Examen midterm temas selectos de estadística II

Rincon Morales David 418080007

January 11, 2021

Contents

1 Análisis exploratorio de datos. 3

2 Ingeniería de datos. 6

3 Modelo supervisado. 8
- 3.1 Tabla analítica de datos (TAD). . . . 8
- 3.2 Valores ausentes e imputación 8
- 3.3 Valores extremos. . . . 9
- 3.4 Modelación. . . . 10
- 3.5 Corriendo los chunks 12
  - 3.5.1 Predecir con todos los modelos 12
- 3.6 Persistencia 12

4 Modelo no supervisado. 13
- 4.1 Lectura de datos y limpieza. . . . 13
- 4.2 Visualización. . . . 13
  - 4.2.1 PCA 14
  - 4.2.2 MDS 14
  - 4.2.3 T-SNE 15
- 4.3 Número de clusters 16
  - 4.3.1 Aglomerativo 16
  - 4.3.2 K-Means 17
  - 4.3.3 Gaussian Mixture 18
  - 4.3.4 Visualización 2D T-SNE y Gauss 20
- 4.4 Perfilamiento. . . . 20
- 4.5 Ejemplos. . . . 20

5 Estrategia de aplicación de los modelos. 22
- 5.1 Modelo supervisado: 22
- 5.2 Modelo no supervisado: 22
- 5.3 Nota. . . . 22

6 Conclusiones. 23

7 Anexos. 23

1 Análisis exploratorio de datos.

Al ser una base de datos sumamente extensa se tuvieron problemas de RAM por falta de memoria,a lo cual para poder hacer el tratamiento y análisis exploratorio de los datos, se hizo una división por "chunks" con 1M (millón) de registros en cada uno. En el caso del modelo supervisado, sólo se elijieron variables que nos ayudaran a predecir el volumen de rides por hora (pickup), las cuales son: Lectura de datos

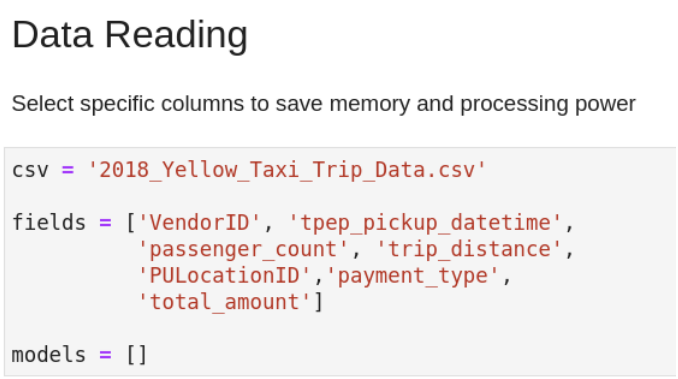


Figure 1: Lectura de datos

Una vez que ya tenemos nuestras variables, se leyó el archivo "csv", asignando los tipos de datos correspondientes para así poder evitar mayor uso de memoria RAM.

Como estamos usando el dataset por "chunk", elegimos utilizar el primero para así poder analizar y procesar la información. Revisamos si no tenemos datos faltantes: Datos faltantes

Figure 2: Check for null variables

No tenemos datos faltantes, por lo que se procede a limpiar información que no necesitamos. Al manejar datasets por particiones, debemos hacer todos los cálculos por funciones, por lo que nuestra primer función es: data_filter

[4]:
```python
def data_filter(df):
    ''' data_filter

    Filters data where:
        -Data is not from 2018
        -Total amount is less than zero
        -Passenger count is less than one
        -Trip distance is less or equal to zero

        -Drop passenger_count (we dont use it later in the model)

    Parameter
    ---------
    df: Dataframe chunk
    '''

    #Filters data not from 2018
    start_date = pd.to_datetime('2018-01-01 00:00:00')
    end_date = pd.to_datetime('2019-01-01 00:00:00')
    mask = (df['tpep_pickup_datetime'] >= start_date) & (df['tpep_pickup_datetime'] < end_date)
    df = df.loc[mask]

    #Filters data where total amount is less than zero
    df  = df[df["total_amount"] > 0.00]

    #Filters where passenger count is less than one
    df = df[df["passenger_count"] >= 1]

    #Filters where trip distance = 0
    df = df[df["trip_distance"] > 0]

    df.drop(['passenger_count'], axis=1, inplace = True)

    return df
```

