VERDICKT_JULIA-PS3

February 17, 2024

1 Problem Set 3

1.0.1 Before You Start

This problem set is fun but challenging. It's going to involve a good amount of debugging and head-scratching, so try to start sooner rather than later!

This problem set has three parts:

- **Part I**: Experimental Setup
- **Part II**: Nearest Neighbor and Cross-Validation
- **Part III**: Overfitting in Model Selection and Nested Cross Validation

For part I and II we'll consider a regression problem. You should *not* be using any built-in ML libraries for nearest neighbors, distance metrics, or cross-validation – your mission is to write those algorithms in Python! For these two first parts we will be working with a modified version of the California Housing Dataset that you can download from bcourses (`cal_housing_data_clean.csv`). Part I will be relatively easy; Part II will take more time.

For part III we'll consider a classification problem. You'll be able to use Python ML built-in libraries (in particular scikit-learn). We'll not be using the California Housing Dataset but rather synthetic data that you'll generate yourself.

Make sure the following libraries load correctly before starting (hit Ctrl-Enter).

[ ]:
```
import IPython
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib
import sklearn
import time
from IPython.display import Image
import dataframe_image as dfi
import random
```

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

1.1 Introduction to the assignment

For this assignment, you will be using a version of the California Housing Prices Dataset with additional information. Use the following commands to load the information in the csv file provided with the assignment in bcourses (`cal_housing_data_clean.csv`). Take some time to explore the data.

```
[ ]: # load Cal data set
cal_df = pd.read_csv('cal_housing_data_clean.csv')
features =␣
 ↪['MedInc','HouseAge','AveRooms','AveBedrms','Population','DistCoast','Inland']
target = 'MedHouseVal'
```

---

2 Part I: Experimental Setup

The goal of the next few sections is to design an experiment to predict the median home value for census block groups. Before beginning the “real” work, refamiliarize yourself with the dataset.

2.0.1 1.1 Begin by writing a function to compute the Root Mean Squared Error for a list of numbers

You can find the sqrt function in the Numpy package. Furthermore the details of RMSE can be found on Wikipedia. Do not use a built-in function to compute RMSE, other than numpy functions like `sqrt` and if needed, `sum` or other relevant ones.

```
[ ]: """
Function
--------
compute_rmse

Given two arrays, one of actual values and one of predicted values,
compute the Root Mean Squared Error

Parameters
----------
y_hat : array
    numpy array of numerical values corresponding to predictions for each of␣
 ↪the N observations

y : array
    numpy array of numerical values corresponding to the actual values for each␣
 ↪of the N observations

Returns
-------
rmse : int
```

```
    Root Mean Squared Error of the prediction

Example
--------
>>> print(compute_rmse((4,6,3),(2,1,4)))
3.16
"""
def compute_rmse(y_hat, y):
    # your code here
    n = len(y)
    rmse = np.sqrt((1/n)*np.sum((y-y_hat)**2))
    return rmse
print(compute_rmse(np.array([4,6,3]),np.array([2,1,4])))
```

```
3.1622776601683795
```

2.0.2 1.2 Divide your data into training and testing datasets

Randomly select 75% of the data and put this in a training dataset (call this “cal_df_train”), and place the remaining 25% in a testing dataset (call this “cal_df_test”). Do not use built-in functions.

To perform any randomized operation, only use functions in the *numpy library (np.random)*. Do not use other packages for random functions.

[ ]:
```
# leave the following line untouched, it will help ensure that your "random"␣
 ↪split is the same "random" split used by the rest of the class
np.random.seed(seed=1948)

# your code here
train_percent = .75
train_number = int(train_percent*len(cal_df))
test_number = len(cal_df) - train_number
print('Total examples: %i' % len(cal_df))
print('Number of training examples: %i' % train_number)
print('Number of testing examples: %i' % test_number)

### SHUFFLE DATAFRAME
ids = np.arange(0, len(cal_df), 1)
np.random.shuffle(ids)
cal_df_shuffled = cal_df.iloc[ids]

### COMPLETE: SPLIT INTO TRAIN AND TEST
cal_df_train = cal_df_shuffled.iloc[:train_number].reset_index(drop=True)
cal_df_test = cal_df_shuffled.iloc[train_number:].reset_index(drop=True)
```

```
Total examples: 20640
Number of training examples: 15480
Number of testing examples: 5160
```

2.0.3 1.3 Use a baseline for prediction, and compute RMSE

Let's start by creating a very bad baseline model that predicts median house values as the average of `MedHouseVal`.

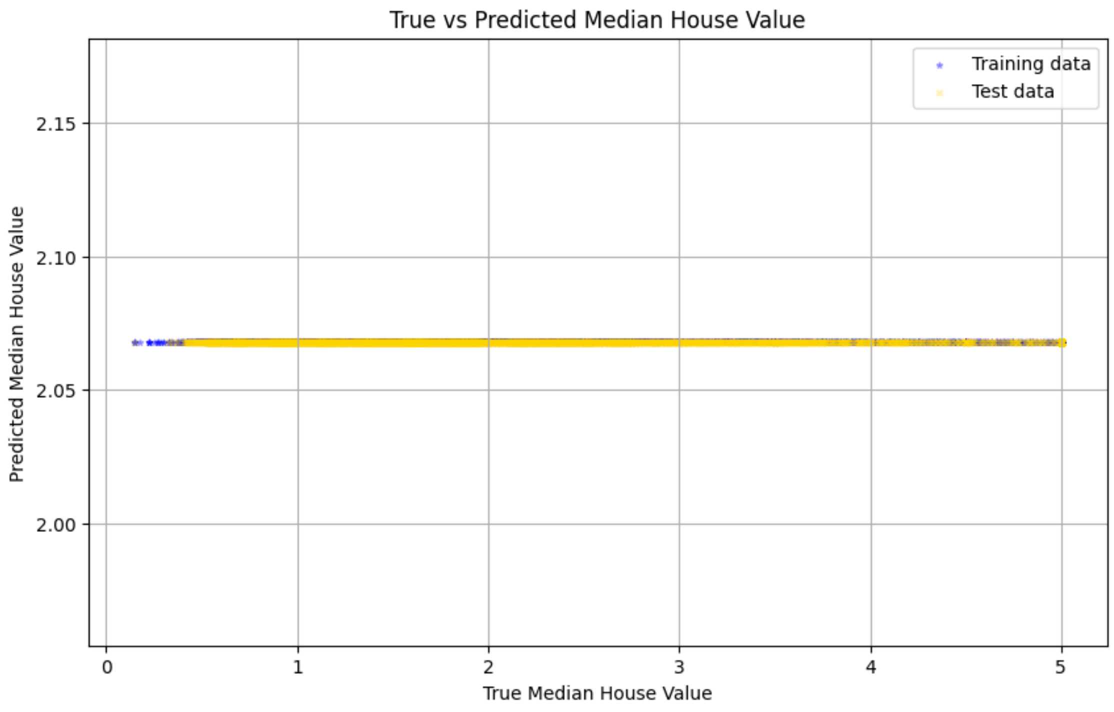
Specifically, create a model that predicts, for every observation X_i, the median home value as the average of the median home values of block groups in the **training set**.

Once the model is built, do the following:

1. Report the RMSE of the training set and report it.
2. Report the RMSE of the test data set (but use the model you trained on the training set!).
3. How does RMSE compare for training vs. testing datasets? Is this what you expected, and why?
4. Add code to your function to measure the running time of your algorithm. How long does it take to compute the predicted values for the test data?
5. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in blue and the test instances in gold. Make sure to label your axes appropriately, and add a legend to your figure.

[ ]:
```python
# your code here

#STEP 1
avg_MHV_train = cal_df_train['MedHouseVal'].mean()
cal_df_train['predicted'] = avg_MHV_train
rmse_train = compute_rmse(cal_df_train['predicted'],
                          cal_df_train['MedHouseVal'])
print(f"training RMSE: {rmse_train}")

#STEP2
t0 = time.time()
cal_df_test['predicted'] = avg_MHV_train
t1 = time.time()
print(f"prediction runtime for test-set: {t1-t0}")
rmse_test = compute_rmse(cal_df_test['predicted'],
                         cal_df_test['MedHouseVal'])
print(f"testing RMSE: {rmse_test}")


plt.figure(figsize=(10, 6))
plt.scatter(cal_df_train['MedHouseVal'],
            cal_df_train['predicted'],
            color='blue', alpha=0.3,
            label='Training data', marker = '*', s = 10)
plt.scatter(cal_df_test['MedHouseVal'],
            cal_df_test['predicted'],
            color='gold', alpha=0.3,
            label='Test data', marker = 'x', s = 10)
plt.xlabel('True Median House Value')
plt.ylabel('Predicted Median House Value')
plt.title('True vs Predicted Median House Value')
plt.legend()
plt.grid(True)
plt.show()
```

```
training RMSE: 1.1560706527361215
prediction runtime for test-set: 0.0005612373352050781
testing RMSE: 1.1474775264840165
```



your answer here 1. Report the RMSE of the training set and report it

The training RMSE is about 1.1560706527361215.

2. Report the RMSE of the test data set (but use the model you trained on the training set!).

The testing RMSE is about 1.1474775264840165

3. How does RMSE compare for training vs. testing datasets? Is this what you expected, and why?

Interestingly, the testing RMSE is lower than the training RMSE. At first, this might appear unexpected since the model was not originally optimized on the testing data, so it seems like it should be designed to fare better with the training data. However, such a general model is not prone to over-fitting, which is why the RMSE might be

lower for the testing data.

