```
import numpy as np
import os
from matplotlib import pyplot as plt
```

In this lab you will apply logistic regression to a dataset of images with the goal of classifying each image as a smile (1) or non-smile (0). We are using a dataset of faces called GENKI. First, we download the dataset and explore it.

```
# Download dataset
# Note: this only needs to be done once!
if not os.path.exists("trainingLabels.npy"):
  !wget https://s3.amazonaws.com/jrwprojects/trainingLabels.npy
  !wget https://s3.amazonaws.com/jrwprojects/testingLabels.npy
  !wget https://s3.amazonaws.com/jrwprojects/trainingFaces.npy
  !wget https://s3.amazonaws.com/jrwprojects/testingFaces.npy
```

```
--2025-10-10 10:02:19--  https://s3.amazonaws.com/jrwprojects/trainingLabels.npy
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.122.64, 52.216.50.16, 16.15.1
93.74, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.217.122.64|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16080 (16K) [binary/octet-stream]
Saving to: ‘trainingLabels.npy’

trainingLabels.npy  100%[===================>]  15.70K  --.-KB/s    in 0s

2025-10-10 10:02:20 (349 MB/s) - ‘trainingLabels.npy’ saved [16080/16080]

--2025-10-10 10:02:20--  https://s3.amazonaws.com/jrwprojects/testingLabels.npy
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.142.14, 52.216.50.16, 16.15.1
93.74, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.142.14|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 14704 (14K) [binary/octet-stream]
Saving to: ‘testingLabels.npy’

testingLabels.npy   100%[===================>]  14.36K  --.-KB/s    in 0s

2025-10-10 10:02:20 (286 MB/s) - ‘testingLabels.npy’ saved [14704/14704]

--2025-10-10 10:02:20--  https://s3.amazonaws.com/jrwprojects/trainingFaces.npy
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.43.96, 52.216.50.16, 3.5.14.1
51, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.43.96|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9216080 (8.8M) [binary/octet-stream]
Saving to: ‘trainingFaces.npy’

trainingFaces.npy   100%[===================>]   8.79M  3.80MB/s    in 2.3s

2025-10-10 10:02:23 (3.80 MB/s) - ‘trainingFaces.npy’ saved [9216080/9216080]

--2025-10-10 10:02:23--  https://s3.amazonaws.com/jrwprojects/testingFaces.npy
Resolving s3.amazonaws.com (s3.amazonaws.com)... 54.231.203.200, 52.216.50.16, 3.5.1
4.151, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|54.231.203.200|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8423504 (8.0M) [binary/octet-stream]
Saving to: ‘testingFaces.npy’

testingFaces.npy    100%[===================>]   8.03M  3.45MB/s    in 2.3s

2025-10-10 10:02:26 (3.45 MB/s) - ‘testingFaces.npy’ saved [8423504/8423504]
```

```python
# Load dataset
trainingFaces = np.load("trainingFaces.npy")
trainingLabels = np.load("trainingLabels.npy")
testingFaces = np.load("testingFaces.npy")
testingLabels = np.load("testingLabels.npy")
# Print out the dimensions of the arrays
print("trainingFaces shape: ", trainingFaces.shape)
```

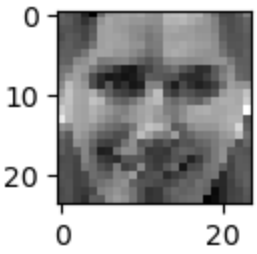
```
print("trainingLabels shape: ", trainingLabels.shape)
print("testingFaces shape: ", testingFaces.shape)
print("testingLabels shape: ", testingLabels.shape)
# Print out basic statistics
print("Proportion 'smile' in training:", trainingLabels.mean())  # Proportion of trai
print("Proportion 'smile' in testing:", testingLabels.mean())  # Proportion of testing
```

```
trainingFaces shape:  (2000, 576)
trainingLabels shape:  (2000,)
testingFaces shape:  (1828, 576)
testingLabels shape:  (1828,)
Proportion 'smile' in training: 0.5355
Proportion 'smile' in testing: 0.5464989059080962
```

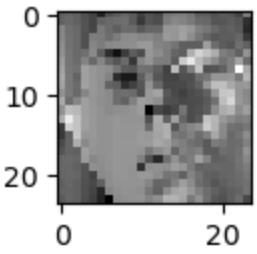
The labels are binary (1 = smile, 0 = non-smile). The images are stored as arrays of grayscale pixel values (each image contains 24*24 = 576 total pixels).

Let's visualize some of the faces. We first have to reshape each image from a 576-dimensional vector (which is convenient for classification with logistic regression) to a 24x24 array (which is convenient for visualization):

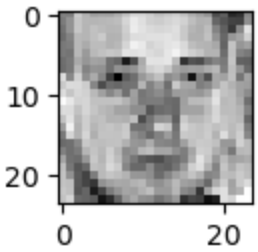
```
for i in range(5):
  # Before rendering, we have to reshape the image vector into a 2-d array
  fig, ax = plt.subplots(figsize=(1.25, 1.25))
  ax.imshow(trainingFaces[i,:].reshape(24, 24), cmap='gray')
  plt.show()
  print("label: ", trainingLabels[i])
```



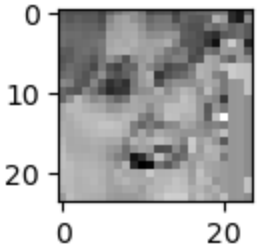

```
label:  1
```



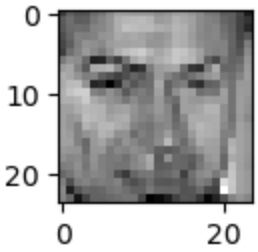

```
label:  0
```




```
label:  0
```



