Modeling

In [15]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# sns.set_theme(style="darkgrid")


from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

import tensorflow as tf
```

So now we can read in the data and work on a random forest classifier, since we are working with a categorical target.

In [16]:
```
df = pd.read_csv('data/esrb_ratings_scraped.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12000 entries, 0 to 11999
Data columns (total 30 columns):
 #   Column                    Non-Null Count  Dtype 
---  ------                    --------------  ----- 
 0   title                     12000 non-null  object
 1   console                   12000 non-null  object
 2   alcohol_reference         12000 non-null  int64 
 3   animated_blood            12000 non-null  int64 
 4   blood                     12000 non-null  int64 
 5   blood_and_gore            12000 non-null  int64 
 6   cartoon_violence          12000 non-null  int64 
 7   crude_humor               12000 non-null  int64 
 8   drug_reference            12000 non-null  int64 
 9   fantasy_violence          12000 non-null  int64 
 10  intense_violence          12000 non-null  int64 
 11  language                  12000 non-null  int64 
 12  mild_blood                12000 non-null  int64 
 13  mild_cartoon_violence     12000 non-null  int64 
 14  mild_fantasy_violence     12000 non-null  int64 
 15  mild_language             12000 non-null  int64 
 16  mild_lyrics               12000 non-null  int64 
 17  mild_suggestive_themes    12000 non-null  int64 
 18  mild_violence             12000 non-null  int64 
 19  nudity                    12000 non-null  int64 
 20  sexual_content            12000 non-null  int64 
 21  sexual_themes             12000 non-null  int64 
 22  simulated_gambling        12000 non-null  int64 
 23  strong_language           12000 non-null  int64 
 24  strong_sexual_content     12000 non-null  int64 
 25  suggestive_themes         12000 non-null  int64 
 26  use_of_alcohol            12000 non-null  int64 
 27  use_of_drugs_and_alcohol  12000 non-null  int64 
 28  violence                  12000 non-null  int64 
 29  esrb_rating               12000 non-null  object
dtypes: int64(27), object(3)
memory usage: 2.7+ MB
```

Drop duplicates

In [17]:
```
df = df.drop_duplicates(keep='first')
```

Select our features and target

In [18]:
```
features = df.drop(['title', 'console', 'esrb_rating'], axis=1)
target = df['esrb_rating']
```

split data into training and test data

In [19]:
```
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, ra
```

In our first few iterations we received about 88% accuracy. I then scraped data from ESRB.org to increase the size of the dataset to 6x the size. I reran this notebook with that data and our accuracy dropped to 85%. Let's incorporate SMOTE to see if that can improve.

In [20]:
```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=39)

X_train_res, y_train_res = smote.fit_resample(X_train, y_train)
```

Now let's initialize our random forest and fit the data.

In [21]:
```
rfc_model = RandomForestClassifier(n_estimators=100, random_state=39)

rfc_model.fit(X_train_res, y_train_res)
```

Out[21]:
```
▾ RandomForestClassifier

RandomForestClassifier(random_state=39)
```

and then let's see how our model works out of the box.

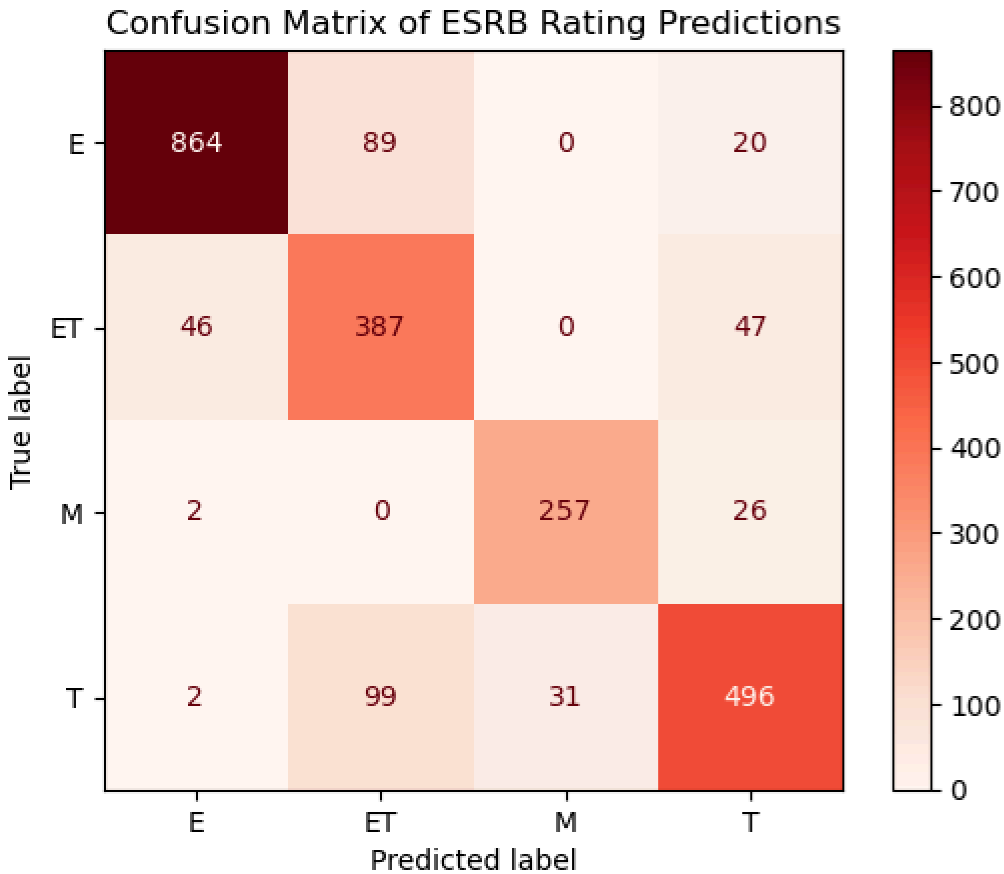
In [22]:
```
y_pred = rfc_model.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.8469991546914624
Classification Report:
               precision    recall  f1-score   support

           E       0.95      0.89      0.92       973
          ET       0.67      0.81      0.73       480
           M       0.89      0.90      0.90       285
           T       0.84      0.79      0.82       628