4. Add code to your function to measure the running time of your algorithm. How long does it take to compute the predicted values for the test data?

It took about 0.001171112060546875 seconds to compute the predicted values for the test data

5. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in blue and the test instances in gold. Make sure to label your axes appropriately, and add a legend to your figure.

See above.

2.0.4 1.4 Use another baseline for prediction, and compute RMSE [extra-credit]

Now consider a baseline model that predicts median house values as the averages of `MedHouseVal` based on whether the census block is adjacent to the coast or inland (note that the `Inland` feature is already computed and ready for you).

Specifically, create a model that predicts, for every observation X_i, the median home value as the average of the median home values of block groups in the **training set** that have the same adjacency value.

For example, for an input observation where `Inland==1`, the model should predict the `MedHouseVal` as the average of all `MedHouseVal` values in the training set that also have `Inland==1`.

Once the model is built, do the following:

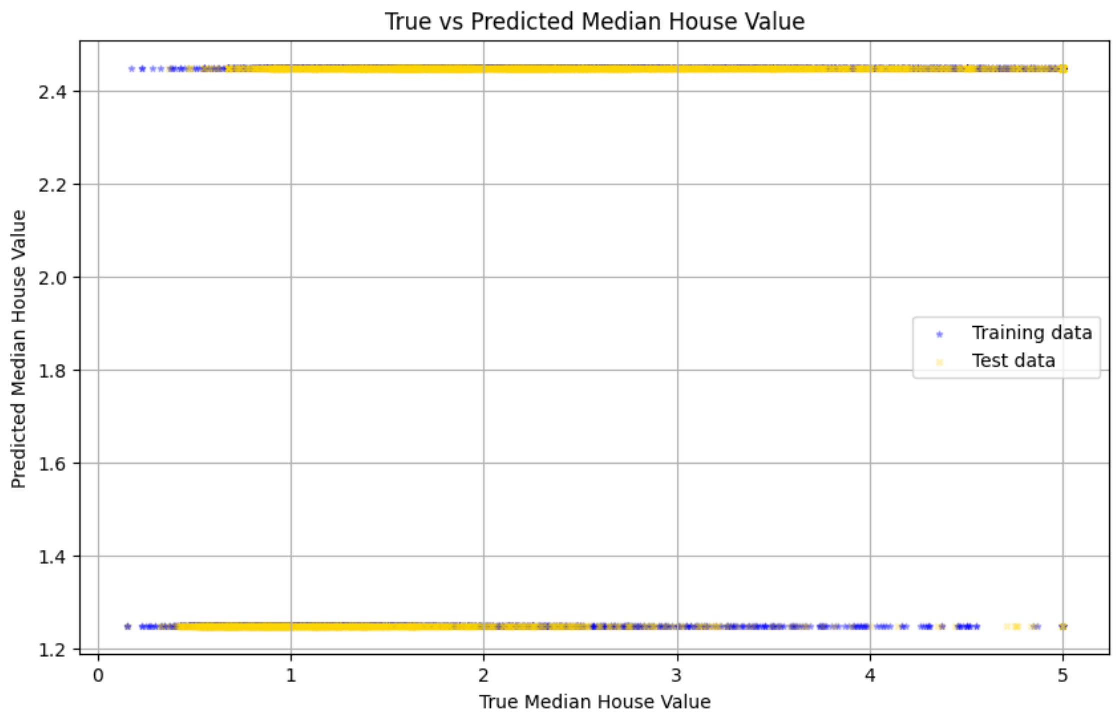
1. Compute the RMSE of the training set.
2. Now compute the RMSE of the test data set (but use the model you trained on the training set!).
3. How does RMSE compare for training vs. testing datasets? Is this what you expected, and why?
4. Add code to your function to measure the running time of your algorithm. How long does it take to compute the predicted values for the test data?
5. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in blue and the test instances in gold. Make sure to label your axes appropriately, and add a legend to your figure to make clear which dots are which.
6. Compare this results to those obtained in 1.3. Is coast adjacency improving the predictions?

Note: The `groupby` operation might come handy here.

```
[ ]: # your code here
inland_0 = cal_df_train.groupby('Inland')['MedHouseVal'].mean()[0]
inland_1 = cal_df_train.groupby('Inland')['MedHouseVal'].mean()[1]
cal_df_train['predicted2'] = np.where(cal_df_train['Inland'] == 0, inland_0,␣
 ↪inland_1)
rmse_train2 = compute_rmse(cal_df_train['predicted2'],
                           cal_df_train['MedHouseVal'])
print(f"training RMSE: {rmse_train2}")
```

```python
#STEP2
t0 = time.time()
cal_df_test['predicted2'] = np.where(cal_df_test['Inland'] == 0, inland_0,␣
 ↪inland_1)
t1 = time.time()
print(f"prediction runtime for test-set: {t1-t0}")
rmse_test2 = compute_rmse(cal_df_test['predicted2'],
                          cal_df_test['MedHouseVal'])
print(f"testing RMSE: {rmse_test2}")


plt.figure(figsize=(10, 6))
plt.scatter(cal_df_train['MedHouseVal'],
            cal_df_train['predicted2'],
            color='blue', alpha=0.3,
            label='Training data', marker = '*', s = 10)
plt.scatter(cal_df_test['MedHouseVal'],
            cal_df_test['predicted2'],
            color='gold', alpha=0.3,
            label='Test data', marker = 'x', s = 10)
plt.xlabel('True Median House Value')
plt.ylabel('Predicted Median House Value')
plt.title('True vs Predicted Median House Value')
plt.legend()
plt.grid(True)
plt.show()
```

```
training RMSE: 1.0119036022521923
prediction runtime for test-set: 0.001043081283569336
testing RMSE: 1.0011137151179164
```



your answer here 1. Compute the RMSE of the training set.

The RMSE of the training set is about 1.0119036022521923.

2. Now compute the RMSE of the test data set (but use the model you trained on the training set!).

The RMSE of the test data set is about 1.0011137151179164.

3. How does RMSE compare for training vs. testing datasets? Is this what you expected, and why?

Once again testing RMSE is lower than training RMSE, even though this model is more restrictive. This is likely because the model is pretty general overall, so there's still not ask big of a risk of overfitting.

4. Add code to your function to measure the running time of your algorithm. How long does it take to compute the predicted values for the test data?

It takes about 0.0011320114135742188 seconds to compute the predicted values for the test data.

5. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. Color the training instances in blue and the test instances in gold. Make sure to label your axes appropriately, and add a legend to your figure to make clear which dots are which.

See above.

6. Compare this results to those obtained in 1.3. Is coast adjacency improving the predictions?

Technically, yes. Using RMSE as a metric, we see that the RMSE decreases when `Inland` is considered. This implies that accounting for it in the model improves prediction accuracy overall. This is the bias-variance tradeoff. A more general model minimizes the variance aspect as RMSE doesn't shoot up when the model is generalized to new data. However, the overall RMSE implies the bias is higher that what we might see for a more complex model.

---

3 Part II: Nearest Neighbors and Cross-Validation

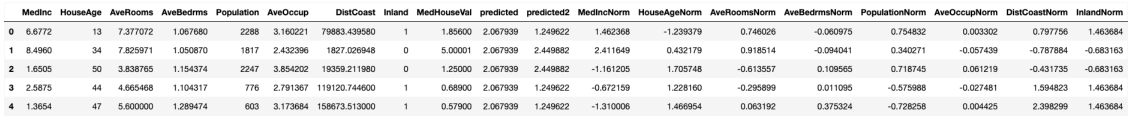
Let's try and build a machine learning algorithm to beat the "Average Values" baselines that you computed above. Your next task is to implement a basic nearest neighbor algorithm from scratch.

3.0.1 2.1 Nearest Neighbors: Normalization

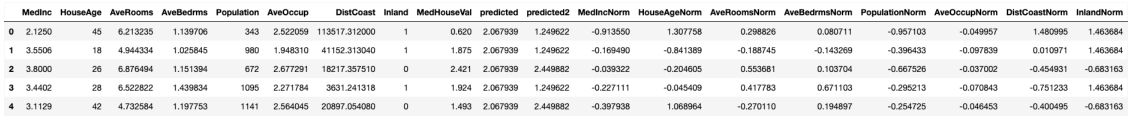
Create normalized analogues of all the features in both the training and test datasets. Recall that this involves substracting the **training** mean and dividing by the **training** standard deviation.

