Algoritmo de Clasificación Naive Bayes implementado en PySpark

Luis Araujo-Lechuga, Luis Flores-Aquino, Manuel Velarde-Flores, Mileydy Ninantay-Diaz, Milagros Yarahuaman-Rojas, Gonzalo Gutierrez-Daza

Facultad de Ingeniería Eléctrica, Electrónica, Informática y Mecánica

Universidad Nacional San Antonio Abad del Cusco

Cusco, Perú

RESUMEN

El algoritmo Naive Bayes es bastante estudiado en la actualidad dada su utilidad para resolver problemas de clasificación, pero en proyectos de Big Data una implementación estándar es muy poco eficiente, asimismo, las librerías conocidas dentro de lenguajes de programación como Python no son óptimas para estos proyectos. Por este motivo se nota la necesidad de realizar una implementación de este algoritmo dentro de un entorno para Big Data, como es Spark, para optimizar este procesamiento. En este trabajo realizamos una implementación basándonos en el uso de la estructura RDD, para poder optimizar el algoritmo de Naive Bayes para datos tanto categóricos como numéricos. Los resultados obtenidos son útiles para poder realizar una discusión y brindar una opinión consistente, la cual se podrá observar en la sección pertinente.

Palabras Clave: Naive Bayes, Spark, Big Data, Clasificador

1. ALGORITMO NAIVE BAYES

1.1 DESCRIPCIÓN

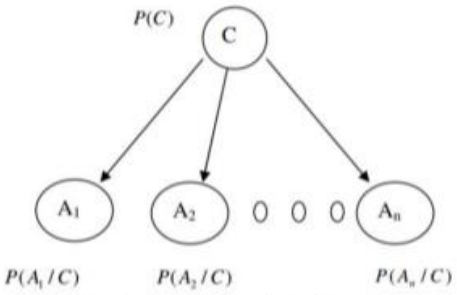
En teoría de la probabilidad y minería de datos, el clasificador Naive Bayes es un clasificador probabilístico fundamentado en el teorema de Bayes y algunas hipótesis simplificadoras adicionales. Es por estas simplificaciones, que recibe el nombre de naive, o sea, ingenuo.

1.2 CARACTERÍSTICAS PRINCIPALES

- La estructura de una red bayesiana puede utilizarse para construir clasificadores, utilizando el nodo raíz para la variable de salida o las clases
- El proceso de clasificación se realiza utilizando el teorema de Bayes, para asignar a un objeto u observación la clase con mayor probabilidad.
- En el problema de clasificación, se tiene la variable de salida o de clase (C), y un conjunto de variables predictoras o atributos {A_1, A_2, …, An}. Para estas variables se obtiene una red bayesiana B, que define una distribución de probabilidad conjunta P (C, A_1, A_2, …, An).
- Por consiguiente, se puede utilizar la red bayesiana resultante para los valores de un conjunto de atributos {a_1, a_2, …, an}, el clasificador basado en B retorna la etiqueta c que maximice la probabilidad a posteriori P (C = c | a_1, a_2, …, an), la clase c toma "m" posibles valores c_1, c_2, …, cm.

$$P(C = c / a_1, a_2, ..., a_n) = \frac{P(C = c)\prod_{i=1}^{n} P(a_i / C = c)}{P(a_1, a_2, ..., a_n)}$$

Ilustración 1. Fórmula matemática



Red de clasificador Naive Bayes

P(C) = P(C=c1) + P(C=c2) + ... + P(C=cn)

Ilustración 2. Modelo visual

1.3 PROPIEDADES PARA LA IMPLEMENTACIÓN

- Para estimar la probabilidad de la clase o variable de salida, y probabilidad condicionada, se aplica frecuencia relativa. En conjuntos de datos reales puede ocurrir que, no se tiene registros para algunas clases, ello implica que aparecen valores iguales a ceros, es típico, para corregir este inconveniente hacer reajuste.
- En el caso de variables continuas, no será posible determinar las frecuencias, entonces podemos emplear algunas soluciones:
    - Descretización de variables continuas
    - Asumir que la variable tiene una distribución. Una de las expresiones a utilizar suele ser: asumir que la distribución es normal, con media μ, y desviación típica σ. El inconveniente será si la distribución de los valores no es normal:

$$P(A_i / c) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(X-\mu)^2}{2\sigma^2}\right)$$

Ilustración 3. Distribución normal

2. SPARK

2.1 DESCRIPCIÓN

Spark es un framework de computación en clúster open-source. Fue desarrollado originariamente en la Universidad de California. El código base del proyecto Spark fue donado más tarde a la Apache Software Foundation que se encarga de su mantenimiento desde entonces. Spark proporciona una interfaz para la programación de clusters completos con Paralelismo de Datos implícito y tolerancia a fallos.

