**ML Practical: Bias and Variance in Leaning**

In this practical you shall practice more data preparation and regression models for machine learning and use the data and model to explore the issues of bias and variance in the learning.

Start with more data preparation techniques. Reminder: preparation entails [**Data Cleansing**](https://en.wikipedia.org/wiki/Data_cleansing), transformation and splitting into **train-validation-test sets**. These processes are usually crucial for the success of the learning.

**Plotly**

  
Link: <https://plotly.com/>  
A useful Python graphing library which produces interactive, publication-quality graphs.  
Plotly supports over 40 chart types and covers a wide range of statistical, financial, geographic, scientific, and 3-dimensional use-cases.  
The package is free and open source, so you can view the source, report issues or contribute on [GitHub](https://github.com/plotly).

**Downloads, Imports and Definitions**

It’s always good to check Plotly’s version on your system using !pip show. And to upgrade it to the latest version.

**To do that, use**: !pip install –upgrade

**Results:**

Name: plotly

Version: 5.6.0

Summary: An open-source, interactive data visualization library for Python

Home-page: <https://plotly.com/python/>

Author: Chris P

Author-email: chris@plot.ly

License: MIT

Location: c:\users\admin\anaconda3\lib\site-packages

Requires: tenacity, six

Required-by:

if you don’t have the latest version use !pip install --upgrade plotly

Then, as in the previous practicals, import numpy, matplotlib, and sklearn libraries

*# import numpy, matplotlib, etc.*

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

*# sklearn imports*

**from** **sklearn** **import** metrics

**from** **sklearn** **import** pipeline

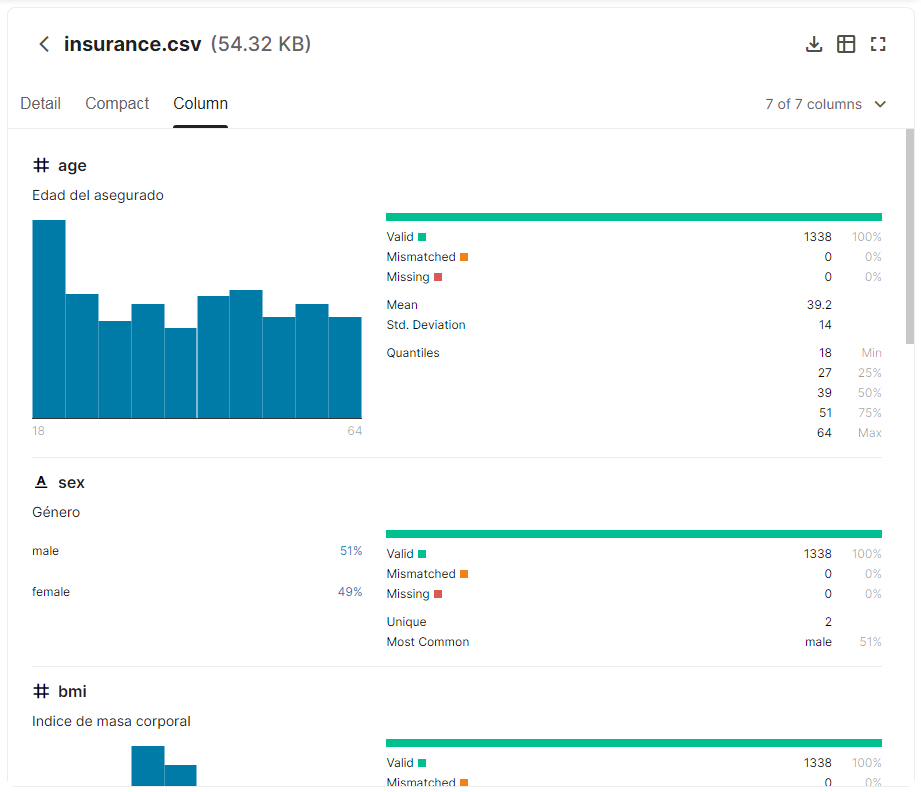
**from** **sklearn** **import** linear\_model

**from** **sklearn** **import** preprocessing

**from** **sklearn** **import** model\_selection

**Data Cleansing**

We will use the [insurance](https://github.com/stedy/Machine-Learning-with-R-datasets/blob/master/insurance.csv) dataset loaded from Github. Our goal is to predict, based on this dataset, individual costs of a health insurance.  
You can read more about the dataset in [Kaggle](https://www.kaggle.com/mirichoi0218/insurance).  
Kaggle also offers a summary of the datasheet's columns at the bottom of the page:



.  
You can grab the dataset from Kaggle servers, but it is simpler to download it from Github, since Kaggle requires an account (You shall do that later in the course assignment).

