Dependable AI

Implementation of LIME, SHAP and CNN Visualizations

Group Members:
Khushi Maheshwari (B21AI052)
Rutuja Janbandhu (B21AI017)

Part A: Implementation of LIME and SHAP:

Dataset Description: Breast Cancer Dataset: The dataset we've chosen for Part A is breast cancer from sklearn. It contains 30 features that are used to predict a tumor in 2 classes: malignant or benign.

Preprocessing: We loaded the dataset, standardized it using StandardScaler and then divided the dataset in training and testing subsets.

Training a Model That is Not Interpretable: We've used a Multi-Layer Perceptron (**MLP**) Classifier, which is a type of neural network, making it a non-interpretable classifier because it's challenging to understand how it makes predictions. We trained this model using our training subset.
Now since our model is not interpretable, we don't really know how the model makes a decision and how different features affect the decisions. We will implement LIME and SHAP and use them to understand our model better

LIME (Local Interpretable Model-agnostic Explanations):

LIME is a model-agnostic method for getting the explanations for the prediction for specific data points (instance). The explanations provided by LIME help users understand which features influence the model's predictions the most. It works by approximating the behavior of the complex model locally around the instance we want to explain. It builds a simpler, interpretable model around the instance we want to explain.
For given a non-interpretable model and an instance to explain (x), this is how lime works:

$$\text{explanation}(x) = \arg\min_{g \in G} L(f, g, \pi_x) + \Omega(g)$$

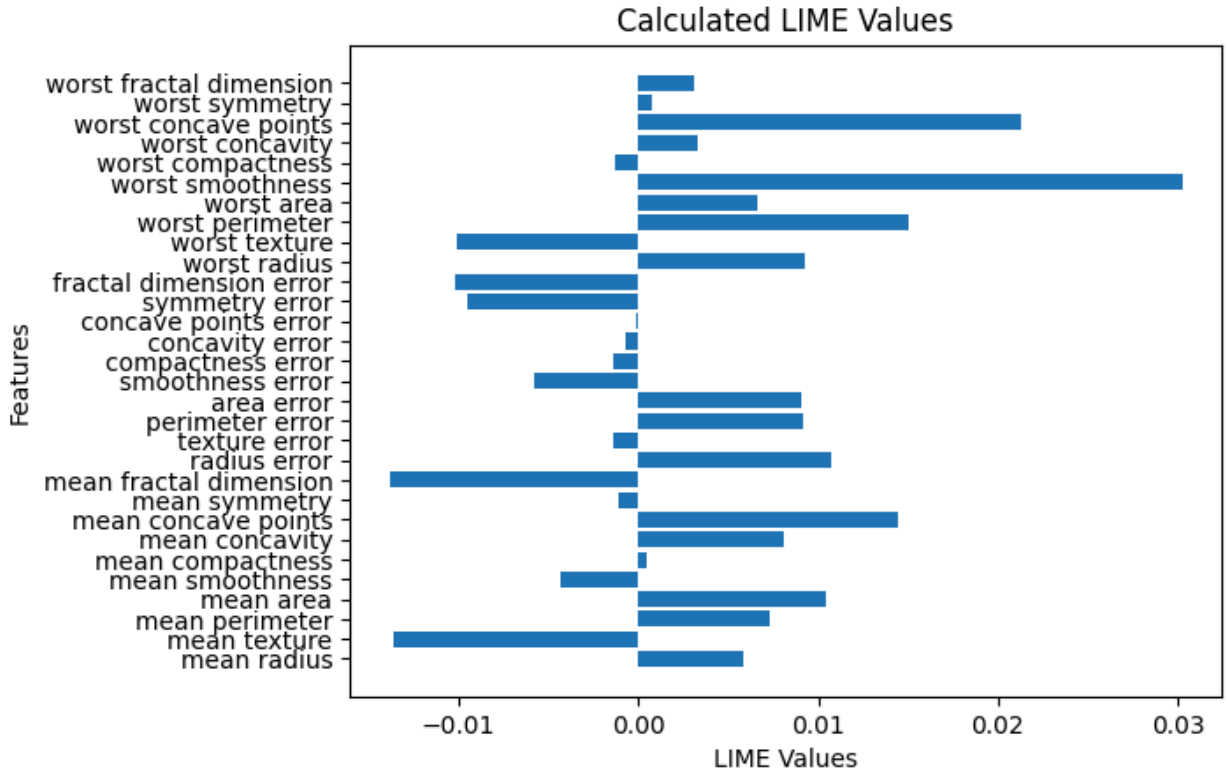
The explanation model for instance x is the model g (linear regression model) minimizes loss L (MSE), which measures how close the explanation is to the prediction of the original model f , while the model complexity Ω(g) is kept low. The proximity measure Π_x defines the size of the neighborhood around instance x that we consider for the explanation.

