**Naive Bayes**

**Naive Bayes** es uno de los **algoritmos más simples de clasificación**. Está basado en el **Teorema de Bayes** con un **supuesto de independencia entre los predictores**. Es **fácil de construir** y es particularmente **útil para conjuntos de datos muy grandes**. Asume que **cualquier feature** no está relacionada con ninguna otra. Esto es lo que lo hace ***Naive* (ingenuo)**. Todas **contribuyen de manera independiente a la probabilidad** de que una **instancia pertenezca a una clase**, incluso si las features dependen entre sí o dependen de la existencia de otras features.

Para que funcione, es importante que se cumplan los supuestos de Naive Bayes. Sus principales ventajas son:

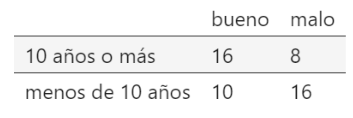
* **Rapidez** para entrenar y predecir.
* Dan una **predicción probabilística** (probabilidades para cada clase)
* **Sencillos** de Interpretar
* **No implican “tunear” ningún parámetro**.

Sirve como modelo **baseline** para problemas de clasificación. Si da bien, nos quedamos con él, si no, podemos mejorar a partir del mismo.

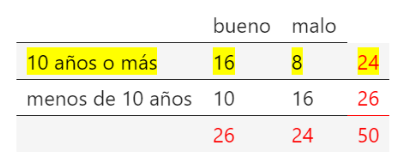
Para que dé bien, es necesario que:

1. **Se cumplan los supuestos de Naive Bayes** (raramente ocurre).
2. **Las clases estén muy bien separadas** (el modelo no requiere tanta complejidad).
3. **Datos con muy Alta Dimensionalidad**. Esto vuelve la complejidad menos importante, ya que hay mucha información para cada observación.

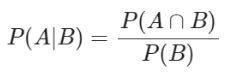
**Probabilidad Condicional:** Calcular probabilidades condicionales es el mismo proceso que el que hacemos cuando elegimos a un vendedor online según su reputación: A partir de esta información, actualizamos nuestra creencia (probabilidad) de que el vendedor es o no confiable. Si tenemos una pequeña población de vendedores en internet, de las cuales para cada vendedor sabemos si lleva o no más de 10 años vendiendo y se sabe si es bueno o malo:



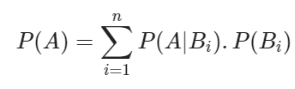
Queremos saber cuál es la probabilidad de que no sea un oportunista, dado que lleva 10 o más años vendiendo en el sitio. **Condicionar** es **fijar la fila que voy a mirar**.Sólo voy a considerar la proporción de vendedores buenos dentro de la primera fila:



La probabilidad de un evento A (que no sea u vendedor oportunista), dado un evento B (que venda hace 10 o más años) es: los casos favorables a A, dado B (16) / Todos los casos donde se da B (24). Se expresa matemáticamente así: La **probabilidad de A, dado B**



**Teorema de Probabilidad Total:** Si tenemos B1,…,Bn una partición de Ω, tal que P(Bi) ≠ 0, desde i = 1 hasta n; para cualquier evento A, entonces:



Si la partición del espacio muestral consta únicamente de los elementos B y Bc, entonces la **fórmula del teorema de probabilidad total** se reduce a:



La **independencia de eventos** es equivalente a decir que la **ocurrencia de un evento no afecta** la **probabilidad de ocurrencia de otro** evento. Este **concepto importante**, en algunos casos va a **simplificar considerablemente** el **cálculo de probabilidades conjuntas**.

Los eventos A y B son independientes si se cumple esta igualdad:



*Chain Rule* para eventos:

Para 2 eventos A y B, 

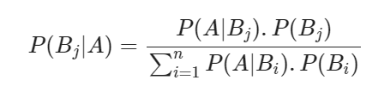
Si tenemos más de 2 eventos, desde A1 hasta An, entonces:



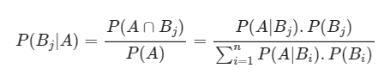
Esta fórmula se transforma en:



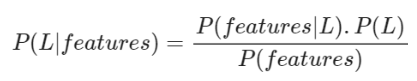
**Teorema de Bayes:** Si tenemos B1,…,Bn una partición de Ω, tal que P(Bi) ≠ 0, desde i = 1 hasta n. Si tenemos un evento A tal que P(A) ≠ 0; entonces para cada j = 1, 2 … n



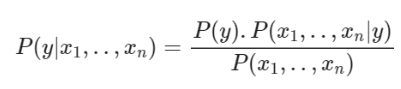
Por definición de probabilidad condicional, y usando el teorema de probabilidad total, tenemos para cada j desde 1 a n:



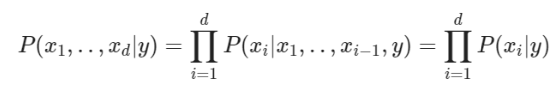
Entonces podemos interpretar que tenemos una **familia de clasificadores** simples, basados en el **Teorema de Bayes:** Si L son las categorías (labels) y features es una matriz de features:



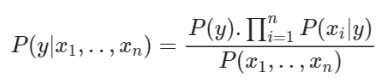
Como con Naive Bayes, suponemos independencia de las features entre sí, el Teorema de Bayes indica:



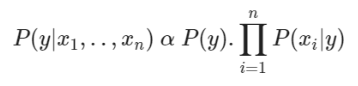
Por chain rule + independencia de probilidades,

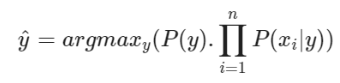


Entonces, reemplazando lo recién descripto en la fórmula del Teorema de Bayes, obtenemos:



El **denominador P(x1,…,xn) es constante** para todas las clases. Entonces **podemos realizar una predicción maximizando el numerador**.

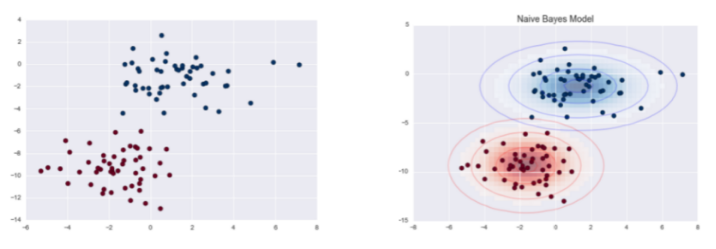


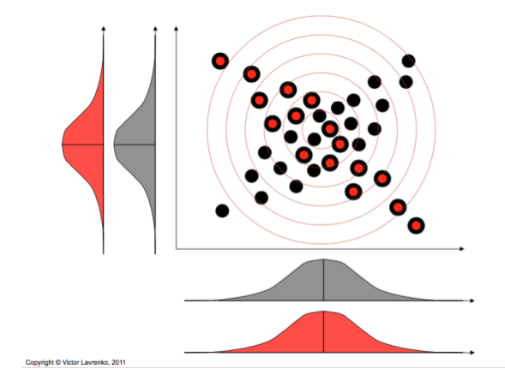


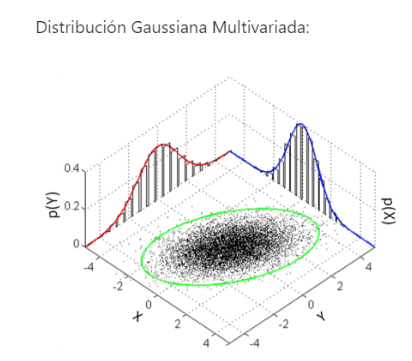
Hay diferentes agoritmos de Naive Bayes que difieren en la distribución que suponen para P(xi|y):

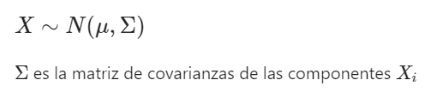
* **Gaussian Naive Bayes**: Supone una distribución Gaussiana multidimensional.
* **Multinomial Naive Bayes**: Supone una distribución Multinomial.

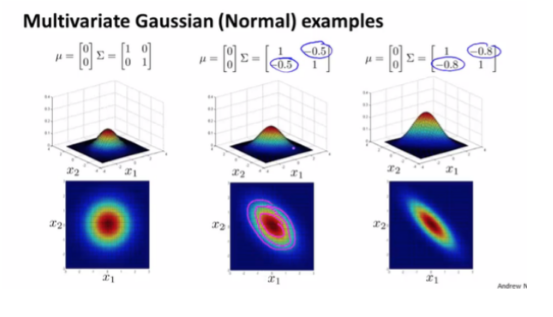
**Gaussian Naive Bayes:** Un modelo tipo Gaussian Naive Bayes toma datos y calcula una distribución para cada una de las clases. Entonces luego, cuando recibe un punto nuevo, calcula la probabilidad de que pertenezca a cada una de las clases. **Supone** que los **datos** se extraen de una **distribución gaussiana multivariada** en la que las **features** son **independientes entre sí.**

****

****

****

****

****

**En Python**:

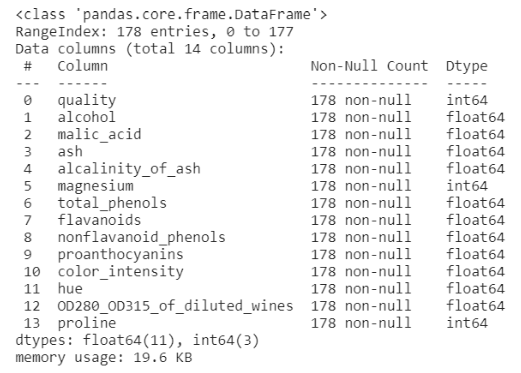
# Vamos a ver un ejemplo de clasificación de vinos en 3 categorías distintas:

data\_columns = [‘quality’, ‘alcohol’, ‘malic\_acid’, ‘ash’, ‘alcalinity\_of\_ash’, ‘magnesium’, ‘total\_phenols’, ‘flavanoids’, ‘nonflavanoids\_phenols’, ‘proanthocyanins’, ‘color\_intensity’, ‘hue’, ‘0D280\_0D315\_of\_diluted\_wines’, ‘proline’]

data = pd.read\_csv(“../Data/wine.data”, header = None)

data.columns = data\_columns

data.info()



data.quality.unique()



# Creamos los conjuntos de entrenamiento y testeo:

X = data.drop([‘quality’], axis = 1)

Y = data[‘quality’]

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size = 0.3, random\_state = 1237)

print(X\_train.shape)

print(X\_test.shape)

print(Y\_train.shape)

print(Y\_test.shape)



# Instanciamos el clasificador Gaussian Naive Bayes y lo entrenamos con los datos de train:

from sklearn import GaussianNB

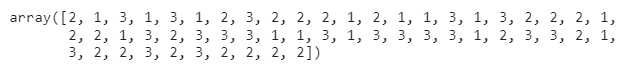
gnb = GaussianNB()

gnb.fit(X\_train, Y\_train)

# Ahora predecimos los valores de quality con los datos de test:

Y\_pred = gnb.predict(X\_test)

Y\_pred



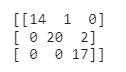
# Evaluamos el accuracy contra los datos reales de testeo y la matriz de confusión:

round(accuracy\_score(Y\_test, Y\_pred), 2)



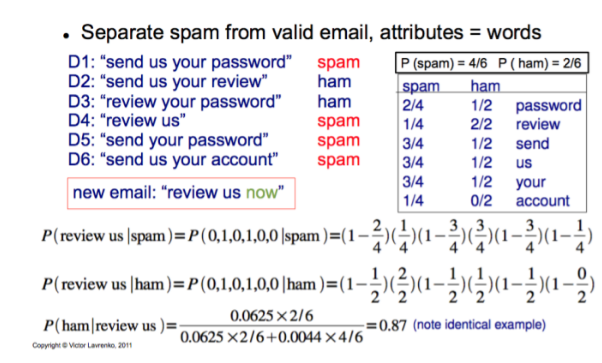
conf\_mat = confusion\_matrix(Y\_test, Y\_pred)

print(‘Confusion Matrix/n/n’, conf\_mat)



**Multinomial Naive Bayes**: Este clasificador **supone** que las **features** tienen **distribución multinomial simple.** La distribución multinomial describe la **probabilidad de observar recuentos entre varias categorías**. Es más apropiado **Naive Bayes multinomial** para **features que representan recuentos o tasas de recuento**. La idea es la misma que con Gaussian Naive Bayes, pero modelando la distribución de datos **con una distribución multinomial de mejor ajuste**.

Ejemplo: Puede aplicarse para detectar Spam:



**En Python**:

Vamos a entrenar un modelo de clasificación **Multinomial Naive Bayes** para predecir si un texto corresponde o no a un mail de spam. Partimos de un dataset original con dos columnas: el texto del mail y la etiqueta que queremos predecir (spam o ham). Vamos a usar un dataset que es una transformación del original, con 8682 columnas, cada una es una palabra que apareció en el campo texto del mail, en alguno de los registros del dataset original. El valor en el dataframe de cada elemento (i, j) es la cantidad de veces que aparece la palabra j en el mensaje i.

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, confusion\_matrix

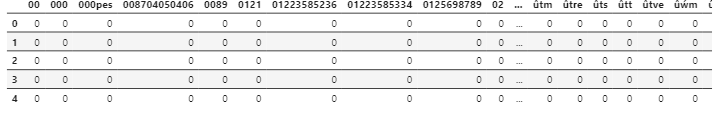
from sklearn.naive\_bayes import MultinomialNB

data = pd.read\_csv(‘..Data/spam\_count\_vectorizer.zip’, sep = ‘\t’)

data.shape



data.head(5)

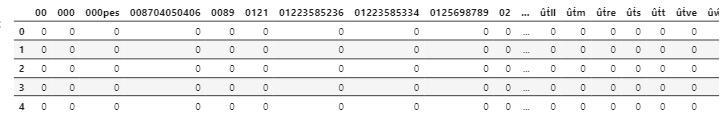


# Creamos la matriz Feature y el vector target:

y = data[‘target’]

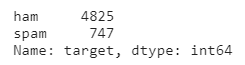
X = data.iloc[:, range(data.shape[1]-1)] # la columna target estaba al final del dataframe.

X.head(5)



# Podemos ver la proporción de las etiquetas en el dataframe:

y.value\_counts()



# Ahora creamos los conjuntos de entrenamiento y testeo:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify = y, random\_state = 12)

# Instanciamos el modelo y lo entrenamos:

model = MultinomialNB()

model.fit(X\_train, y\_train)

# Evaluamos el accuracy en test:

model.score(X\_test, y\_test)



# Calculamos la Matriz de Confusión:

y\_hat = model.predict(X\_test)



confusion\_matrix(y\_test, y\_hat)



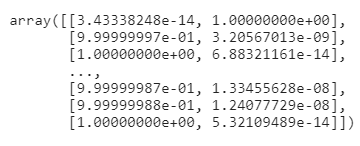
# Si queremos saber qué columna corresponde a cada categoría, usamos **classes\_**

model.classes\_



# Con **predict\_proba** podemos ver qué probabilidad tiene cada registro en cada clase:

model.predict\_proba(X\_test)



# Chequeamos que la suma de cada fila da 1 (probabilidad de ham + probabilidad de spam = 1). Como esta suma no siempre da 1, pero por cuestiones de redondeo, entonces la verificación la hacemos dando una tolerancia de +/- 0.00001, y ahí el resultado nos da bien:

any(model.predict\_proba(X\_test).sum(axis=1) != 1)

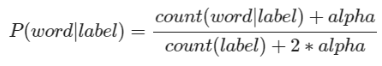


any(abs(model.predict\_proba(X\_test).sum(axis=1)-1) > 1e-5)



**Problemas que se pueden dar con casos discrtos:**

* **Problema de Frecuencia cero:** Ejemplo en el mail: cualquier mail que contenga la palabra ‘account’ va a ir directo a spam porque P(account|ham)=0.2; la solución a este problema es añadir un pequeño valor positivo al conteo (**Laplace Smoothing**)



* **Asume independencia de las Features:** Se pueden agregar muchas palabras asociadas con mail “no spam” a un mail “spam”, y de esta forma se busca engañar al clasificador. Como asume independencia entre features, las palabras “spam” tienen el mismo peso que las palabras “no spam”.