ECE469 - Introduction to ML
Midterm - Part 1

Chase A. Lotito - *SIUC Undergraduate*

October 22, 2024

Question 1. *Regression in machine learning.*

Solution.

(A)

Unedited, the housing dataset contains 10 features, but eventually *ocean_proximity* will be dropped, and *median_house_value* will be chosen as the output target; all remaining columns will be the input features for the linear model.

```
HOUSING INFO
-----------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 1: housing.info()

(B)

```
HOUSING DESCRIPTION
-------------------
          longitude      latitude  housing_median_age  ...    households  median_income  median_house_value
count  20640.000000  20640.000000        20640.000000  ...  20640.000000   20640.000000        20640.000000
mean    -119.569704     35.631861           28.639486  ...    499.539680       3.870671       206855.816909
std        2.003532      2.135952           12.585558  ...    382.329753       1.899822       115395.615874
min     -124.350000     32.540000            1.000000  ...      1.000000       0.499900        14999.000000
25%     -121.800000     33.930000           18.000000  ...    280.000000       2.563400       119600.000000
50%     -118.490000     34.260000           29.000000  ...    409.000000       3.534800       179700.000000
75%     -118.010000     37.710000           37.000000  ...    605.000000       4.743250       264725.000000
max     -114.310000     41.950000           52.000000  ...   6082.000000      15.000100       500001.000000

[8 rows x 9 columns]
```

Figure 2: housing.describe()

(C)

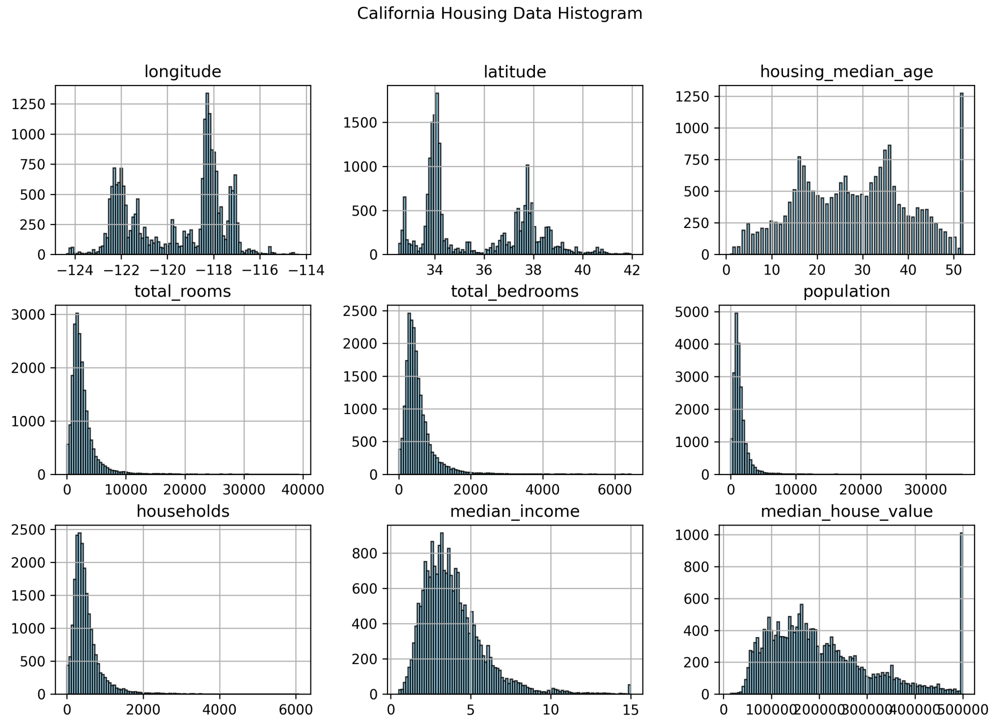

Figure 3: Housing Dataset Histogram

(D)

Using *SimpleImputer* from SciKit-Learn to replace missing data with the median value, and using *StandardScaler* from SciKit-Learn to standardize the dataset, the dataset looks as follows:

```
PREPROCESSED HOUSING DATA
-------------------------
       people_per_house  bedrooms_ratio  rooms_per_house  ...  households  median_income  median_house_value
0             -0.049597       -1.029988         0.628559  ...   -0.977033       2.344766            2.129631
1             -0.092512       -0.888897         0.327041  ...    1.669961       2.332238            1.314156
2             -0.025843       -1.291686         1.155620  ...   -0.843637       1.782699            1.258693
3             -0.050329       -0.449613         0.156966  ...   -0.733781       0.932968            1.165100
4             -0.085616       -0.639087         0.344711  ...   -0.629157      -0.012881            1.172900
...                 ...             ...              ...  ...         ...            ...                 ...
20635         -0.049110        0.165994        -0.155023  ...   -0.443449      -1.216128           -1.115804
20636          0.005021        0.021671         0.276881  ...   -1.008420      -0.691593           -1.124470
20637         -0.071735        0.021134        -0.090318  ...   -0.174042      -1.142593           -0.992746
20638         -0.091225        0.093467        -0.040211  ...   -0.393753      -1.054583           -1.058608
20639         -0.043682        0.113275        -0.070443  ...    0.079672      -0.780129           -1.017878

[20640 rows x 12 columns]
```

Figure 4: Preprocessed dataset

(E)

The standardized dataset when *latitude* and *longitude* are plotted against each other in a scatterplot shows a 2D map of all of the houses in the housing dataset.

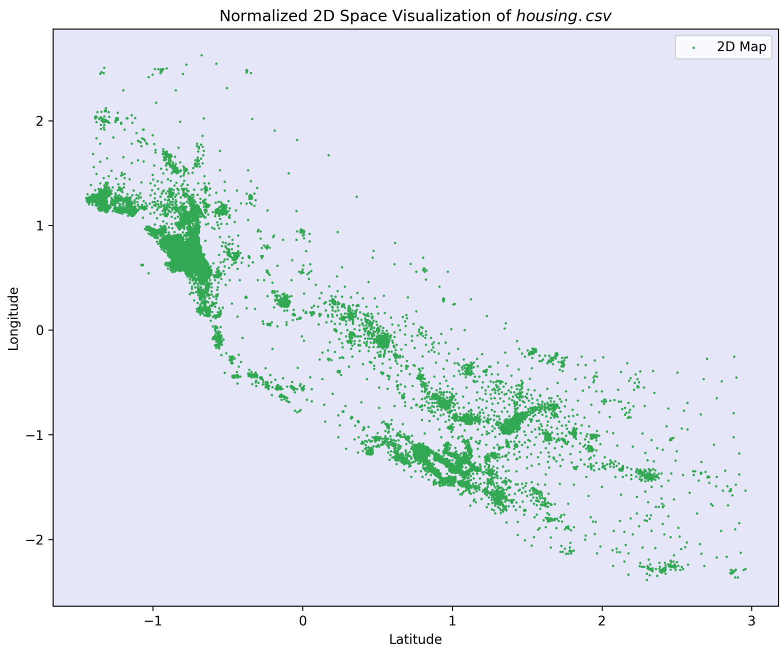

Figure 5: 2D Housing Map - Latitude Longitude Scatterplot

(F)

Finding the correlation between the input features and *median_house_value* we find *median_income* has a strong positive influence on the value of a home, but for example, *total_bedrooms* is not as important.

```
CORRELATION of ______ TO median_house_value
-------------------------------------------
median_house_value    1.000000
median_income         0.688075
rooms_per_house       0.151948
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049457
people_per_house     -0.023737
population           -0.024650
longitude            -0.045967
latitude             -0.144160
bedrooms_ratio       -0.233303
Name: median_house_value, dtype: float64
```

Figure 6: housing Scatter Matrix

(G)

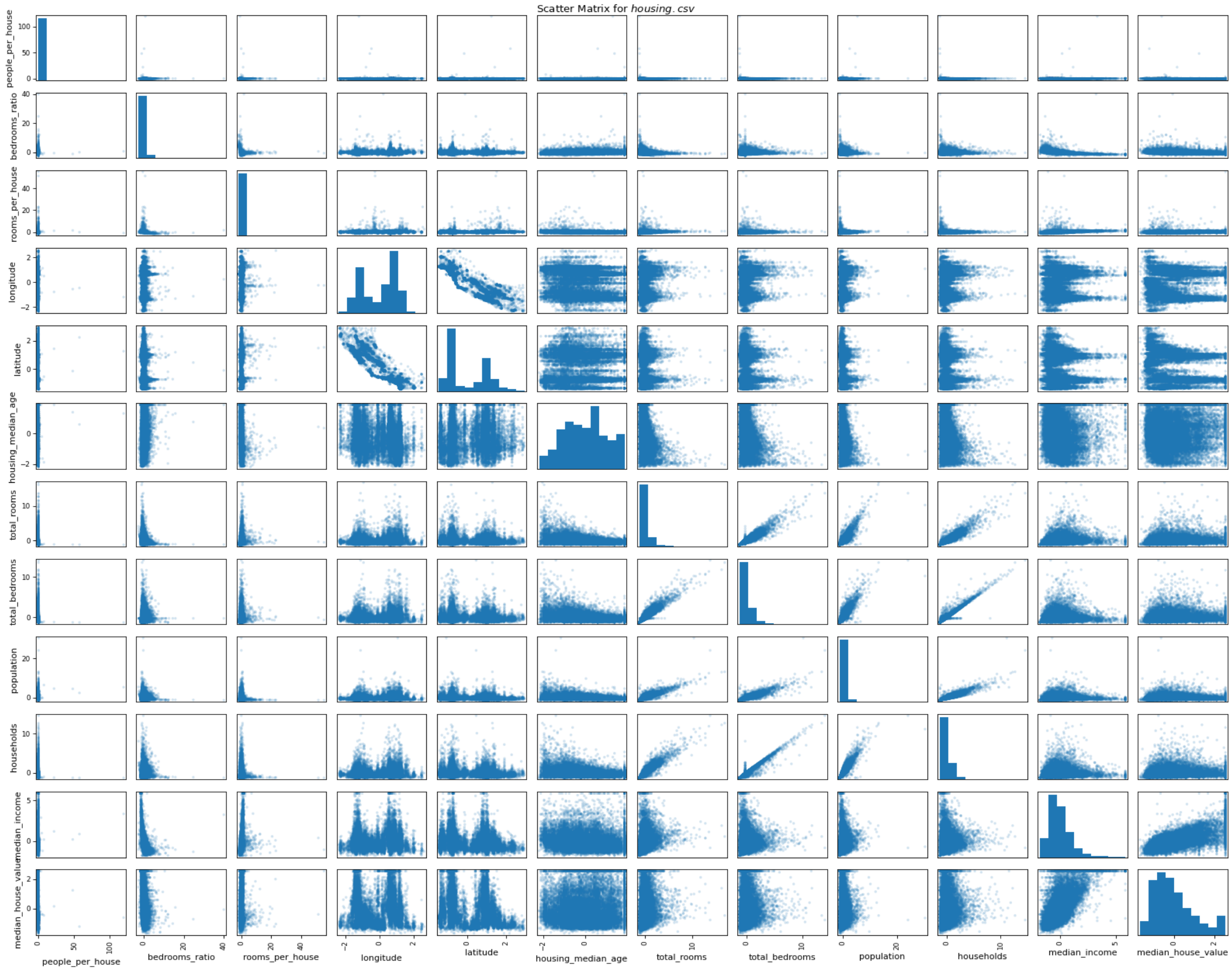

Figure 7: housing Scatter Matrix

(H)

The derived attributes *rooms per house*, *bedrooms ratio*, and *people per house* are created and inserted as input features in *q1.py* lines 73-88.

(I)

I chose to go the first route, where in lines 62-66, the missing values of *total_bedrooms*, and the rest of the dataset, were replaced with median values using *SimpleImputer*.

(J)

Below, in Fig. 8, we can see the predictions from our model plotted against the actual target values. Ideally, we would have all predicitions equal to the actual values, and every orange marker would lie along the blue curve. However, the linear model struggles to predict accurately, and we find a spread.

Also, the *for* loop defined on line 171 trains the linear model for varying sizes of the training set size (almost a kind of cross-validation), where the mean-square-error is calculated for each size (10%-100%), and are plotted in Fig. 8. Here we see testing error falling as the model is trained on larger and larger datasets, but drops off beyond 80% as the model loses generalization.

It is worth noting that attempting to send the input features of the housing dataset into polynomial space, and then doing a linear regression on that dataset kills any kind of accuracy for any degree other than 1 (1 being nothing changed, still linear).

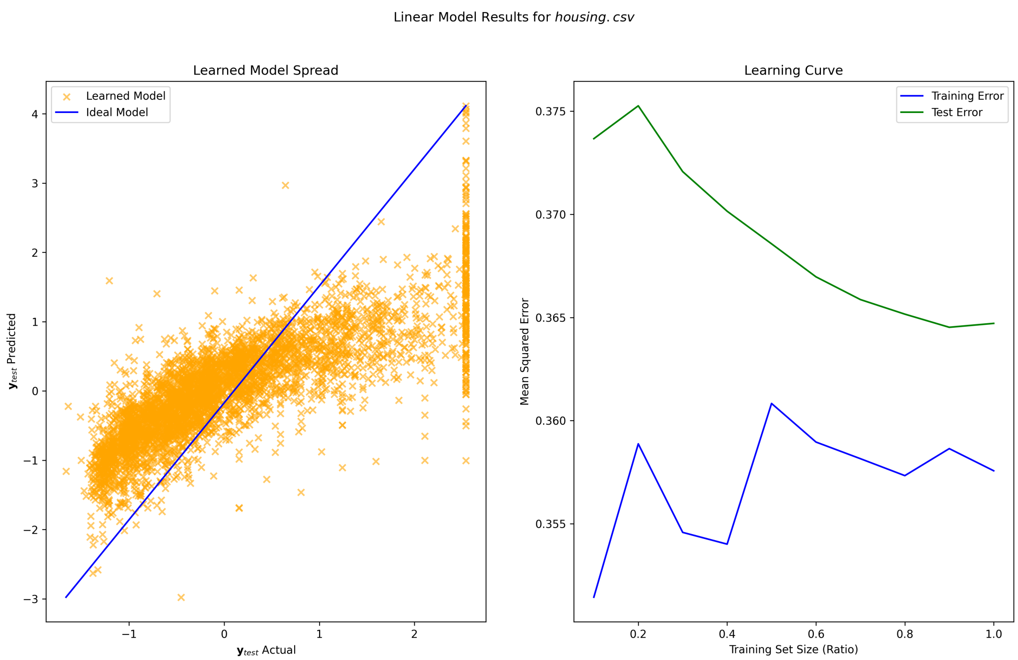

Figure 8: Linear Model Results: Model Spread (left), Learning Curve (right)

Below is the python code used for Question 1:

Listing 1: File: q1.py

```
1 """
2 Southern Illinois University Carbonale
3 Department of Electrical Engineering
4 --------------------------------------
5 ECE469: Intro to Machine Learning
6 Midterm Exam: Question 1
```

```
Chase Lotito

10/14/2024

"California Housing Prices"
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from pandas.core.common import random_state
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# import housing data from repository
url = "https://github.com/ageron/data/raw/main/housing/housing.csv"
housing = pd.read_csv(url)      # Store housing data in DataFrame
temp = housing

# important variables
plots_url = "./plots/q1/"

# (a) Use info() method to identify the attributes of this data-set
print('HOUSING INFO\n-----------')
housing.info()


# (b) Use describe() method to identify and peek at a summary of
#     the numerical attributes.
housing_desc = housing.describe()
print('\nHOUSING DESCRIPTION\n-------------------')
print(housing_desc)


# (c) Use hist() method on the whole dataset and plot a histogram
#     for each numerical attribute. Notice that many histograms
#     are skewed right.
housing.hist(bins=100, color='skyblue', alpha=0.8, edgecolor='black', figsize=(12,8))
plt.suptitle('California Housing Data Histogram')
#plt.savefig(plots_url + 'housing_histogram.png', dpi=300)
plt.close()

# (d) Clean and normalize/standardize the data-set to make it appropriate
#     for training a regression model. Creating training and test sets.
#     Create a copy of the data with only the numerical attributes by
#     excluding the text attribute ocean proximity from the data-set.

# remove ocean_proximity from the dataset
housing = housing.drop('ocean_proximity', axis=1)

# clean data via imputer; replacing missing values with median or mean
# this satisfies part (i) since total_bedrooms now is complete
simple_imputer = SimpleImputer(strategy='median')
```

```
housing_imputed = simple_imputer.fit_transform(housing)
housing = pd.DataFrame(housing_imputed, columns=housing.columns)

#  STICKING PART H HERE SO I CAN ADD THESE BEFORE SPLITTING DATASET
#  (h) Add three new attributes; (i) rooms per house = total rooms/households,
#     (ii) bedrooms ratio = total bedrooms/total rooms, and
#     (iii) people per house = population/households.

# assign housing DataFrame attributes to arrays
total_rooms = housing['total_rooms'].values
households = housing['households'].values
total_bedrooms = housing['total_bedrooms'].values
population = housing['population'].values

# calculate new attributes
rooms_per_house = total_rooms / households
bedrooms_ratio = total_bedrooms / total_rooms
people_per_house = population / households

# assign new attributes to housing DataFrame
# using .insert(0, ...) to stack each in front
housing.insert(0, 'rooms_per_house', rooms_per_house)
housing.insert(0, 'bedrooms_ratio', bedrooms_ratio)
housing.insert(0, 'people_per_house', people_per_house)

# standardize housing dataset
standard_scaler = StandardScaler()
housing_scaled = standard_scaler.fit_transform(housing)
housing = pd.DataFrame(housing_scaled, columns=housing.columns)
print('\nPREPROCESSED HOUSING DATA\n-------------------------')
print(housing)

# extract input features and output target
x = housing[[
    'people_per_house',
    'bedrooms_ratio',
    'rooms_per_house',
    'longitude',
    'latitude',
    'housing_median_age',
    'total_rooms',
    'total_bedrooms',
    'population',
    'households',
    'median_income'
    ]].values
y = housing['median_house_value'].values

# split into training and testing datasets
test_ratio = 0.2
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_ratio,
    random_state=42)

# (e) Because this data-set includes geographical information (latitude
#     and longitude), you are asked to create a scatterplot of all the
#     districts to visualize the geographical data in 2D space.

# get arrays containing lattitude and longitudes
lat = housing[['latitude']].values
```

```python
long = housing[['longitude']].values

# plot them as a scatterplot
plt.figure(figsize=(10,8))
plt.scatter(lat, long, marker='o', s=0.75, c='#32a852', alpha=0.95, label='2D Map')
plt.xlabel('Latitude')
plt.ylabel('Longitude')
plt.title(r'Normalized 2D Space Visualization of $housing.csv$')
plt.legend()
plt.gca().set_facecolor((0,0.1,0.8, 0.1))
#plt.savefig(plots_url + '2d_housing_scatter.png', dpi=300)
plt.close()


# (f) Compute the standard correlation coefficient between every pair of
#     attributes using the corr() method.
corr_matrix = housing.corr()
corr_to_median_house_val = corr_matrix['median_house_value'].sort_values(ascending=False)
print('\nCORRELATION of ______ TO median_house_value\n--------------------------------------------')
print(corr_to_median_house_val)


# (g)
pd.plotting.scatter_matrix(housing, alpha=0.2, figsize=(20,16))
plt.suptitle('Scatter Matrix for $housing.csv$')
plt.tight_layout()
# plt.savefig(plots_url + 'housing_scatter_matrix.png', dpi=80)
plt.close()


# (i)

# initalize linear model
linear_regressor = LinearRegression()

# train linear model
linear_regressor.fit(x_train, y_train)

# test linear model
y_train_predicted = linear_regressor.predict(x_train)
y_test_predicted = linear_regressor.predict(x_test)


# now test for different dataset sizes
train_sizes = np.linspace(0.1, 1.0, 10)  # Dataset sizes from 10% to 100% of the training set
train_errors = []
test_errors = []

for train_size in train_sizes:
    # Split data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

    # Use only a portion of the training set defined by `train_size`
    X_train_subset = X_train[:int(train_size * len(X_train))]
```

```
    y_train_subset = y_train[:int(train_size * len(y_train))]

    # Train the model
    lin_reg = LinearRegression()
    lin_reg.fit(X_train_subset, y_train_subset)

    # Make predictions
    y_train_pred = lin_reg.predict(X_train_subset)
    y_test_pred = lin_reg.predict(X_test)

    # Calculate the errors
    train_mse = mean_squared_error(y_train_subset, y_train_pred)
    test_mse = mean_squared_error(y_test, y_test_pred)

    # Store the errors
    train_errors.append(train_mse)
    test_errors.append(test_mse)

# PLOTTING RESULTS

plt.figure(figsize=(16,9))

# plot predicted against actual
ax1 = plt.subplot(1, 2, 1)
ax1.scatter(y_test, y_test_predicted, c='orange', marker='x', label='Learned Model', alpha=0.6)
ax1.plot([min(y_test), max(y_test)], [min(y_test_predicted), max(y_test_predicted)], c='blue', label='Ideal Model')
plt.xlabel('$\mathbf{y}_{test}$ Actual')
plt.ylabel('$\mathbf{y}_{test}$ Predicted')
plt.title('Learned Model Spread')
plt.legend()

## Plot the Learned Weights Coefficients of the model
#weights = linear_regressor.coef_
#feature_names = ['people_per_house', 'bedrooms_ratio', 'rooms_per_house','longitude',
#                 'latitude', 'housing_median_age', 'total_rooms', 'total_bedrooms',
#                 'population', 'households', 'median_income']
#
#ax2 = plt.subplot(1, 3, 2)
#ax2.barh(feature_names, weights)
#plt.xlabel("Model Weights $w_i$")
#plt.title("Linear Regression Feature Weights")

# Plot the learning curve
ax3 = plt.subplot(1, 2, 2)
ax3.plot(train_sizes, train_errors, label='Training Error', color='blue')
ax3.plot(train_sizes, test_errors, label='Test Error', color='green')
plt.title('Learning Curve')
plt.xlabel('Training Set Size (Ratio)')
plt.ylabel('Mean Squared Error')
plt.subplots_adjust(wspace=0.2)

plt.suptitle('Linear Model Results for $housing.csv$')
plt.legend()
plt.savefig(plots_url + "linear_model_results.png", dpi=300)
plt.close()
```

Question 2. *Classification in machine learning.*
Solution.

(A)

My best probabilistic classifier has 92.16% accuracy on the testing set, and my best non-probabilistic classifier has 97.12% on the testing set. Unfortunately, I was unable to achieve 97% accuracy for the probabilistic model on the testing set.

```
MNIST CLASSIFICATION REPORT
###########################
Dataset Size : 70000
Probabilistic Model Tolerance     : 0.01%
Probabilistic Test      RMSE      : 1.16
Probabilistic Test  Accuracy      : 92.16%
Probabilistic Test  F1 Score      : 0.92
Probabilistic Train     RMSE      : 1.07
Probabilistic Train Accuracy      : 93.27%
Probabilistic Train F1 Score      : 0.93
Non-probabilistic Test      RMSE  : 0.76
Non-probabilistic Test  Accuracy  : 97.12%
Non-probabilistic Test  F1 Score  : 0.97
Non-probabilistic Train     RMSE  : 0.52
Non-probabilistic Train Accuracy  : 98.58%
Non-probabilistic Train F1 Score  : 0.99
```

Figure 9: MNIST Classification Results

(B)

I wrote a function called *quad_direction_enricher* (lines 43-73) which takes the MNIST dataset and shifts it a user-defined amount of pixels in each of the cardinal directions. The function returns an enriched version of the input features and the corresponding output target, which can then be appended onto the original dataset to increase the amount of data available to train. However, when trained over each of the now 350,000 datapoints, the model drops in accuracy from 92.16% to 89.97%. The reason for this I suspect is overfitting, or a reduction in the model's generalization.

```
MNIST CLASSIFICATION REPORT
###########################
Dataset Size : 350000
Probabilistic Model Tolerance     : 0.01%
Probabilistic Test      RMSE      : 1.30
Probabilistic Test  Accuracy      : 89.97%
Probabilistic Test  F1 Score      : 0.90
Probabilistic Train     RMSE      : 1.28
Probabilistic Train Accuracy      : 90.29%
Probabilistic Train F1 Score      : 0.90
```

Figure 10: Expanded MNIST Classification Results

(C)

The non-probabilistic KNN classifier, using 3 neighbors, is in lines 130-155.

(D)

No matter what I did, the non-probabilistic KNN classifier always outperformed the probabilistic classifier; for my best model, the non-probabilistic classifier was 4.96% more accurate than the probabilistic classifier. In terms of computational complexity, the KNN classifier is much more computationally expensive and that is heavily dependent on how many neighbors you wish to use per datapoint, and the computational complexity exponentially increases for the increase in dataset size. For the 350,000 enriched dataset, my computer at 5000MHz could not finish the KNN classifier even after several minutes of waiting. So, there's a trade off. For critical applications that demand accuracy, using the non-probabilistic classifer is best, but for saving compute-power, it is best to choose the probabilistic model.

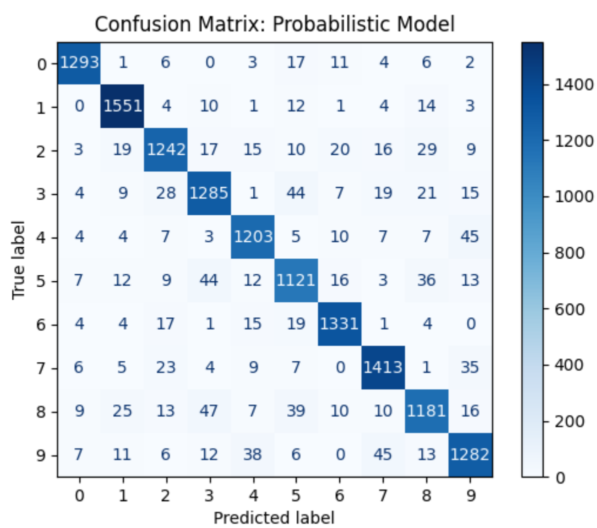

Figure 11: Probabilistic Classifier Confusion Matrix

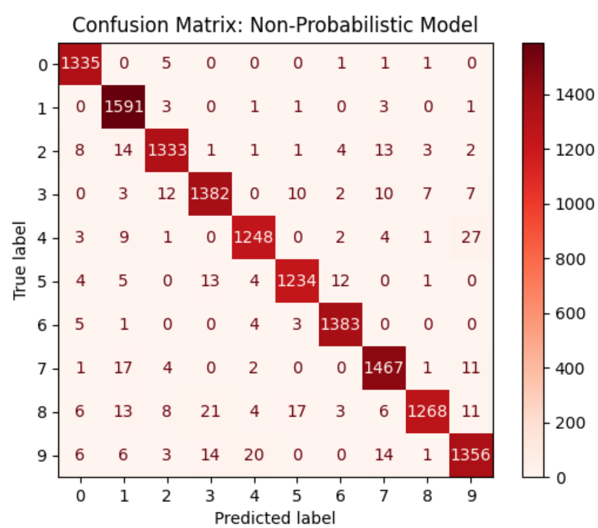

Figure 12: Non-probabilistic Classifier Confusion Matrix

Below is the python code used for Question 2:

Listing 2: File: q2.py

```python
"""
Southern Illinois University Carbondale
Department of Electrical Engineering
--------------------------------------
ECE469: Intro to Machine Learning
Midterm Exam: Question 2
Chase Lotito

10/15/2024

"MNIST"
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import root_mean_squared_error, accuracy_score, f1_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from scipy.ndimage import shift

from sklearn.decomposition import PCA

# fetch the MNIST dataset
mnist = fetch_openml('mnist_784', as_frame=False)

# extract input features and output target
X = mnist['data']
y = mnist['target']

# the target values in y are strings, so we must
# first convert them to integers
y = y.astype(int)

# (b) Write a function that can shift an MNIST image
#     in any direction. Do this in all directions for
#     the training set, and append them to it.
def quad_direction_enricher(X: np.array, y: np.array, size: int, px: int):
    """
    To enrich an image dataset with 4 sets of
    the original set shifted in all directions px.
    (up, down, left, right)
    """
    enriched_X = []
    y_temp = []
    for k in range(0, 4, 1):
        for i in range(0, size, 1):
            img = X[i].reshape(28,28)  # make image 28x28 matrix

            # image shifting logic
            if (k == 0):
```

```
                img = shift(img, [px, 0])  # shift up
            elif (k == 1):
                img = shift(img, [-px, 0]) # shift down
            elif (k == 2):
                img = shift(img, [0, px])  # shift right
            elif (k == 3):
                img = shift(img, [0, -px]) # shift left
            else:
                print('ERROR: image shift bounds error')

            img = img.flatten()      # make image 1D array
            enriched_X.append(img)
            y_temp.append(y[i])

    #enriched_X = np.array([img.flatten() for img in temp])
    enriched_Y = np.array(y_temp)
    return np.array(enriched_X), enriched_Y

# enrich X and y
e_X, e_y = quad_direction_enricher(X, y, len(X), px=1)

X = np.concatenate([X, e_X])
y = np.concatenate([y, e_y])


# scale down size of dataset
factor = 1
subset_size = 70000*factor
X = X[:subset_size]
y = y[:subset_size]

# split dataset into training and testing sets
test_ratio = 0.2
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=test_ratio, random_state=42)

# next we want to standardize our data, but not
# targets, as to preserve y\in(0,9)
minmax = MinMaxScaler()
X_train = minmax.fit_transform(X_train)
X_test = minmax.transform(X_test)

# train logisitic regression classifier

classify = LogisticRegression(multi_class='multinomial', solver='lbfgs', C=0.1, max_iter=1000, random_state=42)
classify.fit(X_train, y_train)

# predict using model
y_test_pred = classify.predict(X_test)
y_train_pred = classify.predict(X_train)

# calc MSE
test_rmse = root_mean_squared_error(y_test, y_test_pred)
train_rmse = root_mean_squared_error(y_train, y_train_pred)

# determine model accuracy
test_accuracy = accuracy_score(y_test, y_test_pred) * 100
train_accuracy = accuracy_score(y_train, y_train_pred) * 100
```

```python

# determine model f1 score
test_f1 = f1_score(y_test, y_test_pred, average='macro')
train_f1 = f1_score(y_train, y_train_pred, average='macro')

print('MNIST CLASSIFICATION REPORT')
print('###########################')
print(f'Dataset Size : {subset_size}')
print(f'Probabilistic Model Tolerance     : {0.01}%')
print(f'Probabilistic Test      RMSE      : {test_rmse:.2f}')
print(f'Probabilistic Test  Accuracy      : {test_accuracy:.2f}%')
print(f'Probabilistic Test  F1 Score      : {test_f1:.2f}')
print(f'Probabilistic Train     RMSE      : {train_rmse:.2f}')
print(f'Probabilistic Train Accuracy      : {train_accuracy:.2f}%')
print(f'Probabilistic Train F1 Score      : {train_f1:.2f}')

# (c) KNN-based algorithms belong to the class of non-probabilistic classifiers.
#     You are asked to design a KNN-based classifier to classify handwritten
#     digits (0-9) in MNIST data-set.

# initialize the knn classifier
n=3
knn = KNeighborsClassifier(n_neighbors=n, metric='minkowski')

# train knn classifier
knn.fit(X_train, y_train)

# predict using knn classifier
y_test_pred_knn = knn.predict(X_test)
y_train_pred_knn = knn.predict(X_train)

# calc MSE
test_rmse_knn = root_mean_squared_error(y_test, y_test_pred_knn)
train_rmse_knn = root_mean_squared_error(y_train, y_train_pred_knn)

# determine model accuracy
test_accuracy_knn = accuracy_score(y_test, y_test_pred_knn) * 100
train_accuracy_knn = accuracy_score(y_train, y_train_pred_knn) * 100

# determine model f1 score
test_f1_knn = f1_score(y_test, y_test_pred_knn, average='macro')
train_f1_knn = f1_score(y_train, y_train_pred_knn, average='macro')

# add KNN model results to finish classification report
print(f'Non-probabilistic Test      RMSE  : {test_rmse_knn:.2f}')
print(f'Non-probabilistic Test  Accuracy  : {test_accuracy_knn:.2f}%')
print(f'Non-probabilistic Test  F1 Score  : {test_f1_knn:.2f}')
print(f'Non-probabilistic Train     RMSE  : {train_rmse_knn:.2f}')
print(f'Non-probabilistic Train Accuracy  : {train_accuracy_knn:.2f}%')
print(f'Non-probabilistic Train F1 Score  : {train_f1_knn:.2f}')


# (D) Confusion Matrix
prob_cmatrix = confusion_matrix(y_pred=y_test_pred, y_true=y_test)
ConfusionMatrixDisplay(confusion_matrix=prob_cmatrix).plot(cmap='Blues')
plt.title('Confusion Matrix: Probabilistic Model')
plt.show()
plt.close()

```

```
nonprob_cmatrix = confusion_matrix(y_pred=y_test_pred_knn, y_true=y_test)
ConfusionMatrixDisplay(confusion_matrix=nonprob_cmatrix).plot(cmap='Reds')
plt.title('Confusion Matrix: Non-Probabilistic Model')
plt.show()
plt.close()
```