    accuracy                           0.85      2366
   macro avg       0.84      0.85      0.84      2366
weighted avg       0.86      0.85      0.85      2366
```

Some of the classes performed better than the others in terms of recall and precision. Let's take a look at this.

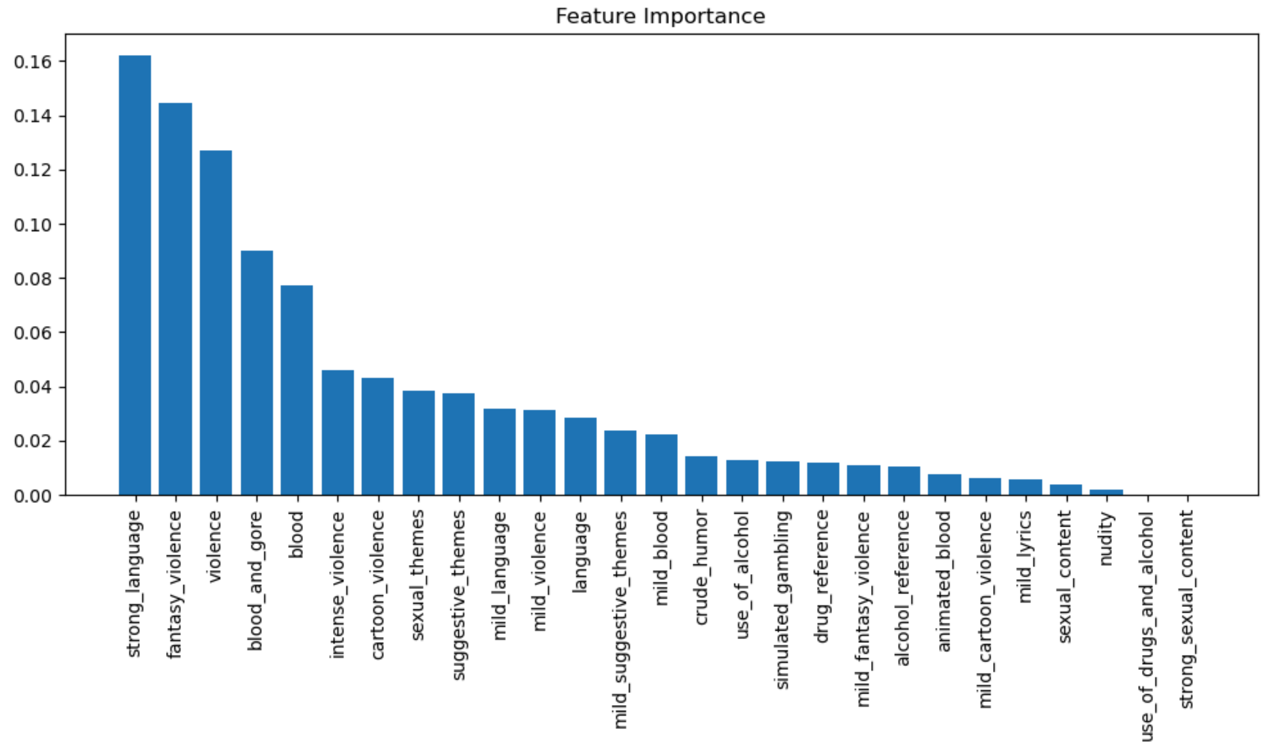
In [23]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

In [24]:
```
# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=rfc_model.classes_)

# Plot the confusion matrix
graph = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rfc_model.classes_)
graph.plot(cmap=plt.cm.Reds)

# Set title and show plot
plt.title("Confusion Matrix of ESRB Rating Predictions")
plt.show()
```



Let's try and improve on this model. We can start with checking out feature importance, since we have around 30 features.

In [25]:
```
importances = rfc_model.feature_importances_
feature_names = features.columns
sorted_indices = importances.argsort()[::-1]

# Plot the feature importance
plt.figure(figsize=(10, 6))
plt.title("Feature Importance")
plt.bar(range(len(sorted_indices)), importances[sorted_indices], align='center')
plt.xticks(range(len(sorted_indices)), feature_names[sorted_indices], rotation=90)
plt.tight_layout()
plt.show()
```



Let's try dropping all features that fall belove 0.01

- After trying this step, this results were not warranted. Keeping all features. Moving forward

In [26]:
```
# # Create a mask for features with importance greater than or equal to 0.01
# mask = importances >= 0.01

# # Filter the features based on the mask
# important_features = feature_names[mask]

# # Drop the features with importance below 0.01 from the dataset
# X_train_reduced = X_train[important_features]
# X_test_reduced = X_test[important_features]
```

In [27]:
```
# model.fit(X_train_reduced, y_train)

# y_pred = model.predict(X_test_reduced)

# # Evaluate the model
# print("Accuracy:", accuracy_score(y_test, y_pred))
# print("Classification Report:\n", classification_report(y_test, y_pred))
```

Our accuracy dropped from that. Let's forgo that and work on another path. We can try a GridSearch

In [28]:
```
from sklearn.model_selection import GridSearchCV

# param_grid = {
#     'n_estimators': [100, 200, 300],
#     'max_depth': [None, 10, 20, 30],
#     'min_samples_split': [2, 5, 10],
#     'min_samples_leaf': [1, 2, 4]
# }

best_params_grid = {'max_depth': [None], 'min_samples_leaf': [2], 'min_samples_split':
```

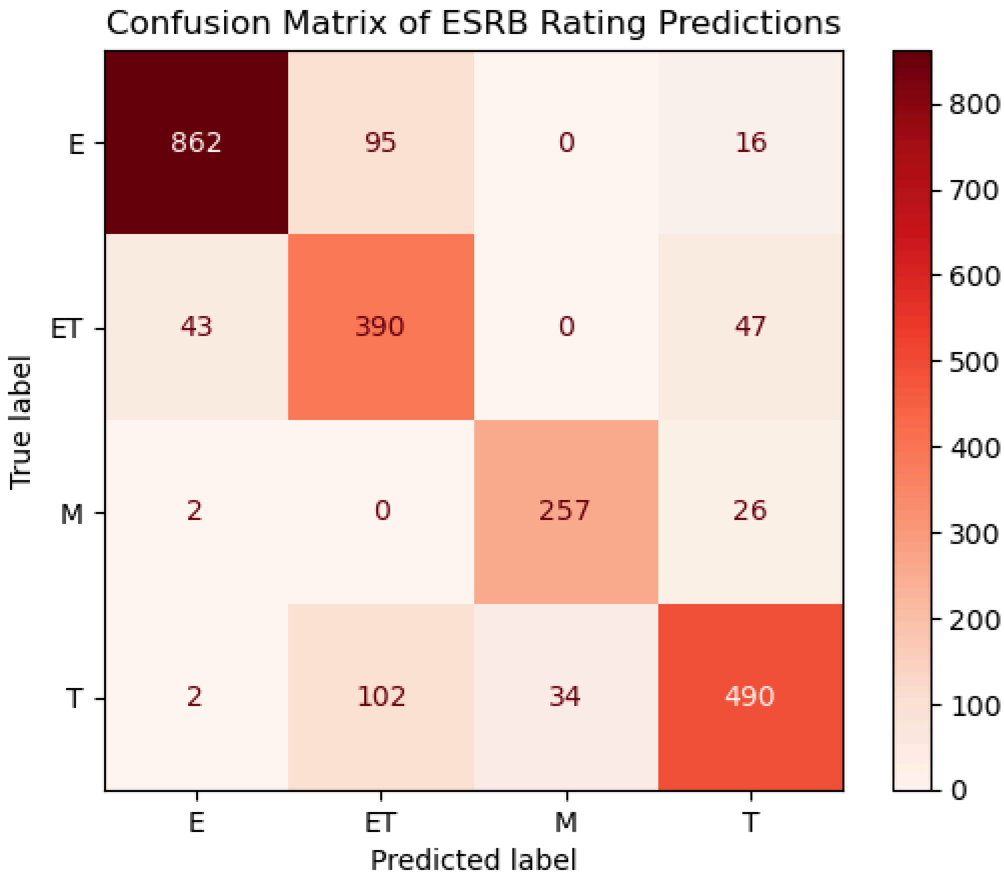
```
grid_search = GridSearchCV(estimator=rfc_model, param_grid=best_params_grid, cv=5, n_jo
grid_search.fit(X_train_res, y_train_res)
print("Best Parameters:", grid_search.best_params_)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
Best Parameters: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_e
stimators': 300}
```

In [29]:
```
y_grid_pred = grid_search.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_grid_pred))
print("Classification Report:\n", classification_report(y_test, y_grid_pred))
```

