# E2E ML project

The main goal of this presentation is to introduce you to data analysis and data preparation using the scikit-learn python library that is targeted at ML applications. I will also demonstrate briefly how to train and evaluate a ML model. The presentation consists of 5 videos:

1. In the first video, I will show how to load a dataset and inspect the data.
2. In the second video, I use a simple technique to analyze the data to see, if some of the features could be removed from the dataset, without affecting the accuracy of the ML model we want to develop.
3. In the 3rd video, I will demonstrate a few important data preparation techniques that are often used in machine learning.
4. In the 4th video, I will demonstrate how to create a pipeline that can perform a sequence of data preparation tasks as a single operation. This is useful, when you want to perform the same sequence of data preparation tasks more than once.
5. In the 5th and last video, I will demonstrate how to train a decision tree to perform regression, and I will also demonstrate how to evaluate the performance of the trained model.

Let’s get started (vis California billede). Our task is to predict median house values in Californian districts, given a number of features from these districts. A district typically has a population of 600 to 3000 people. This is a regression task since we should predict a numerical value.

The first thing we need to do is to load the dataset. Scikit-learn has predefined functions for loading several popular datasets, including the dataset that we will use in this presentation. However, since we must load the dataset from a file in most cases, I will demonstrate how to do that.

## Download

I have already created a function that can download the dataset from a location on Internet. Of course, we could also download the dataset manually from this location.

## Read

Once downloaded, we need to read the dataset. I have created a function that uses the read\_csv() function of the pandas library to load the dataset. It returns a pandas DataFrame, which is a good format when we want to inspect and analyze the data.

We will call this function to load the dataset and then use a few functions to inspect the data.

## Head()

The first function is called head(). It displays the top five rows.

* housing.head()

The values that we want to predict are median house values. This means that we should use the column named *median\_house\_value* as the target column. The remaining 9 columns will be used as features. The *housing\_median\_age* is the median age of the houses in the district. *Total\_rooms* and *Total\_bedrooms* are the total number of rooms and bedroooms in the district. Population is the total population in the district. Note that *Ocean\_proximity* has string values. These values must be converted to numerical values during data preparation.

## Info()

The next function we will use is called info(). It provides a quick description of the data.

* housing.info()

We note that there are 20640 examples in the dataset. We can also see, that the total\_bedrooms feature has many null-values, since there are only 20443 numerical values. These null-values must be handled in some way during data preparation.

## Value\_counts()

For the non-numerical feature, ocean proximity, we will use the value\_counts() function to find out how many different values this feature can have.

* housing["ocean\_proximity"].value\_counts()

We can see that there are 5 possible values. This means that ocean\_proximity is a category attribute with 5 unordered categories.

## Describe()

The describe() method displays some basic statistics for each attribute.

* housing.describe()

We get the total number of values, min, max, mean, standard deviation, 25, 50 and 75 percentiles.

## Histogram()

We can use the matplotlib library to plot a histogram for each numerical attribute.

* *Execute code*

We can see that housing\_median\_age, median\_house\_value and median\_income have been capped. The median\_income has been scaled to tens of thousand dollars. The features have very different scales. Since this is a problem for most ML algorithms, we will re-scale the features during data preparation.

A histogram can show if there is a large number of atypical values – also called outliers. This could significantly reduce the accuracy of the trained ML model. If there are many outliers, we should consider removing rows with outliers from the dataset.

In the next video, we will look at data analysis.

## Data analysis

This is the second of 5 videos covering an E2E ML project, where we train a model using the *California housing Prices* dataset. We will use a simple technique to analyze the data to see, if some of the features could be removed from the dataset, without affecting the accuracy of the ML model we want to develop.

### Correlations

More specifically, we want to know how much each numerical feature is correlated with the target attribute, which is median house value. We can use the corr() function to compute the linear correlation between any pair of numerical attributes in the dataset.

* corr\_matrix = housing.corr()

The result is a matrix consisting of 10 rows and 10 columns, but since we are only interested in correlations with median house value, we will extract and display that column.

* corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

The correlation coefficient ranges between -1 and +1. -1 represents 100% negative correlation, and +1 represents 100% positive correlation. A value close to 0 indicates very low correlation.

We can see that there is a high positive correlation between *median\_income* and *median\_house\_value* (approximately 0.69). So, there is a clear tendency for high-income people to live in expensive houses.

There is some negative correlation between *lattitude* and *median\_house\_value* (approximately -0.14). There is thus a slight tendency for house prices to rise when moving south. This may be because the big cities of Los Angeles and San Diego are in Southern California.

The correlation between *population* and the *median\_house\_value* is very low, indicating that the total population in a district has little influence on house prices. We might consider removing this feature from the dataset to speed up training, but since the total number of features is already very low, we will not do that.

### Attribute combinations

As the last part of the data analysis, we will examine if some attribute combinations make more sense than the original attributes.

Intuitively, the total number of rooms in a district does not make much sense. Instead, we want the number of rooms per household. Similarly, we want the number of bedrooms per room instead of the total number of bedrooms in a district, and the population per household instead of the total population in a district.

