play-store

July 2, 2020

Aloisi Giacomo (giacomo.aloisi@studio.unibo.it) Mat 0000832933

1 Progetto di DI - Google Play Store: Category Classification

Questo progetto prende in esame un dataset ottenuto dal Google Play Store, dove sono state collezionate le informazioni riguardanti piu' di 10.000 app.

Lo scopo del progetto e' quello di utilizzare semplici tecniche di NLP per classificare una app nella sua categoria (eg. Game, Social, Art & Design, etc…) a partire dal suo nome.

Ho trovato il dataset su Kaggle ed e' consultabile qui.

1.1 Descrizione del problema e analisi esplorativa

Si deve realizzare un modello che, dato il nome di una app, la classifichi in base alla sua categoria tra le varie disponibili (eg. Game, Social, Art & Design, etc…)

Per prima cosa, importiamo le librerie che ci serviranno.

[ ]:
```
import os

import numpy as np
import pandas as pd
import sklearn as skl
import tensorflow.keras as ks
import wordcloud
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
%matplotlib inline

from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Perceptron, LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report
```

```
from sklearn.metrics import precision_score, recall_score, f1_score,␣
 ↪accuracy_score
from sklearn.base import BaseEstimator, ClassifierMixin

from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

nltk.download("punkt")
nltk.download('averaged_perceptron_tagger')
nltk.download("stopwords")
nltk.download("wordnet")
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm

[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data…
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

[ ]: True

1.1.1 Caricamento e pulizia del dataset

Carichiamo i dati dal csv `googleplaystore.csv` sulla repo GitHub come un dataframe pandas e diamo un'occhiata alla sua shape e alle prime 5 righe.

[ ]:
```
apps_data = pd.read_csv("https://raw.githubusercontent.com/GiackAloZ/
 ↪di-playstore/master/data/googleplaystore.csv")
print(apps_data.shape)
apps_data.head(5)
```

```
(10841, 13)
```

[ ]:
```
                                                 App  ...   Android Ver
0     Photo Editor & Candy Camera & Grid & ScrapBook  ...  4.0.3 and up
1                                Coloring book moana  ...  4.0.3 and up
2  U Launcher Lite – FREE Live Cool Themes, Hide …  …  4.0.3 and up
3                              Sketch - Draw & Paint  ...    4.2 and up
4              Pixel Draw - Number Art Coloring Book  ...    4.4 and up
```

```
[5 rows x 13 columns]
```

Ci sono numerose informazioni in questo dataset, ma le colonne che andremo a considerare sono due:

- `App`, ovvero i nomi delle app
- `Category`, la categoria a cui appartiene ogni app

Andiamo a osservare quali possono essere le categorie a cui un'applicazione puo' appartenere e la loro frequenza nel dataset.

[ ]:
```
apps_data["Category"].value_counts()
```

[ ]:
```
FAMILY                 1972
GAME                   1144
TOOLS                   843
MEDICAL                 463
BUSINESS                460
PRODUCTIVITY            424
PERSONALIZATION         392
COMMUNICATION           387
SPORTS                  384
LIFESTYLE               382
FINANCE                 366
HEALTH_AND_FITNESS      341
PHOTOGRAPHY             335
SOCIAL                  295
NEWS_AND_MAGAZINES      283
SHOPPING                260
TRAVEL_AND_LOCAL        258
DATING                  234
BOOKS_AND_REFERENCE     231
VIDEO_PLAYERS           175
EDUCATION               156
ENTERTAINMENT           149
MAPS_AND_NAVIGATION     137
FOOD_AND_DRINK          127
HOUSE_AND_HOME           88
LIBRARIES_AND_DEMO       85
AUTO_AND_VEHICLES        85
WEATHER                  82
ART_AND_DESIGN           65
EVENTS                   64
COMICS                   60
PARENTING                60
BEAUTY                   53
1.9                       1
Name: Category, dtype: int64
```

C'e' una categoria che probabilmente e' stata inserita per errore, cioe' la categoria identificata con il nome "1.9", perche' ha una sola occorrenza. Andiamo a vedere di cosa si tratta.

```
[ ]: wrong_data = apps_data[apps_data["Category"] == "1.9"]
     wrong_data
```

```
[ ]:                                         App Category  …  Current Ver
     Android Ver
     10472  Life Made WI-Fi Touchscreen Photo Frame      1.9  …   4.0 and up
     NaN

