(AI screenshots with purposes at bottom)

**NOTE: I used a high-RAM T4 GPU runtime**

# Helper function definitions:

* I first chose a desired image size of 64x64 to try. I used helper functions given in the assignment description and I modified them slightly to work for my data loading.
* I then loaded the data with the size 64x64 into dataframes and used the plot\_digits\_with\_predictions helper function to view a sample of the images (in this case the first 20 in the dataframe from my Xray-data loading function).
* In order to see if this is the size I wanted, I loaded the data again in 128x128 images and compared those visually to my 64x64 images.
* Since I felt I wasn’t losing anything too meaningful with 64x64 instead of 128x128, I decided to stick with 64x64. I then reloaded the data in 64x64
* I defined various helper functions used throughout the assignment to help with the data, such as train/test splitting and a function to train the model over a given number of epochs.
  + For the train/test split, I intentionally did not use randomness just to get a split point and manually slice the data. I instead used a function to randomly shuffle the indices with the given train and test sizes I wanted. I did this because I didn’t want the chronologically first X samples to always be in the training set and last X2 to always be in the testing set.
    - If I had more time, I could refactor my code to use a train, validation, and test set.
  + To build the fullTrainModel function, I used PyTorch gradient descent to iterate over the given number of epochs and update the weights with respect to a loss function (as a parameter of this function) each epoch. I calculated the loss in batches of a given batch\_size parameter (default size 64 since I felt that would be efficient without losing any significant information). I also added options to print the AIC and BIC when using BCEwithLogitsLoss (which I used for binary classification) and CrossEntropyLoss (for multi-class classification), the loss functions over each epoch, the graph of the train and test loss over the epochs, and the training and testing accuracy on the whole train/test datasets at the end of the last epoch.
* I defined functions to carry out linear and nonlinear transforms on the train and test data. I felt this could help my models have more variations in data, such as image orientation or translation, to learn from. I also felt that this increased variation could help prevent overfitting on the training data, since the translations had random components to them. I also knew that CNNs, which were great for image classification, generally benefited from image transforms. In both cases, I wanted to have the option to use transforms.
* I defined classes for my dataset using the NumpyDataset class from our class (no pun intended), linear classifier class, a manually-defined CNN class, and an AI-generated HealthyCNN class.
* When I was on problem 3, I realized I needed some slight modifications to my functions and classes to make them compatible with multi-class classification. I decided that I didn’t want to mess with my original functions’ / classes’ compatibility for binary classification, so I simply remade different versions of them that were designed for multi-class classification and to work with Cross-entropy loss (since that required long-s which my old functions weren’t compatible with) instead of BCEwithLogitsLoss
  + I chose BCEwithLogitsLoss for binary classification because it combined the sigmoid function (which I needed for logistic regression) and cross-entropy loss (which I liked) while making it work for binary classification
  + **BCEwithLogitsLoss involves the sigmoid function, and thus my approach to problem 1 was a form of logistic regression**
* Note: I defined these functions while I was doing the notebook, but just moved them to the same section in the top when I knew they worked. That way, function definitions wouldn’t clutter the code.

# Data preprocessing:

* I initially thought that because each of the pixel values was on the same scale of 0 to 255, I would just divide each value by 255. I noted, however, that this was not good for my neural networks, since I would like to have some negative values to balance out activation function activations. As such, I chose to z-score-normalize my data.
* Noting that what I thought were outliers had lots of extra noticeable whiteness or blackness, I decided that an okay metric to normalize on was the sum of pixel values per image. I then z-score-normalized and removed outliers more than 3 training standard deviations from the training mean (over all training images).
* After outlier removal, I converted my x\_train and y\_train into a NumpyDataset with input tensor x\_train and output tensor y\_train because I felt NumpyDatasets would be good for me to work with.
* I also stored the original mean and std of x\_train as global variables, so I could convert the normalized x\_train back to the original scale by multiplying by the original std and adding the original mean
* I then did some exploratory data analysis in which I printed out the number of samples belonging to each class and found that label=1 and label=2 were relatively underrepresented.
  + In all, I found around 8162 images of label 0 (healthy), 2896 of label 1 (COVID), 1067 of label 2 (pneumonia), and 4752 of label 3 (other infections)
    - Note that these numbers could slightly change randomly depending on the randomness of my manually-defined train/test split
  + I noted this for problem 3 so that I could upsample labels 1 and 2 for multi-class classification
  + I noted that for binary classification, this was okay since the sum of counts of labels 1, 2, and 3 (IE unhealthy) was approximately equal to that of label 0 (healthy)

# 1:

* I made a duplicate of y\_train called y\_train\_binary (where any non-0 labels were converted to 1) and likewise y\_test\_binary from y\_test
* I chose AdamW as my default optimizer because with other optimizers, I noted that my losses sharply decreased in the first few epochs and then stayed around the same. I wanted to make sure to have an adaptive momentum strategy to combat any issues with this.
* I first tried a linear classifier (a 1-layer neural network with each node (corresponding to a pixel) connecting to each other node) with an AdamW optimizer. I noted that this led to significant overfitting in the data (as the training loss was constantly oscillating without diminishing magnitude), so I tried with the SGD optimizer instead, and noticed a diminishing magnitude of oscillations.
  + For both the AdamW and SGD optimizers, I chose hyperparameters that helped to minimize the testing loss with respect to the loss function and prevent overfitting. I tuned these hyperparameters through trial and error.
* I then tried the linear classifier with AdamW and SGD optimizers with my linear train and test transforms enabled. I wanted to see if this would help prevent the overfitting I experienced without these transforms.
  + The linear train and test transforms significantly worsened train and test accuracies for my linear models. I did not use the nonlinear transforms for my linear classifiers because I did not want linear classifiers to try and learn over nonlinearly-transformed data.
    - For linear classifier with AdamW optimizer:
      * no linear transforms -> training accuracy 82.8%, testing 79.1%
      * with linear transforms -> training accuracy 70.8%, testing 64.5%
    - For linear classifier with SGD optimizer:
      * no linear transforms -> training accuracy 81.8%, testing 80.3%
      * with linear transforms -> training accuracy 68.9%, testing 66.8%
  + This was possible because a single linear layer with size 64x64 nodes could not have been complicated enough to capture the complexity of the dataset, and this complexity increased with the linear train and test transforms.
  + I chose a lower number of epochs because I noted that the train/test loss converged at a lower number of epochs (with the linear transforms enabled) than it did with the non-transformed linear classifiers
  + I noted that, compared to the non-transform linear classifiers, the overfitting was not as much with the linear transform-enabled classifiers, but the accuracy decrease and loss increase was still far too much.
* After seeing the overfitting with linear classifiers, I realized that this could be because a linear model might not be suitable for this (IE the data could have a nonlinear distribution) and chose to try a CNN, since I thought that this was usually good for image classification.
  + I did not want my CNN to rely on linear train and test transforms because CNNs were designed for nonlinear data. I also knew from past experience that transforms were often very good in training CNNs.
  + As such, I defined nonlinear train and test transforms that involved rotation and elements of randomness. I knew that these were often very good for CNNs.
  + Since I did not know much about choosing specific functions for CNNs, I just stuck with what the Google Colab AI autofill gave me.
* I also noted that I would want the momentum to be optimized over iterations, so I chose to stick with AdamW as my default optimizer.
* I tried a CNN with the AdamW
  + The CNN with AdamW optimizer did not overfit. Instead, the training and testing loss both decreased, and I took this as a good sign.
  + I used weight decay as well as a nonlinear transform to help reduce overfitting.
  + I tried various dropout\_probability values in the range of 0.4 to 0.7 and found 0.5 to yield the best results
    - Dropout = temporarily removing a random subset of the neurons from the network each iteration, so the model doesn’t overfit on the data; the remaining neurons then have to learn more robust patterns so that the network doesn’t depend too much on certain neurons
  + I was satisfied with this number of epochs because from the ending slope of the graph, it didn’t seem the loss would decrease much more with respect to the number of epochs. Because of this, having more epochs might have even led it to overfit!
    - The testing loss was very slightly fluctuating in the last few epochs
  + I also note that my AIC and BIC were gigantic (AIC was nearly 56K compared to my ≈ -5K on the overfitted linear classifiers without transformations and BIC was around 300K)
    - Perhaps having the transforms was responsible for largely-increased model complexity
  + Accuracy with AdamW: Training 81.2%, testing 82.7%
* To be sure I wasn’t missing out on better performance with the SGD optimizer, I tried that too. Unfortunately, with that, I got plenty of sharp increases in the testing loss followed by sharper decreases. As such, I wasn’t comfortable since it wasn’t consistently good over many epochs. It also had a decreased train/test accuracy versus with the AdamW optimizer.
  + CNN with SGD optimizer: training accuracy 73.3%, testing 76.0%
  + If I had more time, I would experiment more with various dropout probabilities
* I then used my AI-generated HealthyCNN model and noted that while its lowest loss was noticeably lower than with my manual CNN with AdamW optimizer, this was not consistent and there seemed to be slight overfitting from fluctuations in the testing loss.
  + I tried various dropout probabilities in the range of 0.4 to 0.7 and found 0.65 to be the best
  + Training accuracy of HealthyCNN overall: 82.1%, testing accuracy: 81.8%
* To further support my decision for choosing AdamW, I printed out the accuracies per unhealthy class of both models, and found that my AdamW did significantly better on all 3 classes
  + Testing accuracy of CNN/AdamW over labels 1/2/3: 80.4%/85.9%/92.2%
    - Usually significantly lower in previous iterations of the code
  + Testing accuracy of HealthyCNN/AdamW on label 1/2/3: 62.2%/71.3%/84.1%
    - significantly lower than my original CNN with AdamW
* My chosen model (CNN with AdamW optimizer) did not, indeed, classify the different types of unhealthy lungs equally. It usually classified (depending on random train/test splits used to train it) label 1 (COVID) around 8-10 percent worse than label 3 (other infection) around 8-10 percent worse than label 2 (pneumonia), though the actual accuracy values varied per train/test split
  + To reiterate, testing accuracy of CNN/AdamW over labels 1/2/3: 80.4%/85.9%/92.2%
    - Usually significantly lower in previous iterations of the code
  + I note that these categories are not equally-represented, but surprisingly there is no clear trend in num. samples versus accuracy, so it seems to be random based on only classifying label 0 versus labels 1 through 3.
  + This could be because of certain features in the overall unhealthy data being more prominent in specific labels than other labels

