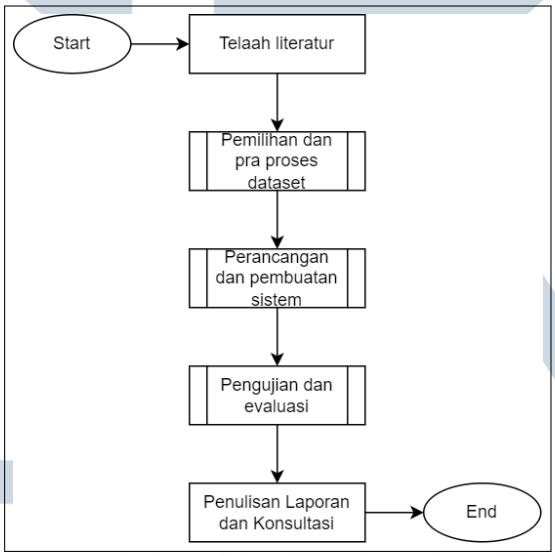
BAB 3
METODOLOGI PENELITIAN

3.1 Gambaran Umum Penelitian

Alur penelitian secara umum dapat dilihat pada Gambar 3.1. Tahap pertama yang dilakukan dari penelitian ini adalah melakukan telaah literatur. Tahap selanjutnya adalah pemilihan dan pra proses *dataset*. *Dataset* akan dipilih berdasarkan beberapa pertimbangan dan kemudian diolah agar dapat dipelajari oleh model *deep learning*. Kemudian, dilakukan perancangan dan pembuatan sistem untuk mengklasifikasi genre musik. Setelah sistem sudah selesai dibuat, sistem akan diuji dan dievaluasi apakah sudah bekerja dengan cukup baik. Setelah semua tahapan selesai, dilakukan penulisan laporan dari tahap awal hingga akhir. Konsultasi dengan pembimbing dilakukan pada setiap tahapan untuk meminta kritik dan saran agar penelitian ini dapat berjalan dengan baik.



Gambar 3.1. Diagram alur penelitian secara umum

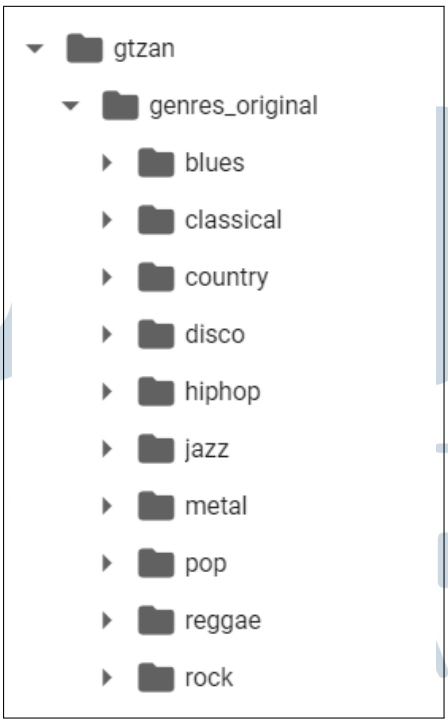
3.2 Telaah Literatur

Pada tahapan ini, dilakukan studi pustaka mengenai informasi dan teori yang relevan dengan sistem yang akan dibuat. Sumber yang digunakan untuk telaah literatur berasal dari buku dan jurnal elektronik. Informasi yang dipelajari pada penelitian ini yaitu penelitian-penelitian terdahulu mengenai klasifikasi genre