The pandas DataFrame class provides many methods and operators for data manipulation and arithmetic operations on the data. As a result, it is relatively easy to add new attributes with values computed from other attribute values, as you can see here. *(vis kodecelle)*

We see that the new attribute, *bedrooms\_per\_room* is much more correlated with the *median\_house\_value* than the original *total\_bedrooms* attribute. The correlation coefficient is negative, indicating that a house or apartment costs less, if a large fraction of the rooms are bedroooms.

For the two other new attributes, the correlation coefficient is not very different than the correlation coefficient for the original attributes. Anyway, we will keep these since they make more sense intuitively than the original attributes.

All we have left is to remove the original 3 attributes that should be replaced by the new attributes. The pandas DataFrame class has a drop() method that makes it easy to do that. *(vis kodecelle)*

The drop() method will not affect the original DataFrame, but it will return a new DataFrame with the specified column removed. Since the new DataFrame is assigned to the same variable, as we used to reference the original DataFrame, the original DataFrame will be overwritten.

In the next video, we will look at data preparation.

## Data preparation 1

This is the 3rd of 5 videos covering an E2E ML project, where we train a model using the *California housing Prices* dataset. I will demonstrate a few important data preparation techniques that are often used in machine learning.

I will demonstrate

* how to split the dataset in a training set and a test set,
* how to clean the data so that there are no missing values,
* how to re-scale attribute values,
* and how to handle a non-numerical feature.

### Split the dataset

Most of the dataset should be used as input during training. The part that is used for training is called the training set. Once a machine learning model has been trained, it must be evaluated to determine if its accuracy is satisfactory. During evaluation, we use a small part of the dataset that the trained model has not yet seen. This part is called the test set. As a rule of thumb, 80% of the dataset should be used for training and the remaining 20% for testing.

*Vis kode*

When we set the random\_state argument to some constant value, the method will always split the dataset in the same way every time it is executed.

If the dataset is not very large, stratified sampling should be considered. Stratified sampling ensures that the test set is representative of the whole dataset. The dataset is first divided into subgroups called strata. A stratum could for example represent a specific income group. After having created the strata, the right number of instances for both training and test sets are picked from each stratum. Stratified sampling can reduce sampling bias. In this presentation, we will omit stratified sampling to keep things simple.

To make the training set appropriate for training, we must remove the target or label attribute, median\_house\_value, and keep it to a separate set. *Vis kode*.

### Clean the data

Let us now look at data cleaning. During this step, we want to remove invalid or unwanted values from the data set. In this presentation, you will learn how to remove missing values. This is the most common form of invalid data.

As we saw earlier, the total\_bedrooms attribute had 207 missing values, and the new attribute, bedrooms\_per\_room, will of course also get 207 missing values. Since no machine learning model can tolerate missing values, these values must be removed. There are 3 options:

1. Remove all the rows that have missing values. This is a good strategy if there are few rows with missing values.
2. Remove all the columns that have missing values. It can be a good strategy if there are many rows with missing values, but if these columns are important, it may be better to use the 3rd strategy, which I will introduce now.
3. Replace missing values with some default value (mean, median, most frequent or some fixed value). We will follow this strategy here, and we will use Scikit-Learn’s SimpleImputer class. *Vis kode*.

First, we will create an instance of SimpleImputer. We should pass the strategy as a parameter to the constructor. As you can see, we have chosen to replace missing values with the median of the attribute in question.

Since the median can only be computed on numerical attributes, we need to create a copy of the data without the ocean\_proximity attribute.

We then call the SimpleImputer’s fit() method. This method will compute the median of each column and store the values in its statistics\_ instance variable. As you can see, this variable is an array.

Finally, we call the SimpleImputer’s transform method() to replace the missing values with the medians that were computed by the fit() method.

### Feature scaling

The next step in data preparation is feature scaling. Most ML model architectures don't perform well when the numerical features have different scales. There are two common ways of scaling:

1. The first is min-max scaling or normalization: here values are rescaled so that they end up ranging from 0 to 1.
2. The second is standardization: here values are rescaled so that they have zero mean and unit variance. Standardization is much less affected by outliers, but the lack of a fixed range from 0 to 1 is a problem for some model architectures, for example neural networks.

Scaling the target values is generally not required, but we will do it anyway for convenience. Scikit-Learn provide scaler classes for each of the two ways of scaling that we have talked above. These classes are called MinMaxScaler and StandardScaler. They both have a fit\_transform() method that will perform the scaling on the data we pass as a parameter. Let us try with min-max scaling.

*Eksekver kode*.

### Handling non-numerical features

The last step in our data preparation is to handle the non-numerical attribute, ocean\_proximity. Let us first extract the attribute and see how many different values it has. *Eksekver kode*.

We see that it can have 5 different values. This indicates that it is a categorial attribute. Since the categories are unordered, we should not convert the attribute to a single numerical attribute, where each category is assigned an integer value. Doing so would make the trained ML model assume that the categories were ordered (like small, medium, and large), and this could have a negative impact on the model’s accuracy. Instead, we should create one binary attribute per category.

