FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Organización de datos - 75.06/95.58

Trabajo Práctico 2

Machine Learning

1º cuatrimestre 2018

Grupo 8

Nombre	**Padrón**
Rodrigo Etchegaray Campisi	96856
Maximiliano Pagani	94754
Fonzalida Miguel Angel	86125

1. Introducción general

Este trabajo consistió en una competencia de machine learning, en la cual debía predecirse si determinado postulante se iba a postular a un determinado aviso.

A nuestro criterio, el objetivo fundamental en este tipo de empresas que ofrecen búsqueda de empleos online (como zonajobs.com), es acercar a la empresa ideal con el empleado ideal, en el menor tiempo posible. Para este fin entendemos que sirven las predicciones de postulaciones mencionadas anteriormente: para ofrecerle a la persona dichos anuncios (de mucha probabilidad de postulación) mediante una recomendación, antes de que se postule a los mismos por cuenta propia. De esta forma, lograríamos acortar el tiempo de conexión entre empresa ideal y empleado ideal.

Desde nuestro punto de vista, entendemos que para realizar estas predicciones hay, fundamentalmente, dos vías de análisis:

- La primera es mediante el estudio de los antecedentes de comportamiento de un usuario. Este estudio se focaliza más en el análisis individual de cada postulante, cosa que para el objetivo del TP no nos sirve, además de que contamos con datos insuficientes para tal fin.
- La segunda vía sería el análisis a nivel macro de las tendencias generales de distintos conjuntos de usuarios o casos de estudios, agrupados por cierta característica. Luego, las recomendaciones a usuarios individuales estarían basadas en las tendencias de comportamiento dichos conjuntos.

Nosotros realizaremos nuestros análisis en base a la segunda vía. Trataremos de extraer conclusiones que nos ayuden a comprender las tendencias de comportamiento, para poder ofrecer mejores recomendaciones y en el momento más oportuno.

En base a el análisis de todos los datos de tiempos anteriores, y contando con la información sobre las características del postulante, y sobre las características de los avisos a los cuales postuló o no, utilizaremos algoritmos de machine learning para tratar de encontrar patrones, y relaciones entre las variables y el target (si se postuló o no), para ver cuales de ellas tienen mayor incidencia en el resultado, buscando predecir futuras postulaciones.

2. Datos disponibles - Preprocesamiento - Feature Engineering

2.0 Información general sobre los análisis realizados

2.0.1 Datos utilizados

Se analizaron los datos provistos por Navent sobre la búsqueda de empleos en el sitio https://www.zonajobs.com.ar/ . En este caso se contó con varios sets de información. Por un lado se tenía el set de datos utilizado para el trabajo práctico 1:

- **fiuba_1_postulantes_educacion.csv**:
  nivel educativo de los postulantes.
- **fiuba_2_postulantes_genero_y_edad.csv**:
  fecha de nacimiento y género de los postulantes.
- **fiuba_3_vistas.csv**:
  vistas de avisos online y offline, del 23 al 28 de Febrero de 2018.
- **fiuba_4_postulaciones.csv**:
  postulaciones del 15 de Enero al 28 de Febrero de 2018.
- **fiuba_5_avisos_online.csv**:
  avisos online al 8 de Marzo de 2018.
- **fiuba_6_avisos_detalle.csv**:
  detalle de avisos vistos y postulados tanto offline como online.

Por otro lado se tenía otro set con la misma estructura de 6 csv que los datos anteriores pero con información que llegaba hasta el 15 de Abril. Luego había información posterior al 15 de Abril pero en 4 csv faltando los de postulaciones y avisos online (la idea era predecir postulaciones de estos datos).

Por último se agregó un csv con detalles de avisos que en principio se habían perdido, llamado **fiuba_6_detalle_missing.csv** .

2.0.2 Repositorio de GitHub

Se utilizó un repositorio en github para la integración del trabajo. En dicho repositorio se pueden encontrar todos los notebooks con el código utilizado para realizar los análisis y predicciones.

Link: https://github.com/eche33/Datos-Tp2

2.1 Preparación básica de los set de datos

2.1.1 Introducción

La idea básica de este trabajo era probar distintos algoritmos de machine learning para predecir postulaciones. Para eso se necesitaba contar con **n** filas de **m** dimensiones y un label final que indicara si el postulante se postula o no a ese aviso. Para lograr eso, se empezó por acomodar los datos que se tenían.

2.1.2 Vistas

Lo primero que se hizo fue concatenar todos los archivos que trataban sobre vistas para así tener todo unificado. Luego de acomodar los datos para que fuera más cómodo procesarlos, como por ejemplo renombrar columnas o pasar datos a tipos más coherentes, se procedió a calcular la fecha en que se había producido la última vista de un usuario a un aviso. También se calculó la cantidad de veces que un usuario había visto determinado aviso.

2.1.3 Educación

Se concatenaron todos los archivos que tenían información acerca de la educación de los postulantes. En este caso en particular se podía tener varios estudios de un mismo postulante, en distintas instancias. Por ejemplo, un postulante podía tener el secundario completo y la universidad en curso.

Inicialmente se optó por sólo mantener un estudio de cada postulante. El criterio que se utilizó para decidir cual estudio mantener, fue ordenar primero por estado y luego por el nivel de estudio, de la siguiente manera:

estado (menor a mayor importancia) = [abandonado, en curso, graduado]
estudio (menor a mayor importancia) = [secundario, otro, terciario/técnico, universitario, postgrado, master, doctorado]

De esta forma, nos quedamos con el estudio graduado más alto, de no tener graduados, con el estudio en curso más alto, y así sucesivamente.

