Assignment 1 - Classification Task

CS985MLDAGroup17

- Alejandro Eisen Jofre: 202385817
- Christy Bridgeman: 202353289
- Michele Romano: 202354540
- Noor Chaloner Yassein: 202375223
- Tricia Govindasamy: 202375355

Introduction

By exploring a range of classification techniques, this project aims to use machine learning to find the most accurate model to predict the top genre of any given song from a dataset of Spotify music hosted on Kaggle.

Before delving into specific machine learning classifiers, this project will first explore the Spotify dataset, construct and justify a generalised approach to address this classification problem before then cleaning and preparing the data. We then test some preliminary regularisation techniques and lastly test, tune and select an appropriate machine learning classifier approach.

Dataset Overview

To view the generalisability of our machine learning models, this project has a training dataset and a testing dataset; the former contains all features to act as inputs in our classifier, and the latter helps to confirm/deny our models' predictions. The training dataset uses a modified version of the 'Spotify Past Decades Songs Attributes' Kaggle dataset by Nicolas Carbone (2020). It consists of 453 entries (songs) with 15 features (attributes for each song) (see Data Preprocessing). Its Kaggle page has a codebook which outlines the meaning of each feature, with columns such as `year`, `bpm` (beats per minute), `nrgy` (how energetic a song is), `spch` (how many spoken words are contained in a song), etc. All these features remain for the testing dataset, with far fewer instances, apart from the `top genre` feature as this is what this project is predicting.

Methods

The data in our training dataset is labelled, meaning that any given instance of data is associated with a class. Whilst in our testing dataset we are aware of which feature a given sample of data in an instance belongs to (eg: `dB` (loudness of a song)), its class (`top genre`) is unknown. The correct class is also featured in a validation dataset

which is not available to us for the duration of this project. With labelled training of our data, this project thusly undertakes supervised learning (Géron, 2019, pp9-13).

All the features in our dataset are numerical, except `title`, `artist` and, most importantly, `top genre`. Predicting for top genre means predicting a class (such as a category of genre), as opposed to a value (such as the danceability of a song, `dnce`). Therefore, as we are predicting a categorical variable, this project makes use of classification methods.

Furthermore, as there exists more than two music genres, our project lends itself to multiclass classification: predicting between more than two classes (Géron, 2019, p100). This does not necessarily prohibit the use of binary classifiers, however, which use multiple one-versus-one or one-versus-rest binary classifiers to train many genres head-to-head (ibid). Despite testing for many genres per instance (and also the nature of music genres often overlapping with each other), we do not employ multi-label classification. Instead of predicting many columns, we predict for one column and classify it in accordance with which genre scored the highest confidence.

Libraries Import

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import RobustScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.linear_model import LogisticRegression
from pathlib import Path
from scipy.stats import f_oneway, kendalltau
import matplotlib.pyplot as plt
```

Classes and Functions Definitions

In [ ]:
```
def transform_outliers(data, column_name, values):
    IQR = ((data[values].quantile(0.75)) - (data[values].quantile(0.25)))
    upper_threshold = data[values].quantile(0.75) + (1.5*IQR)
    lower_threshold = data[values].quantile(0.25) - (1.5*IQR)

    data[column_name] = data[values].clip(lower_threshold, upper_threshold)

def classification(X, y):
    classifiers = {
        "SVC": SVC(random_state=42),
        "RandomForest": RandomForestClassifier(random_state=42),
        "DecisionTree": DecisionTreeClassifier(random_state=42),
        "KNN": KNeighborsClassifier(),
        "GNB": GaussianNB(),
        "LR": LogisticRegression(random_state=42)
    }