1. Download the dataset from Github with Linux command wget. Then load the insurance csv file.

To do that use the following code:

*# download insurance.csv file from Github*

!wget https://github.com/stedy/Machine-Learning-with-R-datasets/raw/master/insurance.csv

**Note:**

wget is a third-party program that doesn't come bundled with Windows, so you need to explicitly install it in order to use it.

You can find (one of) the Windows versions here: <http://gnuwin32.sourceforge.net/packages/wget.htm>

You will need to add the path of the wget.exe file to your PATH environment variable in order to call the executable as in the batch file above without explicitly specifying the path.

For Windows 10: A good link is available here: <https://builtvisible.com/download-your-website-with-wget/>

*# load the insurance csv file*

insurance\_df = pd.read\_csv('insurance.csv')

insurance\_df

You should get a table with 1338 rows and 7 columns which are the 6 features + the 1 label.  
*The Features* are*:*

1. **age**: age of primary beneficiary (int).
2. **sex**: insurance contractor gender (string; female, male).
3. **bmi**: Body mass index, providing an understanding of body, weights that are relatively high or low relative to height, objective index of body weight (kg / m ^ 2) using the ratio of height to weight, ideally 18.5 to 24.9 (float).
4. **children**: Number of children covered by health insurance / Number of dependents (int).
5. **smoker**: Smoking? (string; yes, no).
6. **region**: the beneficiary's residential area in the US (string; northeast, southeast, southwest, northwest).

*The Target variable is:*

* **charges**: Individual cost of health insurance (float).

|  | **age** | **sex** | **bmi** | **children** | **smoker** | **region** | **charges** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 19 | female | 27.900 | 0 | yes | southwest | 16884.92400 |
| **1** | 18 | male | 33.770 | 1 | no | southeast | 1725.55230 |
| **2** | 28 | male | 33.000 | 3 | no | southeast | 4449.46200 |
| **3** | 33 | male | 22.705 | 0 | no | northwest | 21984.47061 |
| **4** | 32 | male | 28.880 | 0 | no | northwest | 3866.85520 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **1333** | 50 | male | 30.970 | 3 | no | northwest | 10600.54830 |
| **1334** | 18 | female | 31.920 | 0 | no | northeast | 2205.98080 |
| **1335** | 18 | female | 36.850 | 0 | no | southeast | 1629.83350 |
| **1336** | 21 | female | 25.800 | 0 | no | southwest | 2007.94500 |
| **1337** | 61 | female | 29.070 | 0 | yes | northwest | 29141.36030 |

1338 rows × 7 columns

**Note that:**

Some of the features are **categorical** (sex, smoker and region).  
Categorical features are features that has no intrinsic order between their values (smoker, non-smoker).

Some of our features are **ordinal** (children).  
Ordinal features are features similar to categorical features, but there is an order between the values (i.e. 1 child, 2 children, etc.).  
In ordinal features, there is no meaning for the values between the integers (i.e. there is no 1.5 child).

And some of our features are **numerical** (bmi, age and the target charges).  
Numerical features are like ordinal features, but there is meaning to the values between the integers (bmi is a scale, there is clinical significance to each fraction in the scale).

1. Let's start with cleansing the dataset. We’ve done this in previous practical but one can never have too much of a good thing. The first thing to do, is to check for empty values.  
   Empty values can be '' in string columns, or NaN values.

**Hint:** To do that use the numpy where() method and check if there is a null value in the dataframe

1. You’ll find that there are no empty values. This is very rare in real world data, where we usually have missing data and need to fill the empty values, or remove the rows/columns that have them.  
   To practice this, Let's synthesized missing data:

Your task is to insert an empty line to the data frame and fill them with np.NaN, **None** and "".

To do it use the following code.

*# add an empty line to the df*

insurance\_df\_cp = insurance\_df.copy()

insurance\_df\_cp.loc[len(insurance\_df)] = [np.NaN, "", np.NaN, **None**, **None**, "", np.NaN]

insurance\_df\_cp.loc[len(insurance\_df\_cp)] = [np.NaN, "", np.NaN, np.NaN, **None**, **None**, np.NaN]

insurance\_df\_cp

1. In some cases we can discard data rows with missing data. In this practical, we will fill the missing values, which we do when we believe that every data row is valuable.   
   Start with checking the types of the columns dtypes.