     [1 rows x 13 columns]
```

Sembrerebbe un dato errato, infatti ha tutte le colonne sbagliate o shiftate. Possiamo rimuoverla dal dataset.

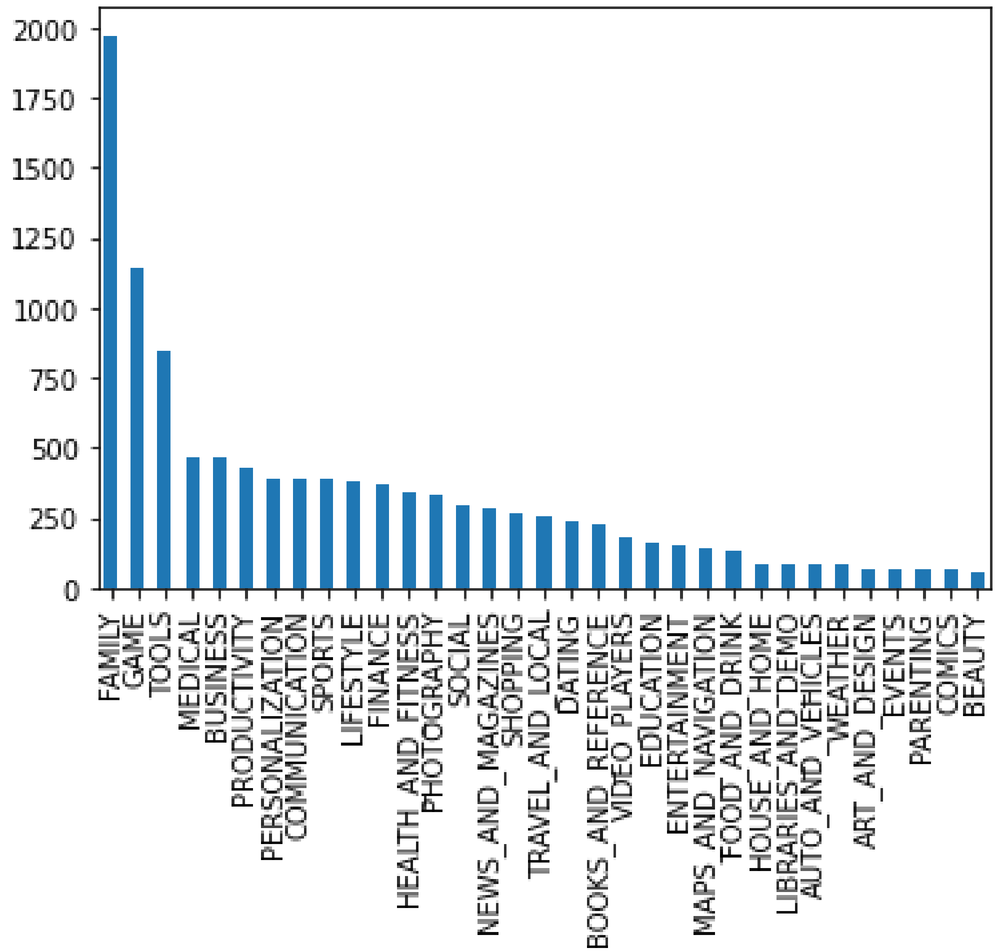
```
[ ]: apps_data = apps_data[~apps_data.index.isin(wrong_data.index)].reset_index()
```

1.1.2 Analisi esplorativa delle feature utilizzate

Andiamo ora a visualizzare le categorie in un grafico a barre e in un grafico a torta.

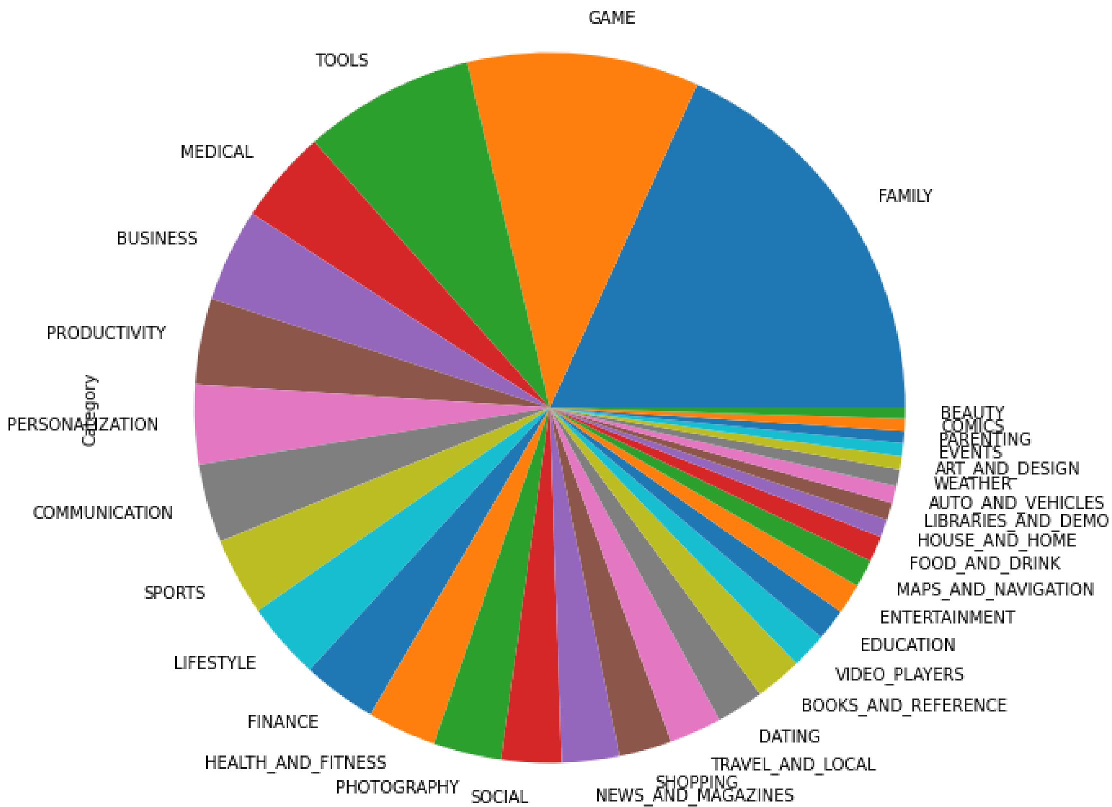
```
[ ]: apps_data["Category"].value_counts().plot.bar()
```

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb2193e92b0>
```



[ ]:
```
plt.figure(figsize=(10,10))
apps_data["Category"].value_counts().plot.pie()
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb2193b9f28>



Notiamo che le categorie spaziano da molto popolate (eg. FAMILY o GAME) a scarsamente popolate (eg. COMICS o BEAUTY). Questo potrebbe causare problemi in fase di addestramento, siccome le classi sono leggermente sbilanciate. Tuttavia, lo sbilanciamento sembra accettabile.

Vogliamo ora farci un'idea di come sono fatti i nomi delle app. Per fare cio', andiamo a concatenare tutti i nomi in una string `text` per poi creare una `cloudword` di tutti i nomi delle app in modo da osservare a colpo d'occhio quali sono le parole che compaiono piu' spesso.

[ ]:
```
text = " ".join(apps_data["App"])
```

[ ]:
```
wc = wordcloud.WordCloud(
    width=800, height=800,
    background_color="white", max_words=200
).generate(text)
```

[ ]:
```
plt.figure(figsize=(10,10))
plt.imshow(wc)
```

[ ]: `<matplotlib.image.AxesImage at 0x7fb217378390>`

Notiamo subito che le parole piu' utilizzate sono quelle che ci potevamo aspettare, come "App" o "Mobile" oppure anche "Free".

1.1.3 Correlazione tra categoria di appartenenza e nomi delle app

Andiamo ora a ottenere le top 5 parole piu' frequenti per ogni categoria. Per farlo, raggruppiamo il dataset per categoria e concateniamo i nomi delle app.

[ ]:
```
names_category = apps_data.groupby("Category")["App"].agg(" ".join)
names_category.head(5)
```

[ ]:
```
Category
ART_AND_DESIGN          Photo Editor & Candy Camera & Grid & ScrapBook…
AUTO_AND_VEHICLES       Monster Truck Stunt 3D 2019 Real Tractor Farmi…
```

```
BEAUTY                      Hush - Beauty for Everyone ipsy: Makeup, Beaut…
BOOKS_AND_REFERENCE         Wattpad   Free Books E-Book Read - Read Book f…
BUSINESS                    Visual Voicemail by MetroPCS Indeed Job Search…
Name: App, dtype: object
```

Poi andiamo a contare le frequenze delle parole di ogni categoria in una BagOfWords utilizzando `CountVectorizer`

[ ]:
```
vect = CountVectorizer()
bow = vect.fit_transform(names_category)
bow.shape
```

[ ]:
```
(33, 8714)
```

Ok, il `CountVectorizer` ha rilevato 8714 diverse parole. Ora andiamo a ordinarle per frequenza definendo una funzione che ci restituisce le k parole piu' frequenti in una categoria.

[ ]:
```
def most_k_freq(cat_freqs, vect, k=1):
    word_freqs = [(word, cat_freqs[0, i]) for word, i in vect.vocabulary_.
 ↪items()]
    word_freqs_sorted = sorted(word_freqs, key=lambda x: x[1], reverse=True)
    return word_freqs_sorted[:k]
```

Quindi per ogni riga della BagOfWords andiamo ad estrarre le k parole piu' frequenti. Controlliamo stampando la parole piu' frequenti della prima categoria. Ogni elemento della lista `top_words` e' una tupla con (`word`, `frequency`).

[ ]:
```
k = 5
top_words = [most_k_freq(row, vect, k) for row in bow]
print(top_words[0])
```

```
[('coloring', 10), ('photo', 9), ('theme', 7), ('editor', 6), ('book', 6)]
```

Mettiamo il risultato in un dataframe, usando un `MultiIndex` per suddividerlo meglio. Per farlo, dobbiamo prima *flattare* la lista di liste di tuple `top_words` per renderla una lista di liste.

[ ]:
```
top_words_flatten = [[x for y in cat for x in y] for cat in top_words]

ranking_labels = np.arange(k) + 1
word_labels = ["word", "freq"]

top_words_df = pd.DataFrame(
    top_words_flatten,
    index=names_category.index,
    columns=pd.MultiIndex.from_product(
        [ranking_labels, word_labels], names=["ranking", "word_freq"]
    )
)
top_words_df.head(5)
```

[ ]:
```
ranking                          1             2        …       4          5
word_freq                      word freq    word freq   …    word freq  word freq
Category                                                …
ART_AND_DESIGN            coloring   10   photo     9   …  editor    6  book    6
AUTO_AND_VEHICLES              car   13    used    11   …     for    6   the    6
BEAUTY                      beauty   18  camera    13   …    step   10   for    6
BOOKS_AND_REFERENCE     dictionary   22    free    19   …     for   16    al   13
BUSINESS                    mobile   32      cv    26   …    free   22   app   22

[5 rows x 10 columns]
```

Vediamo, ad esmpio, che la parola piu' frequente della categoria *ART & DESIGN* e' "coloring"

Andiamo a guardare le statistiche aggregate del dataframe appena creato.

[ ]:
```
top_words_df.describe()
```

[ ]:
```
ranking             1            2           3           4           5
word_freq        freq         freq        freq        freq        freq
count       33.000000    33.000000   33.000000   33.000000   33.000000
mean        50.909091    35.090909   27.454545   22.757576   20.121212
std         35.718836    25.779551   20.032927   16.198052   15.280211
min         10.000000     7.000000    7.000000    5.000000    5.000000
25%         22.000000    16.000000   13.000000   10.000000    9.000000
50%         39.000000    30.000000   24.000000   17.000000   14.000000
75%         67.000000    43.000000   34.000000   29.000000   26.000000
max        138.000000   105.000000   90.000000   73.000000   69.000000
```

Vediamo come la media della parola piu' frequente per ogni categoria e' di circa 50, mentre gia' alla 5a la frequenza media e' di circa 20.

Questo nota il fatto che i nomi delle app della stessa categoria sono piu' o meno simili tra loro, o comunque hanno dei termini ricorrenti. Quindi, abbiamo una certa correlazione tra categoria della app e nome di essa.

1.2 Preprocessing, tokenizzazione ed estrazione delle features dai nomi delle app

Uno dei passi fondamentali in un problema di NLP e' l'estrazione delle features.

Ci sono vari modi per estrarre delle feature dal testo. Si potrebbe procedere con l'utilizzo di un modello booleano per la trasformazione di un testo in vettore booleano, ma si e' preferito usare tecniche **VSM** (vectro space model) per rappresentare il testo in vettori di numeri reali.

Per questa analisi, si e' deciso di utilizzare come informazione principale la frequenza delle diverse parole nei nomi delle app. Per fare cio', si utilizza una tecnica di conteggio delle frequenze delle parole del singolo sample in relazione con le frequenze di tutti i sample, cosi' da ottenere la cosidetta **term frequency-inverse document frequency** (TF-IDF). Usiamo un *transformer* di *scikit-learn* che permette di trasformare una sequenza di testi in una rappresentazione vettoriale usando questa tecninca (`TfidfVectorizer`)

```
[ ]: vect = TfidfVectorizer()
     tokenized_app_names = vect.fit_transform(apps_data["App"])
     print(f"Number of tokens : {len(vect.get_feature_names())}")
```

```
Number of tokens : 8714
```

Usando il tokenizer di default, abbiamo 8714 token differenti.

Possiamo ridurre un po' il numero di token anche senza cambiare il tokenizer, ma solo togliendo quei token che compaiono solo una volta in tutti in nomi delle app. Per farlo, impostiamo il parametro `min_df` (minimum document frequency) pari a 2.

```
[ ]: vect_min_2 = TfidfVectorizer(min_df=2)
     tokenized_app_names_min_2 = vect_min_2.fit_transform(apps_data["App"])
     print(f"Number of tokens : {len(vect_min_2.get_feature_names())}")
```

```
Number of tokens : 3583
```

Gia' togliendo i token che compaiono solo una volta riduciamo abbondantemente il numero di token da 8714 a 3583 quindi di oltre 2 volte.

Andiamo ora a provare lo stemming e la lemmatizzazione. Nelle due funzioni filtriamo anche le stepwords considerate tali dalla libreria `nltk` e consideriamo solo i token che sono alfabetici.

NB: Si presuppone che la maggior parte del testo nella app sia in lingua inglese. In realta', nel dataset compaiono anche altre lingue, come vedremo piu' tardi.

```
[ ]: def get_tokens_no_stopwords_alpha(texts):
         tokens = nltk.tokenize.word_tokenize(texts)
         #token filtering (not stopword and alphabetic)
         return {token for token in tokens
                       if token not in nltk.corpus.stopwords.words("english")
                          and token.isalpha()}
```

```
[ ]: def tokenizer_stem(app_names):
         tokens = get_tokens_no_stopwords_alpha(app_names)
         #stemming
         stemmer = nltk.stem.PorterStemmer()
         stemmed_tokens = {stemmer.stem(token) for token in tokens}
         #one-char token removal
         return [token for token in stemmed_tokens]
```

```
[ ]: def tokenizer_lemm(app_names):
         tokens = get_tokens_no_stopwords_alpha(app_names)
         #lemmatizzazione
         lemmatizer = nltk.wordnet.WordNetLemmatizer()
         lemmatized_tokens = {lemmatizer.lemmatize(token) for token in tokens}
         #one-char token removal
         return [token for token in lemmatized_tokens]
```

```
[ ]: vect_stem = TfidfVectorizer(min_df=2, tokenizer=tokenizer_stem)
tokenized_app_names_stem = vect_stem.fit_transform(apps_data["App"])
print(f"Number of tokens (stem): {len(vect_stem.get_feature_names())}")

vect_lemm = TfidfVectorizer(min_df=2, tokenizer=tokenizer_lemm)
tokenized_app_names_lemm = vect_lemm.fit_transform(apps_data["App"])
print(f"Number of tokens (lem): {len(vect_lemm.get_feature_names())}")
```

```
Number of tokens (stem): 2995
Number of tokens (lem): 3114
```

Le feature diminuiscono, ma non significamente. Probabilmente per via del fatto che i nomi delle app sono per lo piu' nomi propri (es. Instagram, Skype, etc…) e contengono poche parole utilizzate normalmente in documenti testuali.

Andiamo a confrontare la lunghezza media dei token prima e dopo lo stemming.

```
[ ]: token_lengths = pd.Series(map(len, vect_min_2.get_feature_names()))
stem_token_lengths = pd.Series(map(len, vect_stem.get_feature_names()))

print(f"Mean length no stemming: {token_lengths.mean():.2f}")
print(f"Mean length with stemming: {stem_token_lengths.mean():.2f}")
```

```
Mean length no stemming: 5.77
Mean length with stemming: 5.34
```

Sono entrame molto simili, quindi significa che, anche dopo lo stemming, le parole non vengono ridotte troppo in lunghezza.

NB: Siccome sia lo stemming che la lemmatizzazione non sono riuscite a ridurre abbastanza lo spazio delle features, si e' deciso, in primo approcio, di non utilizzarle. Successivamente, quando si fara' un'ultima fase di tuning sui modelli piu' promettenti, si provera' anche il loro utilizzo, per vedere quanto incidano sull'accuratezza del modello.

1.3 Generazione di diversi modelli di learning

Si procede con la generazione di diversi modelli di *supervised learning* per la predizione della categoria dato il nome dell'app. Possiamo dire che il nostro e' un problema di **classificazione multipla**, per cui ho usato alcuni dei possibili modelli utili ad affrontare questo tipo di problemi.

1.3.1 Divisione train e test set

Si procede andando a selezionare le features dal dataset (nel nostro caso solo una, il nome delle app) e il *target* della classificazione (cioe' le categorie delle app).

Si divide il dataset in *train* e *test* set, usando il 90% del dataset come train set e il restante 10% come test set. Nel fare cio', si dividono i dati usando la stratificazione per classe, in modo che le classi siano bilanciate allo stesso modo sia nel train set che nel test set. Questo ci permette di ottenere una misura dell'accuratezza piu' precisa quando si andra' a testare il modello sul test set.

NB: non viene menzionato il validation set perche' tutta la parte sperimentale di scelta degli iperparametri nei vari modelli viene fatta attraverso *cross-fold validation* , per cui il validation set viene ogni volta preso come parte del train set. Si noti anche che il test set, una volta diviso, non viene mai usato come parametro del training o della validation per trovare i migliori iperparametri.

[ ]:
```
X = apps_data["App"]
y = apps_data["Category"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=1/10,
    stratify=y,
    random_state=42
)
print(X_train.shape, X_test.shape)
```

```
(9756,) (1084,)
```

Abbiamo quindi 9756 samples dedicati al training, contro 1084 per testare il modello

Segue una lista di modelli, i quali vengono addestrati sul train set usando una *K-cross-fold validation* stratificata (riduce il bias dato da un training con classi poco distribuite) insieme a una grid-search per trovare i migliori iperparametri per ogni modello.

I modelli presi in esame sono:

- Perceptron
- Regressione logistica
- Support-Vector Machine per classificazione (con vari kernel, anche lineari)
- Multi-layer Perceptron (con due layer nascosti)
- Rete neurale con Keras (con due layer nascosti)

Ogni modello viene successivamente testato sul test set, calcolandone l'accuratezza. Vengono anche stampati gli iperparametri che hanno avuto un'accuratezza media migliore nella varie fold di validation. Queste informazione vengono salvate in due dizionari: `scores` e `best_params`.

NB: in tutti i modelli si utilizza il `TfidVectorizer` come calcolo del VMS con parametro `min_df=2`. Per ogni modello si imposta un `random_state` per rendere riproducibili i risultati.

[ ]:
```
scores = {}
best_params = {}
```

1.3.2 Perceptron

[ ]:
```
param_perceptron = {
    "perc__penalty": ["l1", "l2"],
    "perc__alpha": np.logspace(-7, -2, num=6)
}

model_perceptron = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("perc", Perceptron(random_state=42))
```

```
])

search_perceptron = GridSearchCV(
    model_perceptron,
    param_grid=param_perceptron,
    cv=StratifiedKFold(n_splits=5),
    verbose=2,
    n_jobs=3
)
search_perceptron.fit(X_train, y_train)
scores["perceptron"] = search_perceptron.score(X_test, y_test)
scores["perceptron"]
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
```

```
[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=3)]: Done  35 tasks      | elapsed:    9.6s
[Parallel(n_jobs=3)]: Done  60 out of  60 | elapsed:   13.8s finished
```

[ ]: 0.46863468634686345

[ ]:
```
best_params["perceptron"] = search_perceptron.best_params_
best_params["perceptron"]
```

[ ]: {'perc__alpha': 1e-07, 'perc__penalty': 'l1'}

1.3.3 Logistic regression

[ ]:
```
param_logistic = [
    {
        "logreg__penalty": ["l1", "l2"],
        "logreg__C": [1, 10]
    }
]

model_logistic = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("logreg", LogisticRegression(random_state=42, solver="saga",␣
 ↪multi_class="multinomial"))
])

search_logistic = GridSearchCV(
    model_logistic,
    param_grid=param_logistic,
    cv=StratifiedKFold(n_splits=5),
    verbose=2,
    n_jobs=3
```

```
)
search_logistic.fit(X_train, y_train)
scores["logreg"] = search_logistic.score(X_test, y_test)
scores["logreg"]
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
```

```
[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=3)]: Done  20 out of  20 | elapsed:  2.5min finished
```

[ ]: 0.5673431734317343

[ ]:
```
best_params["logreg"] = search_logistic.best_params_
best_params["logreg"]
```

[ ]: {'logreg__C': 10, 'logreg__penalty': 'l2'}

1.3.4 SVC

[ ]:
```
param_svc = [
    {
        "svc__gamma" : [0.1, 1],
        "svc__C" : [1, 10],
        "svc__kernel" : ["rbf"]
    },
    {
        "svc__gamma" : [0.1, 1],
        "svc__C" : [1, 10],
        "svc__kernel" : ["poly"],
        "svc__degree": [3, 5]
    },
    {
        "svc__C" : [1, 10],
        "svc__kernel" : ["linear"]
    },
]

model_scv = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("svc", SVC(random_state=42))
])

search_svc = GridSearchCV(
    model_scv,
    param_grid=param_svc,
    cv=StratifiedKFold(n_splits=5),
    verbose=2,
    n_jobs=3
```

```
)
search_svc.fit(X_train, y_train)
scores["svc"] = search_svc.score(X_test, y_test)
scores["svc"]
```