Figure 3: Data filter function

Luego pasamos por otra función llamada "dates", la cual hace split de la columna "tpep_pickup_time" y la convierte en dos columnas; además, agrega "pickup_hour" para posteriormente hacer un catálogo de horas. Finalmente, corremos nuestro chunk de datos por una función llamada "hours_catalog" la cual nos regresa la columna "id_hour". hours catalog

Hours Catalog

[6]:
```python
def hours_catalog(df):
    '''hours_catalog

    Creates id_hour (int16) column based on pickup_hour sorted column

    Parameter
    --------
    df: Chunk of Dataframe that contains pickup_hour values:
        First two values: Month number
        Third and Fourth values: Day of the week {01: Monday .... 07:Sunday}
        Last two values: Hour

    '''
    catfh = df[['pickup_hour']].drop_duplicates().sort_values('pickup_hour',ascending=True).reset_index(drop=True)
    catfh["id_hour"] = (catfh.index+1)
    catfh["id_hour"] = catfh["id_hour"].astype('int16')

    df = df.merge(catfh, on='pickup_hour', how='inner')
    df.drop('pickup_hour', axis=1, inplace=True)

    return df
```

Figure 4: Función para catálogo de horas

2 Ingeniería de datos.

Una vez que ya pasamos por el proceso anterior y creamos nuestro catálogo de horas "id_hour", obtenemos el primer y último valor de esa columna para poder definir hora_inicial y hora_final. Luego definimos nuestra ventana de observación, la ventana de desempeño y nuestras anclas.

Además, definimos nuestra unidad muestral, nuestro step y nuestras variables continuas y discretas. variables

Data Engineering

[17]:
```
horai,horaf = df[['id_hour']].describe().T[['min','max']].values[0].tolist()
horai,horaf
```

[17]:
```
(1.0, 134.0)
```

[18]:
```
vobs = 24
vdes = 1
anclai = int(horai)+vobs-1
anclaf = int(horaf)-vdes
anclai,anclaf
```

[18]:
```
(24, 133)
```

[19]:
```
um = ['PULocationID', 'ancla']
ancla = 24
step = 4
varc = ['trip_distance', 'total_amount', 'n']
vard = ['hora']
```

Figure 5: Horas, Anclas y Steps

Posteriormente, pasamos a una función donde nos regrese un dataframe compuesto por todas variedades de nuestras variables, por ejemplo: trans

[20]:
```
%%time
X_test = pd.concat(map(lambda ancla:reduce(lambda x,y:pd.merge(x,y,on=um,how='outer'),
       map(lambda k:trans(df,ancla,k),range(step,vobs+step,step))),range(anclai,anclaf+1)),ignore_index=True)
```

```
CPU times: user 4min 58s, sys: 2.82 s, total: 5min 1s
Wall time: 5min 1s
```

[21]:
```
X_test.head()
```

[21]:

	PULocationID	v_hora_min_n_3_4	v_hora_min_n_total_hora_4	v_hora_min_total_amount_3_4	v_hora_min_total_amount_total_hora_4	v_ho
0	1	1.0	1.0	105.30	105.30	
1	4	1.0	1.0	5.80	5.80	
2	7	1.0	1.0	4.80	4.80	
3	10	1.0	1.0	35.38	35.38	
4	12	1.0	1.0	7.55	7.55	

Figure 6: Función trans

Después de tener X listo, creamos Y para luego proceder a la modelación: y

[22]:
```
y_test= pd.concat(map(lambda ancla: target(df,ancla,vdes),range(anclai,anclaf+1)),ignore_index=True)
y_test.head()
```

[22]:

	PULocationID	y	ancla
0	4	7	24
1	7	21	24
2	10	6	24
3	12	1	24
4	13	27	24

Figure 7: Creación de Y

3 Modelo supervisado.

3.1 Tabla analítica de datos (TAD).

Cuando ya tenemos Nuestras dos tablas X, Y, vamos a correr la función "TAD", la cual hace un merge entre estas dos tablas sobre la unidad muestral. tad

