In [2]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
pd.set_option("future.no_silent_downcasting", True)
pd.options.mode.chained_assignment = None
```

Homework 6: Decision Trees, Linear Regression (50 points)

Please complete this notebook by filling in the cells provided.

!! Please submit a `PDF` + `.ipynb` of your notebook.

Name: Wonjae Oh:wonjae

Names and Onyens of fellow students you discussed Homework4 problems and ideas:

- Students: N/A

We encourage discussing ideas and brainstorming with your peers, but the final text, code, and comments in this homework assignment MUST be 100% written by you as mentioned in syllabus.

Problem 1. You need a loan? Hmm, let me decide (25 points)

Many of you have expressed interest in financial technology, so for this problem you will be using historic data from a bank about whether or not people were granted loans. You'll be using this data to create a predictor for whether the bank would grant loans to future applicants.

Problem 1.1 Data Preparation (4 points)

First, load the loan data into a table named `loan_data_raw`.

In [29]:
```
loan_data_raw = pd.read_csv('loan_data.csv')
```

Problem 1.1.1 (1 point)

We don't need the `Loan_ID` column, so drop that column and store the resulting table in `loan_data`.

In [30]:
```
# write your code here
# BEGIN SOLUTION
loan_data = loan_data_raw.drop(['Loan_ID'], axis=1)
# END SOLUTION
```

Problem 1.1.2 (1 point)

Some of the values are missing from the table, so write the code to drop rows from `loan_data` that have any missing values, and store the result in `loan_data_clean`.

In [31]:
```
# write your code here
# BEGIN SOLUTION
loan_data_clean = loan_data.dropna()
# END SOLUTION
```

Problem 1.1.3 (2 points)

You'll be using functions from `scikit-learn`, which don't know how to handle text values like `Male` or `Female` or `Yes` or `No`, so you'll have to turn them into numbers. For the `Gender` and `Married` columns, use boolean indexing to do this by assigning 0 for `Male` and 1 for `Female` and 0 for `No` and 1 for `Yes` (just like we did in lecture demo). Some of the code is already provided, you fill in the rest.

WARNING: running this cell twice will set `Gender` and `Married` to all zeroes! (think about why this is so) If you do this accidentally, you can always just run the cell above that drops values to re-create `loan_data_clean`.

In [32]:
```
# write your code here
# BEGIN SOLUTION
loan_data_clean.loc[loan_data_clean["Gender"] == "Male", "Gender"] = 0
loan_data_clean.loc[loan_data_clean["Gender"] == "Female ", "Gender"] = 1
loan_data_clean.loc[loan_data_clean["Married"] == "No", "Married"] = 0
loan_data_clean.loc[loan_data_clean["Married"] == "Yes", "Married"] = 1
loan_data_clean.head()
# END SOLUTION
```

Out[32]:

	Gender	Married	Dependents	Education	ApplicantIncome	CoapplicantIncome	LoanAmount	Lo
1	0	1	1	Graduate	4583	1508.0	128.0	
2	0	1	0	Graduate	3000	0.0	66.0	
3	0	1	0	Not Graduate	2583	2358.0	120.0	
4	0	0	0	Graduate	6000	0.0	141.0	
5	0	1	2	Graduate	5417	4196.0	267.0	

Problem 1.1.4 (0 points)

Copy the code below that will do the same thing (a faster way) for the remaining columns.

```
loan_data_clean['Loan_Status'] = loan_data_clean.Loan_Status.replace({'N': 0,
'Y': 1})
loan_data_clean['Property_Area'] =
loan_data_clean.Property_Area.replace({'Rural':0, 'Urban':1, 'Semiurban':2})
loan_data_clean['Education'] = loan_data_clean.Education.replace({'Graduate':0,
'Not Graduate':1})
loan_data_clean['Dependents'] = loan_data_clean.Dependents.replace({'0':0,
'1':1,'2':2,'3+':3})
loan_data_clean = loan_data_clean.astype(float)
loan_data_clean
```

In [42]:
```
# Copy code here and run it.
mapping = {
    'Loan_Status':   {'N': 0, 'Y': 1},
    'Property_Area': {'Rural': 0, 'Urban': 1, 'Semiurban': 2},
    'Education':     {'Graduate': 0, 'Not Graduate': 1},
    'Dependents':    {'0': 0, '1': 1, '2': 2, '3+': 3},
    'Gender':    {'Male': 0, 'Female': 1},
}

loan_data_clean = loan_data_clean.replace(mapping)
encoded_cols = list(mapping.keys())
loan_data_clean[encoded_cols] = loan_data_clean[encoded_cols].astype(float)
loan_data_clean.head()
```

Out[42]:

	Gender	Married	Dependents	Education	ApplicantIncome	CoapplicantIncome	LoanAmount	Loa
1	0.0	1	1.0	0.0	4583	1508.0	128.0	
2	0.0	1	0.0	0.0	3000	0.0	66.0	
3	0.0	1	0.0	1.0	2583	2358.0	120.0	
4	0.0	0	0.0	0.0	6000	0.0	141.0	
5	0.0	1	2.0	0.0	5417	4196.0	267.0	

Problem 1.2 Train and Test, part 1 (12 points)

Next, you'll be creating your training set and your test set.

Problem 1.2.1 (2 points)

As we did in class and lab, create the training data from 80% of `loan_data_clean` by using random sampling. The create the test set by dropping from `loan_data_clean` all the rows that are in the training set. We set the random seed below to make the sampling reproducible so that everyone will be working with the same training and test sets.

In [43]:
```
np.random.seed(420) # this makes the sampling reproducible, so everyone has same train a
# write your code here
# BEGIN SOLUTION
from sklearn.model_selection import train_test_split

train_loan, test_loan = train_test_split(loan_data_clean, test_size=0.2)
# END SOLUTION
print(train_loan.shape, test_loan.shape)
```

```
(414, 11) (104, 11)
```

Problem 1.2.2 (2 points)

You will be trying to predict `Loan_Status` from all the other data. Create the approriate `X_train_loan`, `y_train_loan`, `X_test_loan`, and `y_test_loan` tables for the train and test sets.

In [44]:
```
# write your code here
# BEGIN SOLUTION
y_train_loan = train_loan['Loan_Status']
X_train_loan = train_loan.drop(columns=['Loan_Status'])
```

