FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

[9558] Organización de Datos
Primer cuatrimestre de 2018
Grupo 24

Nombre	Padrón	E-mail
BUSTAMANTE, Jorge Tomás	89603	jbustamante@fi.uba.ar
ROCA MARTÍNEZ, Zibel Carolina	97944	z.carolina.roca@gmail.com

Repositorio: https://github.com/tomasBustamante/postulaciones

Índice

1. Introducción

El objetivo del presente trabajo práctico es el de la aplicación de algoritmos de *machine learning* a un conjunto de datos sobre postulantes, avisos laborales y postulaciones efectuadas para predecir la probabilidad de que cierto postulante se postule a determinado aviso.

El desarrollo está dividido en dos grandes partes. Por un lado la de limpieza y depuración de los datos que consiste en analizar cada una de las columnas, filtrarlas y darles el formato ideal y, por otro lado, la aplicación del algoritmo de clasificación que será entrenado en base a los datos pulidos y se encargará de generar las predicciones deseadas.

2. Limpieza de datos

Además de los conjuntos de datos provistos para el primer trabajo práctico, contamos con otros iguales pero en otras fechas (hasta el 17 de abril). Antes de comenzar a procesarlos los leemos todos, los unimos (con *pd.concat*) y finalmente elminamos los duplicados.

2.1. Datos sobre los postulantes

2.1.1. Sexo y edad de los postulantes

Tenemos dos *data frames* acerca de los postulantes: uno con información acerca de su sexo y fecha de nacimiento y el otro acerca de su nivel de educación.

El *data frame* **postulantes_sexo_y_edad** tiene tres columnas:

- **idpostulante**: cadena de caracteres que representa el código de identificación del postulante
- **fechanacimiento**: año, mes y día de nacimiento del postulante separado por guiones.
- **sexo**: atributo categórico que puede tomar tres valores posibles: *FEM*, *MASC* y *NO_DECLARA*

La columna de la fecha de nacimiento la renombramos a edad y calculamos la edad de cada postulante mediante la biblioteca *datetime* de Python.

Para la columna de sexo tenemos tres valores categóricos y casi el 50 % es masculino, casi el 50 % es femenino y únicamente un porcentaje muy bajo de los postulantes no declararon su sexo. Considerando esto tenemos diversas maneras de procesar esta columna. La más sencilla es la de asignarle un número distinto (0, 1 y 2) a cada uno de los valores categóricos (*label encoding*) pero traería como desventaja que se le estaría dando el mismo peso a una categoría no muy representativa. Una mejora sería adjudicándole a todos aquellos que no hayan declarado sexo alguno entre femenino y masculino (ya sea al azar o arbitrariamente). Finalmente otra opción sería aplicar *one hot encoding*, es decir, crear una columna *femenino* y otra *masculino* y que tengan ceros y unos según corresponda, de manera tal que aquellos que no declaran tendrán ceros en ambas columnas

2.1.2. Nivel de educación de los postulantes

El *data frame* **postulantes_educacion** tiene tres columnas:

- **idpostulante**: cadena de caracteres que representa el código de identificación del postulante.
- **nombre**: atributo categórico que puede tomar los siguientes valores: *Secundario*, *Terciario/Técnico*, *Universitario*, *Posgrado*, *Master*, *Doctorado* y *Otro*.

- **estado**: atributo categórico que puede tomar tres valores posibles: *En Curso*, *Graduado* y *Abandonado*

Para categorizar los distintos niveles de estudios y sus estados probamos dos enfoques distintos:

1. Label Encoding para el máximo nivel de estudios alcanzado

   En este caso creamos una columna nueva (*nivel_alcanzado*) que tendrá números del 0 al 4 jerarquizando los distintos niveles de estudios a los que llegó el postulante.

```
def categorizar_estudios(x):
    if((x is np.nan) | (x == 'Otro')):
        return 0
    if(x == 'Secundario'):
        return 1
    if(x == 'Terciario/Técnico'):
        return 2
    if(x == 'Universitario'):
        return 3
    if((x == 'Posgrado') | (x == 'Master') | (x == 'Doctorado')):
        return 4

postulantes_educacion['nivel_alcanzado'] =
    postulantes_educacion['nombre']\
    .apply(lambda x: categorizar_estudios(x))
postulantes_educacion.head()
```

	idpostulante	nombre	estado	nivel_alcanzado
0	NdJl	Posgrado	En Curso	4
1	8BkL	Universitario	En Curso	3
2	1d2B	Universitario	En Curso	3
3	NPBx	Universitario	En Curso	3