```
Accuracy: 0.8448858833474218
Classification Report:
               precision    recall  f1-score   support

           E       0.95      0.89      0.92       973
          ET       0.66      0.81      0.73       480
           M       0.88      0.90      0.89       285
           T       0.85      0.78      0.81       628

    accuracy                           0.84      2366
   macro avg       0.84      0.85      0.84      2366
weighted avg       0.86      0.84      0.85      2366
```

In [30]:
```
# Generate the confusion matrix
gridCM = confusion_matrix(y_test, y_grid_pred, labels=grid_search.classes_)

# Plot the confusion matrix
graph = ConfusionMatrixDisplay(confusion_matrix=gridCM, display_labels=grid_search.clas
graph.plot(cmap=plt.cm.Reds)

# Set title and show plot
plt.title("Confusion Matrix of ESRB Rating Predictions")
plt.show()
```



We got the exact same numbers. Maybe XGBClassifier will yield improvements?

In [31]:
```
# Run this cell if you need this library
# !pip install xgboost
```

We need to encode our target since XGBclassifier is looking for numeric data.

- We originally applied smote, but I am going to try PCA instead this time around.

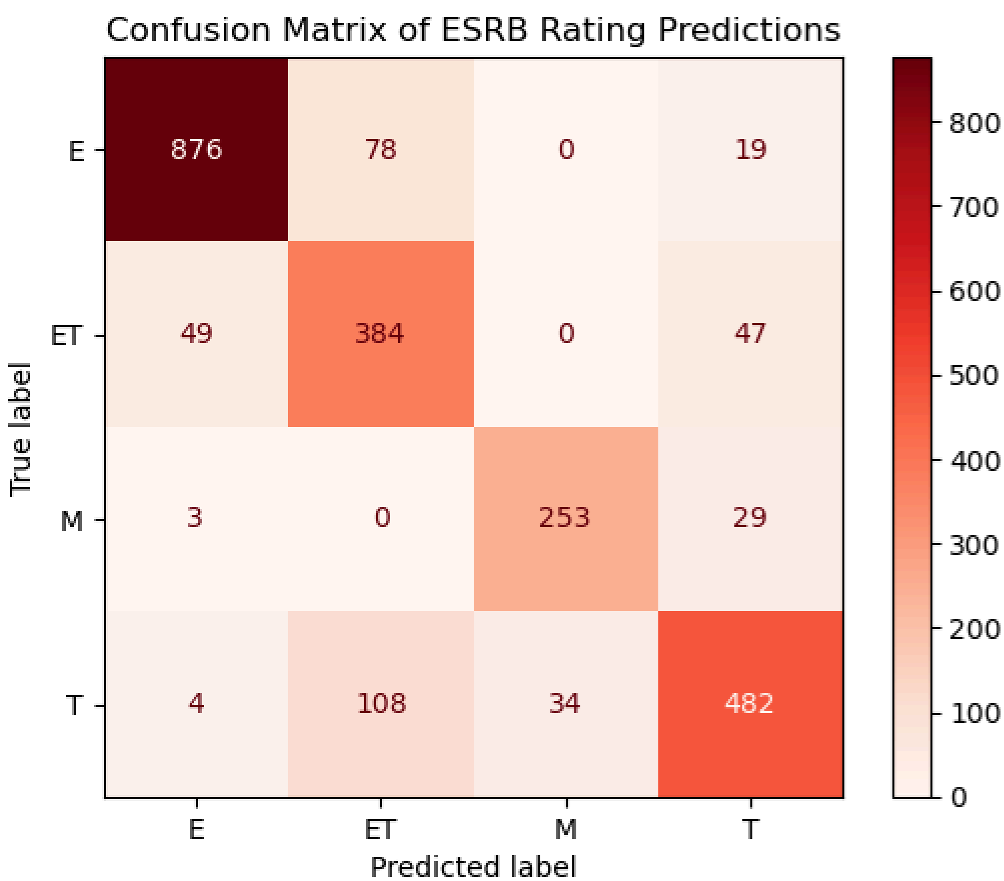
In [32]:
```
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder

# Encode the ESRB ratings as numeric values
le = LabelEncoder()
y_encoded = le.fit_transform(target)  # Convert 'E', 'T', etc. to numbers

# Split the data into training and testing sets (80% train, 20% test)
X_train_boost, X_test_boost, y_train_boost, y_test_boost = train_test_split(features, y

smote=SMOTE(random_state=39)
X_train_boost_res, y_train_boost_res = smote.fit_resample(X_train_boost, y_train_boost)

xgb_clf = XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=5)
xgb_clf.fit(X_train_boost_res, y_train_boost_res)
y_pred_xgb = xgb_clf.predict(X_test_boost)
```

In [33]:
```
# Evaluate the model
print("Accuracy:", accuracy_score(y_test_boost, y_pred_xgb))
print("Classification Report:\n", classification_report(y_test_boost, y_pred_xgb))
```

```
Accuracy: 0.8431952662721893
Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.90      0.92       973
           1       0.67      0.80      0.73       480
           2       0.88      0.89      0.88       285
           3       0.84      0.77      0.80       628

    accuracy                           0.84      2366
   macro avg       0.83      0.84      0.83      2366
weighted avg       0.85      0.84      0.85      2366
```

Accuracy dropped about 3% with PCA

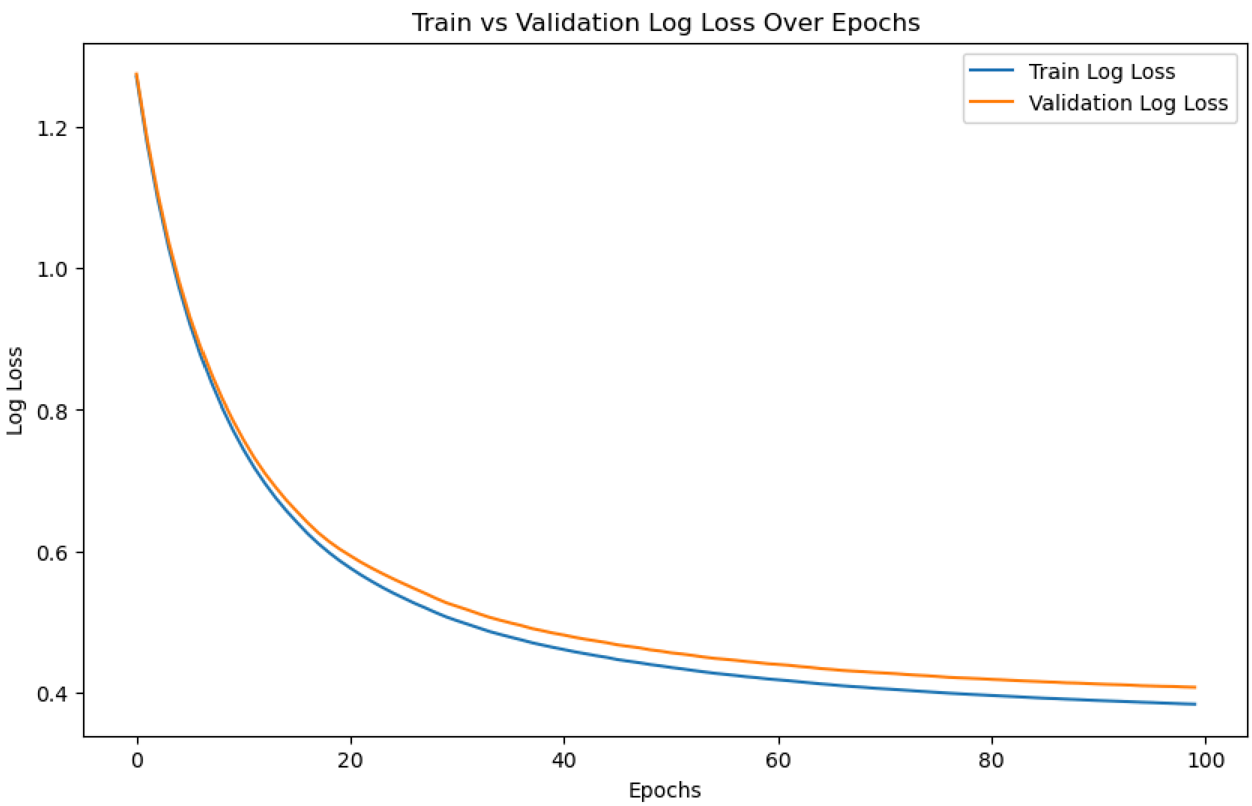
In [34]:
```python
# Generate the confusion matrix
xgbCM = confusion_matrix(y_test_boost, y_pred_xgb)

