Vitamin Vision: Unveiling the Spectrum of Nutrient Detection

Project Description:

The "Vitamin vision" project focuses on utilizing advanced analytical techniques and technologies to accurately identify and quantify the presence of various vitamins in samples such as food, supplements, and biological fluids. By leveraging methods such as chromatography, spectroscopy, and machine learning, the project aims to ensure the quality and safety of products, verify nutritional content, and support research in health and wellness.

Scenario 1: Quality Control in Food and Supplements

The project enables manufacturers and quality control labs to detect and quantify vitamins in food products and dietary supplements. By ensuring products meet label claims and safety standards, the project supports consumer trust and regulatory compliance

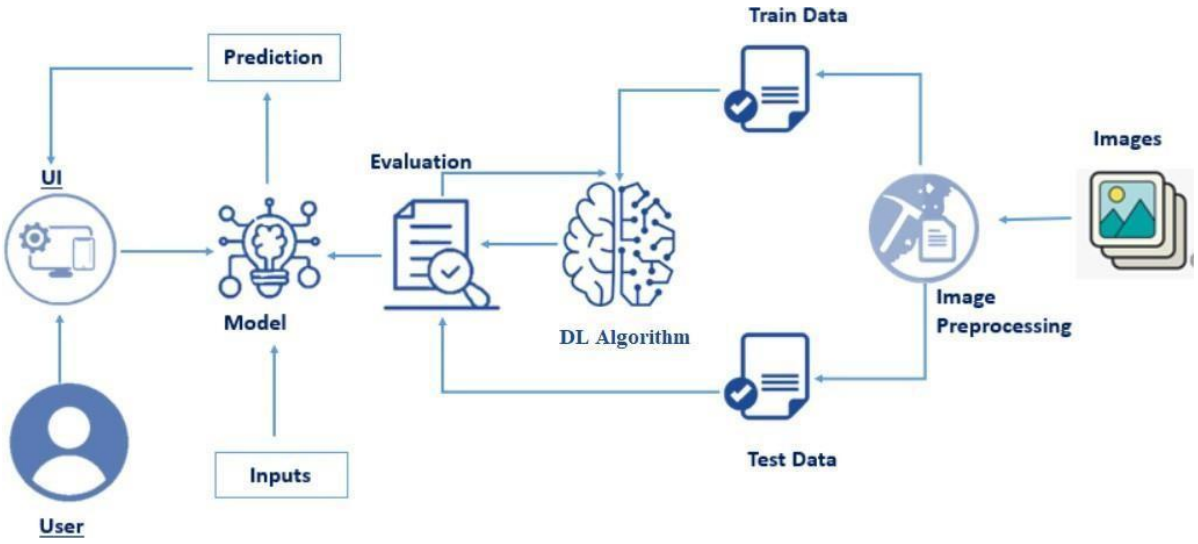
Scenario 2: Nutritional Research and Health Studies

Researchers can use the vitamin detection methods to measure vitamin levels in biological samples such as blood or urine. This data can help identify nutritional deficiencies, inform health studies, and support the development of personalized nutrition plans.

Scenario 3: Regulatory and Compliance Verification

Regulatory bodies can utilize vitamin detection technologies to monitor the accuracy of nutritional labeling and safety of food and supplement products. By conducting routine checks on product samples, they can ensure that manufacturers adhere to regulations and protect public health.

Technical Architecture:



Prerequisites

To complete this project, you must require the following software's, concepts, and packages

Anaconda Navigator is a free and open-source distribution of the Python and R programming languages for data science and machine learning related applications. It can be installed on Windows, Linux, and macOS. Conda is an open-source, cross-platform, package management system. Anaconda comes with so very nice tools like JupyterLab, Jupyter Notebook,

QtConsole, Spyder, Glueviz, Orange, Rstudio, Visual Studio Code. For this project, we will be using Jupyter notebook and VS code.

To install Anaconda navigator and to know how to use Jupyter Notebook & Spyder using Anaconda watch the video

1. To build Machine learning models you must require the following packages

- Numpy:
  - It is an open-source numerical Python library. It contains a multidimensional array and matrix data structures and can be used to perform mathematical operations.

- Scikit-learn:

  - It is a free machine learning library for Python. It features various algorithms like support vector machine, random forests, and k-neighbors, and it also supports Python numerical and scientific libraries like NumPy and SciPy

- Flask:
  Web framework used for building Web applications.

- Python packages:
  - open anaconda prompt as administrator
  - Type “pip install numpy” and click enter.
  - Type “pip install pandas” and click enter.
  - Type “pip install scikit-learn” and click enter.
  - Type “pip install tensorflow==2.12.0” and click enter.
  - Type “pip install keras==2.12.0” and click enter.
  - Type “pip install Flask” and click enter.

