Configuration Manual

MSc Research Project
Artificial Intelligence

Cian McGrane
Student ID: 19181931

School of Computing
National College of Ireland

Supervisor: Sheresh Zahoor

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Cian McGrane
Student ID:	19181931
Programme:	MSc. In Artificial Intelligence **Year:** 2024
Module:	Research Practicum/Internship Part 2
Lecturer:	Sheresh Zahoor
Submission Due Date:	12th August 2024
Project Title:	Visual Harmonies: Investigating the Impact of Album Cover Image Features in Music Genres Classification for Top Spotify Artists in the USA
Word Count:	830 **Page Count:** 8

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: 

Date: 11th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	

Penalty Applied (if applicable):	

Configuration Manual

Cian McGrane
Student ID: 19181931

1. Introduction

This manual provides detailed instructions on how to set up and run the music classification research project. It includes information on system requirements, installation steps, dataset preparation, running model, and troubleshooting. The final section we explain some of the code that was used in the project.

2. System requirements

Hardware requirements

- Processor: Minimum 4-core CPU (8-core recommended)
- RAM: Minimum 8GB (16GB recommended)
- Storage: Minimum 100GB free space

Software requirements

- Operating system: Ubuntu 18.04 or later / Windows 10 / MacOS 10.15 or later
- Python 3.8 or later
- Essential Python package (detailed in installation instructions)

3. Installation Instructions

1. Download the project folder, Final_Project.
2. Import the project to your preferred IDE.
3. Update the FILE_PATH variable in the Config.py file to the location the project is saved.
4. Update CLIENT_ID and CLINET_SECERT with your credentials from the Spotify API, https://developer.spotify.com/documentation/web-api
5. The following libraries need to be installed into your virtual environment:

Item	Name	Version	Link
Library	Scikit-learn	1.4.1.post1	https://scikit-learn.org/stable/
Library	pandas	2.1.4	https://pandas.pydata.org/
Library	matplotlib	3.8.0	https://matplotlib.org/

Library	NumPy	1.26.4	https://numpy.org/
Library	Imbalanced-learn	0.12.3	https://imbalanced-learn.org/stable/
Library	seaborn	0.12.2	https://seaborn.pydata.org/
Library	Opencv-python	4.9.0.80	https://pypi.org/project/opencv-python/
Library	pickle	3.8	https://docs.python.org/3/library/pickle.html
Library	scikeras	0.12.0	https://adriangb.com/scikeras/stable/
Library	astunparse	1.6.3	https://pypi.org/project/astunparse/
Library	tekore	5.4.0	https://tekore.readthedocs.io/en/stable/

4. Dataset Preparation

- The Kaggle dataset is available in the data folder within the project.
- The dataset from Spotify is generated by the script and will be saved to the data folder.

5. Run Model

Open main.py and run the script

6. Code

The code snippets below will detail some of the function that were writing for data sourcing, data preprocessing, data modelling, and data evaluation.

The first code snippet shows how the Spotify API was queried. The object return from the API is a JSON object. A loop is used to loop through the different album for each artist. The artist URIs are broken into batches of 500 and a delay is added in-between requests. The reason for this is to manage the number of requests to the API. If there are too many requests sent to the API consecutively it will time out. The elements that are pulled from the API are arrtistID, albumName, and imageURL. These are added to a DataFrame and each batch are finally concatenated.

```python
for i in range(0, len(artist_id_list), Config.BATCH_SIZE):
  # Get the current batch of album URIs
  batch = artist_id_list[i:i+Config.BATCH_SIZE]
  artistNames=[]
  artistIDs=[]
  albumURIs=[]
  albumNames=[]
  albumImageURLs=[]

  for artist_id in tqdm(batch, desc="Getting data from Spotify API"):
    try:
      # These values will be used to query the API and to populate the Dataframe at the end

      random_number = random.uniform(0.5, 2)
      time.sleep(random_number) #Add delay inbetween request so the API doesn't crash
      albums = spotify.artist_albums(artist_id)

      for album in albums.items:
          albumURIs.append(album.uri)
          albumNames.append(album.name)

          artistIDs.append(artist_id)
          artist_name = album.artists[0].name
          artistNames.append(artist_name)

          try:
              albumImageURL = str(album.images[0].url)
          except IndexError:
              albumImageURL = "No Image"
          albumImageURLs.append(albumImageURL)

    except (ValueError, IndexError):
      # Handle the exception if API doesn't return any data
      print(f"Error occurred for album URI '{artist_id}'")
      continue

    albumData = {'artistID':artistIDs}
    albumDF = pd.DataFrame(albumData)
    albumDF['albumID'] = albumURIs
    albumDF['albumName'] = albumNames
    albumDF['artistName'] = artistNames
    albumDF['Image'] = albumImageURLs
    albumDF.to_csv(Config.FILE_PATH + "/data/albumData_batch" + str(i) + ".csv", index=False)

concat_files.concat_data_files()
```