Creación de TAD (Tabla analítica de datos) $\vec{y} = f(\mathcal{X})$

```
tad_test = TAD(X_test,y_test,um)
tad_test.head()
```

	PULocationID	v_hora_min_n_3_4	v_hora_min_n_total_hora_4	v_hora_min_total_amount_3_4
0	4	1.0	1.0	5.80
1	7	1.0	1.0	4.80
2	10	1.0	1.0	35.38
3	12	1.0	1.0	7.55
4	13	1.0	1.0	0.31

Figure 8: TAD

3.2 Valores ausentes e imputación

Luego de haber conseguido la TAD, nos aseguramos si hay valores ausentes en ella. Si hay valores ausentes en TAD, pasamos por una función "impute" la cual nos va a regresar dos variables: imputar

Impute

[10]:
```python
def impute(tad, varc):
    '''impute

    Returns
    X: Based on TAD column with only columns from varc list
    Xi: Imputed df based on X with median strategy

    Parameters
    ..........
    tad: TAD table
    varc: list of continued variables

    '''

    X = tad[varc].copy()
    im = SimpleImputer(strategy='median')
    im.fit(X)
    Xtrans = im.transform(X)

    Xi = pd.DataFrame(Xtrans, columns=varc)

    #Kolmogorov-Smirnov (Two distributions are statistically equal)
    ks = pd.DataFrame(map(lambda v: (v,ks_2samp(tad[v].dropna(),Xi[v]).statistic),varc),columns=['variable','ks'])
    print(ks.loc[ks['ks']>.1])

    return X, Xi
```

Figure 9: Imputación

3.3 Valores extremos.

Para poder cuantificar todos nuestros valores extremos, pasamos nuestro dataset, unidad muestra y TAD a nuestra función "extreme" la cual nos imprimirá la cantidad de valores extremos en nuestro chunk de datos. extremos

Valores Extremos

[27]:
```
extreme(X_test, um, tad_test)
```

```
count    9585.000000
mean        0.509442
std         0.499937
min         0.000000
25%         0.000000
50%         1.000000
75%         1.000000
max         1.000000
Name: extremo, dtype: float64
```

Figure 10: Valores extremos

Como podemos dar cuenta, la cantidad de valores extremos representa un 0.95% de nuestro chunk.

3.4 Modelación.

Cuando hayamos conseguido nuestra tabla Xi (imputada), podemos seguir con nuestro modelo. Para ello pasamos a nuestra función "regression" la cual necesita los parámetros (tad, tgt, Xi).

En esta función vamos a hacer un split de nuestro chunk entre "train" y "test" para luego buscar los mejores hiperparámetros a través de "GridSearchCV" y para ello ya podemos hacer fit de nuestros datos.

Para poder verificar nuestro modelo, se imprimen los mean absolute error, primero con los valores de entrenamiento y después los valores para testing: mean errors

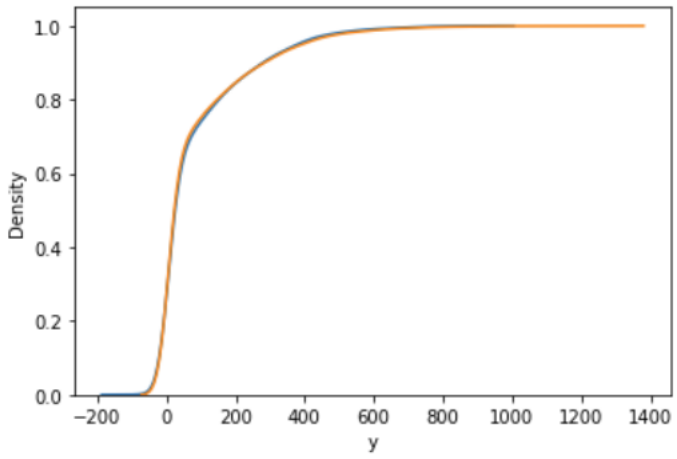
```
[34]: print(mean_absolute_error(y_true=yt_test,y_pred=modelo_test.predict(Xt_test)))
      print(mean_absolute_error(y_true=yv_test,y_pred=modelo_test.predict(Xv_test)))

      22.515774748259656
      22.701732484535974
```