```
y_test_loan = test_loan['Loan_Status']
X_test_loan = test_loan.drop(columns=['Loan_Status'])
X_train_loan.head()
# END SOLUTION
```

Out[44]:

	Gender	Married	Dependents	Education	ApplicantIncome	CoapplicantIncome	LoanAmount
394	0.0	1	2.0	0.0	3100	1400.0	113.0
613	1.0	0	0.0	0.0	4583	0.0	133.0
296	0.0	1	1.0	0.0	6875	0.0	200.0
57	0.0	1	0.0	0.0	3366	2200.0	135.0
463	1.0	0	1.0	1.0	5191	0.0	132.0

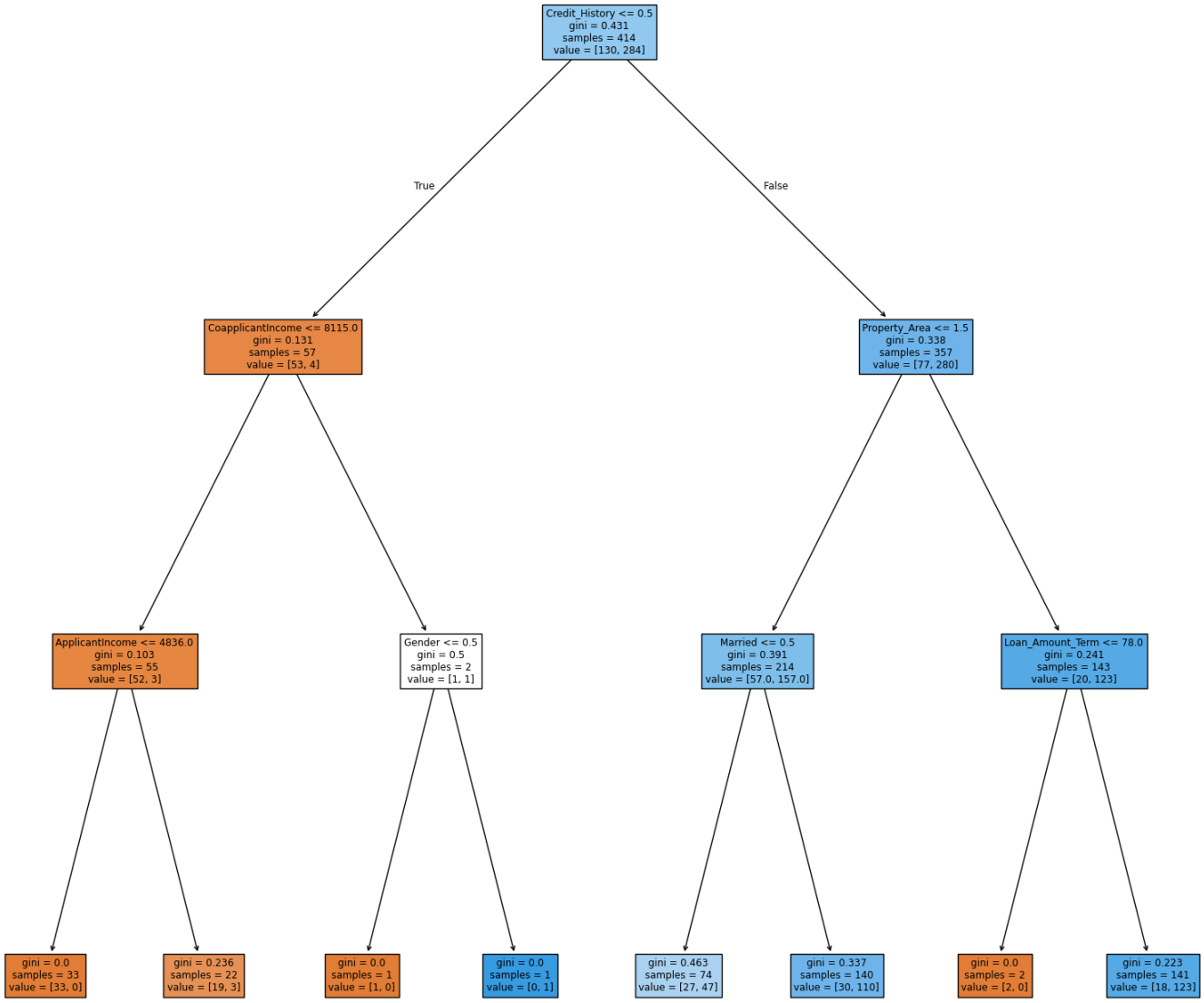
Problem 1.2.3 (2 points)

Finally, use the `DecisionTreeClassifier` to create a decision tree to use for prediction. Finish the code to print the scores of the classifier on the train and test sets.

In [45]:

```
from sklearn.tree import DecisionTreeClassifier, plot_tree

T = DecisionTreeClassifier(max_depth=3)
T.fit(X_train_loan, y_train_loan)
# write your code here
# BEGIN SOLUTION
train_score = T.score(X_train_loan, y_train_loan)
test_score = T.score(X_test_loan, y_test_loan)
# END SOLUTION
print('Score on train:', train_score)
print('Score on test:', test_score)

fig, ax = plt.subplots(1, figsize = (20, 20))
p = plot_tree(T, filled = True, feature_names = X_train_loan.columns)
```

