In [1]:
```python
# Importacion de las bibliotecas de trabajo
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()
```

Paso 1: Lectura del dataset

In [2]:
```python
df = pd.read_csv("https://raw.githubusercontent.com/levraines/Portfolio/master/Master_Artificial%20Intelligence/Data/act1_healthcare-dataset-stroke-data.csv",
                 delimiter = ',')

# Creando copia del dataset
data = df.copy()

# Visualizando dataset
data.head()
```

Out[2]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status
0	9046	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	formerly smoked
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural	202.21	NaN	never smoked
2	31112	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked
3	60182	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	smokes
4	1665	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	never smoked

In [3]:
```python
# Comprobando el tamaño del dataset
data.shape
```

Out[3]:
```
(5110, 12)
```

Se puede ver que el dataset cuenta con un total de 5110 observaciones y un total de 12 características.

In [4]:
```python
# Viendo tipo de las variables
data.dtypes
```

Out[4]:
```
id                     int64
gender                object
age                  float64
hypertension           int64
heart_disease          int64
ever_married          object
work_type             object
Residence_type        object
avg_glucose_level    float64
bmi                  float64
smoking_status        object
stroke                 int64
dtype: object
```

Paso 2: Preprocesado

Lo siguiente que necesitamos hacer es convertir las variables categóricas (object) a tipo número, para este caso vamos a utilizar LabelEncoder ya que es menos costoso a nivel computacional y dado a que se cuentan con diferentes variables que tienen factores que no obedecen a un orden específico el algoritmo no creará asociaciones entre los datos por esta discretización, ya que, el algoritmo que vamos a utilizar es un KNN no hay mayor problema al utilizar dicha forma de preprocesado.

In [5]:
```python
# Hay que transformar a numéricas las variables categóricas (object) para poder trabajar con ellas
obj_df = data.select_dtypes(include=['object']).copy() # se eligen las variables categoricas (object) y se hace una copia
print(obj_df.columns)
```

```
Index(['gender', 'ever_married', 'work_type', 'Residence_type',
       'smoking_status'],
      dtype='object')
```

In [6]:
```python
# Importando Label Encoder
from sklearn.preprocessing import LabelEncoder
```

In [7]:
```python
# Le cambio el nombre para poder manipular
lb_encoder = LabelEncoder()
```

In [8]:
```python
# Voy a crear un ciclo for para que a cada una de las variables que tengan formato 'object' las cambie a formato numero
for col in obj_df.columns:
    data[col] = lb_encoder.fit_transform(data[col])
```

In [9]:
```python
# Comprobando tipo de variable
data.dtypes
```

Out[9]:
```
id                     int64
gender                 int64
age                  float64
hypertension           int64
heart_disease          int64
ever_married           int64
work_type              int64
Residence_type         int64
avg_glucose_level    float64
bmi                  float64
smoking_status         int64
stroke                 int64
dtype: object
```

Como podemos observar, todas las variables que antes eran categóricas ahora han sido modificadas y son numéricas (ya que se les asignó nuevas etiquetas por cada factor presente dentro de la variable). Además, el ejercicio nos pide lo siguiente:

"Existen 1544 personas de las que se desconoce si fuman o no; vamos a asignar también un número para esta posibilidad."

In [10]:
```python
# Comprobando el nuevo numero asignado a la clase "unkown"
data["smoking_status"].value_counts()
```

Out[10]:
```
2    1892
0    1544
1     885
3     789
Name: smoking_status, dtype: int64
```

Ccmo se puede observar, ahora el factor "unknow" ha sido sustituido por el número 0 cumpliéndose por lo requerido por el ejercicio.

In [11]:
```python
# Comprobando variables nulas o registros faltantes
np.sum(data.isnull())
```

Out[11]:
```
id                     0
gender                 0
age                    0
hypertension           0
heart_disease          0
ever_married           0
work_type              0
Residence_type         0
avg_glucose_level      0
bmi                  201
smoking_status         0
stroke                 0
dtype: int64
```

Se puede apreciar que en la variable "bmi" existen 201 registros nulos, por ende se va a proceder a eliminarlos ya que sólo corresponden a un 4% del total de registros.

In [12]:
```python
# Eliminando los 201 registros nulos en la variable bmi
data = data.dropna()

# Reseteando el indice para evitar futuros problemas
data = data.reset_index()
```

In [13]:
```python
# Comprobando de nuevo las variables despues de la eliminacion
np.sum(data.isnull())
```

Out[13]:
```
index                0
id                   0
gender               0
age                  0
hypertension         0
heart_disease        0
ever_married         0
work_type            0
Residence_type       0
avg_glucose_level    0
bmi                  0
smoking_status       0
stroke               0
dtype: int64
```

Se puede observar que ya no contamos con variables que contengan registros nulos o con NA.