Our implementation:
In our implementation of LIME, we created a class called LIMEValueEstimator. This class provides explanations for predictions made by a given model. Here's a breakdown of the key components and how it calculates LIME values:

The explain_instance method explains the prediction for a specific instance. It selects a random instance from the dataset if no index is provided. It generates **random samples around the instance** to explain using the sample_around method which creates random samples from a normal distribution centered around each feature of the instance.
Then, it computes weights for each sample based on the distance from the instance using the compute_weights method. Next, it fits a **local linear regression model** using the samples and weights to approximate the behavior of the complex model locally. It uses the predicted probabilities from the global model (MLP) as the target variable for fitting the local model.
Finally, it returns the coefficients of the local model multiplied by the instance's features.

Explanation for Instance at Index 0:
We used the explain_instance method to compute LIME coefficients for the instance at index 0. LIME coefficients that we get represent the importance of each feature in the MLP model's prediction for the given instance. Further, we created a horizontal bar plot to visualize the calculated LIME values to gain a clear understanding of the relative importance of different features in the prediction process.

The visualization result is:



Further, we also used the standard LIME library to explain the prediction at the instance and observed the values as well as the visualizations.

SHAP (Shapely Additive Explanations):

This method calculates the Shapley values for all features based on the results given by game theory. Shapley value is the average marginal contribution of a feature across all possible combinations(coalitions) of features.

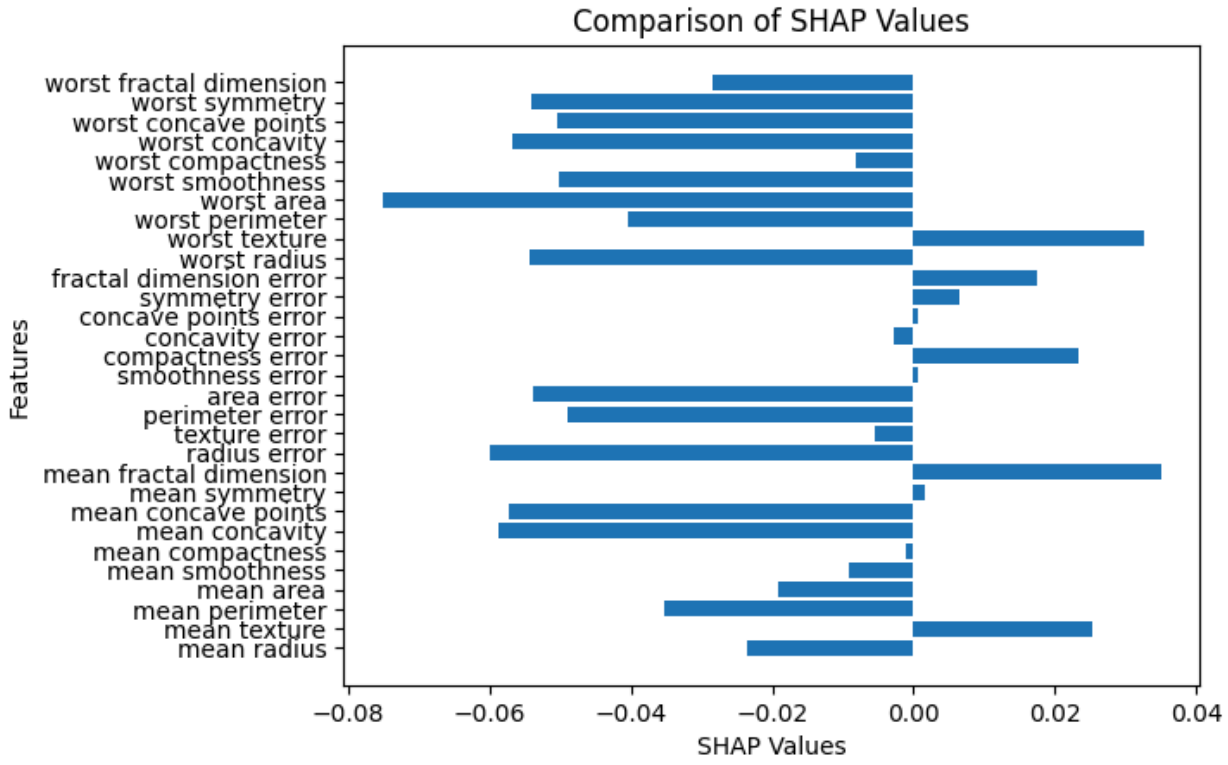
$$\phi_i = \sum_{S \subseteq M \setminus i} \frac{|S|!(|M| - |S| - 1)!}{|M|!} [f(S \cup i) - f(S)]$$

Here, Φi is the Shapely value for feature i, M is the complete set of features, and S is the subset of features excluding i. The difference between the prediction of the model including the feature i and excluding it is calculated.

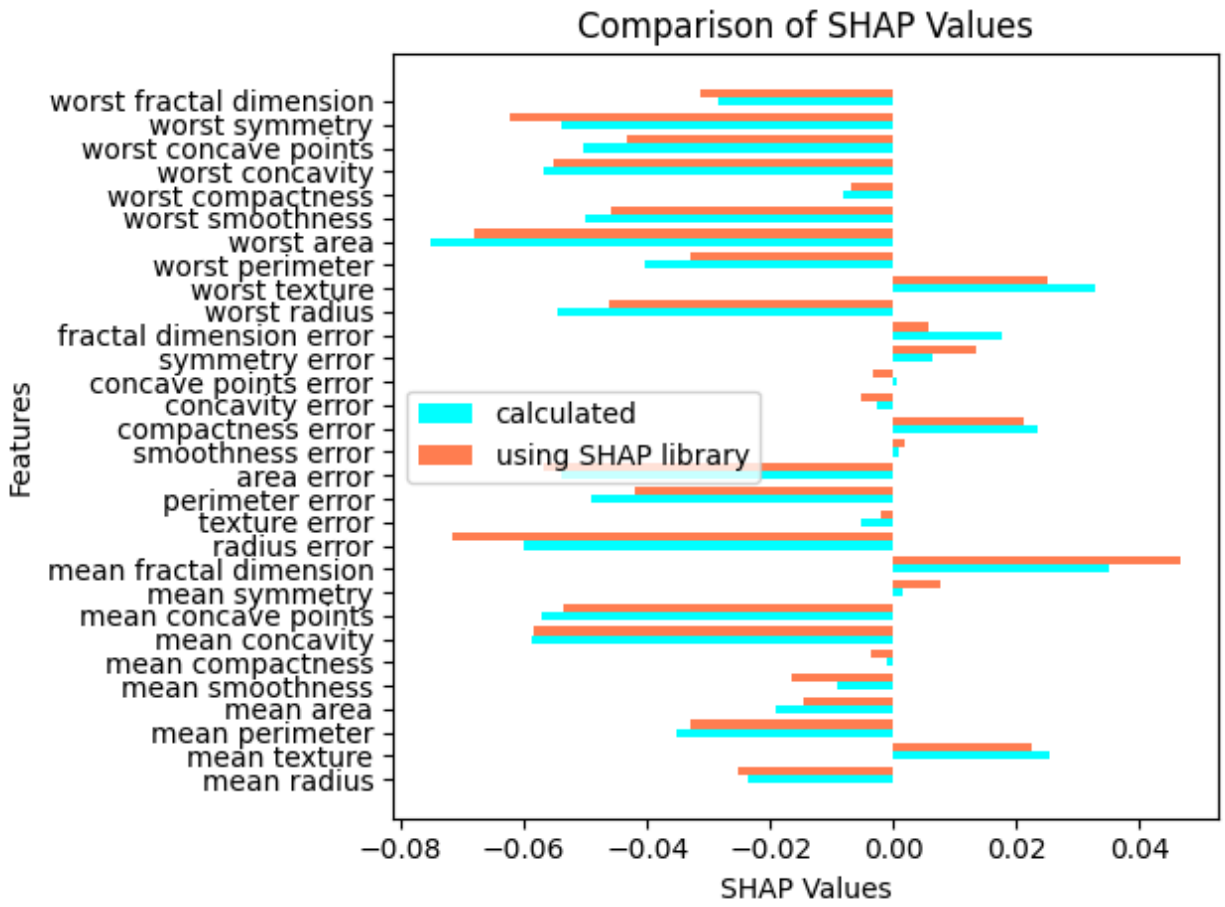
Our implementation:
In our implementation of SHAP, we created a class called ShapleyValueEstimator which takes as input the dataset along with the target function. The class initializes with a dataset and a target function.

The compute_shapley_values() method gives us the shapely values corresponding to all the features by using the generate_MC_estimates() method. The compute_shapley_estimates_for_feature() method calculates Shapley estimates for a specific feature using Monte Carlo sampling. The _generate_random_instance() method randomly selects an instance from the dataset. The get_updated_instances() method generates instances with and without the feature under consideration, given a reference point and a set of features.

Explanation for Instance:
We create an instance of the class and compute the shap values for an example which provide us the importance of each feature in the MLP model's prediction and then we create horizontal bar plots to visualize the comparison for feature importances.
This is the visualization that we get:



Then, we also use the standard SHAP library to compute the feature importance values for the same instance and then compare the values that we get with our calculated values above.
We plot the visualization for comparison:



When we calculate the difference between the actual and calculated values, we get to know that our calculated values are very much similar to the actual values and the error is almost zero.

Part B: CNN Visualizations:

In this, we use Grad-CAM (Gradient-weighted Class Activation Mapping) to visualize which parts of an image are most important for a deep learning model's prediction. We employ a pre-trained VGG16 model on the ImageNet dataset to classify images, and then generate heatmaps using Grad-CAM to highlight the regions of the image that the model focused on for making its prediction.

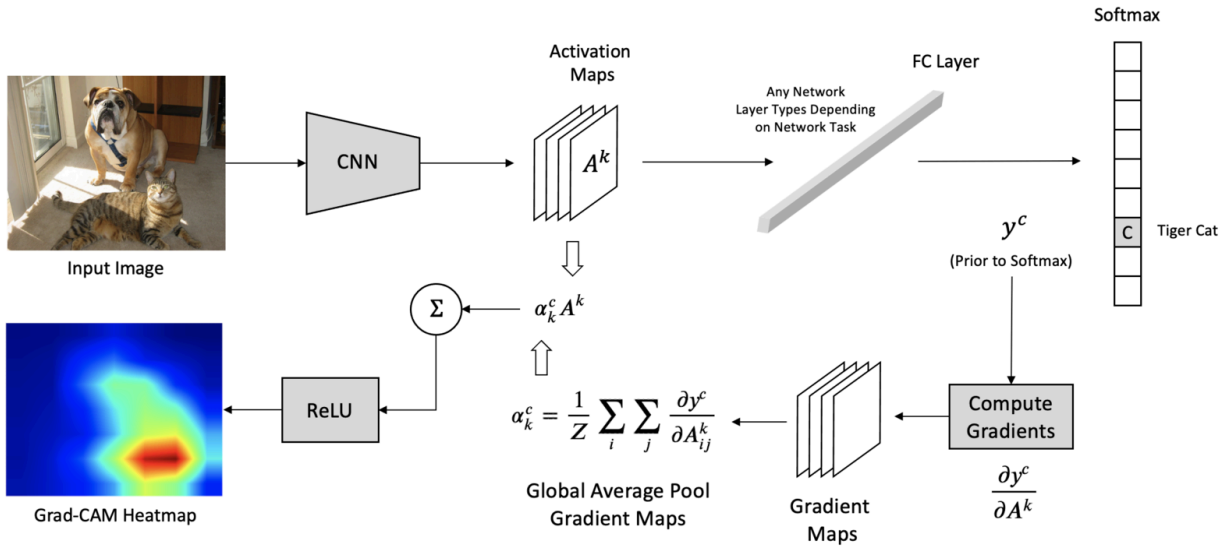
Loading Pre-trained VGG16 Model:

We start by loading a pre-trained VGG16 model using TensorFlow's tf.keras.applications.VGG16 module. This model has been trained on the ImageNet dataset, a large-scale dataset containing over 14 million images categorized into 1,000 classes.

Model Details:
The VGG16 model is a convolutional neural network architecture.The model consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. VGG16 accepts input images of size 224x224 pixels and produces predictions for 1,000 different object categories. It utilizes small 3x3 convolutional filters with a stride of 1 pixel throughout the network, and max-pooling layers with a 2x2 window and stride of 2 pixels to reduce spatial dimensions

Grad-CAM Implementation:
Grad-CAM (Gradient-weighted Class Activation Mapping) is a visualization technique used to understand and interpret deep neural networks, particularly convolutional neural networks (CNNs), by highlighting the regions of an input image that contribute the most to the model's prediction.

We implement the Grad-CAM technique to visualize which parts of the image contribute the most to the model's prediction. Grad-CAM computes the gradient of the predicted class with respect to the feature maps of a specified convolutional layer in the model. We define functions to load an image, compute Grad-CAM heatmap, and overlay the heatmap on the original image.

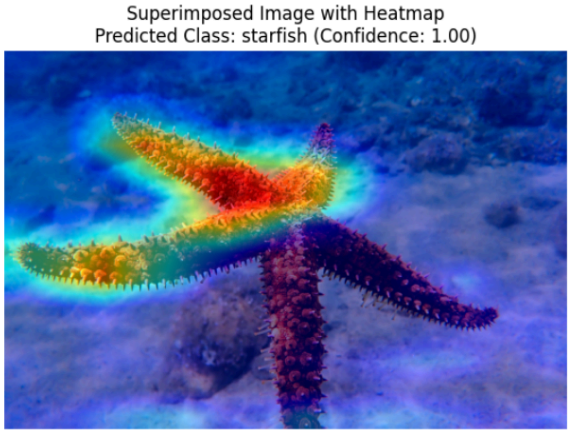


The get_grad_cam function implements Grad-CAM, a technique to visualize important regions in an input image for a specific class prediction. It loads and preprocesses the image, constructs a model that outputs both target convolutional layer activations and final predictions, computes the loss and gradients, and then combines them to generate a normalized heatmap. The heatmap highlights regions influencing the model's prediction, providing interpretability into the model's decision-making process.

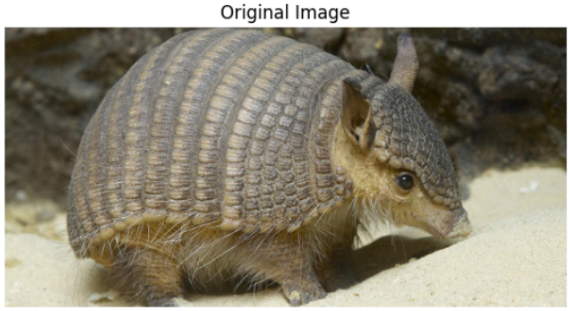
Visualizing Grad-CAM:
We visualize Grad-CAM for a sample image by first loading the image, computing Grad-CAM heatmap, and overlaying the heatmap on the original image. We also print the predicted class and confidence score above the superimposed image.

Plotting Images Side by Side:
Finally, we plot the original image and the superimposed image with heatmap side by side for comparison. We use Matplotlib to create a subplot with two columns to display the images.We predicted the model's prediction on 11 different images, to visualize its prediction and confidence score.

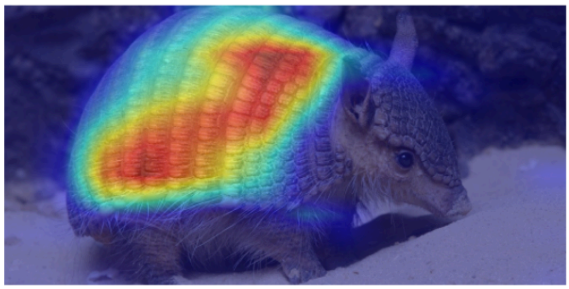
In order to visualize the original image and superimposed image with heatmap, we created a function visualize_original_and_superimposed_image, which takes the input as model and image path. The model is the pretrained model (VGG16 model) and we tested on 11 different images.It then calls another function visualize_grad_cam to generate the Grad-CAM visualization for the input image.



Original Image



Superimposed Image with Heatmap
Predicted Class: armadillo (Confidence: 0.92)

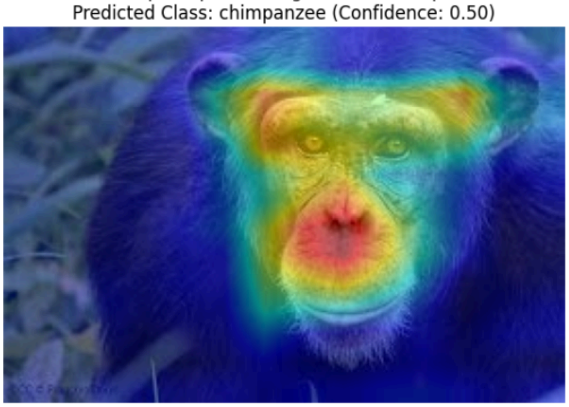


Original Image

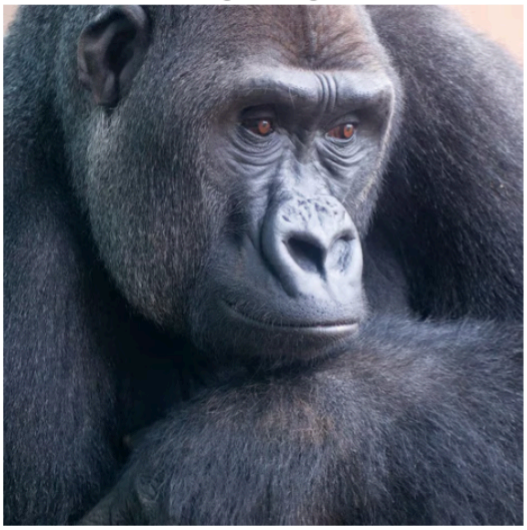
Superimposed Image with Heatmap
Predicted Class: cheetah (Confidence: 0.43)

Original Image

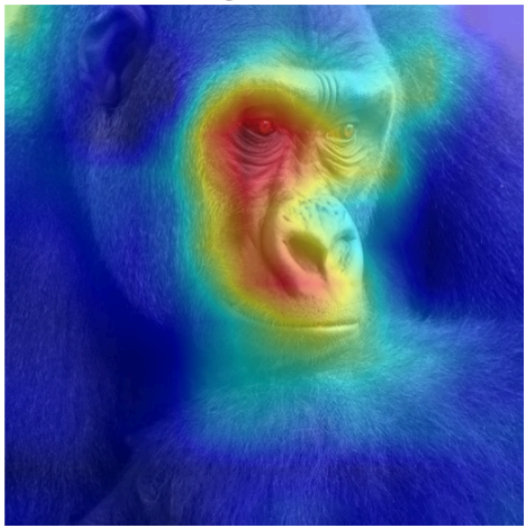
Superimposed Image with Heatmap
Predicted Class: chimpanzee (Confidence: 0.50)



