

AGiulianoMirabella

April 6, 2021

1 Análisis de la felicidad con Data Science

(Parte individual)

- Autor: A. Giuliano Mirabella
- Última modificación: 06/04/2021

Este notebook sirve como memoria para la entrega de la parte individual de la asignatura de Data Science, donde mi compañero y yo hemos aplicado técnicas de Data Science a un dataset que guarda datos sobre la felicidad de varios países a lo largo de varios años.

A lo largo de este documento, se intentará responder a las preguntas:

- ¿se podría comprimir la información del dataset?
- ¿puede una red neuronal predecir la felicidad?

1.1 Índice:

- Principal Components Analysis
- MultiLayer Perceptron

1.2 PCA

En esta sección estudiare la pregunta: ¿se podría comprimir la información del dataset?

Para ello, recurriré a la técnica de *Principal Component Analysis* (PCA), que consiste en crear una combinación lineal de las columnas originales para dar lugar a nuevas columnas artificiales que puedan *describir mejor* el dataset. Describir “mejor” significa proporcionar suficiente variabilidad a pesar de la compresión de la información.

Las nuevas variables artificiales no tienen (*a priori*) un significado semántico en el dominio del problema, pero pueden ser útiles para el tratamiento de datos cuando el número de columnas es excesivo, o incluso para especular sobre la *riqueza* de información presente en el dataset: si pocas *principal components* (PCs) son suficientes para conseguir una alta variabilidad entonces podemos suponer que las variables originales no difieren mucho entre sí, están correlacionadas, y por tanto no aportan mucha información relevante. Si, por el contrario, son necesarias muchas PCs, entonces podemos suponer que estamos ante un dataset donde cada variable aporta información significativa para la extracción de conocimiento.

Por último, es importante destacar que en este apartado se han tomado como referencia las prácticas de la asignatura de Machine Learning Engineering (MLE). De hecho, para mayor claridad, he modificado lo mínimo posible el código, con ánimo de no dar lugar a posibles sospechas.

Entiendo que como ingeniero tenemos la habilidad de saber utilizar herramientas adecuadas cuando ya existen y saber desarrollar nuevas cuando no: en este caso, la técnica de PCA se aborda muy bien en dichas prácticas, por lo que me ha parecido inteligente reutilizar ciertos aspectos. Además, entiendo que se me evaluará por la capacidad de comprensión de la técnica, más que por la aplicación en sí.

Empecemos importando los paquetes necesarios y definiendo funciones útiles:

```
[1]: from sklearn.decomposition import PCA
from sklearn.feature_selection import f_classif, SelectPercentile
from sklearn.preprocessing import MinMaxScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```



```
[2]: def read_data(year=None):
    if year:
        return __read_year(year)
    df = __read_year(2015)
    for year in [2016, 2017, 2018, 2019]:
        df = pd.concat([df, __read_year(year)], axis=0)
    df = df.drop(columns = ['country', 'year'])
    columns = [c for c in df.columns if not df[c].isnull().values.all()]
    df = pd.DataFrame(SimpleImputer(strategy='mean').fit_transform(df), columns=columns)
    df = pd.DataFrame(MinMaxScaler().fit_transform(df), columns=df.columns)
    return df

def __read_year(year):
    df = pd.read_csv('../data/' + str(year) + '.csv')
    df['year'] = year
    return df

def evalua(X, y):
    return LinearRegression().fit(X, y).score(X, y)
```



```
[3]: df = read_data()
df.head()
```



```
[3]:   health   family   economy   freedom  generosity  corruption  dystopia \
0  0.825092  0.820870  0.666274  0.919296  0.354121  0.760595  0.623743
1  0.830710  0.852938  0.621336  0.868467  0.520598  0.256292  0.676357
2  0.766556  0.827603  0.632385  0.896934  0.407350  0.876175  0.616521
3  0.775819  0.809580  0.696088  0.925041  0.414032  0.661394  0.608904
```