Apache Spark se puede considerar un sistema de computación en clúster de propósito general y orientado a la velocidad. Proporciona APIs en Java, Scala, Python y R. También proporciona un motor optimizado que soporta la ejecución de graficos en general.

2.2 PYSPARK

PySpark es una interfaz para Apache Spark en Python. No solo le permite escribir aplicaciones Spark utilizando las API de Python, sino que también proporciona el shell de PySpark para analizar de forma interactiva sus datos en un entorno distribuido. PySpark es compatible con la mayoría de las funciones de Spark, como Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) y Spark Core.

3. IMPLEMENTACIÓN DEL ALGORITMO

3.1 PREPARACIÓN DE DATOS Y ANÁLISIS DEL CONJUNTO DE DATOS

- Antes que realizar cualquier tipo de acción, configuramos nuestro entorno Spark.
- Nuestro conjunto de datos está conformado por información correspondiente a usuarios de una entidad bancaria portuguesa, que tiene como objetivo verificar qué clientes se suscribirán o no el servicio ofrecido.
- Este conjunto de datos fue obtenido del repositorio Kaggle y consiste de 17 columnas ('age', 'job', 'marital', 'education', 'default', 'balance', 'housing', 'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays', 'previous', 'poutcome', 'Target') y 45211 filas.
- Separamos las columnas en categóricas y numéricas para poder tratarlas de manera correcta.
- Tras realizar el análisis pertinente, se tomó la decisión de eliminar dos columnas: "default" y "loan", debido a que no presentan diferencias útiles para nuestro procesamiento, esto se puede apreciar en las siguientes imágenes:

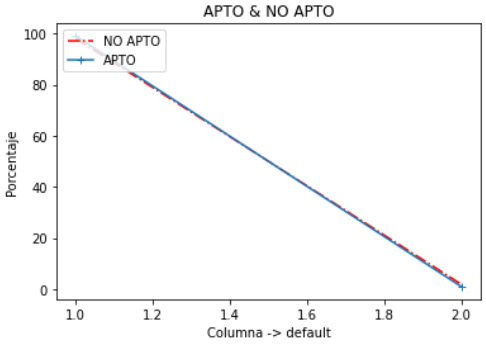


Ilustración 4. Comparativa para determinar relevancia de columna default

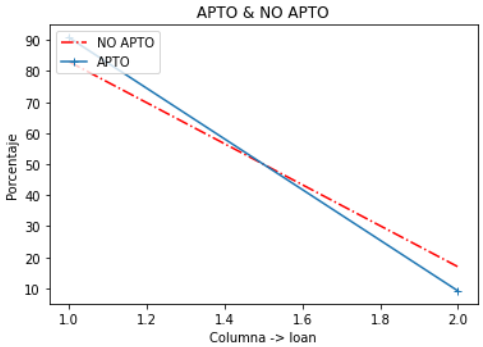


Ilustración 5.Comparativa para determinar relevancia de columna loan

- Luego, realizamos una normalización de valores numéricos para poder manejar nuestros datos de manera correcta para procesarlos de forma idónea.
- Finalmente, dividimos nuestros datos en los conjuntos de entrenamiento y prueba, obteniendo 31630 filas para entrenamiento y 13581 filas para pruebas.

3.2 IMPLEMENTACIÓN DE NAIVE BAYES

- Debido a que contamos con dos tipos de datos (numéricos y categóricos) requerimos dos funciones con las cuales poder tratarlos de manera apropiada, estas funciones fueron denominadas ResColNum y ResColCategorica, las cuale se pueden ver a continuación

```
def ResColNum(dataRDD,NumCol,NumColKey):
    intento01 = dataRDD.groupBy(lambda x: x[NumColKey]) #Agrupamos los datos segun la variable de salida
    intento02 = intento01.map(lambda x: (x[0],[y[NumCol] for y in x[1]])) #Guardar solo la columna y la salida
    intento03 = intento02.map(lambda x: (x[0],CalcMeanSTD(x[1]))) #Calculamos el mean y el std
    return intento03.collect() # Retornamos los valores encontrado
```