```
Score on train: 0.8115942028985508
Score on test: 0.8653846153846154
```



Problem 1.2.4 (2 points)

Looking at the decision tree, which feature is the most important for predicting if a loan would be approved?

Credit_History <= 0.5 is the most important feature for predicting if a loan would be approved because it is the root node, and a decision tree works by finding the feature that will minimize the initial impurity (gini impurity) when the data is split by that feature.

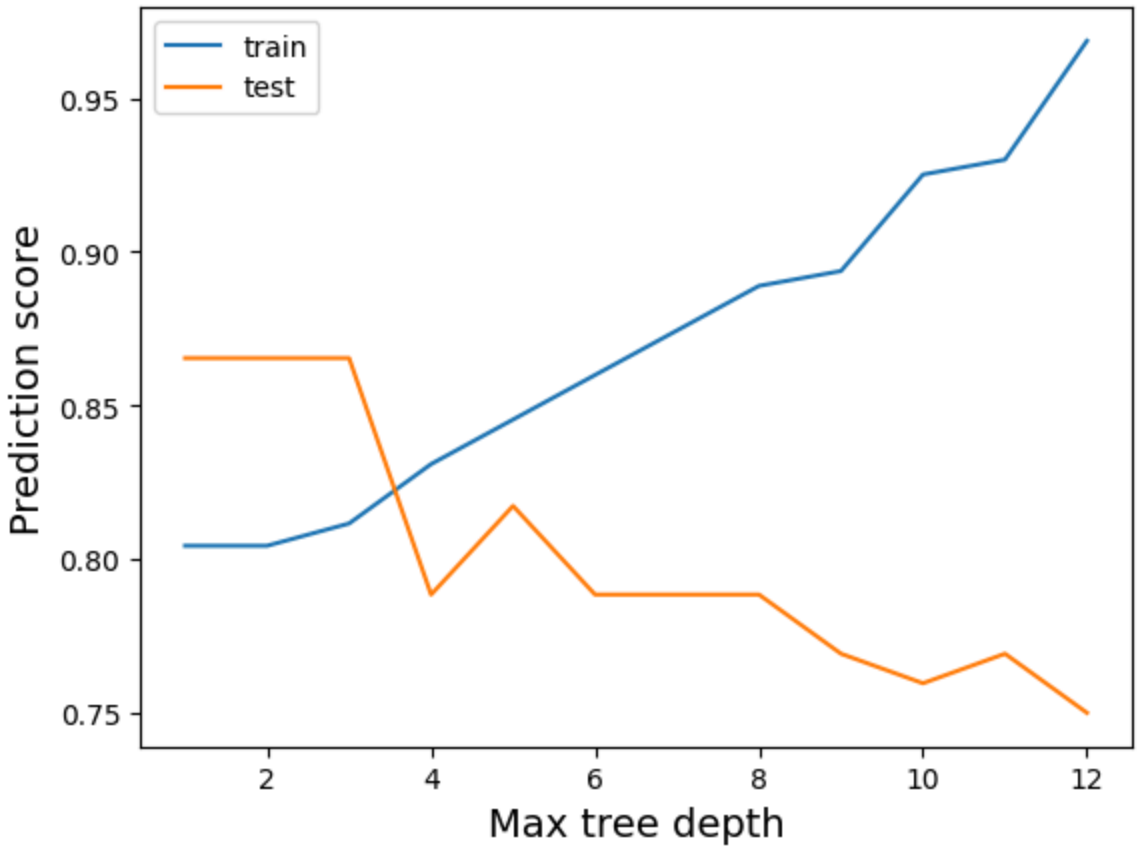
Problem 1.2.5 (2 points)

Next, we're going to gather some data for creating decision trees of different maximum tree depths. Finish the code below to call the decision tree classifier with different maximum depths.

In [47]:
```
train_scores=[]
test_scores=[]
```

```
r=range(1,13)
for i in r:
    # write your code here
    # BEGIN SOLUTION
    T = DecisionTreeClassifier(max_depth=i)
    # END SOLUTION
    T.fit(X_train_loan, y_train_loan)
    train_scores.append(T.score(X_train_loan, y_train_loan))
    test_scores.append(T.score(X_test_loan, y_test_loan))
fig, ax = plt.subplots(1)
sns.lineplot(x=r,y=train_scores,label='train')
sns.lineplot(x=r,y=test_scores,label='test')
ax.set_ylabel('Prediction score', fontsize=14)
ax.set_xlabel('Max tree depth', fontsize=14)
ax.legend()
```

Out[47]: <matplotlib.legend.Legend at 0x7f16edc9fb60>



Problem 1.2.6 (2 points)

Observe the generated plot.

1. Does the accuracy improve for the **train** set as the maximum tree depth increases?
2. Does the accuracy improve for the **test** set as the maximum tree depth increases?

1. Yes.
2. No, after a tree depth of 3, test accuracy decreases while train accuracy increases.

Problem 1.3 Train and Test, part 2 (9 points)

You may have noticed that the `Credit_History` feature dominates the loan decision process (which makes sense!) Using just this feature (max tree depth == 1), the prediction score is over 80%. But what

if you didn't have access to that feature? How well could you predict what the bank would do?

Problem 1.3.1 (1 point)

Finish the code below to drop the `Credit_History` column from your train and test data.

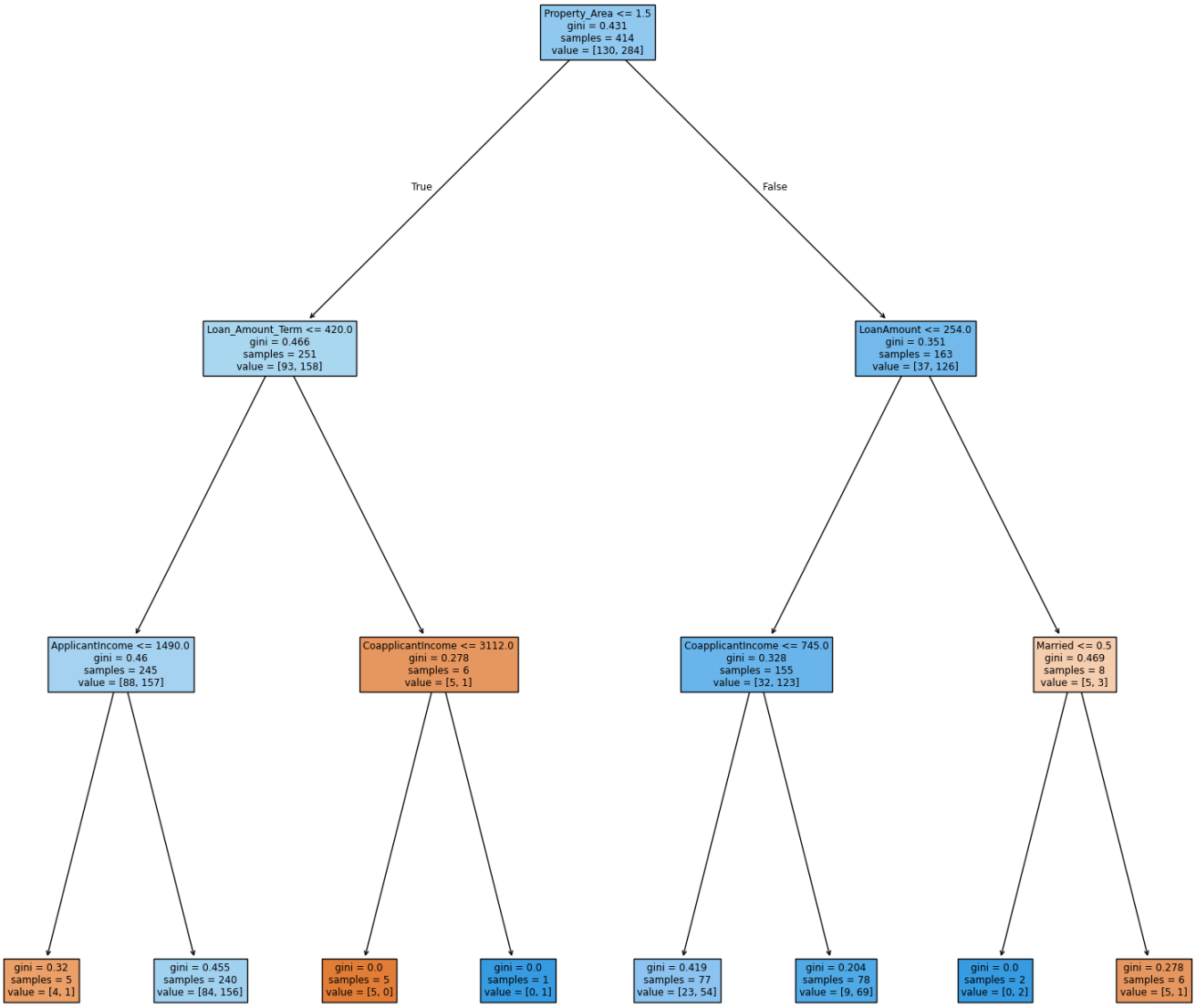
In [48]:
```
# write your code here
# BEGIN SOLUTION
X2_train_loan = X_train_loan.drop(columns=['Credit_History'])
X2_test_loan = X_test_loan.drop(columns=['Credit_History'])
# END SOLUTION
```

Problem 1.3.2 (1 point)

Finish the code below to create a new classifier fit to the new train and test data.

In [50]:
```
T2 = DecisionTreeClassifier(max_depth=3)
# write your code here
# BEGIN SOLUTION
T2.fit(X2_train_loan, y_train_loan)
# END SOLUTION