In [14]:
```python
# Comprobando tamano del dataset
data.shape
```

Out[14]:
```
(4909, 13)
```

In [15]:
```python
# Redefiniendo dataset con variables originales
data = data[['id', 'gender', 'age', 'hypertension', 'heart_disease',
        'ever_married', 'work_type', 'Residence_type', 'avg_glucose_level',
        'bmi', 'smoking_status', 'stroke']]
```

Ahora bien, después del trabajo previo y habiendo eliminado las 201 observaciones que se solicitaba se cuenta con un nuevo dataset de 4909 registros.

Escalado de los datos numéricos

In [16]:
```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled = scaler.fit(data.iloc[:,0:-1]).transform(data.iloc[:,0:-1])
```

In [17]:
```python
dfs = pd.DataFrame(scaled, columns=df.columns[:-1])
dfs.head()
```

Out[17]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi
0	-1.334653	1.198428	1.070138	-0.318067	4.381968	0.729484	-0.155697	0.985640	2.777698	0.981345
1	-0.283539	1.198428	1.646563	-0.318067	4.381968	0.729484	-0.155697	-1.014569	0.013842	0.459269
2	1.101211	-0.833023	0.272012	-0.318067	-0.228208	0.729484	-0.155697	0.985640	1.484132	0.701207
3	-1.686247	-0.833023	1.602222	3.143994	-0.228208	0.729484	0.759651	-1.014569	1.549193	-0.623083
4	0.933870	1.198428	1.690903	-0.318067	-0.228208	0.729484	-0.155697	0.985640	1.821368	0.013595

In [18]:
```python
# Uniendo dataset con variable de repuesta
dfs = pd.concat([dfs, data["stroke"]], axis=1)
dfs.head(5)
```

Out[18]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi
0	-1.334653	1.198428	1.070138	-0.318067	4.381968	0.729484	-0.155697	0.985640	2.777698	0.981345
1	-0.283539	1.198428	1.646563	-0.318067	4.381968	0.729484	-0.155697	-1.014569	0.013842	0.459269
2	1.101211	-0.833023	0.272012	-0.318067	-0.228208	0.729484	-0.155697	0.985640	1.484132	0.701207
3	-1.686247	-0.833023	1.602222	3.143994	-0.228208	0.729484	0.759651	-1.014569	1.549193	-0.623083
4	0.933870	1.198428	1.690903	-0.318067	-0.228208	0.729484	-0.155697	0.985640	1.821368	0.013595

In [19]:
```python
# Comprobando que la estandarización se haya hecho de forma correcta, ya que tendria que se media 0 y desviación estándar de 1.
print(round(dfs["age"].mean(),2))
print(round(dfs["age"].std(),2))
```

```
0.0
1.0
```

In [20]:
```python
# Comprobando clase a predecir y verificando si esta balanceada
data["stroke"].value_counts()
```

Out[20]:
```
0    4700
1     209
Name: stroke, dtype: int64
```

In [21]:
```python
round((data["stroke"].value_counts()/len(data))*100, 2)
```

Out[21]:
```
0    95.74
1     4.26
Name: stroke, dtype: float64
```

Podemos observar que el factor "Sí" o 1, tiene un total de 209 observaciones que indican que el paciente sí ha tenido un accidente cerebro vascular, mientras que tenemos 4700 observaciones que indican que "No" lo ha tenido, por ende tenemos clases bastante desbalanceadas en una relación de 95.74% para pacientes que no han sufrido accidentes cerebro vasculares versus 4.26% que sí han sufrido dichos accidentes.

Paso 3: Generación del Modelo KNN

Antes de comenzar con el modelo es importante aclarar que se va a eliminar la variable "id" ya que esta variable es el identificador único y esta es una variable que no le añade valor por sí solo al modelo, ya que es única y no se puede utilizar para algún tipo de asociación por ello la vamos a eliminar.