# Plot the confusion matrix
graph = ConfusionMatrixDisplay(confusion_matrix=xgbCM, display_labels=grid_search.class
graph.plot(cmap=plt.cm.Reds)

# Set title and show plot
plt.title("Confusion Matrix of ESRB Rating Predictions")

plt.savefig('graphs/cfm_xgboost.png', bbox_inches = 'tight', edgecolor='w')
plt.show()
```



In [35]:
```python
# Define the evaluation set (training and validation data)
eval_set = [(X_train_boost_res, y_train_boost_res), (X_test_boost, y_test_boost)]
```

```
# Train the model and track performance
xgb_clf.fit(X_train_boost_res, y_train_boost_res, eval_set=eval_set, verbose=False)

# Retrieve performance metrics
results = xgb_clf.evals_result()

print('Available metrics:', results['validation_0'].keys())

# Extract logloss for training and validation (assuming mlogloss is available for multi
train_metric = results['validation_0']['mlogloss']  # or use 'logloss' for binary class
test_metric = results['validation_1']['mlogloss']


# Plot the training and validation metric over epochs
epochs = len(train_metric)
x_axis = range(0, epochs)

plt.figure(figsize=(10, 6))

plt.plot(x_axis, train_metric, label='Train Log Loss')
plt.plot(x_axis, test_metric, label='Validation Log Loss')

# Set the plot details
plt.xlabel('Epochs')
plt.ylabel('Log Loss')
plt.title('Train vs Validation Log Loss Over Epochs')
plt.legend()
plt.show()
```

```
Available metrics: odict_keys(['mlogloss'])
```



Doesn't seem to be much in the way of over or underfitting. We aren't getting too much difference. It seems be to dropping as we try more. So let's move onto something more advanced.

- Adding PCA definitely made this graph worse. We won't go forward with this.

Advanced Machine Learning

Let's now try to work on a neural network.

- First iteration: 3 Dense layers.
- Second iteration: changed learning rate to 0.0001
- Third iteration: Added another Dense layer and a dropout layer.
- Fourth iteration: We now have 12k rows of data. We added smote. Accuracy is lower at 85% and stays there very early on in the process.
- Fifth iteration: Lowering epoch size since learning is stopping. Adding another Dense layer.
- Sixth iteration: Adding a regularizer to help prevent overfitting

In [36]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization, LeakyReLU
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.regularizers import l2

# # Encode the ESRB ratings as numeric values
# le = LabelEncoder()
# y_encoded = le.fit_transform(target)  # Convert 'E', 'T', etc. to numbers

# Split the data into training and testing sets (80% train, 20% test)
# X_train_boost, X_test_boost, y_train_boost, y_test_boost = train_test_split(features,

# smote=SMOTE(random_state=39)
# X_train_boost_res, y_train_boost_res = smote.fit_resample(X_train_boost, y_train_boos

# # Encode the target variable as integers
# y_train_encoded = le.fit_transform(y_train_boost)
# y_test_encoded = le.transform(y_test_boost)


# One-hot encode the target variable for use in categorical cross-entropy
y_train_one_hot_res = to_categorical(y_train_boost_res, num_classes=len(le.classes_))
y_test_one_hot = to_categorical(y_test_boost, num_classes=len(le.classes_))


nn_model = Sequential()
nn_model.add(Dense(256, input_dim=X_train.shape[1], activation='relu', kernel_regulariz
nn_model.add(Dense(128, activation='relu'))
nn_model.add(Dense(64, activation='relu'))
nn_model.add(BatchNormalization())
nn_model.add(Dropout(0.7)) # dropout some data to help with overfitting
nn_model.add(Dense(32, activation='relu'))
nn_model.add(Dense(16, activation='relu'))
nn_model.add(Dense(len(le.classes_), activation='softmax'))  # Output layer for classif

# Set a custom learning rate for the Adam optimizer
learning_rate = 0.0005  # You can adjust this value as needed
optimizer = Adam(learning_rate=learning_rate)
```

```
nn_model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accura


nn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               7168

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 64)                8256

 batch_normalization (BatchN  (None, 64)               256
 ormalization)

 dropout (Dropout)           (None, 64)                0

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 16)                528

 dense_5 (Dense)             (None, 4)                 68

```



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               7168

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 64)                8256

 batch_normalization (BatchN  (None, 64)               256
 ormalization)

 dropout (Dropout)           (None, 64)                0

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 16)                528

 dense_5 (Dense)             (None, 4)                 68

```



```
=================================================================
Total params: 51,252
Trainable params: 51,124
Non-trainable params: 128
_________________________________________________________________
```

X_train_boost, X_test_boost, y_train_boost, y_test_boost = train_test_split(features, y_encoded, test_size=0.2, random_state=39)

smote=SMOTE(random_state=39) X_train_boost_res, y_train_boost_res = smote.fit_resample(X_train_boost, y_train_boost)

In [37]:

```
#train model
history = nn_model.fit(X_train_boost_res, y_train_one_hot_res, epochs=75, batch_size=16
```

```
Epoch 1/75
922/922 [==============================] - 8s 8ms/step - loss: 1.4837 - accuracy: 0.7408
```