Nivel de educación con Label Encoding

   Luego ordenamos en orden descendiente por esta nueva columna, agrupamos por idpostulante y elminamos los duplicados, quedándonos así con los registros con el mayor nivel de estudios alcanzado. En este ejemplo no estamos teniendo en cuenta el estado (por eso se llama "nivel **alcanzado**") pero en otras pruebas le añadimos unos centécimos más en la función de categorización. Es decir, si el estado es "Graduado", entonces le sumamos 1 al valor de la nueva columna o 0,5 si el estado es "En Curso". Sin embargo esta complejización no logró generar mejores resultados en la predicción posterior.

2. One Hot Encoding para los distintos niveles completados

   Una variante más precisa es generando nuevas columnas para cada uno de los valores categóritos del nivel de estudios posibles del postulante (exceptuando a "Otros" y agrupando a "Posgrado", "Master" y "Doctorado" en simplemente "posgrado").

   Como varios postulantes declararon diversos estudios, establecemos con el valor 1 en todas las columnas que coincida, reduciendo así una única fila por cada idpostulante. Además dejamos en 1 únicamente a aquellos que figuren con el estado "Graduado".

Finalmente agregamos un 1 en la columna de secundario en todos aquellos registros que tengan estudios superiores.

	idpostulante	secundario	terciario	universitario	posgrado
0	0z5Dmrd	1	0	1	0
1	0z5JW1r	1	1	0	0
2	0z5VvGv	1	0	0	0

Nivel de educación con One Hot Encoding

Unimos todos los datos sobre los postulantes en un único *dataframe* para ser tratado posteriormente junto con los avisos.

2.2. Datos sobre los avisos

Tenemos dos *data frames* acerca de los avisos: uno con un listado de los que se encuentraban publicados online el 8 de marzo de 2018 y otro con la información detallada de cada aviso.

El *data frame* **avisos_online** tiene un total de 5028 registros con una sola columna:

- **idaviso**: cadena de caracteres que representa el código de identificación del aviso.

Para procesar esta información agregamos una columna binaria que determine si el aviso estuvo o no online. Debido a lo poco que aporta esta columna y a los malos resultados obtenidos en las predicciones, directamente no consideramos este factor en la mayoría de las pruebas.

El *data frame* **avisos_detalle** tiene un total de once columnas:

- **idaviso**: cadena de caracteres que representa el código de identificación del aviso.
- **idpais**: número entero que representa el código de identificación del país del aviso.
- **titulo**: cadena de caracteres con el nombre del puesto de trabajo.
- **descripcion**: texto de varias líneas con la descripción del aviso en formato HTML.
- **nombre_zona**: cadena de caracteres que puede tener los siguientes valores: *Gran Buenos Aires*, *Capital Federal*, *GBA Oeste* y *Buenos Aires (fuera de GBA)*
- **ciudad**: cadena de caracteres con el nombre de la ciudad del puesto de trabajo.
- **mapacalle**: cadena de caracteres con la dirección del lugar de trabajo.
- **tipo_de_trabajo**: atributo categórico que puede adoptar los siguientes valores: *Full-time*, *Part-time*, *Teletrabajo*, Pasantía, Por Horas, Temporario, Por Contrato, Fines de Semana* y *Primer empleo*.
- **nivel_laboral**: atributo categórico que puede adoptar los siguientes valores: *Senior / Semi-Senior*, *Junior*, *Jefe / Supervisor / Responsable*, *Gerencia / Alta Gerencia / Dirección* y *Otro*.
- **nombre_area**: cadena de caracteres que representa el área de trabajo, como por ejemplo: *Ventas*, *Contabilidad* o *Recursos Humanos*.
- **denominacion_empresa**: nombre de la empresa que publicó el aviso.

La columna **idpais** contiene un 1 para todas las filas (representando al país Argentina), por lo que la descartamos desde el principio. Las columnas **ciudad** y **mapacalle** tienen información interesante acerca de la ubicación del puesto de trabajo ofrecido pero solamente unas pocos avisos nos proveen de ese detalle. Como no llega a ser una cantidad significativa eliminamos esas columnas también para que no generen ruido.