- Deep Learning Concepts
  - **CNN:** a convolutional neural network is a class of deep neural networks, most commonly applied to analyzing visual imagery.
    CNN Basic
  - **VGG19:** VGG16 is a deep convolutional neural network architecture for image classification, consisting of 19 layers with small convolution filters.
    VGG19
  - **Flask:** Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.

Flask Basics

If you are using Pycharm IDE, you can install the packages through the command prompt and follow the same syntax as above.

Project Objectives

By the end of this project you will:

- Know fundamental concepts and techniques of Convolutional Neural Network.
- Gain a broad understanding of image data.
- Know how to pre-process/clean the data using different data preprocessing techniques.
- know how to build a web application using the Flask framework.

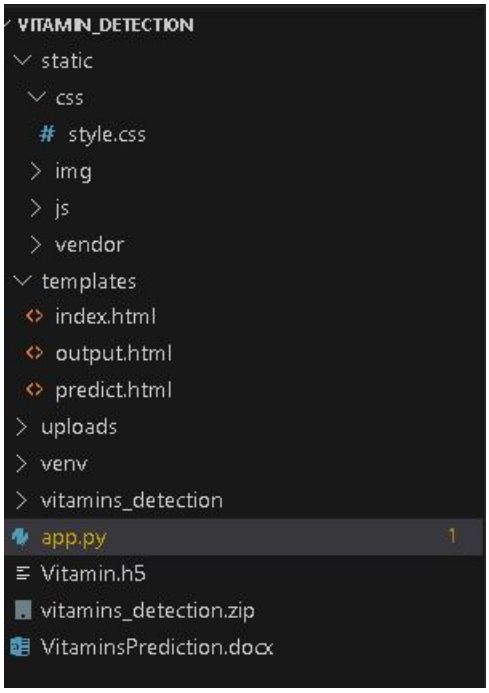
Project Flow

- The user interacts with the UI (User Interface) to choose the image.
- The chosen image is analyzed by the model which is integrated with flask application.
- CNN Models analyze the image, then prediction is showcased on the Flask UI. To accomplish this, we must complete all the activities and tasks listed below.

- Data Collection.
  - Create Train and Test Folders.
- Data Preprocessing.
  - Import the ImageDataGenerator library
  - Configure ImageDataGenerator class
  - ApplyImageDataGenerator functionality to Train dataset and Test dataset
- Model Building
  - Import the model building Libraries.
  - Importing the VGG19.
  - Initializing the model
  - Adding Fully connected Layer
  - Configure the Learning Process
  - Training and Testing the model.
  - Save the Model
- Application Building
  - Create an HTML file
  - Build Python Code

Project Structure

Create a Project folder which contains files as shown below.



- The Data folder contains the training and testing images for training our model.
- We are building a Flask Application that needs HTML pages stored in the templates.
  folder and a python script app.py for server-side scripting
- we need the model which is saved and the saved model in this content is a Vitamin.h5
- templates folder contains index.html, predict.html & output.html pages.

Data Collection & Image Preprocessing

In this milestone First, we will collect images of Food then organized into subdirectories based on their respective names as shown in the project structure. Create folders of types of Vitamin Food that need to be recognized. In this project, we have collected images of 5 types of Images like Vitamin A, Vitamin B, Vitamin C, Vitamin D, Vitamin E. they are saved in the respective sub directories with their respective names.

Download the Dataset - https://www.kaggle.com/code/allulucky27/vitamin-detection deep-learning

In Image Processing, we will be improving the image data that suppresses unwilling distortions or enhances some image features important for further processing, although perform some geometric transformations of images like rotation, scaling, translation, etc.

1: Import the ImageDataGenerator library.

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.

The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the ImageDataGenerator class.
Let us import the ImageDataGenerator class from tensorflow Keras.

```
#import image datagenerator library
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

2: Configure ImageDataGenerator class

ImageDataGenerator class is instantiated and the configuration for the types of data augmentation. There are five main types of data augmentation techniques for image data; specifically:

- Image shifts via the width_shift_range and height_shift_range arguments.
- The image flips via the horizontal_flip and vertical_flip arguments.
- Image rotations via the rotation_range argument
- Image brightness via the brightness_range argument.
- Image zoom via the zoom_range argument.

An instance of the ImageDataGenerator class can be constructed for train and test.

