FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# Trabajo Práctico 2 - Checkpoint 1

## Grupo 21 - 'df' is not defined

[7506/9558] Organizacion de Datos
Primer cuatrimestre de 2023


| Alumno | Padron | Email |
|---|---|---|
| Martín Pata Fraile de Manterola | 106226 | mpata@fi.uba.ar |
| Tobías Emilio Serpa | 108266 | tserpa@fi.uba.ar |
| Juan Francisco Cuevas | 107963 | jcuevas@fi.uba.ar |

# 1. Objetivo

Llevamos a cabo el desarrollo de este proyecto, con el fin de aplicar de manera práctica distintos modelos de clasificación enseñados por la cátedra, en búsqueda de analizar texto en lenguaje natural para detectar los sentimientos de una manera básica, identificando si es un sentimiento positivo o negativo.

# 2. Modelo de Bayes Naïve

A la hora de trabajar con este modelo utilizamos la implementación MultinomialNB() de Bayes Naive e hicimos un Grid Search para buscar un valor de alfa conveniente. Estos fueron los resultados:

```
Best: 0.816600 using {'alpha': 1}
0.816600 (0.005379) with: {'alpha': 1}
0.816400 (0.005207) with: {'alpha': 0.1}
0.816325 (0.005169) with: {'alpha': 0.01}
0.816325 (0.005169) with: {'alpha': 0.001}
0.816325 (0.005169) with: {'alpha': 0.0001}
0.816325 (0.005169) with: {'alpha': 1e-05}
```

Como podemos observar, el mejor resultado es el primero, el cual es el predeterminado por MultinomialNB(). Al calcular las metricas de manera local obtuvimos los siguientes valores: precisión = 0.83, F1 = 0.817, Recall = 0.804 y Accuracy 0.814 Y en Kaggle obtuvimos un valor de F1 = 0.741.

# 3. Modelo Random Forest

Para random forest hicimos un random search bastante basico ya que para PLN suelen tardar mucho. Se utilizaron los siguientes parametros para la busqueda:

```
param_grid = {
    'n_estimators': [100, 200],
    'criterion': ['gini'],
    'max_depth': [15, None],
    'max_features': ['sqrt',None],
}
```

Esta búsqueda nos devolvió los siguientes valores:

```
{'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 'n_estimators': 200}
```

Con la construcción de un nuevo RF utilizando los anteriores resultados, obtuvimos un accuracy = 0.815, F1 = 0.815, Recall = 0.814 y precisión = 0.817 en nuestros datos locales, y un F1 = 0.689 en Kaggle. Con esto podemos observar que los cambios respecto al primer RF son mínimos, y que se sigue sobre-ajustando a los datos iniciales.

# 4. Modelo XGBoost

Al igual que Random Forest la búsqueda de hiperparametros para XGBoost va ser pequeña ya que si no suelen ser muy lenta. En este caso para XGBoost utilizamos dos tree_methods: exact,gpu_hist

### 4.1. XGBoost - tree_method = exact

Este primer modelo utiliza solamente la CPU para su entrenamiento y arrojó los siguientes resultados en pruebas locales: Accuracy = 0.8364, precisión = 0.8518, Recall = 0.826 y f1-score = 0.838; dando como resultado de F1 = 0.711 en Kaggle. Para este modelo se realizo un Grid Search con los siguientes valores:

```
param_grid = {'learning_rate': [0.1,0.01,0.001],
              'n_estimators': [200,300],
              'subsample': [0.5,0.8,1],
              'colsample_bytree': [0.5,0.8,1],
              "max_depth": [3,5,10]
              }
```

y el arbol definitivo fue construido con los siguientes parámetros:

```
{'colsample_bytree': 0.8,
 'learning_rate': 0.1,
 'max_depth': 10,
 'n_estimators': 300,
 'subsample': 0.5}
```

### 4.2. XGBoost - tree_method = gpu_hist

Luego utilizamos el siguiente modelo, ya que este utiliza la GPU para el entrenamiento. Esto permite un entrenamiento mucho más veloz, permitiendo un mejor manejo de los recursos del notebook de Collab. Además utilizamos un Grid Search con estos valores:

```
param_grid = {learning_rate: [0.1,0.01,0.001],
              n_estimators: [50,100,200],
              subsample: [0.3, 0.5],
              colsample_bytree: [0.3,0.5],
              max_depth: [9,12,15]
              }
```

y el árbol definitivo fue construido con los parámetros obtenidos de la búsqueda:

```
{colsample_bytree: 0.5,
 learning_rate: 0.1,
 max_depth: 12,
 n_estimators: 200,
 subsample: 0.5}
```

