TP N°2 : MACHINE LEARNING

Padrón	Apellido y Nombre
98124	Horn, Miguel Agustín
98591	Fonseca, Matias Manuel
98570	Zambianchi, Victoria Laura
99732	Fernandez, Olivia

Repositorio: https://github.com/matfonseca/OrgaDatos

Índice:

1 - **Introducción:**

El objetivo del presente trabajo práctico es predecir la probabilidad que un usuario se postule a un determinado aviso. Para esto, se nos brindó un set de datos del presente año hasta el 15 de abril, en los cuales poseemos información tanto de los avisos como de los postulantes. Entre algunas características están el sexo, la fecha de nacimiento, la educación, la descripción de los avisos, los títulos, etc. También se nos otorgaron 100 mil datos del estilo de postulante-aviso, los cuales son los que tuvimos que predecir su probabilidad de postulación.
Para lograr este objetivo, se necesitó generar sets de datos que nos sirvan para entrenar modelos de machine learning, así con estos poder predecir la probabilidad de las postulaciones mencionados anteriormente, a los que nombramos como test. Debido a que no existe una solución inmediata, es necesario encontrar un set de datos que contenga la información más relevante y así, ir probando distintos modelos con distintos datasets. Asimismo, debemos variar los hiperparámetros de nuestros modelos, ajustándose al resultado más óptimo.

2 - **Exploración, depuración y algoritmos:**

Como se dijo en la introducción, se nos otorgó con varios datasets como por ejemplo: detalles sobre los avisos, detalles sobre los postulantes, detalles sobre las vistas, entre otros. Para empezar, decidimos hacer 2 datasets, uno que sea solamente de los postulantes y otro solamente sobre los avisos. Al realizar esto, pudimos analizar los datos, depurarlos y aplicar algunos algoritmos para poder utilizar mejor esta información. Algunos algoritmos utilizados fueron la función *education_cleaning()*, y más adelante la función *datetime()* y una clase *DataFrameImputer()* para rellenar los valores nulos.

Luego, generamos un set de datos de postulaciones de usuarios, que contiene tanto información sobre el postulante como el aviso al que se postula. Además, le agregamos una columna que indica la probabilidad de postulación, y, como es un set de postulaciones, son todos 1.

Como solo tenemos datos sobre las postulaciones, debíamos generar ruido o “postulaciones falsas” para poder entrenar mejor los distintos algoritmos. Por este motivo, creamos un set de datos de no postulaciones. Indicando en una columna una probabilidad de postulación igual a 0. Para generar este set de ruido, utilizamos una función que iteraba, una cierta cantidad de veces, y matcheaba con un random un id de un postulante con un id de los avisos. A este set también le agregamos los detalle de cada postulante y aviso.

Con estos dos sets, generamos nuestro set de entrenamiento concatenando estos en un gran dataframe, al cual lo usaremos como base para aplicar los algoritmos de machine learning.

Ahora debíamos definir cómo mapear nuestros datos para que generen resultados óptimos.Por ejemplo, un campo que tenía mucha información, lo cual generaba un problema para la memoria, era el de descripción del aviso. Entonces,decidimos aplicarle un algoritmo que encontraba las 100 palabras con más frecuencia (Most Frequent Words) y a partir de esto elegir 23 que nos parecían más relevantes. Luego, reemplazar el contenido de este campo con un algoritmo que colocaba un 1 si el campo tenía al menos 4 palabras de la lista de 23 y un 0 en caso contrario.

En los datos, había varios campos que tenían valores categóricos repetidos, como el caso de: estado, nivel laboral, tipo de trabajo, sexo, nombre de zona,nombre(educación), etc. Lo que hicimos con estos features, fue reemplazar ese string que tenían en el campo con un número que identificaba a ese valor, en cada campo donde aparecía(codificación ordinal); esto no siempre ayudó, ya que daba indicio de un cierto orden (prioridad) dentro de las categorías, lo cual no es cierto. Por lo explicado recientemente, en algunos algoritmos realizamos un mapeo de One Hot Encoding con algunos features para ver cómo funcionaban de ésta manera. También generamos algunos features extra que pudimos rescatar del primer trabajo práctico, como cantidad de vistas por aviso, cantidad de postulantes por aviso, área con más postulaciones, etc. Y reemplazamos la columna de fecha de nacimiento por una que es edad del postulante, información que nos parecía más relevante a la hora de predecir.

Para decidir qué cantidad de “no postulaciones” generar, lo que hicimos fue submitear en la competencia un csv de test con todos unos y otro con todos ceros. Así, obtuvimos una estimación de la proporción que teníamos de postulaciones-nopostulaciones. Como en ambos casos obtuvimos un 0,5 de score en kaggle, decidimos entrenas con esa proporción de datos (50% - 50%). Probamos entrenar con distintas proporciones pero siempre dio malos resultados (predecía más 1s que 0s).

