02 - Multigenre prediction

February 12, 2019

1 Prediction: Multigenre

This notebook explores various algorithms' ability to classify songs as pop, rap, rock or country. This notebook compares the same algorithms as the ones in Pop vs. Rap except we increase the number of genres to four.

```
In [1]: import sys
        sys.path.insert(0, "..//..//..//scripts")

        import xgboost as xgb
        import seaborn as sns
        import re
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt

        from itertools import chain
        from NonParametricClassifier import *
        from CDFClassifier import *
        from HelperFunctions import *
        from sklearn.naive_bayes import BernoulliNB, MultinomialNB
```

To decide which genres to add, we found the top four most popular genres. They are, in order, pop, rap, rock and country.

```
In [2]: df = pd.read_csv("..//..//..//..//data//Weekly_data_tokenized.csv")
        genre = []

        for unique in df.ID.unique():
            genre.append(df[df.ID == unique].iloc[0].Genre)

        genre = [x.split(",") for x in genre]
        genre = Counter(list(chain.from_iterable(genre)))
        genre = sorted(genre.items(), key = lambda x: x[1], reverse = True)

        genre[:8]
```

```
Out[2]: [('Pop', 1783),
         ('Rap', 1427),
         ('Rock', 721),
         ('Country', 692),
         ('R&;B', 661),
         ('Trap', 359),
         ('Canada', 266),
         ('Pop-Rock', 207)]
```

This time, the minimum Gini index is .125.

```
In [3]: df["Pop"] = df.apply(lambda row: create_genre(row, "pop"), axis = 1)
        df["Rap"] = df.apply(lambda row: create_genre(row, "rap"), axis = 1)
        df["Rock"] = df.apply(lambda row: create_genre(row, "rock"), axis = 1)
        df["Country"] = df.apply(lambda row: create_genre(row, "country"), axis = 1)

        df = df[["word", "ID", "Pop", "Rap", "Rock", "Country"]]

        tmp = df.groupby(["word", "Pop", "Rap", "Rock", "Country"]).count().unstack().unstack(

        gini = calculate_gini_index(tmp)
        useless_words = [x for x in gini if gini[x] <= .236]

        df = df[~df.word.isin(useless_words)]
```

We remove words with the bottom 3.2% of Gini indexes.

```
In [4]: len(useless_words) / len(df.word.unique())

Out[4]: 0.033193979933110365
```

Again, we opted for a 80-20 split between the training and validation set.

```
In [5]: np.random.seed(1)

        IDs = df.ID.unique()
        np.random.shuffle(IDs)

        train = df[df.ID.isin(IDs[:int(.8 * len(IDs))])]
        test = df[df.ID.isin(IDs[int(.8 * len(IDs)):])]
```

2 Classification by distribution comparison

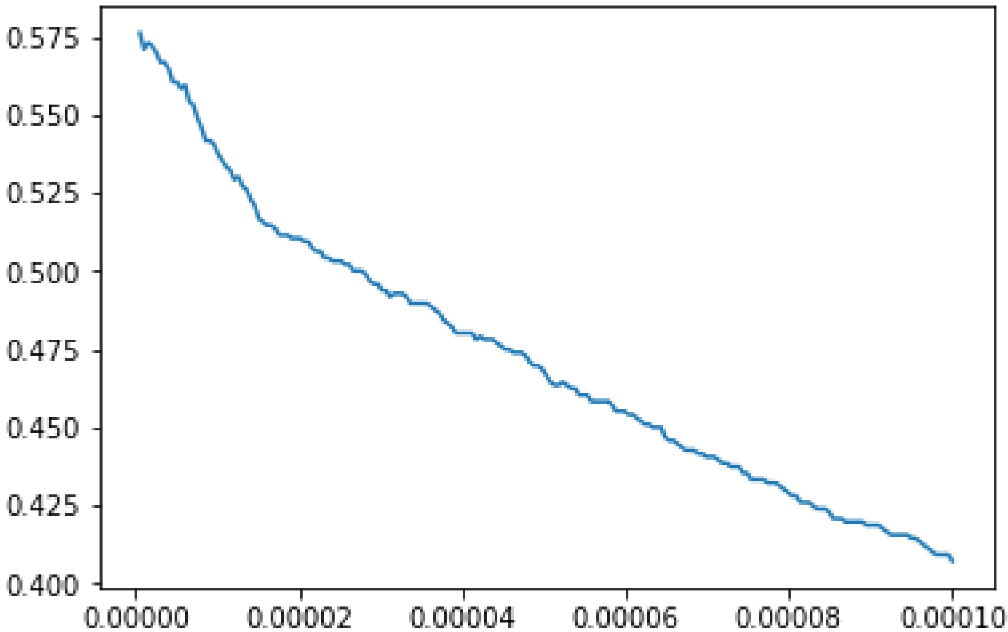
An explanation of this algorithm is available in the notebook `01 - Pop vs. Rap Prediction`.