2.2.1. Categorización del nivel laboral

Ánalogamente al nivel de estudios alcanzado por los postulantes, tenemos dos opciones para categorizar los niveles laborales de los avisos: jerarquizándolos con *Label Encoding* en una sola columna o *One Hot Encoding* para cada uno de los valores categóricos.

1. Label Encoding para el nivel laboral

   Jerarquizamos con valores del 0 al 4 para cada una de las categorías posibles atribuyendo el número más alto a los cargos gerenciales y los más bajos a los de *junior*. Como las categorías de Jefe y de Gerente son muy similares (y no son tantos) en algunas pruebas las unificamos para que no se le de tanta relevancia a esa distancia que es mucho más sutil que las demás.

   ```
   def categorizar_nivel_laboral(x):
       if((x is np.nan) | (x == 'Otro')):
           return 0
       if(x == 'Junior'):
           return 1
       if(x == 'Senior / Semi-Senior'):
           return 2
       if(x == 'Jefe / Supervisor / Responsable'):
           return 3
       if(x == 'Gerencia / Alta Gerencia / Dirección'):
           return 4
   ```

   Dado que la gran mayoría de los avisos son para el nivel “Senior / Semi-Senior” es que también tiene sentido realizar una codificación binaria en donde se adjudicaría un 1 para esos casos y un 0 para todos los demás.

2. One Hot Encoding para los distintos niveles laborales

   Para este caso no consideramos los registros con el nivel “Otro” (ni los *NaN*s) de manera tal que figuran con 0 en las tres columnas.

	idaviso	junior	senior	jefe
0	8725750	0	1	0
2	1000150677	0	1	0
3	1000610287	0	1	0
4	1000872556	0	1	0

One Hot Encoding para nivel laboral

2.2.2. Categorización del tipo de trabajo

Dado que la mayoría de las categorías posibles para los tipos de trabajo son “Full-time” y “Part-time” lo más simple y efectivo era procesar esta columna de manera binaria para esos dos valores y adjudicándole algún valor al azar para los pocos otros casos.

Para probar la categorización mediante One Hot Encoding adoptamos tres columnas, agrupando en “Otro” todos aquellos tipos que no sean “Part-time” ni “Full-time”.

	idaviso	part-time	full-time	otro
0	8725750	0	1	0
2	1000150677	0	1	0
3	1000610287	0	1	0
4	1000872556	0	1	0

One Hot Encoding para el tipo de trabajo

2.2.3. Categorización del área de trabajo

Las áreas de trabajo son muy diversas y resulta efectivo hacer un simple Label Encoding para cada una de ellas. Sin embargo, realizando un análisis manual se puede apreciar que varias áreas se pueden agrupar entre sí, como "Recursos Humanos" con "Administración de Personal" como también las diversas áreas de programación y sistemas. Para ello implementamos una función que se fijara en cada uno de ellos y devolviera la nueva área generalizada.

2.2.4. Categorización de la zona

Esta columna no nos provee de mucha ayuda dado que la gran mayoría (casi el 100 %) de los avisos son del Gran Buenos Aires o de la Ciudad Autónoma de Buenos Aires. En algunos casos conviene directamente descartar la columna entera o a lo sumo aplicar un *Label Encoding* binario discriminando entre CABA y GBA.

2.2.5. Título y descripción de los avisos

La columna de la **descripción** es la que más información nos provee pero a su vez la más difícil de procesar dado que se trata de un texto largo y variable y con tags de HTML. Dado que se obtuvieron buenos resultados sin considerar esta columna es que no se realizó una depuración del texto pero una posibilidad habría sido removiendo todos los tags y *tokenizándolo*, es decir, separando cada palabra y quedándonos con las relevantes para finalmente aplicar *Label Encoding*.

La columna del **título** también es un texto variable pero mucho más corto y limpio de manera tal que se puede aplicar *Label Encoding* directamente. Ralizando una inspección manual se aprecian que algunos títulos podrían estar agrupados dado que son muy similares (por ejemplo "Desarrolladores" y "Developers"). Para ello se probó mediante una *tokenización* quedándonos con la primera palabra del título a la cual se le aplica un *stemming* o *lematización*, es decir, quedarnos con la raíz de la palabra.

2.3. Datos sobre las transacciones

Tenemos dos *data frames* con información de dos hechos a partir de las dimensiones anteriores. El *data frame* vistas contiene el detalle de todas las veces que los postulantes visualizaron los diversos avisos y el *data frame* postulaciones registra quiénes se postularon a qué avisos y cuándo.

