

**ENTREGA FINAL
PROYECTO INTELIGENCIA ARTIFICIAL
DEFAULT PREDICTION PROJECT**

**SANTIAGO GARCIA CASTRILLON
JUAN DANIEL TABARES GOEZ**

**PROFESOR:
RAUL RAMOS POLLÁN**



**UNIVERSIDAD[®]
DE ANTIOQUIA**
1 8 0 3

**UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERIA
DEPARTAMENTO DE INGENIERIA MECANICA
MEDELLIN
2022**

1. INTRODUCCIÓN

La inteligencia artificial o Machine Learning es una herramienta que hoy en día nos permite mejorar procesos y actividades, volviendo estas más eficientes de manera que generen más ganancia para la empresa o usuario que hace uso de ellos. Por ello para este proyecto seleccionamos un dataset obtenido de la página Kaggle, la cual ofrece diferentes competencias para sus usuarios. Nuestra selección fue hacia el reto de predecir la probabilidad de que un usuario cualquiera no llegase a pagar el monto del saldo de su tarjeta de crédito en el futuro en función de su historial de pago, deudas y otras variables que se obtenían en el dataset brindado por [American Express - Default Prediction](#). Este dataset será procesado con la intención de generar modelos de mayor score al validarlos, con la intención de que la empresa pueda mejorar su margen neto en al menos un 3%, siendo esta la condición necesaria para poner el modelo en producción.

2. EXPLORACIÓN DESCRIPTIVA DE LOS DATOS

Inicialmente observamos que el dataset tiene una abrumadora cantidad de datos, por lo que se toma una cierta cantidad de datos a procesar y a visualizar. En entregas previas se utilizaron solamente 100.000 datos, para esta entrega se intentó utilizar una mayor cantidad de datos para mejorar el aprendizaje del modelo, pero se tuvieron varios inconvenientes a la hora de subir el dataframe a github, por lo que se continuo con la cantidad de datos previamente establecida.

Agrupamos los datos de train por cliente con su respectiva media para facilitar el manejo de los datos

[+ Code](#) [+ Markdown](#)

Borramos los datos de las fechas para poder tratar los datos por cada cliente; además estas fechas no tienen un orden lógico, por lo tanto no se prestan para ser tratadas como series temporales de Pandas

```
del(train['S_2'])
```

```
data = train.groupby('customer_ID').mean()
data.head()
```

	P_2	D_39	B_1	B_2	R_1	S_3	D_41
customer_ID							
0000099d6bd597052cdca90ffabf56573fe9d7c79be5fbac11a8ed792feb62a	0.933824	0.010704	0.012007	1.005086	0.004509	0.113215	0.005021
00000fd6641609c6ece5454664794f0340ad84ddce9a267a310b5ae68e9d8e5	0.899820	0.215205	0.025654	0.991083	0.006246	0.120578	0.004993
00001b22f846c82c51f6e3958ccd81970162bae8b007e80662ef27519fcc18c1	0.878454	0.004181	0.004386	0.815677	0.006621	NaN	0.006842
000041bdba6ecadd89a52d11886e8eaaec9325906c9723355abb5ca523658edc	0.598969	0.048862	0.059876	0.955264	0.005665	0.247750	0.005490
00007889e4fcd2614b6cbe7f8f3d2e5c728eca32d9eb8ad51ca8b8c4a24cefed	0.891679	0.004644	0.005941	0.814543	0.004180	0.173102	0.005352

Imagen 1. Exploración inicial de los datos.

Inicialmente para una mejor visualización se elimina una columna de fechas que no brindara nada de información; además, se agrupan los datos para cada cliente con la media de estos.

Luego de esta agrupación de los datos, se crea una función para llenar columnas con datos categóricos y para la selección de estas columnas buscamos cuales de ellas tenían mayor cantidad de datos faltantes, debido a que mientras más datos faltantes posea menos información nos brindara y por el contrario generará ruido en el modelo.

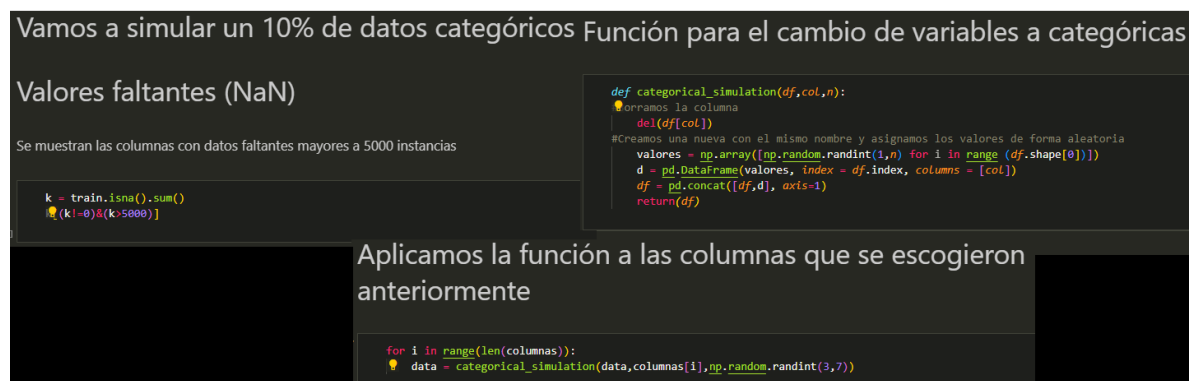


Imagen 2. Simulación de columnas categóricas.

Luego de realizar esta organización anexamos la columna de los targets, esto con el objetivo de poder visualizar las variables que tengan mayor relación con nuestro target que oscila entre 0 y 1, donde el 0 representa el buen pago por parte de los usuarios y el 1 expresa el incumplimiento por parte de los usuarios.

Luego de tener esta limpieza inicial, se utilizó el pairplot con las variables que mayor correlación tienen con nuestra variable objetivo para observar cómo se comportan los datos y si hay una separación clara de los datos, lo cual sería beneficioso para el modelo.