Fue con este segundo modelo que obtuvimos un accuracy = 0.834, F1 = 0.836, Recall = 0.825 y precisión = 0.848 en el entorno local, y un F1 = 0.711 (mismo valor que el anterior modelo). Hay que tener en cuenta que este modelo depende del conjunto de prueba, ya que es menos exacto. Por ejemplo si se dejan todos los features al vectorizar, el score daba un valor ampliamente inferior al primer modelo, es decir, es sensible al conjunto ded pruebas.

## 5. Modelos de Redes Neuronales

Para este modelo, realizamos distintas pruebas ya que no se puede hacer algo similar a un Grid Search en búsqueda de los mejores hiper-parámetros en las capas, y además este modelo era el que mejores resultados estaba produciendo.

### 5.1. Red Simple

En esta primera red, elaboramos una capa con 500 neuronas de activación ReLu, con la intención de observar como se comportaban las redes para PLN. Cabe destacar que la capa de predicción posee una activación sigmoidea, la cual da un resultado entre 0 y 1, lo cual se puede utilizar para mejorar nuestros scores (en este caso utilizamos un umbral de 0.4, valor que nos arrojaba buenos resultados en Kaggle). Con este modelo conseguimos las siguientes metricas: Accuracy = 0.8379, F1 = 0.8399, Recall = 0.8295, precisión = 0.8506 en nuestro entorno local, y en Kaggle un F1 = 0.7313

### 5.2. Grid Search de epoch y batchsize

Para la siguiente red hicimos un Grid Search con el objetivo de conseguir una optima cantidad de batch_size, para ello utilizamos los siguientes parámetros:

```
param_grid = {
    'epochs': [20],
    'batch_size' : [16,32,64],
    'callbacks' : [tf.keras.callbacks.EarlyStopping("accuracy",patience=5)]
}

modelo_cv = KerasClassifier(build_fn=create_model)
grid = GridSearchCV(estimator=modelo_cv, param_grid=param_grid,n_jobs=-1,cv=4)
```

Esta red es similar a la red anterior solo que la cantidad de neuronas estaba determinada por la cantidad de features de la vectorización dividido 2(EL mejor caso nos dio con 1000 features)

El mejor batch_size fue de 64 y algo importante que notamos es que a partir de 10 epochs el conjunto estaba overfiteando

### 5.3. Mejor Red

Después de probar varias arquitectura con mas capas de ReLu y capas Dropout para evetira el overfitting, la mejor red fue igual a la de Grid Search:

```
d_in = x_train.shape[1]
modelo= keras.Sequential([
keras.layers.Dense((d_in/2), activation='relu'),
keras.layers.Dense(1, activation='sigmoid')])

modelo.compile(optimizer="adam",loss='binary_crossentropy',metrics=['accuracy'])

red=modelo.fit(x_train,y_train,epochs=5)
```

Obtuvimos en testing local un accuracy es: 0.8368, f1 de: 0.847, un recall de: 0.795 y la precisión es: 0.907. Podemos notar que con el umbral mayor a 0.4 estamos prediciendo mejor los positivos que los negativos.

## 6. Voting

Para la parte de ensamblado decidimos optar por un voting, el cual nos parece que es mejor en comparación con otros ensambles como stacking. Los modelos que decidimos usar para armarlo son los siguientes:

```
rf = RandomForestClassifier(n_estimators = 200)
bayes = MultinomialNB()
xgb = XGBClassifier(n_estimators = 300, colsample_bytree = 0.8,
            learning_rate = 0.1, max_depth=10, subsample=0.5)
```

Con estos modelos pudimos conseguir localmente un accuracy = 0.837, un f1-score = 0.839, un recall = 0.828 y una precisión = 0.851. Estos resultados obtenidos la verdad que fueron bastantes buenos, algo que era de esperar ya que para la creación de este voting se utilizaron los mejores parámetros de los 3 modelos elegidos.