Include the normalized features as additional columns in the train an test dataframes and call them `MedIncNorm`, `HouseAgeNorm`, `AveRoomsNorm`, `AveBedrmsNorm`, `PopulationNorm`, `AveOccupNorm`, `DistCoastNorm and InlandNorm` respectively.

```
[ ]: # your code here

columns_to_normalize = ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms',␣
 ↪'Population', 'AveOccup', 'DistCoast', 'Inland']

# Normalizing the columns
for column in columns_to_normalize:
    # Calculate the mean and std from the training set
    mean = cal_df_train[column].mean()
    std = cal_df_train[column].std()

    # Normalize the training data
    cal_df_train[f'{column}Norm'] = (cal_df_train[column] - mean) / std

    # Normalize the testing data using the training mean and std
    cal_df_test[f'{column}Norm'] = (cal_df_test[column] - mean) / std

# Display the first few rows of the updated training and testing DataFrames to␣
 ↪verify
dfi.export(cal_df_train.head(), 'cal_df_trainhead.jpeg')
display(Image('cal_df_trainhead.jpeg'))
dfi.export(cal_df_test.head(), 'cal_df_testhead.jpeg')
display(Image('cal_df_testhead.jpeg'))
```



	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	DistCoast	Inland	MedHouseVal	predicted	predicted2	MedIncNorm	HouseAgeNorm	AveRoomsNorm	AveBedrmsNorm	PopulationNorm	AveOccupNorm	DistCoastNorm	InlandNorm
0	6.6772	13	7.377072	1.067680	2288	3.160221	79883.439580	1	1.85600	2.067939	1.249622	1.462368	-1.239379	0.746026	-0.060975	0.754832	0.003302	0.797756	1.463684
1	8.4960	34	7.825971	1.050870	1817	2.432396	1827.026948	0	5.00001	2.067939	2.449882	2.411649	0.432179	0.918514	-0.094041	0.340271	-0.057439	-0.787884	-0.683163
2	1.6505	50	3.838765	1.154374	2247	3.854202	19359.211980	0	1.25000	2.067939	2.449882	-1.161205	1.705748	-0.613557	0.109565	0.718745	0.061219	-0.431735	-0.683163
3	2.5875	44	4.665468	1.104317	776	2.791367	119120.744600	1	0.68900	2.067939	1.249622	-0.672159	1.228160	-0.295899	0.011095	-0.575988	-0.027481	1.594823	1.463684
4	1.3654	47	5.600000	1.289474	603	3.173684	158673.513000	1	0.57900	2.067939	1.249622	-1.310006	1.466954	0.063192	0.375324	-0.728258	0.004425	2.398299	1.463684



	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	DistCoast	Inland	MedHouseVal	predicted	predicted2	MedIncNorm	HouseAgeNorm	AveRoomsNorm	AveBedrmsNorm	PopulationNorm	AveOccupNorm	DistCoastNorm	InlandNorm
0	2.1250	45	6.213235	1.139706	343	2.522059	113517.312000	1	0.620	2.067939	1.249622	-0.913550	1.307758	0.298826	0.080711	-0.957103	-0.049957	1.480995	1.463684
1	3.5506	18	4.944334	1.025845	980	1.948310	41152.313040	1	1.875	2.067939	1.249622	-0.169490	-0.841389	-0.188745	-0.143269	-0.396433	-0.097839	0.010971	1.463684
2	3.8000	26	6.876494	1.151394	672	2.677291	18217.357510	0	2.421	2.067939	2.449882	-0.039322	-0.204605	0.553681	0.103704	-0.667526	-0.037002	-0.454931	-0.683163
3	3.4402	28	6.522822	1.439834	1095	2.271784	3631.241318	1	1.924	2.067939	1.249622	-0.227111	-0.045409	0.417783	0.671103	-0.295213	-0.070843	-0.751233	1.463684
4	3.1129	42	4.732584	1.197753	1141	2.564045	20897.054080	0	1.493	2.067939	2.449882	-0.397938	1.068964	-0.270110	0.194897	-0.254725	-0.046453	-0.400495	-0.683163

3.0.2 2.2 Basic Nearest Neighbor algorithm

Use your training data to "fit" your model that predicts `MedHouseVal` from `MedIncNorm`, `HouseAgeNorm` and `AveRoomsNorm`, although as you know, with Nearest Neighbors there is no real training, you just need to keep your training data in memory. Write a function that predicts the median home value using the nearest neighbor algorithm we discussed in class. Since this is a small dataset, you can simply compare your test instance to every instance in the training set, and return the `MedHouseVal` value of the closest training instance. Have your function take L as an input, where L is an integer >= 1 representing the norm choice. Use the Euclidean distance (L=2) for all questions henceforth unless explicitly stated otherwise.

Make sure to do the following - 1. Use your algorithm to predict the median home value of every instance in the test set. Report the RMSE ("test RMSE") 2. Use your algorithm to predict the median home value of every instance in the training set and report the training RMSE. 3. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. 4. Report an estimate of the total time taken by your code to predict the nearest neighbors for all the values in the test data set. 5. How does the performance (test RMSE and total runtime) of your nearest neighbors algorithm compare to the baseline in part 1.4? Explain the

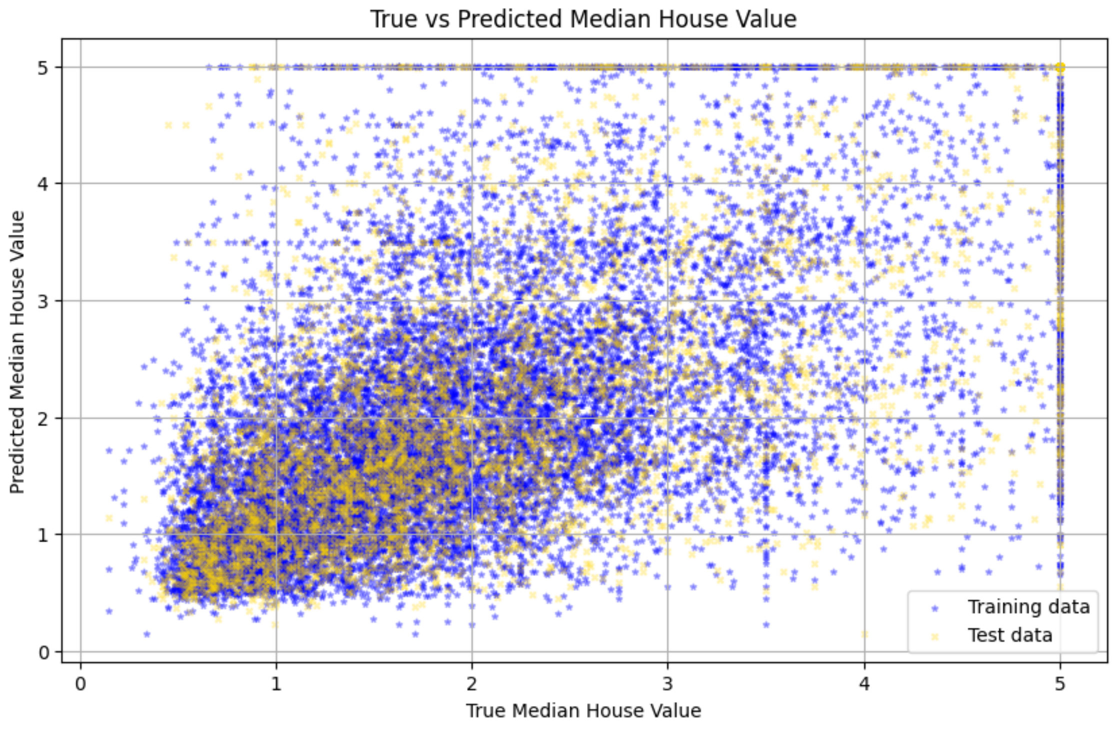
Note: Runtime should not exceed a couple of minutes. If its taking longer then we strongly suggest you go back to your code and make it more efficient.

```
[ ]: # your code here
     def basicnn(X_train, X_test, y_train, y_test, L=2, training = 'yes'):
         # simplying code by making it generalizable to both training and testing␣
      ↪cases
         X_test_np = X_test if training == 'no' else X_train
         y_test_np = y_test if training == 'no' else y_train

         # Preallocate memory for predictions
         predictions = np.empty(X_test_np.shape[0])

         # Compute distances and predictions
         for i, test_row in enumerate(X_test_np):
             distances = np.linalg.norm(X_train - test_row, ord=L, axis=1)
```

```
        if training == 'yes':
            # Ignore self in nearest neighbor calculation for training
            distances[i] = np.inf
        nearest_index = np.argmin(distances)
        predictions[i] = y_train[nearest_index]
    # Compute RMSE
    rmse = compute_rmse(predictions, y_test_np)
    return predictions, rmse

#Saving new dataframes with only the norm columns.
# I will use these going forward

X_train_normf = cal_df_train.iloc[:,11:]
X_test_normf = cal_df_test.iloc[:,11:]
y_train = cal_df_train['MedHouseVal']
y_test = cal_df_test['MedHouseVal']


cal_df_trainnorm = X_train_normf.copy()
cal_df_testnorm = X_test_normf.copy()
cal_df_trainnorm['MedHouseVal'] = y_train
cal_df_testnorm['MedHouseVal'] = y_test

#I use the pandas dataframes for easy column indexing, but using numpy in the␣
 ↪actual
#algorithms are so much faster, so I convert them first.
X_train22 = X_train_normf[['MedIncNorm', 'HouseAgeNorm','AveRoomsNorm']].
 ↪to_numpy()
X_test22 = X_test_normf[['MedIncNorm', 'HouseAgeNorm','AveRoomsNorm']].
 ↪to_numpy()


#STEP 1
cal_df_trainnorm['NN_predtrain1'], train_rmse = basicnn(X_train = X_train22,
                                                        X_test = X_train22,
                                                        y_train = y_train,
                                                        y_test = y_test)
print(f"This is the the training RMSE: {train_rmse}")

#STEP 2
t0 = time.time()
cal_df_testnorm['NN_predtest1'], test_rmse = basicnn(X_train = X_train22,
                                                  X_test =  X_test22,
                                                  y_train = y_train,
                                                  y_test = y_test,
                                                  training = 'no')
t1 = time.time()
```

```python
print(f"Prediction Time: {t1-t0} seconds")
print(f"This is the the testing RMSE: {test_rmse}")

#STEP 3
plt.figure(figsize=(10, 6))
plt.scatter(cal_df_trainnorm['MedHouseVal'],
            cal_df_trainnorm['NN_predtrain1'],
            color='blue', alpha=0.3,
            label='Training data', marker = '*', s = 10)
plt.scatter(cal_df_testnorm['MedHouseVal'],
            cal_df_testnorm['NN_predtest1'],
            color='gold', alpha=0.3,
            label='Test data', marker = 'x', s = 10)
plt.xlabel('True Median House Value')
plt.ylabel('Predicted Median House Value')
plt.title('True vs Predicted Median House Value')
plt.legend()
plt.grid(True)
plt.show()
```

```
This is the the training RMSE: 1.0227369569530085
Prediction Time: 0.7995908260345459 seconds
This is the the testing RMSE: 1.0350385789713217
```



your answer here 1. Use your algorithm to predict the median home value of every instance in the test set. Report the RMSE ("test RMSE")

This is the testing RMSE: 1.0350385789713217.