print('Score on train:', T2.score(X2_train_loan, y_train_loan))
print('Score on test:', T2.score(X2_test_loan, y_test_loan))

fig, ax = plt.subplots(1, figsize = (20, 20))
p = plot_tree(T2, filled = True, feature_names = X2_train_loan.columns)
```

```
Score on train: 0.714975845410628
Score on test: 0.6634615384615384
```



Problem 1.3.3 (1 point)

How do the new scores for the train and test data compare to the previous scores that you printed? Explain why you might have expected this.

Both the new train set accuracy and test set accuracy are less than those of the previous values. This is expected because we lost an important feature, Credit_History<=0.5. What's interesting is that previously, test accuracy was higher than train accuracy for a max_tree_depth of 3, but without Credit_History<=0.5, test accuracy is less than train accuracy.

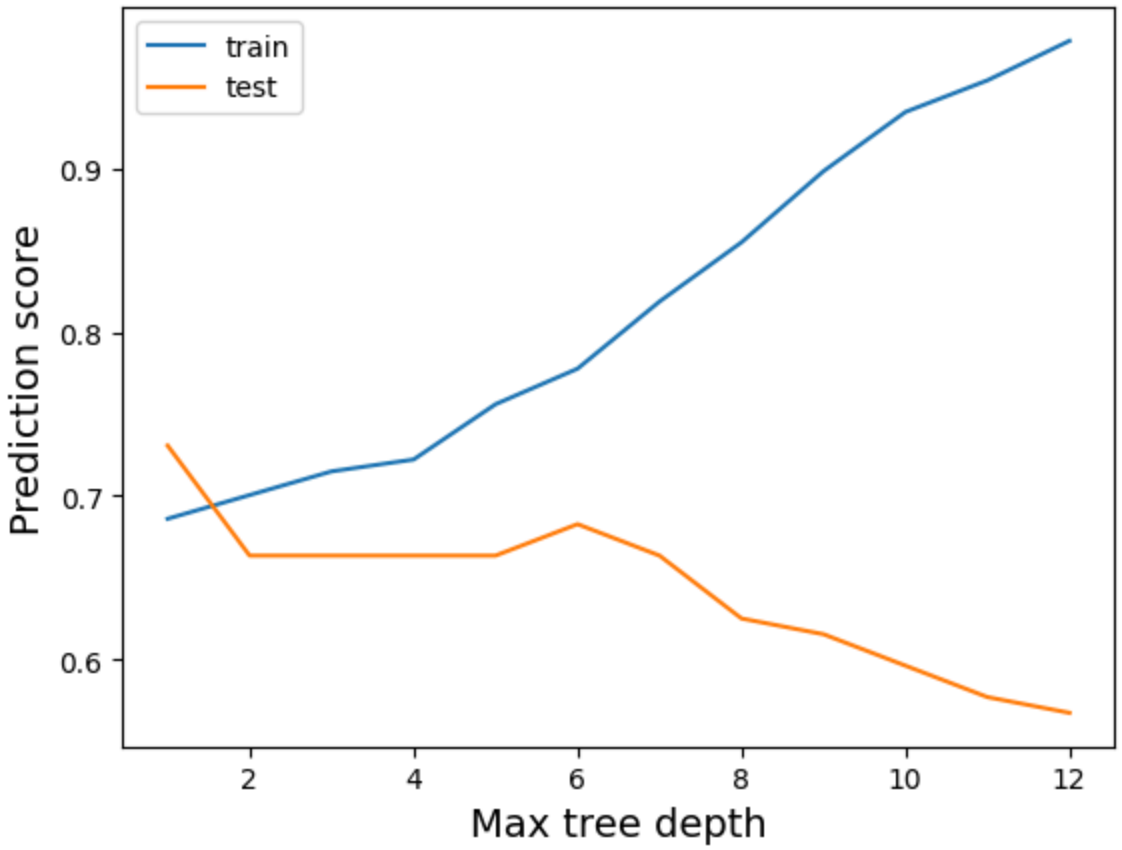
Problem 1.3.4 (1 point)

Once again, finish the code to gather prediction scores at different max tree depths.

In [51]:
```
train_scores=[]
test_scores=[]
```