Ilustración 6. Función ResColNum

```
def ResColCategorica(dataRDD,NumCol,NumColKey):
    intento01 = dataRDD.groupBy(lambda x: x[NumCol]) # Agrupar mediante columna 0
    intento02 = intento01.map(lambda x: (x[0],[y[NumColKey] for y in x[1]])) #Guardar solo el tipo y la salida
    intento03 = intento02.map(lambda x: (x[0],collections.Counter(x[1]))) #Contar apariciones y total
    #intento04 = intento03.map(lambda x: (x[0], ConvertirPorcentaje(x[1]))) #Convertir a porcentaje y convertir a diccionario
    intento04 = intento03.map(lambda x: (x[0], ConvertirPorcentaje(x[1]))) #Convertir a porcentaje y convertir a diccionario
    return intento04.collect()
```

Ilustración 7. Función ResColCategorica

- Asimismo, estas funciones mencionadas anteriormente dependen de otras dos funciones acorde al tipo de dato que se tratará, en este caso tendremos la función CalcMeanSTD para los datos numéricos y la función ConvertirPorcentaje para los datos categóricos. Estas funciones se pueden apreciar a continuación

```
def CalcMeanSTD(Valores):
    meanVals = np.mean(Valores)
    stdDev = np.std(Valores,ddof=1)
    return {'mean':meanVals,'std':stdDev}
```

Ilustración 8. Función CalcMeanSTD

```
def ConvertirPorcentaje(CounterApar):
    dictPorcentaje = {}
    for key in Salida.keys():
        dictPorcentaje[key] = CounterApar[key]/Salida[key]
    return dictPorcentaje
```

Ilustración 9. Función ConvertirPorcentaje

- Posteriormente vemos la cantidad de datos de salida con los que contamos, en este caso nuestra variable objetivo posee una salida binaria (“no” y “yes”). Así pues, contamos con 27890 filas con salida “no” y 3740 filas con salida “yes”.
- Procedemos a aplicar Naive Bayes a nuestros datos de entrenamiento e imprimimos nuestros resultados parciales. Los cuales se pueden apreciar a continuación, aunque no se puede mostrar todo dado el tamaño de la salida, se puede notar los resultados para los datos categóricos y numéricos.