2. Use your algorithm to predict the median home value of every instance in the training set and report the training RMSE.

This is the training RMSE: 1.0227369569530085.

3. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis.

See above

4. Report an estimate of the total time taken by your code to predict the nearest neighbors for all the values in the test data set.

For the testing data set, the prediction time is about 0.8260979652404785 seconds.

5. How does the performance (test RMSE and total runtime) of your nearest neighbors algorithm compare to the baseline in part 1.4?

The the total run time is proportionally much larger to the runtime in part 1.4 and part 1.3. The basic NN algorithm is obviously more computationally expensive. Interestingly, the RMSE for both the testing and training sets are higher than for the baseline model in 1.4 but lower than the RMSE values for the baseline in 1.3. These metrics imply the basic NN algorithm performed worse than the baseline in 1.4 as it took longer and resulted in higher RMSE. However, compared to the baseline in 1.3, the basic NN algorithm does have a lower RMSE and performs better in that regard.

3.0.3 2.3 Optimization

Try to increase the performance of your nearest neighbor algorithm by adding features that you think might be relevant, and by using different values of L in the distance function. Try a model that uses a different set of 2 features, then try at least one model that uses more than 4 features, then try using a different value of L. If you're having fun, try a few different combinations of features and L! Use the test set to report the RMSE values.

What combination of features and distance function provide the lowest RMSE on the test set? Do your decisions affect the running time of the algorithm?

Note: For this and all subsequent questions, you should use normalized features.

```
[ ]: # your code here

     #STEP 1: Suggested Exploration
     t0 = time.time()
     _, test_rmse = basicnn(X_train = X_train_normf[['AveBedrmsNorm',␣
      ↪'PopulationNorm']].to_numpy(),
                                                         X_test =␣
      ↪X_test_normf[['AveBedrmsNorm', 'PopulationNorm']].to_numpy(),
                                                         y_train = y_train,
                                                  y_test = y_test,
                                                  training = 'no')
t1 = time.time()
print(f'''Suggested Model Case 1:
      L: 2
      Features: 'AveBedrmsNorm', 'PopulationNorm'
      Prediction Time: {t1-t0} seconds
      Testing RMSE: {test_rmse}''')

t0 = time.time()
_, test_rmse = basicnn(X_train = X_train_normf[['AveBedrmsNorm',
                                                'PopulationNorm',
                                                'DistCoastNorm',
                                                'InlandNorm',
                                                'HouseAgeNorm',
                                                'AveRoomsNorm']].to_numpy(),
                       X_test = X_test_normf[['AveBedrmsNorm',
                                              'PopulationNorm',
                                              'DistCoastNorm',
                                              'InlandNorm',
                                                'HouseAgeNorm',
                                                'AveRoomsNorm']].to_numpy(),
                       y_train = y_train,
                       y_test = y_test,
                       training = 'no')
t1 = time.time()
print(f'''Suggested Model Case 2:
      L: 2
      Features: 'AveBedrmsNorm', 'PopulationNorm', 'DistCoastNorm',␣
 ↪'InlandNorm'
      Prediction Time: {t1-t0} seconds
      Testing RMSE: {test_rmse}''')

t0 = time.time()
_, test_rmse = basicnn(X_train = X_train_normf[['AveBedrmsNorm',
                                                'PopulationNorm',
                                                'DistCoastNorm',
                                                'InlandNorm',
                                                'HouseAgeNorm',
                                                'AveRoomsNorm']].to_numpy(),
                       X_test = X_test_normf[['AveBedrmsNorm',
                                              'PopulationNorm',
                                              'DistCoastNorm',
                                              'InlandNorm',
                                                'HouseAgeNorm',
                                                'AveRoomsNorm']].to_numpy(),
                       y_train = y_train,
                        y_test = y_test, L = 3,
                        training = 'no')
t1 = time.time()
print(f'''Suggested Model Case 3:
      L: 3
      Features: 'AveBedrmsNorm', 'PopulationNorm', 'DistCoastNorm',␣
 ↪'InlandNorm'
      Prediction Time: {t1-t0} seconds
      Testing RMSE: {test_rmse}''')

print('========================================================================')

#STEP 2: Self Exploration, Random
columns_num = list(range(1, 8))
l_vals = list(range(5))

for i in range(5):
    feature_num = random.sample(columns_num, k = 1)[0]
    l_val = random.sample(l_vals, k = 1)[0]
    X_train_temp = X_train_normf.sample(n = feature_num, axis = 1)
    X_test_temp = X_test_normf.sample(n = feature_num, axis = 1)
    X_train_tempnp = X_train_temp.to_numpy()
    X_test_tempnp = X_test_temp.to_numpy()
    t0 = time.time()
    _, test_rmse = basicnn(X_train =X_train_tempnp,
                        X_test = X_test_tempnp,
                        y_train = cal_df_trainnorm['MedHouseVal'],
                        y_test = cal_df_testnorm['MedHouseVal'], L = l_val,
                        training = 'no')
    t1 = time.time()
    print(f'''Self Exploration, Random Case {i+1}:
          L: {l_val}
          Features: {X_train_temp.columns}
          Prediction Time: {t1-t0} seconds
          Testing RMSE: {test_rmse}''')
print('========================================================================')

#STEP 3: Self Exploration, All Features, Sequential L
X_train_normfnp =  X_train_normf.to_numpy()
X_test_normfnp =  X_test_normf.to_numpy()
for i in range(1,4):
      t0 = time.time()
      _, test_rmse = basicnn(X_train = X_train_normfnp,
                              X_test = X_test_normfnp,
                              y_train = cal_df_trainnorm['MedHouseVal'],
                              y_test = cal_df_testnorm['MedHouseVal'], L = i,
                              training = 'no')
      t1 = time.time()
      print(f'''Self Exploration,  All Features, Sequential L Case {i}:
            L: {i}
            Testing RMSE: {test_rmse}
            Prediction Time: {t1-t0} seconds''')
```

```
Suggested Model Case 1:
      L: 2
      Features: 'AveBedrmsNorm', 'PopulationNorm'
      Prediction Time: 0.939460039138794 seconds
      Testing RMSE: 1.5909217441351218
Suggested Model Case 2:
      L: 2
      Features: 'AveBedrmsNorm', 'PopulationNorm', 'DistCoastNorm', 'InlandNorm'
      Prediction Time: 3.298137903213501 seconds
      Testing RMSE: 1.0940797279392114
Suggested Model Case 3:
      L: 3
      Features: 'AveBedrmsNorm', 'PopulationNorm', 'DistCoastNorm', 'InlandNorm'
      Prediction Time: 20.984900951385498 seconds
      Testing RMSE: 1.0964697395718512
=======================================================================
Self Exploration, Random Case 1:
          L: 0
          Features: Index(['AveRoomsNorm', 'InlandNorm', 'AveBedrmsNorm'],
dtype='object')
          Prediction Time: 1.065901756286621 seconds
          Testing RMSE: 1.1959567854772957
Self Exploration, Random Case 2:
          L: 4
          Features: Index(['AveRoomsNorm', 'InlandNorm', 'HouseAgeNorm',
'MedIncNorm',
       'AveBedrmsNorm', 'AveOccupNorm'],
      dtype='object')
          Prediction Time: 18.65466022491455 seconds
          Testing RMSE: 1.9378393197283144
Self Exploration, Random Case 3:
          L: 4
          Features: Index(['DistCoastNorm', 'AveOccupNorm'], dtype='object')
          Prediction Time: 7.76350998878479 seconds
          Testing RMSE: 1.637517114486739
Self Exploration, Random Case 4:
          L: 4
          Features: Index(['AveRoomsNorm', 'MedIncNorm', 'AveBedrmsNorm',
'PopulationNorm',
       'InlandNorm', 'AveOccupNorm', 'DistCoastNorm'],
      dtype='object')
```

```
          Prediction Time: 19.697559118270874 seconds
          Testing RMSE: 1.8679386051626732
Self Exploration, Random Case 5:
          L: 2
          Features: Index(['HouseAgeNorm', 'AveBedrmsNorm', 'InlandNorm',
'PopulationNorm',
       'MedIncNorm', 'AveRoomsNorm', 'DistCoastNorm'],
      dtype='object')
          Prediction Time: 2.5410521030426025 seconds
          Testing RMSE: 1.2706477665834799
=======================================================================
Self Exploration,  All Features, Sequential L Case 1:
            L: 1
            Testing RMSE: 0.8379137060969674
            Prediction Time: 1.2049121856689453 seconds
Self Exploration,  All Features, Sequential L Case 2:
            L: 2
            Testing RMSE: 0.8432303827864862
            Prediction Time: 1.2536530494689941 seconds
Self Exploration,  All Features, Sequential L Case 3:
            L: 3
            Testing RMSE: 0.8572684799851482
            Prediction Time: 19.758270978927612 seconds
```

your answer here

What combination of features and distance function provide the lowest RMSE on the test set? Do your decisions affect the running time of the algorithm?

In my exploration, I found that the combination of all features with L=1 led to the lowest RMSE on the test set. Of course, my combinations were not exhaustive, so there may be a better combination. However, as much as I was able to test, that model was the best. It becomes apparent that adding more features and increasing L can increase the run-time of the model. From what I see, increasing L to 3 can cause the largest increasing in run-time.

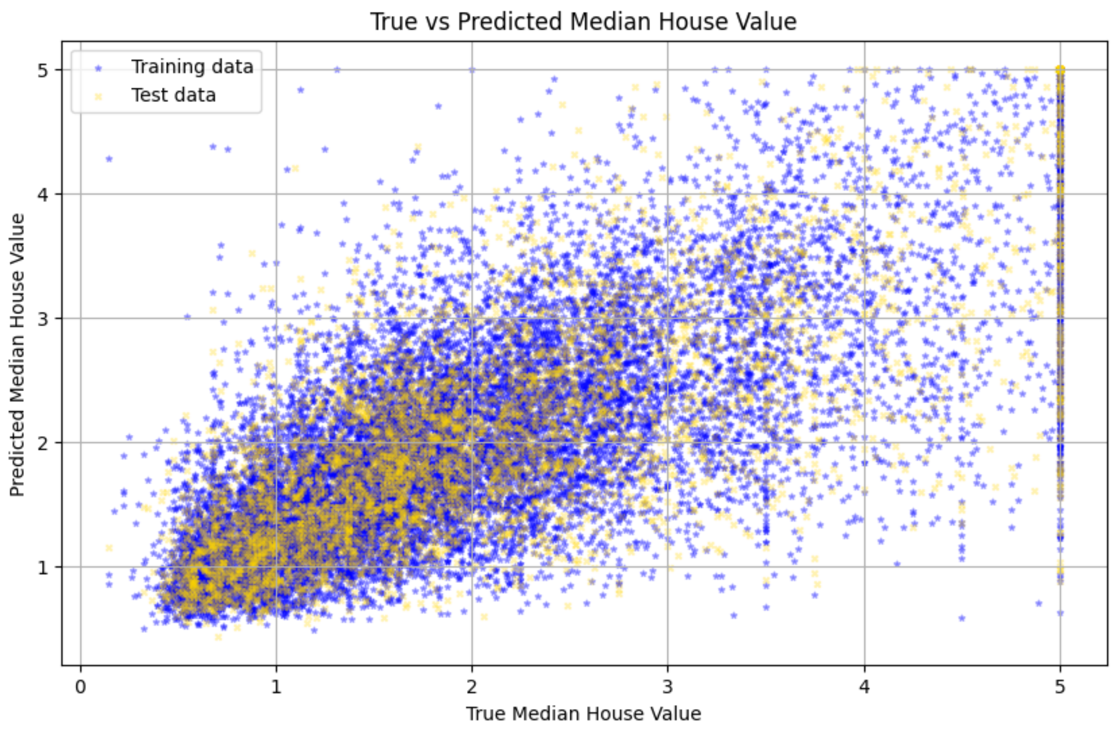
3.0.4 2.4 K-nearest neighbors algorithm

Now, implement the K-nearest neighbors algorithm and repeat the analysis in 2.2 by using 5 neighbors (`K=5`). The function(s) you write here will be used several more times in this problem set, so do your best to write efficient code! Make sure to do the following: 1. Use your algorithm to predict the median home value of every instance in the test set. Report the RMSE ("test RMSE") 2. Use your algorithm to predict the median home value of every instance in the training set and report the training RMSE. 3. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis. 4. Report an estimate of the total time taken by your code to predict the nearest neighbors for all the values in the test data set. 5. How does the performance (test RMSE and total runtime) of your nearest neighbors algorithm compare to the baseline in part 1.4?

Note: Runtime should not exceed a couple of minutes. If its taking longer then we strongly suggest you go back to your code and make it more efficient.

```
[ ]: # your code here

     # Defining the K-Nearest Neighbors function

     def kNN(X_train, X_test, y_train, y_test,k = 5, L=2, training = 'yes'):
         #almost identical to basic NN function
         X_test_np = X_test if training == 'no' else X_train
         y_test_np = y_test if training == 'no' else y_train

         # Preallocate memory for predictions
         predictions = np.empty(X_test_np.shape[0])

         # Compute distances and predictions
         for i, test_row in enumerate(X_test_np):
             distances = np.linalg.norm(X_train - test_row, ord=L, axis=1)
             if training == 'yes':
                 distances[i] = np.inf  # Ignore self in nearest neighbor␣
       ↪calculation for training
             nearest_neighbors = np.argsort(distances)[:k]
             prediction = np.mean(y_train[nearest_neighbors])
             predictions[i] = prediction
         # Compute RMSE
         rmse = compute_rmse(predictions, y_test_np)
         return predictions, rmse

     X_train_normfnp = X_train_normf[['MedIncNorm',
                                      'HouseAgeNorm',
                                      'AveRoomsNorm']].to_numpy()
     X_test_normfnp = X_test_normf[['MedIncNorm',
                                    'HouseAgeNorm',
                                    'AveRoomsNorm']].to_numpy()

     cal_df_trainnorm['KNN_predtrain'], train_rmse = kNN(X_train = X_train_normfnp,
                                                         X_test = X_test_normfnp,
                                                         y_test =␣
       ↪cal_df_testnorm['MedHouseVal'],
                                                         y_train␣
       ↪=cal_df_trainnorm['MedHouseVal'])
     print(f"This is the the training RMSE: {train_rmse}")

     #STEP 2
     t0 = time.time()
     cal_df_testnorm['KNN_predtest'], test_rmse = kNN(X_train = X_train_normfnp,
                                                      X_test = X_test_normfnp,
                                                                         y_test =␣
       ↪cal_df_testnorm['MedHouseVal'],
                                                                         y_train␣
       ↪=cal_df_trainnorm['MedHouseVal'],
                                                                         training = 'no')
     t1 = time.time()
     print(f"Prediction Time: {t1-t0} seconds")
     print(f"This is the the testing RMSE: {test_rmse}")

     #STEP 3
     plt.figure(figsize=(10, 6))
     plt.scatter(cal_df_trainnorm['MedHouseVal'],
                 cal_df_trainnorm['KNN_predtrain'],
                 color='blue', alpha=0.3,
                 label='Training data', marker = '*', s = 10)
     plt.scatter(cal_df_testnorm['MedHouseVal'],
                 cal_df_testnorm['KNN_predtest'],
                 color='gold', alpha=0.3,
                 label='Test data', marker = 'x', s = 10)
     plt.xlabel('True Median House Value')
     plt.ylabel('Predicted Median House Value')
     plt.title('True vs Predicted Median House Value')
     plt.legend()
     plt.grid(True)
     plt.show()
```

```
This is the the training RMSE: 0.788374008782271
Prediction Time: 10.32345175743103 seconds
This is the the testing RMSE: 0.7949710611649853
```



your answer here

1. Use your algorithm to predict the median home value of every instance in the test set. Report the RMSE ("test RMSE")

This is the the testing RMSE: 0.7949710611649853.

2. Use your algorithm to predict the median home value of every instance in the training set and report the training RMSE.

This is the the training RMSE: 0.788374008782271

3. Create a scatter plot that shows the true value of each instance on the x-axis and the predicted value of each instance on the y-axis.

See above.

4. Report an estimate of the total time taken by your code to predict the nearest neighbors for all the values in the test data set.

The prediction time for all values in the test data set is about 11.735754013061523 seconds seconds.

5. How does the performance (test RMSE and total runtime) of your nearest neighbors algorithm compare to the baseline in part 1.4?

The run-time is proportionally much larger than either of the baseline models (part 1.3 and 1.4), but the RMSE is much smaller overall (i.e. for both the training and

testing set). Thus, even though the algorithm runs slower, we have higher predictive strength.

3.0.5 2.5 Cross-Validation

How can we choose K without overfitting? As discussed during lecture time, one possible solution is to use k-fold cross-validation on the training sample. Here you must implement a simple k-fold cross-validation algorithm yourself. The function(s) you write here will be used several more times in this problem set, so do your best to write efficient code!

Use 20-fold cross-validation and report the average RMSE for your K-nearest neighbors model using Euclidean distance with the same set of features used in 2.4 (`MedIncNorm, HouseAgeNorm and AveRoomsNorm`) and 5 neighbors (`K=5`) as well as the total running time for the full run of 20 folds.

In other words, randomly divide your training dataset (created in 1.2) into 20 equally-sized samples. For each of the 20 iterations (the "folds"), use 19 samples as "training data" (even though there is no training in k-NN!), and the remaining 1 sample for validation. Compute the RMSE of that particular validation set, then move on to the next iteration.

- Report the average cross-validated RMSE across the 20 iterations and compare to the result you obtained in 2.3. What do you observe?
- Report the runtime of your algorithm.How does it compare to your previous results?

Note 1: Runtime should not exceed a couple of minutes. If its taking longer then we strongly suggest you go back to your code and make it more efficient.

Note 2: The sklearn package has a built-in K-fold iterator – you should *not* be invoking that or any related algorithms in this section of the problem set.

Note 3: To perform any randomized operation, only use functions in the *numpy library (np.random)*. Do not use other packages for random functions.

```
[ ]: np.random.seed(seed=1948)


     # Shuffle indices for cross-validation
     n_splits = 20
     n_rows = len(cal_df_trainnorm)
     indices = np.arange(n_rows)
     np.random.shuffle(indices)
     cal_train_normf_shuffled = cal_df_trainnorm[['MedIncNorm',
                                             'HouseAgeNorm',
                                             'AveRoomsNorm', 'MedHouseVal']].
       ↪iloc[indices]
     X_train_normfnp = cal_train_normf_shuffled[['MedIncNorm', 'HouseAgeNorm',␣
       ↪'AveRoomsNorm']].to_numpy()
     y_train_normfmnp = cal_train_normf_shuffled['MedHouseVal'].to_numpy()

     # Calculate fold size
     fold_size = n_rows // n_splits
```

```python
# Placeholder for the RMSE of each fold
rmse_vals = np.empty(n_splits)

t0 = time.time()
for i in range(n_splits):
    # Indices for the validation set
    start_val = i * fold_size
    end_val = start_val + fold_size
    val_indices = indices[start_val:end_val]

    # Indices for the training set
    train_indices = np.concatenate((indices[:start_val], indices[end_val:]))

    # Select the data for the current fold using the indices
    X_train_temp = X_train_normfnp[train_indices]
    y_train_temp = y_train_normfmnp[train_indices]

    X_test_temp = X_train_normfnp[val_indices]
    y_test_temp = y_train_normfmnp[val_indices]

    # Run kNN on the selected data
    _, test_rmse = kNN(X_train = X_train_temp,
                       y_train = y_train_temp,
                       X_test = X_test_temp,
                       y_test= y_test_temp,
                       training = 'no')
    rmse_vals[i] = test_rmse

t1 = time.time()

# Calculate average RMSE across all folds
average_rmse = np.mean(rmse_vals)
print(f"The average testing RMSE for 20-fold cross-validation: {average_rmse}")
print(f"Time for Cross-Validation: {(t1-t0)} seconds")
```

```
The average testing RMSE for 20-fold cross-validation: 0.7886362383707489
Time for Cross-Validation: 31.98101782798767 seconds
```

your answer here

- Report the average cross-validated RMSE across the 20 iterations and compare to the result you obtained in 2.4. What do you observe?

The average testing cross-validated RMSE is 0.7886362383707489. It is slightly higher than but very close/similar to the RMSE found in 2.4. We can see how cross-validation prevents a overly small RMSE by accounting for multiply possible training and test sets due to variability in data splits. The result is an RSME that accounts better for overfitting or biases that weren't apparent in the initial test split. The RMSE is

higher, but now presents a more accurate estimate of the model's error on unseen data.

- Report the runtime of your algorithm. How does it compare to your previous results?

The run-time is much slower now. It now costs about 30 seconds to run what previously took about 12 seconds. However, the value of cross-validation is not lost due to its longer run-time. The goal is to account for those unseen biases and overfitting in the simply train-test split. Thus, there is benefit in spite of the run time.

3.0.6 2.6 Using cross validation to find the optimal value for K

Compute the cross-validated RMSE for values of K between 1 and 25 using 10-fold cross-validation and L2 normalization. Use the following features in your model: `MedIncNorm, HouseAgeNorm and AveRoomsNorm`. Create a graph that shows how cross-validated RMSE changes as K increases from 1 to 25. Label your axes, and summarize what you see. What do you think is a reasonable choice of K for this model?

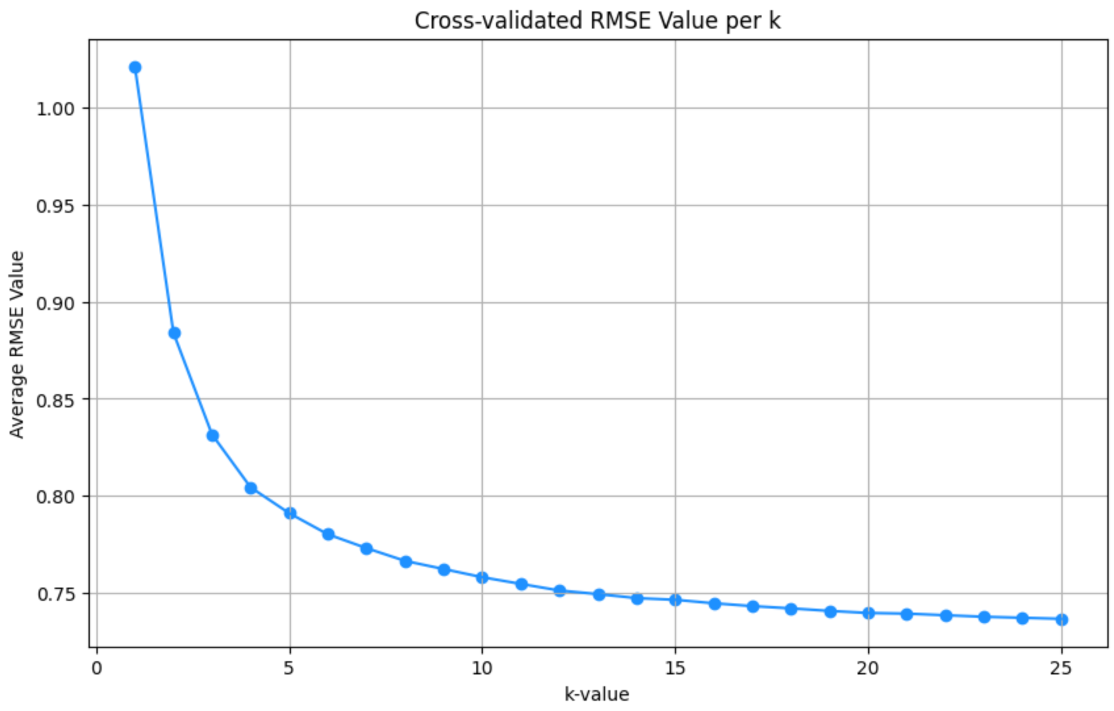
Finally, "train" a K-nearest neighbor model using the value of K that minimized the cross-validated RMSE and report the test RMSE. (Continue to use L2 normalization and the same set of features). How does the test RMSE compare to the cross-validated RMSE, and is this what you expected?

Note: Runtime should not exceed ~30 min. If its taking longer then we strongly suggest you go back to your code and make it more efficient.

```
[ ]: # your code here

np.random.seed(seed=1948)

# Shuffle indices for cross-validation
n_splits = 10
n_rows = len(cal_df_trainnorm)
indices = np.arange(n_rows)
np.random.shuffle(indices)
cal_train_normf_shuffled = cal_df_trainnorm[['MedIncNorm',
                                        'HouseAgeNorm',
                                        'AveRoomsNorm', 'MedHouseVal']].
  ↪iloc[indices]
X_train_normfnp = cal_train_normf_shuffled[['MedIncNorm', 'HouseAgeNorm',␣
  ↪'AveRoomsNorm']].to_numpy()
y_train_normfmnp = cal_train_normf_shuffled['MedHouseVal'].to_numpy()

# Calculate fold size
fold_size = n_rows // n_splits

t0 = time.time()
rmse_avgs = np.empty(25)
for j in range(1,26):
    # Placeholder for the RMSE of each fold
    rmse_vals = np.empty(n_splits)
    for i in range(n_splits):
        # Indices for the validation set
        start_val = i * fold_size
        end_val = start_val + fold_size
        val_indices = indices[start_val:end_val]

        # Indices for the training set
        train_indices = np.concatenate((indices[:start_val], indices[end_val:]))

        # Select the data for the current fold using the indices
        X_train_temp = X_train_normfnp[train_indices]
        y_train_temp = y_train_normfmnp[train_indices]

        X_test_temp = X_train_normfnp[val_indices]
        y_test_temp = y_train_normfmnp[val_indices]

        # Run kNN on the selected data
        _, test_rmse = kNN(X_train = X_train_temp,
                           y_train = y_train_temp,
                           X_test = X_test_temp,
                           y_test= y_test_temp,
                           k = j,
                           training = 'no')
        rmse_vals[i] = test_rmse
    rmse_avgs[j-1] = np.mean(rmse_vals)
t1 = time.time()

# Calculate average RMSE across all folds
average_rmse = np.mean(rmse_vals)
print(f"Time for Cross-Validation: {(t1-t0)/60} minutes")
```

```
Time for Cross-Validation: 12.245089630285898 minutes
```

[ ]:
```
k_graphdf = pd.DataFrame({'k':np.arange(1,26)})
k_graphdf['avgrmse'] = rmse_avgs



#STEP 3
plt.figure(figsize=(10, 6))
plt.plot(k_graphdf['k'],
         k_graphdf['avgrmse'],
         color='dodgerblue')
plt.scatter(k_graphdf['k'],
         k_graphdf['avgrmse'],
         color='dodgerblue')
plt.xlabel('k-value')
plt.ylabel('Average RMSE Value')
plt.title('Cross-validated RMSE Value per k')
plt.grid(True)
plt.show()
```



[ ]:
```
X_train_normfnp = X_train_normf[['MedIncNorm',
                                 'HouseAgeNorm',
                                 'AveRoomsNorm']].to_numpy()
X_test_normfnp = X_test_normf[['MedIncNorm',
                               'HouseAgeNorm',
                               'AveRoomsNorm']].to_numpy()


#STEP 3
t0 = time.time()
a, test_rmse = kNN(X_train = X_train_normfnp,
                   X_test = X_test_normfnp,
                   y_test = cal_df_testnorm['MedHouseVal'],
                   y_train =cal_df_trainnorm['MedHouseVal'],
                   k = 25,
                   training = 'no')
t1 = time.time()
print(f"Prediction Time: {t1-t0} seconds")
print(f"This is the the testing RMSE: {test_rmse}")
print(f"This is the the cross-validated RMSE for k = 25: {rmse_avgs[-1]}")
```

```
Prediction Time: 10.530919075012207 seconds
This is the the testing RMSE: 0.7439009482082435
This is the the cross-validated RMSE for k = 25: 0.7367691310268063
```

your answer here

What do you think is a reasonable choice of K for this model?

I think that a k between 10 and 25 is reasonable. While k = 25 is the minimizing k-value, it is more time consuming to run a model with k = 25. Thus, if the relative change in RMSE is minimal at a certain point, a smaller k might be fine. Of course, if the ultimate goal is minizing RMSE at all costs, then k = 25 is the optimal k from what we have seen above. However, when accounting for the risk of overfitting, a smaller k might be ideal for external validity of the model.

How does the test RMSE compare to the cross-validated RMSE, and is this what you expected?

The test RMSE is higher than the cross-validated RMSE, which I found unexpected at first since k = 25 already seems quite large and at risk for underfitting. Due to the bias-variance tradeoff, the quality of the model may decline as k gets too large. When k is equal to the size of the dataset, then the algorithm is not different than using the mean to predict the outcome like in model 1.3. At the same time, for kNN, overfitting occurs when k in smaller thus having a higher k minimize RMSE is not unexpected. It might be that the risk of underfitting increases at k values much larger than 25. We might expect the graph from above to become more clearly convex as k's larger than 25, which visually demonstrats the bias-variance trade-off. Since we can't see that convex shape, we can't tell if a higher k is more ideal. From the k's test, k = 25 is the best.

Overall, we can see that that k = 25 is probably not too large of a k considering that some level of overfitting still seems to be ocurring.

---

4 Part III: Overfitting in Model Selection and Nested Cross Validation

In this last part of the problem set, we will examine why overfitting is a serious concern when estimating hyperparameters and how to address it.

For this part of the problem set you are allowed to use machine learning libraries. We don't expect you to use your own algorithms developed in part 2. We strongly suggest that you use the following libraries and resources, but feel free to choose your favorite Python ML libraries.

```
[ ]: from sklearn.model_selection import GridSearchCV, KFold, cross_val_score
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.metrics import roc_auc_score
     import joblib
```

For this part of the problem set we will no longer be using the California Housing Dataset. Instead, we will generate our own synthetic data. The advantage of doing so is that we get to choose the data generating process. We will use the knowledge about the data generating process to test the robustness of different approaches to estimating out-of-sample performance.

We will attempt the following classification problem: predict a binary response variable $y \sim Bernoulli(p = 1/2)$ from a set of independent features $X = [x_1, ..., x_J]$ where $x_j \sim Unif(a = 0, b = 1)$, $1 \leq j \leq J$.

You can use the following function to generate samples from this distribution.

```
[ ]: def generate_random_sample(nobs,J):
         X = pd.DataFrame(np.random.random_sample(size=(nobs, J)),␣
      ↪columns=[f'feature_{x}' for x in range(J)])
         y = np.random.binomial(n=1,p=1/2,size=nobs)
         return X,y

     X_train,y_train = generate_random_sample(nobs=2*10**3,J=100)
```

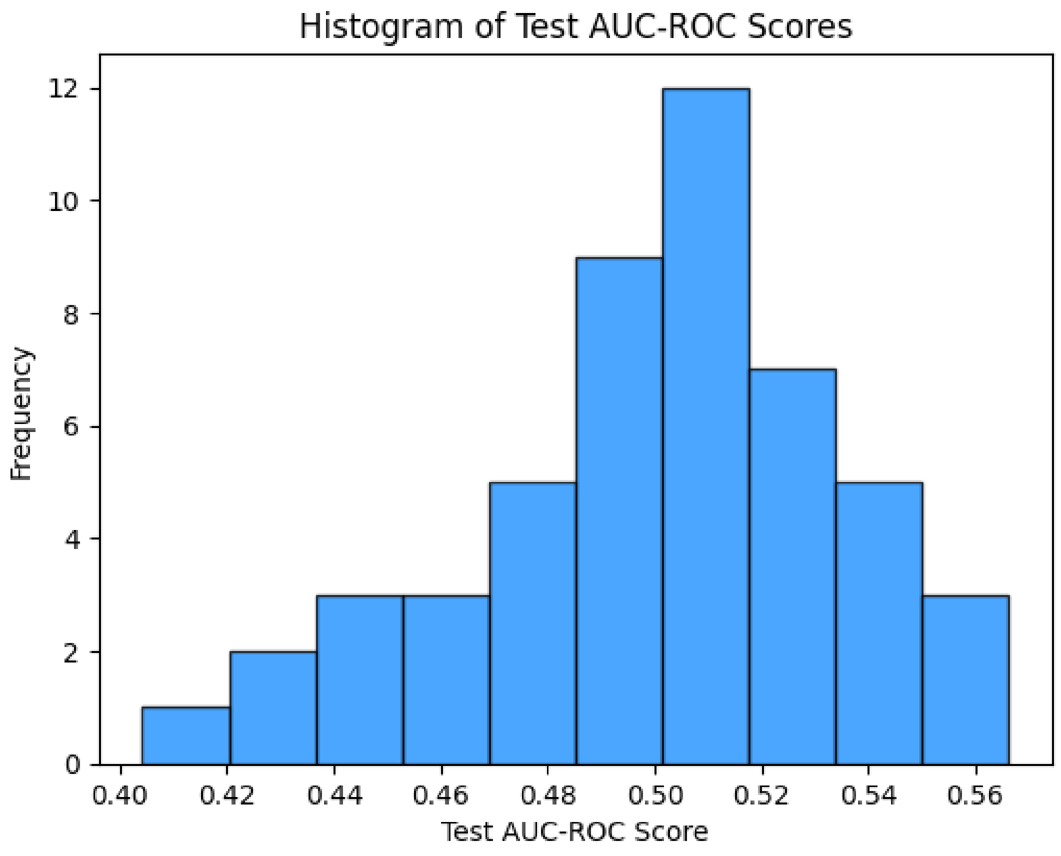
4.0.1 3.1 Out-of-sample performance

We are going to be using the area under the ROC curve (AUC-ROC) as the evaluation score. What kind of out-of-sample performance would you expect from classification models trained and tested on this data? Test whether your intuition is correct by carrying out the following iterative procedure:

1. For each iteration in 1,2,3,...,50:
   - Generate a training sample containing 2,000 observations and J=100 features. Likewise, generate a test sample containing 200 observations and J=100 features.
   - Train some K-nearest neighbors model on the training sample with some arbitrary choice of K (no need to cross validate the choice of K or put any work into it, we'll get to that later on).
   - Evaluate the AUC-ROC on the test set.
2. Plot a histogram of the test AUC-ROC scores.
3. Report the average of the test AUC-ROC scores.

```
[ ]: # your code here

     np.random.seed(1948)

     iterations = 50
     k = 5  # arbitrary choice of K
     auc_roc_scores = []

     for i in range(iterations):
         # Generate training and test samples
         X_train, y_train = generate_random_sample(nobs=2000, J=100)
         X_test, y_test = generate_random_sample(nobs=200, J=100)

         # Train K-nearest neighbors model
```

```
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    # Predict probabilities for the test set
    y_pred_proba = knn.predict_proba(X_test)[:, 1]

    # Evaluate the AUC-ROC on the test set
    roc_auc = roc_auc_score(y_test, y_pred_proba)
    auc_roc_scores.append(roc_auc)

# Plot a histogram of the test AUC-ROC scores
plt.hist(auc_roc_scores, bins = 10, color='dodgerblue', alpha = 0.8,␣
  ↪edgecolor='black')
plt.xlabel('Test AUC-ROC Score')
plt.ylabel('Frequency')
plt.title('Histogram of Test AUC-ROC Scores')
plt.show()

# Report the average of the test AUC-ROC scores
average_auc_roc = np.mean(auc_roc_scores)
print(f'The average of the test AUC-ROC scores: {average_auc_roc}')
```



```
The average of the test AUC-ROC scores: 0.5003461348776238
```

your answer here

The average of the test AUC-ROC scores is 0.5003461348776238 which is really close to about 1/2. This is not unexpected since the data is completely randomly generated. There is unlikely to be a strong relationship between the features and outcome variable. This means the models ranks a random positive example higher than a random negative example around 50% of the time. Its predicitive ability is basically no better than random guessing, which aligns with the randomness of the data.

4.0.2 Fix a sample

In real life settings we wouldn't be able to draw test and train samples at will. For the rest of the pset (3.2-3.6) we will fix a training and test sample:

```
[ ]: X_train,y_train = generate_random_sample(nobs=2*10**3,J=100)
     X_test,y_test = generate_random_sample(nobs=2*10**2,J=100)
```

4.0.3 3.2 k-fold cross-validation

Use 10-fold cross-validation on the train sample to find the optimal K and report the hyperparameter value. Report also the average of the cross validated scores for the optimal hyperparameter value.

```
[ ]: # your code here

     # Create KNN model
     knn = KNeighborsClassifier()

     # Create a dictionary of all values we want to test for n_neighbors
     param_grid = {'n_neighbors': np.arange(1, 51)}

     # Use grid search to test all values for n_neighbors
     knn_gscv = GridSearchCV(knn, param_grid,
                             cv= KFold(n_splits=10,
                                       shuffle=True,
                                       random_state=1948),
                             scoring='roc_auc')

     # Fit model to data
     knn_gscv.fit(X_train, y_train)

     # Check top-performing n_neighbors value
     optimal_k = knn_gscv.best_params_['n_neighbors']

     # Check mean score for the top-performing value of n_neighbors
```

```
#based on the sklearn documentation best_score_ is the mean score
optimal_score = knn_gscv.best_score_

print(f"Optimal k (number of neighbors): {optimal_k}")
print(f"Average cross-validated AUC-ROC score for the optimal k:␣
 ↪{optimal_score}")
```

```
Optimal k (number of neighbors): 38
Average cross-validated AUC-ROC score for the optimal k: 0.5046658327099013
```

4.0.4 3.3 Nested cross-validation

Use nested cross validation (3,4,5,6) on the training sample. In the outer loop you should be estimating model performance and in the inner loop you should be doing regular k-fold cross validation to find the optimal K. Use 10 folds for the inner cv and 3 folds for the outer cv. Report the average of the cross-validated scores of the outer loop.