Algo que se tuvo muy en cuenta fue el tamaño del set de entrenamiento. Como la cantidad de postulaciones supera los seis millones de registros, generamos 6 millones de “no postulaciones”. Por lo que, en total había más de 12 millones de registros. Esto era un serio problema, ya que no era soportado por la memoria de la mayoría de las computadoras que teníamos a disposición. La realidad que para poder trabajar con esa cantidad de datos necesitamos una computadora de mas 8gb de ram, como teníamos solamente 1 que cumplía esa característica, decidimos

usar menos cantidad de datos(seleccionados de manera aleatoria, manteniendo la proporción de 50%) que serán aclarados en la sección de algoritmos.

Aclaraciones: A continuación se verán términos que utilizamos a la hora de mapear los datos, por eso definiremos aquí estos conceptos. *Simple map*: es el map como aparece en github. Los features son tratados con codificación ordinal, la cual hicimos manualmente. Además otros features que tenían muchas categorías distintas eran quitados. *MFW*: acrónimo para Most Frequent Words. Es nuestro mapeo para las descripciones, el cual se explicó más arriba. *OHE*: One Hot Encoding. Quiere decir que utilizamos la táctica de One Hot con algún feature. Además aclaramos que siempre se usaron mitad de datos como postulaciones y la otra mitad como ruido, excepto que se indique lo contrario.
Utilizamos la librería sklearn para obtener la mayoría de los algoritmos para nuestros modelos de machine learning.

3 - **Algoritmos:**

3.1 - Random Forest:

El primer algoritmo que decidimos utilizar para realzar las predicciones fue Random Forest. Este algoritmo es un método que combina una gran cantidad de árboles de decisión independientes, probados sobre conjuntos de datos aleatorios con igual distribución, donde cada árbol no usa el total de los atributos sino un subset de los mismos. Una de nuestras principales razones por la que elegimos este algoritmo fue que suele predecir buenos resultados para set de datos grandes y además,suele evitar overfitting.

A la hora de aplicarlo, tuvimos en cuenta el hiperparámetro que indica la cantidad de árboles a crear, y lo fuimos variando en busca de predicciones más exactas.
A continuación, se muestra el registro que llevamos con los resultados obtenidos al aplicar Random Forest con diferente cantidad de árboles, variando la forma de mapear los datos y cambiando la cantidad de datos a analizar.

Utilizando mapeo simple:

Algoritmo	Cantidad de datos	Hiperara-metros	Score Entrenamiento	Score Kaggle
Random Forest	12 millones	20 árboles	0,592324629	0,61967
Random Forest	12 millones	100 árboles	0,592327659	0,61989
Random Forest	12 millones	30 árboles	0,748260729	0,53232

Observaciones: Debemos decir que los primeros tres resultados no pueden ser tenidos en cuenta ya que en ese momento enviamos a kaggle un archivo que en vez de tener la probabilidad de postularse, tenía un valor que indicaba con un uno si se postuló o un cero si no se postuló. Por este motivo suponemos que recibimos scores bajos.

Cambiando el set de datos :

Algoritmo	Cantidad de datos	Hiperparametros y Set de datos	Score entrenamiento	Score Kaggle
Random Forest	6 millones	100 árboles	0,931311174	0,55094
Random Forest	4 millones	30 árboles, simple map sin ESTADO,SEXO,ZONA, ÁREA,NIVEL_Lab	0,559355455	0,76097
Random Forest	4 millones	100 árboles, simple map sin ESTADO,SEXO,ZONA, ÁREA,NIVEL_Lab	0,559374294	0,68168
Random Forest	4 millones	20 árboles, simple map sin ESTADO,SEXO,ZONA, ÁREA,NIVEL_Lab	0,559362990	0,68166
Random Forest	4 millones	30 árboles, simple map con MFW sin ESTADO,SEXO,ZONA ,ÁREA,NIVEL_Lab	0,561503102	0,65739

Observaciones: Variar el set de datos y disminuir la cantidad de features ayudó a conseguir un mejor porcentaje en Kaggle y además a realizar mejores predicciones.

De aquí podemos deducir que hay features más importantes que otros. Y además vemos exclusivamente cómo el hiperparámetro modifica nuestro score.