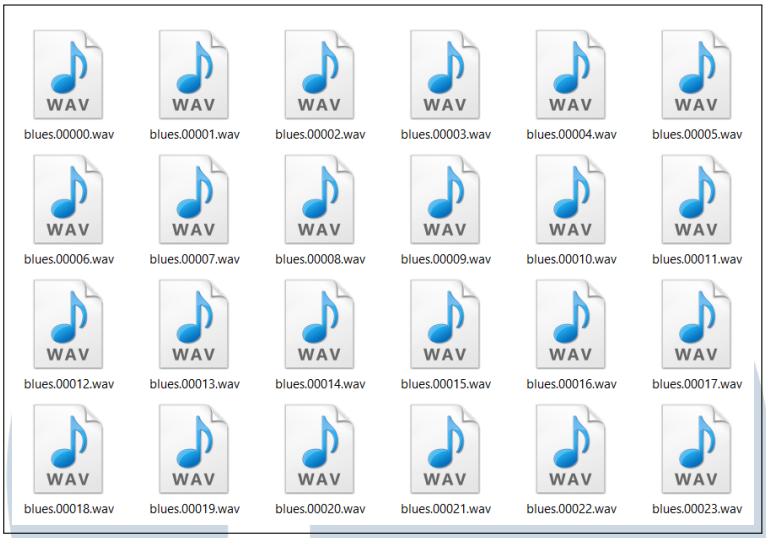
musik. Setelah melakukan studi pustaka, ditemukan bahwa gabungan algoritma CNN dan RNN atau algoritma CRNN menghasilkan akurasi yang lebih tinggi dibandingkan algoritma *deep learning* yang lain sehingga algoritma tersebut digunakan pada penelitian ini. *Feature extraction* yang digunakan untuk melatih model adalah MFCC. MFCC seringkali digunakan sebagai *feature extraction* pada penelitian yang berhubungan dengan audio.

3.3 Pemilihan dan Pra Proses Dataset

Pada tahap ini, dilakukan pencarian dan pemilihan *dataset* yang akan digunakan untuk melatih model pada penelitian ini. *Dataset* yang digunakan berisi *file* audio (seperti *.wav*) agar dapat diubah menjadi MFCC. Terdapat beberapa opsi *dataset* yang digunakan, seperti FMA (*Free Music Archive*) *dataset*, tetapi *dataset* yang digunakan adalah GTZAN *dataset*. Hal ini karena adanya keterbatasan penyimpanan dan sumber daya lainnya. GTZAN *dataset* adalah *dataset* yang cukup ringan karena hanya memuat 1000 cuplikan audio dengan durasi 30 detik. Total penyimpanan yang dibutuhkan GTZAN *dataset* kurang lebih 1,3 GB, sedangkan FMA *dataset* membutuhkan penyimpanan sebesar 7,2 GB.



Gambar 3.2. Isi dari GTZAN *dataset*



Gambar 3.3. Isi dari folder genre *jazz*

Berdasarkan pada Gambar 3.2, struktur *folder* ini terdiri dari 10 *folder*, antara lain *blues*, *classical*, *country*, *disco*, *hiphop*, *jazz*, *metal*, *pop*, *reggae*, dan *rock*. Pada masing-masing *folder*, terdapat 100 *file* audio dengan ekstensi *.wav* berdurasi 30 detik. Gambar 3.3 menampilkan contoh dari salah satu *folder*, yaitu *folder* musik bergenre *jazz*. Namun, salah satu *folder*, yaitu *folder jazz*, hanya memiliki 99 audio karena terdapat satu *file* yang rusak sehingga total data pada *dataset* ini hanya 999 lagu.

```
from sklearn.metrics import confusion_matrix, precision_score,
    recall_score, f1_score
import seaborn as sns
import os
import librosa
import math
import json
import matplotlib.pyplot as plt
import numpy as np
import itertools
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    BatchNormalization, Flatten, Bidirectional, LSTM, Dense,
    Dropout, Reshape, Activation

DATASET_PATH = "/content/drive/MyDrive/gtzan/genres_original"
JSON_PATH = "/content/drive/MyDrive/gtzan/data.json"
```

```
SAMPLE_RATE = 22050
DURATION = 30 # in seconds
SAMPLES_PER_TRACK = SAMPLE_RATE * DURATION
```

Kode 3.1: Potongan kode untuk mengimpor *library* dan *dataset*

Kode 3.1 merupakan potongan kode yang digunakan untuk mengimpor *dataset* dan *library* serta variabel konstan yang digunakan pada penelitian ini. Variabel "DATASET_PATH" berisi *path* atau tempat penyimpanan dari *dataset*. Variabel "JSON_PATH" berisi *path* atau tempat penyimpanan *file* JSON yang akan digunakan untuk melatih data. Variabel "SAMPLE_RATE" adalah variabel konstan untuk menyatakan *sample rate* dari sebuah audio. Variabel "DURATION" adalah variabel konstan untuk menyatakan durasi dari masing-masing audio, yaitu 30 detik. Variabel "SAMPLES_PER_TRACK" adalah variabel konstan yang digunakan untuk menghitung ada berapa *sample rate* dalam suatu *file* audio.