```
r=range(1,13)
for i in r:
    T = DecisionTreeClassifier(max_depth=i)
    # write your code here
    # BEGIN SOLUTION
    T.fit(X2_train_loan, y_train_loan)
    # END SOLUTION
    train_scores.append(T.score(X2_train_loan, y_train_loan))
    test_scores.append(T.score(X2_test_loan, y_test_loan))
fig, ax = plt.subplots(1)
sns.lineplot(x=r,y=train_scores,label='train')
sns.lineplot(x=r,y=test_scores,label='test')
ax.set_ylabel('Prediction score', fontsize=14)
ax.set_xlabel('Max tree depth', fontsize=14)
ax.legend()
```

Out[51]: <matplotlib.legend.Legend at 0x7f16eedb6030>



Probelm 1.3.5 (2 points)

You should notice that increasing the tree depth dramatically improves the prediction score for the train set. This is because without `Credit_History`, there is no one primary feature correlated to the outcome, so we need more features to explain the loan decision data. At roughly what tree depth is the predictor able to predict as well for the train data as the previous predictor did using just `Credit_History`? (i.e. > 80%)?

At a max tree depth of 10, the new model's train prediction score is roughly as good as the previous model's train prediction score.

Problem 1.3.6 (3 points)

You should see that for the test data, the predictor never gets near to the 80% mark. No matter how deep the tree, the prediction score for the test data does not improve the same way as it did for the train data. Why is this?

This is due to the bias-variance tradeoff. When a model gets more complex (i.e., more features, higher tree depth), it could perform better and better on the train set, but when it does so well to the point where it "memorizes" the data, it cannot perform as well on data it has not seen before due to the loss of the ability to generalize across both seen and unseen distributions of data.

Problem 2. Parkinsons progression detection (25 points)

We will reproduce some of the qualitative results from the following paper, which you should find on canvas:

> A. Tsanas, M. A. Little, P. E. McSharry and L. O. Ramig, "Accurate Telemonitoring of Parkinson's Disease Progression by Noninvasive Speech Tests," in IEEE Transactions on Biomedical Engineering, vol. 57, no. 4, pp. 884–893, April 2010, doi: 10.1109/TBME.2009.2036000.

The Parkinsons Telemonitoring dataset was downloaded from https://archive.ics.uci.edu/dataset/189/parkinsons+telemonitoring.

It is not necessary for you to read or understand the paper in order to complete this problem. You only need to look at the Tables and Figures in the paper that I point to.

For context, this dataset is composed of a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease recruited to a six-month trial of a telemonitoring device for remote symptom progression monitoring. The recordings were automatically captured in the patient's homes.

Physical test observations are mapped to a metric specifically designed to follow disease progression, typically the unified Parkinson's disease rating scale (UPDRS) that reflects the presence and severity of symptoms (but does not quantify their underlying causes). For untreated patients, the **total UPDRS spans the range 0–176**, with 0 representing healthy state and 176 representing total disabilities, and consists of three sections: 1) mentation, behavior, and mood; 2) activities of daily living; and 3) motor. The **motor UPDRS ranges from 0 to 108**, with 0 denoting symptom free and 108 denoting severe motor impairment, and encompasses tasks such as speech, facial expression, tremor, and rigidity.

Columns in the table contain subject number, subject age, subject gender, time interval from baseline recruitment date, motor UPDRS, total UPDRS, and 16 biomedical voice measures. Each row corresponds to one of 5,875 voice recording from these individuals. **The main aim is to predict the motor and total UPDRS scores from the 16 voice measures.**

In [3]:
```
parkinsons = pd.read_csv('parkinsons_updrs.csv')
parkinsons
```

Out[3]:

	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP
0	1	72	0	5.6431	28.199	34.398	0.00662	0.000034	0.00401
1	1	72	0	12.6660	28.447	34.894	0.00300	0.000017	0.00132
2	1	72	0	19.6810	28.695	35.389	0.00481	0.000025	0.00205
3	1	72	0	25.6470	28.905	35.810	0.00528	0.000027	0.00191
4	1	72	0	33.6420	29.187	36.375	0.00335	0.000020	0.00093
...	...	...	...	...	...	...	...	...	...
5870	42	61	0	142.7900	22.485	33.485	0.00406	0.000031	0.00167
5871	42	61	0	149.8400	21.988	32.988	0.00297	0.000025	0.00119
5872	42	61	0	156.8200	21.495	32.495	0.00349	0.000025	0.00152
5873	42	61	0	163.7300	21.007	32.007	0.00281	0.000020	0.00128
5874	42	61	0	170.7300	20.513	31.513	0.00282	0.000021	0.00135

5875 rows × 22 columns

Problem 2.1 (2 points) Concepts of machine learning

Given the data format and the description above,

- Is this more of a **regression** problem or a **classification** problem? Pick one closest answer.
- What do the **samples** correspond to, and how many are there?
- What are the **target variables (labels, y)**, and how many are there?
- What are the **predictor variables (features, X)**, and how many are there?

1. This is more of a regression problem because we are predicting the value of a continuous numeric variable using features.
2. The samples correspond to each row, and there are 5875 samples.
3. The target variables are total_UPDRS and motor_UPDRS, but due to the high multicollinearity between motor_UPDRS and total_UPDRS, one of them should be dropped, or they should be merged as one target.
4. The initial predictor variables are Jitter(%), Jitter(Abs), Jitter:RAP, Jitter:PPQ5, Jitter:DDP, Shimmer, Shimmer(dB), Shimmer:APQ3, Shimmer:APQ5, Shimmer:APQ11, Shimmer:DDA, NHR, HNR, RPDE, DFA, PPE, but we might need to consolidate or drop some features after testing for multicollinearity.

Problem 2.2 (6 points) Exploring correlations

Open the paper pdf file and scroll down to page 887, Table 1. It looks scary, but all it's doing is calculating the **correlations** between each of the voice features with the UPDRS scores. We'll recreate this table step by step.

Step 1 (1 point)

Drop the columns `subject#`, `age`, `sex`, `test_time` and save the resulting table to a new variable `parkinsons_small`. Your new table should be of size (5875, 18).

In [4]:
```
# Step 1
# write your code here
# BEGIN SOLUTION
parkinsons_small = parkinsons.drop(columns=["subject#", "age", "sex", "test_time"])
for index, column_name in enumerate(parkinsons_small.columns):
    print(f"Column Index: {index}, Column Name: {column_name}")
# END SOLUTION
print(parkinsons_small.shape)
parkinsons_small.head()
```

```
Column Index: 0, Column Name: motor_UPDRS
Column Index: 1, Column Name: total_UPDRS
Column Index: 2, Column Name: Jitter(%)
Column Index: 3, Column Name: Jitter(Abs)
Column Index: 4, Column Name: Jitter:RAP
Column Index: 5, Column Name: Jitter:PPQ5
Column Index: 6, Column Name: Jitter:DDP
Column Index: 7, Column Name: Shimmer
Column Index: 8, Column Name: Shimmer(dB)
Column Index: 9, Column Name: Shimmer:APQ3
Column Index: 10, Column Name: Shimmer:APQ5
Column Index: 11, Column Name: Shimmer:APQ11
Column Index: 12, Column Name: Shimmer:DDA
Column Index: 13, Column Name: NHR
Column Index: 14, Column Name: HNR
Column Index: 15, Column Name: RPDE
Column Index: 16, Column Name: DFA
Column Index: 17, Column Name: PPE
(5875, 18)
```

Out[4]:

	motor_UPDRS	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PPQ5	Jitter:DDP	Shimmer
0	28.199	34.398	0.00662	0.000034	0.00401	0.00317	0.01204	0.02565
1	28.447	34.894	0.00300	0.000017	0.00132	0.00150	0.00395	0.02024
2	28.695	35.389	0.00481	0.000025	0.00205	0.00208	0.00616	0.01675
3	28.905	35.810	0.00528	0.000027	0.00191	0.00264	0.00573	0.02309
4	29.187	36.375	0.00335	0.000020	0.00093	0.00130	0.00278	0.01703

Step 2 (0 point, it's just copy pasting)

Copy and run the following code, which calculates the correlation between each pairs of columns in `parkinsons_small`. Your new table should be of size (18, 18).