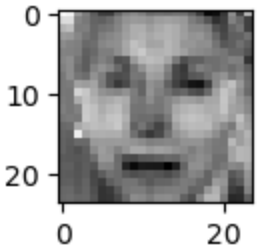

```
label:  1
```

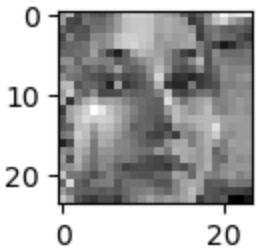


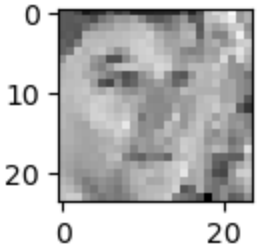

```
label:  1
```

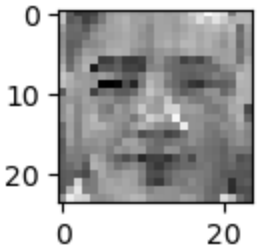
In most of the images, the label (smile vs. non-smile) is quite clear. However, in some it is not. Examine the 4 images below, and ask yourself: how would *you* classify them? (Note: you do not need to answer on Moodle.)

```
ambiguousIdxs = [ 187, 192, 566, 1401 ]
for idx in ambiguousIdxs:
  fig, ax = plt.subplots(figsize=(1.25, 1.25))
  ax.imshow(testingFaces[idx,:].reshape(24, 24), cmap='gray')
  plt.show()
```









Task 1a: Logistic Regression for Smile Classification (1 Point)

Your task: Train a logistic regression classifier (use the LogisticRegression class in sklearn) on the entire training dataset, and measure its accuracy (proportion of correctly classified images) on the testing data.

```
import sklearn.linear_model

logisticRegressor = sklearn.linear_model.LogisticRegression(max_iter=500)
# TODO: implement the rest here

logisticRegressor.fit(trainingFaces, trainingLabels)
testingAccuracy = logisticRegressor.score(testingFaces, testingLabels)
print("Testing accuracy: ", testingAccuracy)
```

```
Testing accuracy:  0.7538293216630197
```

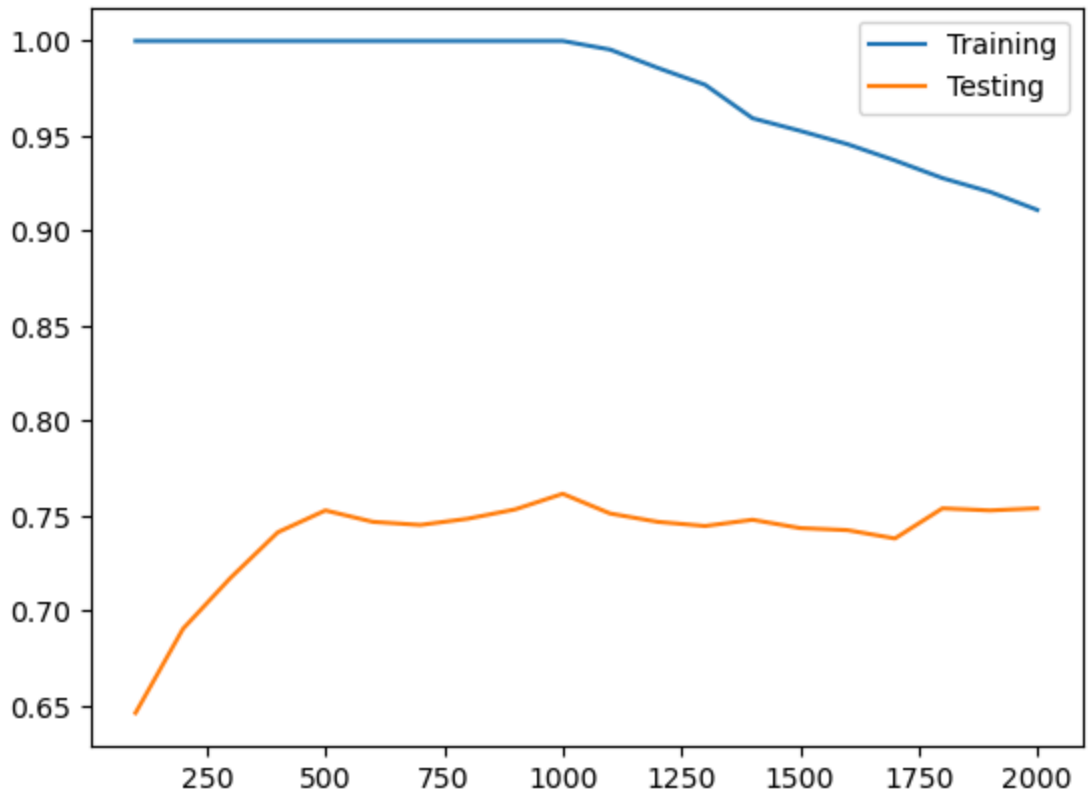
🦾 **HAND-IN** 🦾 : Enter the testing accuracy of the trained model on Moodle.

Task 1b: Measure the Effect of Training Set Size on Training & Testing Accuracies (4 pts)

As a general rule: the more training data you have, the higher your testing accuracy will be. In this next task, you will explore this issue quantitatively.

Your task: Using the provided training and test data, experiment with how the size of the training set impacts the test accuracy. In particular, vary M over 100, 200, 300, ..., 2000. For each M, train a logistic regression classifier containing M training examples (just use the first M examples) to recognize a smile (1) or non-smile (0). Then, measure *training* accuracy of the trained classifier (i.e., just the M examples you actually trained on) as well as the *test* accuracy (proportion of examples classified correctly). Plot these two curves in the same plot as a function of M. Why do you think the curves look the way they do?

```
import sklearn.linear_model

trainingAccuracies = []
testingAccuracies = []
Mvalues = np.arange(100, 2001, 100)
for M in Mvalues:
  logisticRegressor = sklearn.linear_model.LogisticRegression(max_iter=500)
  # TODO: implement the rest here
  # ...
  logisticRegressor.fit(trainingFaces[:M], trainingLabels[:M])
  train_acc = logisticRegressor.score(trainingFaces[:M],trainingLabels[:M])
  test_acc = logisticRegressor.score(testingFaces, testingLabels)
  trainingAccuracies.append(train_acc)
  testingAccuracies.append(test_acc)


plt.plot(Mvalues, trainingAccuracies)
plt.plot(Mvalues, testingAccuracies)
plt.legend([ "Training", "Testing" ])
plt.show()
```



📣 **HAND-IN** 📣 : On Moodle, upload your plot.

Task 2a: Examining the "Worst" Mistakes (2 pts)

You can sometimes learn something about both the dataset and a trained model by examining its "worst" mistakes, i.e., examples on which the model is *highly confident* but *wrong*. For this purpose, you can call the LogisticRegressor.predict_proba function, which returns a probability in the interval (0,1) rather than a binary prediction.