3.2 - Gradient Boosting:

Este algoritmo está basado en la idea de crear una regla de predicción altamente precisa combinando muchas reglas relativamente débiles e imprecisas. El objetivo es el de mejorar el rendimiento del algoritmo de aprendizaje, ya que lo trata como una "caja negra" al que se lo puede llamar repetidamente, como una subrutina.
En este caso el algoritmo base utilizado es el de Árboles de Decisión. La idea es utilizar árboles de regresión que producen valores reales para las divisiones y cuya salida se puede sumar, permitiendo que los resultados de los modelos subsiguientes sean agregados y corrijan los errores promediando las predicciones. Además, Gradient Boosting necesita de una función de pérdida para optimizar, que sea diferenciable.
A la hora de aplicar este algoritmo decidimos utilizar la función de pérdida por default que es la de regresión logística, y además fuimos variando la cantidad de árboles con el cual trabajará Gradient Boosting. También variamos la tasa de aprendizaje, el cual es el encargado de controlar el grado en que a cada árbol se le permite corregir los errores de los árboles anteriores.
Se probó también la performance del algoritmo utilizando distintos datasets, es decir dejar o sacamos algunos features.
El siguiente registro muestra las distintas formas en las cuales probamos el algoritmo y los resultados obtenidos.

Utilizando mapeo simple del set de datos:

Algoritmo	Cantidad de datos	Hiperparametros	Score Entrenamiento	Score Kaggle
Gradient Boosting	12 Millones	n_estimators=100, learning_rate=0.5	0,669457702	0,50634
Gradient Boosting	12 Millones	n_estimators=100, learning_rate=0.2	0,655366806	0,53324
Gradient Boosting	12 Millones	n_estimators=100, learning_rate=0.2	0,656817021	0,56762

Gradient Boosting	6 Millones	n_estimators=100, learning_rate=0.4	0,931179150	0,52921
Gradient Boosting	6 Millones	n_estimators=100, learning_rate=0.15	0,931280075	0,40641
Gradient Boosting	4 Millones	n_estimators=100, learning_rate=0.5	0,559537146	0,68082

Observaciones: Se decidió variar el learning rate para ver si logramos una mejora a la hora de predecir, lo cual podemos observar como influye esta tasa en las predicciones. El learning rate también nos ayuda a no caer en overfitting, lo cual nos empeoraría el modelo para predicciones futuras. Además variamos la cantidad de datos en el set(siempre utilizando mitad de datos de postulaciones reales y mitad de datos de no postulaciones) y pudimos concluir que la utilización de menor cantidad de datos nos daba mayor porcentaje a la hora de predecir.

3.3 - Desicion Tree:

Un árbol de decisión es un árbol binario que representa gráficamente las posibles decisiones que se basan en ciertas condiciones.
En cada nodo del árbol se divide el set de datos en dos de acuerdo a un cierto criterio, y se busca llegar a los nodos hoja, donde podemos clasificar los datos de manera correcta.
A la hora de probar este algoritmo, fuimos variando la cantidad de profundidad de los árboles de decisión y en algunas pruebas decidimos normalizar los valores de nuestro set de datos una vez que fueron mapeados.
A continuación mostramos una tabla con los valores obtenido en cada prueba.

Utilizando mapping de datos simple:

Algoritmo	Cantidad de datos	Hiperparametros	Score Entrenamiento	Score Kaggle
Decision Tree	8 Millones (2 M no postulaciones)	max_depth=4	0,756023031	0,50195
Decision Tree	8 Millones (2M no postulaciones)	max_depth=5	0,756023031	0,50354

Decision Tree	8 Millones (2M no postulaciones)	max_depth=6	0,756023031	0,69341

Observaciones: Remitiéndose a las pruebas realizadas, podemos decir que nos dio un mayor resultado utilizar mayor profundidad ya que obtuvimos un score más alto que utilizando una profundidad más chica. Notar que esta vez, decidimos utilizar un set de datos en el cual no tenemos la misma cantidad de postulaciones como no postulaciones. En cambio, utilizamos 6 Millones de postulaciones y 2 Millones de no postulaciones.

Utilizando mapeo simple y normalizando valores:

Algoritmo	Cantidad de datos	Hiperparametros	Score Entrenamiento	Score Kaggle
Decision Tree	8 Millones (2M no postulaciones)	simple map valores normalizados, max_depth=5		0,49519
Decision Tree	8 Millones (2M no postulaciones)	simple map valores normalizados, max_depth=6		0,49477

Observaciones: Al utilizar los datos normalizados el score obtenido descendió en 0.1%, por lo que al no conseguir mejoras no lo implementamos más.

3.4 - Gaussian Naive Bayes:

Es un algoritmo basado en la regla de bayes. A pesar de que se lo conoce por no ser el “mejor” algoritmo, a nosotros nos dio muy buenos resultados, de hecho, los mejores.
A continuación presentamos la tabla que se creó con las diferentes formas en las que probamos este algoritmo. Tener en cuenta que a lo largo de las pruebas fuimos cambiando la forma de probarlo, es decir fuimos cambiando el set de datos, agregando y sacando features, además de cambiar ciertos tipos de mapeos, especialmente a la hora de mapear la edad. En las siguientes tablas se ven especificadas las características que tuvo cada prueba y los resultados obtenidos. Utilizando el set de datos junto con un feature agregado, most frequent word(MFW), el cual contiene las palabras con más frecuencia en cada descripción.