2.0.1 KL Divergence

Our best accuracy of 57.6% was obtained with a parameter of $\alpha = 5.12 \times 10^{-7}$.

```
In [6]: klgrid = grid_search_nonparametric(0.00000001, 0.0001, 200, NonParametricClassifier, tr
```

```
Best accuracy: 0.5762004175365344
Parameter 5.12462311557789e-07
```

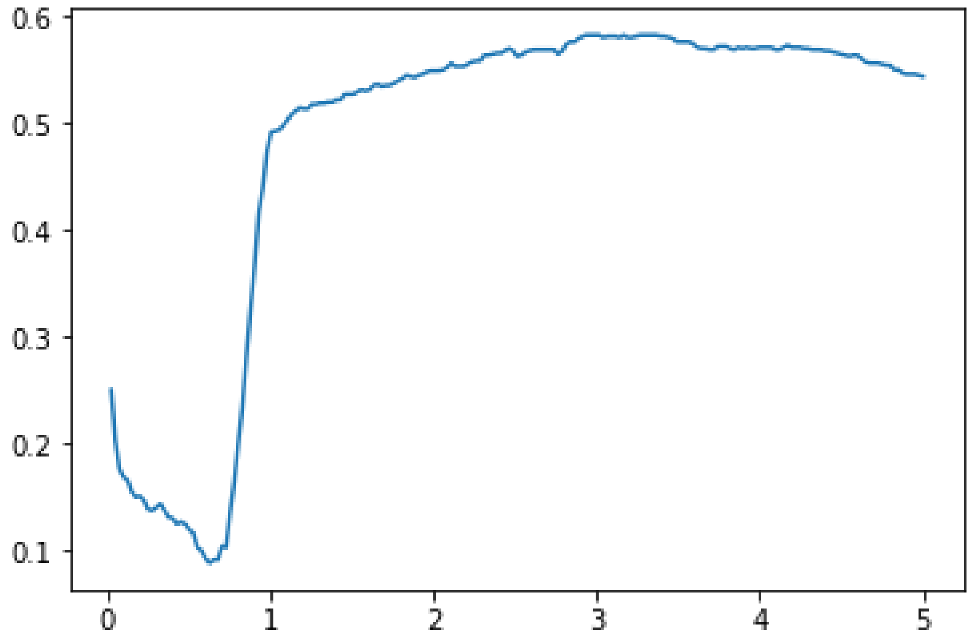


2.0.2 Hellinger

Our best accuracy of 58.2% was obtained with a parameter of $\alpha = 2.9397$.

```
In [7]: hellingergrid = grid_search_nonparametric(0, 5, 200, NonParametricClassifier, train, te
```

```
Best accuracy: 0.5824634655532359
Parameter 2.9396984924623117
```