Figure 11: Mean absolute error

Otra forma de ver la funcionalidad de nuestro modelo, es con las siguientes gráficas. La primer gráfica es con datos de training y la segunda con datos de testing: graficas

[35]:
```
sns.distplot(modelo_test.predict(Xt_test),hist=False,kde_kws={'cumulative':True})
sns.distplot(yt_test,hist=False,kde_kws={'cumulative':True})
```

[35]:
```
<AxesSubplot:xlabel='y', ylabel='Density'>
```



[36]:
```
sns.distplot(modelo_test.predict(Xv_test),hist=False,kde_kws={'cumulative':True})
sns.distplot(yv_test,hist=False,kde_kws={'cumulative':True})
```

[36]:
```
<AxesSubplot:xlabel='y', ylabel='Density'>
```

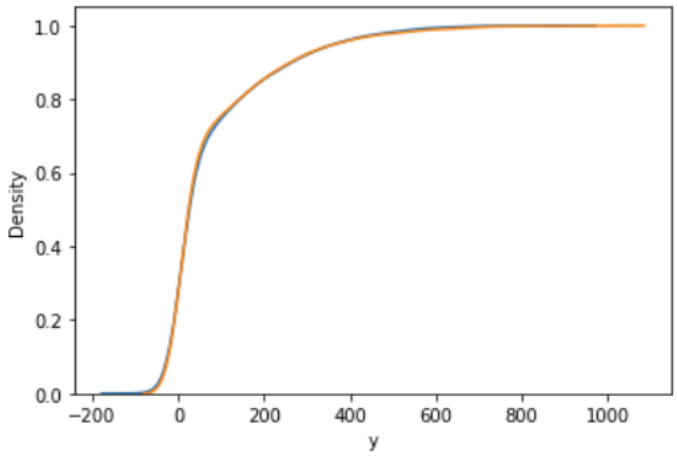


Figure 12: Gráficas del modelo

3.5 Corriendo los chunks

Al inicio de esta sección, se comentó que se usaron funciones al correr el proceso y, al momento de decirle a pandas que usaremos chunks, este nos regresa un iterador, el cual podemos usar en un for y así correr todos los modelos y haciendo append a una lista.

3.5.1 Predecir con todos los modelos

En estas ocaciones de modelado, lo ideal es usar la media de las predicciones de todos los modelos y así tener un resultado óptimo.

3.6 Persistencia

Ya que este es un modelado bastante pesado computacionalmente, lo ideal es que guardemos todos los modelos de nuestros chunks, para eso usaremos la librería "pickle" de la siguiente forma: pickle

[40]:
```
import pickle
```

[45]:
```
with open("models_other.pkl", "wb") as f:
    for model in models:
         pickle.dump(model, f)
```

Si queremos abrir los modelos de nuevo para evitar entrenamiento

[42]:
```
new_models = []
with open("models.pckl", "rb") as f:
    while True:
        try:
            new_models.append(pickle.load(f))
        except EOFError:
            break
```

[44]:
```
len(new_models)
```

[44]: 112

Figure 13: Persistencia modelo supervisado

Lo que estamos haciendo aquí, es guardar todos los modelos de nuestra lista en un mismo archivo. Si luego queremos leer esos modelos, simplemente los guardamos en una lista.

4 Modelo no supervisado.

4.1 Lectura de datos y limpieza.

En el caso del modelo no supervisado usamos la librería "Dask" de computación paralela al leer nuestro dataset y limpiarlo más eficiente. Similar a nuestro modelo supervisado, pasamos las variable que vayamos a utilizar para el clustering: lectura

```
csv = '2018_Yellow_Taxi_Trip_Data.csv'

fields = ['passenger_count', 'trip_distance', 'tip_amount',
          'total_amount', 'PULocationID', 'DOLocationID']
```

```
temp =dd.read_csv(csv,
                  usecols=fields,
                  dtype={'passenger_count':'int8', 'trip_distance':'float32',
                         'total_amount':'float32', 'PULocationID': 'int16',
                         'DOLocationID':'int16', 'tip_amount': 'float16'})
```

Figure 14: Lectura de datos modelo no supervisado

Una vez leído nuestro "dask dataframe" hacemos una descripción para datos ilógicos y procedemos a limpiarlos. filtro

```
[4]: temp = temp[temp['passenger_count'] > 0]
     temp = temp[temp['passenger_count'] < 7]
     temp = temp[temp['trip_distance'] > 0]
     temp = temp[temp['total_amount'] > 0]
```

Figure 15: Limpieza datos modelo no supervisado

4.2 Visualización.

Después de haber filtrado los datos, sacamos una muestra de nuestra información y lo convertimos a un dataframe de pandas, luego seleccionamos nuestras variables continuas y procedemos con el procesamiento.