```
- val_loss: 0.5299 - val_accuracy: 0.8390
Epoch 2/75
922/922 [==============================] - 7s 8ms/step - loss: 0.5978 - accuracy: 0.7998
- val_loss: 0.4730 - val_accuracy: 0.8385
Epoch 3/75
922/922 [==============================] - 8s 9ms/step - loss: 0.5367 - accuracy: 0.8084
- val_loss: 0.4820 - val_accuracy: 0.8377
Epoch 4/75
922/922 [==============================] - 9s 9ms/step - loss: 0.5249 - accuracy: 0.8142
- val_loss: 0.4432 - val_accuracy: 0.8390
Epoch 5/75
922/922 [==============================] - 9s 9ms/step - loss: 0.5041 - accuracy: 0.8145
- val_loss: 0.4872 - val_accuracy: 0.8254
Epoch 6/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4880 - accuracy: 0.819
3 - val_loss: 0.4422 - val_accuracy: 0.8326
Epoch 7/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4940 - accuracy: 0.8212
- val_loss: 0.4273 - val_accuracy: 0.8381
Epoch 8/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4854 - accuracy: 0.8244
- val_loss: 0.4711 - val_accuracy: 0.8221
Epoch 9/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4784 - accuracy: 0.826
9 - val_loss: 0.4321 - val_accuracy: 0.8381
Epoch 10/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4726 - accuracy: 0.8266
- val_loss: 0.4627 - val_accuracy: 0.8280
Epoch 11/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4687 - accuracy: 0.828
9 - val_loss: 0.4357 - val_accuracy: 0.8331
Epoch 12/75
922/922 [==============================] - 8s 8ms/step - loss: 0.4667 - accuracy: 0.8283
- val_loss: 0.4646 - val_accuracy: 0.8288
Epoch 13/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4634 - accuracy: 0.8273
- val_loss: 0.4224 - val_accuracy: 0.8411
Epoch 14/75
922/922 [==============================] - 7s 8ms/step - loss: 0.4565 - accuracy: 0.8287
- val_loss: 0.4393 - val_accuracy: 0.8381
Epoch 15/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4527 - accuracy: 0.8334
- val_loss: 0.4349 - val_accuracy: 0.8449
Epoch 16/75
922/922 [==============================] - 7s 8ms/step - loss: 0.4524 - accuracy: 0.8326
- val_loss: 0.4212 - val_accuracy: 0.8381
Epoch 17/75
922/922 [==============================] - 7s 8ms/step - loss: 0.4492 - accuracy: 0.8333
- val_loss: 0.4409 - val_accuracy: 0.8195
Epoch 18/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4482 - accuracy: 0.8343
- val_loss: 0.4318 - val_accuracy: 0.8407
Epoch 19/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4465 - accuracy: 0.8382
- val_loss: 0.4190 - val_accuracy: 0.8415
Epoch 20/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4417 - accuracy: 0.8354
- val_loss: 0.4205 - val_accuracy: 0.8394
Epoch 21/75
922/922 [==============================] - 7s 7ms/step - loss: 0.4404 - accuracy: 0.8353
- val_loss: 0.4421 - val_accuracy: 0.8356
Epoch 22/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4425 - accuracy: 0.8370
- val_loss: 0.4964 - val_accuracy: 0.8107
Epoch 23/75
```

```
922/922 [==============================] - 9s 10ms/step - loss: 0.4433 - accuracy: 0.836
3 - val_loss: 0.4172 - val_accuracy: 0.8356
Epoch 24/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4416 - accuracy: 0.836
4 - val_loss: 0.4351 - val_accuracy: 0.8360
Epoch 25/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4352 - accuracy: 0.8395
- val_loss: 0.4621 - val_accuracy: 0.8297
Epoch 26/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4406 - accuracy: 0.8338
- val_loss: 0.4587 - val_accuracy: 0.8178
Epoch 27/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4352 - accuracy: 0.838
5 - val_loss: 0.4398 - val_accuracy: 0.8419
Epoch 28/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4404 - accuracy: 0.8367
- val_loss: 0.4278 - val_accuracy: 0.8419
Epoch 29/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4358 - accuracy: 0.838
0 - val_loss: 0.4073 - val_accuracy: 0.8263
Epoch 30/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4296 - accuracy: 0.840
6 - val_loss: 0.4232 - val_accuracy: 0.8440
Epoch 31/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4325 - accuracy: 0.836
4 - val_loss: 0.4237 - val_accuracy: 0.8385
Epoch 32/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4307 - accuracy: 0.8381
- val_loss: 0.4372 - val_accuracy: 0.8352
Epoch 33/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4350 - accuracy: 0.838
5 - val_loss: 0.4249 - val_accuracy: 0.8284
Epoch 34/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4355 - accuracy: 0.8373
- val_loss: 0.4157 - val_accuracy: 0.8453
Epoch 35/75
922/922 [==============================] - 8s 8ms/step - loss: 0.4311 - accuracy: 0.8387
- val_loss: 0.4191 - val_accuracy: 0.8390
Epoch 36/75
922/922 [==============================] - 6s 7ms/step - loss: 0.4310 - accuracy: 0.8389
- val_loss: 0.4334 - val_accuracy: 0.8267
Epoch 37/75
922/922 [==============================] - 7s 8ms/step - loss: 0.4314 - accuracy: 0.8391
- val_loss: 0.4130 - val_accuracy: 0.8407
Epoch 38/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4271 - accuracy: 0.8383
- val_loss: 0.4256 - val_accuracy: 0.8402
Epoch 39/75
922/922 [==============================] - 7s 7ms/step - loss: 0.4268 - accuracy: 0.8374
- val_loss: 0.4182 - val_accuracy: 0.8335
Epoch 40/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4298 - accuracy: 0.8369
- val_loss: 0.4277 - val_accuracy: 0.8254
Epoch 41/75
922/922 [==============================] - 7s 8ms/step - loss: 0.4263 - accuracy: 0.8391
- val_loss: 0.4233 - val_accuracy: 0.8449
Epoch 42/75
922/922 [==============================] - 8s 8ms/step - loss: 0.4284 - accuracy: 0.8391
- val_loss: 0.4097 - val_accuracy: 0.8449
Epoch 43/75
922/922 [==============================] - 7s 7ms/step - loss: 0.4254 - accuracy: 0.8385
- val_loss: 0.4332 - val_accuracy: 0.8369
Epoch 44/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4283 - accuracy: 0.8397
- val_loss: 0.4242 - val_accuracy: 0.8398
```