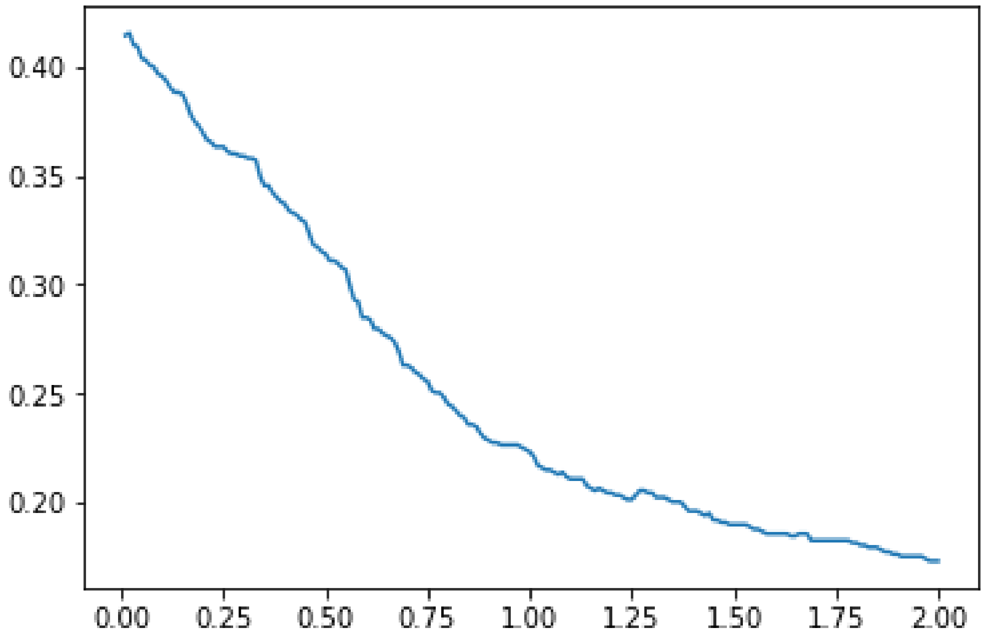
3 Rank-based classification

3.0.1 Mann-Whitney

Our best accuracy of 41.5% was obtained with a parameter of $\alpha = 0.02$. This is much smaller than what we had for the two genre case. We hypothesize this is the case because by including more genres, words that don't show up in the training set but do in the validation set are much more likely to be more rare. Increasing the number of genres decreases the size of the empirical distribution for each genre.

```
In [9]: mwgrid = grid_search_cdf(0.01, 2, 200, CDFClassifier, train, test, ["Pop", "Rap", "Rocl
```

```
Best accuracy: 0.4154488517745303
Parameter 0.02
```



4 Comparison to standard algorithms

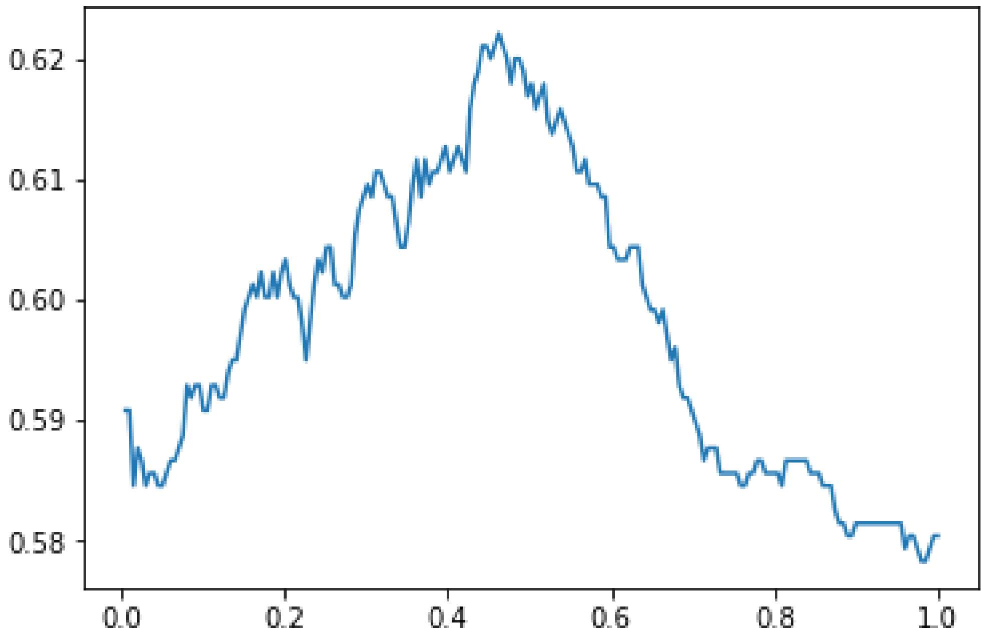
```
In [ ]: X_train, y_train, X_test, y_test = prepare_multigenre_data(train, test)
```