# Problem 2:

* I decided to reduce the dimensionality of the dataset using PCA to get the number of components that explained various amounts of the variance (no pun originally intended).
* I first started by setting my transforms to None since PCA was a linear transformation of the data and I didn’t want any of my other transformations to confound the PCA.
* I initially solved for the number of components that explained 95% of the variance, which in this case was 186, but I figured this was too many components after printing out the number of components. Luckily, there was a simple trick with my already-calculated PCA to get the minimum number of components to explain other, lower thresholds of the variance proportion. I printed this out for the thresholds of 0.9, 0.85, 0.8, and 0.75 after already printing it out for 0.95.
* I decided that since the number of components wasn’t too large for 0.8 to 0.9 accuracy, I would prioritize accuracy over having more components in that range.
* In order to decide which one of these thresholds to use, I ran a random forest classifier for each of the thresholds, but this didn’t show any significant accuracy differences for each of the numbers of components. It also was good but not good enough for me to conclude I wanted to use that over SKLearn’s many other classifiers.
  + Accuracies: With 58 components, around 83.2% accuracy; 30 components –> around 84.0% accuracy; 20 components → around 82.9% accuracy
  + Note: I used the PCA-transformed training datasets for each of the corresponding numbers of components
* I then tried AdaBoost with decision trees because I heard that it was good for fixing the errors caused by decision-based classifiers like random forests and decision trees. Unfortunately, the accuracy not only was the same for all numbers of components but also much lower than the decision trees themselves
  + It was around 76.5% for 58 and 30 components (exactly the same for both) and 75.8% for 20 components.
* My AI then gave me the idea to try standard SKLearn classifiers for each of the numbers of components. It chose a GradientBoostingClassifier, SVC with RBF kernel, SVC with polynomial kernel, KNN with k=5, KNN with k=3, and MLP with hidden layer sizes (100, 50) and max iterations = 500.
  + I noted that the accuracies for all classifiers were significantly higher for 58 PCA components than for 30 or 20 components. As such, I chose to stick with 58 components for the future.
    - The accuracy values aren’t too important right now, just their size compared to the other accuracy values, and I’ll report them for after the models have been trained
  + I noted that over the various times I ran the notebook, the accuracies were consistently highest for the SVC with RBF and with polynomial kernels and the MLP classifier.
    - Consistently, KNN had slightly lower accuracy than the rest of the classifiers and the GradientBoostingClassifier had even lower accuracy. As such, I chose to continue testing the SVC with RBF and with polynomial kernel and the MLP classifier
* I first decided to test the SVC with RBF and polynomial kernels and MLP for overfitting. I planned to do this by testing various hyperparameter configurations for each model and checking both the training and testing accuracy
* I started by testing the SVC/polynomial kernel with different values of C (being 0.01, 0.1, 0.5, and 1.0). I noted that the training and testing accuracies were similar (IE no overfitting) for C = 0.01 and 0.1 and the training accuracy was slightly higher than the testing accuracy for C=0.5 and 1.0.
  + Although I thought there was some overfitting for C=0.5 and 1.0, the training and testing accuracies were significantly higher for C=0.5 and 1.0 than for C=0.01 and 0.1. This difference was enough for me to very confidently suggest using C=1.0 due to its 89% training accuracy and 85.8% testing accuracy.
  + I did not tune the value of gamma because I felt that would be computationally-expensive (due to the long time it took to test all of the tested C-values) and the default value of gamma was already calculated as 1/(num\_features \* X.var() ) rather than a fixed number overall.
* I did the same thing for the SVC w/ RBF kernel as for the SVC w/ polynomial kernel. I tested the same values of C (being 0.01, 0.1, 0.5, and 1.0) and made the same observations as I did for the polynomial kernel. The RBF kernel-classifier had slightly higher accuracy than the polynomial kernel-classifier, but it was close so I decided to keep both of them in mind.
* I then tested the MLP classifier with various hidden layer sizes ( (20, 10), (35, 25), (45, 25), (60, 30), (80, 40), and (100, 50) ) and noted that for every hidden layer size, it overfitted vastly on the training data (for smallest hidden layer size of (20, 10), training accuracy was 6% higher than testing accuracy; for largest hidden layer size of (100, 50), training accuracy was over 15% higher than testing accuracy; also, for the last 3 sizes, the training accuracy was almost always at least 99.9% and it even was 100% for the last 2 sizes.
* Then, I decided to concatenate the training and testing datasets to perform multiple train/test splits, IE k-fold cross validation, on both of the SVC classifiers and my CNN classifier from part 1. This way, I could get a better idea of the true metrics of each of the models. (I chose accuracy, precision, recall, and F1 as my metrics)
  + I chose k=3 for computational efficiency
  + NOTE: the average metrics for SVC classifiers were very similar even across different iterations of the notebook. The CNN metrics, however, varied somewhat.
  + (These may not be updated for the most recent occurrence of these cells in my notebook)
  + I also noted that my SVC classifiers had noticeably lower test metrics than train metrics, while my CNN did not
  + Train metrics for CNN, SVC(poly), SVC(rbf):
    - accuracy 0.720, 0.890, 0.901
    - precision 0.679, 0.880, 0.915
    - recall 0.867, 0.910, 0.891
    - f1 0.762, 0.895, 0.903
  + Test metrics for CNN, SVC(poly), SVC(rbf):
    - accuracy: 0.753, 0.846, 0.888
    - precision: 0.711, 0.849, 0.893
    - recall: 0.885, 0.875, 0.864
    - f1: 0.789, 0.860, 0.876
* Because I wanted to focus most on recall (since I want to minimize type 2 errors) on testing data and there wasn’t much of a difference between SVC with poly and rbf kernel on other metrics, I chose the SVC with poly kernel
  + I wanted to emphasize recall because I felt that a type 2 error (false negative, incorrectly diagnosing someone as healthy when they’re infected and thus not giving needed treatment) was much worse than a type 1 error (false positive, incorrectly diagnosing someone as infected when they’re healthy and thus giving them unneeded treatment)

# Problem 3

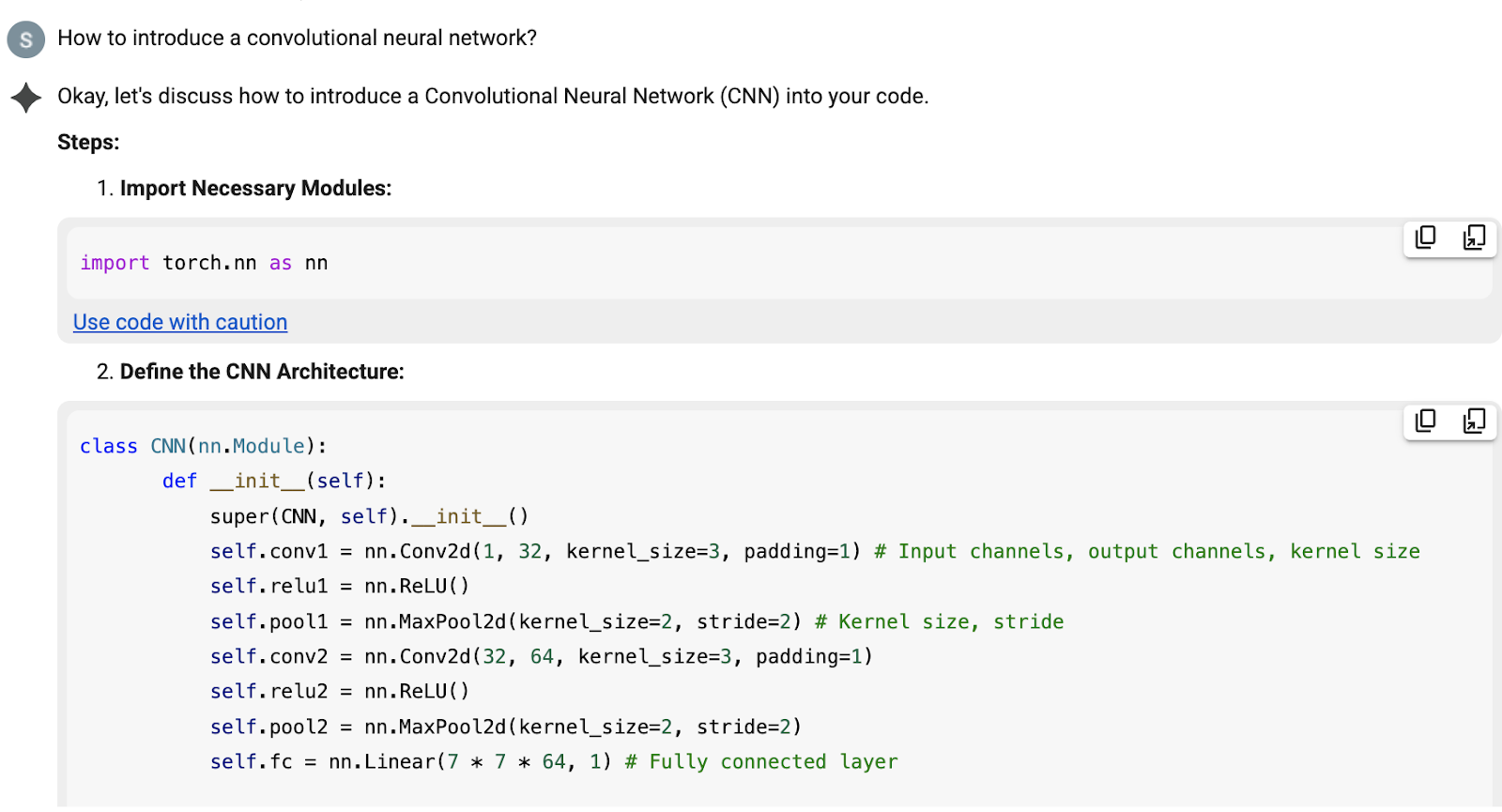
* I remade my dataset from the original, outlier-removed dataset prior to problem 1. I then printed the label counts again just to make sure that I didn’t mess up.
* I remembered that my label=1 and especially label=2 were underrepresented in the training dataset, so I decided to apply data upsampling to increase the counts of these labels (again, in the training dataset only)
  + My label=0 had around 8162 samples, my label=1 around 2917 samples, my label=2 originally around 1087 samples, and my label=3 around 4704 samples.
  + I thought that we should keep a very high number of healthy samples so as to not overfit on unhealthy samples. I also felt that there could be much more variety in healthy samples (due to genetics, life factors, etc.) than in infected samples.
  + I then decided that label=3, corresponding to all non-COVID and non-pneumonia infections, should have more samples than COVID and pneumonia but not too much more. This is because I felt non-COVID and non-pneumonia infections could have a much greater variety of data than COVID and pneumonia infections. I felt 4700+ samples was enough for label 3.
  + WIth all this information, I decided to upsample labels 1 and 2 to have 3500 samples of each of them in the dataset. I also decided to not do any sampling on label 0 and 3.
  + I didn’t apply upsampling in the test set because I noted that was not needed since we were only training the model on the training data and there wasn’t much of, if any, element of randomness in my models.
* I made sure to store a copy of the dataset before being upsampled in case I needed to come back to it later.
* With this upsampling, I made sure to filter out outliers (by sum of pixel values) again going off of images with a pixel sum greater than 3 standard deviations from the mean over all images. I found around 12 outliers this way and removed them.
* To verify all the outliers were removed, I printed the number of images in the training dataset greater than 3 standard deviations away from the mean (using the standard deviation and mean from the dataset prior to removing the outliers) to make sure all the relevant outliers were removed. In this case, it was 0, so I knew I was good.
* I made several adaptations to the functions and classes that I used for binary classification to work with multi-class classification. However, because I didn’t want to mess with the working of these functions for binary classification, I made separate functions that were identical to the binary functions with a few adjustments. These adjustments were necessary to work with multiclass classification.
* I also used a new loss function, CrossEntropyLoss, instead of BCEwithLogitsLoss from binary classification, since CrossEntropyLoss was tuned for multi-class classification.
* I trained a multi-class CNN (identical to my original CNN but designed to work with multiple classes) and got a training accuracy of 78.3%and testing accuracy of 81.0% this time around.
  + When training it with 10 epochs like my old model, I noted that the loss could still go lower after those 10 epochs. As such, I decided to try with 20 epochs, in which case I noticed the loss started fluctuating without decreasing overall. I decided from my plot of epoch number versus loss that 15 was a good number of epochs to use in this case.
* I computed its accuracy over the entire testing dataset as 81.0% and then proceeded to check its accuracy over each of the labels.
  + label 0 had 1849/2016 (91.7%) of its testing samples classified correctly (as label 0)
  + label 1 had 405/644 (59.8%) of its testing samples classified correctly
  + label 2 had 299/258 (88.8%) of its testing samples classified correctly
  + label 3 had 934/1269 (73.6%) of its testing samples classified correctly
  + Looking back, I could have taken the number of testing samples for each of these labels into account when deciding how much upsampling to do in the training dataset.
* If I had time, I would have adapted and tested more models (including adapting linear classifiers, SKLearn models, and the AI-generated HealthyCNN model) on the multiclass data.

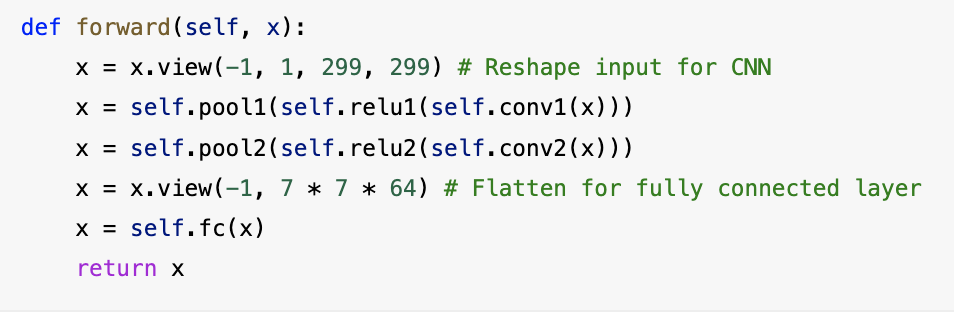
# Problem 4

* For label 0, the most confident case had lots of blackness and almost no whiteness in the lungs. This would make sense since most respiratory infections lead to buildup (white on the X-rays) in the lungs
* For label 1, the most confident case was extremely white and blurry to the point that it hurt my eyes (not physically hurt) to look at. This makes sense since COVID images are extremely white, implying that COVID leads to extreme build-up in the lungs.
* For label 2, I noted that there was significant whiteness except on the top left and bottom right on the image. This would make sense compared to other pneumonia images in the dataset.
* For label 3, I noticed that there was extreme white buildup like with COVID, but that there were discernable black portions throughout the lungs including some at the bottom, which is not a characteristic I saw with COVID images.
* Using a confusion matrix, I noted that the 3 most common types of misclassifications were:
  + truly otherInfection, misdiagnosed as normal (297 cases)
  + Truly COVID, misdiagnosed as normal (192 cases)
  + Truly normal, misdiagnosed as otherInfection
  + This makes some sense since non-COVID / non-pneumonia infections could very well lead to lungs that are similar to healthy lungs, such as low-intensity infections like the common cold or flu.
  + Also, some people could be non-infected but have other respiratory problems that make it seem purely from X-rays that they have one of the infections.
    - For people with common respiratory problems like asthma
  + I am not really sure how COVID cases could be misdiagnosed as normal, except maybe in cases where people with healthy lungs have lots of buildup such as (I’m not medically sure about this) maybe from asthma or other common respiratory problems.
* I noted that the label 3 being misclassified as label 1 made some sense, as the label 1 (COVID) images were very often extremely white and this example was extremely white at the bottom.
* I realized I could have filtered outliers based on label. I would have done this if I had more time (and before the upsampling).

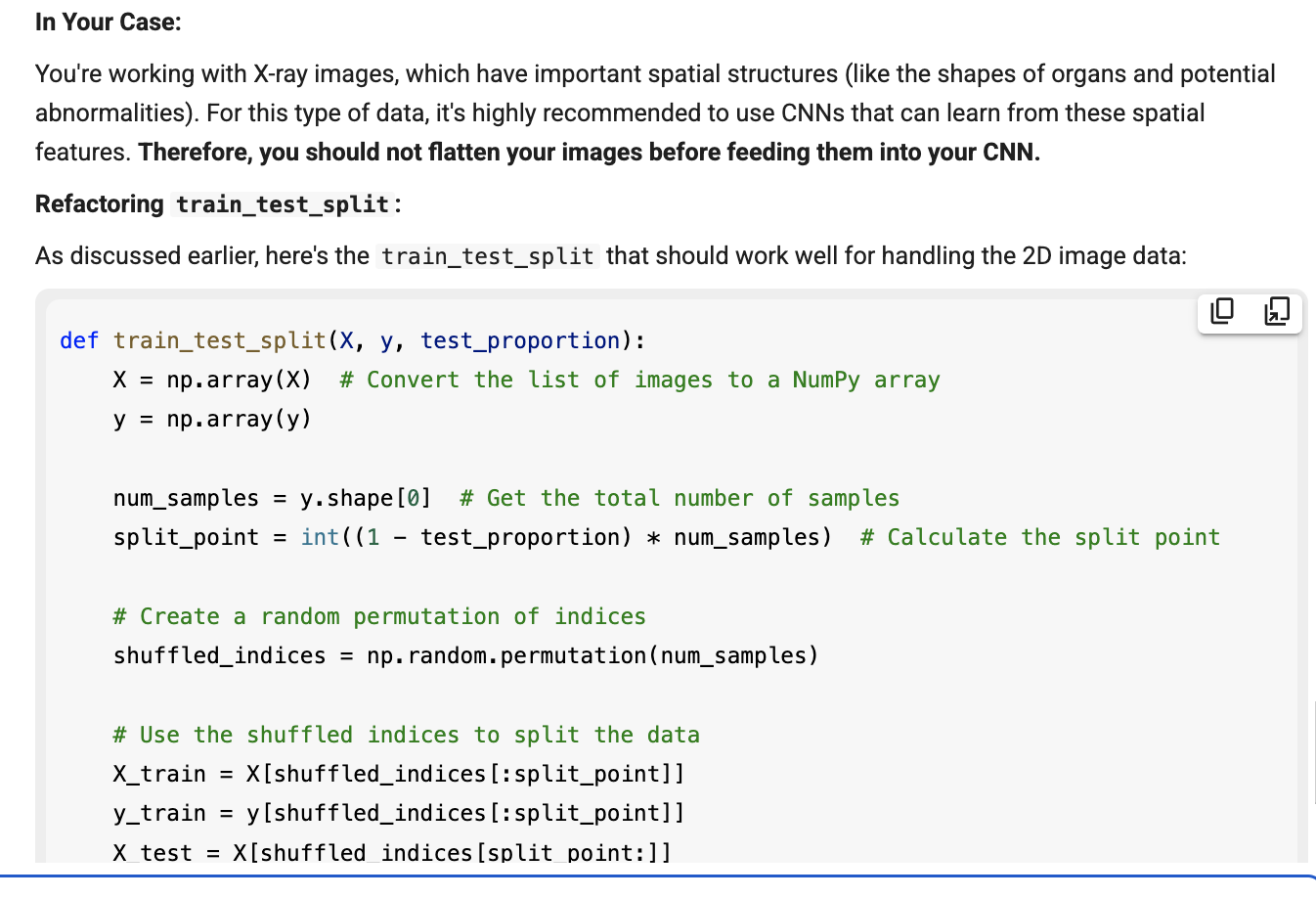
# AI screenshots

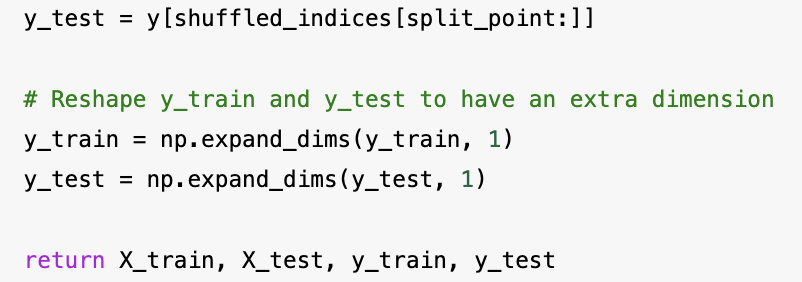
AI architecture for CNN





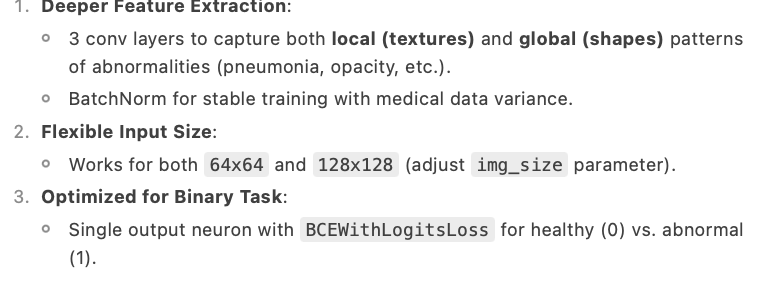
Refactor train-test split to handle 2D data



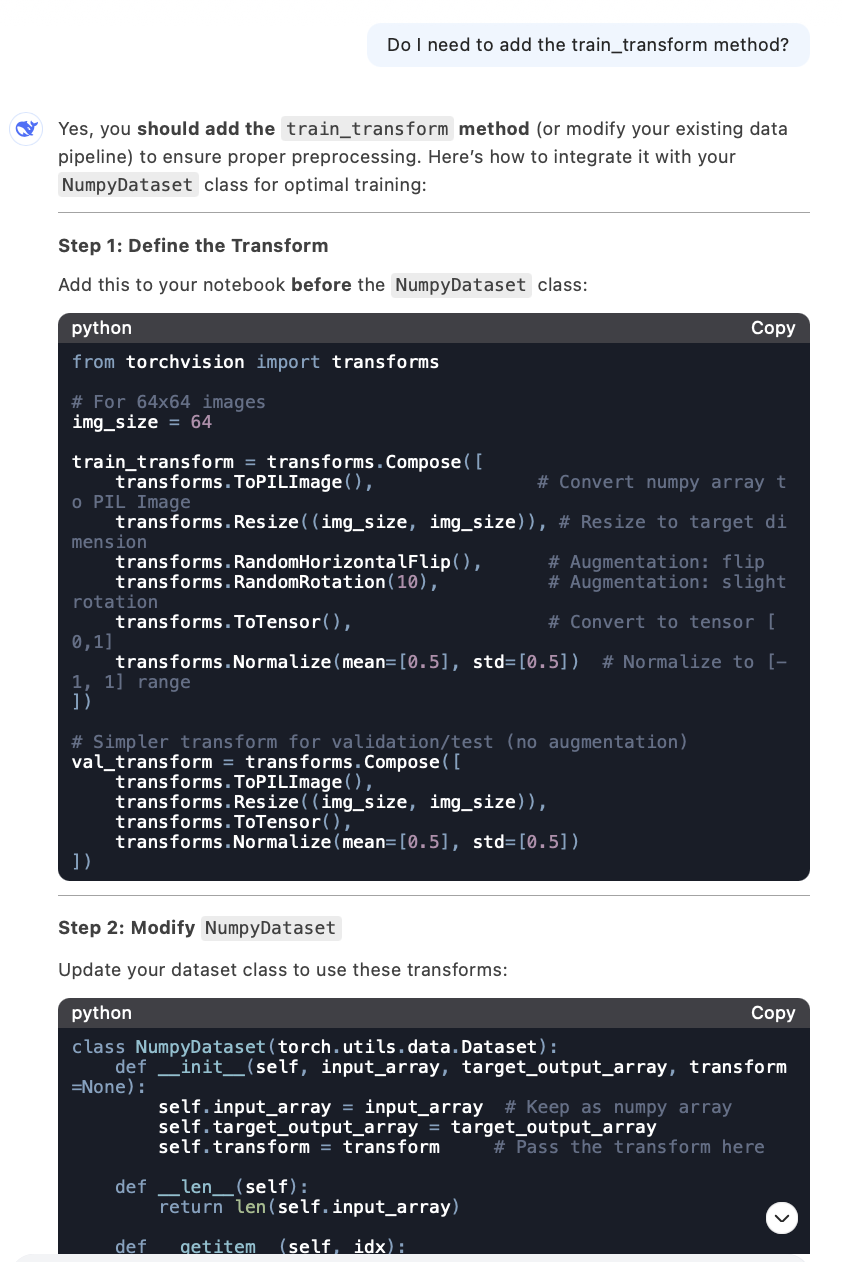


Deepseek architectures for CNN (which I used for HealthyCNN but not “CNN” )



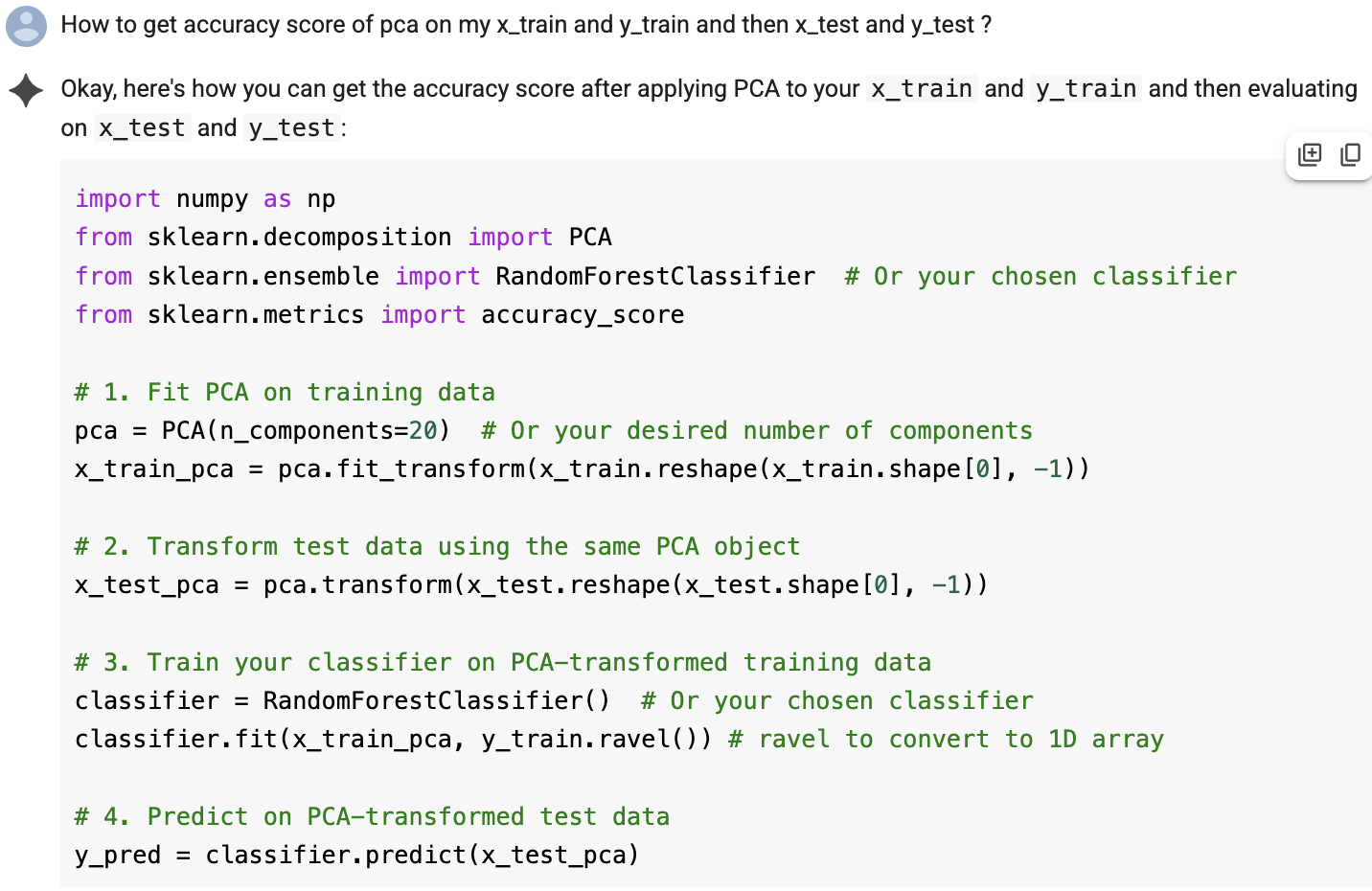


Integrate train\_transform method into HealthyCNN

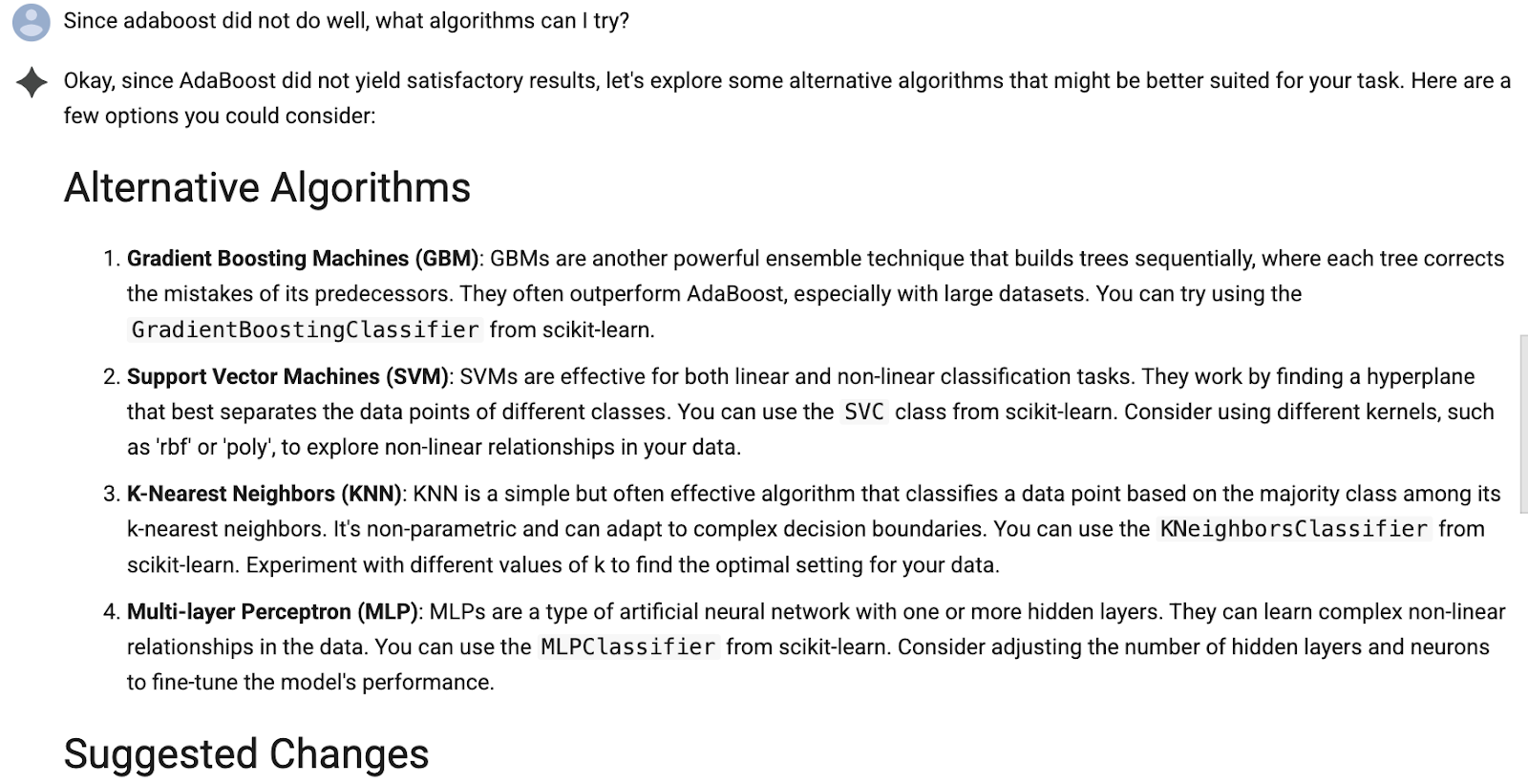




Getting accuracy score of PCA on train/test datasets

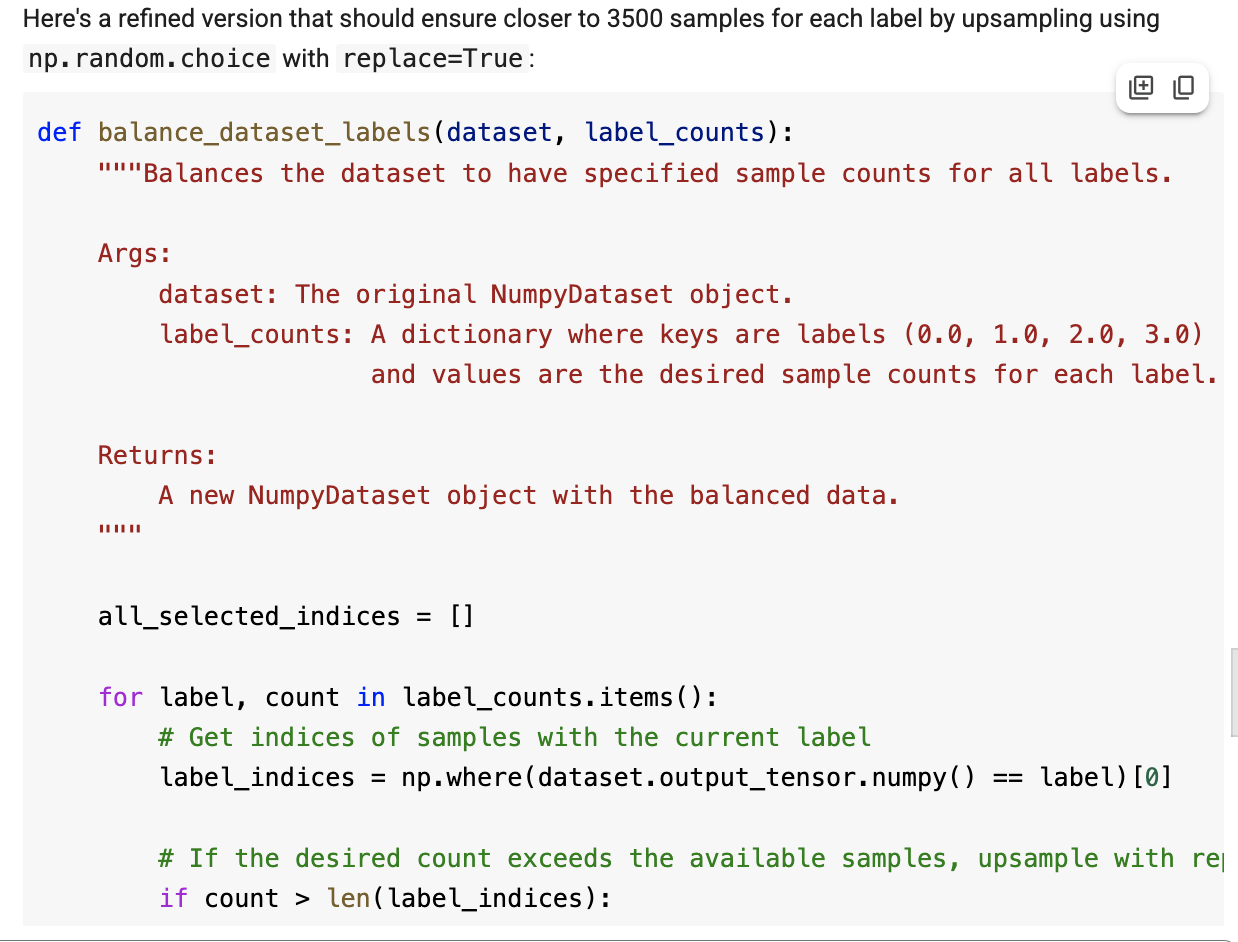


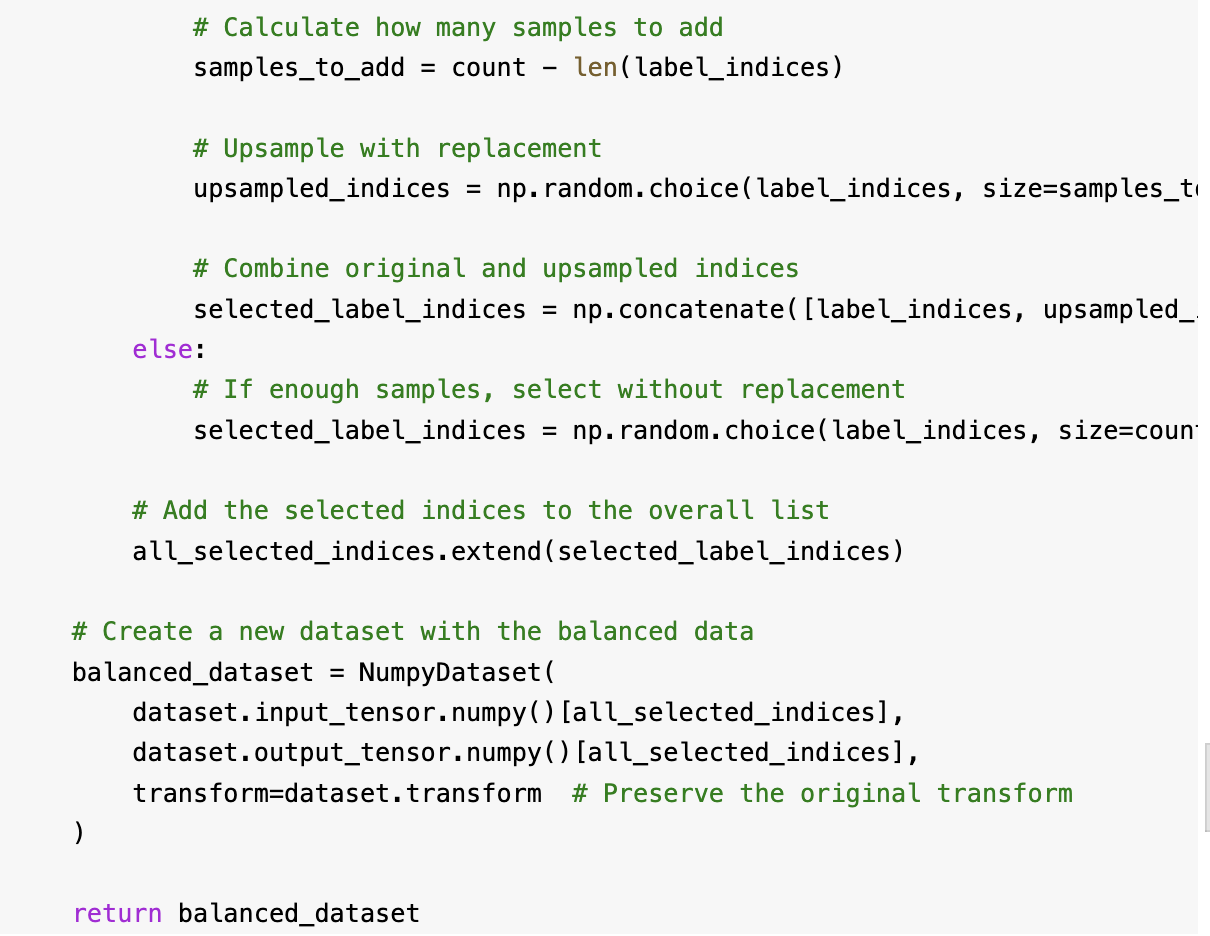
Suggestion to try various SKLearn classifiers



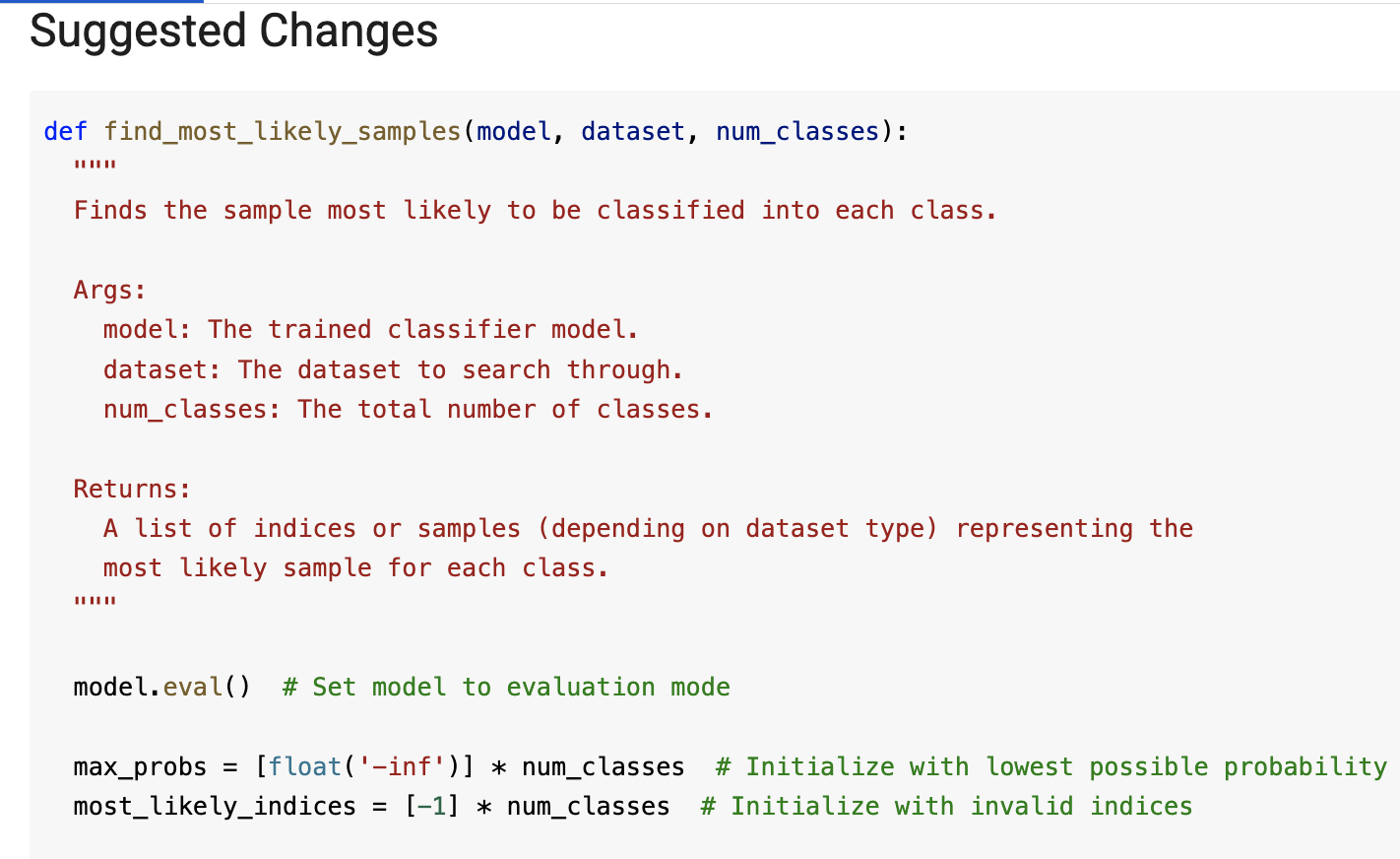


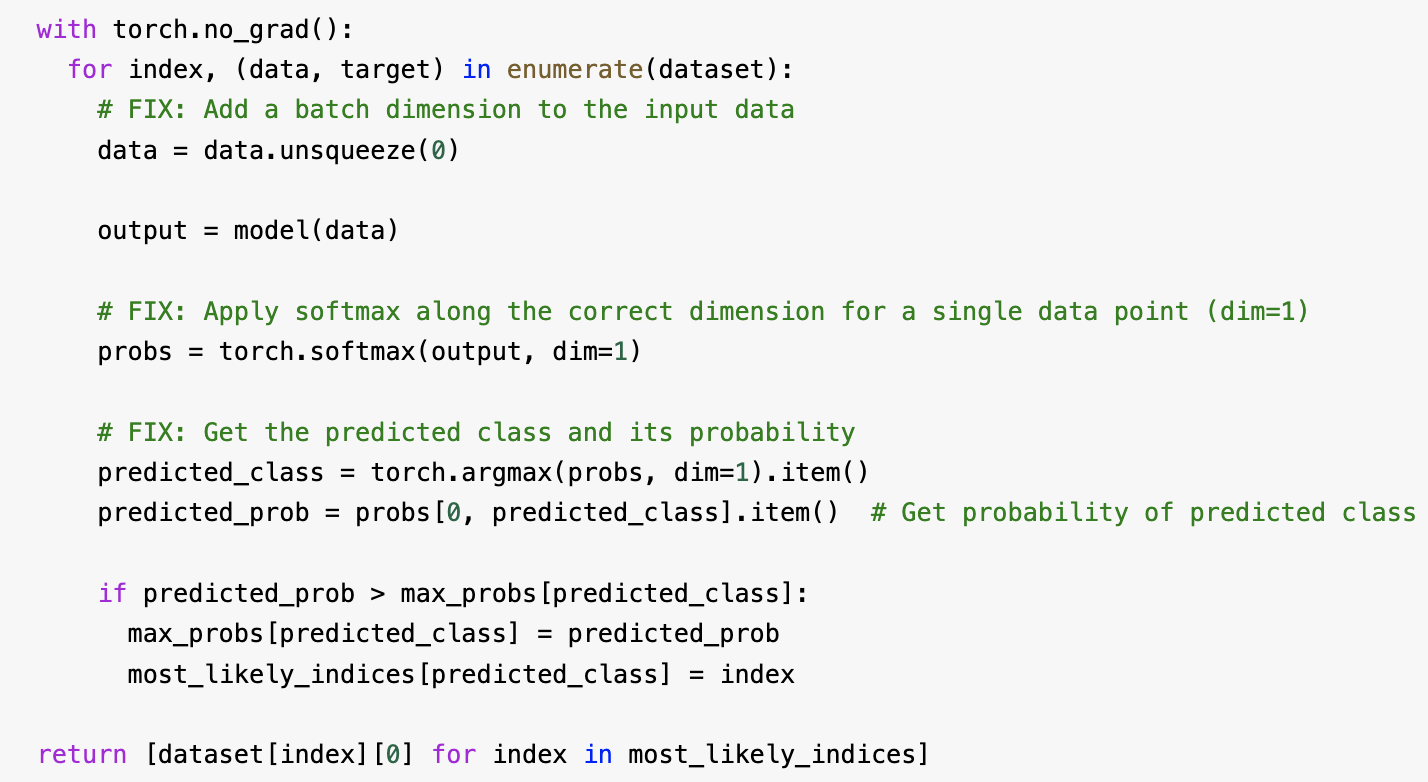
Function to upsample labels in training dataset

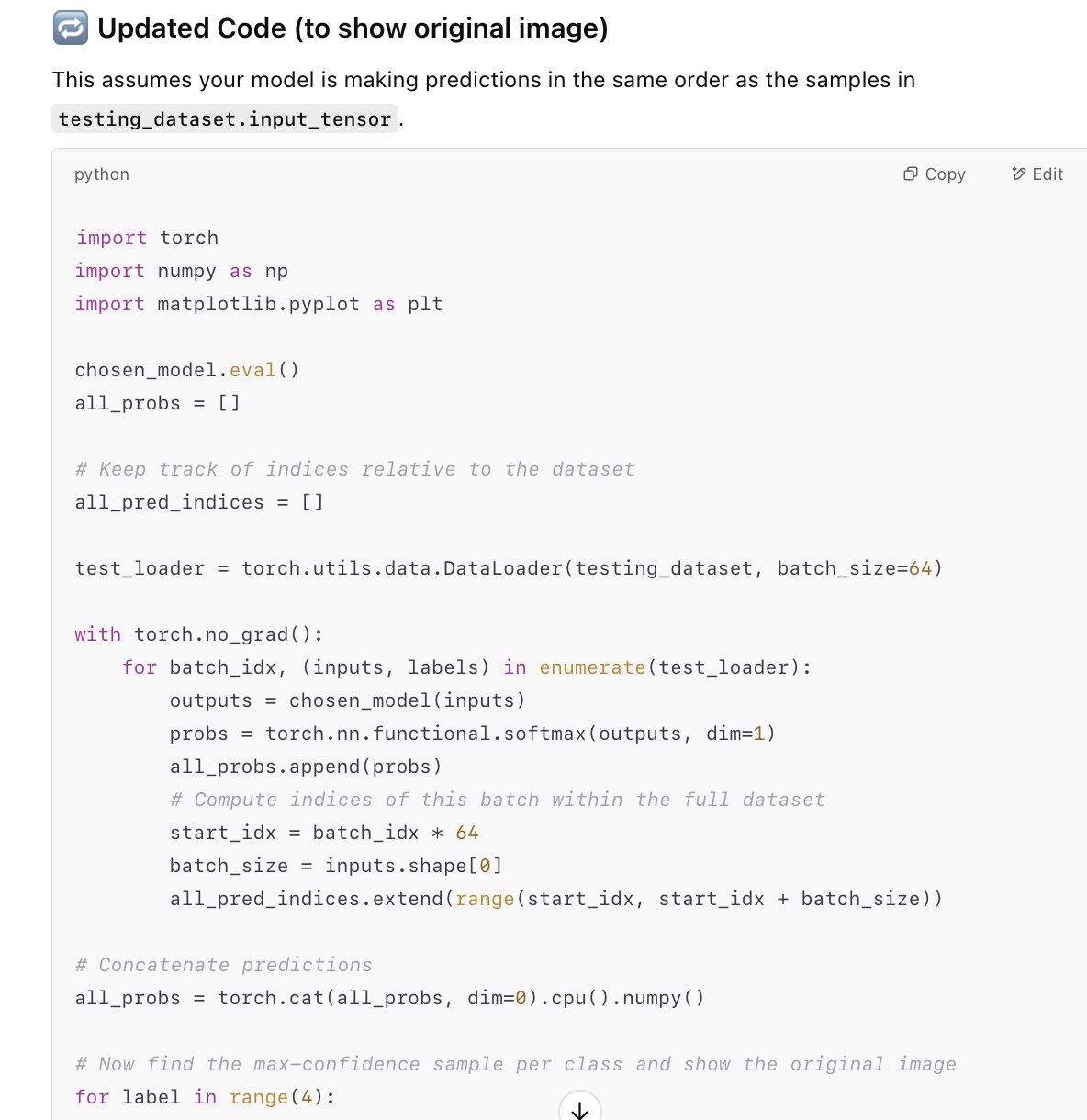


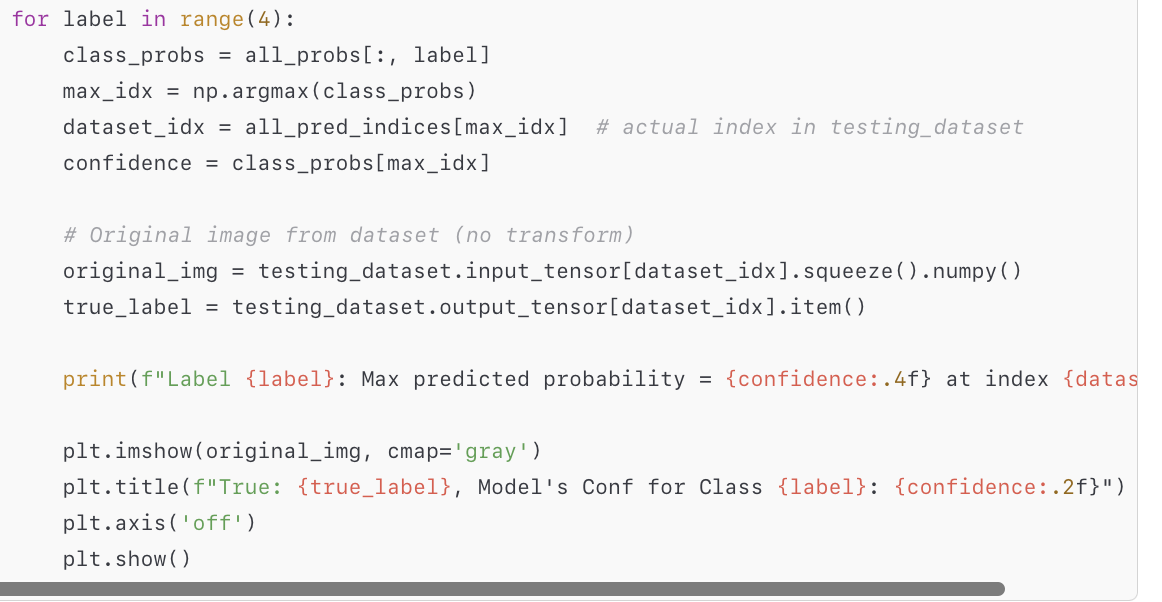


Code to find most likely samples and then show corresponding images









Code to upscale images for display



Code to compute and display confusion matrix