In [22]:
```python
# Extrayendo los nombres de las variables
dfs.columns
```

Out[22]:
```
Index(['id', 'gender', 'age', 'hypertension', 'heart_disease', 'ever_married',
       'work_type', 'Residence_type', 'avg_glucose_level', 'bmi',
       'smoking_status', 'stroke'],
      dtype='object')
```

In [23]:
```python
# Creando nuevo subset sin la variable "id"
dfs = dfs[['gender','age', 'hypertension', 'heart_disease', 'ever_married', 'work_type',
       'Residence_type', 'avg_glucose_level', 'bmi', 'smoking_status','stroke']]

# Visualizando resultados
dfs.head(5)
```

Out[23]:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_statu
0	1.198428	1.070138	-0.318067	4.381968	0.729484	-0.155697	0.985640	2.777698	0.981345	-0.35178
1	1.198428	1.646563	-0.318067	4.381968	0.729484	-0.155697	-1.014569	0.013842	0.459269	0.58523
2	-0.833023	0.272012	-0.318067	-0.228208	0.729484	-0.155697	0.985640	1.484132	0.701207	1.52225
3	-0.833023	1.602222	3.143994	-0.228208	0.729484	0.759651	-1.014569	1.549193	-0.623083	0.58523
4	1.198428	1.690903	-0.318067	-0.228208	0.729484	-0.155697	0.985640	1.821368	0.013595	-0.35178

Ahora que hemos eliminado la variable "id" es momento de empezar a modelar nuestro problema. Para ello para agregarle un poco de aleatoridad y modificar el orden original, por ello vamos a utilizar la función "sample" para aleatorizar la base y vamos a usar el 100% de los registros, por ello:

In [24]:
```python
# Aleatorizando la base original
dfs = dfs.sample(frac=1.0, random_state=42)

# Comprobando que todos los registros sigan permaneciendo
len(data)
```

Out[24]:
```
4909
```

Selección del modelo KNN

In [25]:
```python
# Creando subsets de variables de entrada y salida
X = dfs.iloc[:,0:-1]
y = dfs.iloc[:,-1]
```

In [26]:
```python
# Haciendo split de los datos, 20% de testeo y 80% de training
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42, test_size=0.2)
print (X_train.shape, y_train.shape)
print (X_test.shape, y_test.shape)
```

```
(3927, 10) (3927,)
(982, 10) (982,)
```

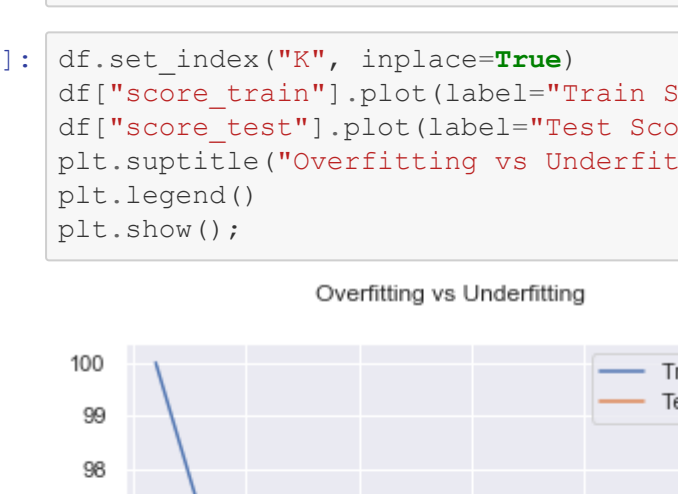
In [27]:
```python
# Importando bibliotecas del KNN junto con sus metricas
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_predict, cross_val_score
```

Ahora lo que vamos a hacer es calcular el valor de **k** mediante un for loop, para valorar cuál es el valor de K que mejor "accuracy" tiene.

Este es un método que se puede utilizar, aunque existen otras técnicas para calcular la **k**, ya que una de las más usuales es tomar la población total, en este caso 4909 y sacarle la raíz cuadrada y después al resultado redondearlo al número impar próximo, así nos garantizamos de que cuando creamos los vecinos no tener vecinos pares (**k**) por ende el algoritmo cuando vaya a hacer la clasificación y cuente con observaciones pares no se confunda y siempre se pueda decidir por una en concreto. Para este caso la cantidad de vecinos sería 71 sin embargo si lo hacemos de esta forma nuestro proceso computacional se vuelve muy costoso y no necesariamente vamos a optimzar la clasificación.