The technique that we will use is called one-hot encoding, because only one attribute value will be 1 (or hot) while the others will be 0 (or cold). We will use Sciikit-Learn’s OneHotEncoder class to one-hot encode the ocean\_proximity attribute. The default output is a sparse matrix, which only stores the location of the non-zero elements. This will save memory. Here we set sparse to false and use a normal dense matrix, since there are only a few categories. We call the OneHotEncoder’s fit\_transform() method to perform the one-hot encoding. *Eksekver kode*.

As you can see, the result is a matrix with 5 columns – one for each category, and each row contains only a single hot value.

The original categories are stored in the OneHotEncoder’s categories\_ instance variable. *Eksekver kode*.

The last thing we need to do is to replace the original ocean\_proximity attribute with the one-hot encoded version. We will do that in the next video.

## Data preparation 2: using a pipeline

This is the 4th of 5 videos covering an E2E ML project, where we train a model using the *California housing Prices* dataset. I will demonstrate how to create a pipeline that can perform a sequence of data preparation tasks as a single operation. This is useful, when you want to perform the same sequence of data preparation tasks more than once. In our case, we want to perform the data preparation tasks on both the training set and the test set. We could of course have performed all the data preparation tasks on the whole dataset before we split it into training and test sets, but then we would not have had any reason to use a pipeline.

Scikit\_learn provides a Pipeline class for creating a pipeline. A Pipeline object can perform a sequence of data preparation tasks, specified by the programmer, by calling a single method. We should pass the desired sequence of data preparation tasks as constructor parameters, when we create the Pipeline object. In our case we have specified a SimpleImputer to replace missing values with medians, and a MinMaxScaler to normalize the numerical features. We execute the whole pipeline by calling its fit\_transform() method, passing the data that should be prepared as a parameter. In our case, we pass a DataFrame, containing all the numerical features. The fit\_transform method returns the transformed data as a NumPy array. Let’s execute the code. *Execute code*.

We did not handle the ocean\_proximity attribute in the pipeline that we just executed, since it is a non-numerical category attribute. Scikit-Learn provides a ColumnTransformer class that can perform different transformations on different sets of columns and return a numpy array with all the transformed columns. When we create a ColumnTransformer object, we must pass the desired sequence of data preparation tasks as constructor parameters, just as we did when we created a Pipeline object. For each data preparation task, we must provide an array containing the names of the attributes which should be transformed. To execute the ColumnTransformer, we call the fit\_transform() method, passing the DataFrame that contains all the features of the training set. The fit\_transform method returns the transformed data as a NumPy array. Let’s execute the code. *Execute the code*.

Let us look at the shape of the transformed data. *Execute the code*.

We can see that there are 16512 rows, just as there were in the original training set. There are 13 columns, while there were only 9 feature columns in the original training set. This is because we performed one-hot encoding on the categorial ocean\_proximity attribute. Since there were 5 categories, the one-hot encoding resulted 5 binary attributes.

In the next and last video, we will look at how to train and evaluate a model.

## Training and evaluation

This is the last of 5 videos covering an E2E ML project, where we train a model using the *California housing Prices* dataset. I will demonstrate how to train a decision tree to perform regression, and I will also demonstrate how to evaluate the performance of the trained model.

### Training

To train a decision tree for regression, we should create a DecisionTreeRegressor and call its fit() method, passing the features and the labels from the training set. *Eksekver kode*.

Now that we have a trained model, we would like to evaluate its performance. The preferred performance measure for regression problems is the Root Mean Square Error. It measures the distance between the vector of predictions and the vector of target values.

We will first measure the RMSE on the training set. To do that, we should make a prediction for every example in the training set using the DecisionTreeRegressor’s predict() method. We can then compute the MSE by calling the mean\_squared\_error() method, passing the predictions and the corresponding target values. The RMSE is computed as the square root of the MSE. *Eksekver kode*.

We see that the RMSE is 0. I will comment on that in a moment.

If there are many outliers in the dataset, you may consider using the Mean Absolute Error instead. We can compute it by calling the mean\_absolute\_error(), and again passing the predictions and the corresponding target values. *Eksekver kode*.

Again, we get a value of 0. This could indicate that the model overfits the training data badly. Let’s try to evaluate the model on the test set instead.

### Evaluation

Before we can evaluate on the test set, we must perform the same data preparation tasks as we did on the training set. First, we split the training set in a set, containing the features, and a set containing the labels. Then we will execute the a ColumnTransformer on the set containing the features. The ColumnTransformer was discussed in the previous video. It replaces missing values with medians, normalizes the numerical features, and one-hot encodes the categorial ocean-proximity attribute. Let’s run the cell to perform the data preparation on the test set. *Eksekver kode*.

All we have left to now, is to make some predictions using the test set, and then compute the RMSE. *Eksekver kode*.

The RMSE on median house value predictions is more than 68000 dollars, which is quite a lot. This confirms our suspicion that the trained decision tree overfits badly.

We have now completed the E2E ML project, where we have analyzed and prepared the *California housing Prices* dataset, and then trained and evaluated a decision tree too predict median house values. Thank you for listening.