Original Image



Superimposed Image with Heatmap
Predicted Class: gorilla (Confidence: 1.00)



Original Image

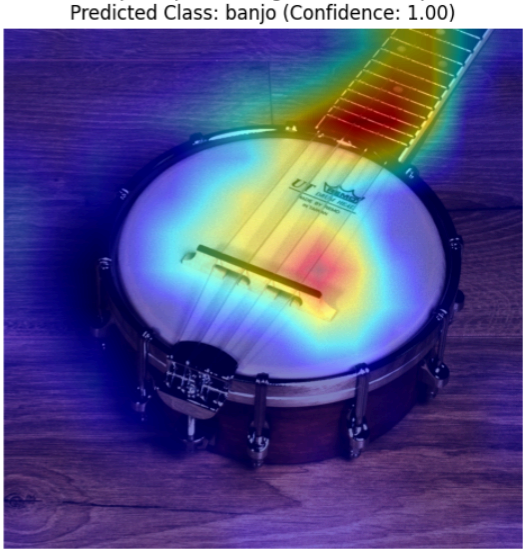
Superimposed Image with Heatmap
Predicted Class: Lhasa (Confidence: 0.26)

Original Image

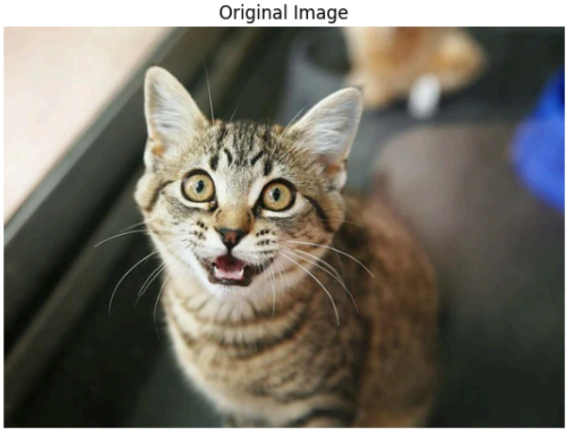
Superimposed Image with Heatmap
Predicted Class: zebra (Confidence: 1.00)

Original Image

Superimposed Image with Heatmap
Predicted Class: banjo (Confidence: 1.00)



Original Image



Superimposed Image with Heatmap
Predicted Class: Egyptian_cat (Confidence: 0.65)

Original Image

Superimposed Image with Heatmap
Predicted Class: Afghan_hound (Confidence: 0.26)

The visualize_grad_cam_multi_class function takes a pre-trained model, an image path, and optional parameters such as target size and layer name as input. It first loads and

preprocesses the input image using the specified target size. Then, it predicts the classes of the image using the provided model and extracts the top 3 predicted class indices and their corresponding labels.
Next, for each of the top predicted classes, it computes the Grad-CAM heatmap using the get_grad_cam function, which highlights the regions of the image important for the prediction of each class. These heatmaps are then overlaid on the original image using the overlay_heatmap function.

In the context of image classification tasks, it's important to acknowledge that certain objects or categories, such as dogs with different breeds, can exhibit similarities in appearance. For instance, breeds like Labrador Retrievers and Golden Retrievers might share common visual characteristics, making it challenging for the model to distinguish between them accurately. As a result, the prediction confidence scores generated by the model might be affected, leading to lower confidence scores or incorrect classifications in some cases. This phenomenon is particularly relevant when dealing with fine-grained classification tasks where subtle differences between classes exist. It underscores the inherent complexity of image recognition tasks and the need for robust models capable of capturing intricate details to make accurate predictions across diverse categories.