In [28]:
```python
k_list = []
score_train_list = []
score_test_list = []

for K in list(range(1,30,3)):
  knn_model = KNeighborsClassifier(n_neighbors=K, weights="uniform", metric="minkowski")
  knn_model.fit(X_train, y_train)

  ## Generación de listas
  k_list.append(K)
  score_train_list.append(100.0*knn_model.score(X_train, y_train))
  score_test_list.append(100.0*knn_model.score(X_test, y_test))

df = pd.DataFrame({"K":k_list,
                   "score_train":score_train_list,
                   "score_test":score_test_list})
```

In [29]:
```python
df.set_index("K", inplace=True)
df["score_train"].plot(label="Train Score")
df["score_test"].plot(label="Test Score")
plt.suptitle("Overfitting vs Underfitting")
plt.legend()
plt.show();
```



Con la gráfica se puede observar como el valor de K para cuando esta toma el valor de 7 (aproximadamente) el accuracy ronda más del 96% y después la gráfica se estabiliza, entonces podríamos tomar valores como 7 u 9 (hay que evitar tomar valores pares por lo que se mencionó antes). Vamos a proceder a utilizar K = 7.

Aaunque viendo la gráfica lo normal es que la curva de Train esté siempre por encima de Test cuando hablamos de "accuracy". Pero también hay que tener en cuenta que "accuracy" no es la mejor métrica a tener en cuenta cuando hablamos de clasificación (con clases altamente desbalanceadas). Aunque sí nos sirve como referencia para seleccionar el valor de K.

Entrenamiento y Evaluación del modelo

Para este apartado vamos a trabajar con:

1. K = 7.
2. Weights = "distance", de este modo no sólo hace un recuento de los puntos sino que los pondera por la inversa de su distancia al objetivo.
3. Metric = vamos a dejar la minkowski que viene por defecto.

In [30]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn_model = KNeighborsClassifier(n_neighbors=7, weights="distance", metric="minkowski")
knn_model.fit(X_train, y_train)
pred_train = knn_model.predict(X_train)
pred_test = knn_model.predict(X_test)
print("Precisión sobre los datos de entrenamiento: {:.2f}".format(100.0*knn_model.score(X_train, y_train)))
print("Precisión sobre los datos de test: {:.2f}".format(100.0*knn_model.score(X_test, y_test)))
```

```
Precisión sobre los datos de entrenamiento: 100.00
Precisión sobre los datos de test: 95.62
```

Si solo tomamos en cuenta el valor del accuracy, tenemos resultados muy parecidos a los que ya indicamos en el apartado anterior, es decir mayores a 96%, en este caso podemos observar como la precisión de los datos de entrenamiento es de un 100%, mientras que la del test es de un 95.62%.

¡**Cuidado**! Cuando tenemos clases altamente desbalanceadas el accuracy no es un buen indicador.

In [31]:
```python
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, pred_test)

##          predicciones
##             0      1
##       0    TN     FP
## real
##       1    FN     TP
```

Out[31]:
```
array([[938,   2],
       [ 41,   1]])
```

Con la matriz de confusión podemos ver que en realidad el accuracy de más de un 95% pertenecía a la clase dominante, es decir a todos aquellos pacientes que no han tenido un accidente cerebro vascular, de ahí la importancia de calcular siempre la matriz de confusión.

Además si nos adentramos a dicha matriz podemos ver que de 940 observaciones del no, logró detectar correctamente 938 (TP) y se equivocó en 2 (FN), mientras que de las 42 observaciones que corresponden al sí, solo pudo predecir correctamente 1 (TN) y se equivocó en 41 (FP).

Podemos observar como el error tipo I o alfa, es el más alto de los dos, ya que el error tipo II es apenas notorio.

Recall, Precision y F1-Score

In [32]:
```python
from sklearn.metrics import classification_report
print(f"Informe de Clasificación:\n"
      f"{classification_report(y_test, pred_test)}")
```

```
Informe de Clasificación:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       940
           1       0.33      0.02      0.04        42