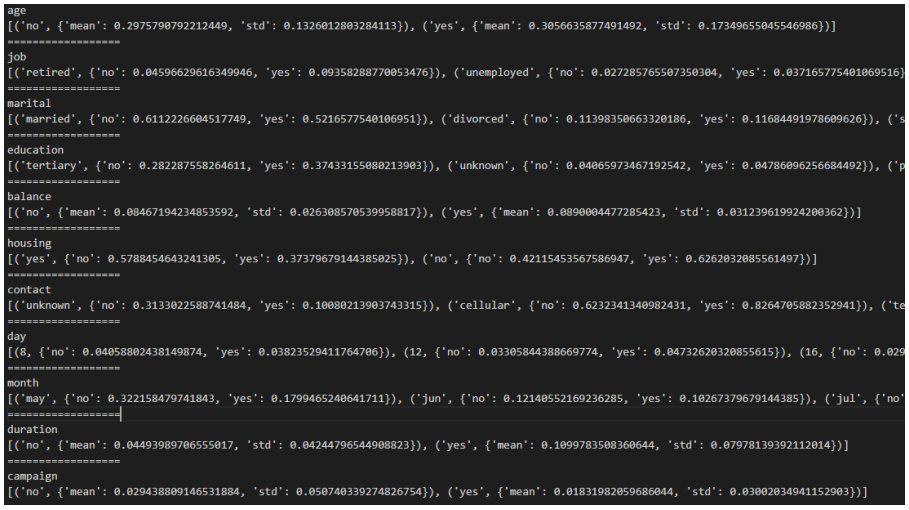


Ilustración 10. Resultados de entrenamiento

- Finalmente evaluamos el modelo con los datos de prueba.
- Aplicamos nuestro clasificador a los elementos del RDD, para esto tenemos una función denominada Evaluar, la cual se puede apreciar a continuación

```
def Evaluar(TuplaEvaluar):
    ProbabilidadesTotales = {}
    for Caso in Salida_Keys:
        ProbTot = Salida[Caso]/CantData
        for k in range(len(TuplaEvaluar)):
            if(TipoCols[k] == 'cat'):
                dataTemp = ResColumnas[columnas[k]]
                Encontrado = False
                i = 0
                P = 0
                while(not Encontrado):
                    if (dataTemp[i][0]==TuplaEvaluar[k]):
                        P = dataTemp[i][1][Caso]
                        Encontrado=True
                    i+=1
                    if(i>=len(dataTemp)):
                        break
            else:
                dataTemp = ResColumnas[columnas[k]]
                Encontrado = False
                i = 0
                P = 0
                while(not Encontrado):
                    if (dataTemp[i][0]==Caso):
                        mean = dataTemp[i][1]['mean']
                        std = dataTemp[i][1]['std']
                        P = 1/(np.sqrt(2*np.pi)*std)*np.e**(-(TuplaEvaluar[k]-
mean)**2/(2*std**2))
                        Encontrado=True
```

```
                    i+=1
                    if(i>=len(dataTemp)):
                        break
            ProbTot = ProbTot*P
        ProbabilidadesTotales[Caso]=ProbTot
        max_key = max(ProbabilidadesTotales, key=ProbabilidadesTotales.get)
    return(max_key)
```

- Esta función puede resultar confusa ya que aparentemente realiza uso de un bucle for para analizar los elementos del RDD, pero no, esta función se aplica sobre cada elemento de manera individual mediante la función map. De este modo, se confirma que nuestro algoritmo puede ser aplicable dentro de un cluster para un entorno de Big Data.
- Asimismo, contamos con una función adicional, denominada ContarAciertosErrores, con la cual podemos verificar nuestros resultados finales, esta función está dada de la siguiente manera

```
def ContarAciertosErrores(tupla): #ACIERTOS, FALLOS
    RptReal = tupla[0]
    Acertadas = len([RptCalculada for RptCalculada in tupla[1] if RptCalculada==RptReal])
    Fallidas = len(tupla[1])-Acertadas
    return((Acertadas,Fallidas))
```

Ilustración 11. Función ContarAciertosErrores

- Como se mencionó anteriormente, se hace uso de la función map para aplicar las funciones dadas, en el RDD, lo que se puede apreciar a continuación

```
TestEvalRDD = TestRDD.map(lambda x: (x[-1], Evaluar(x[0:-1]))) #Rpt real, Rpt calculada
TestGroupRDD = TestEvalRDD.groupBy(lambda x: x[0]).map(lambda x: (x[0],[a[1] for a in x[1]]))
TestMatrixConfRDD = TestGroupRDD.map(lambda x: (x[0],ContarAciertosErrores(x)))
DataMatrix = TestMatrixConfRDD.collect()
```

Ilustración 12. Aplicación final de nuestro algoritmo

- Con estos datos finales obtenidos podemos mostrar nuestros resultados en la siguiente sección.

4. RESULTADOS, CONCLUSIONES Y DISCUSIONES

- Para demostrar nuestros resultados decidimos hacer uso de una matriz de confusión simple, la cual se puede ver a continuación

```
print('  MATRIZ DE CONFUSIÓN')
print(f'  |  V | F  ')
print(f'V |{DataMatrix[1][1][0]} | {DataMatrix[1][1][1]}')
print(f'F |{DataMatrix[0][1][1]} | {DataMatrix[0][1][0]}')
```

```
 MATRIZ DE CONFUSIÓN
  |  V | F
V |806 | 743
F |904 | 11128
```

Ilustración 13. Matriz de confusión resultante

- Como se puede apreciar en los resultados, la mayor cantidad de datos clasificados son verdaderos negativos, o sea, resultados "no" correctamente clasificados, que son 11128 y en comparativa a estos se tienen muy pocos falsos negativos, los cuales son solo 904.

- Por otro lado, en el lado de los resultados "yes" la cantidad de positivos y negativos es muy similar, con lo cual podemos llegar a la conclusión de que nuestros datos en cierto punto son insuficientes para poder determinar de manera correcta esta salida, ya que la mayoría de nuestros datos tienen como salida "no", lo cual desequilibra nuestro proceso desde el entrenamiento.

5. BIBLIOGRAFÍA

- https://www.kaggle.com/krantiswalke/bankfullcsv
- https://medium.com/datos-y-ciencia/algoritmos-naive-bayes-fudamentos-e-implementación-4bcb24b307f
- Rish, Irina. (2001). An Empirical Study of the Naïve Bayes Classifier. IJCAI 2001 Work Empir Methods Artif Intell. 3.
- https://spark.apache.org
- https://spark.apache.org/docs/latest/api/python/