There are two features extracted from the images most dominant colour (MDC) and pixel intensity histogram (PIH). The k-means clustering algorithm is used to extract the colours from the images. The number of clusters is the number of colours that are to be extracted. If the number of clusters is less than the expected number of dominant colures the remaining colours are populated with the first colour that is extracted.

```python
def extract_dominant_colours(image, desired_clusters):
    # Flatten the image array to a 2D shape
    flattened_array = image.reshape(-1, 3)

    # Determine unique colors and set the appropriate cluster count
    unique_colors = np.unique(flattened_array, axis=0)
    num_clusters = min(desired_clusters, unique_colors.shape[0])

    try:
        # Perform KMeans clustering to find dominant colors
        kmeans = KMeans(n_clusters=num_clusters, n_init=10)
        kmeans.fit(flattened_array)
        dominant_colours = kmeans.cluster_centers_

        # If the number of clusters is less than desired, fill with a repeating colour
        if dominant_colours.shape[0] < desired_clusters:
            first_colour = dominant_colours[0]
            fill_colours = np.tile(first_colour, (desired_clusters - dominant_colours.shape[0], 1))
            dominant_colours = np.vstack([dominant_colours, fill_colours])

        # Adjust reshaping based on desired_clusters
        # If reshaping doesn't work, raise an exception or provide a default
        expected_size = desired_clusters * 3
        if dominant_colours.size != expected_size:
            raise ValueError(f"Expected {expected_size} elements, but found {dominant_colours.size}.")

        reshaped_colours = np.round(dominant_colours, decimals=0).astype(int).reshape((desired_clusters, 3))

        return reshaped_colours

    except Exception as e:
        print(f"Error extracting dominant colours: {e}")
        # Return a default array with the correct shape filled with zeros
        return np.zeros((desired_clusters, 3), dtype=int)
```

The Python library cv2 has a method, calcHist, that generates a pixel intensity histogram. There are three histograms created one for each colour channel, red, green, and blue.

```python
# Function to extract histogram features from an image
def extract_histograms(image, image_url):
    try:
        # Compute histogram for each RGB channel
        hist_channels = [cv2.calcHist([image], [i], None, [256], [0, 256]) for i in range(3)]
        hist_channels = np.array(hist_channels)

        # Create a DataFrame for color histograms
        colour_histogram = pd.DataFrame({
            'ImageURL': image_url,
            'Pixel Intensity': range(256),
            'Red Frequency': hist_channels[0].flatten(),
            'Green Frequency': hist_channels[1].flatten(),
            'Blue Frequency': hist_channels[2].flatten(),
        })

        return colour_histogram

    except Exception as e:
        print(f"Error extracting histogram features: {e}")
        # Return default empty DataFrames
        return pd.DataFrame()
```