The results should look like:

age float64

sex object

bmi float64

children object

smoker object

region object

charges float64

dtype: object

* Type float64 indicates that a column is a floating point number. So, in such column, the only empty value possible is np.NaN.
* Type object, indicates that a column is a string or a floating point number (with None values). So, in such column, the empty values possible are np.NaN, "" and None.
* The type hierarchy is: **int64 < float64 < object**.
* When there is at least one float64 element in the column, the column type will be float64.
* When there is at least one object element in the column, the column type will be object.

1. Replace all the empty values to np.NaN values (Pandas works best with these values).

**Hint:** use the replace and the fillna methods

1. Now detect and print the empty values (rows, cols), using where. The outcome should look like:

There are 14 empty values in the dataframe:

(array([1338, 1338, 1338, 1338, 1338, 1338, 1338, 1339, 1339, 1339, 1339,1339, 1339, 1339]), array([0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6]))

1. Count them and print the number of empty values in each of the column. The ourcome should look like:

empty values:

`age`: 2

`sex`: 2

`bmi`: 2

`children`: 2

`smoker`: 2

`region`: 2

`charges`: 2

There are several options for completing these missing values (no “correct” one):

1. Enter a **constant** value.  
   For continuous values, the constant value can be calculated using the rest of the values in the column (min, max, mean, median, etc.) or derived from expert/prior knowledge.  
   For categorical values, the constant value can be one of the values in the column or a different value not present in the column.
2. Enter **random** values.  
   Continuous values can be randomly picked from the values of the column, or be randomly generated from the range of the optional values in the column.  
   Categorical variables can be randomly picked from the values in the column. We can use normal distribution or column distribution.
3. Enter **prediction** of the values.  
   For continuous values, we can use *regression methods* to predict the missing values.  
   For categorical values, we can use *classification methods* to predict the missing values.

Ordinal features are a type between categorical and numerical features, so any of the above methods can be applied on them.

1. To complete the empty values in a column, we need information on the column value:  
   Let's start with the categorical columns. Use Visualization – import **plotly** to plot the distribution of each column in [pie charts](https://plotly.com/python/pie-charts/).

**Hint:** Use “**plotly.express** **as** **px**” and the create a function that creates pie chart. The function should get 2 variables; the dataframe and the column name that you want to make the pie chart. To do that use the following code:

*# import px and create pie charts for each categorical feature*

**import** **plotly.express** **as** **px**

**def** create\_pie\_chart\_of\_count(df, column\_name):

df\_not\_null = df[~df[column\_name].isnull()]

fig = px.pie(df\_not\_null.groupby([column\_name]).size().reset\_index(name='count'), names=column\_name, values='count')

fig.show()

Use the provided function to plot pie charts for all the categories. An example of pie chart can be found in the figure below.

Chart, pie chart

Description automatically generated

1. **Advance:** Now show all the plots as [subplots](https://plotly.com/python-api-reference/generated/plotly.subplots.make_subplots.html) with [graph objects](https://plotly.com/python/creating-and-updating-figures/#figures-as-graph-objects) and [pie charts](https://plotly.github.io/plotly.py-docs/generated/plotly.graph_objects.Pie.html).

The output should look like the figure below

Chart, pie chart

Description automatically generated

1. A good way to show the inner distribution in the data is [sunburst charts](https://plotly.com/python/sunburst-charts/).  
   This is useful even for the non-categorical features. We can also limit the depth (inner circles) of the chart and put the non-categorical features at the end of the chain. Read the information that are provided in the ploty’s sunburst charts website and create sunburst charts of the features in the data.

**To implement a sunburst chart read:** <https://plotly.com/python/sunburst-charts/>

**The outcome should look like:**

Chart, sunburst chart

Description automatically generated

1. The most popular (since it’s simple, but not always the best) way to fill missing values is to randomly pick one of the values for categorical features, and pick the mean or median value for numerical features. Use this to fill the empty values in the dataframe. Then print the dataframe to see how the missing values have been filled.

**Hint:** Implement 3 functions one for each mean, one for median and one for randomly picking columns

1. To ensure that the operation was successful, check again for missing values, Count and print the # of missing values in each column (Similar to task 7).