```
4  0.793716  0.804507  0.632772  0.874268      0.546622      0.597144  0.605043  
    score  
0  0.964145  
1  0.959023  
2  0.952325  
3  0.951340  
4  0.932624
```

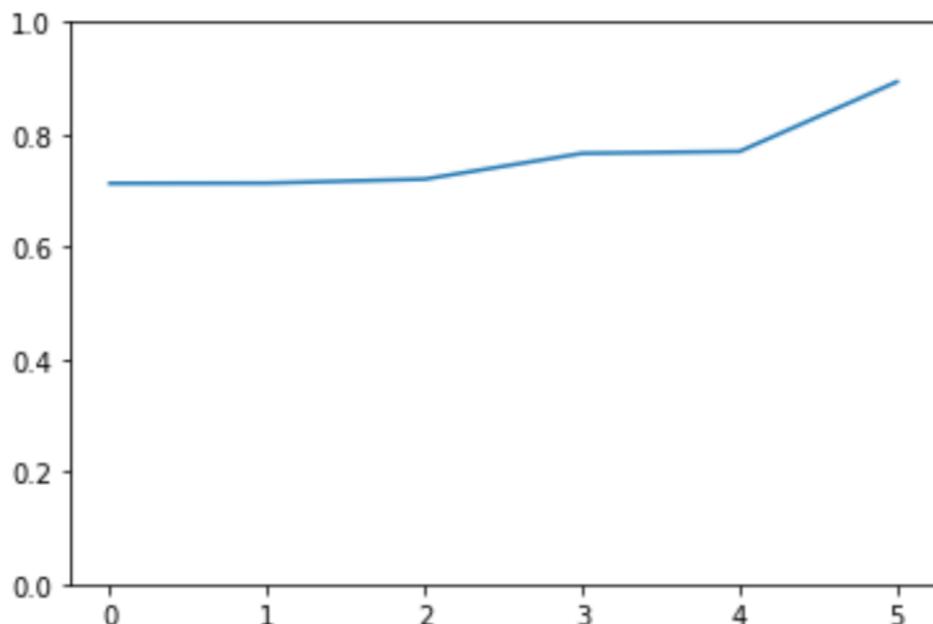
```
[4]: TARGET = 'score'
```

```
y = df.pop(TARGET)  
X = df
```

```
[5]: def evolucion_n_components(X, y):
```

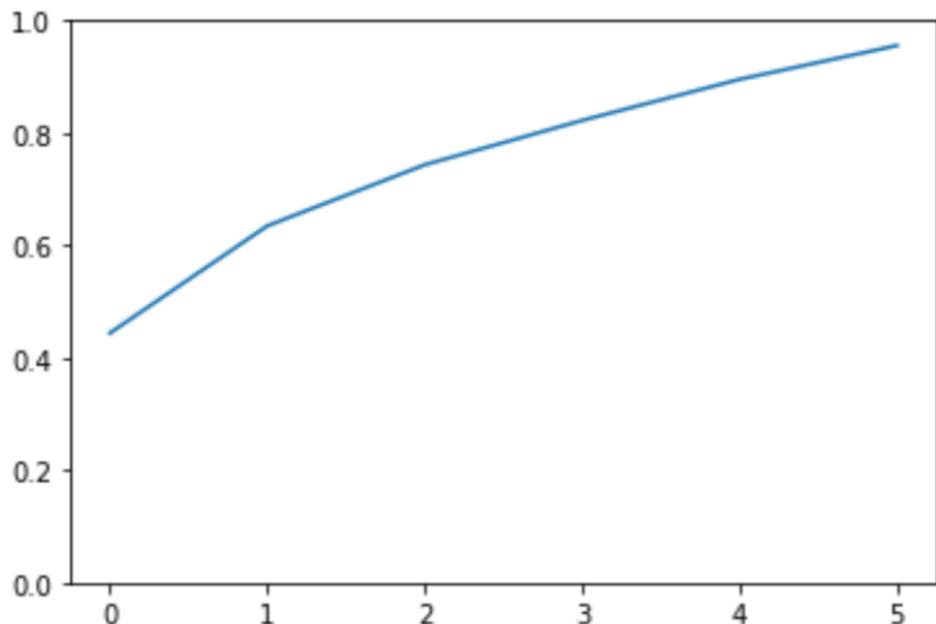
```
    resultados = []  
    for i in range(1,len(X.columns)):  
        pca = PCA(n_components=i)  
        X_pca = pca.fit_transform(X)  
        resultados.append(evalua(X_pca,y))  
    plt.plot(resultados)  
    plt.ylim(0, 1)  
    plt.show()
```