```
corrs = parkinsons_small.corr(method='spearman')
print(corrs.shape)
corrs
```

How to read the table: The correlation between `Jitter(%)` and `total_UPDRS` is 0.129237.

Out of scope for DATA110 but just in case you're curious: There are a few mathematical definitions of how to calculate correlation, and Pearson's and Spearman's are two of the most popular methods. The main difference is that Spearman's cares about the ranking of the values, and not the raw values. So it has fewer assumptions on the data than Pearson's and can handle outliers better. But the way the correlation values are interpreted are the same.

In [5]:
```
# Step 2
# copy code here and run it
corrs = parkinsons_small.corr(method='spearman')
print(corrs.shape)
corrs
```

```
(18, 18)
```

Out[5]:

	motor_UPDRS	total_UPDRS	Jitter(%)	Jitter(Abs)	Jitter:RAP	Jitter:PPQ5	Jitter:
motor_UPDRS	1.000000	0.957818	0.127905	0.072467	0.106679	0.120092	0.10
total_UPDRS	0.957818	1.000000	0.129237	0.104178	0.109214	0.118347	0.10
Jitter(%)	0.127905	0.129237	1.000000	0.902065	0.956166	0.959100	0.95
Jitter(Abs)	0.072467	0.104178	0.902065	1.000000	0.821846	0.886542	0.82
Jitter:RAP	0.106679	0.109214	0.956166	0.821846	1.000000	0.948001	0.99
Jitter:PPQ5	0.120092	0.118347	0.959100	0.886542	0.948001	1.000000	0.94
Jitter:DDP	0.106731	0.109251	0.956176	0.821845	0.999996	0.948016	1.00
Shimmer	0.136186	0.137550	0.652681	0.624907	0.645059	0.686685	0.64
Shimmer(dB)	0.140127	0.139915	0.673114	0.634172	0.657433	0.698519	0.65
Shimmer:APQ3	0.113784	0.119909	0.616051	0.579700	0.632204	0.657140	0.63
Shimmer:APQ5	0.121267	0.124940	0.622316	0.606298	0.616982	0.666872	0.61
Shimmer:APQ11	0.163761	0.161151	0.635287	0.634621	0.598533	0.668118	0.59
Shimmer:DDA	0.113785	0.119912	0.616048	0.579697	0.632201	0.657138	0.63
NHR	0.135626	0.143972	0.797751	0.745656	0.745099	0.752913	0.74
HNR	-0.158061	-0.162284	-0.756633	-0.756796	-0.726644	-0.790342	-0.72
RPDE	0.117980	0.149926	0.529491	0.634311	0.445449	0.510383	0.44
DFA	-0.131377	-0.141538	0.440976	0.494995	0.430145	0.480241	0.43
PPE	0.163229	0.155236	0.845357	0.804174	0.772361	0.843826	0.77

Step 3 (2 points)

Since we're only interested in the correlations with the UPDRS scores,

1. Keep only the columns `motor_UPDRS`, `total_UPDRS` and save the new table to `corrs_table1`.
2. Within the table `corrs_table1`, drop the rows with index `motor_UPDRS`, `total_UPDRS` and save the new table to `corrs_table1` (overwrite it).

In [6]:
```
# Step 3
# write your code here
# BEGIN SOLUTION
corrs_table1 = corrs[['motor_UPDRS', 'total_UPDRS']]
corrs_table1 = corrs_table1.drop(index=['motor_UPDRS', 'total_UPDRS'])
# END SOLUTION
print(corrs_table1)
```

	motor_UPDRS	total_UPDRS
Jitter(%)	0.127905	0.129237
Jitter(Abs)	0.072467	0.104178
Jitter:RAP	0.106679	0.109214
Jitter:PPQ5	0.120092	0.118347
Jitter:DDP	0.106731	0.109251
Shimmer	0.136186	0.137550
Shimmer(dB)	0.140127	0.139915
Shimmer:APQ3	0.113784	0.119909
Shimmer:APQ5	0.121267	0.124940
Shimmer:APQ11	0.163761	0.161151
Shimmer:DDA	0.113785	0.119912
NHR	0.135626	0.143972
HNR	-0.158061	-0.162284
RPDE	0.117980	0.149926
DFA	-0.131377	-0.141538
PPE	0.163229	0.155236

Step 4 (3 points) Interpretation

Compare what you have with the numbers in Table 1, and check if they are **roughly** the same.

Note: The numbers won't be exactly the same, because 5923 recordings are analyzed in the paper, but we only have access to 5875 of them.

Question: What are your interpretations of these values? Do any of the biomedical voice variables seem highly correlated with the Parkinson's scores? Which variable do you think would be the most important in predicting UPDRS?

Based on the Pearson's correlation coefficients between each of our predictors and motor_UPDRS and total_UPDRS, none of them seem to be highly linearly correlated with Parkinson's scores. Intuitively, though, Shimmer:APQ11 not only has the highest correlation but also makes intuitive sense because shimmer measures amplitude (loudness) variation in the voice, which is inversely correlated with vocal stability and tends to worsen as Parkinson’s progresses.

Problem 2.3 (6 points) Prepare data for ML

Now comes the actual prediction! We will train and test a **linear regression** model to predict motor_UPDRS from the 16 voice variables. In other words, we are assuming that the motor_UPDRS scores are **approximately** equal to

c1*Jitter(%) + c2*Jitter(Abs) + c3*Jitter:RAP + ... + c16*PPE

for some set of coefficients (c1, c2, c3, ..., c16). And when we train or fit a linear regression model, we are looking for the best set of coefficients that describes the training dataset. When we test the model, we compare the model's predictions with the actual motor_UPDRS scores in the test dataset. If any of this sounds foreign to you, now would be a good time to review the past few lectures.

Problem 2.3.1 (2 points)

In an ideal world, we would train our model on the entire dataset, collect data from new patients, then test it on the new data. However, this is obviously not feasible in this case. In cases like this, most people randomly split the *existing* samples into train and test, and "pretend" like the samples in the test set are actually coming from future patients.

We can split the data by patients or by recordings. The authors of this paper chose to split by recordings.

Fill in the blank such that this sentence describes what the code does:

```
We will randomly put __1__% of the ___2____ into the __3___ set, and put
the remaining __2_____ into the ____3___ set.
```

- Options for 1: number between 0 and 100
- Options for 2: rows or columns
- Options for 3: train or test

The corresponding code:

```
train = parkinsons_small.sample(frac=0.9)
test = parkinsons_small.drop(index=train.index)
```
And then copy and run the code.

We will randomly put 90% of the rows into the training set, and put the remaining 10% into the test set.

In [7]:
```
# copy code here and run it
train = parkinsons_small.sample(frac=0.9)
test = parkinsons_small.drop(index=train.index)
```

Problem 2.3.2 (4 points)

Given your knowledge of what predictor variables and target variables are, create the following four variables: `X_train`, `y_train`, `X_test`, `y_test`. They are derived from tables `train` and `test`.

Here's one way to check your answer:

```
print(X_train.shape, y_train.shape)
    (5288, 16) (5288,)
print(X_test.shape, y_test.shape)
    (587, 16) (587,)
```

In [8]:
```
# write your code here
# BEGIN SOLUTION
y_train = train['motor_UPDRS']
X_train = train.drop(columns=['motor_UPDRS', 'total_UPDRS'])
y_test = test['motor_UPDRS']
X_test = test.drop(columns=['motor_UPDRS', 'total_UPDRS'])
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
# END SOLUTION
```

```
(5288, 16) (5288,)
(587, 16) (587,)
```

Problem 2.4 (2 points) Train and test model

Copy the code, fill in the blanks (i.e., call with appropriate parameters), and run the code in the next cell.

```
from sklearn.linear_model import LinearRegression

lr = LinearRegression(fit_intercept=False)
lr.fit(..., ...)
print(lr.score(..., ...))

coefs = pd.DataFrame(lr.coef_,
                     index=lr.feature_names_in_,
                     columns=['Motor UPDRS LR coefficients'])
coefs
```

In [9]:

```
# Copy the skeleton code here and complete it
from sklearn.linear_model import LinearRegression
from scipy import stats

lr = LinearRegression(fit_intercept=False)
lr.fit(X_train, y_train)
print(lr.score(X_test, y_test))

coefs = pd.DataFrame(lr.coef_,
                     index=lr.feature_names_in_,
                     columns=['Motor UPDRS LR coefficients'])
coefs

# --- build the design matrix (with intercept) ---
X = np.asarray(X_train)
y = np.asarray(y_train).reshape(-1,)

n, p = X.shape
X_design = np.c_[np.ones(n), X]          # add column of ones
beta_hat = np.r_[lr.intercept_, lr.coef_]  # [intercept, coefs...]

# --- residual variance (sigma^2) and covariance of betas ---
y_hat = lr.predict(X_train)
resid = y - y_hat
df = n - (p + 1)                         # parameters include intercept
sigma2 = (resid @ resid) / df

XtX_inv = np.linalg.pinv(X_design.T @ X_design)  # pinv is safer than inv
var_beta = sigma2 * np.diag(XtX_inv)
se_beta = np.sqrt(var_beta)

# --- t-stats and two-sided p-values ---
t_stats = beta_hat / se_beta
p_vals = 2 * (1 - stats.t.cdf(np.abs(t_stats), df=df))

# tidy table
names = ["Intercept"] + list(getattr(lr, "feature_names_in_", [f"x{i}" for i in range(p)
summary = pd.DataFrame({
    "coef": beta_hat,
    "std_err": se_beta,
    "t": t_stats,
    "p_value": p_vals
}, index=names)

print(summary)
```

```
0.02390335690007206
                       coef       std_err          t       p_value
Intercept          0.000000      2.521889   0.000000  1.000000e+00
Jitter(%)         97.027686    187.625151   0.517136  6.050830e-01
Jitter(Abs)   -81584.093800   8105.505652 -10.065269  0.000000e+00
Jitter:RAP    -48584.524320  40093.481068  -1.211781  2.256505e-01
Jitter:PPQ5     -209.137901    160.071729  -1.306526  1.914307e-01
Jitter:DDP     16343.653521  13366.133792   1.222766  2.214728e-01
Shimmer           56.565437     54.693352   1.034229  3.010767e-01
Shimmer(dB)        2.329925      4.115452   0.566141  5.713222e-01
Shimmer:APQ3  -16952.073892  40238.703411  -0.421288  6.735621e-01
Shimmer:APQ5    -170.024710     46.607777  -3.647990  2.668539e-04
Shimmer:APQ11    101.718809     20.537240   4.952896  7.539964e-07
Shimmer:DDA     5652.873667  13412.890087   0.421451  6.734431e-01
NHR               -0.306170      5.169696  -0.059224  9.527759e-01
HNR                0.484422      0.059162   8.188038  4.440892e-16
RPDE              17.499411      1.546964  11.312100  0.000000e+00
DFA               -8.660222      1.934888  -4.475827  7.771987e-06
PPE               32.641682      2.494931  13.083202  0.000000e+00
```

Problem 2.5 (4 points) Analyze model results

Problem 2.5.1 (2 points)

Given the R^2 score that is printed out, do you think this linear regression model is a good predictive model of motor_UPDRS or not?

The R^2 score of 0.024839642182812205 on the test set suggests that this linear regression model is not a good predictive model of modor_UPDRS. Perhaps a non-linear model may perform better on this dataset.

Problem 2.5.2 (2 points)

Out of the 16 features, which one(s) are deemed the most *important* or *useful* at predicting motor_UPDRS according to the model? Comment on both the magnitude and direction (negative vs. positive) of the coefficient values and what that means in the context of the problem we're interested in. Are they the same variables you mentioned in Problem 4.2 (based on correlations)? You are welcome to compare your numbers with the reulsts in Table 3 (first column), but not required.

Significant features = p-value < 0.05

Strongest, extremely significant predictors (p ≈ 0):

1. Jitter(Abs) (very negative, extremely strong t = –10.07)
2. HNR (positive, extremely strong t = 8.19)
3. RPDE (strong positive, t = 11.31)
4. DFA (moderate negative, t = –4.48)
5. PPE (very strong positive, t = 13.08)

Problem 2.6 (5 points) Repeating the experiment multiple times

You might say - but Harlin, this is only based on a specific train/test split. What if it just so happened that this set of samples leads to this result? How can we trust our interpretations are generally true?

You are absolutely right! That's why the authors of this paper repeated this 1000 times to create Table 3.

Fill in the missing parts of the code below (you shouldn't have to write new code, just copy paste from earlier problems).