Algoritmo	Cantidad de datos	Hiperara-metros	MFW	Set de Datos	Score entrenamiento	Score Kaggle
Gaussian Naive Bayes	2 Millones	Default	Si	simple map	0,5439250229	0,55297
Gaussian Naive Bayes	4 Millones	Default	Si	simple map	0,5513589209	0,55297
Gaussian Naive Bayes	4 Millones	Default	Si	simple map **sin** ESTADO,SEXO	0,5514192057	0,66720
Gaussian Naive Bayes	4 Millones	Default	Si	simple map **sin** ESTADO,SEXO, ZONA	0,5516980232	0,68077
Gaussian Naive Bayes	4 Millones	Default	SI	simple map **sin** ESTADO,SEXO, ZONA,ÁREA	0,5406633845	0,72305
Gaussian Naive Bayes	4 Millones	Default	Si	simple map, **sin** ESTADO,SEXO, ZONA,AREA.EDAD	0,5153048153	0,50307
Gaussian Naive Bayes	4 Millones	Default	Si	simple map, **sin** ESTADO,SEXO ,ZONA,ÁREA. NIVEL_Lab	0,5353897164	0,74414
Gaussian Naive Bayes	4 Millones	Default	Si	simple map, **sin** ESTADO,SEXO, ZONA,ÁREA. NIVEL_Lab, Nombre_sort	0,5359611665	0,50307

Observaciones: En esta oportunidad probamos el algoritmo variando la cantidad de features utilizadas, concluyendo que al sacar las columnas “estado”,”sexo”,”zona”,”área” y “nivel_lab” tenemos una mejor predicción. Por lo que deducimos que pueden estar generando ruido a la hora de entrenar nuestro algoritmo. A partir de esto, empezamos a entrenar con una menor cantidad de features.

Sin sacar features:

Algoritmo	Cantidad de datos	Hiperpara-metros	MFW	Set de Datos	Score entrenamiento	Score Kaggle
Gaussian Naive Bayes	9 Millones	Default	No	simple map valores normalizados	0,745395661	0,50343
Gaussian Naive Bayes	3.5 Millones	Default	No	One Hot Encoding	0,5609228571	0,54349
Gaussian Naive Bayes	6 Millones	Default	No	simple map	0,9999924724	0,54349

Observaciones: En este caso tuvimos en cuenta todos los features del set de datos, sin incluir las descripciones. Notamos que al probar distintas formas de mapeo, es decir encoding numérico(simple mapping), normalizar los valores luego del mapeo y One Hot Encoding, los resultados no eran del nivel esperado, de hecho obtuvimos porcentajes menores que en el caso anterior. Por lo que claramente veíamos que nuestro algoritmo funcionaba mejor con menos cantidad de features.

Sin MFW y variando los features:

Algoritmo	Cantidad de datos	Hiperpara -metros	MFW	Set de Datos	Score entrenamiento	Score Kaggle
Gaussian Naive Bayes	4 Millones	Default	No	simple map, **sin** ESTADO,SEXO, ZONA,ÁREA. NIVEL_Lab	0,5380560649	0,78381
Gaussian Naive Bayes v8	6 Millones	Default	No	simple map, **SOLO** vistas, edad y tipoDeTrabajo	0,99982376	0,93006
Gaussian Naive Bayes v9	6 Millones	Default	No	simple map, **SOLO** vistas, edad y nivelLaboral	0,99982376	0,92005

Gaussian Naive Bayes v10	6 Millones	Default	No	simple map, **SOLO** vistas	0,99982376	0,81380
Gaussian Naive Bayes v11	6 Millones	Default	No	simple map, **SOLO** vistas, edad, tipoDeTrabajo y nivelLab	0,99982376	0,91862
Gaussian Naive Bayes	6 Millones	Default	No	simple map, **SOLO** vistas,edad y descripciones	0,9998406284	0,63505
Gaussian Naive Bayes	6 Millones	Default	No	simple map (Normalizados), **SOLO** vistas,edad , descripciones y tipo de trabajo	0,9998406284	0,61787
Gaussian Naive Bayes	6 Millones	Default	No	sin map. SOLO edades y vistas	0,9998406284	0,69391
Gaussian Naive Bayes	6 Millones	Default	No	simple map SOLO vistas,edad,sexo y descripciones	0,9998406284	0,62701
Gaussian Naive Bayes	6 Millones	Default	No	simple map agrego cantPostulaciones y cantPostulantes y SOLO agrego edad y cantidad de vistas	0,9898475625	0,95383
Gaussian Naive Bayes	6 Millones	Default	No	simple map agrego cantPostulaciones y cantPostulantes y SACO estado,zona, nombre_sort, descripcion	0,9898475625	0,94485
Gaussian Naive Bayes	6 Millones	Default	No	agrego cantPostulaciones y cantPostulantes y SOLO agrego edad y cantidad de vistas,	0,9905269638	0,95553