The two models that are used to classify the image feature are the k-nearest neighbour (KNN) and support vector machines (SVM). The MDC features and PIH feature undergo a clustering process before they are fed into the KNN model. The cluster process using the k-means algorithm to identify the images with similar features. The images are flattened, and a principal component analysis is performed to reduce the number of features. The cluster where the data points are the closest is selected and a new DataFrame is created that will be used for the model.

```python
def perform_pca_cluster(genre_df, genre):

    # Assuming create_image_array function is defined elsewhere and works correctly
    reshaped_list = create_image_array(genre_df)
    # Normalize and reshape images
    X_flat = np.array([img.reshape(3 * 3 * 3) / 255 for img in reshaped_list])

    # Apply PCA
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X_flat)

    # Apply KMeans clustering
    kmeans = KMeans(n_clusters=6, random_state=42)  # Adjust the number of clusters as needed
    kmeans.fit(X_flat)
    labels = kmeans.labels_

    # Plotting
    plt.figure(figsize=(10, 7))
    sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=labels, palette="viridis", s=100, alpha=0.7)
    plt.title(f"KMeans Clustering of Image Data (Reduced Dimensions) {genre}")
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.show()

    # Add the cluster labels to the dataframe
    genre_df['cluster'] = labels

    # Compute WCSS for each cluster
    wcss = compute_wcss(X_flat, labels, kmeans.cluster_centers_)

    # Identify the most compact cluster
    most_compact_cluster = np.argmin(wcss)
    #print(f"Most compact cluster: {most_compact_cluster} with WCSS = {wcss[most_compact_cluster]}")

    # Identify the largest cluster
    largest_cluster = genre_df['cluster'].value_counts().idxmax()
    #print(f"Largest cluster: {largest_cluster} with {genre_df['cluster'].value_counts().max()} points")

    # Subset the data based on the most compact cluster
    closest_df = genre_df[genre_df['cluster'] == most_compact_cluster]
    largest_df = genre_df[genre_df['cluster'] == largest_cluster]

    # Optionally, reset the index of the subset DataFrames
    closest_df = closest_df.reset_index(drop=True)
    largest_df = largest_df.reset_index(drop=True)

    # Apply t-SNE for visualization
    tsne = TSNE(n_components=2, random_state=42)
    images_tsne = tsne.fit_transform(X_flat)

    # Create a scatter plot
    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(images_tsne[:, 0], images_tsne[:, 1], c=labels, cmap='viridis', alpha=0.6)
    plt.colorbar(scatter)
    plt.title(f't-SNE Visualization of Image Clusters {genre}')
    plt.xlabel('t-SNE Component 1')
    plt.ylabel('t-SNE Component 2')
    plt.show()

    return closest_df, largest_df
```

The first model that is used is the KNN model. The image data is flattened the data. Prior to the data being fitted to the model a PCA and linear discriminant analysis (LDA) are performed on the data to reduce the number of features. The data is split into test and training subsets. Hyperparameter tunning is perform for the following parameters: number of neighbours, weights, algorithm, leaf size, and power. The best model is selected and the performed is assessed using accuracy, precision, recall, F1-score, and area under the curve (AUC).

```
def run_knn_model(data, y_labels, IMAGE_HEIGHT, IMAGE_WIDTH, CHANNELS, data_type, feature, SAMPLE_SI
    label_encoder = LabelEncoder()

    # Step 1: Flatten feature vectors
    if feature == 'RGB' or feature == 'His' or feature == 'Comb':
        X_flat = np.array([img.reshape(IMAGE_HEIGHT * IMAGE_WIDTH * CHANNELS) for img in data])
        COMPONENTS = ((IMAGE_HEIGHT * IMAGE_WIDTH) * CHANNELS)
    elif feature == 'RGB_Base':
        reshaped_list = mdc.create_image_array(data)
        X_flat = np.array([img.reshape(3 * 3 * 3) / 255 for img in reshaped_list])
        COMPONENTS = 8
    elif feature == 'His_Base':
        X_flat = data[['R1', 'G1', 'B1', 'R2', 'G2', 'B2', 'R3', 'G3', 'B3', 'R4', 'G4', 'B4', 'R5',
        X_flat = X_flat/255
        COMPONENTS = 8

    # Step 2: Normalize the data
    scaler = StandardScaler()
    X_flat_scaled = scaler.fit_transform(X_flat)

    # Step 3: Apply PCA and LDA
    pca = PCA(n_components=COMPONENTS)
    X_pca = pca.fit_transform(X_flat_scaled)

    pg.plot_pca_variance(X_pca)

    n_classes = len(np.unique(y_labels))
    n_components = min(20, min(n_classes - 1, COMPONENTS))
    #print(f"LDA Components for {data_type}: {n_components}")
    lda = LDA(n_components=n_components)
    X_lda = lda.fit_transform(X_pca, y_labels)

    pg.plot_lda_variance(n_components, X_pca, y_labels)

    # Step 4: Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X_lda, y_labels, test_size=0.2, random_state

    # Step 5: Grid Search for optimal k in k-NN
    param_grid = {
        'n_neighbors': [3, 5, 7, 9, 11, 13, 15, 17, 19, 21],  # Experiment with these values
        'weights': ['uniform', 'distance'],  # Weighting by distance or uniformly
        'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],  # Algorithm for nearest neighbors s
        'leaf_size': [10, 20, 30, 40, 50, 60, 70],  # Leaf size passed to BallTree or KDTree
        'p': [1, 2]  # Power parameter for the Minkowski metric (1: Manhattan distance, 2: Euclidean
    }
    grid_search = GridSearchCV(KNeighborsClassifier(metric='euclidean'), param_grid, cv=5, scoring=
    grid_search.fit(X_train, y_train)

    best_params = grid_search.best_params_
    print(f"Best parameters for {data_type}: {best_params}")

    best_knn = grid_search.best_estimator_  # Get the best model
    y_pred = best_knn.predict(X_test)

    X_preds_train = best_knn.predict(X_train)
    X_preds_test = best_knn.predict(X_test)

    # Step 6: Performance metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = 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')
    auc = roc_auc_score(y_test, best_knn.predict_proba(X_test), multi_class='ovr')
```