Your task: Find the top 25 *positively-labeled* examples in the test set on which the trained classifier (using M=2000) was *most confident* but *wrong*, i.e., yhat nearest to 0. Show these images and the machine's probability estimate yhat. Do the same thing for the top 25 *negatively-labeled* examples in the test set on which yhat was closest to 1.

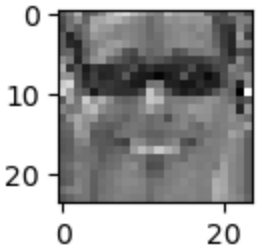
What do you observe about these particular examples and their labels?

```
# Utility method to plot the faces whose indices are specified in idxs
def show (idxs):
  for idx in idxs:
    fig, ax = plt.subplots(figsize=(1.25, 1.25))
    ax.imshow(testingFaces[idx,:].reshape(24,24), cmap='gray')
    plt.show()
    print(testingLabels[idx], yhat[idx])
```

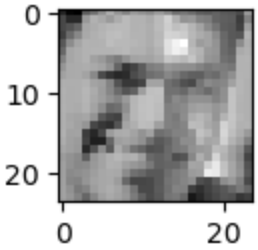
```
# In the line below, ",1" picks out the probability of the positive class (smile)
yhat = logisticRegressor.predict_proba(testingFaces)[:,1]

print("Positive")
# TODO: find the top 25 least worst mistakes for the positive examples

# calculate the differences between predicted and actual
diffs = yhat - testingLabels
# filter for positive examples that were misclassified (i.e., actual label is 1 but p
positive_misclassified_idxs = np.where((testingLabels == 1) & (yhat < 0.5))[0]
# get the corresponding differences
positive_diffs = diffs[positive_misclassified_idxs]
# get the indices of the top 25 least worst mistakes (i.e., smallest differences)
top_25_positive_idxs = positive_misclassified_idxs[np.argsort(np.abs(positive_diffs))
show(top_25_positive_idxs)

print("======================")
print("Negative")
# TODO: find the top 25 least worst mistakes for the negative examples

# calculate the differences between predicted and actual
diffs = yhat - testingLabels
# filter for negative examples that were misclassified (i.e., actual label is 0 but p
negative_misclassified_idxs = np.where((testingLabels == 0) & (yhat > 0.5))[0]
# get the corresponding differences
negative_diffs = diffs[negative_misclassified_idxs]
# get the indices of the top 25 least worst mistakes (i.e., smallest differences)
```

```
top_25_negative_idxs = negative_misclassified_idxs[np.argsort(np.abs(negative_diffs))]
show(top_25_negative_idxs)
```