				VALORES NORMALIZADOS		

Observaciones: En esta ocasión, decidimos variar los features sin tener en cuenta el MFW y obtuvimos un rango bastante grande de porcentajes, pero al agregar las columnas, “cantVistas”, “cantPostulaciones” y “cantPostulantes” pudimos notar una gran mejora a la hora de predecir, lo que creemos que fue información muy importante para que el algoritmo pueda entrenar de manera más eficiente. Estas columnas nuevas se explicaron anteriormente que son del primer trabajo práctico. “cantVistas” es la cantidad de usuarios que visitaron cada aviso. “cantPostulaciones” es la cantidad de postulaciones que realizó dicho usuario y “cantPostulantes” es la cantidad de postulaciones que obtuvo cada aviso.

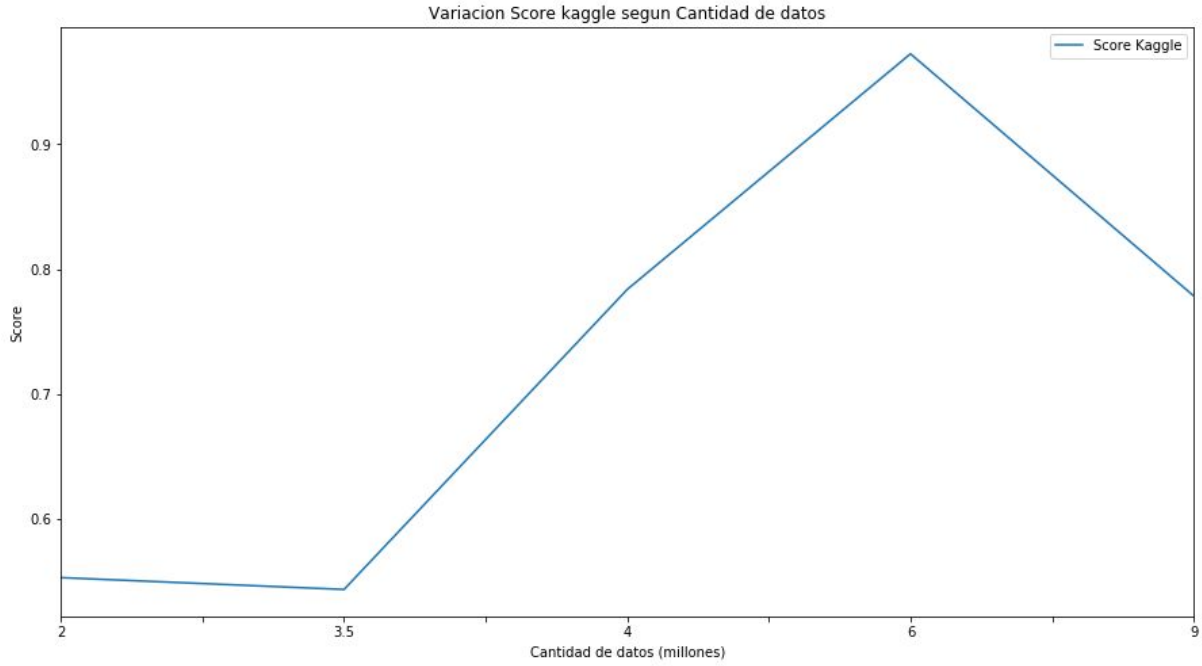
Utilizando solo las columnas “vistas” y “edad”:

Algoritmo	Cantidad de datos	Hiperpara-metros	MFW	Set de Datos	Score entrenamiento	Score Kaggle
Gaussian Naive Bayes	6 Millones	default	No	vistas,edad (map1 de la edad)	0,9998237609	0,92391
Gaussian Naive Bayes	6 Millones	default	No	vistas,edad (map2 de la edad)	0,9998237609	0,81380
Gaussian Naive Bayes	6 Millones	default	No	vistas,edad (map3 de la edad)	0,9998237609	0,92222
Gaussian Naive Bayes	6 Millones	default	No	vistas,edad (map4 de la edad y normalizado)	0,9998237609	0,93696
Gaussian Naive Bayes	6 Millones	default	No	vistas,edad (map4 de la edad)	0,9998237609	0,93696
Gaussian Naive Bayes	6 Millones	default	No	simple map SOLO vistas y edad	0,9998148783	0,94108

Observaciones: Aquí pudimos ver que utilizando sólo dos columnas de todo el dataset hace una gran predicción de las futuras postulaciones. Por lo que

determinamos que son las que tuvieron mayor peso. Luego, variamos las distintas maneras de mapear la edad, agrandando o achicando los rango en cada intervalo, así, generamos más o menos intervalos. Esto último también obtuvo resultados satisfactorios.
Finalmente podemos concluir que este algoritmo obtuvo siempre buenos resultados cuando se lo entrenó con features de cantidades y no categóricos.