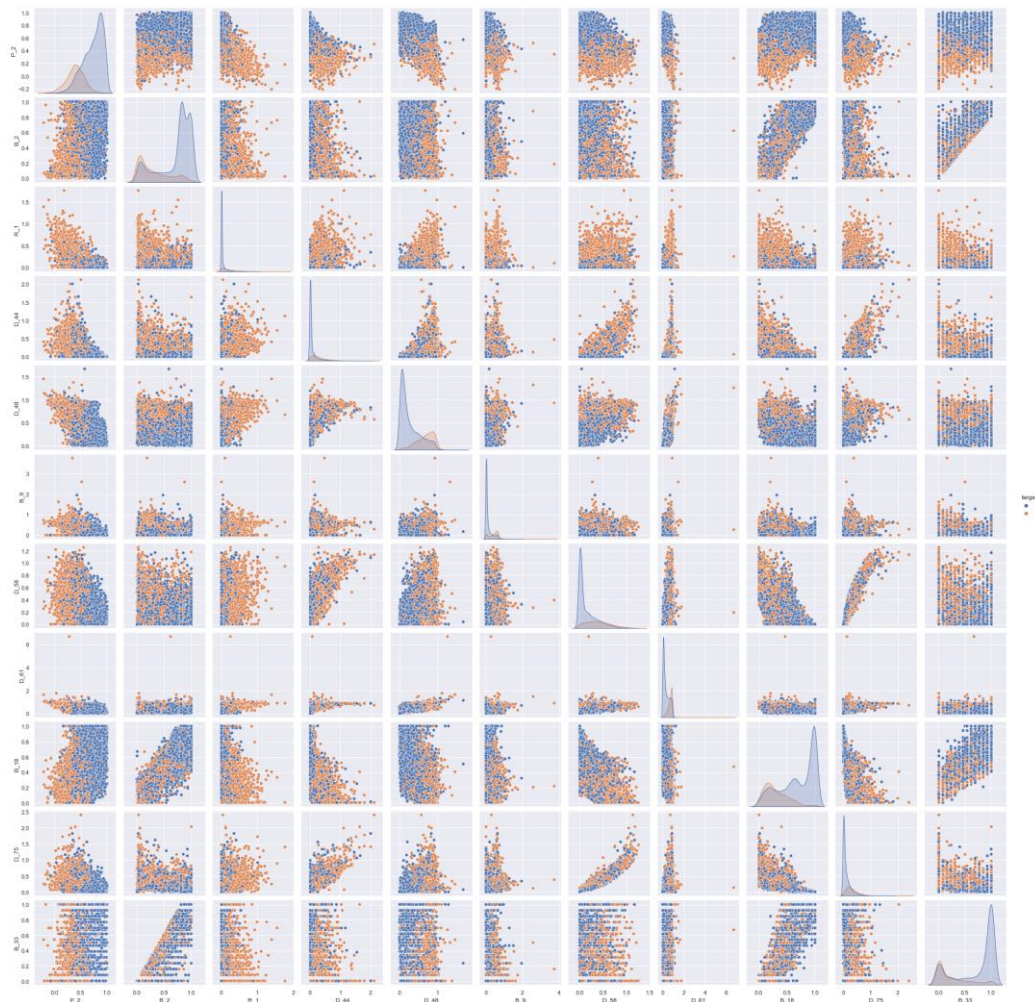


Imagen 3. Pairplot de variables con mayor correlación.

En el pairplot observamos que en realidad no hay ninguna variable que separe por completo los datos, están bastante mezclados, a pesar de ser las variables con mayor correlación. Es importante resaltar que los colores azules corresponden a un target 0 y los puntos naranjados al target 1.

Para una visualización extensa, también se utilizó una matriz de correlación con la intención de buscar si algunas estaban altamente relacionadas entre sí, pero más precisamente para ver cuáles eran esas variables que tenían una mejor relación con nuestro target e identificarlas puntualmente.

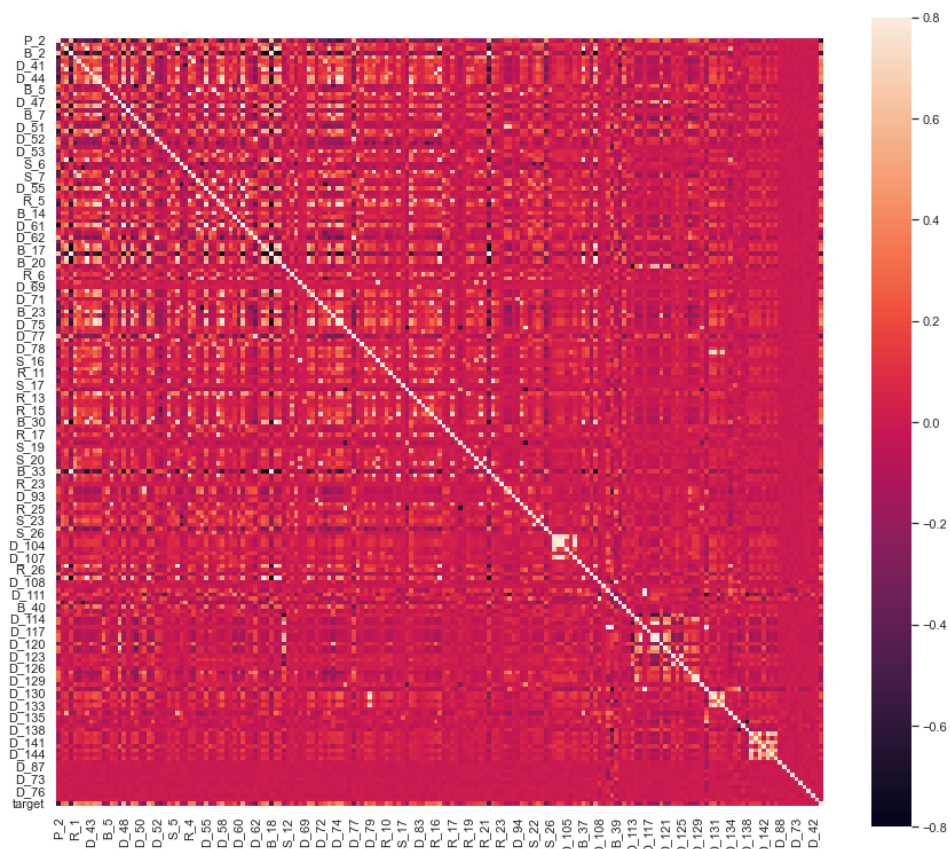


Imagen 4. Matriz de correlación.