Setelah *dataset* ditentukan, audio kemudian akan diubah menjadi MFCC. Koefisien *cepstral* yang akan digunakan adalah 13 koefisien. Audio juga dipecah menjadi 10 segmen pada tahap ini untuk memperbanyak data yang akan digunakan untuk melatih model. Hal ini akan dilakukan ke semua audio dari semua *folder*. Hasil proses ini akan disimpan ke dalam bentuk *file* JSON yang berupa *array* berisikan 13 koefisien MFCC yang sudah diekstrak dari seluruh audio di dalam *dataset*.

```
def save_mfcc(dataset_path, json_path, n_mfcc=13, n_fft=2048,
    hop_length=512, num_segments=5):
  # store data
  data = {
    "mapping": [],
    "mfcc": [],
    "labels": []
  }

  num_samples_per_segment = int(SAMPLES_PER_TRACK / num_segments)
  expected_num_mfcc_vectors_per_segment = math.ceil(
    num_samples_per_segment / hop_length)

  # loop through all genres
  for i, (dirpath, dirnames, filenames) in enumerate(os.walk(
    dataset_path)):

    # ensure not at root level
    if dirpath is not dataset_path:
```

```
17
18      # save semantic label
19      dirpath_components = dirpath.split("/")
20      semantic_label = dirpath_components[-1]
21      data["mapping"].append(semantic_label)
22      print("\nProcessing: {}".format(semantic_label))
23
24      # process files for a specific genre
25      for f in filenames:
26        # load audio file
27        file_path = os.path.join(dirpath, f)
28        signal, sr = librosa.load(file_path, sr=SAMPLE_RATE)
29
30        # process segments extracting mfcc and storing data
31        for s in range(num_segments):
32          start_sample = num_samples_per_segment * s
33          finish_sample = start_sample + num_samples_per_segment
34
35          mfcc = librosa.feature.mfcc(y=signal[start_sample:
    finish_sample],
36                                      sr=sr,
37                                      n_fft=n_fft,
38                                      n_mfcc=n_mfcc,
39                                      hop_length=hop_length)
40          mfcc = mfcc.T
41
42          # store mfcc for segment if it has the expected length
43          if len(mfcc) == expected_num_mfcc_vectors_per_segment:
44            data["mfcc"].append(mfcc.tolist())
45            data["labels"].append(i-1)
46            print("{}, segment: {}".format(file_path, s+1))
47
48  with open(json_path, "w") as fp:
49    json.dump(data, fp, indent=4)
```

Kode 3.2: Fungsi yang digunakan untuk mengekstraksi MFCC dari audio

Kode 3.2 adalah potongan kode yang berisi fungsi untuk melakukan ekstraksi MFCC dari audio. Data akan disimpan dalam bentuk *array*, antara lain "*mapping*", "mfcc", dan "*labels*". Parameter *n_mfcc* adalah parameter untuk menentukan jumlah koefisien *cepstral* yang akan diekstrak. Jumlah koefisien yang diekstrak berkisar dari angka 12 sampai 20 karena kisaran tersebut memiliki informasi paling relevan pada audio, tetapi jumlah koefisien yang digunakan pada

penelitian ini adalah 13. Parameter *n_fft* adalah parameter untuk menentukan panjang jendela FFT yang akan diekstrak. Parameter *hop_length* adalah parameter yang menentukan jarak bergesernya jendela FFT. *For loop* yang pertama digunakan untuk melakukan *looping* pada genre-genre yang terdapat pada *folder* tersebut. Kondisi *if* pertama digunakan untuk memastikan bahwa *loop* sudah memasuki *folder* genre. Setelah itu, *dataset_path* akan dipotong menjadi *array* berdasarkan tanda "/" dan *array* terakhir akan diambil untuk disimpan pada *array* "*mapping*". *For loop* kedua digunakan untuk memproses *file* yang berada pada folder tersebut. Audio akan dimuat menggunakan salah satu fungsi milik Librosa, yaitu *librosa.load*. *For loop* ketiga digunakan untuk memotong audio berdasarkan jumlah segmen yang ditentukan. Variabel "start_sample" berfungsi sebagai penanda sampel dimulai dari mana dan variabel "finish_sample" berfungsi untuk menandai sampel berakhir sampai mana. Setelah itu, dilakukan ekstraksi koefisien MFCC dengan menggunakan fungsi dari Librosa, yaitu *librosa.feature.mfcc*. Fungsi ini memerlukan parameter antara lain sinyal yang akan diproses, *sample rate* (*sr*), panjang jendela FFT (*n_fft*), dan jarak pergeseran FFT (*hop_length*). Setelah proses ekstraksi selesai dan panjang dari MFCC sudah sesuai dengan jumlah MFCC yang diharapkan, maka hasil akan disimpan ke dalam *array* "mfcc". Setelah semua *file* pada seluruh *folder* telah selesai diekstraksi, *array* akan disimpan ke dalam *file* JSON.

```
# removed jazz.00054 (corrupted file)
# save mfcc to 10 segments
save_mfcc(DATASET_PATH, JSON_PATH, num_segments=10)
```