4.2.1 PCA

Para las variables que seleccionamos, y al hacer un "StandardScaler", obtenemos una explicación de la varianza del 74% en dos dimensiones.
pca

PCA

```
[10]: sc = StandardScaler()
      sc.fit(X)
      Xs = pd.DataFrame(sc.transform(X),columns=varc)
      pca = PCA(n_components=2)
      pca.fit(Xs)
      pca.explained_variance_ratio_.cumsum()

[10]: array([0.5179963 , 0.74349976], dtype=float32)

[11]: Xp = pd.DataFrame(pca.transform(Xs),columns=['d1','d2'])

[12]: sns.lmplot(data=Xp,x='d1',y='d2',fit_reg=False)

[12]: <seaborn.axisgrid.FacetGrid at 0x7f574c407550>
```

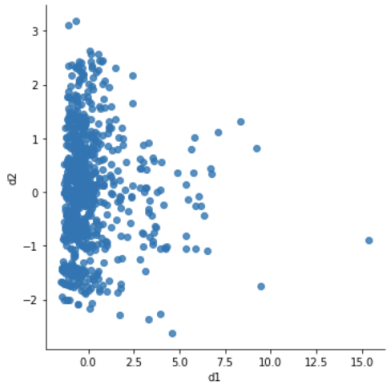


Figure 16: PCA

4.2.2 MDS

mds

MDS

```
[13]: mds = MDS(n_components=2,n_jobs=-1)
      Xm = pd.DataFrame(mds.fit_transform(X),columns=['d1','d2'])
      sns.lmplot(data=Xm,x='d1',y='d2',fit_reg=False)
```

```
[13]: <seaborn.axisgrid.FacetGrid at 0x7f574c3175f8>
```

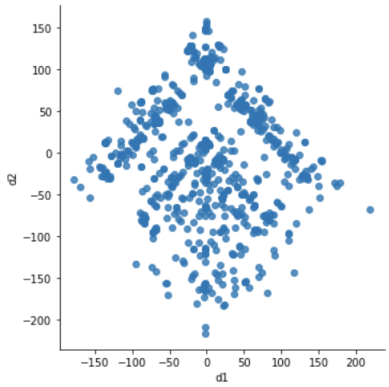


Figure 17: MDS

En MDS no podemos ver algún cluster con claridad

4.2.3 T-SNE

tsne

T-SNE

```
[14]: tsne = TSNE(n_components=2,n_jobs=-1)
      Xt = pd.DataFrame(tsne.fit_transform(X),columns=['d1','d2'])
      sns.lmplot(data=Xt,x='d1',y='d2',fit_reg=False)
```

```
[14]: <seaborn.axisgrid.FacetGrid at 0x7f574c2dfa58>
```

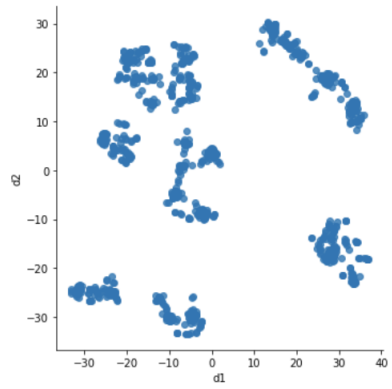


Figure 18: T-SNE

En T-SNE podemos ver un poco mejor algunos clusters

4.3 Número de clusters

Luego de haber visto la reducción de dimensiones y el análisis de correlaciones, pasamos a ver una gráfica de "codo" para determinar cuántos clusters haremos. codo

```
[18]: [<matplotlib.lines.Line2D at 0x7f5768fc6240>]
```

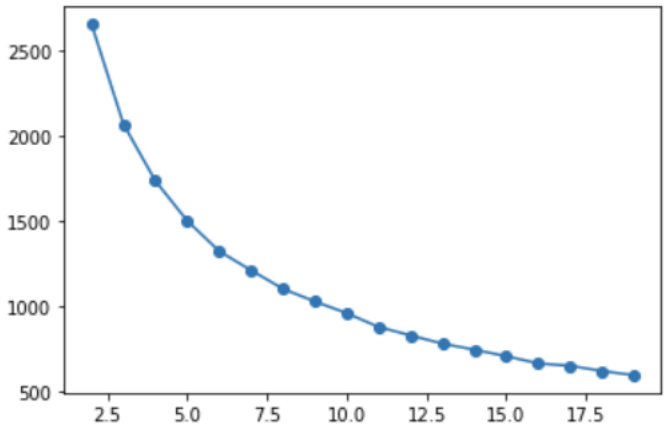


Figure 19: Gráfica de "codo"

En este caso vamos a tomar de referencia 7 clusters

4.3.1 Aglomerativo

Los porcentajes en este algoritmo son: aglomerativo

```
0    0.052761
1    0.360736
2    0.020859
3    0.134969
4    0.139877
5    0.160736
6    0.130061
Name: agg, dtype: float64
```

[21]: <AxesSubplot:ylabel='agg'>

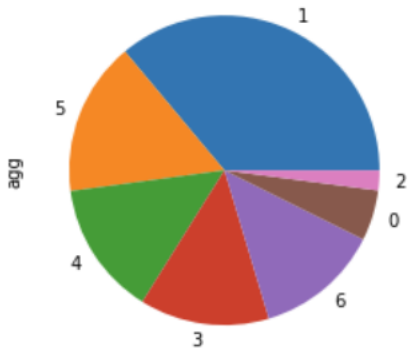


Figure 20: Aglomerativo

4.3.2 K-Means

Porcenyajes en K-Mean: k-means

```
0    0.166871
1    0.195092
2    0.285890
3    0.022086
4    0.072393
5    0.236810
6    0.020859
Name: kme, dtype: float64
```

[22]: <AxesSubplot:ylabel='kme'>

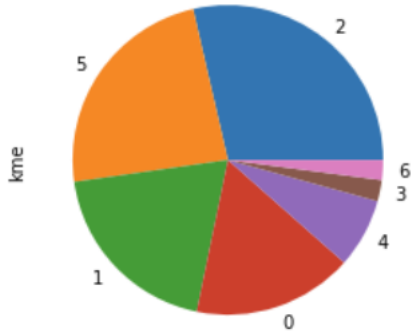


Figure 21: K-Means

4.3.3 Gaussian Mixture

Como podemos ver, la mezcla Gaussiana nos da una mejor distribución en el cluster gauss

```
0    0.159509
1    0.050307
2    0.269939
3    0.132515
4    0.169325
5    0.107975
6    0.110429
Name: gau, dtype: float64
```