En este plot se puede ver la variación del score kaggle según la cantidad de datos utilizada, fueron tomados los mejores resultados de cada cantidad y se puede ver que el score más alto fue con 6 millones de datos.



Concluimos que 6 millones es una cantidad óptima de datos para entrenar el algoritmo, sin incluir demasiado “ruido” (no postulaciones). Por ésto mismo, es con la cantidad de datos que más entrenamos, porque generaba los mejores resultados.

3.5 - MLP Classifier (Red Neuronal):

Multi-layer perceptron classifier es un algoritmo supervisado, basado en redes neuronales, sirve como lo dice el nombre para clasificar y tiene la capacidad de aprender modelos no lineales.

Realizamos varias pruebas con hiperparametros por default:

Algoritmo	Cantidad de datos	Set de datos	Score Entrenamiento	Score Kaggle
MLP Classifier (Red Neuronal)	6 Millones	simple map	0,5843499179	0,75164
MLP Classifier (Red Neuronal)	6 Millones	simple map SOLO vistas tipodeTrabajo nivelLab desc nombre_sort	0,999828326	0,87553
MLP Classifier (Red Neuronal)	6 Millones	simple map	0,999828326	0,78395
MLP Classifier (Red Neuronal)	6 Millones	simple map SOLO vistas edad	0,999828326	0,93368
MLP Classifier (Red Neuronal)	12 Millones	default	0,6247973841	0,49516

Observaciones: Como pudimos observar en otros algoritmos previos como Gaussian Naive Bayes, la cantidad de features es muy importante. Esta vez, los mismos dos features nos volvieron a dar el mejor resultado para esta red neuronal. Además agregándole una mayor cantidad de datos, se puede apreciar cómo empeoró el resultado, comparando la primer fila con la última. A pesar de todo, podemos observar que los scores de kaggle fueron bastante buenos exceptuando el último.

Luego realizamos algunas pruebas variando un hiperparamatro llamado “hidden_layer_sizes” que indica la cantidad de capas y neuronas por capa.

Algoritmo	Cantidad de datos	Set de datos	Score Entrenamiento	Score Kaggle
MLP Classifier (Red Neuronal)	6 Millones	simple map valores normalizados hidden_layer_sizes=(100 ,100)	0,5862514993	0,64358
MLP Classifier (Red Neuronal)	6 Millones	simple map hidden_layer_sizes=(100 ,100)	0,5867092563	0,73273

Observaciones: En este caso, variamos el hidden_layer_sizes mientras que anteriormente lo dejábamos como default. Ahora asignándole un valor nosotros podemos ver que predice un buen resultado sin normalizar. Mientras que normalizando los datos no tanto. Esto es un comportamiento extraño ya que los algoritmos suelen funcionar mejor con los datos normalizados.

3.6 - Light GBM:

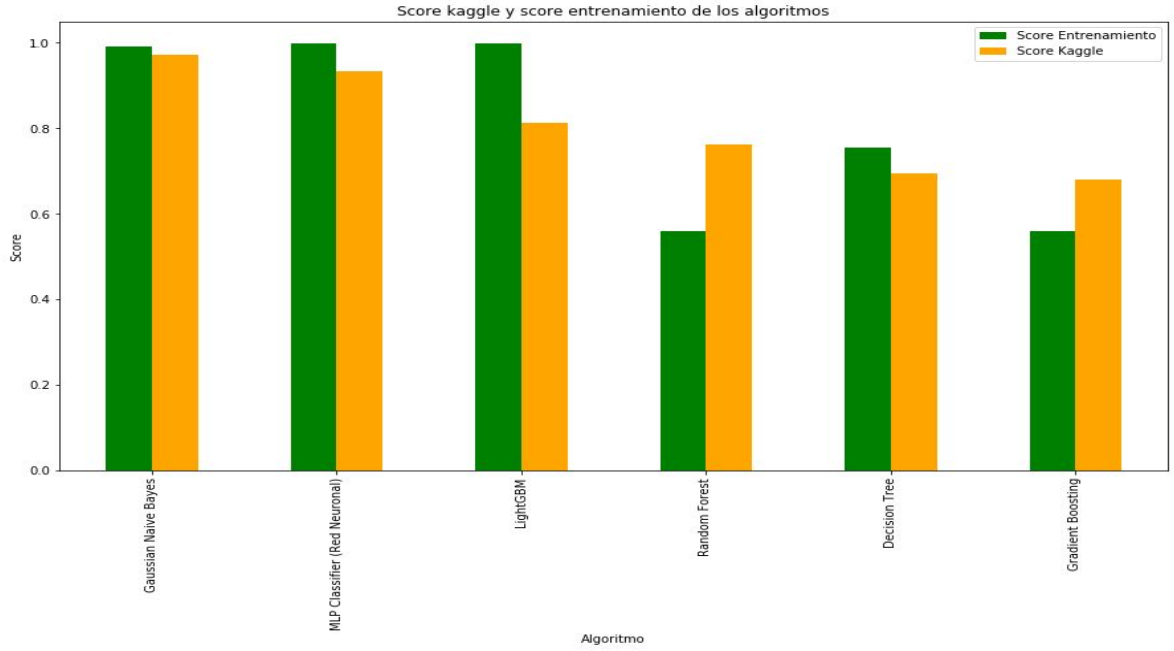
Light gradient boosting es un algoritmo basado en árboles con la diferencia que éstos crecen de manera vertical mientras que en otros algoritmos lo hacen de manera horizontal. Éste fue el único caso donde tuvimos que descargarnos otra libreria para poder utilizarlo.
Realizamos varias pruebas, como se ve en el cuadro. A la mayoría de los hiperparametros los dejamos fijos. Sólo variamos n_stemators que indica la cantidad de árboles que se usan.
Con respecto al set de datos también fuimos variando primero utilizamos un simple map y luego utilizamos features categóricos, que en especial este algoritmo suele tener un buen resultado para estos tipo de datos.