El *data frame* **vistas** tiene tres columnas:

- **idAviso**: cadena de caracteres que representa el código de identificación del aviso visto.
- **timestamp**: año, mes día y hora exacta en la que se produjo la vista del aviso.
- **idpostulante**: cadena de caracteres que representa el código de identificación del postulante que vio el aviso.

En la mayoría de los casos probados no fue tenida en cuenta la información de las vistas dado que solamente revelan aquellas que fueron producidas por usuarios registrados en la aplicación y que hayan iniciado sesión, de manera tal que incluirla puede generar ruido al no considerar todas las vistas de usuarios anónimos.

El *data frame* **postulaciones** tiene tres columnas:

- **idaviso**: cadena de caracteres que representa el código de identificación del aviso al cual se realizó la postulación.
- **idpostulante**: cadena de caracteres que representa el código de identificación del postulante.
- **fechapostulacion**: año, mes día y hora exacta en la que se produjo la postulación.

Para confeccionar el conjunto de datos de entrenamiento unimos el *data frame* de las postulaciones con el de los avisos y el de los postulantes procesados previamente y le agregamos la columna **sepostulo** con el valor 1 para todas sus filas. Luego completamos el *data frame* con varias filas más con "no postulaciones", es decir, combinaciones aleatorias entre postulantes y avisos que no hayan estado incluidas entre las postulaciones. Estas nuevas filas tendrán el valor 0 para la nueva columna. La columna de la fecha de postulación la descartamos inmediatamente dado que no aporta información relevante para predecir una postulación futura dado que no tendría sentido que existan horas para las "no postulaciones".

3. Algoritmos de predicción

3.1. KNN

El primer algoritmo probado fue el de *K-Nearest Neighbours* cuya velocidad de procesamiento es muy alta. Recibe como parámetro la cantidad de vecinos (k) a ser considerados. Comenzamos probando con 7 vecinos y observamos *underfitting* por lo que fuimos reduciéndolo hasta lograr el mejor resultado con 4 vecinos.En las pruebas con valores inferiores a 4 se podía apreciar un claro ejemplo de overfitting, dado que se ajustaba demasiado a los vecinos más cercanos sin tener en cuenta otros factores que eran importantes.

3.2. Perceptrón

Recibe dos parámetros

- "n_iter": cantidad de iteraciones por todo el conjunto de datos.
- "eta0": es la tasa de aprendizaje que se encarga de controlar la actualización del peso luego de cada iteración y solo se da si la salida difiere de la salida deseada.

3.3. Naive Bayes

Este algoritmo está basado en la aplicación del teorema de Bayes. Utilizamos GaussianNB de sklearn que no recibe parámetros, obteniendo resultados ligeramente mejores que con KNN.

3.4. Gradient Boost

Este es el algoritmo que mejores resultados obtuvo y se dio en situaciones en las que los datos estaban formateados con*Label Encoding*. Utiliza el método de *boosting* que consiste en la construcción del modelo de manera escalonada ponderando los resultados en cada paso. Recibe dos parámetros:

- Tasa de aprendizaje (*"learning_rate"*): Determina el impacto de cada árbol en el resultado final. Cuanto más bajo es su valor el algoritmo requiere de una cantidad mayor de árboles aumentando así su complejidad. Para valores inferiores a 0,001 no se apreciaron mejoras sustanciales.
- Número de estimador (*"n_estimators"*): Determina la cantidad de árboles a ser modelados. Los mejores resultados se obtuvieron con un valor de 200.

3.5. Random Forest

Este algoritmo es un meta estimador que se adapta a una serie de clasificadores de árboles de decisión en varias submuestras del conjunto de datos y utiliza un promedio para mejorar la precisión predictiva y controlar el ajuste excesivo. El tamaño de submuestra siempre es el mismo que el tamaño de muestra de entrada original, pero las muestras son reemplazadas si el parámetro *bootstrap* es verdadero (valor por defecto).

3.6. Extra Trees

Utilizamos la clase ExtraTreesClassifier de *sklearn.ensemble* que no recibe parámetros. Funciona de manera similar a Gradient Boost pero utiliza promedios para mejorar la precisión de la predicción y controlar el *overfitting*. Los resultados obtenidos se acercaron a los de Gradient Boost pero sin lograr superarlo.