[23]: <AxesSubplot:ylabel='gau'>

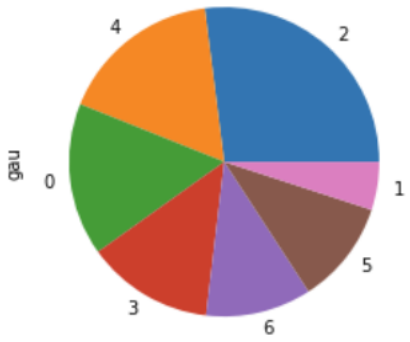


Figure 22: Gaussiana

4.3.4 Visualización 2D T-SNE y Gauss

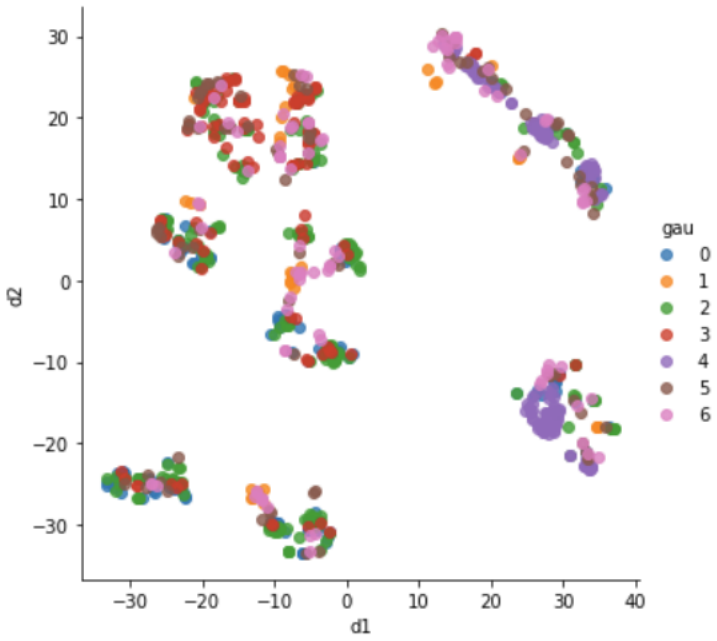


Figure 23: T-SNE con Gauss

En la figura anterior podemos observar que, aunque los clusters están mezclados, algunos colores dominan, en el lado derecho se puede aprecuar un dominio del cluster 4 mientras que en la esquina inferior izquierda hay más cluster 2

4.4 Perfilamiento.

Para poder verificar qué variables nos son más útiles para hacer clustering, debemos hacer un perfilamiento considerando el p-value. Una forma alternativa es usando el método "SelectKBest", el cual nos dice que todas nuestras variables son significativas.

4.5 Ejemplos.

Para observar nuestro clustering, podemos hacer lo siguiente: ejemplos

[64]:
```
k = 5
cl = 2
for id in df.loc[df['gau']==cl].sample(k).DOLocationID:
    print(id)
```

```
140
142
246
224
238
```

[69]:
```
i = 5
cluster = 4
for id in df.loc[df['gau']==cluster].sample(i).trip_distance:
    print(id)
```

```
0.699999988079071
1.2999999523162842
0.4699999988079071
0.6000000238418579
0.49000000953674316
```

Figure 24: Ejemplos modelo no supervisado

5 Estrategia de aplicación de los modelos.

Algunas propuestas para hacer uso de estos modelos son:

5.1 Modelo supervisado:

1. Predicción de demanda general y/o por zona: Al tener como base la locación donde se tomó el taxi, podemos generalizar para todo Nueva York o para una zona en específico. Además de mejorar costos al reducir el número de taxis dependiendo la hora del día.

2. Predicción taxis disponible: Con una pequeña modificación, podemos predecir cuántos taxis se liberan cada hora para así poder mover a los taxis a las zonas con mayor tráfico de clientes.

5.2 Modelo no supervisado:

1. Clusterización de zonas: Podemos hacer clusters de qué zonas son las que toman viajes más largos/cortos, las zonas con ticket promedio más alto. Y haciendo una combinación con el modelo supervisado, optimizar recursos para obtener mayores ingresos.

2. Clusters con mayor número de pasajeros: Con una modficación podríamos arreglar este algoritmo para mover carros más grandes a zonas donde los viajes tienen más ocupantes.

5.3 Nota.

En ambos modelos, las propuestas se pueden implementar de diferentes formas, ya sea que se predija con un servicio cloud a una aplicación web ó podemos exportar los modelos y hacer un dashboard para que equipos de business development tomen las mejores decisiones.

6 Conclusiones.

Al hacer este tipo de proyectos, se debe tener muy en cuenta toda la ingeniería de datos; ya que, si no la hacemos de la forma correcta nuestros modelos serán erroneos y no generalizarán. Por otro lado es interesante ver cómo se desarrolla el backend de muchas de las aplicaciones que usamos en la vida cotidiana en las cuales existen algoritmos de recomendación, mejor ruta, etc.

Una alternativa para el modelo supervisado sería probar con el algoritmo de redes neuronales y ver la diferencia de eficiencia comparado con regresión lineal. Además indagar en otras formas de procesamiento de la información; ya que, con chunks es bastante tardado debido a que pandas sólo utiliza un núcleo del procesador, tres posibles soluciones son: pasar la información a una base de datos SQL (aprovechando nuestras funciones), sacando una muestra para así ahorrarnos poder de cómputo ó finalmente usando la librería "Dask" de cómputo paralelo para poder procesar la información.

7 Anexos.

Los anexo con los códigos en Jupyter empiezan en la siguiente página.