    accuracy                           0.96       982
   macro avg       0.65      0.51      0.51       982
weighted avg       0.93      0.96      0.94       982
```

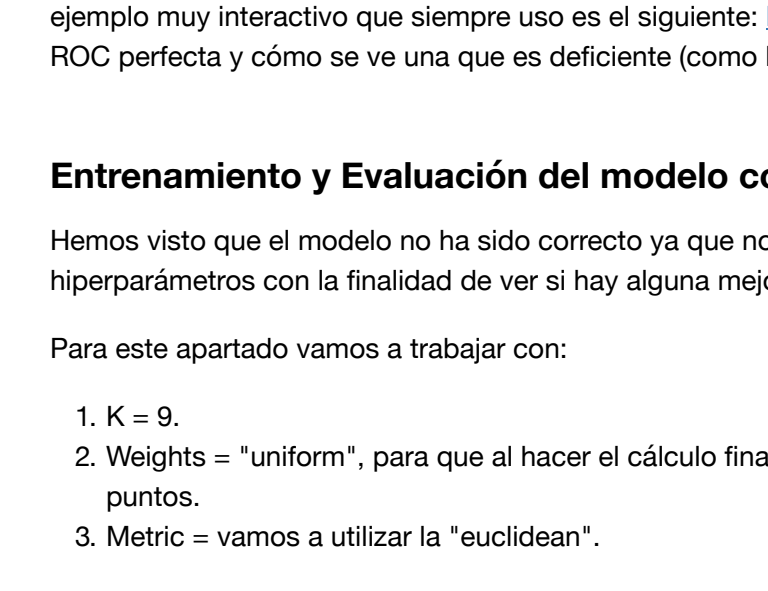
Aunque a simple vista podemos ver que el valor 0 tiene muy buenas métricas, nos vamos a fijar en la fila donde dice 1. Ahí es donde se produce el fallo del modelo. Al ser un dataset desbalanceado las métricas para 0 son casi perfectas pero para el caso de sí son bastante deficientes.

Por otro lado si vemos el F1 la métrica del "sí" es apenas un 0.04 por no decir, 0, lo cual nos permite ver que el modelo es bastante deficiente en la clase menos privilegiada.

Vamos a calular la CURVA ROC de dicho modelo para ver cómo se ve.

Calculando CURVA ROC

In [33]:
```python
from sklearn.metrics import plot_roc_curve
knn_model = KNeighborsClassifier(n_neighbors=7, weights="distance", metric="minkowski")
knn_model.fit(X_train, y_train)
plot_roc_curve(knn_model, X_test, y_test)
plt.suptitle("ROC-AUC-CURVE")
plt.show()
```



Como era de esperarse la curva ROC es bastante deficiente, ya que una buena curva ROC tiene que ir bastante pegada a la esquina superior izquierda mientras que esta está totalmente inclinada. En este caso esta CURVA ROC no es capaz de diferenciar bien los FN. Un ejemplo muy interactivo que siempre uso es el siguiente: http://www.navan.name/roc/ aquí se puede observar cómo se vería una CURVA ROC perfecta y cómo se ve una que es deficiente (como la que estamos observando en este momento).

Entrenamiento y Evaluación del modelo con cambios en los hiperparámetros

Hemos visto que el modelo no ha sido correcto ya que no logra hacer una clasificación correcta, entonces vamos a probar a cambiar los hiperparámetros con la finalidad de ver si hay alguna mejora.

Para este apartado vamos a trabajar con:

1. K = 9.
2. Weights = "uniform", para que al hacer el cálculo final de estimación/predicción no tome en cuenta las distancias, sólo el número de puntos.
3. Metric = vamos a utilizar la "euclidean".

In [34]:
```python
from sklearn.neighbors import KNeighborsClassifier
knn_model = KNeighborsClassifier(n_neighbors=9, weights="uniform", metric="euclidean")
knn_model.fit(X_train, y_train)
pred_train = knn_model.predict(X_train)
pred_test = knn_model.predict(X_test)
print("Precisión sobre los datos de entrenamiento: {:.2f}".format(100.0*knn_model.score(X_train, y_train)))
print("Precisión sobre los datos de test: {:.2f}".format(100.0*knn_model.score(X_test, y_test)))
```

```
Precisión sobre los datos de entrenamiento: 95.75
Precisión sobre los datos de test: 95.72
```

Si solo tomamos en cuenta el valor del accuracy, tenemos resultados ligeramente más bajos a los del modelo anterior, ya que la precisión de los datos de entrenamiento del modelo pasado era de 100% y para este es de 95.75% mientras el modelo pasado nos dio que en el test era de un 95.62% y para este es de 95.72% (en este caso es ligeramente superior).

Vamos a ver cómo se comporta la clase menos privilegiada.

In [35]:
```python
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, pred_test)

##          predicciones
##             0      1
##       0    TN     FP
## real
##       1    FN     TP
```

Out[35]:
```
array([[940,   0],
       [ 42,   0]])
```

Con la matriz de confusión lo que podemos ver es que el test dio mejores resultados porque en realidad de 940 observaciones en el testing, 940 pudo clasificarlas de forma correcta (TP) no tuvo FN, pero tuvo 42 FP y ni uno solo de TN, esto lo que nos dice es que este algoritmo es extremadamente pobre.

Recall, Precision y F1-Score

In [37]:
```python
from sklearn.metrics import classification_report
print(f"Informe de Clasificación:\n"
      f"{classification_report(y_test, pred_test)}")
```

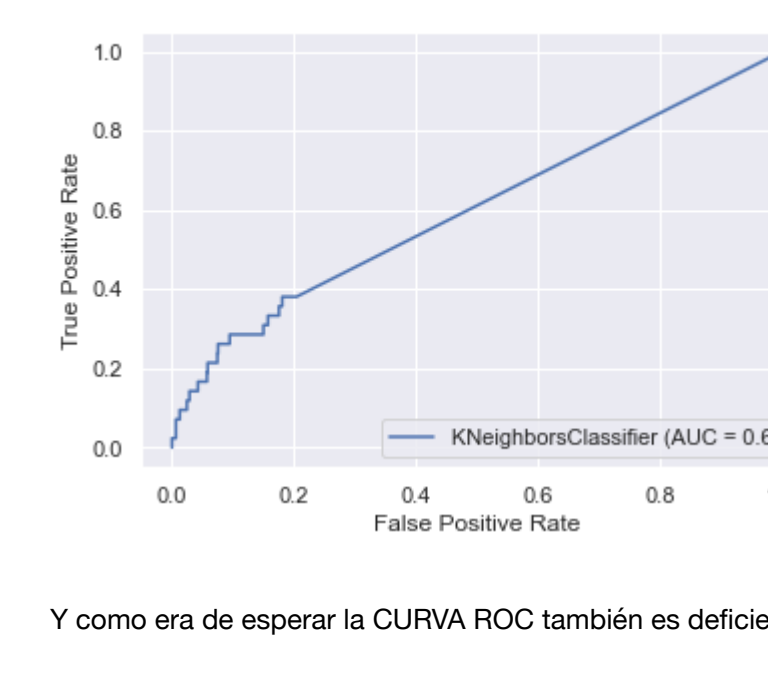
```
Informe de Clasificación:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       940
           1       0.00      0.00      0.00        42

    accuracy                           0.96       982
   macro avg       0.48      0.50      0.49       982
weighted avg       0.92      0.96      0.94       982
```

```
/Users/heinerleivagmail.com/opt/anaconda3/lib/python3.7/site-packages/sklearn/metrics/_classification.py:1221: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Como era de esperarse a nivel del factor 0 se obtienen muy buenos resultados, pero si se ve en este caso el factor o la línea que dice 1 se ve que todos los resultados son muy pobres y que en efecto el modelo no se puede utilizar para detectar futuros o posibles accidentes cerebro vasculares.

Calculando CURVA ROC

In [38]:
```python
from sklearn.metrics import plot_roc_curve
knn_model = KNeighborsClassifier(n_neighbors=7, weights="distance", metric="minkowski")
knn_model.fit(X_train, y_train)
plot_roc_curve(knn_model, X_test, y_test)
plt.suptitle("ROC-AUC-CURVE")
plt.show()
```



Y como era de esperar la CURVA ROC también es deficiente ya que no logra diferenciar los FP y los confunde con TP.

Conclusiones

El KNN puede ser un modelo poderoso cuando estamos en presencia de clases balanceadas y generalmente da buenos resultados.

Para nuestro caso en especifico no fue así y dio resultados bastante pobres, incluso cambiando los hiperparámetros, sin embargo esto es muy usual que pase en la realidad, por eso se han desarrollado técnicas para solucionar las clases desbalanceadas como las técnicas SMOTE (Synthetic Minority Oversampling Technique) que lo que hacen es tratar de balancear el dataset aplicando un oversampling a la clase menos privilegiada y así se puede llegar a resultados bastante notables entre otras técnicas más. En lo personal algo que siempre consulto cuando tengo estos problemas en el trabajo es el siguiente link: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/. También podemos hacer Downsampling o Upweighting que son otras técnicas bastante utilizadas.

Aunque también todo va a depender del problema, porque por ejemplo, para este ejercicio si para una persona es muy importante que el algoritmo siempre detecte cuando un paciente NO va a tener algún accidente cerebro vascular, este modelo es bastante robusto porque cumple su cometido, pero en la vida real eso raras veces pasa, porque siempre vamos a querer predecir de alguna enfermedad que un paciente puede llegar a padecer, por eso cuando aplicamos ML en salud es tan común que no nos interese clasificar mal a una persona sana con tal de que no se nos escape alguna que pueda llegar a tener o desarrollar alguna enfermedad. Lo ideal sería clasificar bien a ambos tipos de pacientes pero en la vida real eso raras veces sucede.