[ ]:
```
# your code here

# Define the inner and outer cross-validation settings
inner_cv = KFold(n_splits=10, shuffle=True, random_state=1948)
outer_cv = KFold(n_splits=3, shuffle=True, random_state=1948)

# Define the range of k values to search
param_grid = {'n_neighbors': np.arange(1, 51)}

# Setup the kNN classifier
knn = KNeighborsClassifier()

# Initialize the grid search with the kNN model, parameter grid, and inner␣
 ↪cross-validation method
clf = GridSearchCV(estimator=knn, param_grid=param_grid, cv=inner_cv)

# Perform nested cross-validation and store the scores
nested_score = cross_val_score(clf, X=X_train, y=y_train, cv=outer_cv)

# Report the average of the cross-validated scores of the outer loop
print(f"Average ROC-AUC score of the outer loop: {nested_score.mean()}")
```

```
Average ROC-AUC score of the outer loop: 0.49699324511918225
```

4.0.5 3.4 Take stock of the results so far

Based on the results of 3.1, 3.2 and 3.3, what can you say about estimating out-of-sample performance? Is the average of the cross-validated scores a good estimator? How about the average of the nested cross-validated scores? Are they underestimating or overestimating true out-of-sample performance?

your answer here

Based on the results of 3.1, we know that directly evaluating the kNN model on a separate test set results in an ROC-AUC score very close to 0.5, which is what we'd expect from a model making predictions no better than random guessing since the data is random.

In 3.2, the results showed a slightly higher average ROC-AUC score. This implies a small level though ultimately miniscule degree of overestimating true out of sample performance. It's still very close to 0.5, so this increase might not be meaningful and could simply be due to variance in the data splits.

On the other hand, the nested cross-validated scores in 3.3 have an average that is slightly lower than 0.5. This is likely the most robust score, and is close to 0.5 like the rest of the average scores found. The lower score might be due to the more conservative nature of nested cross-validated scores. It accounts more for the variability and bias introduced by the hyperparameter tuning process which results in a less optimistic score for out-of-sample preformance.

Thus, for this particular example, all the scores seems relatively fine. They are all close to 0.5 and show no incredibly large differences. I wouldn't call any of the bad estimators and I wouldn't say one is wildly better than another. At most, we can say that the average of the cross-validated scores might be overestimating out-of-sample performance and the average of the nested cross-validated scores might be underestimating out-of-sample performance. The estimator we prefer might in other situations will likely be the more conservative nested-cross validation score. In situations without such curated data, we find the score differ by much more and would prefer to stay on the more conservative side.

4.0.6 3.5 Comparing k-fold and nested cross-validation [extra-credit]

We would like to better assess the difference between the k-fold and nested cross-validation scores and make sure that the results we observed in 3.2 and 3.3 are not a fluke. To do this, repeat both experiments 50 times. In each iteration, pass a different value for the "random_state" parameter in the KFold function to ensure that there is variation in the fold splitting.

In a single figure, plot two histograms. One showing the distribution of the k-fold scores, another showing the distribution of the nested scores. Use gold for the color of the objects related to the nested scores and blue for the color of the objects related to the k-fold scores.

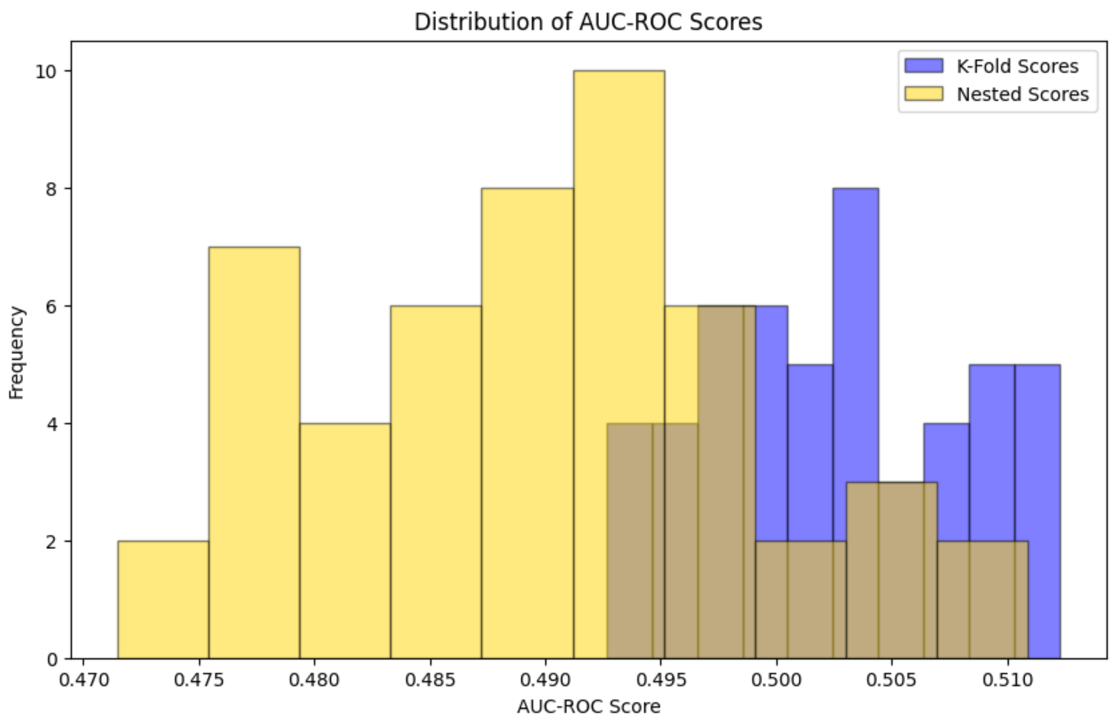
Note 1: you should NOT be generating a new sample – continue working with the dataset fixed ahead of question 3.2.

Note 2: Runtime should not exceed 30 min. If its taking longer then we strongly suggest you go back to your code and make it more efficient.

[ ]:
```python
# your code here

# Define the range of k values to search and the classifier
param_grid = {'n_neighbors': np.arange(1, 51)}
knn = KNeighborsClassifier()

# Lists to store scores
```

```
k_fold_scores = []
nested_scores = []

# Repeat both experiments 50 times
for i in range(50):
    # Update the random state for each iteration
    random_state = i

    # K-fold cross-validation setup
    inner_cv = KFold(n_splits=10, shuffle=True, random_state=random_state)

    # Perform k-fold cross-validation
    knn_gscv = GridSearchCV(knn, param_grid, cv=inner_cv, scoring='roc_auc')
    knn_gscv.fit(X_train, y_train)
    best_score = knn_gscv.best_score_
    k_fold_scores.append(best_score)

    # Nested cross-validation setup
    outer_cv = KFold(n_splits=3, shuffle=True, random_state=random_state)

    # Perform nested cross-validation
    nested_score = cross_val_score(knn_gscv, X=X_train, y=y_train, cv=outer_cv,␣
  ↪scoring='roc_auc').mean()
    nested_scores.append(nested_score)
```

```
[ ]: # Plotting the histograms
plt.figure(figsize=(10, 6))

plt.hist(k_fold_scores, color='blue', alpha=0.5, label='K-Fold Scores',␣
  ↪bins=10, edgecolor='black')
plt.hist(nested_scores, color='gold', alpha=0.5, label='Nested Scores',␣
  ↪bins=10, edgecolor='black')

plt.xlabel('AUC-ROC Score')
plt.ylabel('Frequency')
plt.title('Distribution of AUC-ROC Scores')
plt.legend()
plt.show()
```



4.0.7 3.6 Conclusion [extra-credit]

Based on the figure from 3.5, would you adjust your answer to question 3.4? In a couple of sentences, explain why overfitting can arise when doing model selection, and why nested cross-validation is a useful tool in preventing it.

your answer here

The figure confirms my answer in 3.4. It is true that the nested cross-validation results in more conservative average ROC-AUC scores when compared to the K-fold scores. The more conservative nature of the nested cross-validation results, can help improve estimates of out of sample performance especially when attempting to be wary of overestimating the accuracy.

Overall, nested cross-validation is a way to better account for the biases of overfitting

Overfitting arises when the hyperparameter tuning process captures noise in the training data, which leads to an overly complex model that performs well on the training data but poorly on unseen data. There is a greater risk for this in k-fold cross-validation because the same data is used to both select the model (tune hyperparameters) and evaluate its performance.

On the other hand, nested cross-validation accounts for overfitting in model selection by separating the data used for hyperparameter tuning (inner loop) from the data used for performance estimation (outer loop). This ensures that the performance estimate is based on completely unseen data, not influenced by the data used in the

model selection process, providing a more unbiased evaluation of the model's ability to generalize.

This isn't to say that the k-fold cross-validation is incredibly inaccurate. However when attempting to avoid overfitting, nested cross-validation appears to be a better option.