```
Epoch 45/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4298 - accuracy: 0.838
7 - val_loss: 0.4332 - val_accuracy: 0.8352
Epoch 46/75
922/922 [==============================] - 7s 8ms/step - loss: 0.4277 - accuracy: 0.8393
- val_loss: 0.4289 - val_accuracy: 0.8347
Epoch 47/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4258 - accuracy: 0.8399
- val_loss: 0.4568 - val_accuracy: 0.8250
Epoch 48/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4257 - accuracy: 0.8370
- val_loss: 0.4244 - val_accuracy: 0.8343
Epoch 49/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4267 - accuracy: 0.8401
- val_loss: 0.4203 - val_accuracy: 0.8390
Epoch 50/75
922/922 [==============================] - 7s 7ms/step - loss: 0.4226 - accuracy: 0.8415
- val_loss: 0.4128 - val_accuracy: 0.8411
Epoch 51/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4222 - accuracy: 0.8425
- val_loss: 0.4163 - val_accuracy: 0.8428
Epoch 52/75
922/922 [==============================] - 7s 7ms/step - loss: 0.4230 - accuracy: 0.8400
- val_loss: 0.4184 - val_accuracy: 0.8423
Epoch 53/75
922/922 [==============================] - 7s 7ms/step - loss: 0.4284 - accuracy: 0.8399
- val_loss: 0.4169 - val_accuracy: 0.8352
Epoch 54/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4240 - accuracy: 0.8434
- val_loss: 0.4249 - val_accuracy: 0.8335
Epoch 55/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4224 - accuracy: 0.8385
- val_loss: 0.4161 - val_accuracy: 0.8385
Epoch 56/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4197 - accuracy: 0.839
1 - val_loss: 0.4115 - val_accuracy: 0.8385
Epoch 57/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4250 - accuracy: 0.840
4 - val_loss: 0.4102 - val_accuracy: 0.8347
Epoch 58/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4234 - accuracy: 0.8431
- val_loss: 0.4100 - val_accuracy: 0.8411
Epoch 59/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4175 - accuracy: 0.841
3 - val_loss: 0.4081 - val_accuracy: 0.8381
Epoch 60/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4199 - accuracy: 0.841
9 - val_loss: 0.4385 - val_accuracy: 0.8343
Epoch 61/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4247 - accuracy: 0.841
3 - val_loss: 0.4207 - val_accuracy: 0.8352
Epoch 62/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4217 - accuracy: 0.8417
- val_loss: 0.4123 - val_accuracy: 0.8432
Epoch 63/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4237 - accuracy: 0.840
7 - val_loss: 0.4227 - val_accuracy: 0.8402
Epoch 64/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4204 - accuracy: 0.839
7 - val_loss: 0.4172 - val_accuracy: 0.8398
Epoch 65/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4209 - accuracy: 0.8411
- val_loss: 0.4317 - val_accuracy: 0.8331
Epoch 66/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4194 - accuracy: 0.844
```

```
0 - val_loss: 0.4104 - val_accuracy: 0.8407
Epoch 67/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4219 - accuracy: 0.840
6 - val_loss: 0.4090 - val_accuracy: 0.8402
Epoch 68/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4248 - accuracy: 0.841
2 - val_loss: 0.4193 - val_accuracy: 0.8318
Epoch 69/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4202 - accuracy: 0.842
3 - val_loss: 0.4181 - val_accuracy: 0.8398
Epoch 70/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4220 - accuracy: 0.8421
- val_loss: 0.4114 - val_accuracy: 0.8373
Epoch 71/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4189 - accuracy: 0.842
9 - val_loss: 0.4164 - val_accuracy: 0.8415
Epoch 72/75
922/922 [==============================] - 9s 9ms/step - loss: 0.4204 - accuracy: 0.8406
- val_loss: 0.4201 - val_accuracy: 0.8440
Epoch 73/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4188 - accuracy: 0.840
4 - val_loss: 0.4178 - val_accuracy: 0.8322
Epoch 74/75
922/922 [==============================] - 9s 10ms/step - loss: 0.4167 - accuracy: 0.841
5 - val_loss: 0.4105 - val_accuracy: 0.8445
Epoch 75/75
922/922 [==============================] - 8s 9ms/step - loss: 0.4160 - accuracy: 0.8396
- val_loss: 0.4237 - val_accuracy: 0.8411
```

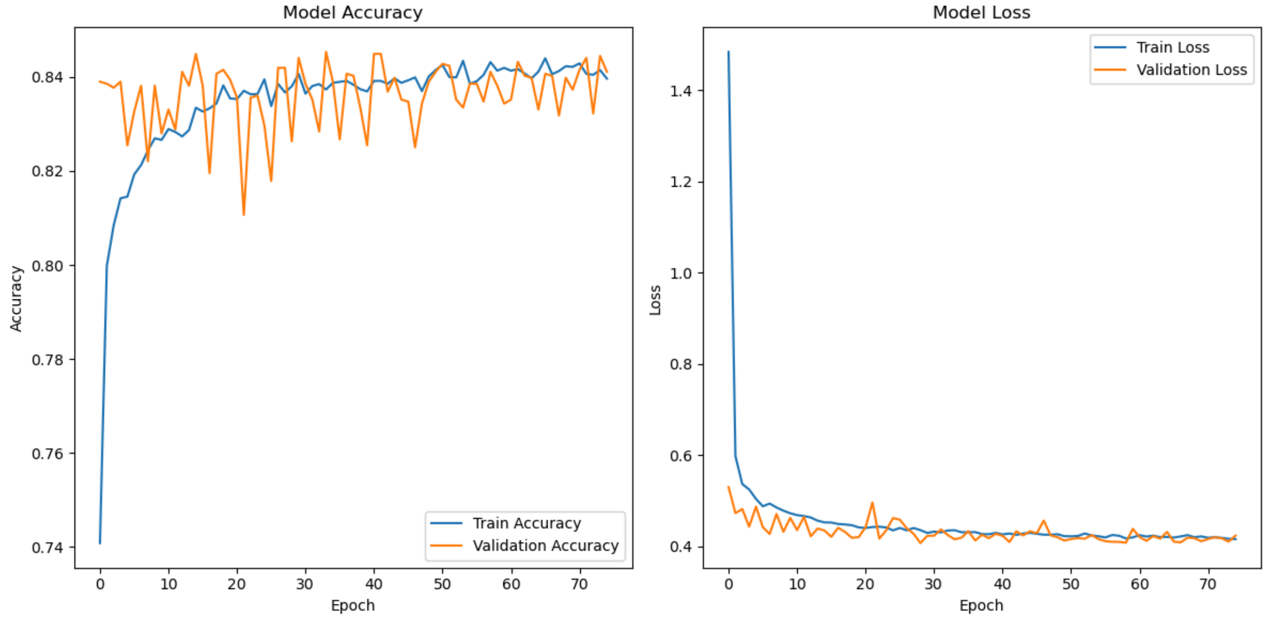
Let's plot to check for overfitting

In [38]:

```python
def plot_ml_acc_loss (history):
    # Plot training & validation accuracy values
    plt.figure(figsize=(12, 6))