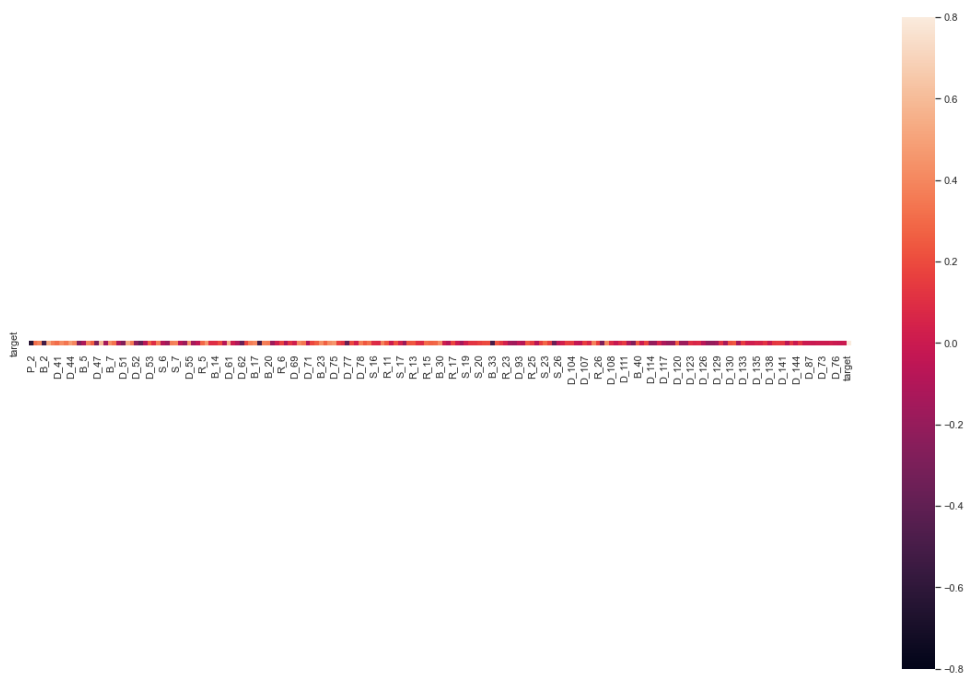



Imagen 5. Correlación del target con las demás variables.

También se realizaron algunas gráficas para visualizar los datos faltantes, aunque estas graficas no brindan mucha información. De igual forma se realizó para seleccionar aquellas columnas que no brindaban nada de información para eliminarlas del dataframe y para ver que columnas rellenaríamos con algún tipo de distribución específica.

Ahora se muestra como en una sección de una columna teníamos varios datos faltantes.


```
datos['S_3'][100:120] 
```

customer_ID	
000f4b8159b2bbe635ace9493931e7ada2ddcb3fdc2f88fbd9f89d463b93b06b	0.133971
000f6da00ca504da14c0079936e8dc9f4971fc4405e45db424848a3015dde525	NaN
000f8675ede66cc6affd4c048db11a00246d7ee623f453aae22d693c4a82f832	NaN
000f8fcfe7ca030fc553a55027ecdd0219fc8e2794c9af713f1ace182fd3064d	0.260371
000fbdd8416278a0be960edf1b06e37eaa94db5c3536fe91eacca2b83119ad5b	0.378042
000ff27e5fa776da26ae9b404699acdb036f79875e466cecfc05765dcab50d447	0.147522
001089806583b378098bd81a88eb877a821202b1dfe257f963835d803c4debf0	0.619490
001089b6fb7c690f753cf4b6f05344e693f1c9248bd642d30e5f7762694380ab	0.170616
0010fc44055df7e9d81c24dcba9b10c16a1d01412c12c053632816a12b1c1a6	0.132495
00115d9ba7650295cdfc73537fe5efad4f3de6838064ce805e5ede8fe772bda5	0.171684
00116df300d0b6762bf4661c53fa499bd4b8915b6a6636012f1000bd30fabac3	0.166842
0011c2e793d6eded8ab7d4c76c14278bfd870ab91918a17fdeba7d287f0051a2	0.247079
0011d1930636c6a57fab755d232cab01d68d838a090ca348b0c5aeaf66e982a3	0.168875
0012251d7f58c0b74b04235aae964e2bd0057f9063c3b43b628ddc4da73dac32	NaN
001248c3a0838fcab80ecc8693ac841053c114b9b1316dd60057e9ebfb992577	0.138996
00125d34dd5d1927e7a37a0824aa449477a596240870938999265535443c6d34	0.077409
0012ba67e6f054122e685070618c45c5cac9730ca60415710a629d8c5f22c58d	0.208795
0012d8a68050cb362ce5ea56889ab5a6f47b93061b2642e88df269a757d257af	0.042295
0012e41fe6caa3ba31b55b3de2030cbb77b01203aeb4a5c6677de5b80f15cebe	NaN
0013037420169086d185f38034d9c49d748867436f88113fa2680651b9c98384	0.494444

Name: S_3, dtype: float64

Imagen 6. Datos con valores faltantes.

```
Función para rellenar los datos numericos faltantes con una distribucion normal
```

```
def fillna_numerics(df,col):  
      
    for pos, i in enumerate(df[col].isna().values):  
        if i == True:  
            df.loc[df.index[pos], col] = np.random.normal(loc = df[col].mean(), scale = df[col].std())  
    return df
```

Python

Imagen 7. Función para rellenar datos numéricos.

En la imagen anterior se muestra la función que se utilizó para rellenar los datos faltantes en las columnas numéricas, de modo que en este punto puede cambiarse la distribución seleccionada de manera fácil para rellenar los datos. En nuestro caso utilizamos una distribución normal para llenar estos datos y ver cómo se comportan los modelos con esta decisión.

```
datos['S_3'][100:120]
```

customer_ID	
000f4b8159b2bbe635ace9493931e7ada2ddcb3fdc2f88fbd9f89d463b93b06b	0.133971
000f6da00ca504da14c0079936e8dc9f4971fc4405e45db424848a3015dde525	0.323597
000f8675ede66cc6affd4c048db11a00246d7ee623f453aae22d693c4a82f832	0.063605
000f8fcfe7ca030fc553a55027ecdd0219fc8e2794c9af713f1ace182fd3064d	0.260371
000fbdd8416278a0be960edf1b06e37eaa94db5c3536fe91eacca2b83119ad5b	0.378042
000ff27e5fa776da26ae9b404699acdb036f79875e466cecf05765dcab50d447	0.147522
001089806583b378098bd81a88eb877a821202b1dfe257f963835d803c4debf0	0.619490
001089b6fb7c690f753cf4b6f05344e693f1c9248bd642d30e5f7762694380ab	0.170616
0010fc44055df7e9d81c24dcb4a9b10c16a1d01412c12c053632816a12b1c1a6	0.132495
00115d9ba7650295cdfc73537fe5efad4f3de6838064ce805e5ede8fe772bda5	0.171684
00116df300d0b6762bf4661c53fa499bd4b8915b6a6636012f1000bd30fabac3	0.166842
0011c2e793d6eded8ab7d4c76c14278bfd870ab91918a17fdeba7d287f0051a2	0.247079
0011d1930636c6a57fab755d232cab01d68d838a090ca348b0c5aeaf66e982a3	0.168875
0012251d7f58c0b74b04235aae964e2bd0057f9063c3b43b628ddc4da73dac32	0.101571
001248c3a0838fcab80ecc8693ac841053c114b9b1316dd60057e9ebfb992577	0.138996
00125d34dd5d1927e7a37a0824aa449477a596240870938999265535443c6d34	0.077409
0012ba67e6f054122e685070618c45c5cac9730ca60415710a629d8c5f22c58d	0.208795
0012d8a68050cb362ce5ea56889ab5a6f47b93061b2642e88df269a757d257af	0.042295
0012e41fe6caa3ba31b55b3de2030cbb77b01203aeb4a5c6677de5b80f15cebe	0.483631
0013037420169086d185f38034d9c49d748867436f88113fa2680651b9c98384	0.494444

```
Name: S_3, dtype: float64
```

Imagen 8. Datos faltantes numéricos llenados.

Ahora observamos el mismo tramo donde se rellenan los datos que anteriormente no estaban de manera coherente. Además, se crea una función para llenar los datos categóricos que falten en las columnas definidas por el dataset como categóricas.

Función para rellenar los datos categoricos faltantes

```
def fillna_categoricals(df,col):  
    for pos, i in enumerate(df[col].isna().values):  
        if i == True:  
            df.loc[df.index[pos], col] = np.random.randint(df[col].max()+1)  
    return df
```

Imagen 9. Función para llenar datos categóricos.

Debido al agrupamiento de los datos por la media para cada cliente, algunos datos categóricos quedaron con valores incoherentes a lo que se tenía, por lo que se creó una función adicional que redondeara estos valores al número entero más cercano.

Función para estandarizar los datos categoricos (redondear valores a enteros)

```
def rounding(df,cols):  
    for i in cols:  
        for pos, item in enumerate(df[i].values):  
            df.loc[df.index[pos], i] = df.loc[df.index[pos], i].round()
```

Imagen 10. Función para redondear datos.

Estos datos categóricos, según lo aprendido deben separarse, mediante el llamado “onehot encoding” para ello es necesario renombrar las columnas, de manera que las nuevas columnas creadas tengan información de la variable de donde fueron tomadas. Para ello usamos la siguiente función.

Función para renombrar los datos de las columnas categoricas para posteriormente realizar el one hot encoding

```
def num_to_categoricals(df, cols):  
    for i in cols:  
        for pos, item in enumerate(df[i].values):  
            df.loc[df.index[pos], i] = i+'_'+str(int(df.loc[df.index[pos], i]))
```

Imagen 11. Función para renombrar los datos categóricos.

Luego de aplicar el onehot encoding y ver el resultado, observamos que apareció duplicado el nombre de la columna por lo que se creó luego una función para organizar este pequeño error.

Se separan los valores de las variables categoricas en columnas diferentes

```
datos = pd.get_dummies(datos)
datos.head(20)
```

D_76_D_76_3	D_42_D_42_1	D_42_D_42_2	D_42_D_42_3	D_42_D_42_4	D_82_D_82_1	D_82_D_82_2	D_82_D_82_3	D_82_D_82_4
1	0	0	1	0	1	0	0	0
0	0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1
0	0	0	1	0	1	0	0	0

Imagen 12. Onehot encoding.

Funciones necesarias para renombrar adecuadamente las columnas categóricas

```
def cutting(nombre_col):
    contador = 0
    for index,item in enumerate(nombre_col):
        if (item == '_'):
            contador += 1
        if contador == 2:
            return(nombre_col[index+1:])
```

```
def rename_cols(df):
    for item in df.columns:
        if len(item)>6:
            df = df.rename(columns={item:cutting(item)})
    return(df)
```

Imagen 13. Renombramiento de las columnas categóricas.

Posterior al onehot encoding se creó una función para normalizar todos los datos que no fueran categóricos, teniendo valores mínimos de 0 y máximos de 1.

```

def normalization(df, columns, categorics):
    for col in columns:
        if np.array([True if e == col else False for i, e in enumerate(categorics)]).any() == True:
            continue
        else:
            min = df[col].min()
            max = df[col].max()
            s_i = [(i-min)/(max-min)) for i in df[col]]
            sol = pd.DataFrame({col: s_i, index=df.index})
            df.update(sol)
    return df

```

Imagen 14. Normalización de los datos de 0 a 1.

Luego de todo este tratamiento de los datos se generó un nuevo archivo .csv para ejecutarlo en distintos notebooks y no tener que correr este código continuamente.

3. DESARROLLO.

Para comenzar con el desarrollo de los modelos usamos el dataframe previamente organizado cargado en un notebook diferente donde se separan los datos para 70% para entrenamiento y 30% para validación

Train and Test Split Dataset

```

[ ] 1 X = data.drop(['target'], axis=1)
    2 y = data['target']
    3
    4 test_size = 0.3
    5
    6 Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=test_size, random_state=42)

```

Imagen 15. Partición de los datos.

Teniendo el dataframe seccionado y listo para el entrenamiento de los modelos, se continua con la selección de uno de estos.

```
Random Forest Classifier

Cross Validation for Random Forest

[ ] 1 estimator_1 = RandomForestClassifier(n_estimators=5, max_depth=10)

[ ] 1 cross_val = cross_validate(estimator_1, Xtv, ytv, return_train_score=True, return_estimator=True,
2   | | | | | cv=ShuffleSplit(n_splits=10, test_size=val_size))

1 def report_cv_score(est):
2   | print ("validation score   %.3f (±%.4f) with %d splits"%(np.mean(est["test_score"]), np.std(est["test_score"]), len(est["test_score"])))
3   | print ("train score       %.3f (±%.4f) with %d splits"%(np.mean(est["train_score"]), np.std(est["train_score"]), len(est["train_score"])))
+ Código + Texto

[ ] 1 report_cv_score(cross_val)

validation score   0.847 (±0.0062) with 10 splits
train score       0.964 (±0.0045) with 10 splits
```

Imagen 16. Random Forest Classifier.

Con el Random Forest Classifier obtuvimos unos buenos resultados iniciales de precisión, pero para corroborar esto se realizarán las curvas de aprendizaje que nos ayudarán de una mejor manera a determinar el valor de nuestro hiperparametro.

```
Function to graph learning curves

[ ] 1 def plot_learning_curve(estimator,X,y,cv):
2   | from sklearn.model_selection import learning_curve
3   | train_sizes, train_scores, test_scores = learning_curve(estimator_1, X, y, cv=cv, shuffle=True)
4   | train_scores_mean = np.mean(train_scores, axis=1)
5   | train_scores_std = np.std(train_scores, axis=1)
6   | test_scores_mean = np.mean(test_scores, axis=1)
7   | test_scores_std = np.std(test_scores, axis=1)
8   | # Plot learning curve
9   | plt.grid()
10  | plt.fill_between(
11  | | | train_sizes,
12  | | | train_scores_mean - train_scores_std,
13  | | | train_scores_mean + train_scores_std,
14  | | | alpha=0.1,
15  | | | color="r",
16  | | )
17  | plt.fill_between(
18  | | | train_sizes,
19  | | | test_scores_mean - test_scores_std,
20  | | | test_scores_mean + test_scores_std,
21  | | | alpha=0.1,
22  | | | color="g",
23  | | )
24  | plt.plot(train_sizes, train_scores_mean, "o-", color="r", label="train score")
25  | plt.plot(train_sizes, test_scores_mean, "o-", color="g", label="test score")
26  | plt.legend(loc="best")
27  | plt.ylabel("Score")
28  | plt.xlabel("Training examples")
29  | plt.title(estimator.__class__.__name__)
30  | return plt.show()
```

Imagen 17. Función para curvas de aprendizaje.

Para elaborar estas curvas de aprendizaje realizamos una función para poder hacerlo de manera más sencilla y ágil, teniendo en cuenta que se harán múltiples gráficas para llegar a un valor idóneo.

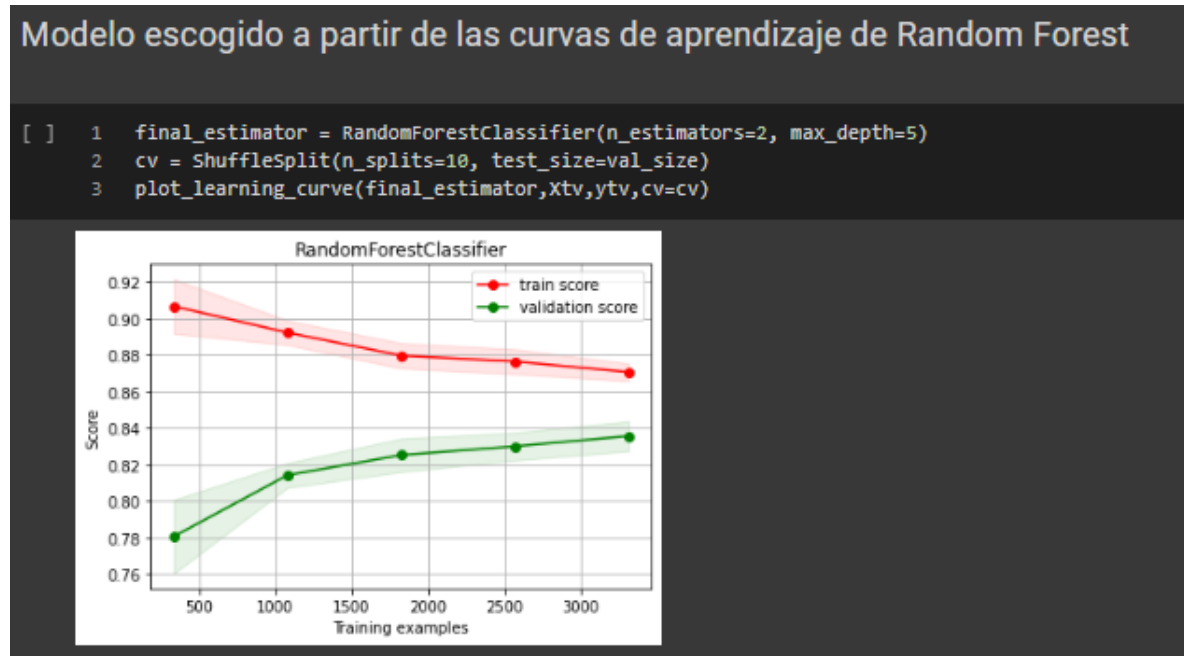


Imagen 18. Hiperparametros seleccionados con las curvas de aprendizaje.

Luego de realizar varias gráficas y analizar las mismas, oscilando entre curvas de aprendizaje sesgadas y otras con overfitting, llegamos a la selección de los hiperparametros mostrados en la imagen anterior, que, aunque no representa una total perfección del modelo, fue el que menos sesgo presento en los resultados.

Luego de tener estos resultados se decide aplicar el PCA, en búsqueda de una mejora; sin embargo, por el contrario, se encuentra una reducción en la precisión del modelo.

Random Forest with PCA

```
[ ] 1  xtr_v, xts_v, ytr_v, yts_v = train_test_split(xtv,ytv,test_size=val_size)

[ ] 1  dt = final_estimator
    2  cs = range(10,200,5)
    3  dtr, dts = [], []
    4  for n_components in cs:
    5      print(".", end=" ")
    6      pca = PCA(n_components=n_components)
    7      pca.fit(xtr_v)
    8
    9      xt_tr = pca.transform(xtr_v)
   10      xt_ts = pca.transform(xts_v)
   11
   12      dt.fit(xt_tr,ytr_v)
   13      ypreds_tr = dt.predict(xt_tr)
   14      ypreds_ts = dt.predict(xt_ts)
   15      ypreds_tr.shape, ypreds_ts.shape
   16      dtr.append(np.mean(ytr_v==ypreds_tr))
   17      dts.append(np.mean(yts_v==ypreds_ts))
   18
```

Imagen 19. PCA aplicado al Random Forest.

También tenemos la siguiente grafica que nos hace evidenciar la cantidad de ruido que aun poseen los datos o la incorrecta selección del modelo.

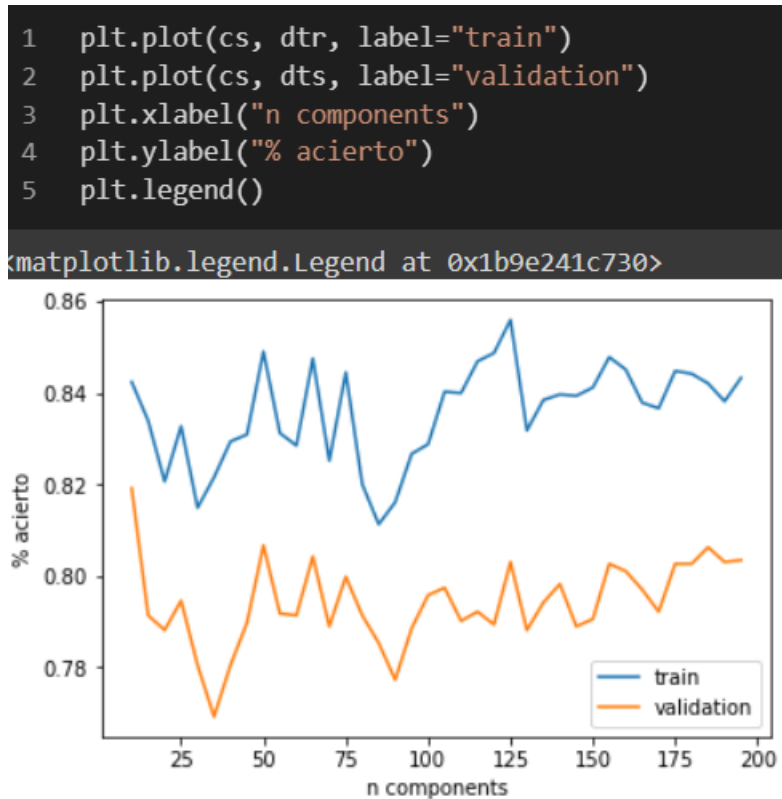


Imagen 20. Acierto vs N° de componentes.

```

1 estimator_2 = Pipeline(((("pca", PCA(n_components=best_cs)), ("estimator", final_estimator)))

1 cross_val = cross_validate(estimator_2, Xtv, ytv, return_train_score=True, return_estimator=True,
2 | | | | | cv=ShuffleSplit(n_splits=10, test_size=val_size))

1 report_cv_score(cross_val)

```

validation score 0.808 (±0.0080) with 10 splits
train score 0.842 (±0.0030) with 10 splits

Imagen 21. Precisión del Random Forest con PCA.

Se observa con claridad la disminución en el score tanto de validación como de entrenamiento.

Ahora comparando ambos modelos para una selección final tenemos lo siguiente.

Model selection

```
1  zscores = []
2  estimators = [final_estimator, estimator_2]
3  for estimator in estimators:
4      print("---")
5      z = cross_validate(estimator, Xtv, ytv, return_train_score=True, return_estimator=False,
6                          cv=ShuffleSplit(n_splits=10, test_size=val_size))
7      report_cv_score(z)
8      zscores.append(np.mean(z["test_score"]))
9  best = np.argmax(zscores)
10 print ("selecting ", best)
11 best_estimator = estimators[best]
12 print ("\nselected model")
13 print (best_estimator)
```

```
--
validation score    0.867 (±0.0047) with 10 splits
train score        0.900 (±0.0044) with 10 splits
--
validation score    0.809 (±0.0102) with 10 splits
train score        0.839 (±0.0065) with 10 splits
selecting 0

selected model
RandomForestClassifier(max_depth=5, n_estimators=20)
```

Imagen 22. Selección del modelo.

Con todo lo anterior claro, procedemos a calcular la métrica establecida en Kaggle con el código que se muestra a continuación.

American Express Metric

```
1 def amex_metric(y_true: pd.DataFrame, y_pred: pd.DataFrame) -> float:
2
3     def top_four_percent_captured(y_true: pd.DataFrame, y_pred: pd.DataFrame) -> float:
4         df = (pd.concat([y_true, y_pred], axis='columns')
5             .sort_values('prediction', ascending=False))
6         df['weight'] = df['target'].apply(lambda x: 20 if x==0 else 1)
7         four_pct_cutoff = int(0.04 * df['weight'].sum())
8         df['weight_cumsum'] = df['weight'].cumsum()
9         df_cutoff = df.loc[df['weight_cumsum'] <= four_pct_cutoff]
10        return (df_cutoff['target'] == 1).sum() / (df['target'] == 1).sum()
11
12    def weighted_gini(y_true: pd.DataFrame, y_pred: pd.DataFrame) -> float:
13        df = (pd.concat([y_true, y_pred], axis='columns')
14            .sort_values('prediction', ascending=False))
15        df['weight'] = df['target'].apply(lambda x: 20 if x==0 else 1)
16        df['random'] = (df['weight'] / df['weight'].sum()).cumsum()
17        total_pos = (df['target'] * df['weight']).sum()
18        df['cum_pos_found'] = (df['target'] * df['weight']).cumsum()
19        df['lorentz'] = df['cum_pos_found'] / total_pos
20        df['gini'] = (df['lorentz'] - df['random']) * df['weight']
21        return df['gini'].sum()
22
23    def normalized_weighted_gini(y_true: pd.DataFrame, y_pred: pd.DataFrame) -> float:
24        y_true_pred = y_true.rename(columns={'target': 'prediction'})
25        return weighted_gini(y_true, y_pred) / weighted_gini(y_true, y_true_pred)
26
27    g = normalized_weighted_gini(y_true, y_pred)
28    d = top_four_percent_captured(y_true, y_pred)
29
30    return 0.5 * (g + d)
```

Imagen 23. Métrica de desempeño de American Express.

Con este modelo al final obtenemos los siguientes resultados para la métrica de desempeño.

```
1 predicts = best_estimator.predict_proba(X)
2
3 prob = np.array([predicts[item][1] for item in range(len(predicts))])
4
5 data['prediction'] = prob
6
7 df_pred = pd.DataFrame(data['prediction'])
8
9 df_y = pd.DataFrame(y)
10
11 print('La métrica evaluada por American Express es %.3f ' % amex_metric(df_y, df_pred))
La métrica evaluada por American Express es 0.725
```

Imagen 24. Evaluación de la métrica de desempeño.

Esta métrica de desempeño compara evalúa la probabilidad de acierto, por lo cual se convierte en un problema de regresión y no de clasificación como se cree en un principio.

Ahora compararemos estos resultados con otro modelo seleccionado, que para este caso será el LogisticRegression, partiendo el dataframe de la misma manera que se realizó previamente y obteniendo lo que se muestra a continuación.

```
Logistic Regression

Cross Validation for Logistic Regression

[ ] 1 estimator = LogisticRegression(max_iter=500)

[ ] 1 cross_val = cross_validate(estimator, Xtv, ytv, return_train_score=True, return_estimator=True,
2   | | | | | cv=ShuffleSplit(n_splits=10, test_size=val_size))

[ ] 1 def report_cv_score(est):
2   |   print("validation score   %.3f (±%.4f) with %d splits"%(np.mean(est["test_score"]), np.std(est["test_score"]), len(est["test_score"])))
3   |   print("train score       %.3f (±%.4f) with %d splits"%(np.mean(est["train_score"]), np.std(est["train_score"]), len(est["train_score"])))

[ ] 1 report_cv_score(cross_val)

validation score   0.872 (±0.0065) with 10 splits
train score       0.893 (±0.0044) with 10 splits
```

Imagen 25. Precisión Logistic Regression.

En una primera instancia se alcanza a apreciar una ligera mejora en el score tanto en entrenamiento como en validación, pero se seguirá con el protocolo para determinar cuál se acomoda mejor a los datos.

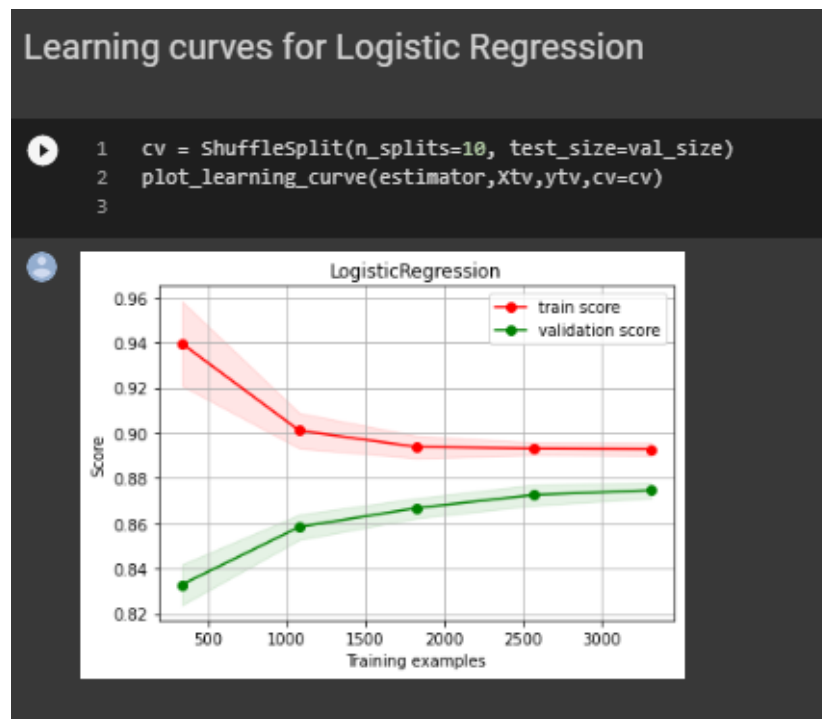


Imagen 26. Curva de aprendizaje para la Logistic Regression.

En cuanto a las curvas de aprendizaje al no poseer hiperparametros solo obtuvimos la siguiente, donde se ve un pequeño caso de sesgo, pero no se tiene la oportunidad de aumentar la cantidad de datos por los problemas que se presentan para cargarlos desde la nube y desde el mismo github.

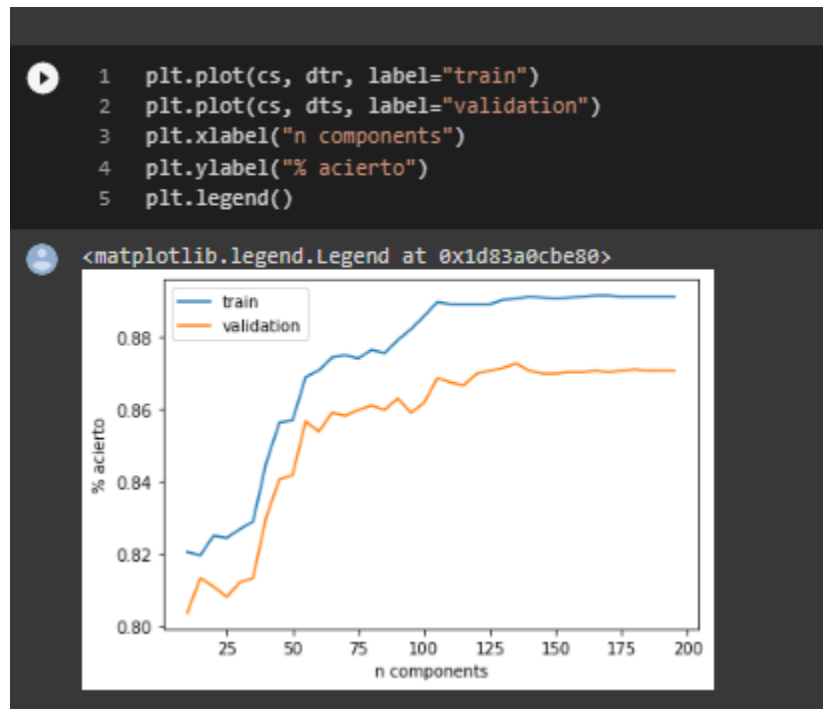


Imagen 27. Acierto vs N° de componentes.

En la gráfica anterior se aplica PCA y se observa un mejor comportamiento de los datos a diferencia del modelo anterior, lo que nos indica que este modelo puede acomodarse mejor a los datos.

Model selection

```
1  zscores = []
2  estimators = [estimator, estimator_2]
3  for estimator in estimators:
4      print("--")
5      z = cross_validate(estimator, Xtv, ytv, return_train_score=True, return_estimator=False,
6                          cv=ShuffleSplit(n_splits=10, test_size=val_size))
7      report_cv_score(z)
8      zscores.append(np.mean(z["test_score"]))
9  best = np.argmax(zscores)
10 print ("selecting ", best)
11 best_estimator = estimators[best]
12 print ("\nselected model")
13 print (best_estimator)

--
validation score    0.873 (±0.0058) with 10 splits
train score        0.894 (±0.0053) with 10 splits
--
validation score    0.872 (±0.0070) with 10 splits
train score        0.892 (±0.0029) with 10 splits
selecting  0

selected model
LogisticRegression(max_iter=500)
```

Imagen 28. Selección del modelo.

De igual manera que se realizó para el modelo anterior, este se compara a si mismo con su versión aplicada con PCA, para seleccionar el que obtenga un mejor score, el cual es el modelo sin PCA.

También se utilizó la métrica de desempeño para este modelo obteniendo los siguientes resultados.

```
1  predicts = best_estimator.predict_proba(X)

1  prob = np.array([predicts[item][1] for item in range(len(predicts))])

1  data['prediction'] = prob

1  df_pred = pd.DataFrame(data['prediction'])

1  df_y = pd.DataFrame(y)

1  print('La métrica evaluada por American Express es %.3f ' % amex_metric(df_y, df_pred))

La métrica evaluada por American Express es 0.732
```

Imagen 29. Métrica de desempeño de AMEX para logistic regression.

4. RETOS Y CONSIDERACIONES DE DESPLIEGUE DEL MODELO

El principal reto para el despliegue inicial es la gestión de los datos brindados, como se muestra a lo largo de este informe se tiene un tratado bastante específico para los datos, por lo cual si no se poseen todos los datos que se necesitan del cliente, el modelo podrá reducir aún más su precisión.

Uno de los mayores retos es la constante evaluación del desempeño del modelo dadas las variaciones y cambios que se pueden producir en los nuevos datos. Siempre se recomienda realizar este monitoreo evaluando el desempeño del modelo con porciones de datos nuevos, los cuales tengan el dato de la variable objetivo para calcular la nueva precisión.

La cantidad de datos con la que se realiza el monitoreo es en teoría pequeña, dependiendo del contexto operacional de cada empresa, analizando las posibles pérdidas o ganancias que pueda obtener de ello.

También se debe tener en cuenta la variabilidad de los datos, debido a que estos cambios pueden afectar directamente la precisión y en última instancia la métrica de desempeño.

5. CONCLUSIONES

- El modelo que mayor precisión nos brinda y más se acomoda a los datos entre los evaluados en el presente informe es LogisticRegression sin PCA, obteniendo una precisión por encima del 80% y una evaluación de la métrica de desempeño del 73% lo cual es bastante positivo.
- Cuando no se tiene una buena calidad de los datos, los métodos no supervisados “entorpecen” la precisión del modelo, provocando un mayor ruido y reducciones en la métrica de desempeño, por lo que no se recomienda para este caso.
- Para mejorar el desempeño obtenido se recomienda entrenar el modelo con una mayor cantidad de datos y así evitar el sesgo.
- A la hora de utilizar modelos más complejos por la poca cantidad de datos con las que se estaba entrenando, teníamos un grave problema de overfitting, por lo que se recomienda modelos sencillos cuando la cantidad de datos es mínima.