```
[6]: evolucion_n_components(X, y)
```



```
[7]: def evolucion_varianza(X, y):
    resultados = []
    for i in range(1,len(X.columns)):
        pca = PCA(n_components=i).fit(X, y)
        resultados.append(sum(pca.explained_variance_ratio_))
    plt.plot(resultados)
    plt.ylim(0, 1)
    plt.show()
```

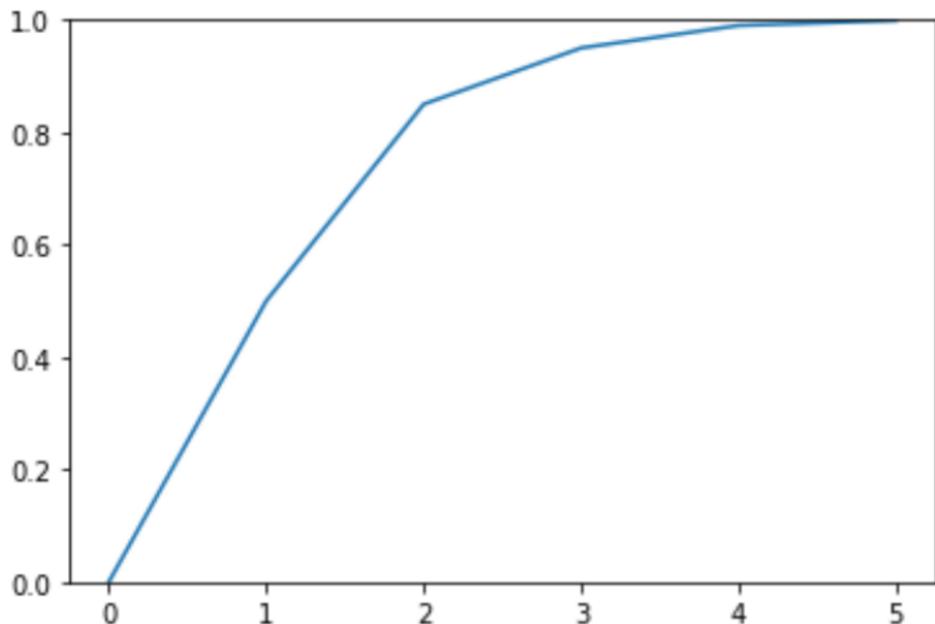
```
[8]: evolucion_varianza(X, y)
```



Como podemos observar en la gráfica anterior, la pendiente de la varianza porcentual acumulada se mantiene relativamente constante con el aumento de las PCs. Esto significa que las últimas PCs (que son las menos importantes) también aportan información relevante al dataset en cuanto a varianza.

De no ser así, observaríamos una gráfica más parecida a ésta:

```
[9]: ejemplo = [0, 0.5, 0.85, 0.95, 0.99, 0.999]
plt.plot(ejemplo)
plt.ylim(0, 1)
plt.show()
```



En cualquier caso, también hay que tener en cuenta que, aunque sea constante, la pendiente inicial no es muy alta, y de hecho únicamente con dos variables se podría llegaría a un 70% de variabilidad aproximadamente.

1.3 MLP

En esta segunda parte vamos a estudiar la capacidad que tiene una red neuronal de predecir algo tan complejo como la felicidad. Tanto en la parte grupal como en la sección anterior hemos comprobado que un regresor lineal simple puede conseguir hasta un ~80% de precisión en predecir los valores de felicidad. Ahora vamos a utilizar un perceptrón multicapa (red neuronal), que es un sistema no lineal y que por tanto debería ser capaz de aprender relaciones de segundo orden entre variables y en última instancia de predecir con más precisión la felicidad. Veamos si efectivamente es así.

Para este apartado hay que tener en cuenta que, por premiar la simplicidad, he implementado toda la lógica en ficheros a parte (no en este mismo notebook). Para revisar el código basta con abrirlo en cualquier editor de texto.

Finalmente, he intentado que el **valor añadido** principal de este estudio sea la implementación de un sistema que automáticamente elija la mejor configuración de hiperparámetros para una determinada tarea, dados unos hiperparámetros a testear.

Empecemos, como antes, cargando la funcionalidad necesaria:

```
[17]: from root.read_data import read_data as read
from root.mlp.auto import AutoRegressor
from root.preprocessing import regressor_preprocess

raw_data = read()
```

```

preprocessed_data = regressor_preprocess(raw_data)

architectures  = ['arch0', 'arch1']
batch_sizes    = [2, 4, 8, 16, 32]
learning_rates = [0.01, 0.05, 0.1]
momentums     = [0, 0.01, 0.1]
optimizers     = ['Adam', 'sgd']

auto = AutoRegressor(
    preprocessed_data,
    architecture_names = architectures,
    batch_sizes = batch_sizes,
    learning_rates = learning_rates,
    momentums = momentums,
    optimizers = optimizers
)
auto.best()

```

Model: "sequential_36"

Layer (type)	Output Shape	Param #
<hr/>		
dense_144 (Dense)	(None, 16)	160
<hr/>		
dense_145 (Dense)	(None, 8)	136
<hr/>		
dropout_36 (Dropout)	(None, 8)	0
<hr/>		
dense_146 (Dense)	(None, 4)	36
<hr/>		
dropout_37 (Dropout)	(None, 4)	0
<hr/>		
dense_147 (Dense)	(None, 2)	10
<hr/>		
dense_148 (Dense)	(None, 1)	3
<hr/>		

Total params: 345

Trainable params: 345

Non-trainable params: 0

Epoch 1/100

250/250 [=====] - 0s 1ms/step - loss: 544.1500 -
mean_squared_error: 31.0279 - mean_absolute_percentage_error: 100.3488 -
val_loss: 542.8607 - val_mean_squared_error: 30.2501 -
val_mean_absolute_percentage_error: 100.0000

```
125/125 [=====] - 0s 869us/step - loss: 30.2524 -
mean_squared_error: 30.2524 - mean_absolute_percentage_error: 100.0000 -
val_loss: 31.4886 - val_mean_squared_error: 31.4886 -
val_mean_absolute_percentage_error: 100.0000
Epoch 47/100
125/125 [=====] - 0s 856us/step - loss: 30.2524 -
mean_squared_error: 30.2524 - mean_absolute_percentage_error: 100.0000 -
val_loss: 31.4886 - val_mean_squared_error: 31.4886 -
val_mean_absolute_percentage_error: 100.0000
40/40 [=====] - 0s 650us/step - loss: 29.0238 -
mean_squared_error: 29.0238 - mean_absolute_percentage_error: 100.0000
```

```
[17]: {'mode': 'holdout',
'n_train': 625,
'n_test': 157,
'n_columns': 9,
'epochs': 100,
'test_loss': 29.02383804321289,
'test_mse': 29.02383804321289,
'test_mape': 100.0,
'year': None,
'name': 'arch1_B4_Osgd_L0.1_M0',
'architecture': 'arch1',
'batch_size': 4,
'optimizer': 'sgd',
'learning_rate': 0.1,
'momentum': 0}
```

```
[18]: auto.print_scores()
```

```
Total number of experiments: 8
The best model is:
{
    "architecture_name": "arch1",
    "batch_size": 4,
    "learning_rate": 0.1,
    "momentum": 0,
    "name": "arch1_B4_Osgd_L0.1_M0",
    "optimizer": "sgd"
}
The scores obtained:
{
    "test_loss": 29.02383804321289,
    "test_mse": 29.02383804321289
}
```

Como podemos ver en el anterior output y a lo largo del entrenamiento, parece ser que los resultados

son pésimos. No sólo el error es alto, sino que es incluso más alto que el obtenido en la regresión lineal simple que hemos aplicado en la parte grupal.

A primera vista, podría parecer que el problema es de overfitting, sin embargo esa amenaza ha sido controlada por un callback de *EarlyStopping*, como puede verse en los ficheros de `root.mlp`. Esto me deja con la única hipótesis de que quizás haya que probar más exhaustivamente los hiperparámetros, considerando más valores.