Kode 3.3: Potongan kode yang memanggil fungsi untuk menyimpan MFCC

Fungsi akan dipanggil seperti pada Kode 3.3. Saat memanggil kode ini, terdapat beberapa parameter yang dikirim, yaitu DATASET_PATH, JSON_PATH, dan *num_segments*. DATASET_PATH akan melakukan *overwrite* pada parameter *dataset_path*, JSON_PATH melakukan *overwrite* pada parameter *json_path*, dan *num_segments* diubah menjadi 10. Contoh *output* yang dihasilkan pada saat menjalankan fungsi ini dapat dilihat pada Gambar 3.4.

```
Processing: blues
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 1
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 2
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 3
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 4
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 5
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 6
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 7
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 8
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 9
/content/drive/MyDrive/gtzan_small/genres_original/blues/blues.00001.wav, segment: 10
```

Gambar 3.4. Hasil *output* ketika melakukan ekstraksi MFCC

3.4 Perancangan Sistem

Pada tahap ini, dilakukan perancangan dan pembuatan model yang akan digunakan untuk mengklasifikasi genre musik. Algoritma akan terbagi menjadi dua lapisan karena model merupakan gabungan dari algoritma CNN dan RNN (BiLSTM). Lapisan CNN akan terdiri dari tiga *convolution layer*, *max pooling layer*, dan *batch normalization*. Lapisan RNN akan terdiri dari dua lapisan BiLSTM. Setelah melewati lapisan CNN dan RNN, model akan dilakukan *flatten* terlebih dahulu. Kemudian, model akan masuk ke *fully connected layer* untuk mengkoneksikan semua neuron dengan *output layer*. Penelitian ini akan menguji dua rangkaian model, yaitu menggunakan *dropout* sebelum masuk ke lapisan BiLSTM dan tidak menggunakan *dropout*. Potongan kode pada Kode 3.4 adalah fungsi yang menggunakan *dropout* sebelum memasuki lapisan BiLSTM.

```
1  def build_model_crnn_do(input_shape):
2    # create model
3    model = Sequential()
4
5    # 1st conv layer
6    model.add(Conv2D(filters=32, kernel_size=(3, 3), activation="
       relu", input_shape=input_shape))
7    model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding
       ="same"))
8    model.add(BatchNormalization())
9
10   # 2nd conv layer
11   model.add(Conv2D(filters=32, kernel_size=(3, 3), activation="
       relu", input_shape=input_shape))
12   model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding
       ="same"))
13   model.add(BatchNormalization())
```

```
14
15  # 3rd conv layer
16  model.add(Conv2D(filters=32, kernel_size=(2, 2), activation="
      relu", input_shape=input_shape))
17  model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding
      ="same"))
18  model.add(BatchNormalization())
19
20  # dropout before reshape
21  model.add(Dropout(0.3))
22
23  # reshape model
24  model.add(Reshape((-1, model.output_shape[-1])))
25
26  # bilstm layer
27  model.add(Bidirectional(LSTM(units=64, return_sequences=True)))
28  model.add(Bidirectional(LSTM(units=64)))
29
30  # flatten output to 2d array & feed to dense/fully connected
      layer
31  model.add(Flatten())
32  model.add(Dense(units=64, activation="relu"))
33  model.add(Dropout(0.3))
34
35  # output layer
36  model.add(Dense(units=10, activation="softmax"))
37
38  # summary
39  model.summary()
40
41  return model
```

Kode 3.4: Fungsi untuk membuat model CRNN (dengan *dropout*)

Kode 3.4 adalah potongan kode yang berisi fungsi untuk membuat model CRNN. Fungsi aktivasi yang digunakan pada lapisan CNN adalah fungsi aktivasi ReLU. ReLU lebh efisien dibandingkan dengan fungsi aktivasi lainnya karena ReLU tidak mengaktifkan semua neuronnya. Selain itu, ReLU lebih cepat dalam mencapai konvergensi dan dapat mengurangi isu *vanishing gradient*. Jumlah filter yang digunakan untuk *convolution layer* pada CNN adalah 32 pada seluruh lapisan. Ukuran *kernel* yang digunakan untuk *convolutional layer* adalah 3×3 pada dua lapisan pertama dan 2×2 pada lapisan ketiga. Pada *max pooling layer*, ukuran *kernel* yang digunakan adalah 3×3 untuk dua lapisan pertama dan 2×2 untuk

lapisan terakhir. Jarak pergeseran atau *stride* yang digunakan berukuran 2×2. Karena operasi *max pooling* yang digunakan, maka nilai maksimum yang akan diambil dalam *kernel*. *Batch normalization* digunakan pada model karena *batch normalization* dapat mencegah adanya perubahan secara tiba-tiba pada tiap *layer*. Nilai *dropout* yang digunakan sebesar 0,3 atau 30%. *Dropout* dilakukan untuk mencegah terjadinya *overfitting* karena *dropout* menghilangkan neuron secara acak.

Algoritma yang digunakan pada lapisan RNN adalah BiLSTM dengan jumlah filter sebesar 64. Lapisan pertama BiLSTM menggunakan *return sequences* agar *output* pada setiap *timestep* akan diteruskan ke lapisan selanjutnya. Lapisan kedua BiLSTM tidak menggunakan *return sequences* karena *output* yang ingin dimasukkan ke *fully connected layer* hanya dari *timestep* yang paling terakhir. *Output* dari lapisan CNN dan BiLSTM tersebut akan diubah menjadi satu dimensi melalui proses *flatten* agar dapat diproses oleh *fully connected layer*. Jumlah filter yang digunakan untuk *fully connected layer* adalah 64 dengan fungsi aktivasi ReLU yang kemudian diikuti dengan *dropout* sebesar 30%. Setelah itu, hasil akan masuk ke *output layer* dengan fungsi aktivasi *softmax* dan jumlah filter sebanyak 10 karena filter mengikuti jumlah kelas untuk klasifikasi.

3.5 Pembuatan Sistem

Pada tahap ini, dilakukan pelatihan model agar dapat melakukan klasifikasi. *Dataset* akan dibagi menjadi data *training* sebesar 60%, data validasi sebesar 20%, dan data *testing* sebesar 20%. Data validasi digunakan untuk mengevaluasi model selama pelatihan, sedangkan data *testing* adalah data yang sama sekali tidak digunakan baik dalam pelatihan, maupun evaluasi. Data *testing* hanya akan dipakai untuk melakukan prediksi menggunakan model yang sudah dibuat.

```
def load_data(dataset_path):
  with open(dataset_path, "r") as fp:
    data = json.load(fp)

  # convert from json to numpy arrays
  X = np.array(data["mfcc"])
  y = np.array(data["labels"])

  return X, y

def prepare_dataset_crnn(test_size, validation_size):
  # load data
```

```
13  X, y = load_data(JSON_PATH)
14  print("X.shape", X.shape)
15  print("y.shape", y.shape)
16
17  # create train test split
18  X_train, X_test, y_train, y_test = train_test_split(X, y,
      test_size=test_size)
19
20  # create train val split
21  X_train, X_val, y_train, y_val = train_test_split(X_train,
      y_train, test_size=validation_size)
22
23  print("\nX_train.shape before", X_train.shape)
24  print("X_val.shape before", X_val.shape)
25  print("X_test.shape before", X_test.shape)
26
27  # change 3d array to 4d array
28  X_train = X_train[..., np.newaxis]
29  X_val = X_val[..., np.newaxis]
30  X_test = X_test[..., np.newaxis]
31
32  print("\nX_train.shape after", X_train.shape)
33  print("X_val.shape after", X_val.shape)
34  print("X_test.shape after", X_test.shape)
35  print("\n==================================\n")
36
37  return X_train, X_val, X_test, y_train, y_val, y_test
```