```
Fitting 5 folds for each of 14 candidates, totalling 70 fits

[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=3)]: Done  35 tasks      | elapsed:  3.9min
[Parallel(n_jobs=3)]: Done  70 out of  70 | elapsed:  7.0min finished
```

[ ]: 0.5701107011070111

[ ]:
```
best_params["svc"] = search_svc.best_params_
best_params["svc"]
```

[ ]: {'svc__C': 10, 'svc__gamma': 0.1, 'svc__kernel': 'rbf'}

1.3.5 Multi-layer perceptron

[ ]:
```
param_mlp = {
    "mlp__hidden_layer_sizes" : [(size, size) for size in np.logspace(4, 6,␣
 ↪num=3, base=2, dtype=np.int)],
    "mlp__alpha" : np.logspace(-3, -2, num=2)
}

model_mlp = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("mlp", MLPClassifier(random_state=42))
])

search_mlp = GridSearchCV(
    model_mlp,
    param_grid=param_mlp,
    cv=StratifiedKFold(n_splits=5),
    verbose=2
)
search_mlp.fit(X_train, y_train)
scores["mlp"] = search_mlp.score(X_test, y_test)
scores["mlp"]
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16) …

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
```

```
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   25.1s remaining:    0.0s

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16), total=  25.1s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16), total=  25.5s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16), total=  25.6s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16), total=  25.1s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(16, 16), total=  25.3s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32), total=  36.8s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
```

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32), total=  36.5s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32), total=  36.4s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32), total=  36.9s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(32, 32), total=  36.7s
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64), total= 1.0min
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64), total= 1.0min
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
```

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64), total= 1.1min
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64), total= 1.0min
[CV] mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.001, mlp__hidden_layer_sizes=(64, 64), total= 1.0min
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16), total=  25.8s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16), total=  25.5s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16), total=  25.6s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
```

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16), total=  25.5s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(16, 16), total=  25.6s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32), total=  36.4s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32), total=  36.5s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32), total=  35.9s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32), total=  35.8s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
```

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(32, 32), total=  36.0s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64), total=  59.1s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64), total=  59.9s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64), total= 1.1min
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64), total=  60.0s
[CV] mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64) …

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed: 20.6min finished

[CV]  mlp__alpha=0.01, mlp__hidden_layer_sizes=(64, 64), total= 1.0min

/usr/local/lib/python3.6/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:571:
```

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached and
the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

[ ]: 0.5369003690036901

[ ]:
```
best_params["mlp"] = search_mlp.best_params_
best_params["mlp"]
```

[ ]: {'mlp__alpha': 0.01, 'mlp__hidden_layer_sizes': (64, 64)}

1.3.6 Keras neural network

Per realizzare il modello Keras, si e' pensato di wrappare tutto il modello in una classe compatibile con `scklearn`. Si poteva usare anche un wrapper `KerasClassifier`, ma ho preferito non farlo, perche' ho voluto inserire dentro al modello anche un trasformatore di etichette `LabelEncoder` che trasforma le etichette in maniera che il modello Keras le accetti. In particolare, ogni categoria viene convertita in un numero intero (label) e poi passata al modello. Durante l'inferenza, la label intera viene riconvertita in stringa, cosi' da rendere del tutto trasparente la conversione all'esterno.

[ ]:
```
class KerasModel(BaseEstimator, ClassifierMixin):
    def __init__(self,
                 hidden_layer_sizes=(64,64),
                 reg_rate=1e-4,
                 epochs=20,
                 batch_size=64):
        # label encoder (string -> int)
        self._encoder = LabelEncoder()
        self.hidden_layer_sizes = hidden_layer_sizes
        self.reg_rate = reg_rate
        self.epochs = epochs
        self.batch_size = batch_size
        super().__init__()

    def fit(self, X, y, **kargs):
        # encode string classes to integers
        y_encoded = self._encoder.fit_transform(y)

        # keras model with multiple hidden layers with "relu" activation
        # and one final layer with "softmax" activation for mutli-class␣
↪classification
        self._model = ks.models.Sequential([
            ks.Input(shape=X.shape[1:2])] + [     # input layer with shape␣
↪equal to number of classes
            ks.layers.Dense(size,
                            activation="relu",    # ReLU activation and weights␣
↪L2 regularization
```

```
                                kernel_regularizer=ks.regularizers.l2(self.
↪reg_rate)) for size in self.hidden_layer_sizes
            ] + [ks.layers.Dense(self._encoder.classes_.size,␣
↪activation="softmax")]    # output layer proba
        )

        self._model.compile(
            optimizer="adam",
            loss="sparse_categorical_crossentropy",
            metrics=["acc"]
        )

        # show only epoch number while fitting
        self._model.fit(X.toarray(), y_encoded, batch_size=self.batch_size,␣
↪epochs=self.epochs, verbose=0, **kargs)
        return self

    def predict(self, X):
        y_pred = self._model.predict_classes(X.toarray())
        # inverse transform classes (int -> string)
        return self._encoder.inverse_transform(y_pred)
```

[ ]:
```
param_keras = {
    "keras__hidden_layer_sizes": [(size, size) for size in np.logspace(4, 6,␣
↪num=3, base=2, dtype=np.int)],
    "keras__reg_rate": np.logspace(-3, -2, num=2)
}

model_keras = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("keras", KerasModel(epochs=20, batch_size=256))
])

search_keras = GridSearchCV(
    model_keras,
    param_grid=param_keras,
    cv=StratifiedKFold(n_splits=5),
    verbose=2
)
search_keras.fit(X_train, y_train)
scores["keras"] = search_keras.score(X_test, y_test)
scores["keras"]
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001 …

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
```

```
WARNING:tensorflow:From <ipython-input-36-8ef66e68ad56>:40:
Sequential.predict_classes (from tensorflow.python.keras.engine.sequential) is
deprecated and will be removed after 2021-01-01.
Instructions for updating:
Please use instead:* `np.argmax(model.predict(x), axis=-1)`,   if your model
does multi-class classification   (e.g. if it uses a `softmax` last-layer
activation).* `(model.predict(x) > 0.5).astype("int32")`,   if your model does
binary classification   (e.g. if it uses a `sigmoid` last-layer activation).
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001, total=   4.9s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001 …

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    4.9s remaining:    0.0s

[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001, total=   4.1s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001, total=   3.9s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001, total=   4.0s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.001, total=   4.6s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01, total=   4.2s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01, total=   4.0s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01, total=   3.5s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01, total=   4.0s
[CV] keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(16, 16), keras__reg_rate=0.01, total=   4.1s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001, total=   4.4s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001, total=   4.5s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001, total=   4.2s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001, total=   4.5s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.001, total=   4.5s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01, total=   4.5s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01, total=   4.5s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01, total=   4.2s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01, total=   4.6s
[CV] keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01 …
```

```
[CV]  keras__hidden_layer_sizes=(32, 32), keras__reg_rate=0.01, total=   4.9s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001, total=   9.1s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001, total=   5.7s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001, total=   5.5s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001, total=   5.6s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.001, total=   5.6s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01, total=   5.6s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01, total=   6.0s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01, total=   5.3s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01, total=   5.5s
[CV] keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01 …
[CV]  keras__hidden_layer_sizes=(64, 64), keras__reg_rate=0.01, total=   5.5s

[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:  2.4min finished
```

[ ]: 0.540590405904059

[ ]:
```
best_params["keras"] = search_keras.best_params_
best_params["keras"]
```

[ ]: {'keras__hidden_layer_sizes': (64, 64), 'keras__reg_rate': 0.001}

1.4 Scelta dei modelli migliori e fine tuning

Vediamo ora le performance di tutti i modelli che sono stati addestrati

[ ]:
```
scores_df = pd.DataFrame([[score] for _, score in scores.items()], index=scores.
 ↪keys(), columns=["accuracy"])
scores_df
```

[ ]:

	accuracy
perceptron	0.468635
logreg	0.567343
svc	0.570111
keras	0.540590
mlp	0.536900

Come possiamo vedere, la `LogisticRegression` e il modello `SVC` sono i due che hanno un'accuratezza piu' alta del resto dei modelli.

1.4.1 Fine tuning dei migliori due modelli

Andiamo ora a fare un po' di *fine tuning* dei due modelli migliori. Proviamo stemming e lemmatizzazione, inoltre introduciamo anche la possibilita' di considerare *n-grammi* di lunghezza due o tre per il calcolo del TF-IDF.

Vediamo i risultati della cross validation di entrambi.

[ ]:
```
pd.DataFrame(search_logistic.cv_results_).sort_values("mean_test_score",␣
↪ascending=False).head(5)
```

[ ]:
```
   mean_fit_time  std_fit_time  …  std_test_score  rank_test_score
3       3.238283      0.128604  …        0.007979                1
2      61.717902      7.873328  …        0.008331                2
0      13.772558      0.429080  …        0.004308                3
1       1.330408      0.083292  …        0.009673                4

[4 rows x 15 columns]
```

[ ]:
```
pd.DataFrame(search_svc.cv_results_).sort_values("mean_test_score",␣
↪ascending=False).head(5)
```

[ ]:
```
    mean_fit_time  std_fit_time  …  std_test_score  rank_test_score
2       18.385293      0.394898  …        0.008076                1
12      10.803240      0.198507  …        0.005594                2
3       25.487276      0.184415  …        0.005184                3
1       24.527414      0.281655  …        0.003460                4
13      10.498846      0.190637  …        0.008663                5

[5 rows x 17 columns]
```

Per la `LogisticRegression` usiamo sempre la regolarizzazione *ridge* (l2) che sembra dare i migliori risultati. Per il `SVC` usiamo il kernel `rbf` per lo stesso motivo.

[ ]:
```
scores_ft = {}
best_params_ft = {}
```

[ ]:
```
param_logistic = {
    "vect__tokenizer": [None, tokenizer_lemm, tokenizer_stem],
    "vect__ngram_range": [(1,1), (1,2)],
    "logreg__C": np.logspace(0, 1, num=3)
}

model_logistic = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("logreg", LogisticRegression(random_state=42, solver="saga",␣
↪multi_class="multinomial", penalty="l2"))
])
```

```
search_logistic = GridSearchCV(
    model_logistic,
    param_grid=param_logistic,
    cv=StratifiedKFold(n_splits=5),
    verbose=2,
    n_jobs=3
)
search_logistic.fit(X_train, y_train)
scores_ft["logreg"] = search_logistic.score(X_test, y_test)
scores_ft["logreg"]
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
```

```
[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=3)]: Done  35 tasks      | elapsed:  2.3min
[Parallel(n_jobs=3)]: Done  90 out of  90 | elapsed:  6.9min finished
```

[ ]: 0.5728782287822878

[ ]:
```
best_params_ft["logreg"] = search_logistic.best_params_
best_params_ft["logreg"]
```

[ ]:
```
{'logreg__C': 3.1622776601683795,
 'vect__ngram_range': (1, 2),
 'vect__tokenizer': None}
```

[ ]:
```
param_svc = {
    "vect__tokenizer": [None, tokenizer_lemm, tokenizer_stem],
    "vect__ngram_range": [(1,1), (1,2), (1,3)]
}

model_scv = Pipeline([
    ("vect", TfidfVectorizer(min_df=2)),
    ("svc", SVC(random_state=42, C=10, gamma=0.1, kernel="rbf"))
])

search_svc = GridSearchCV(
    model_scv,
    param_grid=param_svc,
    cv=StratifiedKFold(n_splits=5),
    verbose=2,
    n_jobs=3
)
search_svc.fit(X_train, y_train)
scores_ft["svc"] = search_svc.score(X_test, y_test)
scores_ft["svc"]
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=3)]: Using backend LokyBackend with 3 concurrent workers.
[Parallel(n_jobs=3)]: Done  35 tasks      | elapsed:  6.4min
[Parallel(n_jobs=3)]: Done  45 out of  45 | elapsed:  8.7min finished
```

[ ]:
```
0.5701107011070111
```

[ ]:
```
best_params_ft["svc"] = search_svc.best_params_
best_params_ft["svc"]
```

[ ]:
```
{'vect__ngram_range': (1, 1), 'vect__tokenizer': None}
```

Otteniamo che la lemmatizzazione o lo stemming non aiutano. Invece, aggiungere i bigrammi nel modello di regressione logistica aumente la accuratezza.

Ricontrolliamo le performance dei due modelli scelti.

[ ]:
```
scores_ft_df = pd.DataFrame([[score] for _, score in scores_ft.items()],␣
 ↪index=scores_ft.keys(), columns=["accuracy"])
scores_ft_df
```

[ ]:

	accuracy
logreg	0.572878
svc	0.570111

1.5 Valutazione modelli

Prendiamo il modello `LogistiRegression` per valutarlo un po' piu' nello specifico. In particolare, vorremo sapere informazioni sui coefficenti delle varie parole, ovvero i pesi del modello.

Come si relazionano i vari coefficenti delle parole con le classi da predirre? Creiamo un dataframe che metta in relazione le classi con i coefficenti delle parole. In particolare, ogni cella del dataframe avra', come intersezione di classe e parola, il peso che quella parola ha per quella particolare classe.

Il peso e' il coefficente di correlazione tra la parola e la classe (positivo -> correlazione alta diretta, negativo -> correlazione alta inversa, zero -> poca o nessuna correlazione).

[ ]:
```
model_logistic = search_logistic.best_estimator_
coeffs_df = pd.DataFrame(
    model_logistic.named_steps["logreg"].coef_,
    index=model_logistic.named_steps["logreg"].classes_,
    columns=model_logistic.named_steps["vect"].get_feature_names())
coeffs_df.head(5)
```

[ ]:

	000	000 forums	…	željezničar	
ART_AND_DESIGN	-0.065572	-0.011764	…	-0.015781	-0.013895
AUTO_AND_VEHICLES	0.697847	-0.021327	…	-0.021865	-0.018973
BEAUTY	-0.028150	-0.009568	…	-0.012505	-0.011205
BOOKS_AND_REFERENCE	-0.112937	-0.030813	…	-0.041016	-0.034411
BUSINESS	-0.152068	-0.056275	…	-0.101700	-0.089059

```
[5 rows x 6702 columns]
```

Notiamo anche che ci sono lingue differenti dall'inglese nel dataset.

Vediamo, ad esempio, i coefficenti delle parole "calculator" e "flashlight" rispettivamente nelle classi "PRODUCTIVITY" e "GAME"

```
[ ]: coeffs_df.loc["PRODUCTIVITY", "calculator"]
```

```
[ ]: 2.4275511321940764
```

```
[ ]: coeffs_df.loc["GAME", "flashlight"]
```

```
[ ]: -0.6159452637385386
```

Come si poteva immaginare, "calculator" e' positivamente correlata con "PRODUCTIVITY", il che significa che si un'applicazione ha nel suo nome la parola "calculator" e' probabile che sia appartenga alla classe "PRODUCTIVITY".

Viceversa, siccome "flashlight" e' negativamente correlata con "GAME", difficilmente un'applicazione che contiene questa parola verra' etichettata come "GAME".

Andiamo ora a mostrare, per ogni classe, la parola con coefficente piu' alto.

```
[ ]: best_coeffs_df = pd.concat([coeffs_df.idxmax(axis=1), coeffs_df.max(axis=1)],␣
     ↪axis=1)
     best_coeffs_df.columns = ["word", "coeff"]
     best_coeffs_df
```

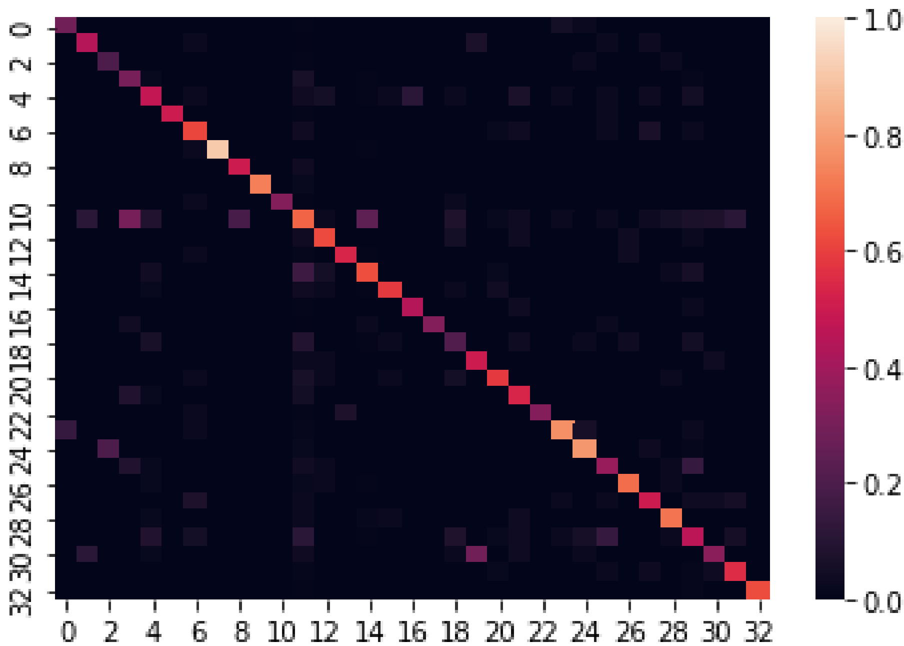
```
[ ]:                            word     coeff
     ART_AND_DESIGN          drawing  4.349131
     AUTO_AND_VEHICLES          cars  4.650402
     BEAUTY                   beauty  6.873256
     BOOKS_AND_REFERENCE  dictionary  5.046695
     BUSINESS                    job  4.187305
     COMICS                   comics  6.746328
     COMMUNICATION           browser  6.310651
     DATING                   dating  9.891073
     EDUCATION                 learn  6.498220
     ENTERTAINMENT                tv  5.846519
     EVENTS                   events  5.053975
     FAMILY                simulator  5.052634
     FINANCE                    bank  6.671597
     FOOD_AND_DRINK          recipes  5.769846
     GAME                      poker  5.029902
     HEALTH_AND_FITNESS      fitness  5.611241
     HOUSE_AND_HOME             home  4.645607
     LIBRARIES_AND_DEMO       bn pro  4.826261
```

```
LIFESTYLE                   church    4.051348
MAPS_AND_NAVIGATION           taxi    4.574222
MEDICAL                    anatomy    5.338231
NEWS_AND_MAGAZINES            news   10.098907
PARENTING                     baby    7.648491
PERSONALIZATION              theme    8.149679
PHOTOGRAPHY                 camera    7.853599
PRODUCTIVITY             microsoft    4.608133
SHOPPING                  shopping    8.114268
SOCIAL                       amino    5.113486
SPORTS                      sports    6.428454
TOOLS                   flashlight    5.185759
TRAVEL_AND_LOCAL            hotels    5.929624
VIDEO_PLAYERS                video    7.047645
WEATHER                    weather   11.030866
```

Controlliamo ora la **matrice di confusione**, che e' stata normalizzata e plottata come *heatmap*

[ ]:
```
y_pred = model_logistic.predict(X_test)
conf_logistic = confusion_matrix(y_test, y_pred)
conf_logistic = conf_logistic / conf_logistic.sum(axis=1)
sns.heatmap(conf_logistic, vmin=0, vmax=1)
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb211b0c978>
```



Per capire meglio quanto il modello sia preciso rispettivamante ad ogni classe, andiamo a stampare il `classification report`.

[ ]:
```python
print(classification_report(y_test, y_pred))
```

```
                     precision    recall  f1-score   support

      ART_AND_DESIGN       0.67      0.29      0.40         7
   AUTO_AND_VEHICLES       0.67      0.44      0.53         9
              BEAUTY       0.50      0.20      0.29         5
 BOOKS_AND_REFERENCE       0.37      0.30      0.33        23
            BUSINESS       0.52      0.48      0.50        46
              COMICS       1.00      0.50      0.67         6
       COMMUNICATION       0.65      0.62      0.63        39
              DATING       1.00      0.91      0.95        23
           EDUCATION       0.73      0.50      0.59        16
       ENTERTAINMENT       1.00      0.73      0.85        15
              EVENTS       1.00      0.33      0.50         6
              FAMILY       0.38      0.68      0.49       197
             FINANCE       0.70      0.62      0.66        37
      FOOD_AND_DRINK       0.88      0.54      0.67        13
                GAME       0.64      0.63      0.63       114
  HEALTH_AND_FITNESS       0.83      0.59      0.69        34
      HOUSE_AND_HOME       0.80      0.44      0.57         9
 LIBRARIES_AND_DEMO       1.00      0.33      0.50         9
           LIFESTYLE       0.36      0.21      0.27        38
 MAPS_AND_NAVIGATION       0.58      0.50      0.54        14
             MEDICAL       0.79      0.59      0.68        46
  NEWS_AND_MAGAZINES       0.60      0.54      0.57        28
           PARENTING       1.00      0.33      0.50         6
     PERSONALIZATION       0.83      0.77      0.80        39
         PHOTOGRAPHY       0.76      0.79      0.78        33
        PRODUCTIVITY       0.53      0.38      0.44        42
            SHOPPING       0.86      0.69      0.77        26
              SOCIAL       0.68      0.50      0.58        30
              SPORTS       0.77      0.71      0.74        38
               TOOLS       0.46      0.46      0.46        84
    TRAVEL_AND_LOCAL       0.64      0.35      0.45        26
       VIDEO_PLAYERS       0.71      0.56      0.63        18
             WEATHER       1.00      0.62      0.77         8

            accuracy                           0.57      1084
           macro avg       0.72      0.52      0.59      1084
        weighted avg       0.62      0.57      0.58      1084
```

Estraiamo dal report le classi con le migliori e le peggiori precisioni, recall e f1-score.