    # Accuracy plot
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc='lower right')

    # Loss plot
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(loc='upper right')

    # Show the plots
    plt.tight_layout()
    plt.show()
```

In [39]:

```python
plot_ml_acc_loss(history)
```



First iteration - It looks like our model accuracy peaks at around 35 epochs and our model loss plateaus at 20 epochs. Let's work on improving that. (Original run through at 0.001 LR)

Changes to 0.0001 learning rate and the graph above looks much more fluid.

Let's use EarlyStopping and Reduce Learning Rate to try and help with overfitting

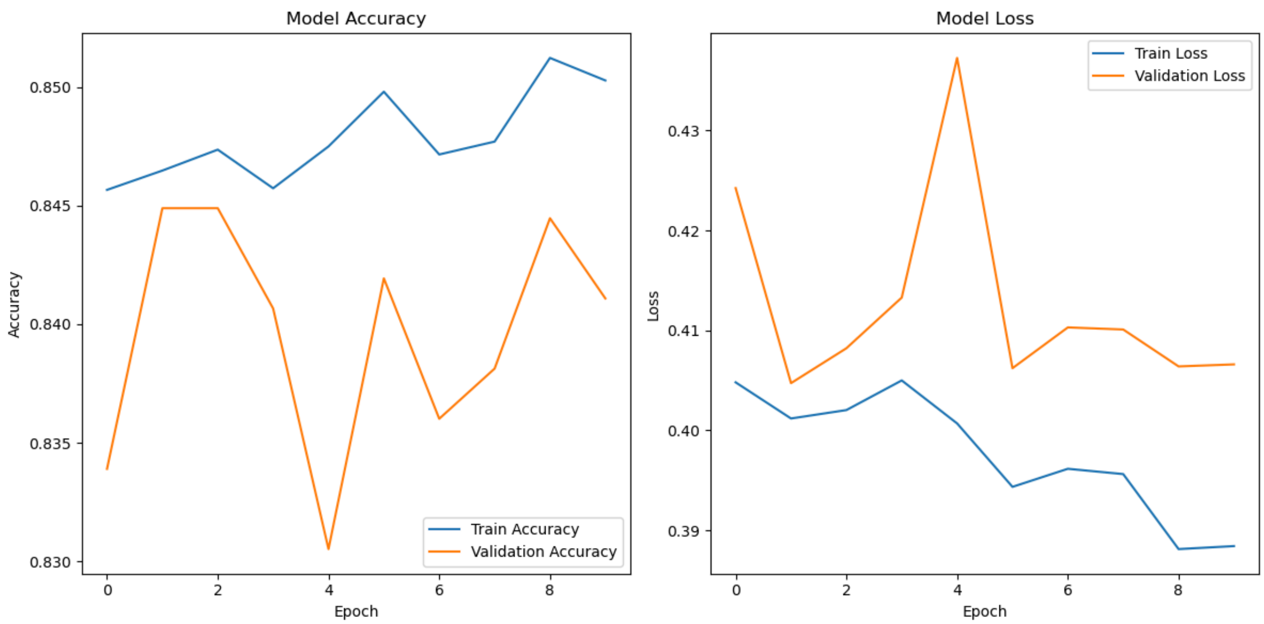
In [40]:
```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Add EarlyStopping to stop training when validation loss doesn't improve
early_stopping = EarlyStopping(monitor='val_loss', patience=8, restore_best_weights=Tru

# ReduceLROnPlateau: Reduce the learning rate if validation loss plateaus for 3 epochs
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=0.0000

# Re-train the model with early stopping
history = nn_model.fit(X_train_res, y_train_one_hot_res, epochs=100, batch_size=32,
                    validation_data=(X_test, y_test_one_hot),
                    callbacks=[early_stopping, reduce_lr])

# After EarlyStopping, the best model will be restored.
# Evaluate the model on the test set
test_loss, test_accuracy = nn_model.evaluate(X_test, y_test_one_hot)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
Epoch 1/100
461/461 [==============================] - 5s 10ms/step - loss: 0.4048 - accuracy: 0.845
7 - val_loss: 0.4242 - val_accuracy: 0.8339 - lr: 5.0000e-04
Epoch 2/100
461/461 [==============================] - 5s 10ms/step - loss: 0.4012 - accuracy: 0.846
5 - val_loss: 0.4047 - val_accuracy: 0.8449 - lr: 5.0000e-04
Epoch 3/100
461/461 [==============================] - 4s 10ms/step - loss: 0.4020 - accuracy: 0.847
4 - val_loss: 0.4082 - val_accuracy: 0.8449 - lr: 5.0000e-04
Epoch 4/100
461/461 [==============================] - 4s 9ms/step - loss: 0.4050 - accuracy: 0.8457
- val_loss: 0.4133 - val_accuracy: 0.8407 - lr: 5.0000e-04
Epoch 5/100
461/461 [==============================] - 4s 10ms/step - loss: 0.4007 - accuracy: 0.847
```

