0.232\times0.5$$

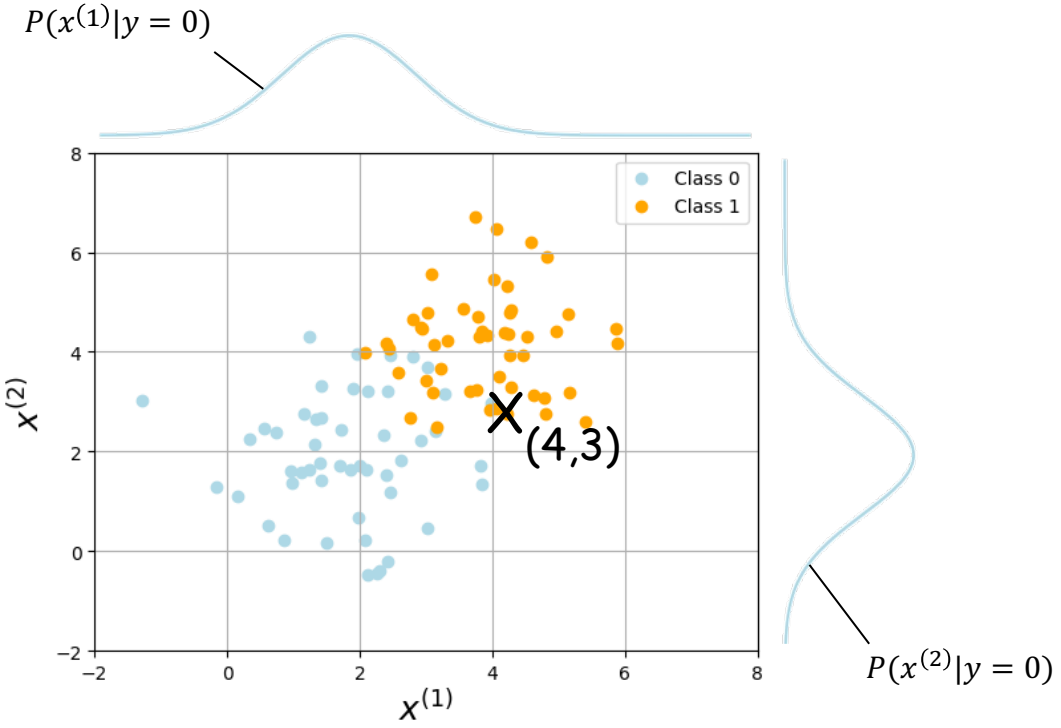
$$=0.010$$

Prior Probability:

$$P\left(y=0\right)=\frac{\text{Samples in Class 0}}{\text{Samples in Class 0}+\text{Samples in Class 1}}$$

$$=\frac{50}{50+50}$$

$$=0.5$$



Posterior Probability (cont.)

$$P\left(y=1|x^{(1)}=4,x^{(2)}=3\right)=P\left(x^{(1)}=4|y=1\right)P\left(x^{(2)}=3|y=1\right)P(y=1)$$

$$P\left(y=1|x^{(1)}=4,x^{(2)}=3\right)=0.399\times0.242\times0.5$$

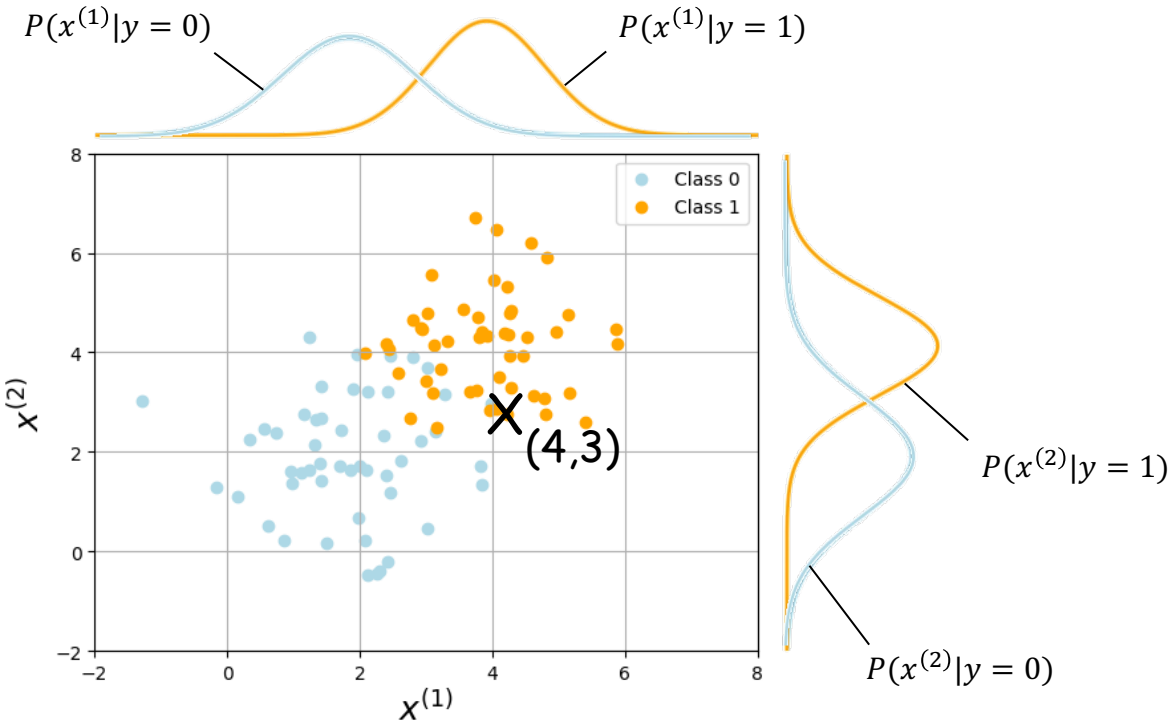
$$=0.048$$

Prior Probability:

$$P(y=1)=\frac{\text{Samples in Class 1}}{\text{Samples in Class 0}+\text{Samples in Class 1}}$$

$$=\frac{50}{50+50}$$

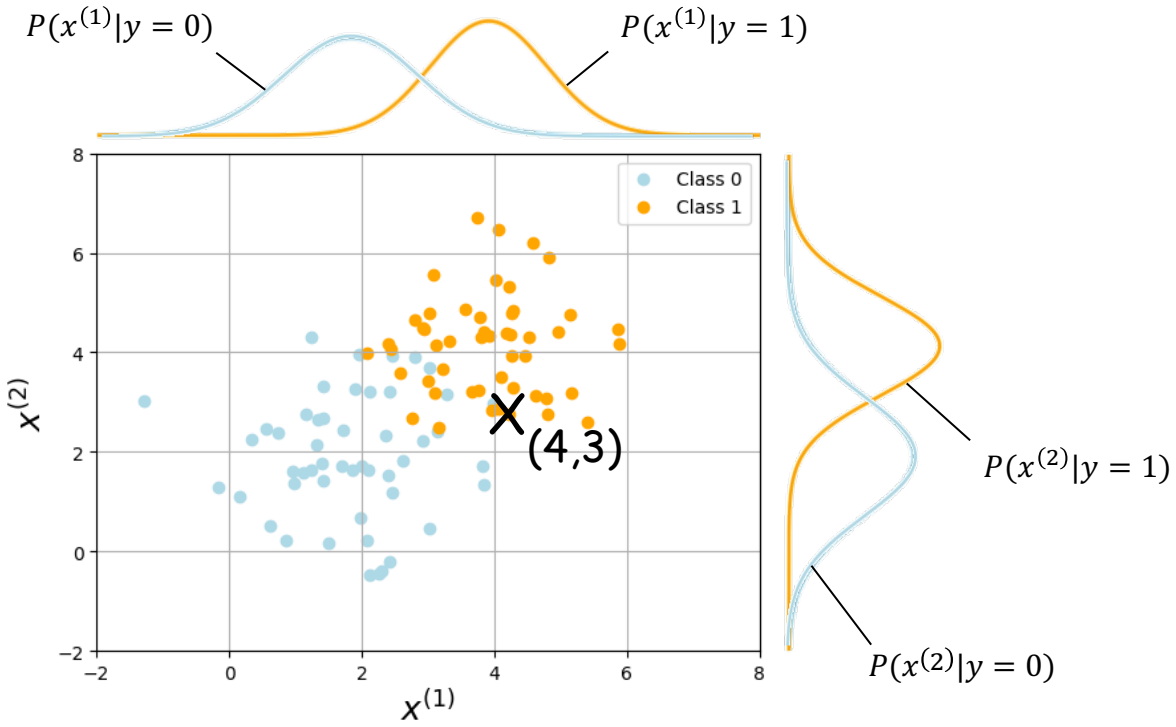
$$=0.5$$



Prediction: Highest Posterior Probability

$$P\,(y = 0|\vec{x} = (4,3)) = 0.010 \qquad P(y = 1|\vec{x} = (4,3)) = 0.048$$

Since 0.048 > 0.010, (4,3) is predicted to belong to class 1.



Classification Rule

Posterior Probability:

$$P(y|\vec{x}) = \left(\prod_{i=1}^{M} P\left(x^{(i)}|y\right)\right) P(y)$$

Q: Why are we using log probabilities?

A: ... Numerical Underflow ...

Log Posterior Probability:

$$\log P(y|\vec{x}) = \log P(y) + \sum_{i=1}^{M} \log P\left(x^{(i)}|y\right)$$

Predicted Class: (Highest Posterior Probability)

$$\hat{y} = \arg\max_{y} \log P(y|\vec{x})$$

Logarithmic Rules: $\log(A \times B) = \log A + \log B$

Pseudocode for Gaussian Naïve Bayes

```
Function Gaussian_Naive_Bayes(new_point, data_points, labels)
    Begin
        // Step 1: Identify unique classes
        classes = UNIQUE(labels)

        // Step 2: Calculate priors, means, and variances for each class
        For each class c in classes do
            // Filter the data points for class c
            X_c = FILTER(data_points WHERE labels == c)

            // Calculate prior probability
            priors[c] = COUNT(X_c) / TOTAL_SAMPLES

            // Calculate mean and variance for each feature
            For each feature i do
                means[c][i] = MEAN(X_c[:, i])
                variances[c][i] = VARIANCE(X_c[:, i])
            End For
        End For

        // Step 3: Calculate probabilities for the new point
        For each class c in classes do
            // Start with the prior probability
            probability[c] = priors[c]

            // Multiply by the likelihood of each feature
            For each feature i do
                probability[c] *= GAUSSIAN(new_point[i], means[c][i], variances[c][i])
            End For
        End For

        // Step 4: Predict the class with the highest probability
        predicted_class = ARGMAX(probability)

        Return predicted_class
    End
```

Python Code Snippet

```
import numpy as np

def gaussian_naive_bayes(new_point, data_points, labels):
    # Step 1: Identify unique classes
    classes = np.unique(labels)
    priors = {}
    stats = {}

    # Step 2: Calculate priors, means, and variances for each class
    for c in classes:
        # Filter the data points for class c
        X_c = data_points[labels == c]

        # Calculate prior probability
        priors[c] = X_c.shape[0] / data_points.shape[0]

        # Calculate mean and variance for each feature
        means = np.mean(X_c, axis=0)
        variances = np.var(X_c, axis=0)

        # Store means and variances in stats
        stats[c] = {'means': means, 'variances': variances}
```

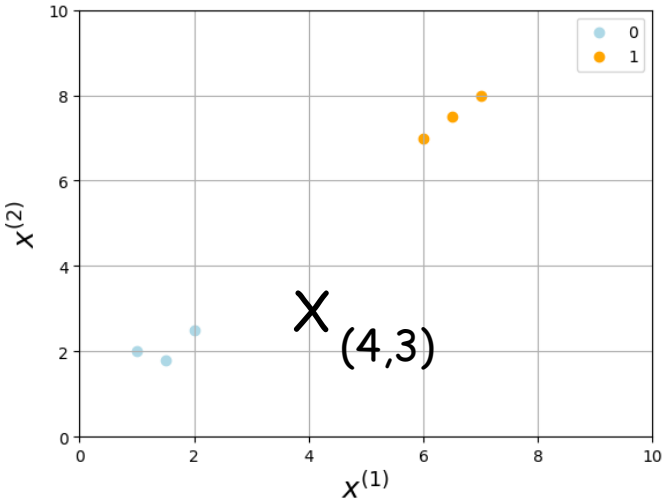
Python Code Snippet (cont.)

```
… cont …

    # Step 3: Calculate probabilities for the new point
    probabilities = {}
    for c in classes:
        # Start with the prior probability
        probabilities[c] = priors[c]

        # Multiply by the likelihood of each feature
        for i in range(len(new_point)):
            mean = stats[c]['means'][i]
            variance = stats[c]['variances'][i]
            probabilities[c] *= (1 / np.sqrt(2 * np.pi * variance)) * \
                                np.exp(-((new_point[i] - mean) ** 2) / (2 * variance))

    # Step 4: Predict the class with the highest probability
    predicted_class = max(probabilities, key=probabilities.get)

    return predicted_class
```

Usage Example

```
# Example Usage
# Define training data and labels
data_points = np.array([[1.0, 2.0], [1.5, 1.8], [2.0, 2.5], [6.0, 7.0], [6.5, 7.5], [7.0, 8.0]])
labels = np.array([0, 0, 0, 1, 1, 1])  # Two classes: 0 and 1

# Define a new point to classify
new_point = np.array([4.0, 3.0])

# Predict the class for the new point
predicted_class = gaussian_naive_bayes(new_point, data_points, labels)

# Output
print(f"New Point: {new_point}")
print(f"Predicted Class: {predicted_class}")
```

```
New Point: [4. 3.]

Predicted Class: 0
```



Step 1: Identifying Unique Classes

```
# Step 1: Identify unique classes
classes = np.unique(labels)
priors = {}
stats = {}
```

$$y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Step 2: Fitting Conditional Probabilities

```
# Step 2: Calculate priors, means, and variances for each class
for c in classes:
    # Filter the data points for class c
    X_c = data_points[labels == c]

    # Calculate prior probability
    priors[c] = X_c.shape[0] / data_points.shape[0]

    # Calculate mean and variance for each feature
    means = np.mean(X_c, axis=0)
    variances = np.var(X_c, axis=0)

    # Store means and variances in stats
    stats[c] = {'means': means, 'variances': variances}
```

Means: $\mu_0^{(1)} = 1.5 \quad \mu_0^{(2)} = 2.1$

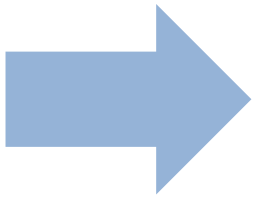
$\mu_1^{(1)} = 6.5 \quad \mu_1^{(2)} = 7.5$

Standard Deviation: $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$

$\sigma_0^{(1)} = 0.408 \quad \sigma_0^{(2)} = 0.294$

$\sigma_1^{(1)} = 0.408 \quad \sigma_1^{(2)} = 0.408$

$$\vec{x} = \begin{bmatrix} 1 & 2 \\ 1.5 & 1.8 \\ 2 & 2.5 \\ 6 & 7 \\ 6.5 & 7.5 \\ 7 & 8 \end{bmatrix} \qquad y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$



Priors: $P(y = 0) = \frac{3}{3+3} = 0.5$

$P(y = 1) = \frac{3}{3+3} = 0.5$

Step 3: Calculating Posterior Probabilities

```
# Step 3: Calculate probabilities for the new point
probabilities = {}
for c in classes:
    # Start with the prior probability
    probabilities[c] = priors[c]

    # Multiply by the likelihood of each feature
    for i in range(len(new_point)):
        mean = stats[c]['means'][i]
        variance = stats[c]['variances'][i]
        probabilities[c] *= (1 / np.sqrt(2 * np.pi * variance)) * \
                            np.exp(-((new_point[i] - mean) ** 2) / (2 * variance))
```

$$P(y=0|\vec{x}=(4,3)) = P\left(x^{(1)}=4\middle|y=0\right)P\left(x^{(2)}=3\middle|y=0\right)P(y=0)$$

$$= 6.875\times10^{-9}\times1.252\times10^{-2}\times0.5$$

$$= 4.451\times10^{-11}$$

$$P(y=1|\vec{x}=(4,3)) = P\left(x^{(1)}=4\middle|y=1\right)P\left(x^{(2)}=3\middle|y=1\right)P(y=1)$$

$$= 6.875\times10^{-9}\times3.756\times10^{-27}\times0.5$$

$$= 1.421\times10^{-35}$$

Step 4: Class Prediction

```
# Step 4: Predict the class with the highest probability
predicted_class = max(probabilities, key=probabilities.get)
```

$$\arg\max_{y}\{P(y=0|\vec{x}=(4,3)), P(y=1|\vec{x}=(4,3))\}$$

$$\Downarrow\Downarrow$$

$$\arg\max_{y}\{4.451\times10^{-11}, 1.421\times10^{-35}\}$$

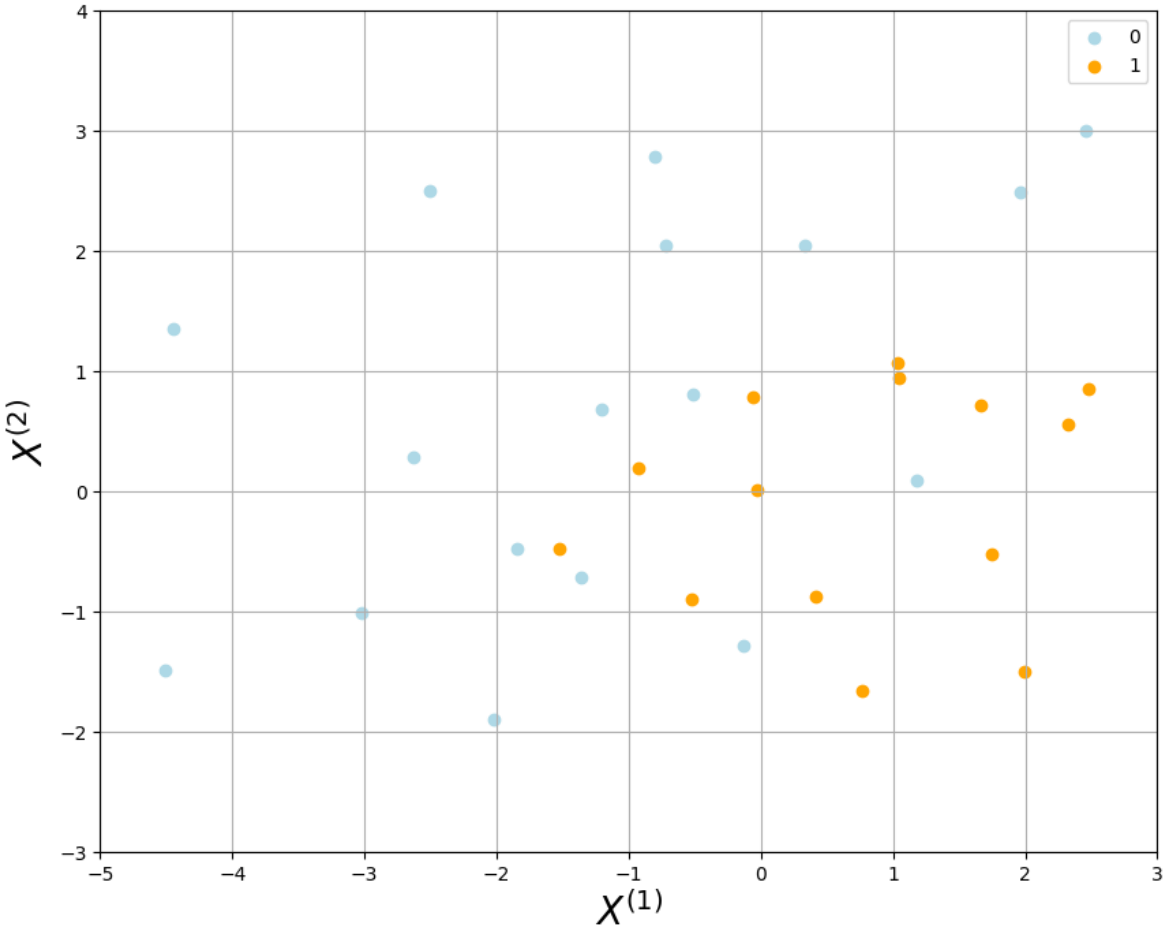
$$0$$

Scikit-learn: Gaussian Naïve Bayes

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

# Step 1: Define the training data and labels
X_train = np.array([[1.0, 2.0], [1.5, 1.8], [2.0, 2.5], [6.0, 7.0], [6.5, 7.5], [7.0, 8.0]])
y_train = np.array([0, 0, 0, 1, 1, 1])  # Classes: 0 and 1

# Step 2: Define the test point
X_test = np.array([[4.0, 3.0]])

# Step 3: Create and train the Gaussian Naive Bayes model
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Step 4: Predict the class for the test point
y_pred = gnb.predict(X_test)

# Step 5: Output the predicted class
print(f"Test Point: {X_test[0]}")
print(f"Predicted Class: {y_pred[0]}")
```

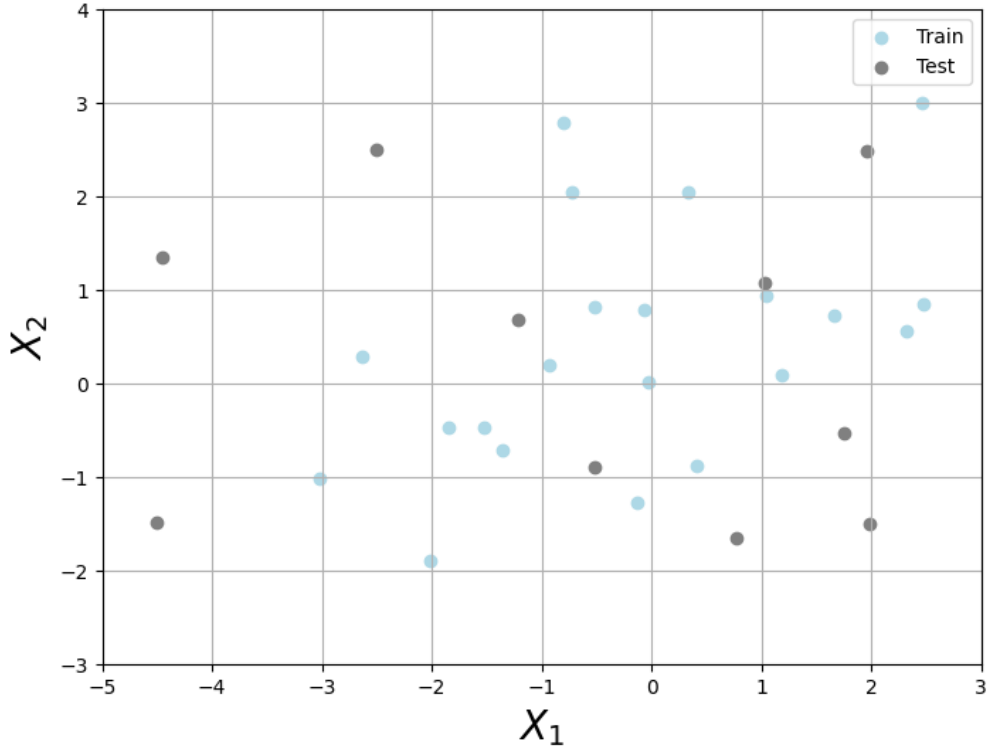

```
New Point: [4. 3.]
Predicted Class: 0
```

Example Dataset



Train, Validation and Test Datasets

```
from sklearn.model_selection import train_test_split

# First, split the data into train (70%) and test (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```



- Train:Test is typicaly either 0.8:0.2 or 0.7:0.3.
- **Test** dataset is a proxy of unseen data, and it will only be used in the final evaluation.
- **Train** dataset is further divided into k folds (e.g., 5 or 10).
- What hyper-parameters are we optimizing? **None for Gaussian Naïve Bayes.** The model simply captures underlying statistics in the data as is. That's all it does. It makes no effort to predict correctly.
- Hence, k-fold cross-validation is irrelevant in the Gaussian Naïve Bayes context.

Decision Boundary

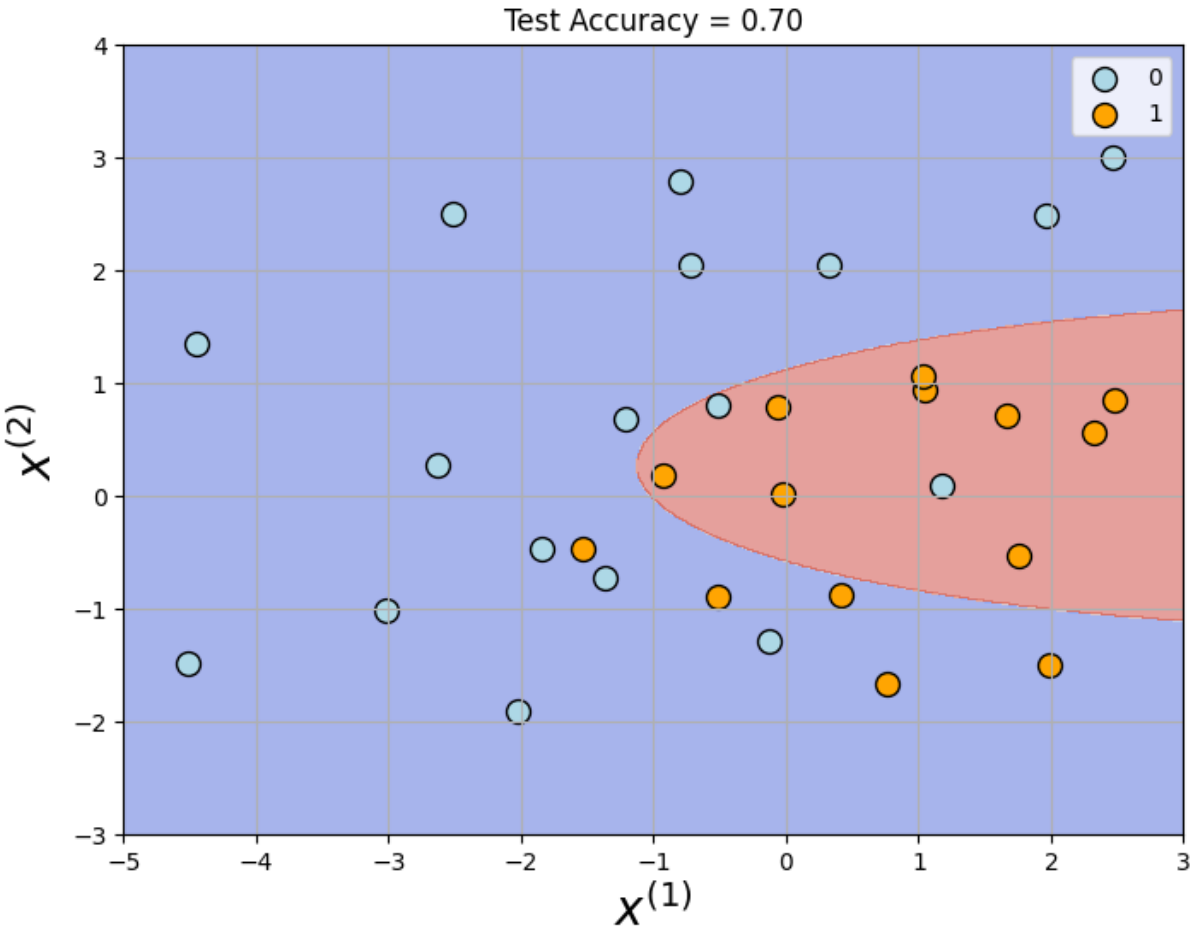
Q: Can accuracy performance of Naïve Bayes be better?

A: No.

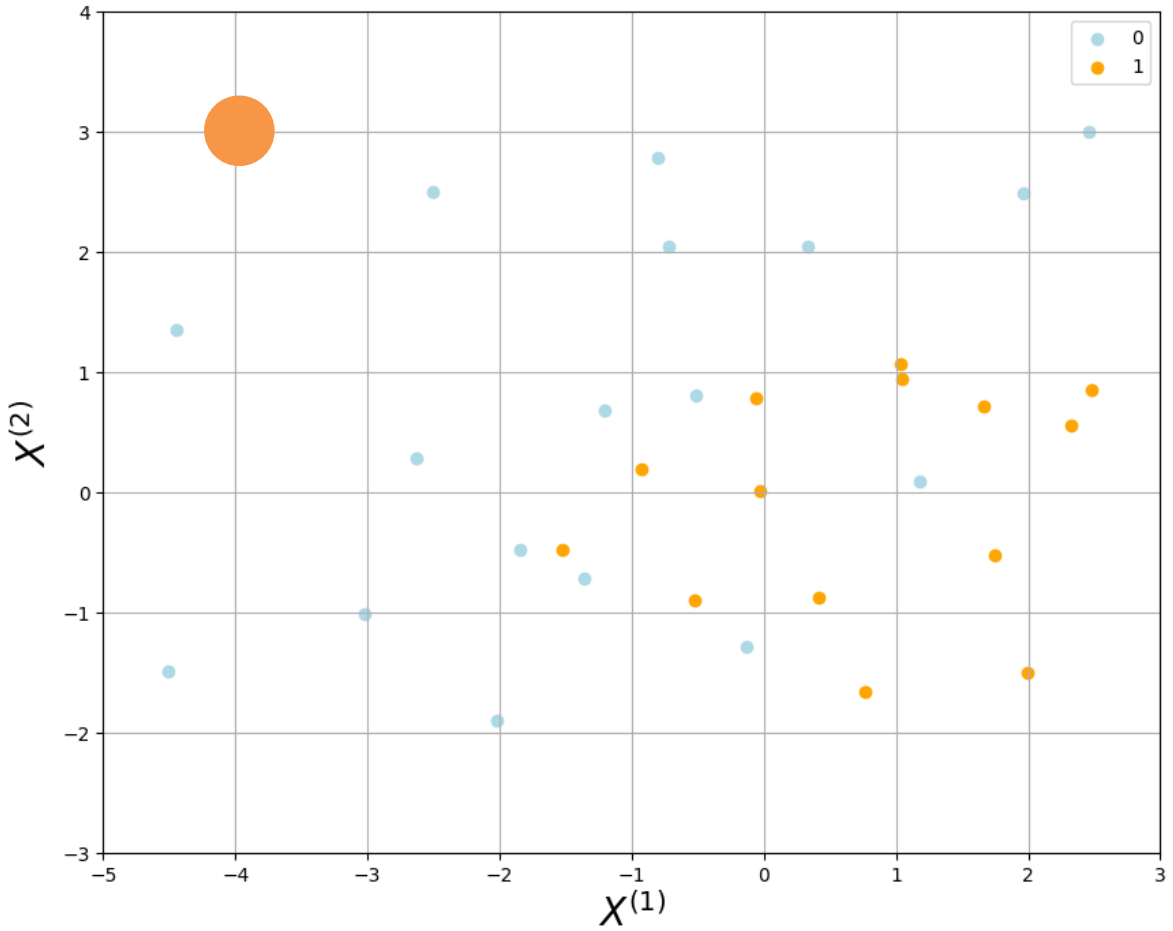
Q: Can we worse than this?

A: (Also) No.

The model simply captures underlying statistics in the data as is. That's all it does. It makes no effort to predict correctly.

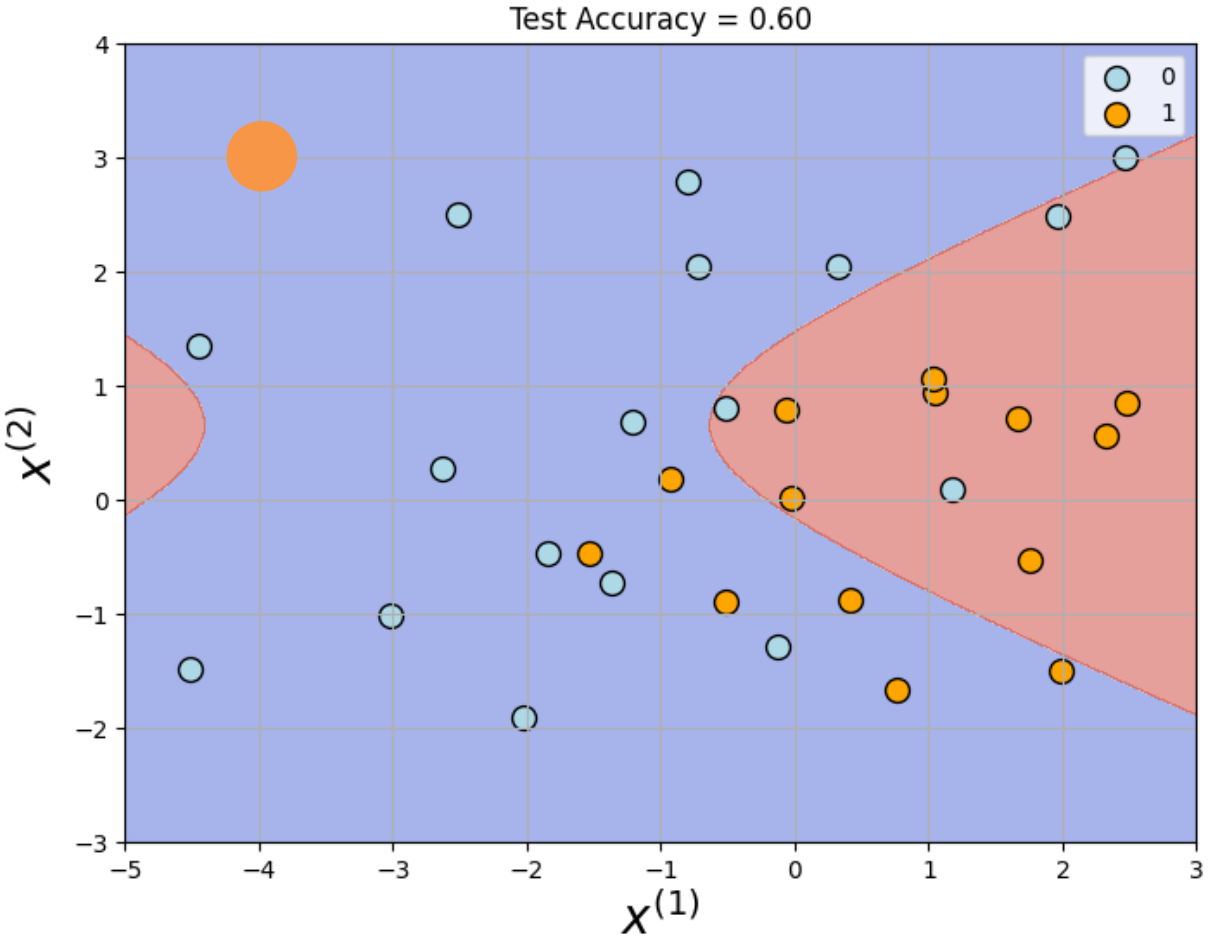


Outliers



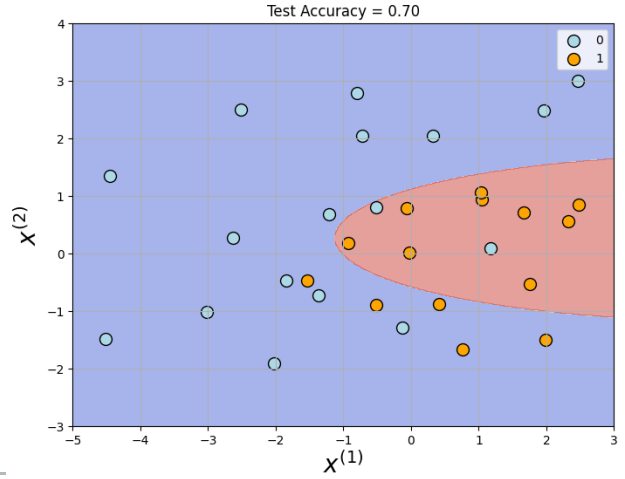
- Outliers can distort underlying statistics by significantly altering mean and variance of the conditional probabilities. The model tries to fit these extreme points rather than the overall data trend.

Outliers (cont.)



Q: What are the underlying causes making our Naïve Bayes performs poorly?

A: We assume our data are normally distributed. The model parameters, i.e. mean and variance, are sensitive to outliers, or so called not statistically robust.



MyGaussianNaiveBayes (Revised)

```
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin

class MyGaussianNaiveBayes(BaseEstimator, ClassifierMixin):
    def __init__(self):
        self.classes = None
        self.priors = {}
        self.stats = {}

    def fit(self, X, y):
        self.classes = np.unique(y)

        for c in self.classes:
            # Filter data points belonging to class c
            X_c = X[y == c]

            # Calculate prior probability
            self.priors[c] = X_c.shape[0] / X.shape[0]

            # Calculate means and variances for each feature
            means = np.mean(X_c, axis=0)
            variances = np.var(X_c, axis=0)

            # Store means and variances
            self.stats[c] = {'means': means, 'variances': variances}

        return self  # Return self to allow chaining

    def _calculate_likelihood(self, x, mean, variance):
        exponent = np.exp(-((x - mean) ** 2) / (2 * variance))
        return (1 / np.sqrt(2 * np.pi * variance)) * exponent
```

MyGaussianNBClassifier (Revised)

```
…cont…

    def predict_proba(self, X):
        probabilities = []
        for x in X:
            posteriors = {}
            for c in self.classes:
                # Start with the log prior
                posterior = np.log(self.priors[c])

                # Add log likelihoods for each feature
                for i in range(len(x)):
                    mean = self.stats[c]['means'][i]
                    variance = self.stats[c]['variances'][i]
                    posterior += np.log(self._calculate_likelihood(x[i], mean, variance))

                posteriors[c] = posterior

            # Normalize to probabilities (softmax-like)
            max_log = max(posteriors.values())
            exp_posteriors = {k: np.exp(v - max_log) for k, v in posteriors.items()}
            total = sum(exp_posteriors.values())
            probabilities.append([exp_posteriors[c] / total for c in self.classes])

        return np.array(probabilities)

    def predict(self, X):
        probabilities = self.predict_proba(X)
        return np.argmax(probabilities, axis=1)
```

Multinomial Naïve Bayes

Spam

Free cash now!
Limited offer!
Cash prize waiting!

Ham

We meet tomorrow, can we?
Have you seen my book?
I will bring cash later.

$P(\text{Spam}|\text{"Limited Cash Offer Now!"})$

$P(\text{Ham}|\text{"Limited Cash Offer Now!"})$

Posterior Probability

Q: How do we calculate these?

A: Multinomial Naïve Bayes

Document Representation: Bag of Words

Spam

Free cash now!
Limited offer!
Cash prize waiting!

Ham

We meet tomorrow, can we?
Have you seen my book?
I will bring cash later.

>10k is common for a vocabulary size.

Unique Words in Corpus

Vocabulary = {book, bring, can, cash, free, have, i, later, limited, meet, my, now, offer, prize, seen, tommorow, waitng, we, will, you }

	book	bring	can	cash	free	have	i	later	limited	meet	my	now	offer	prize	seen	tomorrow	waiting	we	will	you
Free cash now!	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Limited offer!	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
Cash prize waiting!	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
We meet tomorrow, can we?	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	2	0	0
Have you seen my book?	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1
I will bring cash later.	0	1	0	1	0		1	1	0	0	0	0	0	0	0	0	0	0	1	0

Features ($\vec{x}$): M-dimensional vector, where M equals the number of words in the dictionary.

Multinomial Naïve Bayes

$$P(\text{Spam}|\text{offer}, \text{free}, \ldots, \text{cash}) \propto P(\text{offer}|\text{Spam})P(\text{free}|\text{Spam}) \ldots P(\text{cash}|\text{Spam})P(\text{Spam})$$

New Email

Conditions

Q: How do we calculate these?

A: Multinomial Naïve Bayes

Prior Probability:

$$P(\text{Spam}) = \frac{\text{Number of Spam Emails}}{\text{Total Number of Emails}}$$

Conditional Probability:

$$P(\text{offer}|\text{Spam}) = \frac{\text{Count}(\text{offer}|\text{Spam}) + \alpha}{\text{Total Words in Spam Emails} + \alpha|V|}$$

Count(offer|Spam) is 'How many times does the token "offer" appear in all training e-mails that are labelled Spam?'.

It is not the number of spam messages that contain "offer".

Why Multinomial NB uses Token counts instead of document frequency?

Perspective	Token-Count Likelihood (Multinomial NB)	Document-Frequency Likelihood (Bernoulli NB)
What is modelled?	Every *occurrence* of a word is treated as an independent draw from the class-specific distribution $P(w\|c)$.	Only the *presence/absence* of the word in a document is modelled, via $P(\text{appears} = 1\|c)$. Repeated hits add **no** extra evidence.
Sufficient Statistic	Aggregate **token counts** $N_{\text{ic}} = \sum_{d \epsilon c} n_{\text{id}}$.	**Document counts** $df_{\text{ic}} = \{d \epsilon c : n_{\text{id}} > 0\}$.
Signal Retained	Strength of evidence grows with repetition ("offer offer offer" is louder than a single "offer").	All repetitions collapse to one bit.
Best For	Longer texts, bag-of-words spam/news/topic tasks, streaming counts.	Very short texts (tweets, subject lines), IR binary vector space, situations where length bias must be zero.

Example (Spam vs Ham):

Email	Text	"offer" Count
A (Ham)	"... attached is our **offer** letter for your internship ..."	1
B (Spam)	"**OFFER OFFER OFFER**! Limited-time offer now!"	3

Conditional Probability

Spam	Ham
Free cash now! Limited offer! Cash prize waiting!	We meet tomorrow, can we? Have you seen my book? I will bring cash later.

Vocabulary = {book, bring, can, cash, free, have, i, later, limited, meet, my, now, offer, prize, seen, tommorow, waitng, we, will, you }

$$P(\text{cash}|\text{Spam}) = \frac{\text{Count}(\text{cash}|\text{Spam}) + \alpha}{\text{Total Words in Spam Emails} + \alpha|V|}$$

$$= \frac{2+1}{8+1\cdot 20} = \frac{3}{28}$$

$$P(\text{cash}|\text{Ham}) = \frac{\text{Count}(\text{cash}|\text{Ham}) + \alpha}{\text{Total Words in Ham Emails} + \alpha|V|}$$

$$= \frac{1+1}{15+1\cdot 20} = \frac{2}{35}$$

$|V|$: Vocaborary Size

$\propto$: Laplace Smoothing (usually=1)

$\propto$ is for handling zero probabilities for words not seen the training data.

Limited Cash Offer Now! Free Cash!

$$P(y=0|\text{"Limited Cash Offer Now! Free Cash!"}) = P(\text{limited}|y=0)\times P(\text{cash}|y=0)^2\times P(\text{offer}|y=0)$$
$$\times P(\text{now}|y=0)\times P(\text{free}|y=0)\times P(y=0)$$
$$= \frac{1}{35}\times\left(\frac{2}{35}\right)^2\times\frac{1}{35}\times\frac{1}{35}\times\frac{1}{35}\times\frac{1}{2}$$
$$= \frac{2}{35^6}$$
$$= 1.088\times10^{-9}$$

Prior Probability:

$$P(y=0) = \frac{\text{Number of Ham Emails}}{\text{Number of Ham Emails} + \text{Number of Spam Emails}}$$
$$= \frac{3}{3+3}$$
$$= 0.5$$

Spam (1)

Free cash now!
Limited offer!
Cash prize waiting!

Ham (0)

We meet tomorrow, can we?
Have you seen my book?
I will bring cash later.

Posterior Probability (cont.)

Limited Cash Offer Now! Free Cash!

$$P(y=1|\text{"Limited Cash Offer Now! Free Cash!"}) = P(\text{limited}|y=1)\times P(\text{cash}|y=1)^2\times P(\text{offer}|y=1)$$
$$\times P(\text{now}|y=1)\times P(\text{free}|y=1)\times P(y=1)$$
$$= \frac{2}{28}\times\left(\frac{3}{28}\right)^2\times\frac{2}{28}\times\frac{2}{28}\times\frac{2}{28}\times\frac{1}{2}$$
$$= \frac{2^3\times 3^2}{28^6}$$
$$= 1.494\times 10^{-7}$$

Prior Probability:

$$P(y=1) = \frac{\text{Number of Spam Emails}}{\text{Number of Ham Emails} + \text{Number of Spam Emails}}$$
$$= \frac{3}{3+3}$$
$$= 0.5$$

Spam (1)

Free cash now!
Limited offer!
Cash prize waiting!

Ham (0)

We meet tomorrow, can we?
Have you seen my book?
I will bring cash later.

Prediction: Highest Posterior Probability

$$P(y = 0|\text{"Limited Cash offer Now! Free Cash!"}) = 1.088\times10^{-9}$$

$$P(y = 1|\text{"Limited Cash offer Now! Free Cash"}) = 1.494\times10^{-7}$$

Since $1.494\times10^{-7} > 1.088\times10^{-9}$, "Limited Cash Offer Now! Free Cash!" is predicted to belong to class 1, i.e. Spam.

Limited Cash Offer Now! Free Cash!

Pseudocode for Multinomial Naïve Bayes

```
Function MultinomialNaiveBayes(new_sample, data_points, labels, vocabulary, alpha)
    1. Identify unique classes: classes = unique(labels)

    2. For each class c in classes:
       - Compute prior: priors[c] = (# samples of class c) / (total samples)
       - Count how often each word appears in class c: word_counts[c]
       - Track total word occurrences for class c: total_word_counts[c]

    3. For each class c:
       - Initialize probability[c] = priors[c]
       - For each word w in new_sample:
         * If w is in vocabulary:
             probability[c] *= (word_counts[c][w] + alpha)
                               / (total_word_counts[c] + alpha * size(vocabulary))

    4. Return the class with the maximum probability: argmax(probability)
```

Python Code Snippet

```
def multinomial_naive_bayes(new_sample, X_train, y_train, alpha=1.0):
    # Step 1: Identify unique classes
    classes = np.unique(y_train)

    # Initialize dictionaries for priors, word counts, and total word counts
    priors = {}
    word_counts = {}
    total_word_counts = {}

    # Step 2: Calculate priors, word counts, and total word counts for each class
    for c in classes:
        # Extract rows of X_train corresponding to class c
        X_c = X_train[y_train == c]

        # Calculate the prior (class probability)
        priors[c] = X_c.shape[0] / X_train.shape[0]

        # Sum across rows to get total word counts in class c
        word_counts[c] = np.sum(X_c, axis=0)

        # Sum of all word occurrences for class c
        total_word_counts[c] = np.sum(word_counts[c])
```

Python Code Snippet (cont.)

```
… cont …

    # Step 3: Compute the posterior probability for each class
    probabilities = {}
    for c in classes:
        # Start with the prior
        probability = priors[c]

        # For each word in new_sample, multiply by its likelihood
        for i, count in enumerate(new_sample):
            if count > 0:  # Only multiply for words that appear in new_sample
                # Apply Laplace smoothing
                likelihood = (word_counts[c][i] + alpha) / \
                             (total_word_counts[c] + alpha * X_train.shape[1])

                # Raise likelihood to the power of count (if word appears multiple times)
                probability *= likelihood ** count

        probabilities[c] = probability

    # Step 4: Choose the class with the highest probability
    predicted_class = max(probabilities, key=probabilities.get)
    return predicted_class
```

Usage Example

```
# Vocabulary (implicitly):
# Index:  0      1      2     3      4       5    6   7    8    9     10   11   12   13   14
# Word: [limited, cash, now, offer, prize, waiting, can, we, meet, have, you, seen,  i, will, bring]

X_train = np.array([
    [1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],  # "limited cash now"     → Spam
    [0,0,0,1,1,1,0,0,0,0,0,0,0,0,0],  # "offer prize waiting"  → Spam
    [0,2,0,0,0,0,0,0,0,0,0,0,0,0,0],  # "cash cash"            → Spam
    [0,0,0,0,0,0,1,1,1,0,0,0,0,0,0],  # "can we meet"          → Ham
    [0,0,0,0,0,0,0,0,0,1,1,1,0,0,0],  # "have you seen"        → Ham
    [0,1,0,0,0,0,0,0,0,0,0,0,1,1,1]   # "i will bring cash"    → Ham
])

y_train = np.array([1, 1, 1, 0, 0, 0])  # 1 = Spam, 0 = Ham

# New Sample: "limited cash offer"
# BoW vector: limited=1, cash=1, offer=1, everything else=0
new_sample = np.array([1,1,0,1,0,0,0,0,0,0,0,0,0,0,0])

# Predict
predicted_class = multinomial_naive_bayes(new_sample, X_train, y_train, alpha=1.0)
print("Predicted Class:", predicted_class)  # 1 = Spam, 0 = Ham
```

```
Predicted Class: 1
```

Python Code Snippet

```
# Step 1: Identify unique classes
classes = np.unique(y_train)
```

$$y = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Python Code Snippet

```
# Initialize dictionaries for priors, word counts, and total word counts
priors = {}
word_counts = {}
total_word_counts = {}

# Step 2: Calculate priors, word counts, and total word counts for each class
for c in classes:
    # Extract rows of X_train corresponding to class c
    X_c = X_train[y_train == c]

    # Calculate the prior (class probability)
    priors[c] = X_c.shape[0] / X_train.shape[0]

    # Sum across rows to get total word counts in class c
    word_counts[c] = np.sum(X_c, axis=0)

    # Sum of all word occurrences for class c
    total_word_counts[c] = np.sum(word_counts[c])
```

$$y = [1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0]^T$$

Priors: $P(y = 1) = \frac{3}{3+3} = 0.5$

$$P(y = 0) = \frac{3}{3+3} = 0.5$$

$$\text{Count}(1) = [1 \quad 3 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T \Rightarrow \text{Count}_{\text{Total}}(1) = 8$$

$$\text{Count}(0) = [0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1]^T \Rightarrow \text{Count}_{\text{Total}}(0) = 10$$

Python Code Snippet (cont.)

```
# Step 3: Compute the posterior probability for each class
probabilities = {}
for c in classes:
    # Start with the prior
    probability = priors[c]

    # For each word in new_sample, multiply by its likelihood
    for i, count in enumerate(new_sample):
        if count > 0:  # Only multiply for words that appear in new_sample
            # Apply Laplace smoothing
            likelihood = (word_counts[c][i] + alpha) / \
                         (total_word_counts[c] + alpha * X_train.shape[1])

            # Raise likelihood to the power of count (if word appears multiple times)
            probability *= likelihood ** count

    probabilities[c] = probability
```

$$P(\cdot|1) = \frac{\text{Count}(\cdot|1) + \alpha}{\text{Total Words in Class } 1 + \alpha|V|}$$

$$P(\cdot|0) = \frac{\text{Count}(\cdot|0) + \alpha}{\text{Total Words in Class } 0 + \alpha|V|}$$

$$P(\ldots|1) = \left[\frac{2}{26} \quad \frac{4}{26} \quad \frac{2}{26} \quad \frac{2}{26} \quad \frac{2}{26} \quad \frac{2}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26} \quad \frac{1}{26}\right]^T$$

$$P(\ldots|0) = \left[\frac{1}{28} \quad \frac{2}{28} \quad \frac{1}{28} \quad \frac{1}{28} \quad \frac{1}{28} \quad \frac{1}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28} \quad \frac{2}{28}\right]^T$$

$$P(y = 1|\text{"Limited Cash Offer!"}) = \frac{2}{26} \times \frac{4}{26} \times \frac{2}{26} \times \frac{1}{2}$$

$$= 4.55 \times 10^{-4}$$

$$P(y = 0|\text{"Limited Cash Offer!"}) = \frac{1}{28} \times \frac{2}{28} \times \frac{1}{28} \times \frac{1}{2}$$

$$= 4.56 \times 10^{-5}$$

Python Code Snippet (cont.)

```
# Step 4: Choose the class with the highest probability
predicted_class = max(probabilities, key=probabilities.get)
return predicted_class
```

$P(y = 0|\text{"Limited Cash Offer!"}) = 4.56\times10^{-5}$ $P(y = 1|\text{"Limited Cash Offer!"}) = 4.55\times10^{-4}$

$4.55\times10^{-4} > 4.56\times10^{-5}$, hence "Limited Cash Offer!" is 1, i.e. Spam.

Scikit-learn: Multinomial Naïve Bayes

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Define the dataset (text samples and their labels)
texts = [
    "limited cash now",       # Spam
    "offer prize waiting",    # Spam
    "cash cash",              # Spam
    "can we meet tomorrow",   # Ham
    "have you seen my book",  # Ham
    "I will bring cash later" # Ham
]
labels = [1, 1, 1, 0, 0, 0]  # 1 = Spam, 0 = Ham

# Preprocess text using CountVectorizer (Bag-of-Words)
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)  # Convert text to bag-of-words vectors
vocabulary = vectorizer.get_feature_names_out()

# Step 3: Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.33, random_state=42)

# Step 4: Train the Multinomial Naive Bayes model
model = MultinomialNB(alpha=1.0)  # Laplace smoothing with alpha=1.0
model.fit(X_train, y_train)
```

Scikit-learn: Multinomial Naïve Bayes (cont.)

```
# Test the model
y_pred = model.predict(X_test)

# Output results
print("Vocabulary:", vocabulary)
print("\nTest Samples:")
print(X_test.toarray())  # Bag-of-words representation of test samples
print("\nPredicted Classes:", y_pred)
```


```
Vocabulary: ['book' 'bring' 'can' 'cash' 'have' 'i' 'later' 'limited' 'meet' 'my' 'now' 'offer' 'prize'
'seen' 'tomorrow' 'waiting' 'we' 'will' 'you']

Test Samples:
[[0 0 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0]]

Predicted Classes: [1]
```

Numerical Underflow

$P(x^{(i)}|y)$ are less than 1. $P(y|\vec{x}) = \prod_{i=1}^{|V|} P(x^{(i)}|y)$, hence $P(y|\vec{x}) \rightarrow 0$.

This could lead to numerical underflow. *Try: 0.1**1000 (i.e. 1.0×10^{-1000}) in Python.*

Q: How can we handle very very small numbers?

A: We take Log. 1000xLog(0.1) = 1,000x(-1) = -1,000.

Log Likelihood Function:

$$\log P(y|\vec{x}) \quad = \quad \log(y) + \sum_{i=1}^{|V|} \log P(x^{(i)}|y)$$

MyMultinomialNaiveBayes (Revised)

```
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from scipy.special import logsumexp   # robust log-∑-exp

class MyMultinomialNaiveBayes(BaseEstimator, ClassifierMixin):
    def __init__(self, alpha=1.0):
        self.alpha = alpha

    def fit(self, X, y):
        X = np.asarray(X, dtype=np.float64)                 # dense; convert if needed
        y = np.asarray(y)
        self.classes_ = np.unique(y)
        n_classes = len(self.classes_)
        n_features = X.shape[1]

        self.log_prior_ = np.zeros(n_classes)               # log P(c)
        self.log_likelihood_ = np.zeros((n_classes, n_features))

        for idx, c in enumerate(self.classes_):
            X_c = X[y == c]
            self.log_prior_[idx] = np.log(X_c.shape[0]) - np.log(X.shape[0])     # log prior P(c)

            class_counts = X_c.sum(axis=0)                  # 1 × n_features
            smoothed = class_counts + self.alpha            # smoothed word counts
            denom = smoothed.sum()                          # total tokens + α·V

            self.log_likelihood_[idx, :] = np.log(smoothed) - np.log(denom)      # log P(w_i | c)


        return self
```

MyMultinomialNaiveBayes (Revised)

```
... cont ...

    def predict_proba(self, X):
        X = np.asarray(X, dtype=np.float64)
        # log P(c) + Σ_i n_i · log P(w_i | c)
        log_joint = (X @ self.log_likelihood_.T) + self.log_prior_

        # convert log-joint to posterior via log-softmax
        log_post  = log_joint - logsumexp(log_joint, axis=1, keepdims=True)
        return np.exp(log_post)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), axis=1)]
```

Stanford's IMDB Dataset

50k Movie Reviews

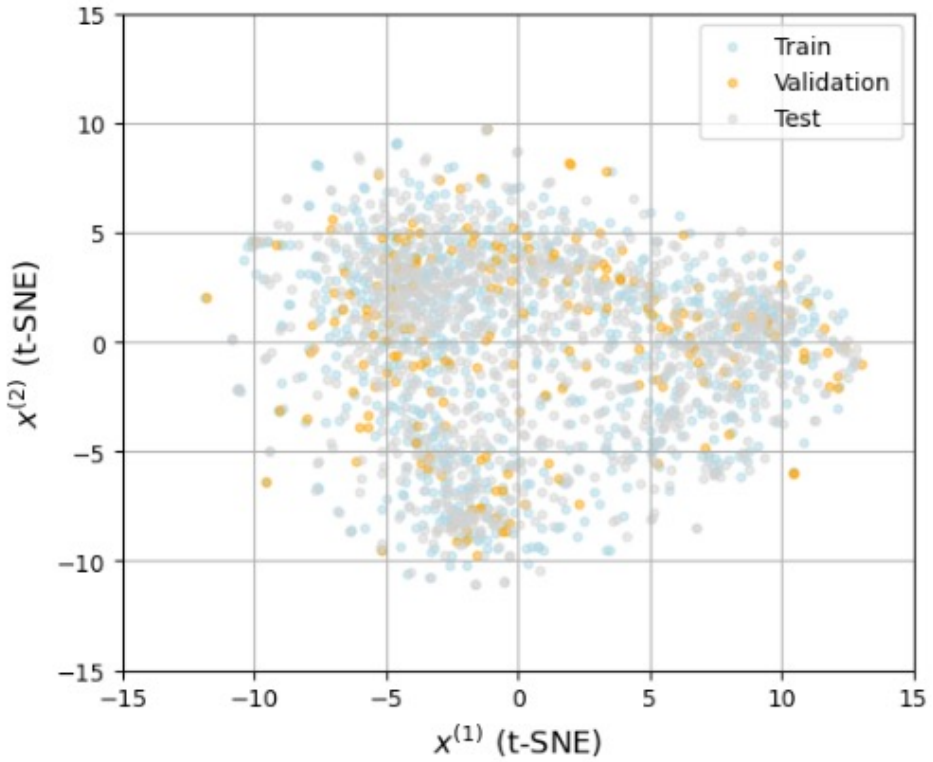
- 25k for Train and 25k for Test
- 2 Classes, Positive and *Negative* Sentiments
- 44,490 Vocabulary Size

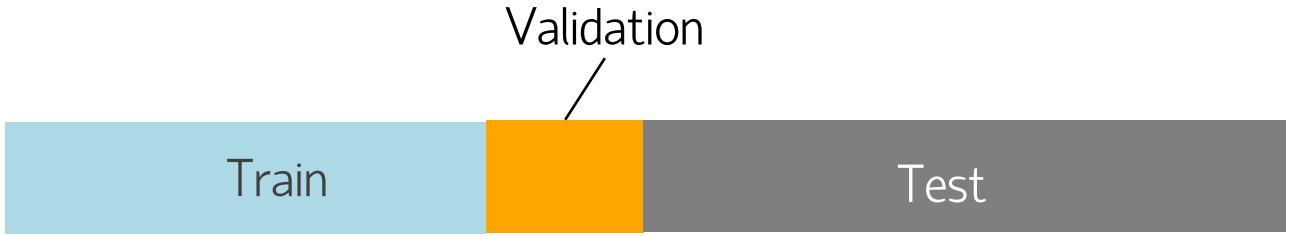

```
If you like adult comedy cartoons, like South Park, then this is nearly a similar
format about the small adventures of three teenage girls at Bromwell High.
Keisha, Natella and Latrina have given exploding sweets and behaved like bitches,
I think Keisha is a good leader. There are also small stories going on with the
teachers of the school. There's the idiotic principal, Mr. Bip, the nervous
Maths teacher and many others. The cast is also fantastic, Lenny Henry's Gina
Yashere, EastEnders Chrissie Watts, Tracy-Ann Oberman, Smack The Pony's Doon
Mackichan, Dead Ringers' Mark Perry and Blunder's Nina Conti. I didn't know this
came from Canada, but it is very good. Very good!
```

https://ai.stanford.edu/~amaas/data/sentiment/

Train, Validation and Test Datasets

```
from sklearn.model_selection import train_test_split

# First, split the data into train (80%) and validation (20%)
X_train, X_val, y_train, y_val= train_test_split(X_train, y_train, test_size=0.2, random_state=42, stratify=y)
```

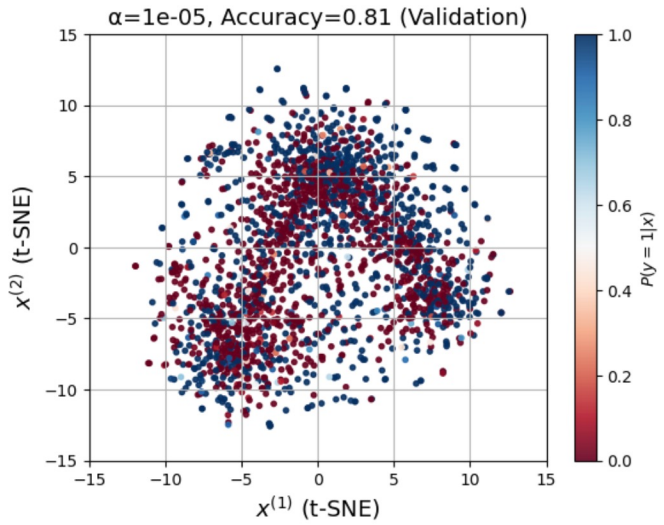


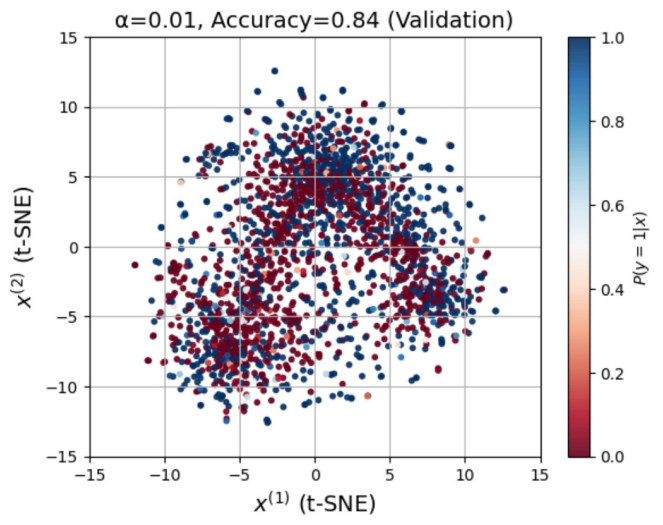


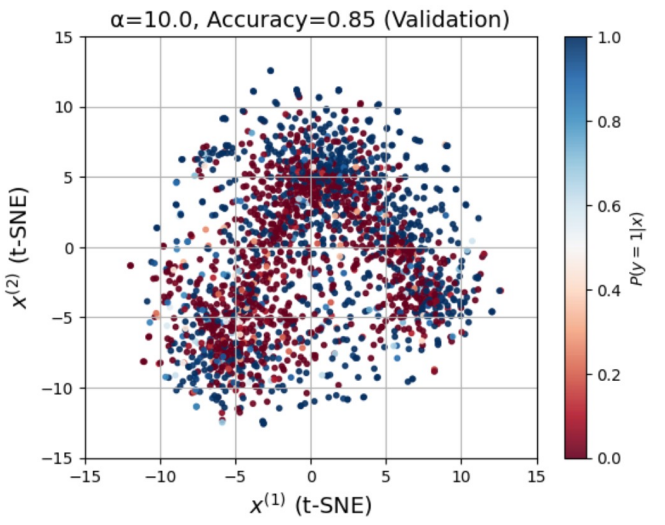
- (Train:Validation):Test is (0.4:0.1):0.5.

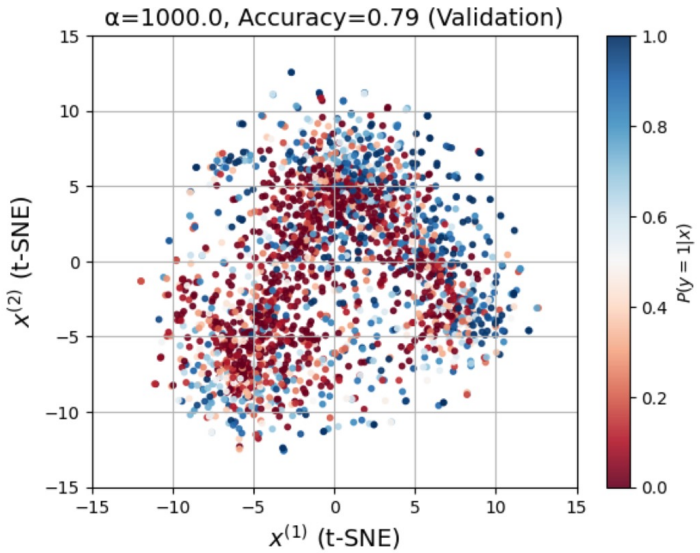
- **Test** dataset is a proxy of unseen data, and it will only be used in the final evaluation.
- We train our ML model on the **Train** dataset.
- **Validation** datest is used to fine-tune or optimise the ML model. Here, we find a smoothing alpha α that will result in the best performance on the Validation dataset.

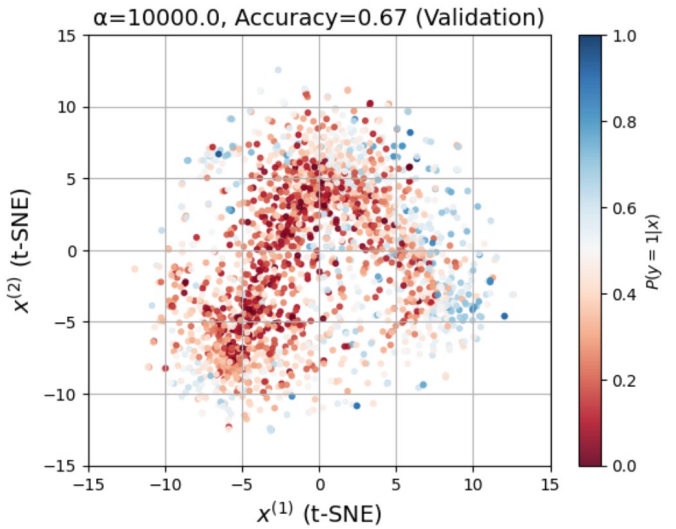
Model Complexity

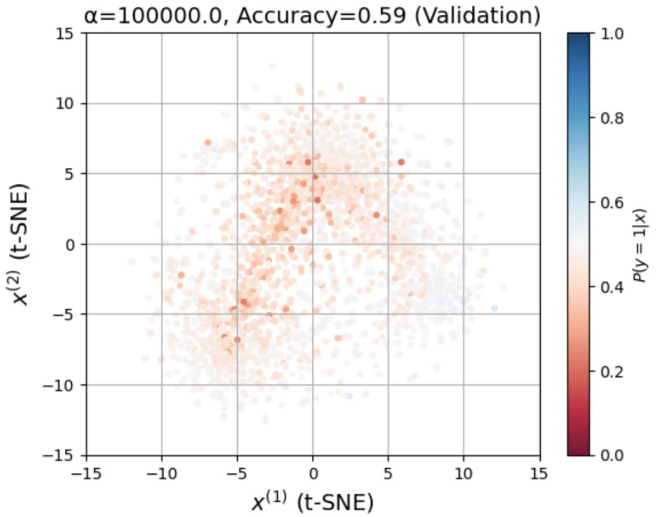




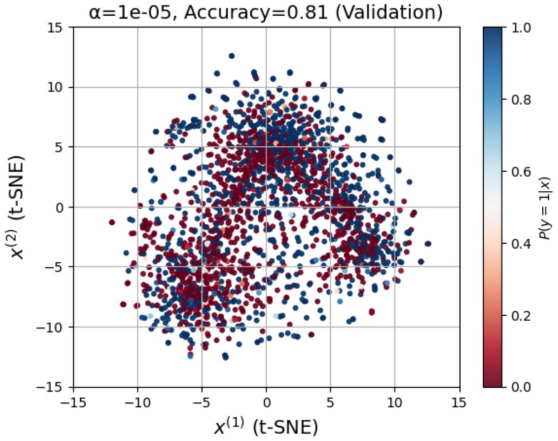




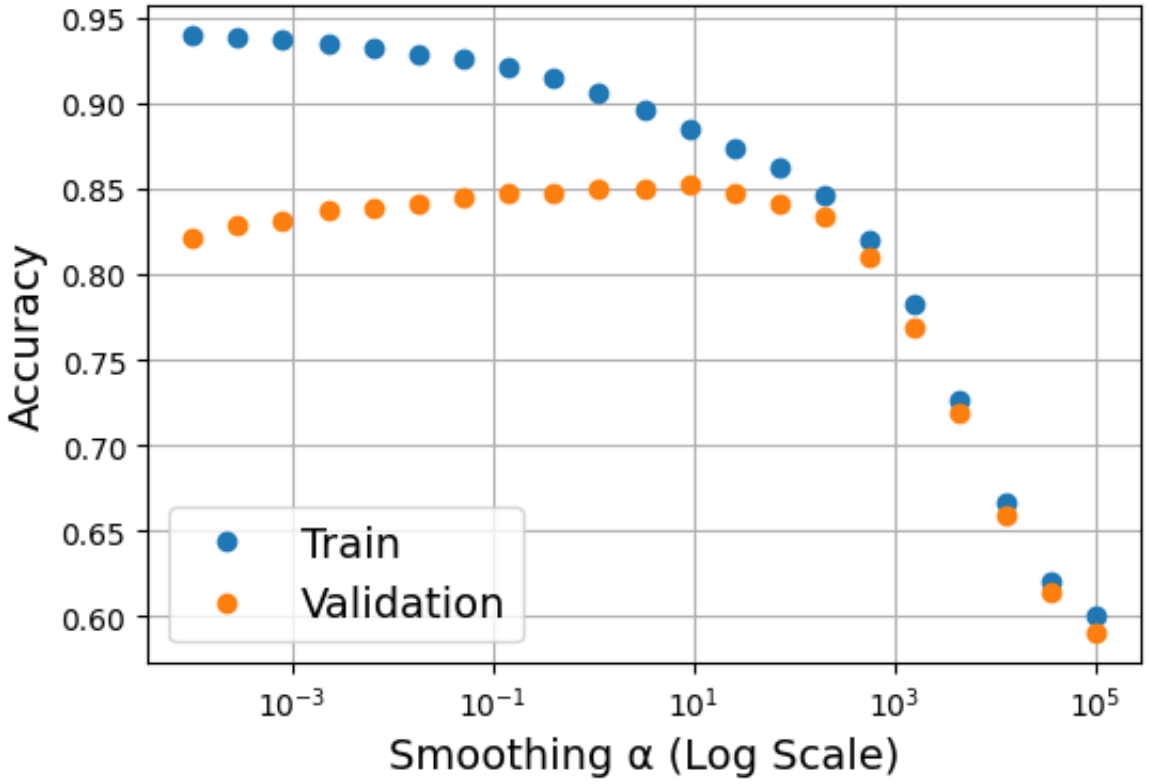


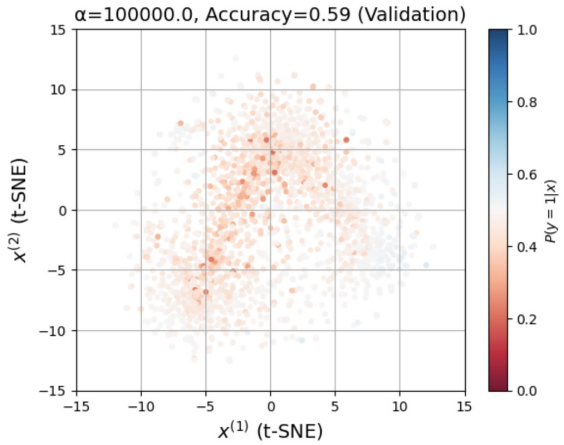


Bias-Variance Trade-Offs



- High Variance, Low Bias
- Complex Probability Density
- High Train Accuracy
- Low Validation Accuracy

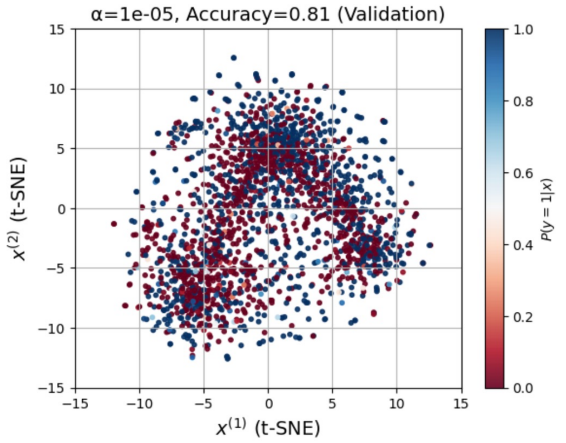




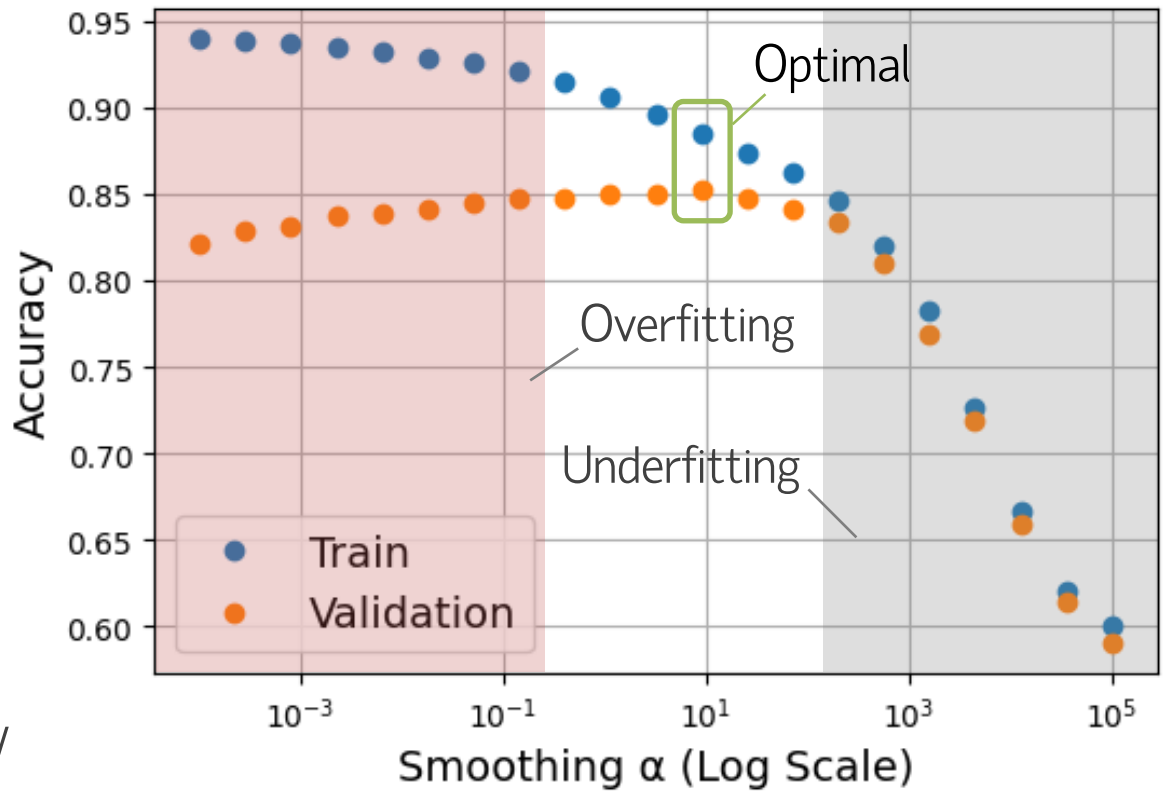
- High Bias, Low Variance
- Simple Probability Density
- Low Train Accuracy
- Low Validation Accuracy

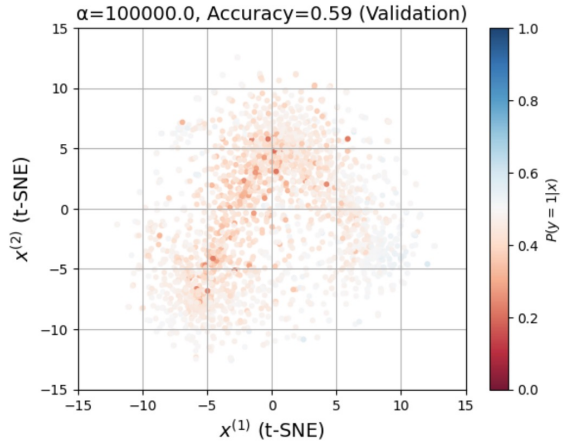
Overfitting vs Underfitting

- **Overfitting** happens when a small smoothing alpha α leads to high training accuracy but poor validation accuracy due to overly complex probability density functions that fail to generalise.
- **Underfitting** occurs when a large smoothing alpha α leads to overly simple probability density functions, resulting in low training and validation accuracies.



- High Variance, Low Bias
- Complex Probability Density
- High Train Accuracy
- Low Validation Accuracy





- High Bias, Low Variance
- Simple Probability Density
- Low Train Accuracy
- Low Validation Accuracy

Why Unigram ≠ Enough for Sentiment or Topic Nuance

A pure bag-of-words (unigrams) treats each token independently, so "good" and "not good" contribute almost the same evidence. Likewise, "important decision" vs "unimportant decision" share the token *decision*.

What Plain Unigrams Miss	How Small *N-Grams* Fix It	Practical Recipe (scikit-learn)
Negation flips polarity "good" vs "not good"; "like" vs "don't like".	The bigram *not good* becomes its own feature, giving the classifier a separate weight from *good*.	CountVectorizer(ngram_range=(1,2)) keeps unigrams and bigrams; set min_df≥2 to drop ultra-rare pairs.
Intensifiers change magnitudeplain *good* vs **"very good", "so bad", "absolutely fantastic"*.	Bigram or trigram captures *very good*, *so bad* etc., allowing a larger positive/negative weight than the base adjective.	Keep intensifiers in the stop-word list; or pre-combine: regex `very\ (good

Example: *"I do not really like this movie at all."*

Feature Type	Tokens Extracted
Unigrams (1-grams)	i, do, not, really, like, this, movie, at, all
Bigrams (2-grams)	i do, do not, not really, really like, like this, this movie, movie at, at all

TF-IDF (Term Frequency-Inverse Document Frequency)

Term Frequency (TF): Measures how often a word appears in a document. A higher frequency suggests greater importance. If a term appears frequently in a document, it is likely relevant to the document's content.

$$TF\,(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Common Locally

Inverse Document Frequency (IDF): Reduces the weight of common words across multiple documents while increasing the weight of rare words. If a term appears in fewer documents, it is more likely to be meaningful and specific.

$$IDF(t,d) = \log \frac{\text{Total number of documents in corpus } D}{\text{Number of documents containing term } t}$$

Rare Globally

Unlike simple word frequency (Bag-of-Words), *TF-IDF* balances common and rare words to highlight the most meaningful terms.

TF-IDF: Example

#1: The cat sat on the mat.
#2: The dog played in the park.
#3: The cat and dog are both great.

The word "cat" appear 1 time. The total of terms in document #1 is 6.

$\rightarrow \mathrm{TF}(\text{"cat"}, \#1) = \frac{1}{6}.$

The total number of documents in the corpus (D) is 3. The number of documents containing the term "cat" is 2.

$\rightarrow IDF(\text{"cat"}, D) = \log\frac{3}{2} = 0.405.$

$\mathrm{TF{-}IDF}\ (\text{"cat"}, \#1, D) = \frac{1}{6} \times \log\frac{3}{2} = 0.068.$

TF-IDF: Example

#1: The cat sat on the mat.
#2: The dog played in the park.
#3: The cat and dog are both great.

Unique Words in Corpus

$$\text{Vocabulary} = \{\text{and, are, both, cat, dog, great, in, mat, on, park, played, sat, the}\}$$

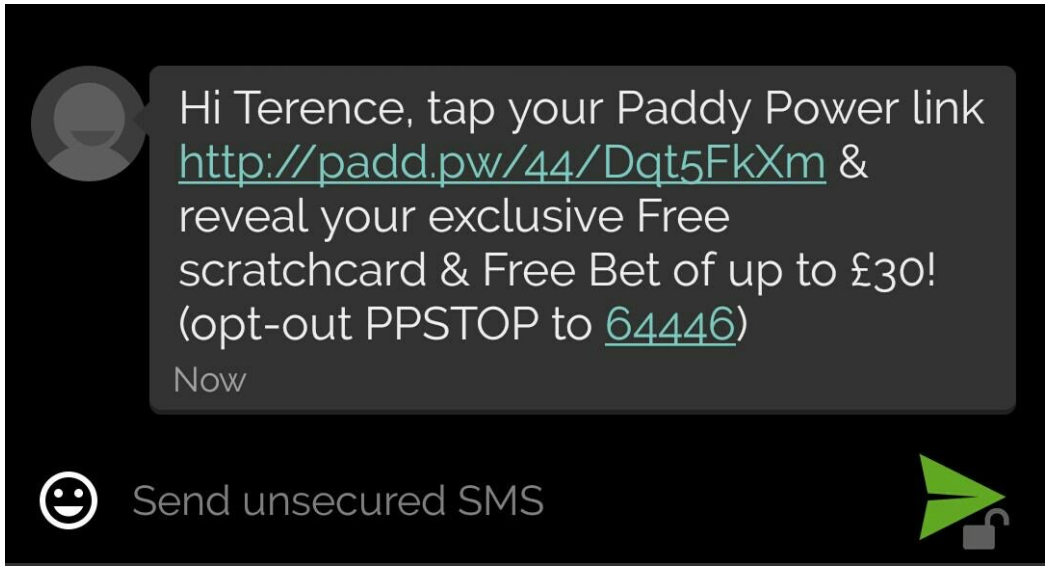
		and	are	both	cat	dog	great	in	mat	on	park	played	sat	the
The cat sat on the mat.	TF	$\frac{0}{6}$	$\frac{0}{6}$	$\frac{0}{6}$	$\frac{1}{6}$	$\frac{0}{6}$	$\frac{0}{6}$	$\frac{0}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{0}{6}$	$\frac{0}{6}$	$\frac{1}{6}$	$\frac{2}{6}$
	IDF	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{2}$	$\log\frac{3}{2}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{1}$	$\log\frac{3}{3}$
	TF-IDF	0	0	0	$\frac{1}{6}\times\log\frac{3}{2}$	0	0	0	$\frac{1}{6}\times\log\frac{3}{1}$	$\frac{1}{6}\times\log\frac{3}{1}$	0	0	$\frac{1}{6}\times\log\frac{3}{1}$	$\frac{2}{6}\times\log\frac{3}{3}$

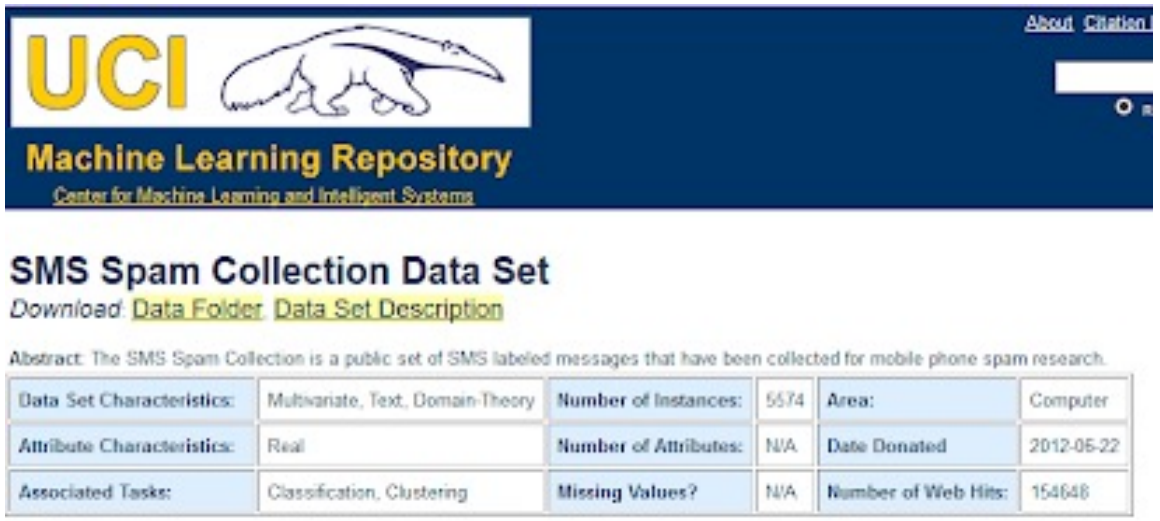
(2/6)x0 = 0

SMS Spam Dataset

5,574 SMS Messages

- 2 Classes, *Spam* and *Ham*
- 7,451 Vocabulary Size





`https://archive.ics.uci.edu/dataset/228/sms+spam+collection`

Bag-of-Words vs TF-IDF

Model + Features	F1 Score	What the score tells us	SMS-Specific Rationale
MNB + TF (α = 1)	0.949	NB excels when a *single* "spam clue" (free, win, cash) should immediately trigger the spam label. Raw counts let those clues contribute their full log-likelihood.	• Messages are **very short** (≈ 15 tokens); a spam word often appears **once** → NB's probability jump is decisive.• Spam vocabulary is **repetitive and class-specific**; IDF would unnecessarily dampen those high-frequency spam tokens.
MNB + TF-IDF (α = 0.1)	0.944	Still strong, but IDF down-weights spam words that appear in *many* spam messages, so NB's evidence is slightly muted.	• Words like "call" and "now" are common to both classes; TF-IDF helps by shrinking their weight – yet it *also* shrinks genuinely useful spam cues that aren't globally rare, costing a little recall/precision.
LR + TF-IDF (C = 1000)	0.939	LR benefits from TF-IDF because it reduces the dominance of background words; high C shows the model wanted minimal regularisation once features were re-weighted.	• With TF-IDF each SMS vector is length-normalised → LR no longer biases toward longer ham texts.• Still trails NB because LR needs more data to learn strong weights for every spam word; NB's generative counts capture that with fewer examples.
LR + TF (C = 1000)	0.936	Lowest score: raw counts make very common tokens ("call", "now") huge; LR can only counter by giving those words tiny or negative weights, a harder optimisation problem in this tiny dataset.	• Sparse, high-dimensional space + few spam samples ⇒ LR risks over- or under-weighting features despite the high C.• Without IDF, long ham messages get larger feature norms, nudging LR toward ham predictions.

Summary

- Naïve Bayes (NB) applies Bayes' theorem under the conditional-independence assumption, turning joint likelihoods into simple 1-D products. Training reduces to counting frequencies or estimating a mean + variance, so NB scales to tens-of-thousands of features in seconds and delivers well-calibrated posterior probabilities.
- Multinomial Naïve Bayes (MNB) interprets every token count as independent evidence; α-smoothing guards against zero probabilities. Because a single rare spam word can flip the posterior, Multinomial NB often beats heavier models on short, sparse documents (spam, sentiment, topic tagging) and needs little data to do so.
- Gaussian Naïve Bayes (GNB) assumes each feature is normally distributed per class, storing only μ and σ^2. It handles real-valued inputs instantly, yet the Gaussian fit is sensitive to outliers—even a few extreme points can skew means/variances and degrade accuracy, so robust preprocessing is essential.
- Both MNB and GNB train ~100× faster than discriminative models, e.g. Logistic Regression, and work well when samples are scarce or features far outnumber observations. Word- or feature-likelihood tables are directly interpretable and can highlight the strongest signals for each class.
- MNB' strong multinomial assumptions yield high bias but low variance: it may underfit when features are correlated or not truly multinomial, yet rarely overfits—even on small data. Proper Laplace (α) smoothing is key: too small α overemphasizes rare counts (risking overfitting), while too large α or heavy smoothing produces a bland, underfitted model.