Kode 3.5: Potongan kode fungsi *load_data* dan *prepare_dataset_crnn*

Kode 3.5 adalah potongan kode yang digunakan untuk membagi *dataset* menjadi tiga data, yaitu data *training*, data validasi, dan data *testing*. Pertama, fungsi *prepare_dataset_crnn* akan memanggil fungsi *load_data*. Fungsi *load_data* digunakan untuk memuat *dataset* ke dalam bentuk *np.array*. Variabel "X" digunakan untuk menampung isi dari *array* "mfcc" sedangkan variabel "y" digunakan untuk menampung isi dari *array* "*labels*". Ketika sudah selesai dimuat, *dataset* dibagi menjadi data *training* dan data *testing* terlebih dahulu, kemudian dari data *training* dibagi lagi sehingga didapatkan data validasi.

```
1  # create train, val, test set
2  X_train, X_val, X_test, y_train, y_val, y_test =
       prepare_dataset_crnn(0.2, 0.25)
3
4  # build crnn net
```

```
input_shape = (X_train.shape[1], X_train.shape[2], 1)
model = build_model_crnn_do(input_shape)

# compile model (learning rate 0.001)
optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer, loss="
    sparse_categorical_crossentropy", metrics=["accuracy"])

# train model
history = model.fit(X_train, y_train, validation_data=(X_val,
    y_val), batch_size=32, epochs=100)
```

Kode 3.6: Potongan kode untuk melatih model CRNN

Kode 3.6 adalah potongan kode yang memanggil fungsi *prepare_dataset_crnn* dan *build_model_crnn* untuk melatih model CRNN. Pembagian *dataset* dilakukan dengan memanggil fungsi *prepare_dataset_crnn* dan memberikan parameter yaitu 0,2 atau 20% dari seluruh data untuk data *testing* dan 0,25 atau 25% dari data *training* untuk data validasi. Setelah itu, dilakukan pembuatan model dengan memanggil fungsi *build_model_crnn*. *Input shape* yang digunakan diambil dari bentuk atau *shape* dari *array* variabel "X_train". Setelah model sudah didefinisikan, dilakukan konfigurasi *optimizer*, fungsi *loss*, dan *metrics* yang digunakan menggunakan *model.compile*. *Optimizer* yang digunakan adalah Adam dengan beberapa parameter *learning rate* yang akan diuji, yaitu 0,1, 0,01, 0,001 dan 0,0001. Fungsi *loss* yang digunakan adalah *sparse categorical crossentropy*. *Metrics* yang digunakan adalah *accuracy* atau akurasi karena performa model dinilai berdasarkan akurasi. Setelah itu, model dilatih menggunakan *model.fit* dengan *batch size* sebanyak 32 dan *epoch* sebanyak 100. Riwayat hasil akurasi dan *loss* yang didapatkan selama pelatihan disimpan dalam variabel "history".

3.6 Pengujian dan Evaluasi

Pada tahap ini, dilakukan pengujian dan evaluasi model. Hal ini dilakukan untuk melihat hasil performa dari model, apakah model tersebut sudah bekerja dengan baik. Pertama, model dievaluasi untuk melihat akurasi dan *loss* yang dihasilkan model. Setelah itu, dilakukan pengujian model untuk melihat apakah model berhasil memprediksi dengan baik. Hasil yang dilihat antara lain *precision*, *recall*, dan F1 *score* dari model tersebut.

```
def plot_history(history):
  fig, axs = plt.subplots(2)

  # create accuracy subplot
  axs[0].plot(history.history["accuracy"], label="train accuracy")
  axs[0].plot(history.history["val_accuracy"], label="test
    accuracy")
  axs[0].set_ylabel("Accuracy")
  axs[0].legend(loc="lower right")

  # create error subplot
  axs[1].plot(history.history["loss"], label="train loss")
  axs[1].plot(history.history["val_loss"], label="test loss")
  axs[1].set_ylabel("Loss")
  axs[1].set_xlabel("Epoch")
  axs[1].legend(loc="upper right")

  plt.show()

def calculate_metrics(model, X_test, y_test):
  # Predict labels
  y_pred = model.predict(X_test)
  # Convert predictions to one-hot encoded labels
  y_pred_labels = np.argmax(y_pred, axis=1)
  # Convert numeric labels to genre labels
  y_test_labels = y_test

  # Calculate precision
  precision = precision_score(y_test_labels, y_pred_labels,
    average='weighted')
  # Calculate recall
  recall = recall_score(y_test_labels, y_pred_labels, average='
    weighted')
  # Calculate F1-score
  f1 = f1_score(y_test_labels, y_pred_labels, average='weighted')

  return precision, recall, f1

def plot_matrix(model, X_test, y_test):
  with open(JSON_PATH, 'r') as f:
    data = json.load(f)
    mapping = data['mapping']