```
5 - val_loss: 0.4373 - val_accuracy: 0.8305 - lr: 5.0000e-04
Epoch 6/100
461/461 [==============================] - 4s 9ms/step - loss: 0.3943 - accuracy: 0.8498
- val_loss: 0.4062 - val_accuracy: 0.8419 - lr: 2.5000e-04
Epoch 7/100
461/461 [==============================] - 4s 9ms/step - loss: 0.3961 - accuracy: 0.8472
- val_loss: 0.4103 - val_accuracy: 0.8360 - lr: 2.5000e-04
Epoch 8/100
461/461 [==============================] - 4s 9ms/step - loss: 0.3956 - accuracy: 0.8477
- val_loss: 0.4101 - val_accuracy: 0.8381 - lr: 2.5000e-04
Epoch 9/100
461/461 [==============================] - 4s 9ms/step - loss: 0.3881 - accuracy: 0.8512
- val_loss: 0.4064 - val_accuracy: 0.8445 - lr: 1.2500e-04
Epoch 10/100
461/461 [==============================] - 4s 10ms/step - loss: 0.3884 - accuracy: 0.850
3 - val_loss: 0.4066 - val_accuracy: 0.8411 - lr: 1.2500e-04
74/74 [==============================] - 0s 4ms/step - loss: 0.4047 - accuracy: 0.8449
Test Loss: 0.4047204554080963
Test Accuracy: 0.8448858857154846
```

In [41]:

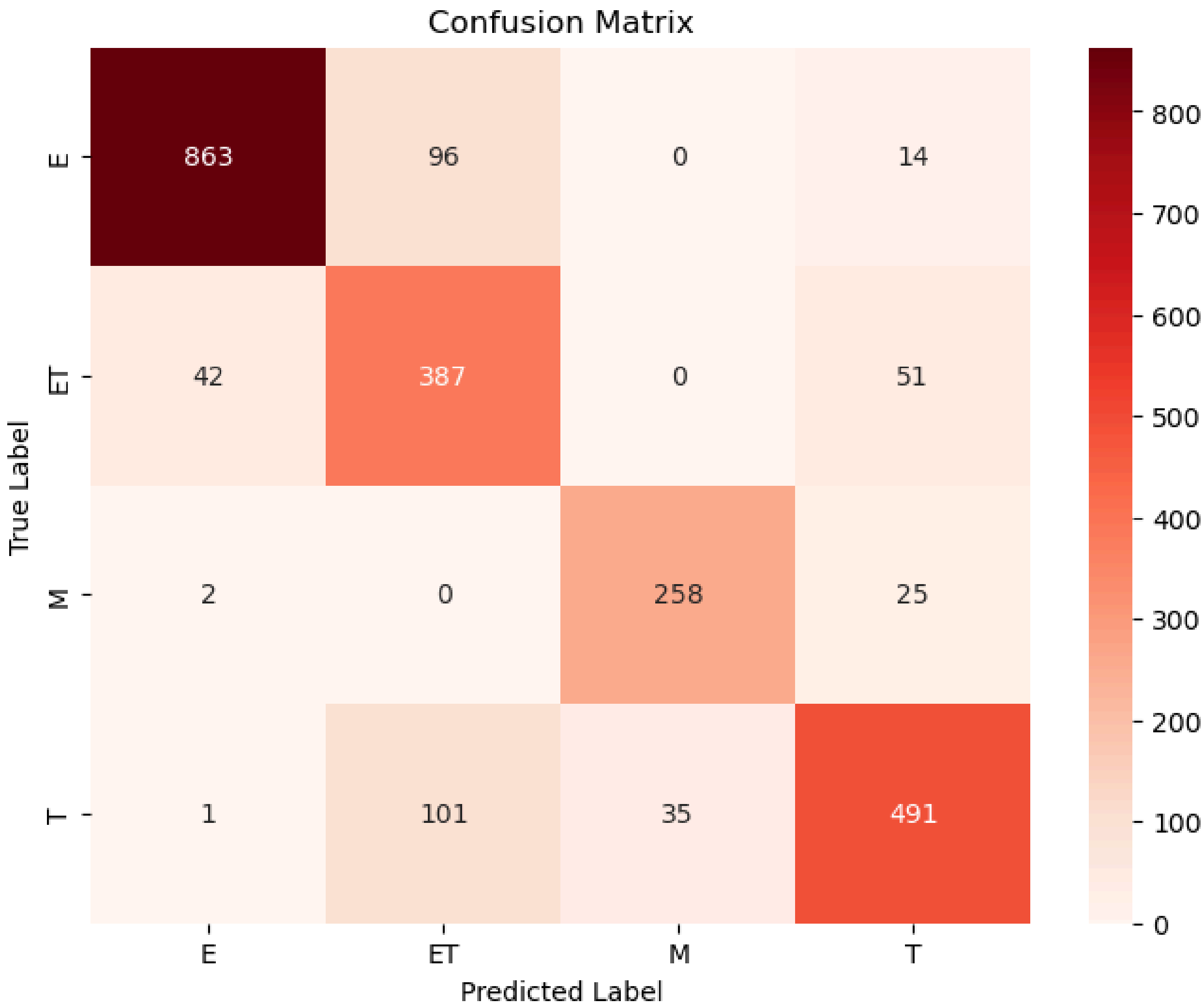
```python
plot_ml_acc_loss(history)
```



In [42]:

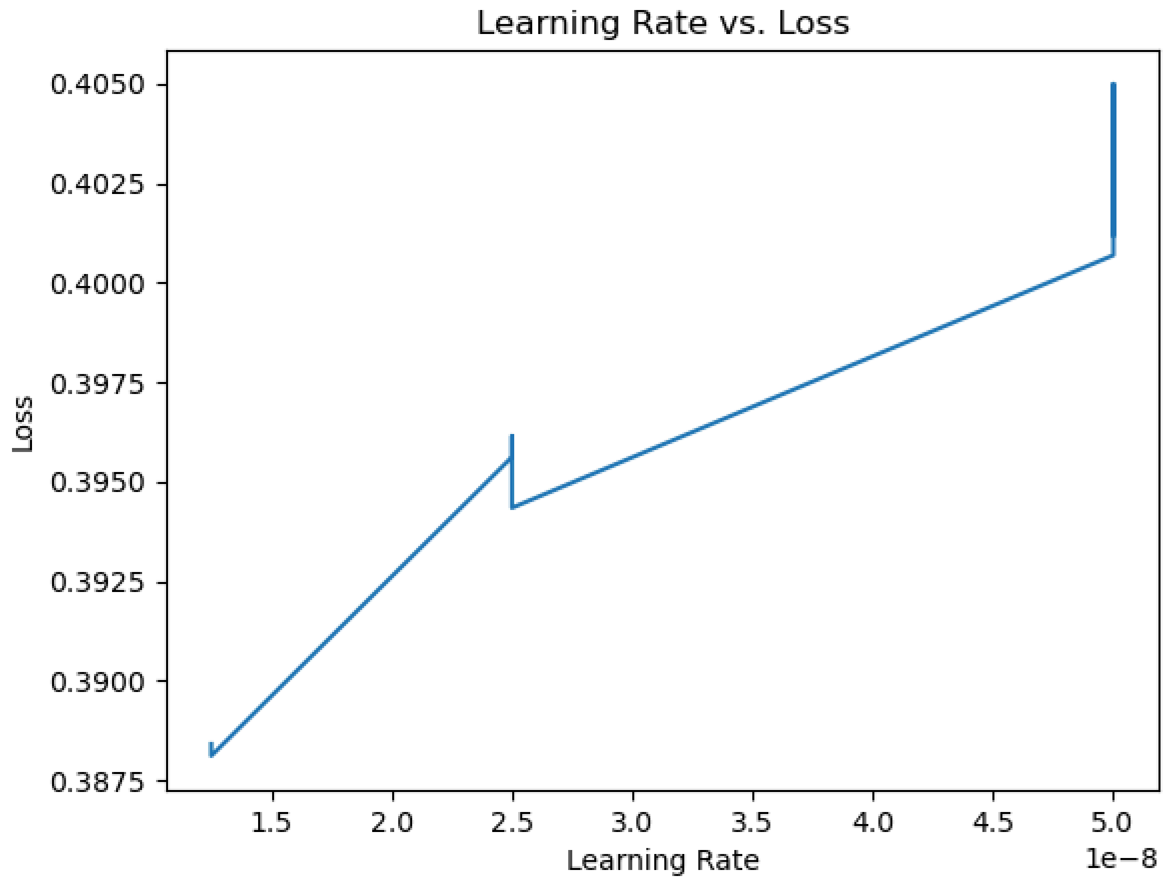
```python
# Predict the classes for the test set
y_pred = nn_model.predict(X_test_boost)
y_pred_classes = np.argmax(y_pred, axis=1)  # Convert to class labels

# Convert one-hot encoded labels back to integers for comparison
y_test_classes = np.argmax(y_test_one_hot, axis=1)

# Compute confusion matrix
conf_matrix = confusion_matrix(y_test_classes, y_pred_classes)

# Plot the confusion matrix
plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Reds", xticklabels=le.classes_, yti
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')

plt.savefig('graphs/cfm_dnn.png', bbox_inches = 'tight', edgecolor='w')
plt.show()
```

```
74/74 [==============================] - 0s 2ms/step
```



In [43]:

```
# Plot learning rate reduction if you used ReduceLROnPlateau
lrs = 0.0001 * np.array(history.history['lr'])
plt.plot(lrs, history.history['loss'])
plt.title('Learning Rate vs. Loss')
plt.xlabel('Learning Rate')
plt.ylabel('Loss')
plt.show()
```



Let's use a smaller dataset for testing

Run on new test data.

In [44]:
```
test_data = pd.read_csv('data/esrb_ratings_test_set.csv')
test_target = 'esrb_rating'
```

We encode the target column in this new set to match our model

In [45]:
```
# Encode the ESRB ratings as numeric values
test_le = LabelEncoder()
test_data[test_target] = test_le.fit_transform(test_data[test_target])  # Convert 'E',
```

In [46]:
```
#
test_data = test_data.drop(['title', 'console', 'esrb_rating'], axis=1)
```

In [47]:
```
# Get predictions from the model
predictions = nn_model.predict(test_data)

# If it's a classification problem and you want the class with the highest probability
predicted_classes = predictions.argmax(axis=1)
```

```
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
```

In [48]:
```
predicted_classes
```

Out[48]:
```
array([1, 0, 3, 2, 1], dtype=int64)
```

In [49]:
```
predicted_labels = test_le.inverse_transform(predicted_classes)
print(predicted_labels)
```

```
['ET' 'E' 'T' 'M' 'ET']
```

We were accurate 4 out of 5 times for a single run of this data. The first ET should be T. Given that the models above were definitely showing some interesting results for T (false positives), it sort of makes sense.

Recommendations

Based off the findings in the original dataset and the enlarged dataset, it seems like the definitions for E10+ and Teen are too ambiguous and close. Some games that are E10+ have too many descriptors that draw them closer to Teen. I would suggest just removing the E10+ since it's only 3 years difference from Teen.