4.0.1 Naive Bayes - Bernoulli

The Bernoulli version of Naive Bayes outperforms our algorithm by 4% with a total accuracy of 62%. We achieved the highest accuracy with $\alpha = 0.462$.

```
In [10]: bernoulligrid = {}
         grid2 = {}

         for n in np.linspace(0, 1, 200)[1:]:
             clf = BernoulliNB(alpha = n)
             clf.fit(X_train, y_train)
             bernoulligrid.update({n: confusion_matrix(clf.predict(X_test), y_test)})
             grid2.update({n: np.diag(bernoulligrid[n]).sum() / bernoulligrid[n].sum()})

         best = sorted(grid2.items(), key = lambda x: x[1], reverse = True)[0]
         print("Best accuracy:", best[1])
         print("Parameter", best[0])

         plt.plot([i for i in grid2], [grid2[i] for i in grid2]);
```

```
Best accuracy: 0.6221294363256785
Parameter 0.4623115577889447
```

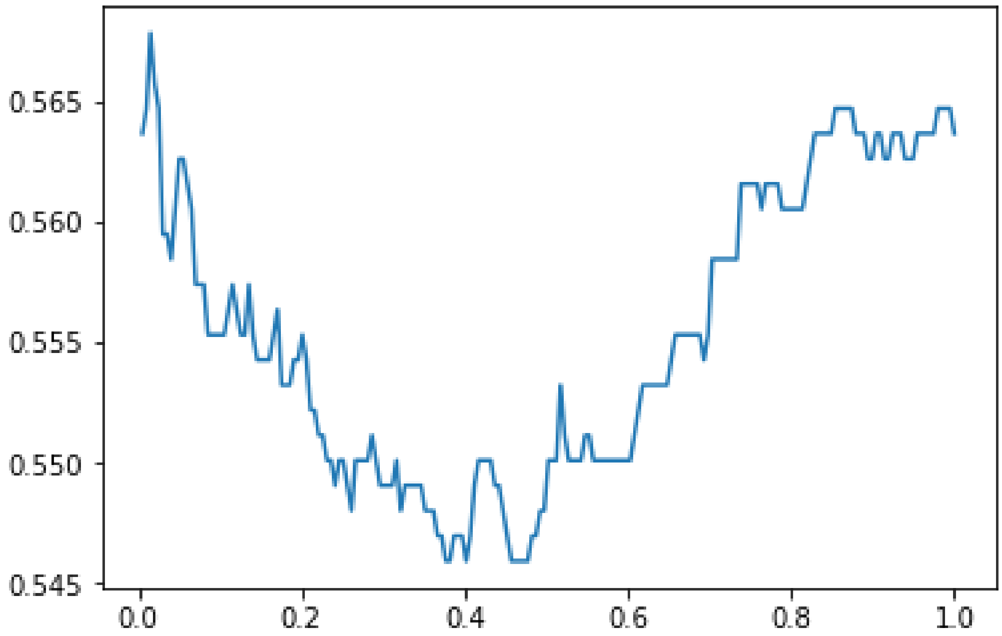


4.0.2 Naive Bayes - Multinomial

Multinomial Naive Bayes does worse than our own algorithm and the Bernoulli version. We achieve an accuracy of 56.79% with $\alpha = 0.01507$.

```
In [11]: multigrid = {}
         grid2 = {}

         for n in np.linspace(0, 1, 200)[1:]:
             clf = MultinomialNB(alpha = n)
             clf.fit(X_train, y_train)
             multigrid.update({n: confusion_matrix(clf.predict(X_test), y_test)})
             grid2.update({n: np.diag(multigrid[n]).sum() / multigrid[n].sum()})

         best = sorted(grid2.items(), key = lambda x: x[1], reverse = True)[0]
         print("Best accuracy:", best[1])
         print("Parameter", best[0])

         plt.plot([i for i in grid2], [grid2[i] for i in grid2]);
```

```
Best accuracy: 0.5678496868475992
Parameter 0.01507537688442211
```



4.0.3 xgboost

For `xgboost`, we changed the objective function to a multiclass probability. The evaluation metric is cross entropy. `xgboost` achieved a best accuracy of 61.4% with ℓ_1 parameter of 0.8 and ℓ_2 parameter of 0.2.

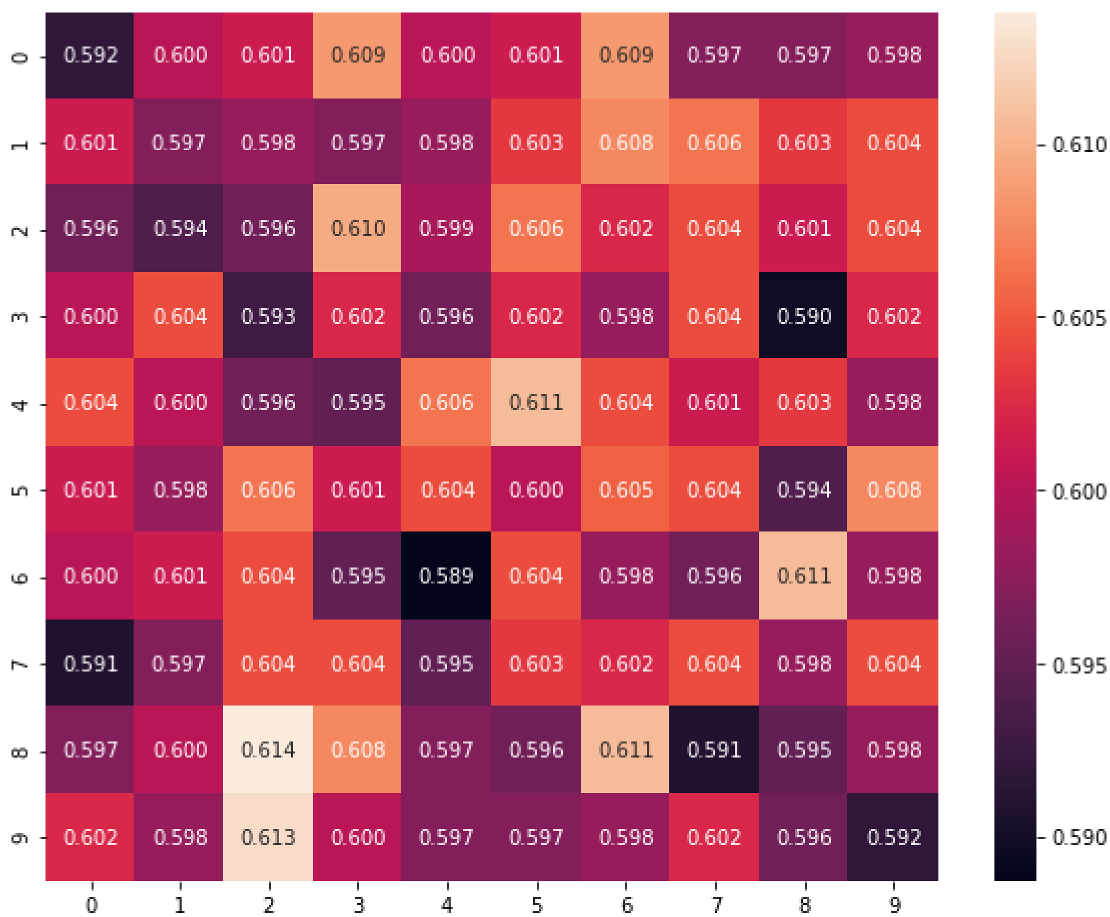
```
In [12]: y_train_binary = convert_genre(y_train)
         y_test_binary = convert_genre(y_test)
```

```
In [ ]: dtrain = xgb.DMatrix(X_train, label = y_train_binary)
        dtest = xgb.DMatrix(X_test, label = y_test_binary)
        evallist = [(dtrain, 'train'), (dtest, 'eval')]

        grid = {}
        dims = 10

        for l1 in np.linspace(0, 1, dims):
            for l2 in np.linspace(0, 1, dims):
                param = {'max_depth': 500, 'eta': 0.2, 'silent': 1, 'objective': 'multi:softpro
                         "lambda": l2, "subsample": 0.9, "num_class": 4, "eval_metric": "mloglo
                bst = xgb.train(params = param, dtrain = dtrain, num_boost_round = 200, evals =
                cfmat = confusion_matrix(np.argmax(bst.predict(dtest), 1), y_test_binary)
                grid.update({(l1, l2): np.diag(cfmat).sum() / cfmat.sum()})
```

```
In [14]: mat = np.zeros((dims, dims))
         row = 0
```

```
col = 0
for (r, c) in grid:
    mat[row, col] = grid[(r, c)]
    col += 1
    if (col) % dims == 0:
        if (row, col) == (0, 1):
            continue
        col = 0
        row += 1
```

```
In [15]: fig = plt.figure(figsize = (10, 8))
         sns.heatmap(mat, annot = True, fmt = ".3f");
```



4.0.4 Feedforward Neural Network

```
In [16]: from keras import Sequential
         from keras.models import load_model
         from keras.layers import Dense, BatchNormalization
```

```
from keras.regularizers import l1, l2
from keras.optimizers import SGD
from keras.callbacks import ModelCheckpoint

from tensorflow import Session, ConfigProto
sess = Session(config=ConfigProto(log_device_placement=True))
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())

from keras import backend as K
K.tensorflow_backend._get_available_gpus()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\h5py\__init__.py:34: FutureWarning: Conversion of th
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 977551298561145 28
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 3157891481
locality {
  bus_id: 1
  links {
  }
}
incarnation: 3788756446154513330
physical_device_desc: "device: 0, name: GeForce GTX 1050 Ti, pci bus id: 0000:01:00.0, compute
]
```

```
Out[16]: ['/job:localhost/replica:0/task:0/device:GPU:0']
```

```
In [17]: from sklearn.preprocessing import OneHotEncoder
         enc = OneHotEncoder(categories = "auto")
         enc.fit(y_train_binary.reshape((len(y_train_binary), 1)))
         y_train_onehot = enc.transform(y_train_binary.reshape((len(y_train_binary), 1))).toar
         y_test_onehot = enc.transform(y_test_binary.reshape((len(y_test_binary), 1))).toarray
```

```
In [51]: arch = [
             Dense(512, input_dim = 23920, activation = "sigmoid"),
             Dense(128, activation = "sigmoid"),
             Dense(32, activation = "sigmoid"),
             Dense(8, activation = "sigmoid"),
```

```
    Dense(4, activation = "softmax")
]

model = Sequential(arch)

model.compile(
    optimizer = SGD(lr = 0.01),
    loss = "categorical_crossentropy",
    metrics = ["categorical_accuracy"]
)

filepath = "..//..//..//..//data//NN weights//weights-improvement-multigenre-{epoch:0
checkpoint = ModelCheckpoint(filepath, monitor='val_categorical_accuracy',
                             verbose=1, save_best_only=True,
                             mode='max')
callbacks_list = [checkpoint]

history = model.fit(
    np.array(X_train),
    np.array(y_train_onehot),
    callbacks = callbacks_list,
    verbose = 1,
    epochs = 40,
    batch_size = 2,
    validation_data = [np.array(X_test), np.array(y_test_onehot)]
)
```

```
Train on 3821 samples, validate on 958 samples
Epoch 1/40
3821/3821 [==============================] - 28s 7ms/step - loss: 1.3104 - categorical_accurac

Epoch 00001: val_categorical_accuracy improved from -inf to 0.38727, saving model to ..//..//.
Epoch 2/40
3821/3821 [==============================] - 27s 7ms/step - loss: 1.3066 - categorical_accurac

Epoch 00002: val_categorical_accuracy did not improve from 0.38727
Epoch 3/40
3821/3821 [==============================] - 27s 7ms/step - loss: 1.3001 - categorical_accurac

Epoch 00003: val_categorical_accuracy improved from 0.38727 to 0.53653, saving model to ..//..
Epoch 4/40
3821/3821 [==============================] - 27s 7ms/step - loss: 1.2821 - categorical_accurac

Epoch 00004: val_categorical_accuracy did not improve from 0.53653
Epoch 5/40
3821/3821 [==============================] - 27s 7ms/step - loss: 1.2129 - categorical_accurac

Epoch 00005: val_categorical_accuracy did not improve from 0.53653
```

```
Epoch 6/40
3821/3821 [==============================] - 27s 7ms/step - loss: 1.0922 - categorical_accurac

Epoch 00006: val_categorical_accuracy improved from 0.53653 to 0.54593, saving model to ..//../
Epoch 7/40
3821/3821 [==============================] - 27s 7ms/step - loss: 1.0272 - categorical_accurac

Epoch 00007: val_categorical_accuracy improved from 0.54593 to 0.55532, saving model to ..//../
Epoch 8/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.9934 - categorical_accurac

Epoch 00008: val_categorical_accuracy did not improve from 0.55532
Epoch 9/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.9691 - categorical_accurac

Epoch 00009: val_categorical_accuracy did not improve from 0.55532
Epoch 10/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.9486 - categorical_accurac

Epoch 00010: val_categorical_accuracy did not improve from 0.55532
Epoch 11/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.9293 - categorical_accurac

Epoch 00011: val_categorical_accuracy did not improve from 0.55532
Epoch 12/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.9158 - categorical_accurac

Epoch 00012: val_categorical_accuracy did not improve from 0.55532
Epoch 13/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.8935 - categorical_accurac

Epoch 00013: val_categorical_accuracy improved from 0.55532 to 0.55846, saving model to ..//../
Epoch 14/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.8715 - categorical_accurac

Epoch 00014: val_categorical_accuracy did not improve from 0.55846
Epoch 15/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.8633 - categorical_accurac

Epoch 00015: val_categorical_accuracy improved from 0.55846 to 0.56054, saving model to ..//../
Epoch 16/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.8466 - categorical_accurac

Epoch 00016: val_categorical_accuracy improved from 0.56054 to 0.57098, saving model to ..//../
Epoch 17/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.8298 - categorical_accurac

Epoch 00017: val_categorical_accuracy did not improve from 0.57098
```

```
Epoch 18/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.8182 - categorical_accuracy

Epoch 00018: val_categorical_accuracy improved from 0.57098 to 0.57516, saving model to ..//..
Epoch 19/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.8015 - categorical_accuracy

Epoch 00019: val_categorical_accuracy did not improve from 0.57516
Epoch 20/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.7859 - categorical_accuracy

Epoch 00020: val_categorical_accuracy did not improve from 0.57516
Epoch 21/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.7716 - categorical_accuracy

Epoch 00021: val_categorical_accuracy improved from 0.57516 to 0.58873, saving model to ..//..
Epoch 22/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.7591 - categorical_accuracy

Epoch 00022: val_categorical_accuracy did not improve from 0.58873
Epoch 23/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.7455 - categorical_accuracy

Epoch 00023: val_categorical_accuracy did not improve from 0.58873
Epoch 24/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.7376 - categorical_accuracy

Epoch 00024: val_categorical_accuracy did not improve from 0.58873
Epoch 25/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.7332 - categorical_accuracy

Epoch 00025: val_categorical_accuracy did not improve from 0.58873
Epoch 26/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.7184 - categorical_accuracy

Epoch 00026: val_categorical_accuracy did not improve from 0.58873
Epoch 27/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.7038 - categorical_accuracy

Epoch 00027: val_categorical_accuracy did not improve from 0.58873
Epoch 28/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.6936 - categorical_accuracy

Epoch 00028: val_categorical_accuracy improved from 0.58873 to 0.59603, saving model to ..//..
Epoch 29/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6901 - categorical_accuracy

Epoch 00029: val_categorical_accuracy did not improve from 0.59603
```