The second model the data is passed into is the SVM model. The correct predictions from the KNN model are used to train this model. Data augmentation and recursive feature elimination are performed before the data is fitted to the model. Hyperparameter tunning is perform on the following parameters: C, kernel, gamm, degree, and coef0. This model is trained using MDC, PIH, and a combination of the two. The output is three SVM models.

```python
def run_svm(X_data, y_labels, feature, GENRES):

    # Split the dataset into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X_data, y_labels, test_size=0.2, random_state=42)

    # Feature Scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Data Augmentation (SMOTE)
    smote = SMOTE(random_state=42, k_neighbors=1)
    X_train_augmented, y_train_augmented = smote.fit_resample(X_train_scaled, y_train)

    # Feature Selection using RFE
    rfe = RFE(estimator=LogisticRegression(), n_features_to_select=10)
    X_train_augmented = rfe.fit_transform(X_train_augmented, y_train_augmented)
    X_test_scaled = rfe.transform(X_test_scaled)

    # Parameter Tuning using Grid Search with Cross-Validation
    param_grid = {
        'C': [0.01, 0.1, 1, 10, 100],
        'kernel': ['linear', 'rbf', 'poly'],
        'gamma': ['scale', 'auto'],
        'degree': [2, 3, 4],
        'coef0': [0.0, 0.1, 0.5],
    }

    # Cross-Validation
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    svm_grid_search = GridSearchCV(SVC(random_state=42, probability=True), param_grid, cv=cv)
    svm_grid_search.fit(X_train_augmented, y_train_augmented)

    # Best parameters found by Grid Search
    print("Best Parameters for {}: {}".format(feature, svm_grid_search.best_params_))

    # Train the SVM model with the best parameters
    best_svm_model = svm_grid_search.best_estimator_
    best_svm_model.fit(X_train_augmented, y_train_augmented)

    # Make predictions on the test set
    y_pred = best_svm_model.predict(X_test_scaled)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy for {feature}: ", accuracy)

    # Classification Report
    print(f"Classification Report for {feature}:")
    print(classification_report(y_test, y_pred))

    return best_svm_model, X_test_scaled, y_test, accuracy
```