```
train_datagen = ImageDataGenerator(rescale=1./255, shear_range=0.2,zoom_range=0.2,horizontal_flip=True)
```

3: Apply ImageDataGenerator functionality to Trainset and Testset

Let us apply ImageDataGenerator functionality to Train set and Test set by using the following code. For Training set using flow_from_directory function.

This function will return batches of images from the subdirectories Vitamin A, Vitamin B, Vitamin C, Vitamin D, Vitamin E together with labels 0 to 4{Vitamin A: 0, Vitamin B: 1, Vitamin C: 2, Vitamin D: 3, Vitamin E: 4 }

Arguments:

- directory: Directory where the data is located. If labels are "inferred", it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
- batch_size: Size of the batches of data which is 15.
- target_size: Size to resize images after they are read from disk.
- class_mode:

- ‘int': means that the labels are encoded as integers (e.g. for sparse_categorical_crossentropy loss).
- 'categorical' means that the labels are encoded as a categorical vector (e.g. for categorical_crossentropy loss).
- 'binary' means that the labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g. for binary_crossentropy).
- None (no labels).

```
[ ] train_data = train_datagen.flow_from_directory(
    '/content/drive/MyDrive/Vitamin/Train',
    target_size=(224,224),
    batch_size=15,
    class_mode='categorical')
```

```
Found 7173 images belonging to 5 classes.
```

```
[ ] test_data = train_datagen.flow_from_directory(
    '/content/drive/MyDrive/Vitamin/test',
    target_size=(224, 224),
    batch_size=15,
    class_mode='categorical')
```

```
Found 1795 images belonging to 5 classes.
```

We notice that 7173 images belong to 5 classes for training and 1795 images belong to 5 classes for testing purposes.

Model Building

Now it's time to build our Convolutional Neural Networking using vgg19 which contains an input layer alongwith the convolution, max-pooling, and finally an output layer.

1: Importing the Model Building Libraries

Importing the necessary libraries

Importing Libraries

```
[23] import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras.preprocessing.image  import ImageDataGenerator
     from tensorflow.keras.layers import Dense
     from tensorflow.keras.activations import softmax
     from keras.api._v2.keras import activations
```

2: Importing the VGG19 model

To initialize the VGG19 model, the weights are usually pre-trained on the ImageNet dataset, which is a large-scale dataset of images belonging to 1,000 different categories. These pre-trained weights can be downloaded from the internet, and they can be used as a starting point to fine-tune the model for a specific task, such as object recognition or classification.

```
[ ]  from tensorflow.keras.applications.vgg19 import VGG19
     from tensorflow.keras.layers import Dense, Flatten
     from tensorflow.keras.models import Model
     from tensorflow.keras.optimizers import Adam
```

3: Initializing the model:

- The model will be initialized with the pre-trained weights from the ImageNet dataset, and the last fully connected layer will be excluded from the model architecture.
- The loop that follows freezes the weights of all the layers in the VGG19 model by setting `i.trainable=False` for each layer in the model. This is done to prevent the weights from being updated during training, as the model is already pre-trained on a large dataset.
- Finally, a `Flatten()` layer is added to the output of the VGG19 model to convert the output tensor into a 1D tensor.
- The resulting model can be used as a feature extractor for transfer learning or as a starting point for building a new model on top of it.

Importing VGG19 Model

```
[ ] from tensorflow.keras.applications.vgg19 import VGG19
    from tensorflow.keras.layers import Dense, Flatten
    from tensorflow.keras.models import Model
    from tensorflow.keras.optimizers import Adam
```

```
[ ] Image_size=[224,224]
```

```
[ ] sol=VGG19(input_shape=Image_size + [3] , weights='imagenet' , include_top = False)
```

```
[ ] for i in sol.layers:
      i.trainable = False
```

```
[ ] y=Flatten()(sol.output)
```

4: Adding Fully connected Layers

- For information regarding CNN Layers refer to the link Link: https://victorzhou.com/blog/intro-to-cnns-part-1/
- As the input image contains three channels, we are specifying the input shape as (128,128,3).
- We are adding a output layer with activation function as “softmax”.

```
[ ] from tensorflow.keras.layers import Dense
    from tensorflow.keras.activations import softmax
```

```
[ ] from keras.api._v2.keras import activations
    final = Dense(5, activation = 'softmax')(y)
```

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer. The number of neurons in the Dense layer is the same as the number of classes in the training set. The neurons in the last Dense layer, use softmax activation to convert their outputs into respective probabilities.

Understanding the model is a very important phase to properly use it for training and prediction purposes. Keras provides a simple method, summary to get the full information about the model and its layers.

```
[ ] from tensorflow.keras.models import Model
```

```
[ ] vgg19_model = Model(sol.inputs, final)
```

```
vgg19_model.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
block3_conv4 (Conv2D)        (None, 56, 56, 256)       590080
```

5: Configure The Learning Process

- The compilation is the final step in creating a model. Once the compilation is done, we can move on to the training phase. The loss function is used to find errors or deviations inthe learning process. Keras requires a loss function during the model compilation process.
- Optimization is an important process that optimizes the input weights by comparing the prediction and the loss function. Here we are using adam optimizer.
- Metrics are used to evaluate the performance of your model. It is similar to the loss function, but not used in the training process.

```
[ ] vgg19_model.compile(optimizer = 'adam' , loss = 'categorical_crossentropy' , metrics = ['Accuracy'])
```

6: Train The model

Now, let us train our model with our image dataset. The model is trained for 9 epochs and after every epoch, the current model state is saved if the model has the least loss encountered till that time. We can see that the training loss decreases in almost every epoch till 30 epochs and probably there is further scope to improve the model.

fit_generator functions used to train a deep learning neural network.

Arguments:

- steps_per_epoch: it specifies the total number of steps taken from the generator as soon as one epoch is finished and the next epoch has started. We can calculate the value of steps_per_epoch as the total number of samples in your dataset divided by the batch size.

- Epochs: an integer and number of epochs we want to train our model for.

- validation_data can be either:
    - an inputs and targets list
    - a generator
    - an inputs, targets, and sample_weights list which can be used to evaluate the loss and metrics for any model after any epoch has ended.
- validation_steps: only if the validation_data is a generator then only this argument can be used. It specifies the total number of steps taken from the generator before it is

stopped at every epoch and its value is calculated as the total number of validation data points in your dataset divided by the validation batch size.

```
vgg19_model.fit(train_data, epochs = 30,validation_data=test_data)
```

```
479/479 [==============================] - 134s 279ms/step - loss: 0.5609 - Accuracy: 0.8249 - val_loss: 3.0151 - val_Accuracy: 0.5315
Epoch 3/30
479/479 [==============================] - 134s 279ms/step - loss: 0.5624 - Accuracy: 0.8266 - val_loss: 2.7510 - val_Accuracy: 0.5203
Epoch 4/30
479/479 [==============================] - 133s 279ms/step - loss: 0.5340 - Accuracy: 0.8344 - val_loss: 2.3881 - val_Accuracy: 0.5682
Epoch 5/30
479/479 [==============================] - 133s 278ms/step - loss: 0.5321 - Accuracy: 0.8365 - val_loss: 2.7862 - val_Accuracy: 0.5326
Epoch 6/30
479/479 [==============================] - 147s 308ms/step - loss: 0.5028 - Accuracy: 0.8450 - val_loss: 2.3637 - val_Accuracy: 0.5699
Epoch 7/30
479/479 [==============================] - 133s 278ms/step - loss: 0.5119 - Accuracy: 0.8446 - val_loss: 2.5710 - val_Accuracy: 0.5359
Epoch 8/30
479/479 [==============================] - 147s 308ms/step - loss: 0.5185 - Accuracy: 0.8458 - val_loss: 2.7075 - val_Accuracy: 0.5448
Epoch 9/30
479/479 [==============================] - 148s 309ms/step - loss: 0.5338 - Accuracy: 0.8356 - val_loss: 2.3180 - val_Accuracy: 0.5855
Epoch 10/30
479/479 [==============================] - 132s 276ms/step - loss: 0.5041 - Accuracy: 0.8510 - val_loss: 2.4584 - val_Accuracy: 0.5610
Epoch 11/30
479/479 [==============================] - 133s 278ms/step - loss: 0.4811 - Accuracy: 0.8493 - val_loss: 2.5832 - val_Accuracy: 0.5476
Epoch 12/30
479/479 [==============================] - 132s 276ms/step - loss: 0.5772 - Accuracy: 0.8359 - val_loss: 2.7242 - val_Accuracy: 0.5699
Epoch 13/30
479/479 [==============================] - 132s 275ms/step - loss: 0.4289 - Accuracy: 0.8670 - val_loss: 2.5921 - val_Accuracy: 0.5688
Epoch 14/30
479/479 [==============================] - 149s 310ms/step - loss: 0.4857 - Accuracy: 0.8546 - val_loss: 2.4359 - val_Accuracy: 0.5766
Epoch 15/30
479/479 [==============================] - 131s 274ms/step - loss: 0.5063 - Accuracy: 0.8532 - val_loss: 2.6995 - val_Accuracy: 0.5649
Epoch 16/30
479/479 [==============================] - 134s 279ms/step - loss: 0.4761 - Accuracy: 0.8575 - val_loss: 2.5029 - val_Accuracy: 0.5733
```

7: Save the Model

The model is saved with .h5 extension as follows.
An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data.

Saving the Model

```
[ ]  vgg19_model.save('Vitamin.h5')
```