```

```
    # Predict labels
    y_pred = model.predict(X_test)
    # Convert predictions to one-hot encoded labels
    y_pred_labels = np.argmax(y_pred, axis=1)
    # Generate confusion matrix
    cm = confusion_matrix(y_test, y_pred_labels)

    # Plot confusion matrix
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=
      mapping, yticklabels=mapping)
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title('Confusion Matrix')
    plt.show()
```

Kode 3.7: Potongan kode fungsi *load_data*, *plot_history*, *calculate_metrics*, dan *plot_matrix*

Kode 3.7 adalah potongan kode yang terdiri dari beberapa fungsi yang digunakan untuk melakukan training data. Fungsi *plot_history* digunakan untuk melakukan *plotting* terhadap akurasi dan *loss* yang didapatkan selama proses *training* atau pelatihan. Fungsi *calculate_metrics* digunakan untuk menghitung *precision*, *recall*, dan F1 *score* dari hasil prediksi menggunakan data *testing*. Fungsi *plot_matrix* digunakan untuk mencetak *confusion matrix* pada saat melakukan prediksi menggunakan data *testing*.

```
# evaluate on test set
test_err, test_accuracy = model.evaluate(X_test, y_test, verbose
    =1)
print("Loss/err on test set: ", test_err)
print("Accuracy on test set: ", test_accuracy)
print("\n")

# precision, recall, f1 score
precision, recall, f1 = calculate_metrics(model, X_test, y_test)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
print("\n")

# plot accuracy/error on training and validation set
plot_history(history)

print("\n\n")

```

```
# plot confusion matrix
plot_matrix(model, X_test, y_test)
```

Kode 3.8: Potongan kode untuk mengevaluasi model CRNN

Kode 3.8 adalah potongan kode yang memanggil fungsi *plot_history*, *calculate_metrics*, dan *plot_matrix* untuk menguji dan mengevaluasi model. Variabel ”history” yang menyimpan riwayat hasil pelatihan model akan direpresentasikan menggunakan grafik, yaitu grafik akurasi dan grafik *loss*. Kedua grafik tersebut menampilkan akurasi (grafik akurasi) dan *loss* (grafik *loss*) yang diperoleh dari data *training* dan data validasi. Setelah itu, model dievaluasi dengan data *testing* menggunakan *model.evaluate*. Hasil yang didapatkan dari proses ini adalah akurasi dari data *testing*. Fungsi *calculate_metrics* juga dipanggil untuk mendapatkan *precision*, *recall*, dan F1 *score* dari model. Parameter yang dimasukkan ke fungsi tersebut yaitu model dan data *testing*. Fungsi *plot_matrix* digunakan untuk melihat hasil klasifikasi yang dilakukan model dengan menggunakan *confusion matrix*.

3.7 Penulisan Laporan dan Konsultasi

Pada tahap ini, dilakukan penulisan laporan dari tahap awal proses hingga akhir proses penelitian. Konsultasi dengan pembimbing juga dilakukan untuk mengetahui apakah ada kesalahan dalam proses penelitian dan penulisan laporan. Hal ini dilakukan agar dapat menyelesaikan masalah tersebut.

3.8 Spesifikasi Sistem

Berikut merupakan perangkat keras yang digunakan untuk membuat model klasifikasi genre musik.

1. *Processor*: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz (8 CPUs), ∼2.1GHz
2. RAM: 8 GB
3. GPU: NVIDIA GeForce MX250
4. Storage: SSD 512 GB
5. OS: Windows 10 Home 64-bit

Berikut merupakan perangkat lunak dan *library* yang digunakan untuk membuat model klasifikasi genre musik.

1. Google Chrome Version 124.0.6367.61 (Official Build) (64-bit)
2. Google Colab (GPU: Tesla T4, GPU RAM: 15GB, System RAM: 12,7GB)
3. Python Version 3.10.12
4. Librosa Version 0.10.1
5. Tensorflow
6. Keras
7. JSON
8. Numpy
9. Matplotlib
10. Seaborn