The final stage in the classification process is to pass the SVM models into an ensemble model. This model combines model into a singular model to help improve performance. The two best models are selected. The model's performance is assessed using precision, recall, F-1 score, accuracy, and AUC.

```python
def ensemble_model(X_test, y_test, combined_svm_model, rgb_svm_model, his_svm_model, concatenated_array, concatenated_labels, acc_combined, acc_rgb, acc_his):
    # Store models and their corresponding accuracy scores in a list
    models_with_acc = [
        ('combined_svm', combined_svm_model, acc_combined),
        ('rgb_svm', rgb_svm_model, acc_rgb),
        ('his_svm', his_svm_model, acc_his)
    ]

    # Sort models by their accuracy scores in descending order
    models_with_acc.sort(key=lambda x: x[2], reverse=True)

    # Select the top 2 models with the highest accuracy scores
    top_models = models_with_acc[:2]

    # Calibrate models for better probability estimates
    calibrated_models = [(name, CalibratedClassifierCV(model, method='sigmoid')) for name, model, _ in top_models]

    # Fit the calibrated models on the concatenated data
    for name, model in calibrated_models:
        model.fit(concatenated_array, concatenated_labels)

    # Create a VotingClassifier with the top 2 models
    voting_clf = VotingClassifier(
        estimators=[(calibrated_models[0][0], calibrated_models[0][1]),
                    (calibrated_models[1][0], calibrated_models[1][1])],
        voting='soft'  # Use soft voting to get probabilities
    )

    # Fit the VotingClassifier
    voting_clf.fit(concatenated_array, concatenated_labels)

    # Evaluate the ensemble model
    y_pred_ensemble = voting_clf.predict(X_test)
    accuracy_ensemble = accuracy_score(y_test, y_pred_ensemble)
    print(f"Ensemble Model Accuracy: {accuracy_ensemble}")
    print("Ensemble Model Classification Report:")
    print(classification_report(y_test, y_pred_ensemble))

    # Calculate the probabilities for the classes
    y_pred_ensemble_proba = voting_clf.predict_proba(X_test)

    auc_ensemble = roc_auc_score(y_test, y_pred_ensemble_proba, multi_class='ovr')
    print(f"Ensemble Model AUC: {auc_ensemble}")
```

The final stage in the process is the data evaluation. The classifation_report method from the SciKit-learn pack is used to assess the perform of each model. This report displays the accuracy, precision, recall, F1-score and AUC for each model. There are also some code sippets that plot the MDC and PIH feature on a graph

```
def plot_colours(colour_values, genre):
    fig, axes = plt.subplots(3, 3, figsize=(9, 9))
    fig.suptitle(f'Most Dominant Colours for {genre}', fontsize=16)

    # Plot each color in the 3x3 grid
    for ax, color in zip(axes.flatten(), colour_values):
        ax.axis('off')  # Turn off the axis
        ax.imshow([[color]], aspect='auto')  # Display the color as an image

    # Hide any remaining empty subplots if there are fewer than 9 colors
    for ax in axes.flatten()[len(colour_values):]:
        ax.axis('off')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()
```

```
def get_historgrams(df_sub):
    r_columns = ['mostFrequentGenre'] + [f"{i}_R" for i in range(256)]
    g_columns = ['mostFrequentGenre'] + [f"{i}_G" for i in range(256)]
    b_columns = ['mostFrequentGenre'] + [f"{i}_B" for i in range(256)]

    red_df = df_sub[r_columns]
    green_df = df_sub[g_columns]
    blue_df = df_sub[b_columns]

    dfs = []

    genres_sub = df_sub['mostFrequentGenre'].to_list()

    for genre in genres_sub:
        temp_df = pd.DataFrame()
        temp_df['Pixel Intensity'] = range(256)

        temp_red = red_df[red_df['mostFrequentGenre'] == genre]
        temp_green = green_df[blue_df['mostFrequentGenre'] == genre]
        temp_blue = blue_df[blue_df['mostFrequentGenre'] == genre]

        temp_red = temp_red.drop(temp_red.columns[0], axis=1)
        temp_green = temp_green.drop(temp_green.columns[0], axis=1)
        temp_blue = temp_blue.drop(temp_blue.columns[0], axis=1)

        row_to_add_r = temp_red.iloc[0]
        row_to_add_g = temp_green.iloc[0]
        row_to_add_b = temp_blue.iloc[0]

        new_column_red = row_to_add_r.to_frame().reset_index(drop=True)
        new_column_green = row_to_add_g.to_frame().reset_index(drop=True)
        new_column_blue = row_to_add_b.to_frame().reset_index(drop=True)

        temp_df['Red Frequency'] = new_column_red.iloc[:, 0]
        temp_df['Green Frequency'] = new_column_green.iloc[:, 0]
        temp_df['Blue Frequency'] = new_column_blue.iloc[:, 0]

        plot_histogram(temp_df, genre)
```