    results = {}
```

```
    for clf_name, clf in classifiers.items():
        #! Usage of cross_val_score can be problematic as folds may not cont
        scores = cross_val_score(clf, X, y, cv=5, scoring="accuracy")
        acc_mean = scores.mean()
        results[clf_name] = acc_mean
    return results
```

Data Preprocessing

Importing Data and Handling with Pandas

Google Colab with Python's pandas library allows for the easy reading and transformation of comma-separated value files straight from GitHub. Here, `pd.read_csv` loads our raw text datasets into a pandas dataframe (a structure of rows and columns for tabular data). To ensure no contamination of the training dataset, using `.copy()` we create a replica and work with this going forward. Pandas is also used for `.info()`; this prints information about our column labels, datatypes and non-null count.

Where an instance has a null value in its top genre column, it is removed entirely with pandas's `.dropna()`. By not removing these instances, our machine learning models may learn songs without an assigned genre as similar to comparable songs with an assigned genre. Issues arise here because similar songs may exist in entirely different genres, thus meaning our training would learn on potentially incorrect data.

Data Import

In [ ]:
```
url = "https://raw.githubusercontent.com/iniznerol/CS985/main/CS98XClassifi
url_2 = "https://raw.githubusercontent.com/iniznerol/CS985/main/CS98XClassi
raw_train_data = pd.read_csv(url)
test_set = pd.read_csv(url_2)
data = raw_train_data.copy()
```

In [ ]:
```
data.info()
```

Regularisation

Outliers in our data mean later machine learning models may struggle to detect underlying patterns and thus generalise poorly (known as overfitting). Likewise, a high number of attributes – especially ones with little effect on models' accuracies – can both slow down the training process and potentially result in overfitting. These processes constitute what is called 'regularisation', which is an important task to minimise the risk of overfitting (Géron, 2019, pp26-27).

Lastly, we explore the shape of our data as this will affect the scaling methods we use, models' hyperparameters and the number of features we include or exclude from training (Géron, 2019, pp50-51, p31 & p226).

Outliers

Using pandas, features with outliers can be captured with `.boxplot()`. Outliers sit outside the 'minimum' and 'maximum' whiskers which are defined as:

$$Min = Q1 - 1.5 \times IQR \quad (1)$$
$$Max = Q3 + 1.5 \times IQR$$

Noisy data means our machine learning models will overfit and learn data which does not represent the data's underlying patterns (Géron, 2019, pp26-28). Features with outliers were identified as `bpm`, `dB`, `live` (likeliness of a song being recorded live), `dur` (duration) and `spch`.

Using a custom function `transform_outliers()` and the pandas `.clip()` function, we reassign outlier values to the nearest non-outlier value: min or max (see above formula). As any singular instance can have features both with and without outliers, omitting an entire row because of this would mean our dataset size is reduced and valuable data is lost. By instead limiting the ranges on these features, we can retain instances at the extremities of our distribution whilst simultaneously not further shrinking our dataset. These variables are separated with the suffix `_clip`. This allows for testing of a trained model with and without outliers.

Moreover, we note that our dataset has many instances with unique top genre values. Whilst performing oversampling techniques such as SMOTE (Synthetic Minority Oversampling Technique) may be productive as it would help with cross validation, issues where some instances occur within the training or test set but not on both (since there is only one of them), we don't make use of this nor any other oversampling techniques as restricted by this challenge's requirements.

In [ ]:
```
data = data.dropna()
```

In [ ]:
```
#Boxplot the variables to check for outliers
data.boxplot(column=['bpm', 'nrgy', 'dnce', 'dB', 'live',
       'val', 'dur', 'acous', 'spch', 'pop'], figsize=(20,10))
plt.title('Boxplot of training dataset', fontsize=30)
plt.xlabel('Features', fontsize=15)
plt.ylabel('Values', fontsize=15)
# We can see only bpm, dB, live, dur and spch have outliers so we only apply
# Also important to note, we are not erasing any rows of information, just
```

In [ ]:
```
#Outlier Transformation for bpm, dB, live, dur and spch
transform_outliers(data, "spch_clip", "spch")
transform_outliers(data, "dur_clip", "dur")
transform_outliers(data, "live_clip", "live")
transform_outliers(data, "bpm_clip", "bpm")
transform_outliers(data, "dB_clip", "dB")
```

Dimensionality

The 'curse of dimensionality' states that the more features a training dataset has, the slower the training shall be. Thus, by omitting features which have little statistical significance, our training models may be less bloated (Géron, 2019, p213).

Using the Scipy library and the Kendall Tau measure of correlation, `kendalltau()`, each feature in our clipped dataset is individually evaluated against top genre (our dependent variable) to measure its strength and direction in predicting genre category.

We also use ANOVA (Analysis of Variance) to determine whether there exists a statistically significant difference in means between groups of genres and individual features with Scipy's `.stats.f_oneway`.

Where returned p-values are less than confidence level 0.05, statistically significant features are captured and assigned to the new variable `data_clip_correlated_cols`. However, upon testing these data with basic models, they performed worse than the ones that included all columns, we therefore keep all moving forward with the original `_clip` data.

In another attempt to reduce dimensionality, this project makes use of PCA (Principal Component Analysis). To preserve the highest amount of variance in our dataset but with the fewest dimensions, PCA projects our data onto a lower-dimensional hyperplane (reducing the axes our data sits on) that most closely follows patterns in the dataset. PCA then finds a secondary axis, orthogonal to the previous, that accounts for the remaining variance. This process continues for however many dimensions we wish to project down to (Géron, 2019, p220). For this task, the dimensionality is reduced with `PCA(n_components = 0.95` with Scikit-Learn. This Scikit-Learn function ensures the dimensions retained make up 95% of the variance in our dataset. As before, this seemed to worsen our results which may go to show a certain linearity in our data. We chose to keep moving forward without PCA.

In [ ]:
```
#lets specify one object for each clipped and non clipped data.
data_clipped = data[['year', 'nrgy', 'dnce', 'val', 'acous', 'pop', 'spch_c
data_no_clip = data[['year', 'bpm', 'nrgy', 'dnce', 'dB', 'live', 'val', 'du
data_clipped.shape, data_no_clip.shape #same dimension datasets
```

In [ ]:
```
#ANOVA
numerical_features = data_clipped.loc[:, data_clipped.columns!="top genre" ]
for feature in numerical_features:
    groups = [data_clipped[data_clipped['top genre'] == genre][feature] for
    stat, p_value = f_oneway(*groups) #perform test
    print(f"\nANOVA for {feature}: {stat}, with a p-value = {p_value}") #pri
    if p_value < 0.05:
        print(f"For {feature}; there is a significant difference in means.")
    else:
        print(f"For {feature}; there is no significant difference in means.'
```

In [ ]:
```
#Kendall Tau
for feature in numerical_features:
    tau, p_value = kendalltau(data_clipped[feature], pd.factorize(data_clipp
    print(f"\nKendall's Tau for {feature} = {tau}, with a p-value = {p_value
    if p_value < 0.05:
        print(f"For {feature}: there is significant correlation.")
    else:
        print(f"For {feature}; there is no significant correlation.")
```

Shape of Data

Using the Pandas `.hist()` function, we capture a visual representation of the distribution of our data. Not only is it evident that our features exist on different scales, but that many have non-normal distributions. This is the case for both the raw and clipped datasets. For example, `acous` (how acoustic a song is) and `live_clip` are skewed with higher frequencies in lower values.

After testing Scikit-Learn's `StandardScaler`, `MinMaxScaler` and `RobustScaler` on some preliminary training models to determine general performance, this project finds that `RobustScaler` fits our purposes best. Standard scaling subtracts the mean from features and then divides by its standard deviation; Minimum-Maximum scaling plots all feature values on the same scale, 0 to 1 (Géron, 2019, p69). Robust scalers performed the best in our preliminary tests likely because, even though we removed outliers, our skewed distributions mean the mean and standard deviations are less representative of underlying patterns in the data (Brownlee, 2020a). Moreover, as standard scalers are most effective when scaling data with outliers, it is expected that they would not perform strongly with our clipped data (Géron, 2019, p69). The robust scaler instead removes the median values and adjusts the remaining data according to the interquartile range (Brownlee, 2020a), as below.

$$X_{\text{robust}} = \frac{x - x_{\text{median}}}{IQR} \tag{2}$$

As evident from the histograms when exploring our datasets, our features tend to be skewed and not normally distributed. Using Scikit-Learn's `.PolynomialFeatures()` class, we tested with a squared polynomial (degree of 2) to work towards a more fitting model: this replicates existing features then squares them. However, it worsened results and was thus removed. Given that a polynomial degree of only two leads to overfitting, further increasing this degree would only make models more inaccurate (Géron, 2019, p128).

In [ ]:
```
data.hist(figsize=(30,15), bins=50) #This is original data, not clipped.

X_clip = data_clipped.loc[:, data_clipped.columns != "top genre"]
y_train = data["top genre"]
robust = RobustScaler()
X_clip_robust = robust.fit_transform(X_clip)
```

In [ ]:
```
classification(X_clip,y_train) #!Results from this code were in general bet
```

In [ ]:
```
classification(X_clip_robust, y_train) #while standardscaler performed a bi
#robust scaler makes statistical sense in our project
```

Training

At this stage, the data to be used for training has transformed beyond our original raw dataset. Going forward, we use robust scaled data with outliers clipped. Dimensionality reduction techniques, PCA and ommitting insignificant features with Kendall Tau and ANOVA, are not used in our training data because of their poor performance in

preliminary tests. Additional polynomial features and oversampling methods are also not used.

In finding the most accurate classifier, we shall a) Test and fine-tune individual classifiers, informed initially by preliminary tests conducted when testing regularisation methods and b) Test ensemble methods of machine learning classifiers.

Fine-tuning Individual Models and Ensemble Learning

Using Scikit-Learn's `GridSearchCV` (grid search cross validation) method we are able to detect the strongest combination of hyperparameters to pass into our training models (Géron, 2019, p76). Even if individual models are not particularly strong, fine-tuning them means more accurate results when later passed into an ensemble method.

KNN

The k-nearest neighbour (KNN) algorithm classifies instances based upon similar features within a local distance. Classes with high confidence (lots of common data) act as a prediction centre, and subsequent instances are classified based on their distance to it. This scored in testing 34.02%, but with the Kaggle validation dataset it reached 35.7% accuracy.

KNN didnt seem to perform too good when run the first time, but it's a model well worth looking into as it performs by looking at the distance between the classes to group them which makes sense in this type of task. We can see that a well tuned KNN model performs as good as, if not better than other models.

In [ ]:
```
parameters = {
    'n_neighbors': np.arange(40,45,1),
    'p': [1,2],
    'algorithm': ['ball_tree', 'kd_tree', 'brute'],
    'weights': ['uniform','distance']
}
knn_clf = KNeighborsClassifier()
grid_knn = GridSearchCV(knn_clf, param_grid = parameters, n_jobs=-1)
grid_knn.fit(X_clip_robust,y_train)
print(f'Best Parameters:{grid_knn.best_params_}, With a score of: {grid_knn
#Best Parameters:{'algorithm': 'ball_tree', 'n_neighbors': 28, 'p': 2, 'weig
```

Decision Trees

We dont use predictions made by simple Decision Trees as our submition, this is because they are prone to overfitting (Géron, 2019, p185). However since they are the basis for other estimators, like GBC and RND, we still keep their code here to tune it and then pass it into other models.

Since Gradient Boosting is better suited for small-medium sized dataset and deals better with imbalanced classes, it makes sense to try this model. However, instead of trying to fine tune GBC directly, we tried to find good parameters in a simple Decision Tree, and then pass these into a GBC and just tune the number of estimators in it. This

being said, we dont keep GBC results in this report as they performed poorly on out data. We instead move forward with `RandomForestsClassifier()`

In [ ]:
```
parameters = {
    'max_depth': [1,2,3,4,5,7],
    'min_samples_split': [2,3,4,5,10],
    'min_samples_leaf': [1,2,3,4],
    'splitter' : ["best", "random"],
    'criterion' : ["gini", "entropy", "log_loss"],
    'max_leaf_nodes': [2,4,6,None]
}
dt_clf = DecisionTreeClassifier(random_state=42)
grid_dt = GridSearchCV(estimator = dt_clf, param_grid=parameters, n_jobs=-1
grid_dt.fit(X_clip_robust,y_train)
best_param_dt = grid_dt.best_params_
print(f'Best Parameters:{grid_dt.best_params_}, With a score of: {grid_dt.be
# Best Parameters:{'criterion': 'entropy', 'max_depth': 2, 'max_leaf_nodes'
```

Stacking

Ensemble learning involves aggregating multiple machine learning methods on top of each other to develop a more accurate model. By manually creating a stacking ensemble learner (as it is not supported natively in Scikit-Learn), a method which combines meta learners into a final predictor (Géron, 2019, p208), we were able to generate a stronger ensemble method.

In [ ]:
```
classifier1 = SVC(random_state=42, probability=True, C= 0.9, kernel= 'rbf')
classifier2 = LogisticRegression(C= 0.4, max_iter= 10, multi_class= 'multinc
classifier3 = DecisionTreeClassifier(criterion= 'entropy', max_depth= 2, max
classifier4 = RandomForestClassifier(random_state=42, max_depth= 10, n_estir
classifier5 = KNeighborsClassifier(algorithm= 'ball_tree', n_neighbors= 28,
stack_clf = StackingClassifier(estimators=[
    ('svc', classifier1),
    ('lr', classifier2),
    ('dt', classifier3),
    ('rf', classifier4)
], final_estimator=classifier5)
stack_clf.fit(X_clip_robust,y_train)
print(cross_val_score(stack_clf, X_clip_robust,y_train))
#[0.375 0.38636364 0.34090909 0.35632184 0.33333333] ->  svc,lr,dt,rf,knn
```

Test Sets Preparation and Output Prediction

- In this section we use the same transformation on the test set as we did in the training set. That is, clipping the outliers, and scaling the values with RobustScaler().
- After that, we use the fitted model to make a prediction on the already transformed test set. We keep those predictions and the *Id* column, and pass them into a .csv which in turn gets uploaded to kaggle to get an accuracy score.

In [ ]:
```
X_test = test_set.loc[:, test_set.columns != ("Id","title","artist") ]
id_column = test_set["Id"]
```

In [ ]:
```
transform_outliers(X_test, "spch_clip", "spch")
transform_outliers(X_test, "dur_clip", "dur")
```

```
transform_outliers(X_test, "live_clip", "live")
transform_outliers(X_test, "bpm_clip", "bpm")
transform_outliers(X_test, "dB_clip", "dB")
X_test_clip = X_test[['year', 'nrgy', 'dnce', 'val', 'acous', 'pop', 'spch_
```

In [ ]:
```
X_test_clip_robust = robust.fit_transform(X_test_clip)
```

In [ ]:
```
stacking_prediction = stack_clf.predict(X_test_clip_robust)
#knn_pred = knn_clf.predict(X_test_clip_robust)

prediction = pd.DataFrame({
    "Id": id_column,
    "top genre": stacking_prediction #Change to knn_pred to export other mod
})
filepath = Path('folder/subfolder/stacking.csv') #change folder/csv name fo
filepath.parent.mkdir(parents=True, exist_ok=True)
prediction.to_csv(filepath, index=False)
```

Conclusion and Reflections

We started this project by exploring the data and evaluating if there were transformations needed in order to feed the data into the models. For this we took care of outliers that may have introduced noise into our models as well as scale the data by using a Robust Scaler. While this doesn't necessarilly fix normality, it wasn't an issue as we didn't assume normality for this classification task. Our two best performing models were KNN and an ensemble of Stacking, which returned a traning accuracy of 34% and 35.8% repectively, and a testing on kaggle accuracy of 35.7% on both accounts.
The stacking classifier used was given a set of 5 different estimators: SVC, LR, DT, and RF in that order, with the final estimator set to KNN, all of which were passed their tuned parameters as found by performing a GridSearchCV in each individually.

The difference in performance from our training data set to the test data on Kaggle can be attributed to different factors.

1. As mentioned before, our training set is fairly small with only 438 rows of songs, for many of which the 'top genre' target is a one-instance case. A low n of rows means our model has less information to learn from which will impact it's ability to generate predictions when very subtle differences in the variables values may be present.
2. On the other hand, classes with few or unique occurences mean that the model can't learn what that class (or genre in this case) looks like, thus predicting it will be extremely difficult.

Our data is not extremely overfitted as can be seen by our estimator scores which are not 1 or too close to it and the performance on kaggle is relatively similar to Kaggle which means it learned how to generalize. However, the low accuracy score is also indicating that, while our models predict some genres well, it doesnt do well in others. However, the final score may vary as the hidden 50% of the Kaggle test set is released.

References

Brownlee, J. (2020a) 'How to Scale Data with Outliers for Machine Learning', *Machine Learning Mastery* [Online]. Available at: https://machinelearningmastery.com/robust-scaler-transforms-for-machine-learning/ (Accessed 13 February 2024)

Brownlee, J. (2020b) 'Naïve Bayes for Machine Learning', *Machine Learning Mastery* [Online]. Available at: https://machinelearningmastery.com/naive-bayes-for-machine-learning/ (Accessed 13 February 2024)

Brownlee, J. (2021) 'SMOTE for Imbalanced Classification with Python', *Machine Learning Mastery* [Online]. Available at: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ (Accessed 13 February 2024)

Carbone, N. (2020) 'Spotify Past Decades Songs Attributes', *Kaggle* [Online dataset (modified)]. Available at: https://www.kaggle.com/datasets/cnic92/spotify-past-decades-songs-50s10s (Accessed 1 February 2024)

Géron, A. (2019) *Hands-On Machine Learning with Scikit-learn, Keras, and TensorFlow*. 2nd edn. California: O'Reilly.