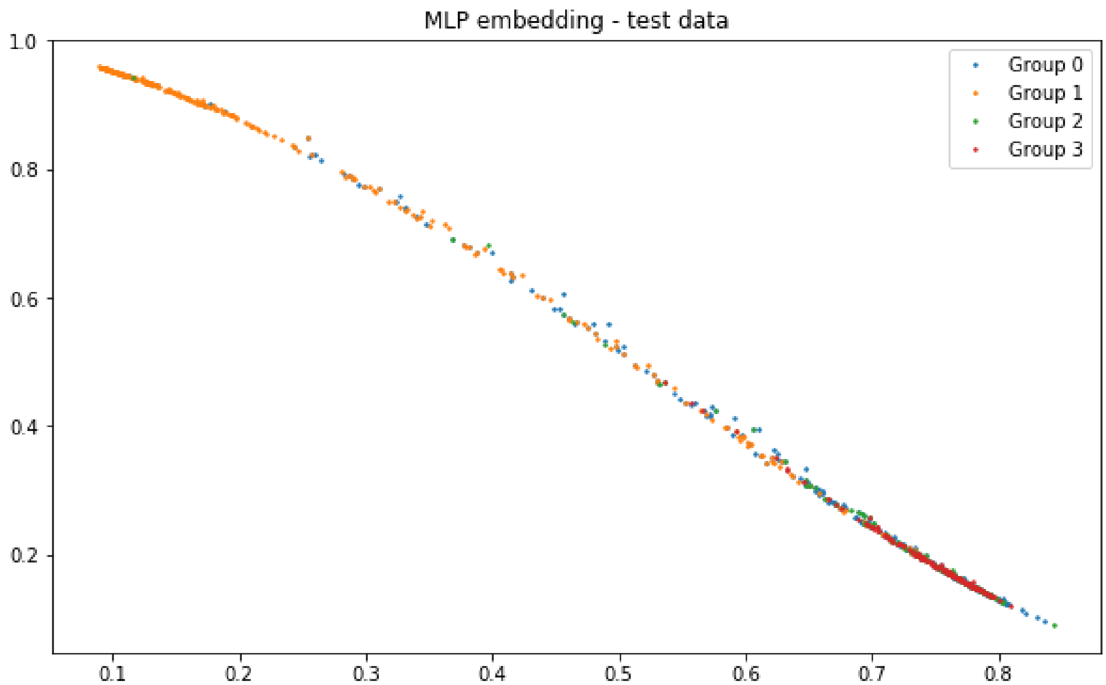
```
Epoch 30/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6813 - categorical_accurac

Epoch 00030: val_categorical_accuracy improved from 0.59603 to 0.60125, saving model to ..//../
Epoch 31/40
3821/3821 [==============================] - 29s 7ms/step - loss: 0.6681 - categorical_accurac

Epoch 00031: val_categorical_accuracy did not improve from 0.60125
Epoch 32/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6625 - categorical_accurac

Epoch 00032: val_categorical_accuracy did not improve from 0.60125
Epoch 33/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.6574 - categorical_accurac

Epoch 00033: val_categorical_accuracy did not improve from 0.60125
Epoch 34/40
3821/3821 [==============================] - 27s 7ms/step - loss: 0.6501 - categorical_accurac

Epoch 00034: val_categorical_accuracy did not improve from 0.60125
Epoch 35/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6436 - categorical_accurac

Epoch 00035: val_categorical_accuracy did not improve from 0.60125
Epoch 36/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6396 - categorical_accurac

Epoch 00036: val_categorical_accuracy did not improve from 0.60125
Epoch 37/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6309 - categorical_accurac

Epoch 00037: val_categorical_accuracy improved from 0.60125 to 0.60230, saving model to ..//../
Epoch 38/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6226 - categorical_accurac

Epoch 00038: val_categorical_accuracy did not improve from 0.60230
Epoch 39/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6202 - categorical_accurac

Epoch 00039: val_categorical_accuracy did not improve from 0.60230
Epoch 40/40
3821/3821 [==============================] - 28s 7ms/step - loss: 0.6144 - categorical_accurac

Epoch 00040: val_categorical_accuracy did not improve from 0.60230
```

In [26]:
```
def plot_embedding(encoder, X, Y):
    fig = plt.figure(figsize = (10, 6))
```

```
h = encoder.predict(np.array(X))
y_tester = np.array(Y)
for i in range(4):
    sel = y_tester == i
    plt.plot(h[sel, 0], h[sel, 1], '.', label='Group %d' % i, markersize = 3, alph
plt.title('MLP embedding - test data')
plt.legend()
plt.show()
```

As an aside, below is the embedding of our test data within our neural network. We can see that the neural network has learned a linearly separable representation of our frequency vectors. We also see that group 2 and 3 (Rock and Country) are very similar to group 0, or Pop. This implies that the text in pop, country, and rock are all very similar.

```
In [27]: nn = load_model("..//..//..//..//data//NN weights//weights-improvement-multigenre-10-0
         model_tmp = Sequential(nn.layers[:-1])
         plot_embedding(model_tmp, X_test, y_test_binary)
```



5 Results

Below is a table summarizing the performance of each algorithm on the validation set.

KL	Hellinger	Mann-Whitney	NB-Bernoulli	NB-Multinomial	xgboost	Neural network
0.5762	0.58246	0.4154	0.6221	0.56785	0.614	0.6023

Again, Bernoulli naive bayes performs the best on this dataset. Our distribution comparison algorithm outperforms the multinomial naive bayes. However, it seems our algorithm does not do well with more genres. This could be due to the smaller vocabulary each genre takes up.