```
coefs = pd.DataFrame([], columns=lr.feature_names_in_)

for i in range(1000):
    train = parkinsons_small.sample(frac=0.9)
    test = parkinsons_small.drop(train.index)

    #####
    # Add four lines of code here that creates X_train, y_train, X_test,
y_test.
    #####

    lr = LinearRegression(fit_intercept=False)
    # fill in the parameters here
    lr.fit(..., ...)

    coefs.loc[len(coefs)] = lr.coef_

coefs.mean()
```

Then tell us, **how did your analysis on the importance of each feature change?** If it didn't change, that's is also okay.

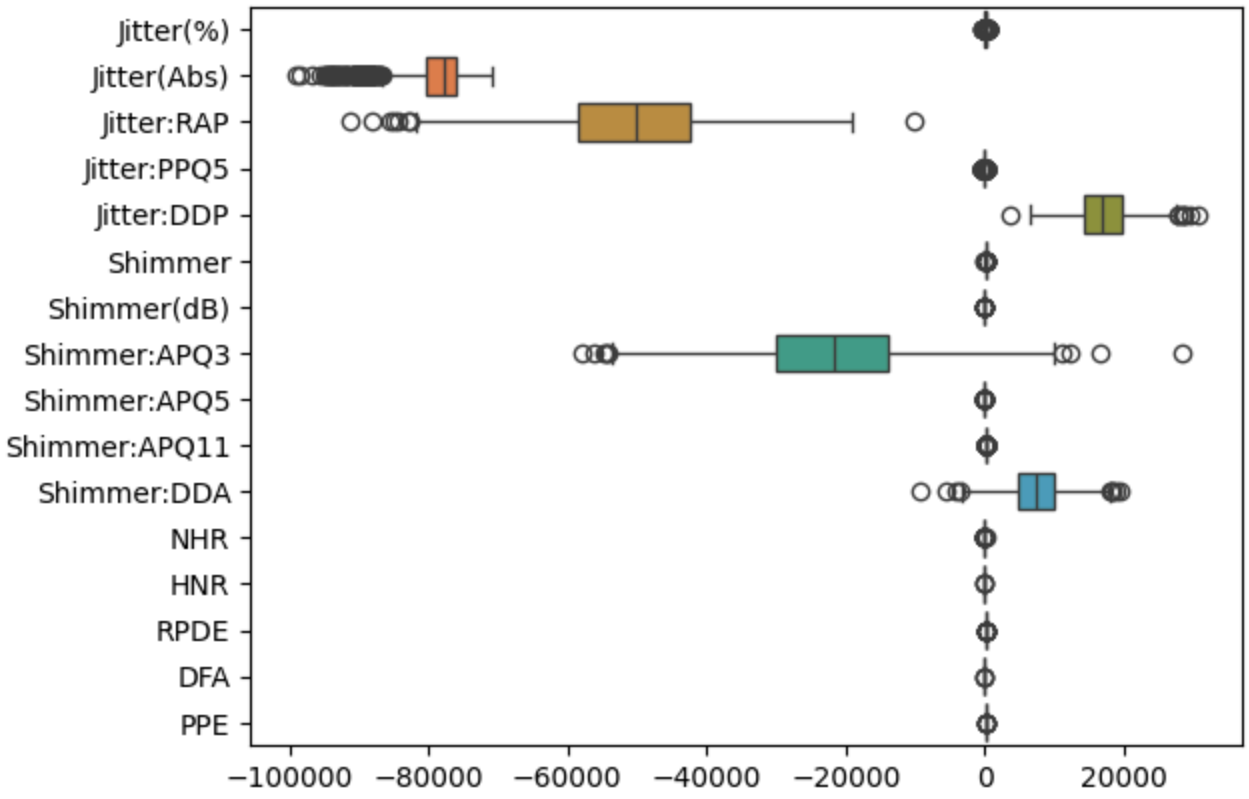
Note: Again, you won't get the same numbers as Table 3 because we don't have access to the full dataset, and there is some randomness involved. But my coefficients except for Jitter (%) looks *qualitatively* similar (I get a 40-ish number for Jitter (%) instead of a -80-ish number, which is odd).

In [10]:
```
# Copy the skeleton code here and complete it
coefs = pd.DataFrame([], columns=lr.feature_names_in_)

for i in range(1000):
    train = parkinsons_small.sample(frac=0.9)
    test = parkinsons_small.drop(train.index)

    #####
    # Add four lines of code here that creates X_train, y_train, X_test, y_test.
    #####
    y_train = train['motor_UPDRS']
    X_train = train.drop(columns=['motor_UPDRS', 'total_UPDRS'])
    y_test = test['motor_UPDRS']
    X_test = test.drop(columns=['motor_UPDRS', 'total_UPDRS'])

    lr = LinearRegression(fit_intercept=False)
    # fill in the parameters here
    lr.fit(X_train, y_train)

    coefs.loc[len(coefs)] = lr.coef_

coefs.mean()
```

Out[10]:
```
Jitter(%)           41.550567
Jitter(Abs)     -79005.497195
Jitter:RAP      -50813.315660
Jitter:PPQ5       -152.374426
Jitter:DDP       17089.508893
Shimmer             46.750458
Shimmer(dB)          0.206157
Shimmer:APQ3    -21670.193710
Shimmer:APQ5      -174.796753
Shimmer:APQ11      115.068945
Shimmer:DDA       7237.358288
NHR                  1.413816
HNR                  0.484790
RPDE                17.512505
DFA                 -8.678687
PPE                 32.738707
dtype: float64
```

In [11]:
```
# this should give you pretty box plots once you're done with the previous question.
sns.boxplot(coefs, orient='h')
```

Out[11]:
```
<Axes: >
```



Write your answer here: How did your analysis of the importance of each feature change?

After viewing the boxplot, my analysis of importance shifted from focusing purely on coefficient magnitude and significance to emphasizing stability and consistency across model runs. Features such as RPDE, PPE, and Shimmer:APQ11 remain strong, reliable predictors of motor_UPDRS, while others with large but highly variable coefficients (like Jitter(Abs) or Shimmer:APQ3) seem less dependable. In other words, importance is now judged by both effect size and robustness, not by magnitude alone.

In [ ]: