ML-A2-Group3

December 14, 2022

1 Iris Plant Species Classification

1.1 Analyze the data using the same techniques as for the last assignment.

Decide for yourself which and how to use the specific commands. Answer the following questions in the report and include figures supporting your answers:

1.1.1 Which classes exist? Are they (roughly) balanced?

[96]:
```
import matplotlib as plt
import pandas as pd
from sklearn import preprocessing

import utils

plt.rc('font', size=16)

df = pd.read_csv('iris.csv')
utils.ratio(df, 'Name')
```

[96]:

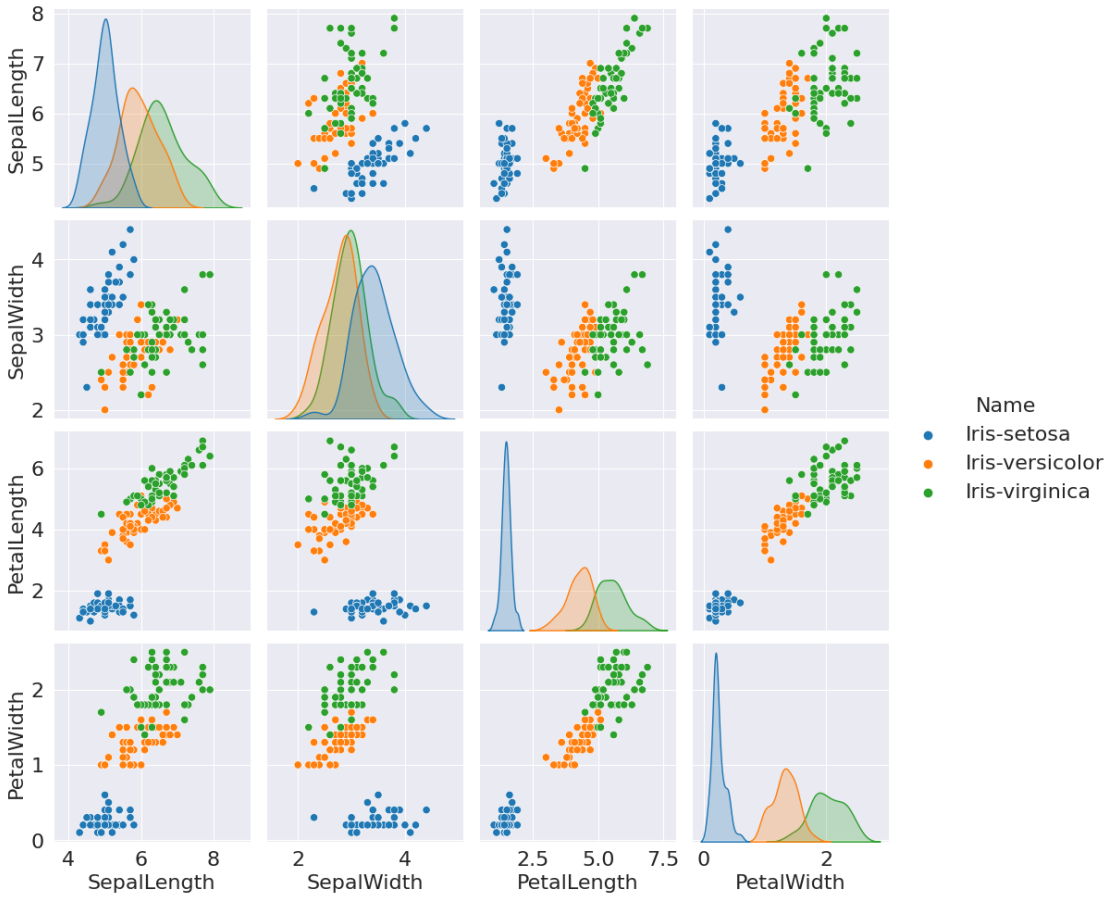
Name	samples	ratio
Iris-setosa	50	1.0
Iris-versicolor	50	1.0
Iris-virginica	50	1.0

Classes: Iris-setosa, Iris-versicolor, Iris-virginica They are perfectly balanced.

1.1.2 Which noteworthy trends of features and relations between features as well as features and Classes do you see?

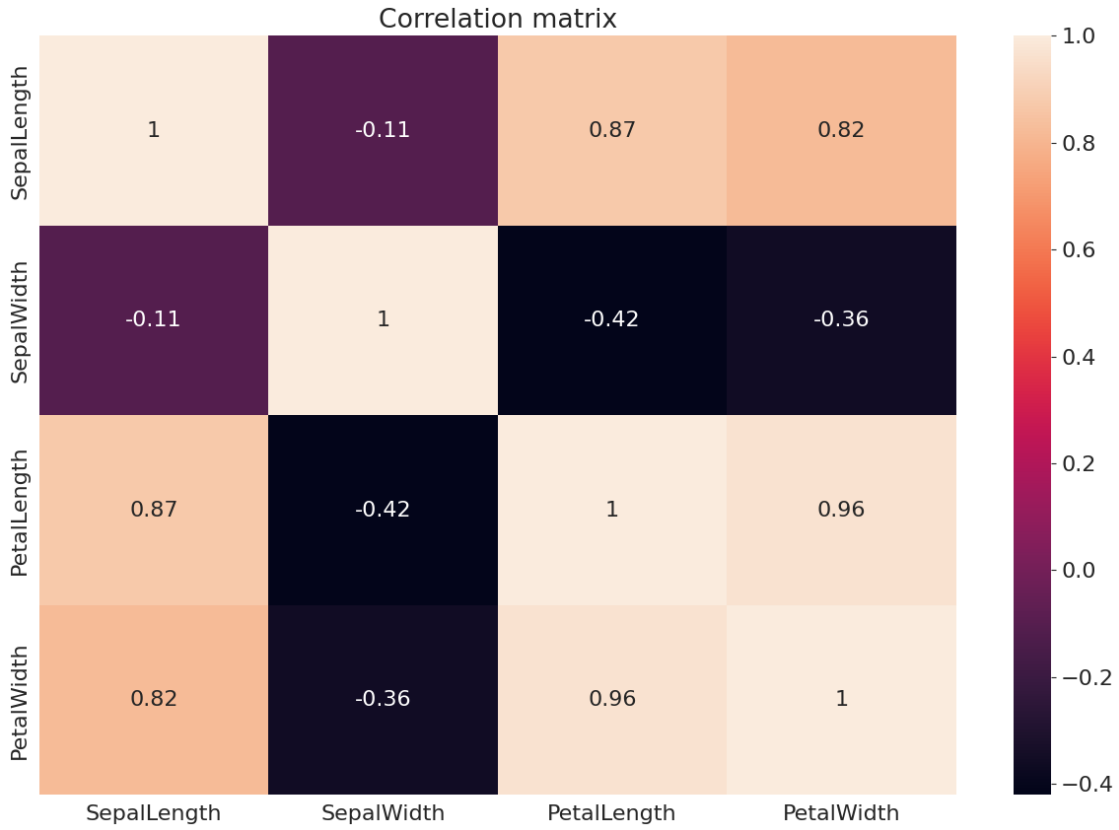
[97]:
```
import seaborn as sns

sns.pairplot(df, hue='Name')
```

[97]:
```
<seaborn.axisgrid.PairGrid at 0x7f91e373f700>
```



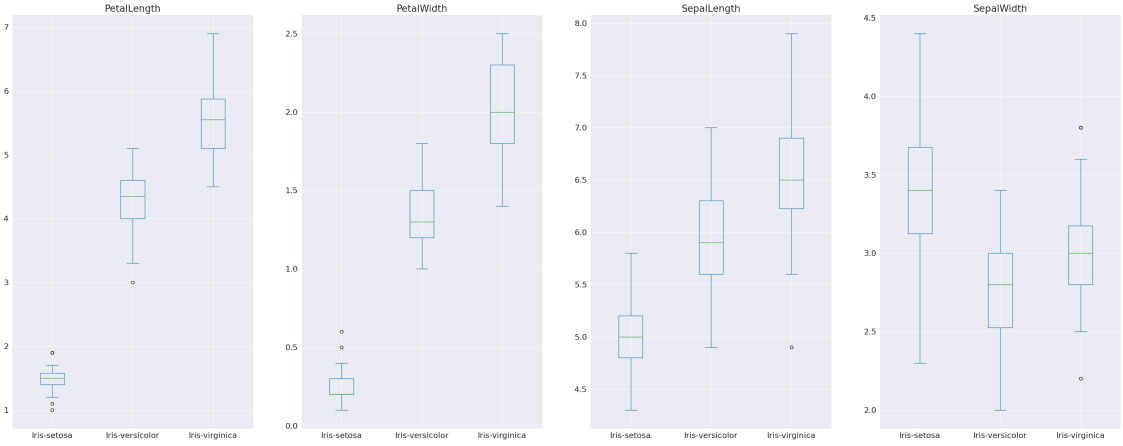
[98]:
```
from matplotlib import pyplot as plt
plt.figure(figsize=(15,10))
sns.heatmap(df.corr(), annot=True)
plt.title("Correlation matrix")
plt.show()
```

```
/tmp/ipykernel_29536/4023090694.py:3: FutureWarning: The default value of
numeric_only in DataFrame.corr is deprecated. In a future version, it will
default to False. Select only valid columns or specify the value of numeric_only
to silence this warning.
  sns.heatmap(df.corr(), annot=True)
```



[99]:
```
df.plot.box(by='Name',figsize=(40, 15), fontsize=16)
```

[99]:
```
PetalLength       AxesSubplot(0.125,0.11;0.168478x0.77)
PetalWidth     AxesSubplot(0.327174,0.11;0.168478x0.77)
SepalLength    AxesSubplot(0.529348,0.11;0.168478x0.77)
SepalWidth     AxesSubplot(0.731522,0.11;0.168478x0.77)
dtype: object
```



PetalLength and PetalWidth correlate well. PetalLength and SepalWidth correlate negatively. (see correlation matrix above) PetalLength and PetalWidth are well segmented and can be used to distinguish.

1.1.3 If you would need to distinguish the classes with those features, which features would you choose, any why?

PetalLength and PetalWidth because they don't overlap significantly. (see boxplot above) SepalLength and SepalWidth are not ideal to distinguish between flowers, since these features tend to overlap more, than any other feature.

1.2 Training

In order to classify the three different Iris plant species, set up your first ML toolchain including the following steps:

1.2.1 Data and Feature Preprocessing (if necessary and applicable)

Are there any outliers in the data which might need to be removed?

[100]:
```
X = df[['PetalLength', 'PetalWidth', 'SepalLength', 'SepalWidth']]
X.describe()
```

[100]:

	PetalLength	PetalWidth	SepalLength	SepalWidth
count	150.000000	150.000000	150.000000	150.000000
mean	3.758667	1.198667	5.843333	3.054000
std	1.764420	0.763161	0.828066	0.433594
min	1.000000	0.100000	4.300000	2.000000
25%	1.600000	0.300000	5.100000	2.800000
50%	4.350000	1.300000	5.800000	3.000000
75%	5.100000	1.800000	6.400000	3.300000
max	6.900000	2.500000	7.900000	4.400000

As we can see from the boxplot and the describe info, we do have some outliers that we could remove. Since we only have 150 samples it is not a good idea to simply remove the outliers (probably never is). A better approach would be to fix them to the mean / median or clip the outliers to some max value.

We decided to go for the second method and simply clip the values to the 99% and 1% quantile.

[101]:
```
y = df['Name']
q_max = X.quantile(.99)
q_min = X.quantile(.01)
# Outlier Removal

X.clip(lower=q_min, upper=q_max, axis=1, inplace=True)
X.describe()
```

```
/tmp/ipykernel_29536/57985973.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  X.clip(lower=q_min, upper=q_max, axis=1, inplace=True)
```

[101]:

	PetalLength	PetalWidth	SepalLength	SepalWidth
count	150.000000	150.000000	150.000000	150.000000
mean	3.758653	1.198667	5.842667	3.053347
std	1.760089	0.763161	0.823672	0.424997
min	1.149000	0.100000	4.400000	2.200000
25%	1.600000	0.300000	5.100000	2.800000
50%	4.350000	1.300000	5.800000	3.000000
75%	5.100000	1.800000	6.400000	3.300000
max	6.700000	2.500000	7.700000	4.151000

- Are there any missing values which need to be taken care of?

[102]:
```
# NaN
df.isnull().values.any()
```

[102]:
```
False
```

Do you need to apply any feature preprocessing steps? (e.g Normalization, Feature Deletion/Reduction/Addition) We do not need to apply normalization nor feature deletion but some models perform better on normalized data. With 150 samples feature deletion does not really provide any performance benefits, but we decided to do it anyway with sklearn.

[103]:
```
# Scaling
scaler = preprocessing.StandardScaler().fit(X, y)

X_scaled = scaler.transform(X)


X_scaled = pd.DataFrame(X_scaled, columns=["PetalLength", "PetalWidth",␣
 ↪"SepalLength", "SepalWidth"])
X_scaled
```

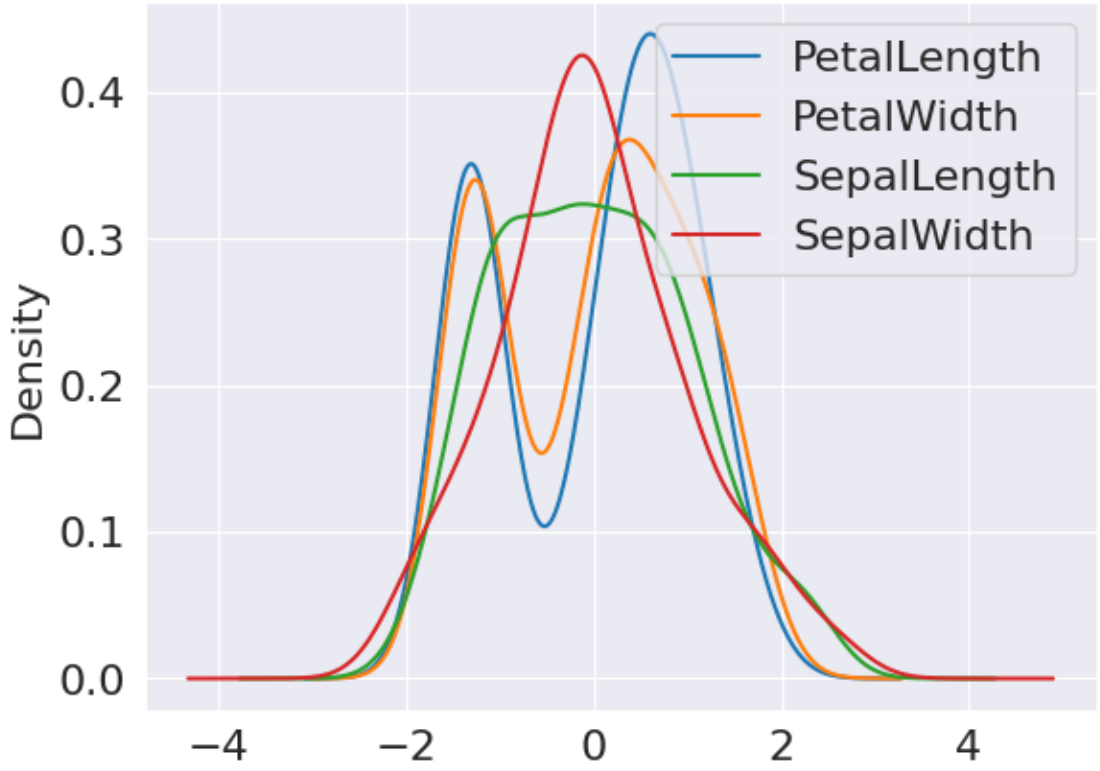
[103]:

	PetalLength	PetalWidth	SepalLength	SepalWidth
0	-1.344565	-1.312977	-0.904674	1.054478
1	-1.344565	-1.312977	-1.148302	-0.125943
2	-1.401571	-1.312977	-1.391931	0.346225
3	-1.287560	-1.312977	-1.513745	0.110141
4	-1.344565	-1.312977	-1.026488	1.290562
..	…	…	…	…
145	0.821649	1.447956	1.044354	-0.125943

```
146     0.707638    0.922064     0.557097   -1.306364
147     0.821649    1.053537     0.800726   -0.125943
148     0.935660    1.447956     0.435283    0.818394
149     0.764643    0.790591     0.069840   -0.125943

[150 rows x 4 columns]
```

[104]:
```
X_scaled.plot.density()
```

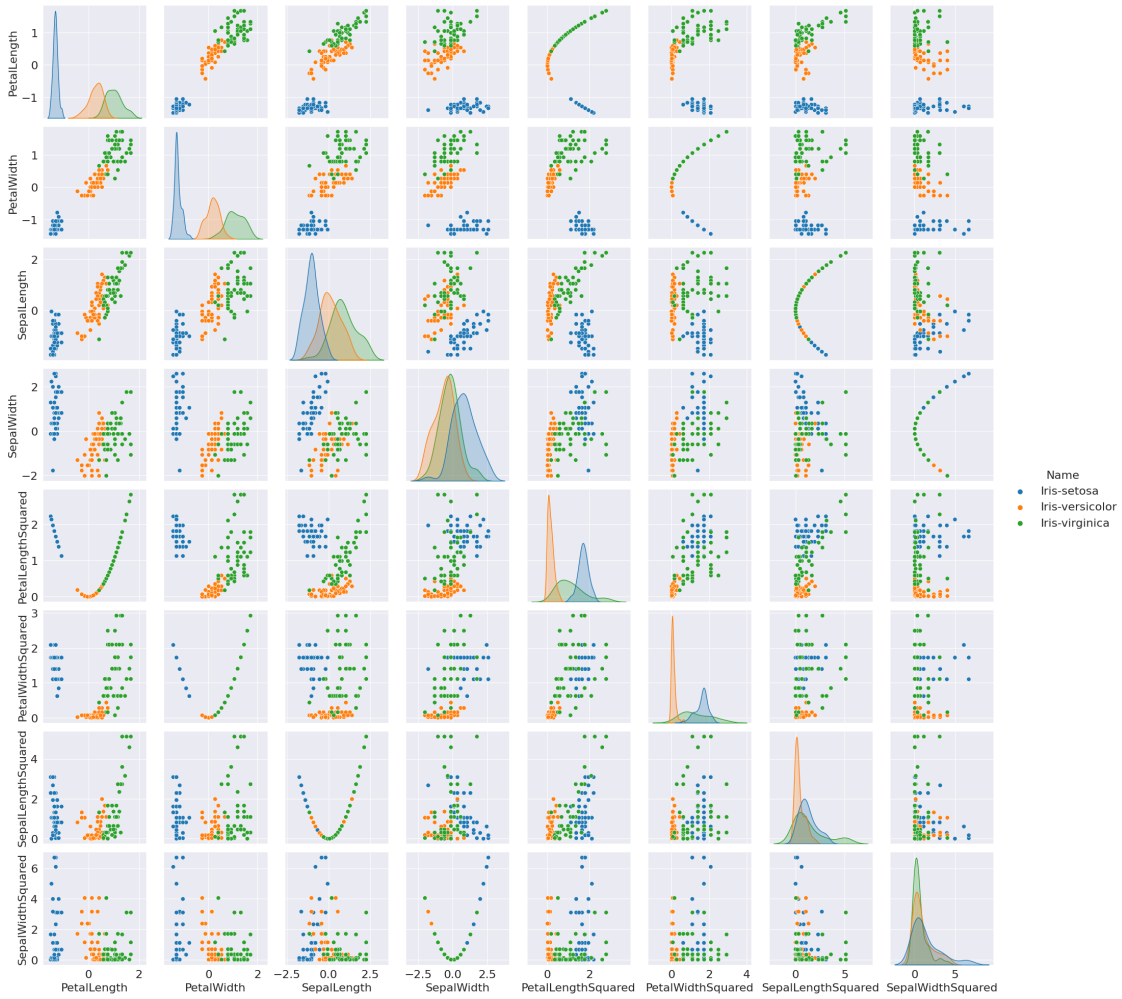
[104]: <AxesSubplot: ylabel='Density'>



We decided to add some features to see if we can gain any useful information.

[105]:
```
X_scaled["PetalLengthSquared"] = X_scaled["PetalLength"] *␣
  ↪X_scaled["PetalLength"]
X_scaled["PetalWidthSquared"] = X_scaled["PetalWidth"] * X_scaled["PetalWidth"]
X_scaled["SepalLengthSquared"] = X_scaled["SepalLength"] *␣
  ↪X_scaled["SepalLength"]
X_scaled["SepalWidthSquared"] = X_scaled["SepalWidth"] * X_scaled["SepalWidth"]

X_scaled["Name"] = y
sns.pairplot(X_scaled, hue="Name")
```

```
X_scaled.drop(columns=["Name"], inplace=True)  # remove target
```



- Are there any categorical features that need to be transformed so that it can be used for classification task?
  - No since all our features are numerical, we do not have any categorical features, besides the target feature `Name`.
- Do you think it makes sense to derive any more features from the given ones? Why/why not?
  - It could make sense, depending on the data. It is possible to generate information that can help a model perform better. Since we do not have a lot of samples we can definitely try to derive new features.
- Split up the dataset into a training and a separate held back test set in a clever way
  - Why is such a train/test split important?
    - A: So we can validate our model and check whether we just made a lookup table of our data. It's our last safety line and important to measure the performance of our model.
  - Which train/test split percentage do you choose and why?
    - A: we choose a 70/30 split since we do not have a lot of samples and want enough

data to validate our model and 70 / 30 % of 150 are integers.
  - Think about how can you make sure to include samples from all three classes in both datasets and why this is important.
    * A: If a class has no samples in our training data, the model can at best make a wild guess if a sample of that class is passed to the model. We ensured that every class is represented by using `sklearn.model_selection.train_test_split` and supplying it with the `stratify` parameter.

Feature Selection In the lecture we learned that feture selection like PCA or LDA should only be applied to the training data and not the test data, so we need to split our data now, we used a 30/70 split where we use 70% of our data for training and 30% for validation.

- Use an appropriate cross-validation setup for the training:
  - `X_train` and `y_train` represents our training data and `X_train` and `y_train` our held back test set.

[106]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, stratify=y,␣
↪test_size=0.30)  # 70/30 split
```

[143]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=4)
pca.fit(X_train)
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)
# X_train
```

1.2.2 Model Training

- Train different classification models to distinguish between the three Iris Plant Species:
  - Use the following models: k Nearest Neighbour, Decision Tree, Support Vector Machine

KNN

[108]:
```
from sklearn.model_selection import GridSearchCV
from sklearn import svm, neighbors, neural_network

knn = GridSearchCV(
    estimator= neighbors.KNeighborsClassifier(),
    param_grid= [{'n_neighbors': [3, 5, 7, 9], 'weights': ['uniform',␣
↪'distance'], 'leaf_size': [15, 20]}],
    scoring= "accuracy",
    cv= 3)

knn.fit(X_train, y_train)
knn.best_params_
```

[108]:
```
{'leaf_size': 15, 'n_neighbors': 3, 'weights': 'distance'}
```

Tree

[109]:
```
from sklearn.tree import DecisionTreeClassifier

tree = GridSearchCV(
    estimator= DecisionTreeClassifier(),
    param_grid= [{
        'splitter': ['best', 'random'],
        'max_depth': [10, 100, 1000],
        'criterion': ['gini', 'entropy', 'log_loss'],
        'class_weight': ['balanced']}],
        scoring= "accuracy",
        cv= 3
    )

tree.fit(X_train, y_train)
tree.best_params_
```

[109]:
```
{'class_weight': 'balanced',
 'criterion': 'gini',
 'max_depth': 100,
 'splitter': 'random'}
```

SVM

[110]:
```
from sklearn.utils.fixes import loguniform

svc = GridSearchCV(
    estimator= svm.SVC(),
    param_grid= [{
        'C': loguniform(0.1, 1, 100, 1000).rvs(20),
        'class_weight': ['balanced'],
        'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
        'gamma': loguniform(0.000035, 0.000245).rvs(20)
    }]
)

svc.fit(X_train, y_train)
svc.best_params_
```

[110]:
```
{'C': 979.5793790821483,
 'class_weight': 'balanced',
 'gamma': 8.987322444026307e-05,
 'kernel': 'linear'}
```

Neural Network playground

[111]:
```
nn = GridSearchCV(
    estimator=neural_network.MLPClassifier(max_iter=10000),
    param_grid= [{
        'hidden_layer_sizes': [6, 9, 12],
        'activation': ['identity', 'logistic', 'tanh', 'relu'],
        'solver': ['lbfgs', 'sgd', 'adam'],
        'learning_rate': ['constant', 'adaptive']
    }]
)

nn.fit(X_train, y_train)
nn.best_params_
```

[111]:
```
{'activation': 'relu',
 'hidden_layer_sizes': 9,
 'learning_rate': 'adaptive',
 'solver': 'lbfgs'}
```

- Use different hyperparameter settings for each model and explain why and how you chose them
  - We selected each hyperparameter by trying different combinations, and then using the best fitting hyperparameters.

1.2.3 Performance Estimates

- Estimate the models' performances on the held back test set:

[112]:
```
knn.score(X_test, y_test)
```

[112]:
```
0.9111111111111111
```

[113]:
```
tree.score(X_test, y_test)
```

[113]:
```
0.8666666666666667
```

[114]:
```
svc.score(X_test, y_test)
```

[114]:
```
0.9111111111111111
```

[115]:
```
nn.score(X_test, y_test)
```

[115]:
```
0.9333333333333333
```

- Compare the models with their hyperparameter settings with two different error/performance measures
- Why did you choose the specific error/performance measures?
  - We chose the build-in report feature of sklearn, since it includes different scoring algorithms and scores for each label

- What do they tell you?
  - It tells us how well a model performs on the held-back testset, for each label and overall

[116]:
```
from sklearn.metrics import classification_report

y_pred = knn.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      0.93      0.97        15
Iris-versicolor       0.82      0.93      0.87        15
 Iris-virginica       0.93      0.87      0.90        15

       accuracy                           0.91        45
      macro avg       0.92      0.91      0.91        45
   weighted avg       0.92      0.91      0.91        45
```

[117]:
```
y_pred = tree.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      0.93      0.97        15
Iris-versicolor       0.80      0.80      0.80        15
 Iris-virginica       0.81      0.87      0.84        15

       accuracy                           0.87        45
      macro avg       0.87      0.87      0.87        45
   weighted avg       0.87      0.87      0.87        45
```

[118]:
```
y_pred = svc.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        15
Iris-versicolor       0.79      1.00      0.88        15
 Iris-virginica       1.00      0.73      0.85        15

       accuracy                           0.91        45
      macro avg       0.93      0.91      0.91        45
   weighted avg       0.93      0.91      0.91        45
```

[119]:
```
y_pred = nn.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      0.93      0.97        15
Iris-versicolor       0.88      0.93      0.90        15
 Iris-virginica       0.93      0.93      0.93        15

       accuracy                           0.93        45
      macro avg       0.94      0.93      0.93        45
   weighted avg       0.94      0.93      0.93        45
```

- Which model performs best with which hyperparameter settings and why do you think it does that way?
  - The KNN and SVM Classifiers perform the best, because each Label is mostly cleanly seperated from the others.

[120]:
```
knn.best_params_
```

[120]:
```
{'leaf_size': 15, 'n_neighbors': 3, 'weights': 'distance'}
```

[121]:
```
svc.best_params_
```

[121]:
```
{'C': 979.5793790821483,
 'class_weight': 'balanced',
 'gamma': 8.987322444026307e-05,
 'kernel': 'linear'}
```

- Explain which model you would use in deployment and why
  - We would use the knn model, since it's the simplest model with the best score.

2 Boston House Price Prediction

2.1 Analyze the data using the same techniques as for the last assignment.

Decide for yourself which and how to use the specific commands. Answer the following questions in the report and include figures supporting your answers:

2.1.1 Which noteworthy trends of features and relations between features as well as features and regression target do you see?

- CRIM:
  - per capita crime rate by town
- ZN:
  - proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS:
  - proportion of non-retail business acres per town

- CHAS:
  - Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX:
  - nitric oxides concentration (parts per 10 million)
- RM:
  - average number of rooms per dwelling
- AGE:
  - proportion of owner-occupied units built prior to 1940
- DIS:
  - weighted distances to five Boston employment centres
- RAD:
  - index of accessibility to radial highways
- TAX:
  - full-value property-tax rate per 10,000
- PTRATIO:
  - pupil-teacher ratio by town
- B:
  - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT:
  - % lower status of the population
- MEDV (TAGET):
  - Median value of owner-occupied homes in $1000's

[122]:
```
import pandas as pd

df = pd.read_csv("housing.csv", sep="\s+",
                 names=["CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE",␣
 ↪"DIS", "RAD", "TAX", "PTRATIO", "B", "LSTAT", "MEDV"],
                 header=None)

df.describe() # list some statistics for the features in the dataset
```

[122]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	

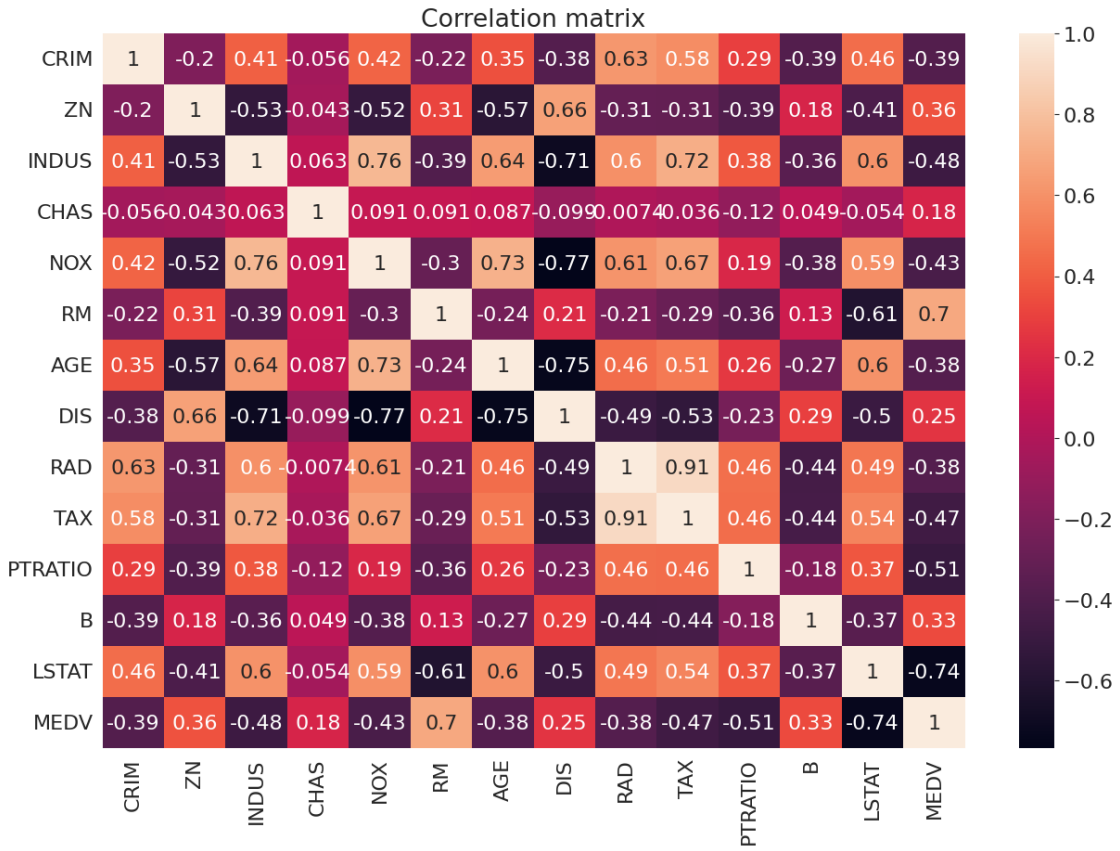
	AGE	DIS	RAD	TAX	PTRATIO	B	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032	
std	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864	
min	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000	

```
25%     45.025000    2.100175    4.000000  279.000000   17.400000  375.377500
50%     77.500000    3.207450    5.000000  330.000000   19.050000  391.440000
75%     94.075000    5.188425   24.000000  666.000000   20.200000  396.225000
max    100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

             LSTAT        MEDV
count  506.000000  506.000000
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

[123]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    int64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 55.5 KB
```

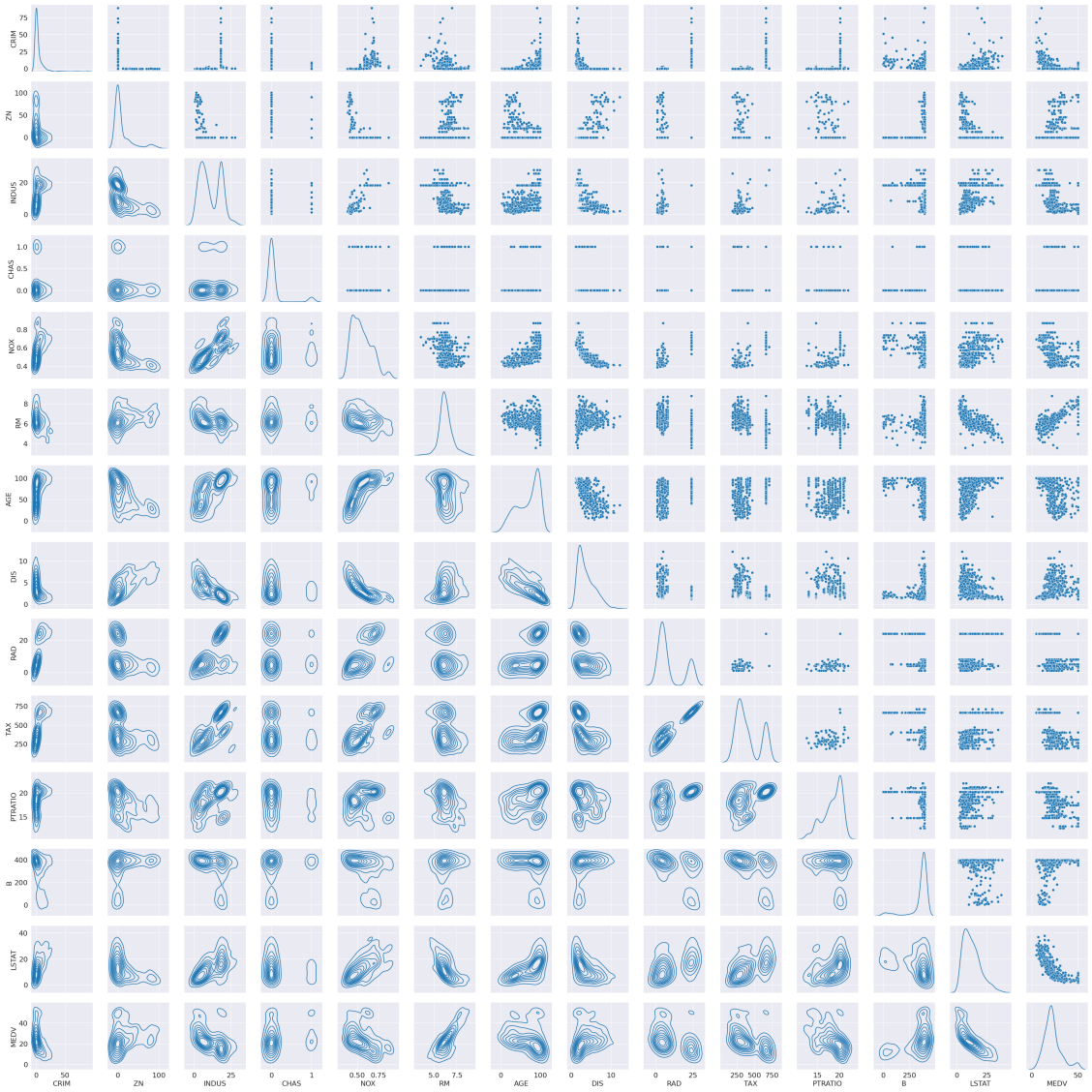
[124]:
```
from matplotlib import pyplot as plt
import seaborn as sns

plt.figure(figsize=(15,10))
sns.heatmap(df.corr(), annot=True)
plt.title("Correlation matrix")
plt.show()
```



It makes sense, that the `MEDV` correlates negatively with the `CRIM`, since demand for houses in areas with high crime rate would be lower than in those with lower rates. It also makes sense, that the prices in low crime rate areas fluctuate more than in those with higher crime rate, since this relation does not factor in other factors such as location.

[125]:
```
g = sns.PairGrid(df, diag_sharey=False)
g.map_upper(sns.scatterplot)
g.map_lower(sns.kdeplot)
g.map_diag(sns.kdeplot)
```

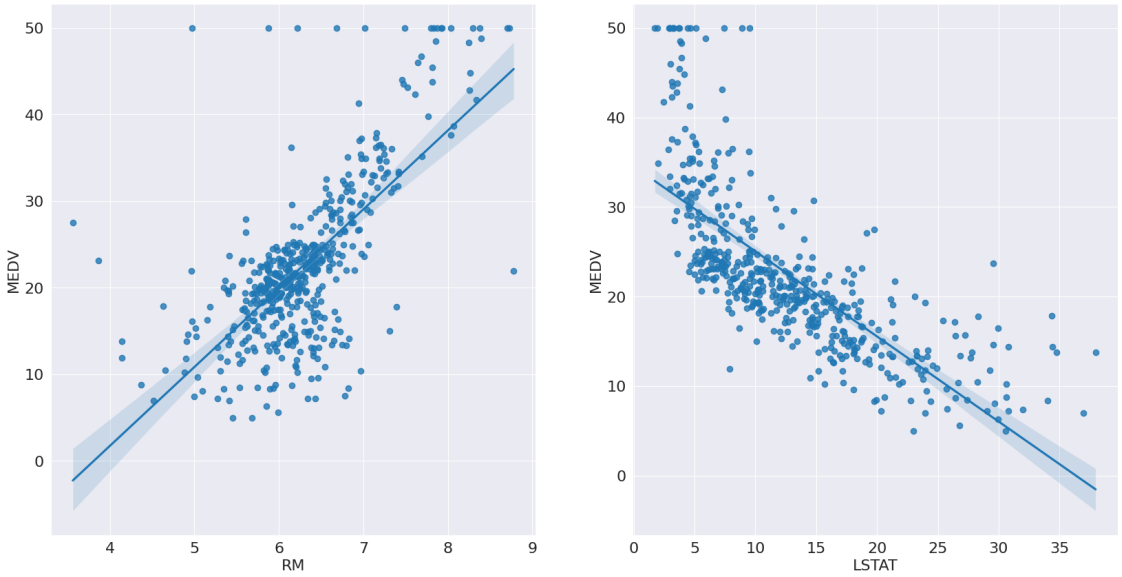
[125]: `<seaborn.axisgrid.PairGrid at 0x7f91d031f730>`



We think the most important features for our goal of predicting `MEDV` would be `RM` and `LSTAT` as they have the highest correlation with our target, though we can still see , that there are many outliers. The relationship between `LSTAT` and `MEDV` seems more like a curve as opposed to a linear function.

[126]:
```
fig, (ax1, ax2) = plt.subplots(ncols=2)
fig.set_size_inches(20, 10)
sns.regplot(df, x="RM", y="MEDV", ax=ax1)
sns.regplot(df, x="LSTAT", y="MEDV", ax=ax2)
```

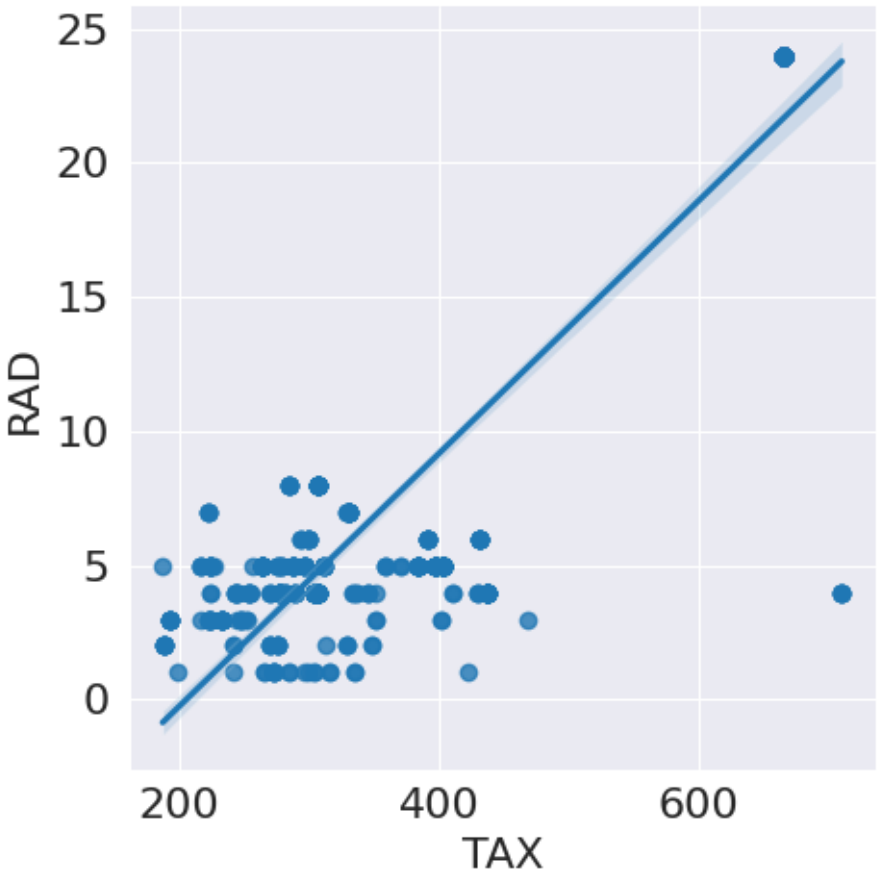
[126]:
```
<AxesSubplot: xlabel='LSTAT', ylabel='MEDV'>
```



We do not really understand how this pair is supposed to have a correlation of 91%

[127]:
```
sns.lmplot(df, x="TAX", y="RAD")
```

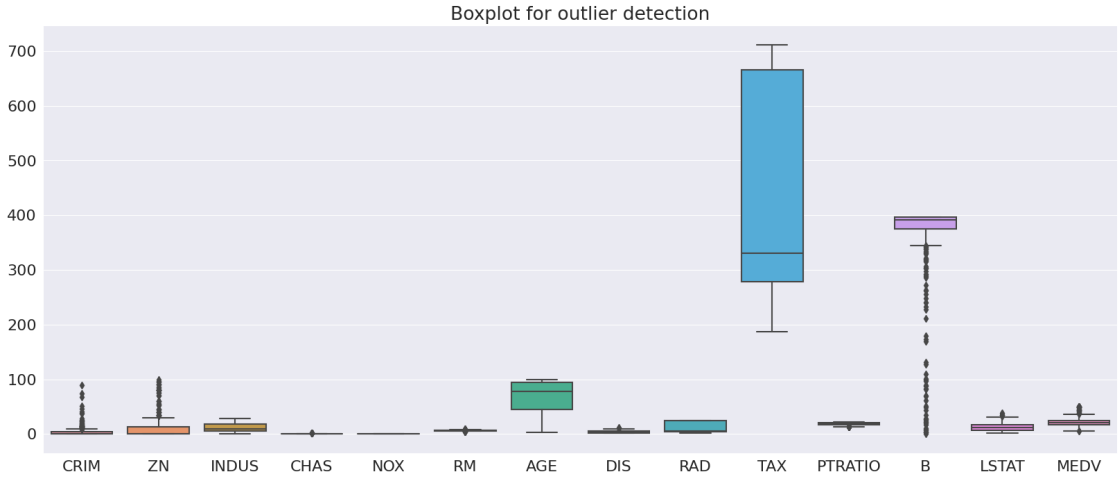
[127]:
```
<seaborn.axisgrid.FacetGrid at 0x7f91c8b24b50>
```



Outlier detection

[128]:
```
plt.figure(figsize=(20, 8))
sns.boxplot(df)
plt.title("Boxplot for outlier detection")
```

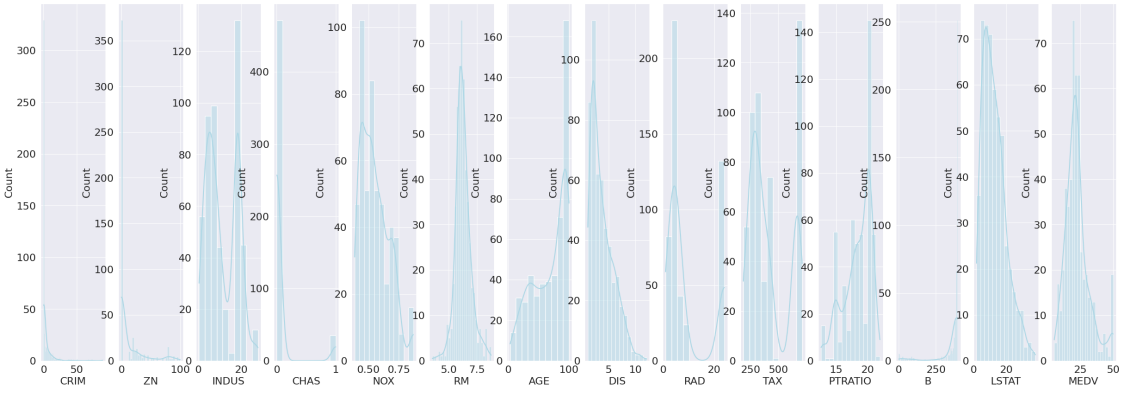
[128]:
```
Text(0.5, 1.0, 'Boxplot for outlier detection')
```



Distribution / Histograms By plotting the histograms and fitting a kde, we can see that features such as `RM` and `MEDV` appear to have a normal distribution, while features such as `LSTAT`, `BIS`, and `NOX` are skewed to the left.

[129]:
```
fig, ax =plt.subplots(1,len(df.columns),figsize=(30,10))
i = 0
for column in df.columns:
    sns.histplot(df[column], kde=True, ax=ax[i], color="lightblue")
    i += 1
fig.show()
```

```
/tmp/ipykernel_29536/1883304565.py:6: UserWarning: Matplotlib is currently using
module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```



2.1.2 Which features would you choose to train the regression models, any why?

We think the features that would make the most sense would be `LSTAT` and `RM` since they have the highest correlation with our target `MEDV` (>= 0.7). From the correlation matrix we can see that we should exclude either `TAX` or `RAD` since they have a really high correlation. The same goes for `DIS` and `AGE` as they have a high negative correlation.

2.2 Build up your ML toolchain for this regression problem similar to the one you did for the classification and again take care of the following points:

- Data and Feature Preprocessing (if necessary and applicable)
- Train/Test split

[130]:
```
y = df["MEDV"]
df.drop(columns=["MEDV"], inplace=True)
X = df
X.describe()
```

[130]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	

	AGE	DIS	RAD	TAX	PTRATIO	B	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032	
std	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864	
min	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000	
25%	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500	
50%	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000	
75%	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000	
max	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000	

	LSTAT
count	506.000000
mean	12.653063
std	7.141062
min	1.730000
25%	6.950000
50%	11.360000
75%	16.955000
max	37.970000

2.2.1 Outlier Removal

[131]:
```
q_max = X.quantile(.99)
q_min = X.quantile(.01)

X.clip(lower=q_min, upper=q_max, axis=1, inplace=True)
X.describe()
```

[131]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	3.375175	11.304348	11.118875	0.069170	0.554770	6.287106	
std	6.908970	23.112644	6.809112	0.253994	0.115773	0.678876	
min	0.013610	0.000000	1.253500	0.000000	0.398000	4.524450	
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	
max	41.370330	90.000000	25.650000	1.000000	0.871000	8.335000	

	AGE	DIS	RAD	TAX	PTRATIO	B	\
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	
mean	68.584506	3.778529	9.549407	407.794466	18.454743	356.715751	
std	28.127163	2.052652	8.707259	167.791388	2.154109	91.133441	
min	6.610000	1.206540	1.000000	188.000000	13.000000	6.730000	
25%	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500	
50%	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000	
75%	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000	
max	100.000000	9.222770	24.000000	666.000000	21.200000	396.900000	

	LSTAT
count	506.000000
mean	12.642073
std	7.074084
min	2.883000
25%	6.950000
50%	11.360000
75%	16.955000
max	33.918500

2.2.2 Scaling

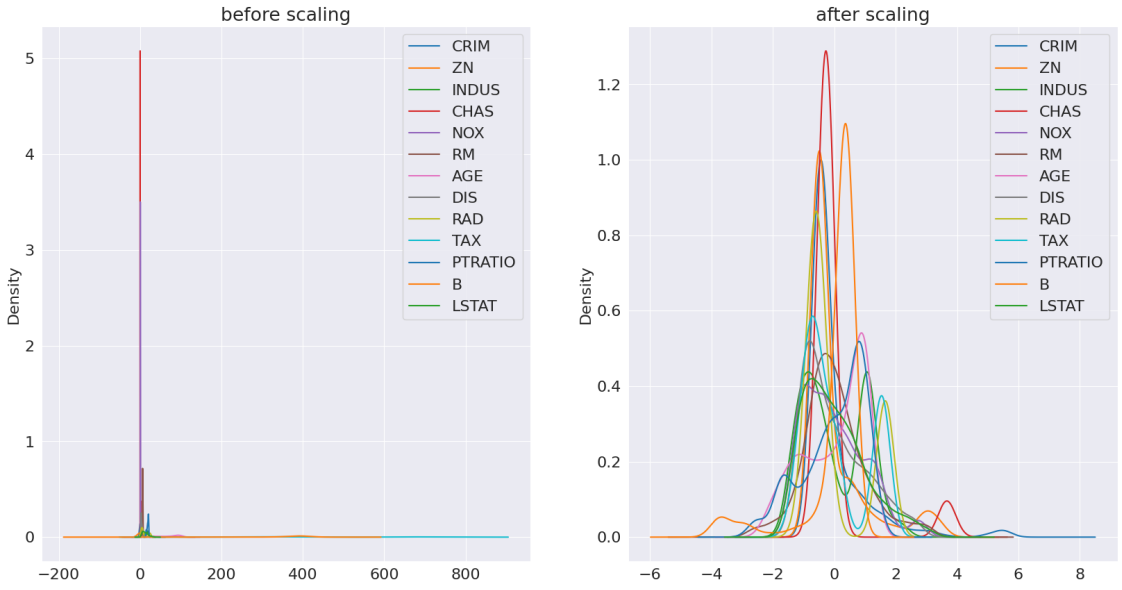
[144]:
```
from sklearn import preprocessing

scaler = preprocessing.StandardScaler().fit(X, y)

X_scaled = scaler.transform(X)
```

```
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)
# X_scaled
```

Density plots before/after

[133]:
```
fig, (ax1, ax2) = plt.subplots(ncols=2)
ax1.set_title("before scaling")

fig.set_size_inches(20, 10)
X.plot.density(ax=ax1)

ax2.set_title("after scaling")
X_scaled.plot.density(ax=ax2)
```

[133]:
```
<AxesSubplot: title={'center': 'after scaling'}, ylabel='Density'>
```



2.2.3 Feature processing:

We noticed that creating arbitrary features (the ones below) makes our model perform way worse. When we used the code below with a pca that used 15 components we got a score of around 0.5 with linear regression and negative scores with polynomial regression. We only achieved higher results with these features after increasing the number of components to 20, after which we scored around 0.79.

So we decided to not add any features and instead reduce the number of components for the pca.

[134]:
```
## Add arbitrary features: the square of each feature and 2 / feature
# for column in df.columns:
#     X_scaled[column + "_q"] = X_scaled[column] ** 2
```

```
#     X_scaled["2_div_" + column] = 2 / X_scaled[column]
# X_scaled.describe()
```

2.2.4 Create train and test split with sklearn

[146]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.
↪30)  # 70/30 split
```

2.2.5 Feature Reduction

Use PCA to reduce features, consider only the 5 most important pca components

[147]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=5) # if features are added, change n_components >= 20 to␣
↪achieve similar performance
pca.fit(X_train)
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)
# X_train
```

2.3 Training

- Use the following Regression models with different hyperparemter settings (where applicable) and an appropriate cross-validation setup for your training:
  - Linear Regression
  - Polynomial Regression
  - Logistic Regression
- Estimate the models' performances on the test set again with two different error/performance measurements
- Explain which model you would use in deployment and why

2.3.1 Linear Regression

[137]:
```
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.linear_model import LinearRegression, LogisticRegression,␣
↪SGDRegressor

linear_reg = LinearRegression()
best_score = -float("inf")
train_x = pd.DataFrame(X_train)
train_y = pd.DataFrame(y_train)
scores = []
for train_idx, test_idx in KFold(n_splits=15).split(X_train):
    kx_train, kx_test = train_x.iloc[train_idx], train_x.iloc[test_idx]
    ky_train, ky_test = train_y.iloc[train_idx], train_y.iloc[test_idx]
```

```
    m = LinearRegression()
    m.fit(kx_train, ky_train)
    current_score = m.score(kx_test, ky_test)
    scores.append(current_score)
    if current_score > best_score:
        best_score = current_score
        linear_reg = m
print("Linear Regression - Scores: ", scores)
print("Linear Regression - Best Score: ", best_score)
```

```
Linear Regression - Scores:  [0.7031879974468778, 0.5051757072425314,
0.6086659992278041, 0.8514958365553887, 0.560092995538598, 0.7206284371241851,
0.7622660547266362, 0.41908338421516944, 0.8351140697359691, 0.8237430814500212,
0.48032935029464885, 0.6326161364804462, 0.7272116126131474, 0.807752394803246,
-0.3961334159158154]
Linear Regression - Best Score:  0.8514958365553887
```

2.3.2 Polynomial Regression

We used 2 different ways of doing polynomial regression, once using kFold with polynom degree 2, and once using GridSearchCV by creating a pipeline for passing the hyperparams to PolynomialFeatures. GridSearchCV produced a different score with the same hyperparams, since it used different data to fit the model

[138]:
```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X_train)

kFold_poly_reg = LinearRegression()
best_score = -float("inf")
train_x = pd.DataFrame(X_poly)
train_x.describe()
train_y = pd.DataFrame(y_train)
scores = []
for train_idx, test_idx in KFold(n_splits=10).split(X_train):
    kx_train, kx_test = train_x.iloc[train_idx], train_x.iloc[test_idx]
    ky_train, ky_test = train_y.iloc[train_idx], train_y.iloc[test_idx]
    m = LinearRegression()
    m.fit(kx_train, ky_train)
    current_score = m.score(kx_test, ky_test)
    scores.append(current_score)
    if current_score > best_score:
        best_score = current_score
        kFold_poly_reg = m
print("kFold Polynomial Regression - Scores: ", scores)
print("kFold Polynomial Regression - Best Score: ", best_score)
```

```
kFold Polynomial Regression - Scores:  [0.7382884590945215, 0.6948418670749507,
0.9371364109954553, 0.5155327479685705, 0.53143053328524, 0.8799072525805655,
0.62749359937219, 0.7322116016462521, 0.7815098127900171, 0.0857632303317255]
kFold Polynomial Regression - Best Score:  0.9371364109954553
```

[139]:
```
from sklearn.pipeline import Pipeline

# Define the pipeline object
pipeline = Pipeline([
    ('poly', PolynomialFeatures()),
    ('reg', LinearRegression())
])

# Define the hyperparameter grid to search over
param_grid = {'poly__degree': [2, 3, 4, 5], 'reg__fit_intercept': [True, False]}

# Create a grid search object
gridSearch_poly_reg = GridSearchCV(pipeline, param_grid, cv=5)

gridSearch_poly_reg.fit(X_train, y_train)

# Print the best hyperparameters and the corresponding score
print("GridSearch Polynomial Regression - Best Hyperparameters: {}".
  ↪format(gridSearch_poly_reg.best_params_))
print("GridSearch Polynomial Regression - Best Score: {}".
  ↪format(gridSearch_poly_reg.best_score_))
```

```
GridSearch Polynomial Regression - Best Hyperparameters: {'poly__degree': 2,
'reg__fit_intercept': True}
GridSearch Polynomial Regression - Best Score: 0.6845904788064197
```

2.3.3 Logistic Regression

We can not use Logistic regression for predicting the house prices, as Logistic regression is a type of statistical model that is used for classification tasks. It is a regression model because it uses a linear combination of input features to make predictions, but it is different from traditional linear regression because the output variable is binary (i.e., it can take only two values, such as 0 or 1) rather than continuous.

2.3.4 Performance estimates

We use the following error/performance metrics to estimate our models: * Mean absolute error (MAE): * This is the average of the absolute differences between the predicted values and the true values. It is computed using the mean_absolute_error() function from sklearn. * Mean squared error (MSE): * This is the average of the squared differences between the predicted values and the true values. It is computed using the mean_squared_error() function from sklearn. * Root mean squared error (RMSE): * This is the square root of the mean squared error. It is computed using the sqrt() function from the NumPy library. * R^2 score (R^2): * This is the coefficient

of determination, which is a measure of how well the model fits the data. It is computed using the r2_score() function from sklearn. * Mean absolute percentage error (MAPE): * This is the average of the absolute percentage differences between the predicted values and the true values. It is often used when the true values are very small or zero, because it is not affected by the scale of the values. * Median absolute error (MedAE): * This is the median of the absolute differences between the predicted values and the true values. It is similar to the mean absolute error, but it is more robust to outliers. * Explained variance score (EVS): * This is a measure of how much of the variance in the true values is explained by the predicted values. It is computed using the explained_variance_score() function from sklearn.

[140]:
```
import numpy as np
from sklearn.metrics import mean_absolute_error, median_absolute_error,␣
 ↪mean_squared_error, r2_score, mean_absolute_percentage_error,␣
 ↪explained_variance_score

def performance_metrics(model_name, y_test, y_pred):
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, y_pred)
    mape = mean_absolute_percentage_error(y_test, y_pred)
    medae = median_absolute_error(y_test, y_pred)
    evs = explained_variance_score(y_test, y_pred)
    return model_name, mae, mse, rmse, r2, mape, medae, evs

performance_metrics_column_names = ["model", "mae", "mse", "rmse", "r2",␣
 ↪"mape", "meade", "evs"]
```

[141]:
```
from itertools import islice

metrics_df = pd.DataFrame(columns=performance_metrics_column_names)
metrics_df.loc[len(metrics_df.index)] = performance_metrics("Linear␣
 ↪Regression", y_test, linear_reg.predict(X_test))
metrics_df.loc[len(metrics_df.index)] = performance_metrics("kFold Polynomial␣
 ↪Regression", y_test, kFold_poly_reg.predict(poly.fit_transform(X_test)))
metrics_df.loc[len(metrics_df.index)] = performance_metrics("GridSearch␣
 ↪Polynomial Regression", y_test, gridSearch_poly_reg.predict(X_test))
# mark the best metric/error for each column depending on whether higher or␣
 ↪lower is better (higher for perf and lower for errors
#(metrics_df.style.highlight_min(color="green", axis=0, subset=["mae", "mse",␣
 ↪"rmse", "mape", "meade"])).highlight_max(color="green", axis=0,␣
 ↪subset=["r2", "evs"]).export()

# sadly, the styler only works in the notebook not in the export
metrics_df
```

[141]:
```
                              model       mae        mse      rmse        r2  \
0                 Linear Regression  3.594819  23.360565  4.833277  0.730813
1       kFold Polynomial Regression  2.720055  13.286671  3.645089  0.846896
2  GridSearch Polynomial Regression  2.680316  13.251017  3.640195  0.847306

       mape     meade       evs
0  0.183801  2.698439  0.731767
1  0.131981  2.055352  0.847344
2  0.129965  2.108711  0.847554
```

2.3.5 Model selection

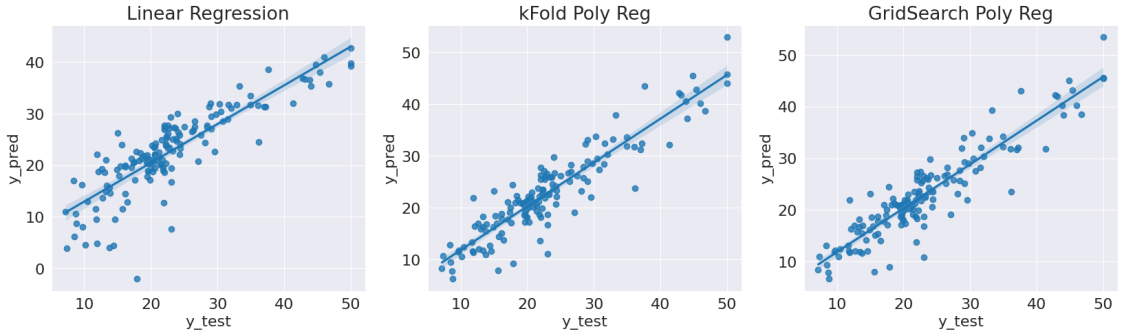
As we can see from the performance estimates, the polynomial regression definitely outperforms the linear regression.

So the obvious chose would be one of the two polynomial regression model. Though it should be noted, that data vastly outside our training data can‘t be predicted and in that case may perform worse than the linear regression model.

Plot y_test against y_pred For fun we wanted to plot a regresseion between y_test and the prediction, if prediction were perfect, the correlation would be 1 and we would have a diagonal line.

[142]:
```
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3)
fig.set_size_inches(20, 5)

ax1.set_title("Linear Regression")
sns.regplot(x=y_test, y=pd.DataFrame(linear_reg.predict(X_test))[0], ax=ax1)
ax2.set_title("kFold Poly Reg")
sns.regplot(x=y_test, y=pd.DataFrame(kFold_poly_reg.predict(poly.
 ↪fit_transform(X_test)))[0], ax=ax2)
ax3.set_title("GridSearch Poly Reg")
sns.regplot(x=y_test, y=pd.DataFrame(gridSearch_poly_reg.predict(X_test))[0],␣
 ↪ax=ax3)

ax1.set_xlabel("y_test")
ax1.set_ylabel("y_pred")
ax2.set_xlabel("y_test")
ax2.set_ylabel("y_pred")
ax3.set_xlabel("y_test")
ax3.set_ylabel("y_pred")
```

[142]:
```
Text(0, 0.5, 'y_pred')
```


Linear Regression
y_pred
y_test
kFold Poly Reg
y_pred
y_test
GridSearch Poly Reg
y_pred
y_test