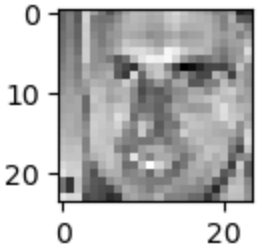
```
Positive
```



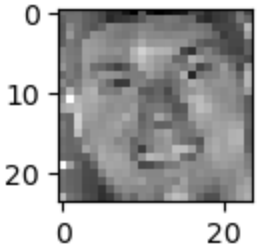

```
1 0.4957425193061427
```



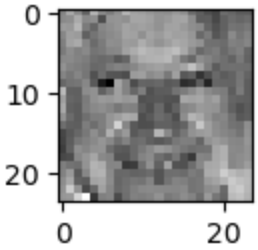

```
1 0.4907497321033063
```



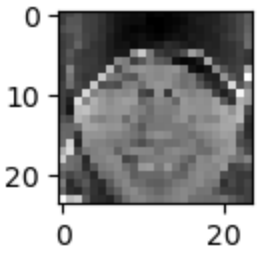

```
1 0.4833679806378545
```



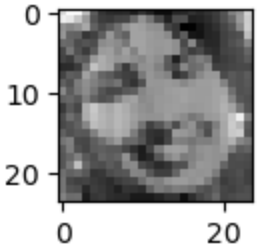

```
1 0.4812735220637092
```



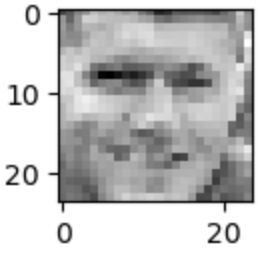

```
1 0.4805672936991524
```



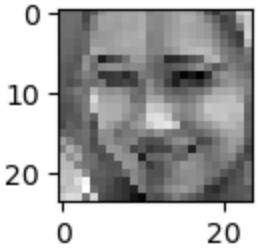
1 0.47837444751080366



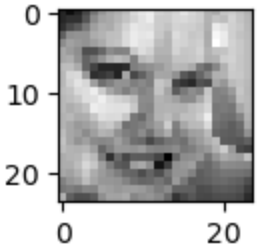
1 0.4763311615846756



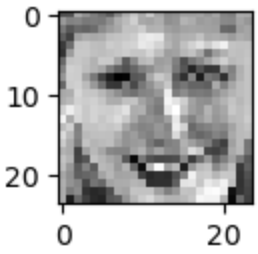
1 0.4756085429356579



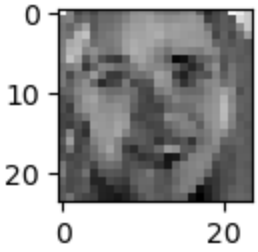
1 0.4746063472031017



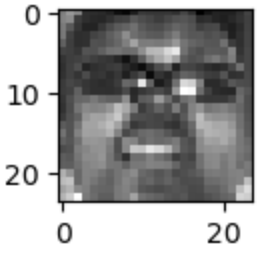
1 0.47134240480730294



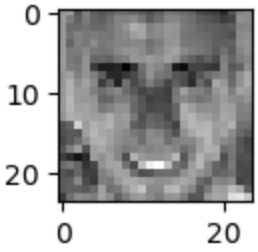
1 0.4707738003523833



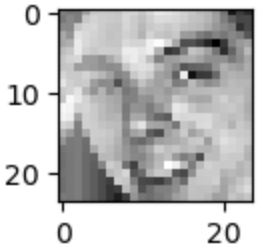
1 0.4690389896004297



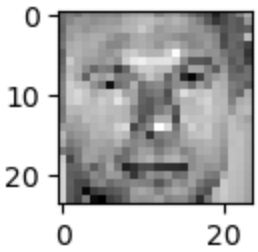
1 0.46202448229369153



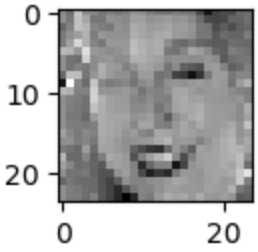
1 0.45890264256673274



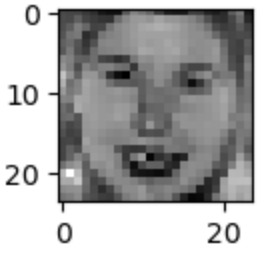
1 0.45311163690290723



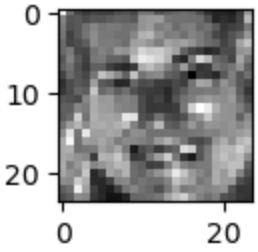
1 0.4526429668826311



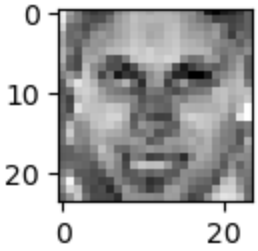
1 0.4509013786355376



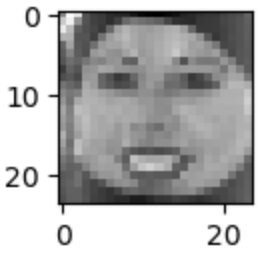
1 0.4500063263337436



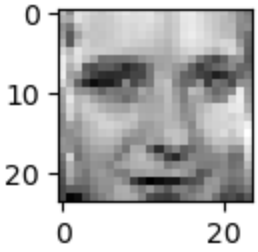
1 0.44639264040008636



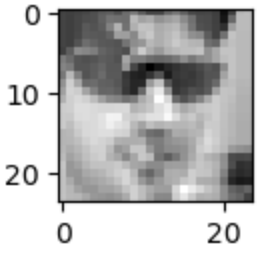
1 0.4448888951526985



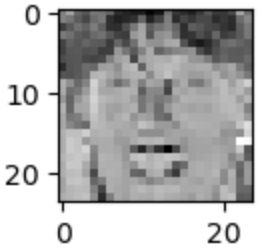
1 0.443997876085316



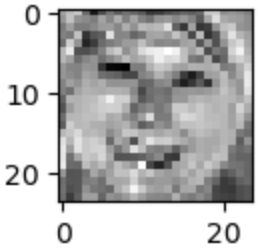
1 0.44371301344897857



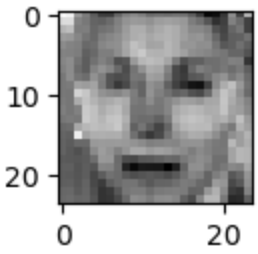
1 0.4413215395658857



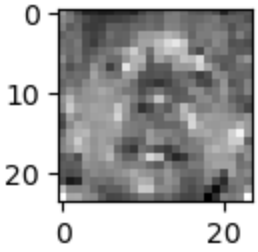
1 0.4383908308063232



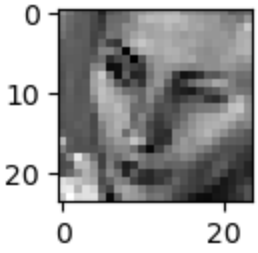
1 0.43277519188748037
==========
Negative