```
[ ]: logistic_report_df = pd.DataFrame(classification_report(y_test, y_pred,␣
     ↪output_dict=True)).T
     logistic_report_df.idxmax()
```

```
[ ]: precision       COMICS
     recall          DATING
     f1-score        DATING
     support      macro avg
     dtype: object
```

```
[ ]: logistic_report_df.idxmin()
```

```
[ ]: precision    LIFESTYLE
     recall          BEAUTY
     f1-score     LIFESTYLE
     support       accuracy
     dtype: object
```

Calcoliamo ora tutte e quattro le metriche principali delle classificazione (accuratezza, precisione, recall e f1-score) per ogni modello.

```
[ ]: def calc_acc_prec_rec_f1(y_true, y_pred):
         acc = accuracy_score(y_true, y_pred, normalize=True)
         prec = precision_score(y_test, y_pred, average="macro")
         recall = recall_score(y_test, y_pred, average="macro")
         f1 = f1_score(y_test, y_pred, average="macro")
         return {
             "accuracy": acc,
             "precision": prec,
             "recall": recall,
             "f1-score": f1
         }
```

```
[ ]: searchs = [
         ("perceptron", search_perceptron),
         ("logreg", search_logistic),
         ("svc", search_svc),
         ("mlp", search_mlp),
         ("keras", search_keras)
     ]

     models = [(name, gs.best_estimator_) for name, gs in searchs]

     metrics = [
         calc_acc_prec_rec_f1(y_test, model.predict(X_test)) for _, model in models
     ]
```

```
names = [name for name, _ in models]

model_metrics_df = pd.DataFrame(metrics, index=names)
model_metrics_df
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:1272:
UndefinedMetricWarning: Precision is 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))
```

[ ]:

	accuracy	precision	recall	f1-score
perceptron	0.468635	0.466564	0.457943	0.450979
logreg	0.572878	0.724932	0.519366	0.588017
svc	0.570111	0.697592	0.532211	0.588577
mlp	0.536900	0.603272	0.510334	0.538145
keras	0.540590	0.571511	0.444603	0.482475

Calcoliamo i range delle accuratezze per ogni modello con una confidenza del 95%.

[ ]:
```
def confidence(acc, N, Z):
    den = (2*(N+Z**2))
    var = (Z*np.sqrt(Z**2+4*N*acc-4*N*acc**2)) / den
    a = (2*N*acc+Z**2) / den
    inf = a - var
    sup = a + var
    return (inf, sup)
```

[ ]:
```
ranges = model_metrics_df["accuracy"].map(lambda acc: confidence(acc,␣
 ↪len(y_test), 1.96))    # Z=1.96 e' per la confidenza al 95%
model_metrics_df["accuracy_inf"] = ranges.map(lambda x: x[0])
model_metrics_df["accuracy_sup"] = ranges.map(lambda x: x[1])
model_metrics_df
```

[ ]:

	accuracy	precision	recall	f1-score	accuracy_inf	accuracy_sup
perceptron	0.468635	0.466564	0.457943	0.450979	0.439091	0.498400
logreg	0.572878	0.724932	0.519366	0.588017	0.543224	0.602017
svc	0.570111	0.697592	0.532211	0.588577	0.540443	0.599283
mlp	0.536900	0.603272	0.510334	0.538145	0.507138	0.566402
keras	0.540590	0.571511	0.444603	0.482475	0.510832	0.570062

Infine, controlliamo che, con una confidenza del 99%, il modello di regressione logistica non sia statisticamente equivalente a un modello randomico.

[ ]:
```
import random
def random_prediction(X):
    return random.choices(model_logistic.classes_, k=len(X))
```

[ ]:
```
def compare_confidence(acc1, acc2, N, Z):
    var_sq = acc1 * (1 - acc1) / N + acc2 * (1 - acc2) / N
    a = abs(acc1 - acc2)
    inf = a - Z * np.sqrt(var_sq)
    sup = a + Z * np.sqrt(var_sq)
    return (inf, sup)
```

[ ]:
```
compare_confidence(
    accuracy_score(y_test, random_prediction(X_test)),
    model_metrics_df.loc["logreg", "accuracy"],
    len(X_test), 2.56)
```

[ ]:
```
(0.4958027244735912, 0.5779980135337888)
```

E' vero che non sono statisticamente equivalenti, perche' il range non comprende lo zero.

1.6 Conclusioni

Mi e' piaciuto molto questo progetto, anche se una accuratezza del 60% circa non e' ottima, il fatto che sia superiore al 50% pur avendo ben 33 classi differenti mi sembra soddisfacente.

L'idea di questo progetto era quella di creare un classificatore automatico per le nuove app che approdano sul Google Play Store.

Nelle prossime celle, faccio un po' di esperimenti, dove inserirsco il nome di una app e vedo come il modello la classifica.

Enjoy :)

[ ]:
```
def predict_one(app_name):
    return model_logistic.predict([app_name])[0]
```

[ ]:
```
predict_one("A beautifil app for make up")
```

[ ]:
```
'LIFESTYLE'
```

[ ]:
```
predict_one("My flashlight pro")
```

[ ]:
```
'TOOLS'
```

[ ]:
```
predict_one("Clash of clans")
```

[ ]:
```
'GAME'
```

[ ]:
```
predict_one("Instagram 2.0")
```

[ ]:
```
'SOCIAL'
```

[ ]:
```
predict_one("App for browsing Reddit")
```

[ ]: 'NEWS_AND_MAGAZINES'

[ ]:
```
predict_one("Pro pic camera")
```

[ ]: 'PHOTOGRAPHY'

[ ]:
```
predict_one("Stock selling and trading")
```

[ ]: 'FINANCE'

[ ]:
```
predict_one("Internet speed test - faster internet in one click")
```

[ ]: 'TOOLS'

[ ]:
```
predict_one("Google Docs")
```

[ ]: 'PRODUCTIVITY'

[ ]:
```
predict_one("Subito.it buy, sell and save money")
```

[ ]: 'SHOPPING'

Non male direi :)