El problema es que estábamos perdiendo cierta información al representar en nuestro set unicamente un solo estudio particular. Entonces decidimos incorporar 2 features en total, el nivel de estudio más alto graduado, y el nivel de estudio más alto en curso. Para calcular numéricamente el nivel de estudio, nos basamos en el orden de importancia explicado arriba. De esta forma, si ambas features fuesen 0 por ejemplo, determinaria que no tiene ni estudios graduados ni en curso.

2.1.4 Detalles de avisos

Se concatenaron todos los archivos que tenían información acerca de los detalles de los avisos. Aquí fue dónde se agregó el archivo con los detalles faltantes mencionados anteriormente.

Se optó por dejar de lado algunas características de los avisos que parecían no aportar mucha información y ocupar espacio de más (algo que en este trabajo era muy valioso). Las características quitadas fueron:

- **País:** todos los detalles de avisos que se tenían, tenían idpais = 1, por lo que ningún algoritmo iba a poder sacar información útil de esa característica.

- **Mapacalle:** en este caso eran muy pocos los avisos que tenían un valor no nulo para mapacalle, por lo que no parecía una característica útil ya que faltaba en la mayoría de los avisos y por lo tanto no iba a servir para que el algoritmo detectara algún patrón.

- **Ciudad:** misma situación que con mapacalle

- **Descripción:** principalmente se dejó de lado porque las entradas de texto eran muy subjetivas al personal que ingresó el texto, y muy puntuales a la publicación. Además que sacarla nos permite un ahorro importante de RAM. Consideramos que el título era una mejor característica para fines similares, ya que es más representativa del anuncio en cuestión, y permite comparaciones con títulos de otros avisos.

- **Denominación de empresa:** se utilizó mediante una técnica, pero luego se descartó. Ver en el punto de análisis de problemas y cambios en los datos para mayor detalle.

Se transformaron a categóricas las variables nivel laboral, tipo de trabajo y nombre del área. Finalmente se eliminaron los avisos duplicados que pudieran existir al concatenar los distintos archivos de avisos.

2.1.5 Edad y sexo

Se concatenaron todos los archivos que tenían información acerca de la fecha de nacimiento y género de los postulantes. Hubo un caso en particular que tenía como genero “0.0”. Para ese caso se decidió tomarlo como “No declara”.

En el caso de las fechas de nacimiento, había algunos postulantes con valores nulos. Para estos casos se decidió usar la fecha de nacimiento promedio. Luego se hizo el cálculo de las edades y los valores que estaban por debajo de 18 o encima de 80, se pasaron a 28 que era la edad promedio.

2.1.6 Postulaciones

Se concatenaron todos los archivos que tenían información acerca de las postulaciones. Se acomodaron los datos pasándose a formatos más coherentes (fechas a datetime por ejemplo).

Para el caso de que un postulante se postularse a un aviso más de una vez, se decidió mantener la primer postulación solamente.

Finalmente se utilizó la información obtenida anteriormente para agregar información a las postulaciones. Es decir, agregar información acerca del postulante (género y edad por ejemplo) y agregar información acerca del aviso (zona y tipo de trabajo por ejemplo), logrando así un archivo de postulaciones con información tanto del postulante, como del aviso.

2.1.7 Generación de no postulaciones

Para lograr predecir si un postulante si iba a postular o no a un aviso, necesitábamos tener información sobre casos negativos. Es decir, tener postulantes con avisos a los que no se habían postulado.

Para eso se usó como base de userid la intersección de los set de edad y género, y set de estudios. Luego se usó el archivo de postulaciones con el mismo tratamiento que en **2.1.6** . Adicionalmente se calculó la cantidad de personas que se postularon a cada aviso, para así generar la misma cantidad de no postulaciones. Esto se hizo con la idea de que el archivo final que tuviera tanto postulaciones, como no postulaciones, estuviera balanceado.

Para una idea más técnica de cómo se generaron las no postulaciones puede consultarse el notebook: **generador_set_no_postulaciones** . La idea básicamente fue buscar por cada aviso una cantidad de postulantes igual a la cantidad de postulantes que se calculó anteriormente, habiendo chequeado vía duplicados que se tratase de postulantes que no se habían postulado al aviso.

Luego de generar las no postulaciones se procedió de la misma forma que en **2.1.6** agregando información tanto de postulantes, como de los avisos.

2.1.8 Generación de un archivo unificado

Simplemente consistió en unir los archivos de postulaciones con el de no postulaciones, teniendo ambos información de los postulantes y de los avisos. Al contar con un set total de postulaciones de alrededor de 6 millones de registros, y ya que generamos un set de no-postulaciones de la misma cantidad, el set total completo inicial nos quedó de alrededor de 12 millones de registros.

2.1.9 Test 100 k

El archivo de test de 100k registros también se trató como a las postulaciones y no postulaciones. Es decir, se le agregó información acerca de los postulantes y de los avisos.

2.2 Preprocesamiento de los datos - Feature Engineering

2.2.1 Introducción

Al tener ya armada una estructura básica de datos completa, con toda la información integrada del aviso y del postulante, nos encontramos en la dificultad de que, lógicamente, los métodos de machine learning no aceptaban los datos en ese formato, con tantos campos en texto, categóricos, o con datos numéricos no preparados.

Luego, era necesario llevar toda la información a una representación más simple que los algoritmos entendiesen, una representación numérica. Las técnicas para hacerlo, y el modo en que se llevó a cabo se detallan a continuación para cada feature.

2.2.2 Principales técnicas usadas

Básicamente se usó **Label Encoding**, que consiste en transformar, en los casos de variables categóricas, todos los labels disponibles a un número. Por ejemplo, en el caso de sexo, que puede tomar únicos valores masculino-femenino-no_declara, label encoding lo transforma a 0-1-2 por ejemplo. Es un acercamiento bastante simple, pero puede generarnos problemas en los casos en los que el encoding resultante nos codifica numéricamente los valores con cierto orden que no es representativo de un orden real entre las variables categóricas (o que quizás ni exista dicho orden). En el ejemplo anterior, podría interpretarse que no_declara (de valor 2), estaría “más cerca” a femenino (de valor 1) que a masculino (de valor 0), lo cual obviamente no es así, y nos puede generar problemas.

Además se usó **One Hot Encoding**, que consiste en transformar un feature que puede tomar N valores acotados, a N features de valores binarios únicos 1 o 0, que marcan cada una si en ese registro existe o no dicho valor. Esto resolvería el problema del orden mencionado anteriormente en Label Encoding. Así, siguiendo el ejemplo, nos quedaría 3 features, c/u representando con 1 o 0 si el sexo es masc,fem o no_decl. Un posible problema que se nos puede presentar es que si una variable categórica presenta muchos valores únicos, esto nos generaría muchas dimensiones extra en nuestro set de datos.

2.2.2 Estudios

Para representar los estudios, como se mencionó antes se usaron 2 features indicando el nivel de estudios máximo de estudios graduados, y en curso. En este caso, utilizando un label encoding del estudio basado en el orden de importancia desde Secundario como el más bajo a Doctorado como más alto, no tenemos el problema mencionado en Label Encoding, porque aquí sí tenemos datos con cierto orden asignado. Secundario está más cerca de Universitario que de Doctorado, y así con el resto.

2.2.3 Edad y género

En este caso, al contar con un set gigante de 12 millones de registros, decidimos descartar los géneros no_declara (muy muy bajo porcentaje sobre el total), y quedarnos con MASC Y FEM. De esta forma, con una única feature, y usando Label Encoding pudimos codificar binariamente ambos valores.

En cuanto a la edad, se dejó numéricamente el valor ya existente, ya que lógicamente el orden existe.

2.2.4 Nivel laboral

Para la información de nivel laboral usamos Label Encoding y One Hot Encoding, resultando en varias features binarias representando Ssr-Sr, Junior, Jefe, etc.

2.2.5 Tipo de trabajo

Aquí nuevamente se procedió a codificar numéricamente la información mediante One Hot Encoding y obtuvimos 3 features binarias para indicar con 1 o 0 si ese anuncio corresponde al tipo particular de la variable. Hay que mencionar que el set se dividía en Full-time, Part-time y un número de valores más (ej teletrabajo, por hora, etc) cuya incidencia en el set total era bajísima. Se procedió a catalogar todos esos valores como tipo “Otro”.

2.2.6 Zona geográfica

Usamos One Hot Encoding para la codificación únicamente en 2 features: zona_caba y zona_gba. Existían muy pocos valores dispersos por BsAs que englobamos dentro de zona_gba.

2.2.7 Nombre del área del anuncio

Evaluando los resultados, vimos que manteniendo el feature original con Label Encoding, y al mismo tiempo sumando N features derivadas de realizar One Hot Encoding sobre las valores de nombre_area, obtenemos una mejora de score mayor que si utilizaramos ambas cosas por separado.

2.2.8 Nuevas features obtenidas

Al realizar las primeras pruebas con los datos iniciales (representados numéricamente, y todo listo para entrenar) observamos que observamos muy bajos scores, y esto no variaba según el algoritmo o los parámetros. Analizando este problema, entendimos que lo más importante era realizar un buen feature engineering y preprocesamiento de los datos que nos aportarán mucha ganancia de información, cosa que no estábamos teniendo. Luego aplicar distintos algoritmos y probar era algo un poco más metódico y trivial, pero si nuestras features no eran buenas, por mas que se prueben algoritmos distintos, el resultado muy bueno no va a resultar.

Luego, decidimos concentrarnos principalmente en la inyección de nuevas features que pudiésemos descubrir que nos generaran una ganancia de información y un mejor score. Entonces, adicional a las features básicas que se podían extraer de los sets provistos, incorporamos la siguiente información derivada de ciertos calculos y analisis:

- Cantidad de veces que vio el idpostulante ese idaviso.
- Cantidad de postulaciones totales de distintos idpostulante para un idaviso.
- Cantidad de vistas totales de distintos idpostulante para un idaviso.
- Historial de cantidad de postulaciones a las distintas áreas para un idpostulante.
- Vectorización del título de un idaviso

2.2.9 Cantidad de veces que el usuario visitó el anuncio

Al momento de pensar de qué forma extraer nuevas features que sumen información y mejoren la predicción de los algoritmos, intuitivamente se pensó en las vistas, ya que se había usado poco y nada de ese set y era posible que alguna información valiosa estuviese escondida a la espera de ser descubierta.

Luego, se pensó que si una persona había visto un anuncio, era porque seguramente le interesaba, y por ende esa información era valiosa. Entonces, se procedió al cálculo de la cantidad de vistas de cada postulante a cada aviso, y la incorporamos como una nueva feature numérica. En este caso tenemos la opción de pensarla como un valor binario (lo vio antes o no) o directamente utilizar la cantidad de veces visto. Decidimos esta última opción porque al comparar ambas, obtuvimos mejor resultado con la cantidad de veces (lo cual tiene sentido ya que cuanta más veces vio el anuncio, debería, en teoría, ser más probable que se postule).

2.2.10 Cantidad de postulaciones y vistas totales para un anuncio

Pensamos que los anuncios de mayor penetración, es decir, los más “populares” por decirlo de cierta forma, tendrán mayores chances de absorber mayor cantidad de postulantes que otros de menor penetración. Es por eso que decidimos incorporar 2 features indicadoras de la cantidad de personas distintas que visitaron y se postularon a un determinado idaviso. Analizando los resultados, obtuvimos un aumento en los scores de predicciones, por lo que resultaron útiles.

2.2.11 Historial de postulaciones a distintas áreas para un usuario

En el TP1, habíamos analizado la relación que existía entre las distintas categorías o áreas a la que un anuncio pertenecía, y mediante el algoritmo apriori pudimos extraer interesantes conclusiones sobre qué porcentaje de las personas que se postulaban a cierta área, se postulaban también a otra área determinada.

Bajo esta idea se nos ocurrió entonces incorporar un historial para cada usuario de cantidad de veces que se postuló a cada una de las distintas áreas. De esta forma, el algoritmo debería aprender por el análisis de miles de casos similares, que una postulación efectiva al anuncio de área X se corresponde con valores altos por ejemplo en el historial de muchos usuarios en área Y. Entonces, de esta manera se pueden obtener relaciones que ayudan a la predicción de las postulaciones. Si en un registro que se intenta predecir, el anuncio es de un área X, y el historial del área Y para esa persona es muy bajo o 0, en base a las relaciones aprendidas y entrenadas la predicción seguramente será que no se postulará.

Nuevamente, se decidió incorporar la cantidad de veces (mayor o igual a 0) y no valor único binario (se postuló o no a esa otra categoría), porque nos genera mayor ganancia de información y score, lo cual además tiene sentido, ya que el caso de una persona que se postuló muchas veces a cierta área, es distinto a otra que se postuló solo 1 vez.

Esta información fue la que más ganancia de score e información nos dió, en un aumento cercanos al orden del 20%, y se incorporó al set mediante 188 features, cada una indicando la cantidad de veces que se postuló a cada una de las 188 áreas totales.

Se probó realizando una estandarización con el historial basada en el StandardScaler de sklearn, pero los resultados no mejoraron.

2.2.12 Word Embedding. Vectorización del título

Otra de las cosas con las que no estábamos trabajando y sabíamos que podía otorgar información útil para las predicciones, era el texto del título del aviso. En un principio intentamos el acercamiento más básico de contar la cantidad de N palabras más comunes que tenia cada título, y con eso armar una cantidad N de features para cada aviso que indicaran la cantidad de veces que contenía cada palabra.

Luego se investigó y una forma que parecía bastante simple y potente era representar las palabras como vectores. Para esto utilizamos el método de Word Vectorizing, que consiste en representar las palabras como vectores de una dimensión deseada N (en nuestro caso utilizamos 100), y para hacerlo se basa en un diccionario de todas las palabras posibles que se pueden recibir.

Se realizó mediante el modelo de Word2Vec, presente en la librería para python Gensim. De esta forma sumamos 100 features extras en punto flotante de 32 bits.

Hay que mencionar que previo a la vectorización de las palabras del título, se realizó un preprocesamiento del mismo. Se quitaron puntuaciones, caracteres o símbolos extraños, caracteres numéricos, palabras de longitud menor a dos caracteres, se pasó todo a minúsculas, se realizó una eliminación de palabras stopwords (para, de, en, el, la, …, etc.), se realizó stemming, filtro de palabras de baja frecuencia total, etc.

Por último, para representar una frase completa (en nuestro caso el título), es decir, una unión de varias palabras, lo que hicimos fue vectorizar cada una de las palabras y aplicar el promedio: sumar todos los vectores y dividirlos por la cantidad de palabras. Es una forma posible de vectorizar toda la oración. Además, se realizó una normalización de todos los vectores palabra, y verificamos un mejor funcionamiento de esta forma al obtener un leve aumento de score.

3.0 Análisis de problemas y cambios en los datos

Luego de varias pruebas con distintos algoritmos, empezó a notarse que por más que el algoritmo fuera de los mejores para clasificar, o incluso utilizar técnicas para lograr los mejores hiper parámetros (grid search+cross validation por ejemplo), o haciendo ensambles, no se lograba superar determinado techo de score. Por lo tanto, se empezó a observar el comportamiento de los algoritmos con los datos que se les proporcionaban. Los problemas encontrados fueron los siguientes:

- Había un caso de overfitting debido al campo **veces_visto**. Esto sucedía porque cuando un usuario no se había postulado a un aviso, en la mayoría de los casos no había visto el aviso. Por lo tanto quedaban las no postulaciones con la mayoría de veces visto en 0 y las postulaciones con veces visto mayor a 1. Además se había tomado la decisión de llenar los valores nulos para las postulaciones con veces visto igual a 1 (por hacer la suposición de que si se había postulado, lo había visto). Esto generó que los algoritmos encontraran como algo fundamental el hecho de si un postulante había visto o no el aviso, para predecir su postulación. Para solucionar esto, no se hizo la suposición mencionada anteriormente, y simplemente se procedió a tomar la información real sobre si existió o no la vista al anuncio, y computar la cantidad de veces que lo vió.

- Mediante un análisis exhaustivo de los resultados con o sin cada una de las features, pudimos observar si éstas nos generaban ganancia de información (y un mejor score) o ruido en nuestros algoritmos de ML. Con dicho análisis pudimos notar que el campo “edad” nos estaba generando ruido y nos bajaba la calificación de

nuestro score, por lo que decidimos dejarla de lado. No así con las demás, que todas reportaban pequeñas ganancias de score.

- La cantidad de datos usada también significó un problema. En total había alrededor de 13 millones de registros, lo que hacía muy difícil correr los algoritmos, incluso para equipos con buenas características. No estaba claro si se podía reducir la cantidad de datos sin afectar la precisión. Haciendo un sucesivas pruebas en las cuales se iba disminuyendo la cantidad de muestras, veíamos una muy leve reducción progresiva de score si bajábamos hasta 1 millón aproximadamente, y luego un poco más marcada desde 1 millon hasta 200 mil, aunque leve igualmente. Por ello comenzamos a realizar la mayoría de las pruebas con samples aleatorias de 1M del el set total de 12M.

- En un principio utilizamos la información del nombre de la empresa del aviso (denominacion_empresa). Realizamos un análisis de las empresas de mayor penetración o dominio sobre el total del set, e incorporamos las 50 primeras en 50 features para indicar si ese anuncio era o no de esa empresa. Esta decisión se basó en que muchas veces las personas buscan trabajo según la empresa, o “siguen” a ciertas personas para ver cuando sacan nuevas búsquedas laborales, por lo que incide en cierta forma en la chanche de postularse para esas personas.

  No obstante, obtuvimos una reducción de score con estas features, por lo que solo nos generaban ruido. Adjudicamos este ruido al hecho de que la mayoría de las empresas con mayor dominio eran las consultoras de RRHH, y no firmas puntuales, por ende saber que un aviso lo postulaba una consultora no nos aportaba demasiado porque no agrega ninguna información valiosa sobre alguna característica de ese aviso (ya que las consultoras ofrecen avisos de todo tipo y demasiado variados).

4.0 Entrenamiento de los datos. Análisis de los resultados y performance.

4.1 Introducción

Los algoritmos probados en este trabajo, fueron los que en teoría se comportan mejor para problemas de clasificación. Se decidió utilizarlos porque este trabajo consistía en una clasificación binaria, acerca de si un postulante se postuló o no a un aviso. De todas formas se sabe que por el teorema de no free lunch, no significa que el mejor algoritmo de clasificación, sea el que mejor predice para este problema.

En todos los casos se dividieron los sets de prueba en 70% para entrenamiento, y el 30% restante para testing, haciendo uso o no varias veces de Cross Validation para obtener el score más verídico.

4.2 Aclaración de conceptos

Se aclaran conceptos que se utilizan a lo largo del informe. De esta manera luego de esta sección, se nombrará al método suponiendo que el lector sabe de qué se trata..

Grid search: es una forma de probar muchos hiper parámetros para un determinado algoritmo y evaluar en base a los resultados cuáles son los óptimos.

Cross Validation: se basa en dividir el set en k conjuntos, entrenar con k-r conjuntos y testear con r conjuntos.

GridSearchCV: En nuestro caso utilizaremos la modalidad de GridSearchCV que provee la librería de machine learning sklearn para python, que realiza ambas operaciones en conjunto para cada combinación de parámetros e intenta reducir el overfitting y obtener un score lo más verídico posible.

Bootstrapping: consiste en tomar una muestra del set de entrenamiento del mismo tamaño del set, pero con reemplazo.

Bagging: consiste en aplicar un clasificador n veces y luego promediar sus resultados. Se usa en conjunto con bootstrapping.

Boosting: se entrena un algoritmo simple, se analizan los resultados y luego se entrena otro algoritmo en donde se da mayor peso a los resultados en los que el algoritmo anterior funcionó peor.

4.3 Árboles de decisión y ensambles de árboles. Bosques.

4.3.1 Árboles de decisión estándar

El primer algoritmo que usamos y probamos. Lógicamente en las primeras pruebas sin ningún tipo de preprocesamiento o feature engineering, los resultados fueron muy bajos, casi se diría que azarosos, pero a medida que se mejoraron las features, el método de árboles de decisión escaló rápidamente en score, pero cuando comenzamos a probar Random Forest, y vimos que obteníamos mejores resultados y menor overfitting, lo descartamos. Ya por el final, corrimos nuevamente varias instancias este método variando algunos parámetros, para compararlo con el Random Forest que veníamos manejando (ya en el orden de 90-91% de score en nuestros sets de test), y lo mejor que obtuvimos fue en el orden del 87%.

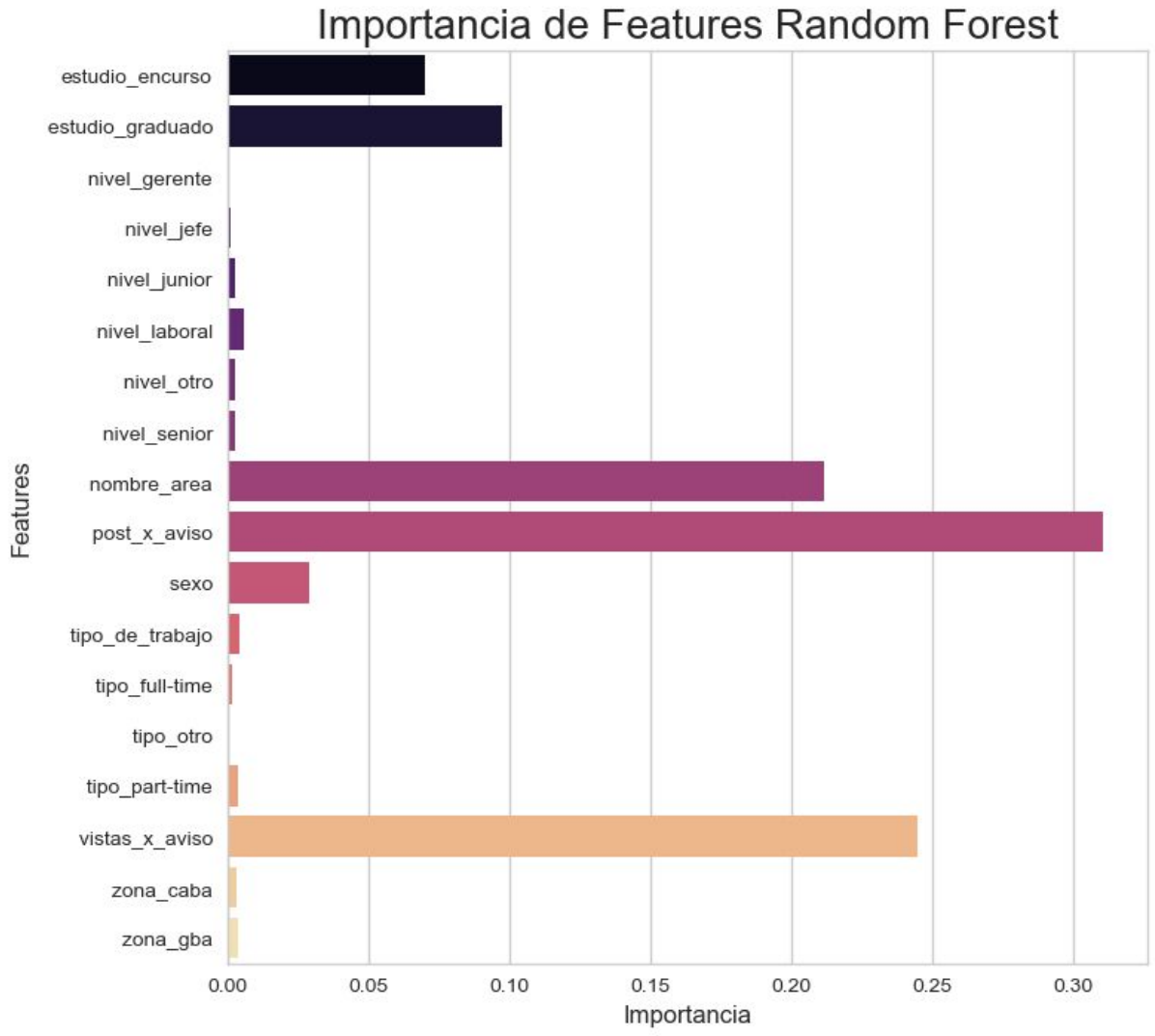
4.3.2 Random Forest

Este algoritmo de aprendizaje trabaja construyendo una gran cantidad de árboles de decisión muy poco profundos y luego toma la clase que cada árbol eligió.

Fue el algoritmo que más usamos para todas las pruebas de mejora de las features, y para los mejores scores finales, ya que en casi todos los casos daba los mejores resultados, y con un tiempo de procesamiento muy bajo, lo que nos permite ir corrigiendo y testeando cosas muy rápido sin tener ese factor en contra.

Los resultados obtenidos fueron escalando desde el 67% test y 98% train inicial que obtuvimos con las features bases que contábamos, hasta un 91% test y 94% train finales que obtuvimos luego de mucho tuning de hiperparametros, GridSerachCV, y de la adición de muchas features importantes como el historial de postulaciones, popularidad del anuncio, o vectorización del título.

A continuación se muestra un ejemplo de la importancia de algunos features en este algoritmo luego de un entrenamiento:



4.3.2 Extreme Random Forest

Probamos esta variante más extrema de random forest, donde se le da un peso importante a la aleatoriedad con la que se eligen la máxima cantidad de features y árboles. No observamos ninguna mejoría notable con respecto a Random Forest, por lo que continuamos con este último porque era más performante. Luego de un poco de tuning, obtuvimos scores similares a los que estábamos observando con el otro algoritmo, un poco por debajo: en la etapa que estábamos con un random forest del orden de 67%, nos arrojaba resultados del 63-64% como mucho, luego de algunos ajustes paramétricos.

4.4 Boosting

4.4.1 Adaptive Boosting

Los métodos de boosting dentro de todo nos arrojaron buenos resultados. Tanto AdaBoost como GTBoosting se acercaron bastante luego de algunos GridSearch y refinamiento de parámetros a lo que veníamos obteniendo con nuestro algoritmo principal.
Cuando con Random Forest estábamos por el orden de 86% en nuestro test score, AdaBoost nos entregó como mejor score:

Test: 0.80519 Entrenamiento: 0.8056842857142857

4.4.2 Gradient Tree Boosting

Otro buen método de boosting, muy cercano al 86% Random Forest en los resultados obtenidos, dándonos, como mejor aproximación luego de GridSearch intensivo:

Test: 0.83367 Entrenamiento: 0.8345742857142857

La ventaja de GradientTreeBosting es que nos reducía un poco el overfitting de nuestro set de entrenamiento, pero también nos generaba en menor medida un aumento de Variance.

Creemos que estos resultados podrían haber sido mejores, pero al requerir mayor consumo de recursos que random forest, y debido a que no escala muy bien para muchos datos y no es paralelizable, el rango de valores investigados con GridSearchCV se acotaba y mucho, en comparación con todo lo ya investigado y “tuneado” exhaustivamente de muchas formas en el Random Forest que estábamos utilizado como “buque insignia”.

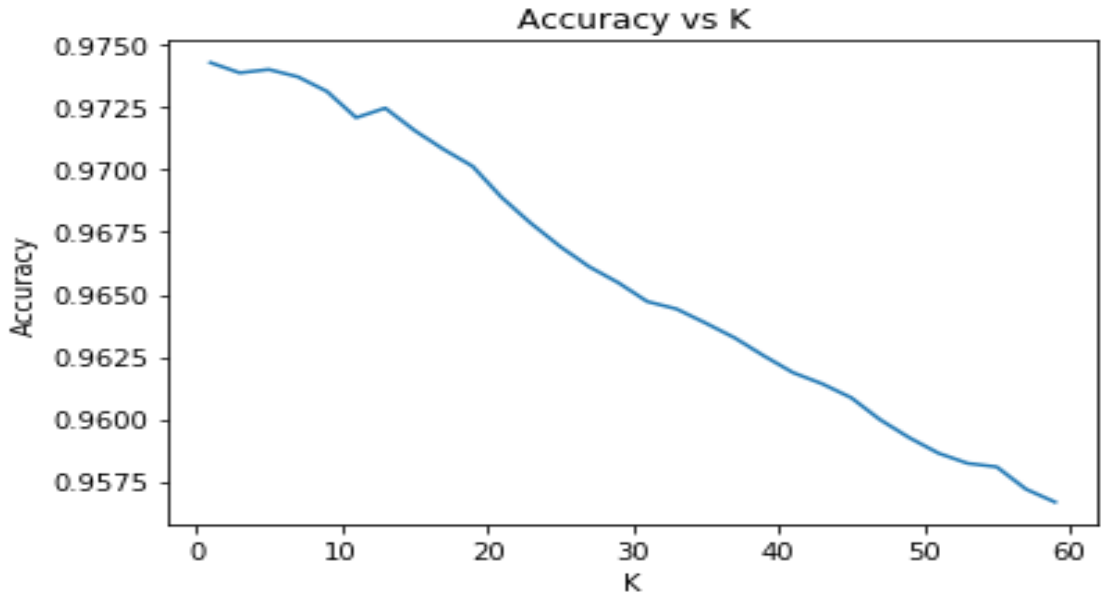
4.5 Otros

- Gaussian Naive Bayes: Uno de los resultados más bajos que obtuvimos, aún luego de cierta búsqueda con distintos parámetros, este método bastante rápido y muy simple nos arrojó como mejor score 0.5366933333333334 cuando con Random Forest en esta etapa estábamos por el 67%.

- Logistic Regression: Los resultados obtenidos fueron pobres, creemos que no tuvo tanto que ver con los parámetros en sí, sino en que este modelo en particular no se adapta bien a nuestros datos. Cuando manejamos scores del 67% con Random Forest, luego de algunas pruebas con distintos parámetros, lo mejor que obtuvimos fue Test: 0.5928966666666666.

4.6 KNN

Knn es un algoritmo que según fue explicado en clase, siempre debe probarse. Es un algoritmo que busca para un determinado punto sus K-vecinos más cercanos. Lo primero que se hizo fue proceder a buscar el K óptimo. Para ello, se usaron varios valores de K mientras se calculaba la precisión del algoritmo con ese determinado hiper parámetro. Probando valores impares hasta el 59, obtuvimos el siguiente gráfico:



El resultado numérico fue el siguiente:

Se concluyó que los mejores valores de K se encontraban entre 5 y 7.
Posteriormente se hicieron predicciones con K = 5 y K = 7, dando mejor score la de K = 7.

Toda esta información se obtuvo usando el set de datos original, es decir, **el que tenía errores que posteriormente fueron corregidos.**

Para todas las pruebas de Knn se utilizó un sample de 1000000 del set de datos, debido a que se hacía imposible correr el algoritmo con el set completo.

```
k=1, accuracy=97.43%
k=3, accuracy=97.39%
k=5, accuracy=97.40%
k=7, accuracy=97.37%
k=9, accuracy=97.31%
k=11, accuracy=97.21%
k=13, accuracy=97.25%
k=15, accuracy=97.16%
k=17, accuracy=97.08%
k=19, accuracy=97.01%
k=21, accuracy=96.89%
k=23, accuracy=96.79%
k=25, accuracy=96.69%
k=27, accuracy=96.61%
k=29, accuracy=96.55%
k=31, accuracy=96.47%
k=33, accuracy=96.44%
k=35, accuracy=96.39%
k=37, accuracy=96.33%
k=39, accuracy=96.25%
k=41, accuracy=96.19%
k=43, accuracy=96.14%
k=45, accuracy=96.09%
k=47, accuracy=96.00%
k=49, accuracy=95.93%
k=51, accuracy=95.86%
k=53, accuracy=95.82%
k=55, accuracy=95.81%
k=57, accuracy=95.72%
k=59, accuracy=95.67%
```

En la parte de validación se obtuvieron muy buenos valores, pero luego el score de Kaggle no era acorde. De hecho una forma de notar que no estaba generándose una predicción muy acertada era porque se sabía que el archivo de test, estaba balanceado. Es decir, que tenía igual cantidad de postulaciones como de no postulaciones. Knn arrojaba siempre una inclinación hacía el lado de no postulaciones.

Se probó con k = 5,7 y 8. Como era de esperarse, el mejor valor obtenido fue con k=7 que arrojó un score de **0.78459**.

Luego de estos resultados se probó con algoritmos diferentes notando un problema con los datos. Por lo tanto las pruebas posteriores, fueron con el set de datos actualizado, como se explicó anteriormente.

Como se explicó al principio de esta sección, knn es un algoritmo a probar, pero no suele ser el que da el mejor resultado. Por lo tanto, luego de actualizar los datos, se consideró volver a probar knn con k = 7, que había sido el mejor resultado obtenido. Esta vez, el algoritmo seteado con exactamente los mismos hiper parámetros que antes dió un

score de **0.81537** demostrando que el cambio en los datos había sido efectivo, pero aún asi continuaba bastante por debajo del resultado de Random Forest, además de que la demanda de tiempo y poder de cálculo que exigía era muchísimo mayor en el caso de KNN, lo que nos obligaba a usar muestras mucho más pequeñas, y nos permitía ir probando los cambios en las features a medida íbamos mejorandolas.

4.7 SVM (Support Vector Machine)

Hicimos algunas pruebas con SVM, variando principalmente el parámetro más importante, C (costo de clasificar mal), en un principio manualmente. Los resultados obtenidos fueron bastante pobres, además de un esfuerzo computacional mayor que en algoritmos de árboles. Pensamos que la razón puede ser que inicialmente no sea un set muy fácil de separar linealmente. Como todavía realizabamos pruebas según mejorabamos el preprocesamiento y feature engineering, lo descartamos por tener una mejor opción como Random Forest. A continuación, el mejor resultado con SVM que obtuvimos fue para un C de 1.0: Test: 0.7825833333333333 Entrenamiento: 0.7822107142857143

4.8 Algoritmo final

Para el resultado final utilizamos lo que mejor nos estuvo funcionando, que fue Random Forest, con “tuning” de hiperparametros lo mejor posible mediante muchas pruebas manuales y **GridSearchCV**, que fuimos repitiendo y modificando según íbamos mejorando el preprocesamiento de los datos, haciendo feature engineering y agregando nuevas variables. En el camino fueron quedando algunos algoritmos como NaiveBayes, LogisticRegression, SVM, que resultaron no adecuados para el tipo de datos que teníamos; otros como KNN, cuya potencia computacional y tiempos de procesamiento al calcular muchas distancias aumentaba considerablemente y lo hacia poco viable, y otros como las otras variantes basadas en ensambles de árboles (que resultaron buenas pero no se acercaron a Random Forest), y en especial los métodos de Boosting, que parecían muy prometedores, pero requerian más tiempo de búsqueda y procesamiento para obtener los mejores hiperparametros que no pudimos encontrar, y no podían competir contra el Random Forest cuyos parámetros ya teníamos muy estudiados y optimizados.

Obtuvimos scores del orden de **88,5%** aproximadamente, que pueden mejorarse algunos puntos más si aumentamos la cantidad de muestras a procesar, pero no disponemos de la potencia de cálculo ni memoria.

La última configuración de parámetros que mejor nos estuvo funcionando fue:

- Nro de árboles: 128
- Mínimo de muestras por hoja: 7
- Mínimo de muestras necesarias para un split: 3
- Máxima profundidad: hasta alcanzar mínimo de muestras por hoja
- Máximo de features por árbol: raíz cuadrada de número de features totales
- Bootstrap: No

5.0 Consideraciones

En un acercamiento inicial, empezamos a intentar correr distintos algoritmos pero obteniamos muy bajos scores sin importar cuanto variasemos los parámetros o algoritmos. Luego de investigar la causa de esto, concluimos en que lo más importante no era el algoritmo en sí a utilizar, o por lo menos no en lo que se refiere al grueso de armar un modelo predictivo, sino que teníamos que concentrarnos en hacer un buen preprocesamiento y feature engineering de nuestros datos, y encontrar la mejor forma de representar la mayor cantidad de información útil para los algoritmos. Ya que por más algoritmos que uno pruebe, si la base no es buena, no funcionará.

Es por ello que nos concentramos en realizar muchas modificaciones a nuestros datos de partida, tratando de incorporar nuevas features de las cuales los algoritmos puedan aprender mejor. Por ende en el informe se explayan más en detalle estos temas, y de querer más detalles sobre los resultados numéricos puede referirse a los notebooks en el repositorio del TP. Como este proceso lo íbamos acompañando con testing y tuning de un random forest, no es de extrañar que el mejor resultado lo hayamos obtenido con dicho algoritmo, que ya estaba mucho más aceitado y adaptado a nuestros datos en términos de parámetros.

Todos los resultados y pruebas de los distintos algoritmos, scores, evaluaciones y comparativas, features, datos cambiados, problemas encontrados, preprocesamiento, feature engineering, etc. se pueden encontrar en los distintos notebooks presentes en el repositorio de GitHub. Los nombres de los notebooks intentan ser lo más descriptivos posibles. Aún así puede ser difícil seguir el hilo, en dicho caso, lo mejor es guiarse por la fecha de commit del archivo, y así se va siguiendo el proceso y progreso de trabajo que fuimos realizando.