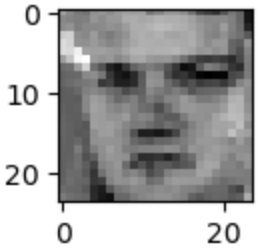
0 0.5007758170825396



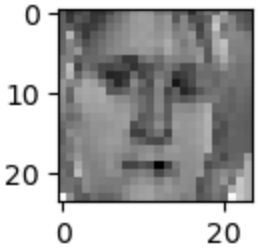
0 0.5016398433073078



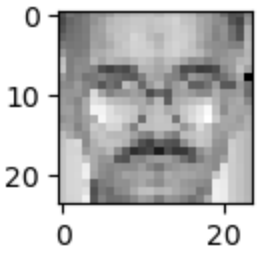
0 0.5038472849473437



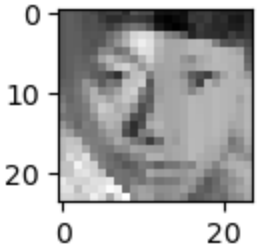
0 0.504465136265899



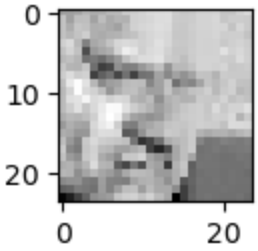
0 0.50491409291641



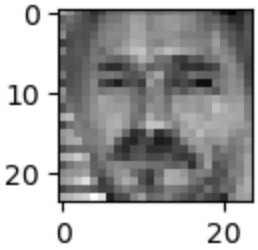
0 0.5115357498941958



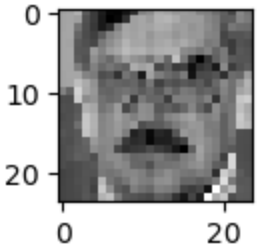
0 0.511615650551214



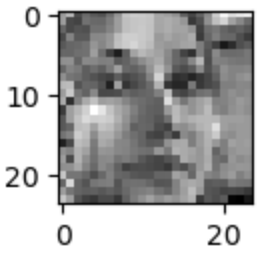
0 0.5177581961763883



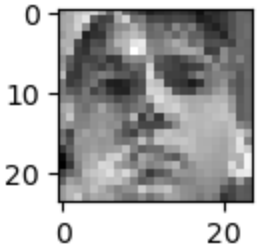
0 0.5237932760404287



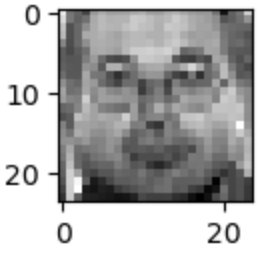
0 0.5249410523209941



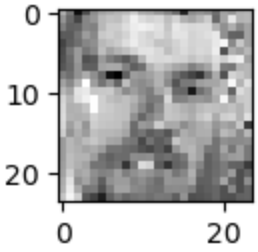
0 0.5249786038800869



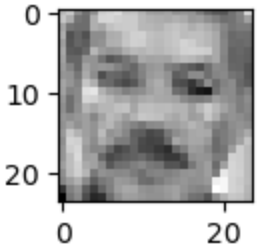
0 0.5251509352051392



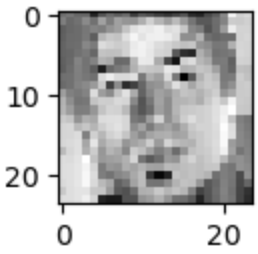
0 0.5282208999075448



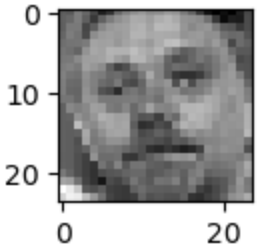
0 0.5369724325377402



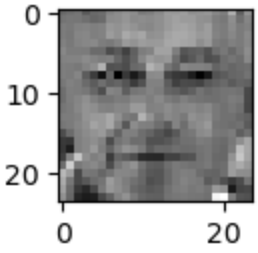
0 0.5374381443404597



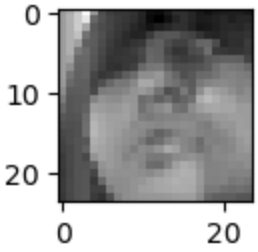
0 0.542060124144103



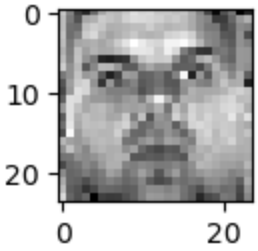
0 0.5421953324876091



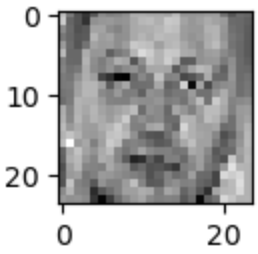
0 0.544406136418243



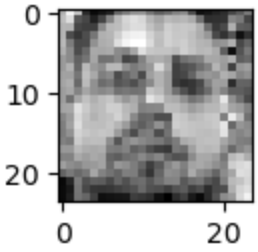
0 0.5458197265635942



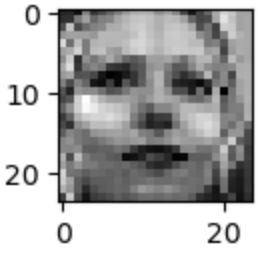
0 0.5563826704910898



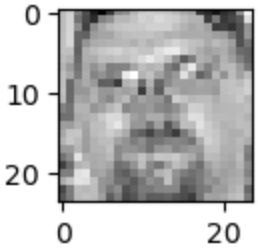
0 0.5610409021991405



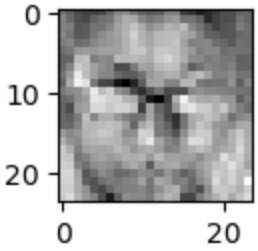
0 0.5612890486654226



0 0.5627219781389055



0 0.5675287848889561



0 0.5698198875132622

📢 **HAND-IN** 📢 : On Moodle, offer an explanation (referring to specific attributes of the faces, e.g., eye-glasses, lighting, etc.) about why those particular images were highly confident but incorrectly classified.

It is remarkable that many of the mistakes seem to have additional attributes like:

- eye-glasses
- Moustaches/beards
- Hats

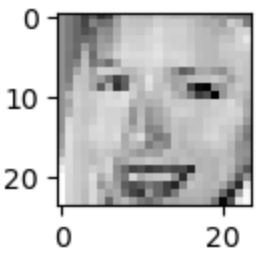
Also sometimes the angle of the face on the picture or the lighting seems to make it hard to classify.

Task 2b: Finding Least Confidently Predicted Examples

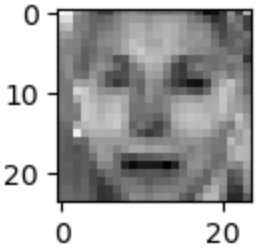
It can be instructive to find examples on which the trained classifier is uncertain/not confident. For this purpose, you can call the LogisticRegressor.predict_proba, which returns a probability in the interval (0,1) rather than a binary probability.

Your task: Find the top 25 examples in the test set on which the trained classifier (using M=2000) was *least confident* (i.e., yhat closest to 0.5) in its prediction. Show these images, the machine's probabilistic prediction, and the true label. Do you see any reason why those images might cause the machine some "confusion"? (Note: there may not be an obvious answer, but you can still speculate.)

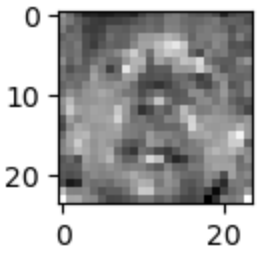
```
# TODO: find 25 least confidently predicted examples
# calculate the absolute differences between predicted probabilities and 0.5
abs_diffs = np.abs(yhat - 0.5)
# get the indices of the top 25 least confidently predicted examples (i.e., smallest a
# differences)
top_25_least_confident_idxs = np.argsort(abs_diffs)[:25]
show(top_25_least_confident_idxs)
```



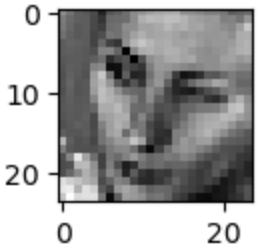

```
1 0.5000724072575876
```



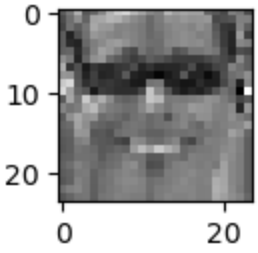

```
0 0.5007758170825396
```



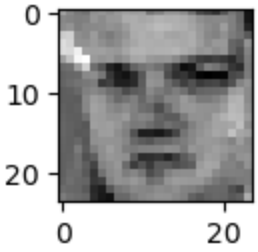
0 0.5016398433073078



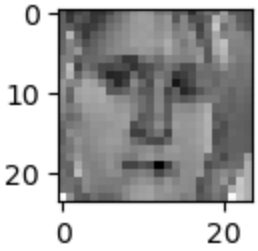
0 0.5038472849473437



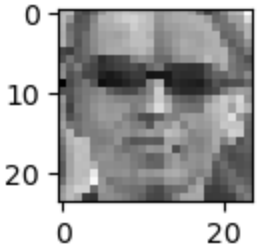
1 0.4957425193061427



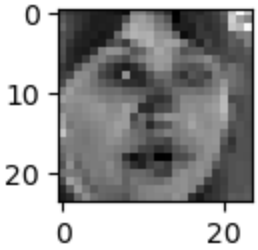
0 0.504465136265899



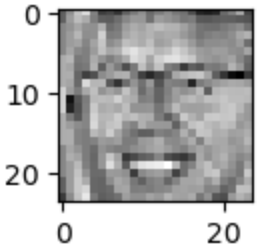
0 0.50491409291641



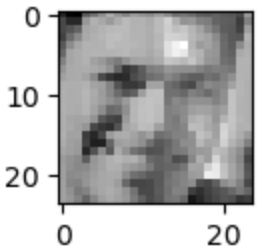
0 0.4938388360982426



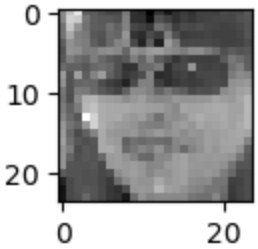
0 0.4934769370913107



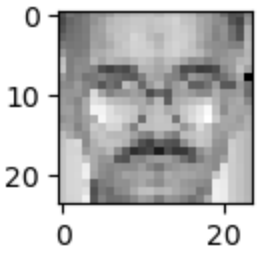
1 0.50768265884905



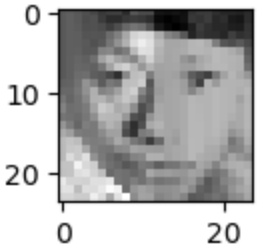
1 0.4907497321033063



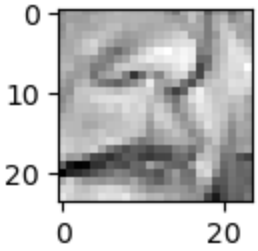
1 0.5099989517554783



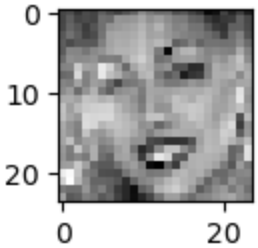
0 0.5115357498941958



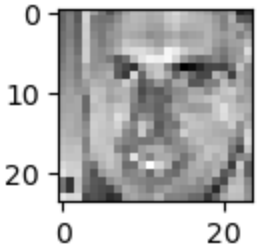
0 0.511615650551214



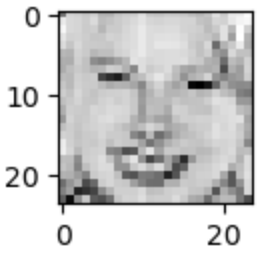
0 0.4841461428391261



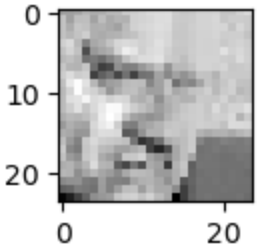
1 0.5160630476416355



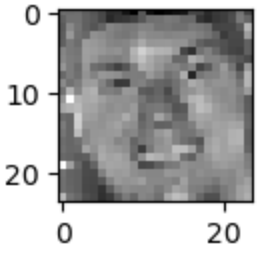
1 0.4833679806378545



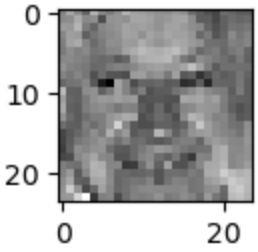
1 0.5166663805997135



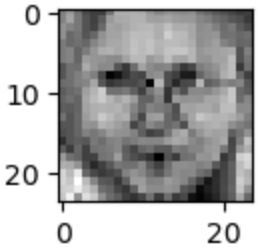
0 0.5177581961763883



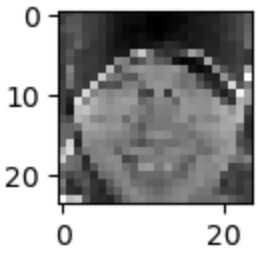
1 0.4812735220637092



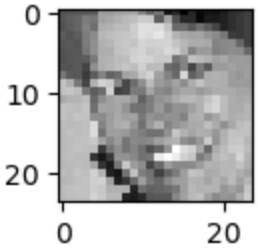
1 0.4805672936991524



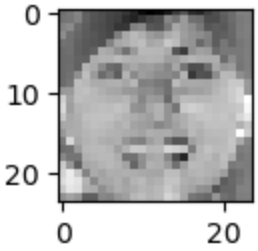
0 0.47944363774862697




```
1 0.47837444751080366
```




```
1 0.5218815253204537
```




```
1 0.5231864659342336
```

You do not need to submit anything on Moodle for this task.

Task 3: Data Augmentation (3 pts)

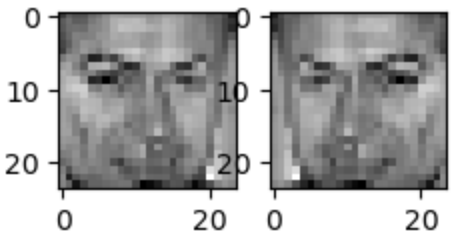
A perennial problem in machine learning is the lack of sufficient training data. One way to increase the amount of training data is to use **data augmentation**, i.e., to synthetically create more training examples from the ones that already exist. While this is possible only in certain situations, it can be very powerful. One setting in which it often is possible is when working with images. Face images in particular have a special kind of symmetry: if you "flip" a face left-to-right, then the person's facial expression basically remains the same. This means that, by flipping a face image left-to-right, we can create another face image with the same training label -- *without* having to manually photograph and label that example ourselves.

Below is a Python function that takes an array (M x 576) of face images and returns another array of the same size and contents, except that all the images have been flipped left-to-right.

```
def flip (faces):
    faces = np.atleast_3d(faces)
    faces = faces.reshape(-1, 24, 24)  # convert faces from vectors to 2-d arrays
    faces = faces[:, :, ::-1]  # flip all the 2-d arrays left-to-right
    return faces.reshape(-1, 24**2)  # convert faces from 2-d arrays to vectors
```

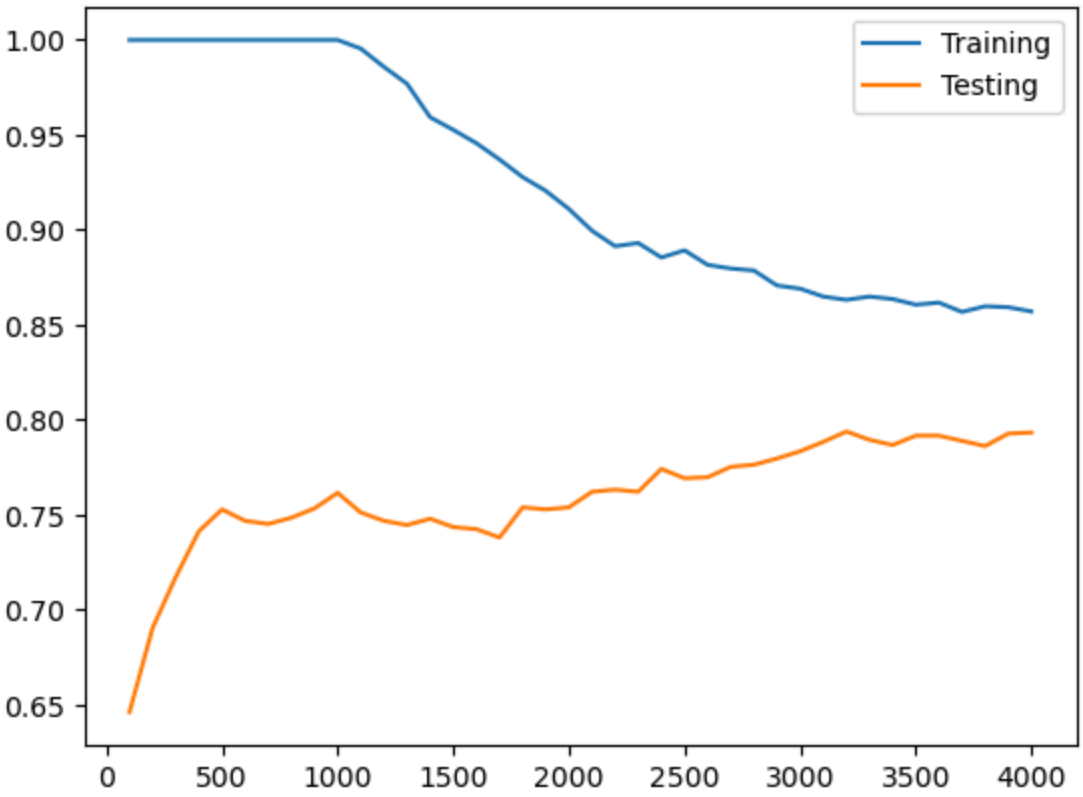
Here's an example of an original and a flipped face:

```
fig, ax = plt.subplots(1, 2, figsize=(2.5, 2.5))
ax[0].imshow(trainingFaces[4,:].reshape(24, 24), cmap='gray')
ax[1].imshow(flip(trainingFaces[4,:]).reshape(24, 24), cmap='gray')
plt.show()
```



Your task: Using the flip function, try training on a larger dataset: the original training set was just 2000 images, but with this data augmentation technique, we can get up to 4000 images. Does training on this larger dataset improve testing accuracy? Create a plot similar to the one you made for Task 1b, but this time vary M up to 4000.

```
# TODO: apply data augmentation and create plot, similar to Task 1b.
import sklearn.linear_model
trainingAccuracies = []
testingAccuracies = []
Mvalues = np.arange(100, 4001, 100)
augmentedTrainingFaces = np.vstack((trainingFaces, flip(trainingFaces)))
augmentedTrainingLabels = np.hstack((trainingLabels, trainingLabels))
for M in Mvalues:
    logisticRegressor = sklearn.linear_model.LogisticRegression(max_iter=500)
    logisticRegressor.fit(augmentedTrainingFaces[:M], augmentedTrainingLabels[:M])
    train_acc = logisticRegressor.score(augmentedTrainingFaces[:M], augmentedTrainingL
    test_acc = logisticRegressor.score(testingFaces, testingLabels)
    trainingAccuracies.append(train_acc)
    testingAccuracies.append(test_acc)
plt.plot(Mvalues, trainingAccuracies)
plt.plot(Mvalues, testingAccuracies)
plt.legend(["Training", "Testing"])
plt.show()
```



📣 **HAND-IN** 📣 : On Moodle, upload your plot (where M ranges up to 4000).

Task 4: Pushing the Limits (4 pts)

Now that you've trained a logistic regression classifier on this task, see how far you can "push the limit" and train as accurate a classifier as possible. This task is open-ended: you can try hyperparameter optimization, new forms of data augmentation, or other techniques.

Hyperparameter tuning: You may be able to increase the accuracy by optimizing the amount of regularization. In sklearn, the regularization strength of the logistic regression classifier can be tuned by setting the C parameter in the constructor of LogisticRegression. To optimize hyperparameters in a principled way, we need three data sets: training, validation, and testing. However, the data you are provided with contains just training and testing sets. Hence, you should sub-divide the training data into two subsets: "real" training (this is what you call fit() on), and validation (which you use to estimate how good each hyperparameter configuration is). As a suggestion, try using 80% of trainingFaces and trainingLabels for training, and the remaining 20% for validation.

Data augmentation: In addition to left-right flips, you can also try adding a small amount of (typically Gaussian) random noise to each pixel (using a small standard deviation of the Gaussian distribution, so that the face images still looks like a face). See np.random.randn. In order to avoid manually overfitting to the test data, you should also experiment with different forms of data augmentation on a separate validation set; then, once you find a good method of augmentation, test out the resulting classifier on the testing set.

📣 **HAND-IN** 📣 : On Moodle, upload your code above, and report the final testing accuracy.

Hyperparameter tuning

```
import sklearn.linear_model
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
```

```
# Split the augmented training data into training and validation sets
train_faces, val_faces, train_labels, val_labels = train_test_split(
    augmentedTrainingFaces, augmentedTrainingLabels, test_size=0.2, random_state=42)

print("Training set size: ", train_faces.shape[0])
print("Validation set size: ", val_faces.shape[0])

# divide the training set into real training and validation sets
real_train_faces, val_faces, real_train_labels, val_labels = train_test_split(
    train_faces, train_labels, test_size=0.2, random_state=42)

print("Real training set size: ", real_train_faces.shape[0])
print("Real validation set size: ", val_faces.shape[0])
```

```
Training set size:  3200
Validation set size:  800
Real training set size:  2560
Real validation set size:  640
```

```
# Rough Hyperparameter tuning
C_values = [0.01, 0.1, 1, 10, 100, 1000]  # Different values of C to try
best_C = None
best_val_accuracy = 0
for C in C_values:
    logisticRegressor = sklearn.linear_model.LogisticRegression(C=C, max_iter=500)
    logisticRegressor.fit(real_train_faces, real_train_labels)
    val_accuracy = logisticRegressor.score(val_faces, val_labels)
    print(f"C={C}, Validation accuracy: {val_accuracy}")
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_C = C
print(f"Best C: {best_C}, Best validation accuracy: {best_val_accuracy}")
```

```
C=0.01, Validation accuracy: 0.809375
C=0.1, Validation accuracy: 0.7796875
C=1, Validation accuracy: 0.7578125
C=10, Validation accuracy: 0.7515625
C=100, Validation accuracy: 0.7515625
C=1000, Validation accuracy: 0.753125
Best C: 0.01, Best validation accuracy: 0.809375
```

```
# Fine Hyperparameter tuning
C_values = np.arange(0.001, 0.1, 0.001)

best_C = None
```

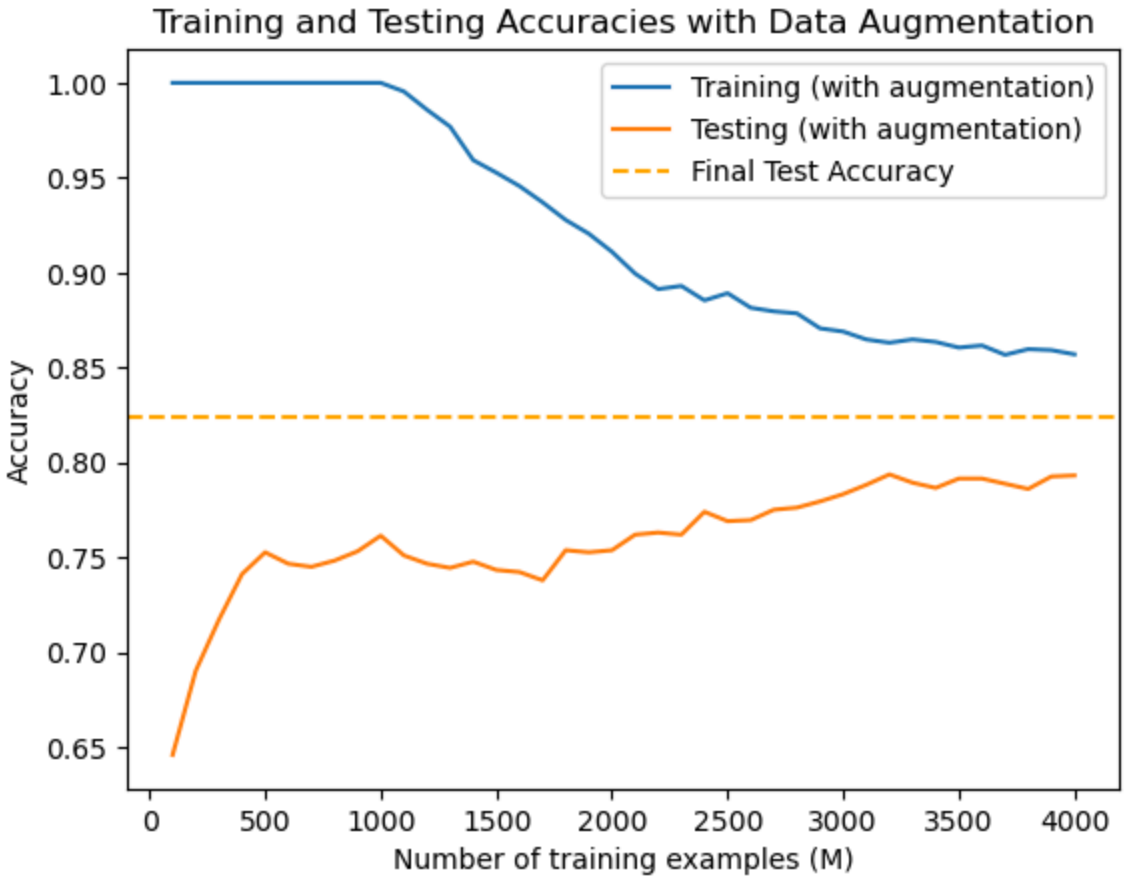
```
best_val_accuracy = 0
for C in C_values:
    logisticRegressor = sklearn.linear_model.LogisticRegression(C=C, max_iter=500)
    logisticRegressor.fit(real_train_faces, real_train_labels)
    val_accuracy = logisticRegressor.score(val_faces, val_labels)
    # print(f"C={C}, Validation accuracy: {val_accuracy}")
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_C = C
print(f"Best C: {best_C}, Best validation accuracy: {best_val_accuracy}")
```

```
Best C: 0.008, Best validation accuracy: 0.809375
```

```
# Train final model with best C on the entire augmented training set
final_logisticRegressor = sklearn.linear_model.LogisticRegression(C=best_C, max_iter=5
final_logisticRegressor.fit(augmentedTrainingFaces, augmentedTrainingLabels)
final_test_accuracy = final_logisticRegressor.score(testingFaces, testingLabels)
print(f"Final testing accuracy: {final_test_accuracy}")
```

```
Final testing accuracy: 0.8238512035010941
```

```
# Replot the accuracies from Task 3 for comparison
plt.plot(Mvalues, trainingAccuracies, label="Training (with augmentation)")
plt.plot(Mvalues, testingAccuracies, label="Testing (with augmentation)")
plt.axhline(y=final_test_accuracy, color='orange', linestyle='--', label="Final Test A
plt.legend()
plt.xlabel("Number of training examples (M)")
plt.ylabel("Accuracy")
plt.title("Training and Testing Accuracies with Data Augmentation")
plt.show()
```


Training and Testing Accuracies with Data Augmentation
Training (with augmentation)
Testing (with augmentation)
Final Test Accuracy
Accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
500
1000
1500
2000
2500
3000
3500
4000
Number of training examples (M)