Algoritmo	cantidad de datos	Set de datos	Parámetros	score entrenamiento	score kaggle
LightGBM	6 Millones	simple map SOLO vistas y edad	learning_rate=0.01, objective='binary', num_leaves=7000, max_depth=19, n_estimators=20	0,9998406284	0,8138
LightGBM	6 Millones	categorical features SOLO vistas edad y tipodeTrabajo	learning_rate=0.01, objective='binary', num_leaves=7000, max_depth=19, n_estimators=100	0,9998406284	0,8138
LightGBM	6 Millones	categorical features SOLO vistas edad y tipodeTrabajo	learning_rate=0.01, objective='binary', num_leaves=7000, max_depth=19, n_estimators=50	0,9998406284	0,8138
LightGBM	6 Millones	categorical features SOLO vistas edad y tipodeTrabajo	learning_rate=0.01, objective='binary', num_leaves=7000, max_depth=19, n_estimators=20	0,9998406284	0,8138
LightGBM	6 Millones	categorical features SOLO vistas edad	learning_rate=0.01, objective='binary', num_leaves=1000 ,max_depth=15, n_estimators=20	0,9998406284	0,8138

Observaciones: Aunque parezca mentira, y a pesar de haber variado un hiperparámetro y los datos, el resultado del algoritmo siempre fue el mismo. Es una rareza y no supimos determinar si era un problema de nuestro de cómo lo estábamos utilizando o una mera coincidencia. Por lo que decidimos no continuar entrenando distintas variantes de este algoritmo.

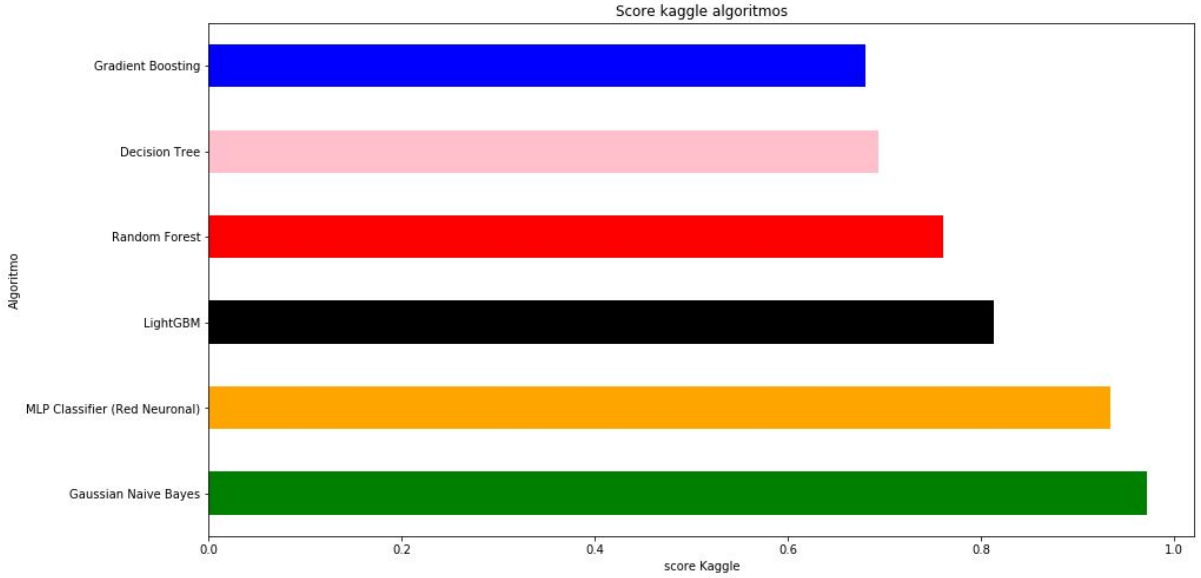
4 - Comparación entre algoritmos:

En el siguiente plot se puede observar los scores, se tomaron los mejores obtenido por cada algoritmo:



Como se puede observar, no hay ninguno que esté haciendo overfitting, ninguno ajusta muy bien al test de entrenamiento y muy mal al test de prueba. También se puede ver que random forest no ajusta muy bien al test de entrenamiento aunque mucho mejor al test de prueba.

En el segundo plot, hemos tomado también el mejor score de cada uno de los algoritmos:



Aquí podemos apreciar que el que mejor resultado nos dio fue “Gaussian naive bayes” con un 0,95, seguido por MLP Classifier con un score de 0,93 y tercero lightGBM con un 0,81 de score.
El peor resultado fue el de Gradient boosting con 0,68 de score.

5 - **Observaciones generales:**

Antes de establecer nuestras conclusiones finales del presente trabajo práctico, debemos realizar algunas observaciones generales que surgieron a lo largo del trayecto:

La medida de error que elegimos para estimar el rendimiento de los algoritmos entrenados fue la función *score*, que se encuentra en la librería de sklearn. Sin embargo, ésta no nos sirvió como indicativo del score que íbamos a obtener en la competencia (Kaggle). Es decir que, si nuestro score subía, no implicaba que el score en Kaggle subiría. Al darnos cuenta tarde no pudimos cambiar la medida de error porque significaba volver a correr todo los algoritmos con las distintas variantes (lo cual implicaba mucho tiempo de trabajo ya hecho). Después de investigar un poco, vimos que teníamos mejores opciones, como por ejemplo, el mean squared error (el error cuadrático medio) o la medida utilizada en la competencia (Area under Receiver Operating Characteristic Curve).

Otra observación a tener en cuenta es que nuestro mejor score (0,97) en la competencia se debe al leaked de datos que hubo; donde algunas (3400 aprox) de las postulaciones que debíamos predecir del set de datos Desde15Abril también estaban incluidas en el set de entrenamiento (Hasta15Abril). Por lo tanto, eran postulaciones que no hacía falta realmente predecir, sino que teníamos la información para determinar si eran o no postulaciones. Como nosotros no

entrenamos con el set de datos completo, esto nos ayudó mucho a mejorar nuestro score en la competencia, a pesar de que no fue obtenido a través del entrenamiento de algún algoritmo (objetivo del trabajo práctico).

6 - **Conclusión:**

Algunas conclusiones que podemos sacar luego de realizar nuestro primer estudio dentro del ámbito de Machine learning son:

Es muy importante la limpieza de los datos y el procesamiento de los mismos; extraer la información que realmente sirva y dejar de lado los datos que sólo hacían ruido.Incluyendo aquí mismo, la cantidad de datos necesaria y la generación de no postulaciones (posible generación de ruido en los datos), en la cual tuvimos que encontrar un balance en la cantidad óptima de datos para predecir correctamente y la proporción de no postulaciones necesarias para entrenar con un set de datos acorde a la situación a predecir; siempre teniendo en cuenta la limitación de recursos que tenemos (poca memoria RAM).

Una buena medida de error es importante para tener un estimativo de cómo va mejorando nuestra predicción. En nuestro caso no pudimos usar realmente ésta estrategia para prevenir resultados muy malos en la competencia. Es importante establecer una medida de error que se ajuste al problema y verificar que funciona de manera acorde, es decir, una disminución del error debería implicar un mejor puntaje en la competencia.

El análisis de datos realizado previo (TP1) fue muy importante a la hora de encontrar las características que mayor influencia generaban en las predicciones; un buen entendimiento de los datos y cómo impactan en la predicción es importante a la hora de elegir sobre qué features elegir.

Finalmente, podemos concluir, que es muy importante probar el modelo ante cualquier duda, uno nunca sabe con certeza qué es lo que puede dar el mejor resultado (puede estimarse según la teoría, pero siempre es conveniente probar). "*No se puede hacer nada contra el éxito*". En nuestro caso particular, el algoritmo Gaussiano de Naive Bayes parecía no ser una buena elección para predecir, pero terminó siendo el algoritmo que mayor rendimiento tuvo; entrenando con poca cantidad de datos y sólo 4 features obtuvimos un muy buen score en la competencia.