Some machine learning algorithms (like logistic and linear regression) cannot work with categorical features and may only work with numerical or ordinal features.  
The next step in preparing the dataset for model learning is converting the categorical features into numerical features.

There are few techniques for doing that:

1. **Label Encoding:**  
   Transform the categorical data into ordinal data, by translating each category to an integer number. This technique is useful when there is an order to the values or when there are too many values to handle.  
   Example:

| **old\_region\_column** | **new\_region\_column** |
| --- | --- |
| southwest | 0 |
| northwest | 1 |
| southeast | 2 |
| northeast | 3 |

1. **One-Hot Encoding:**  
   Transform the categorical data into binary vectors (columns), by translating each category into a column of 0 and 1 values: 1 if the original categorical value is present in the row, and 0 if not.  
   This technique is useful when there is no order to the values and where there aren't as many different values in the column.

This should be used with unregularized regressions (may suffer from bias issues with the added column).

Example:

| **old\_region\_column** | **new\_southwest\_column** | **new\_northwest\_column** | **new\_southeast\_column** | **new\_northeast\_column** |
| --- | --- | --- | --- | --- |
| southwest | 1 | 0 | 0 | 0 |
| northwest | 0 | 1 | 0 | 0 |
| southeast | 0 | 0 | 1 | 0 |
| northeast | 0 | 0 | 0 | 1 |

1. **Dummy Encoding:**  
   Same as one-hot encoding, but omitting one of the columns (the category that was represented by that missing column will be represented by 0 in all the other columns).  
   This should be used with unregularized regressions or with neural networks (may suffer from variance issues with the added column).

| **old\_region\_column** | **new\_southwest\_column** | **new\_northwest\_column** | **new\_southeast\_column** |
| --- | --- | --- | --- |
| southwest | 1 | 0 | 0 |
| northwest | 0 | 1 | 0 |
| southeast | 0 | 0 | 1 |
| northeast | 0 | 0 | 0 |

1. Use Scikit-learn [OneHotEncoder](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) as a Dummy Encoder for the data categorial features and observe the encoded dataframe.

**Hint:** you need to import OneHotEncoder from sklearn.preprocessing

**The outcome should look like:**

|  | **age** | **bmi** | **children** | **charges** | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 19.0 | 27.900000 | 0.0 | 16884.924000 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| **1** | 18.0 | 33.770000 | 1.0 | 1725.552300 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **2** | 28.0 | 33.000000 | 3.0 | 4449.462000 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **3** | 33.0 | 22.705000 | 0.0 | 21984.470610 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| **4** | 32.0 | 28.880000 | 0.0 | 3866.855200 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **1335** | 18.0 | 36.850000 | 0.0 | 1629.833500 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **1336** | 21.0 | 25.800000 | 0.0 | 2007.945000 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **1337** | 61.0 | 29.070000 | 0.0 | 29141.360300 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| **1338** | 39.0 | 30.663397 | 0.0 | 13270.422265 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **1339** | 39.0 | 30.663397 | 3.0 | 13270.422265 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |

You should see that sex column has been converted to 1 binary column, the smoker column has been converted to 1 binary column, and the region column has been converted to 3 binary columns.

1. Create your own Dummy\_Encoder method to perform Dummy encoding for this DF (make use of OneHotEncoder).
2. You can also use Pandas [get\_dummies](https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html) method in one line, and attach names to the new columns.

The difference between the get\_dummies approace and the OneHotEncoder approach is that OneHotEncoder can transform several datasets with the same encoding (for example, both train and test), while get\_dummies only convert one dataframe at a time - which may result in different encodings for the same column in different datasets.

**Data Slicing**

In real life scenarios, we don't have a test data. Naturally we cannot check the performance of the model on the same dataset it was trained on - This will cause wrong estimation for the model generalization capabilities. In order to check our prediction and fine-tune the model parameters, we need to slice the dataset into 2 groups: (1) train (2) validation. We will train on the train data and check the performance on the validation data.

1. Slicing the dataset is performed with Scikit-learn [train\_test\_split](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html). Separate the data into features X and target t and print the features x and targets t. The outcome should look like this:

Targets:

0 16884.924000

1 1725.552300

2 4449.462000

3 21984.470610

4 3866.855200

...

1335 1629.833500

1336 2007.945000

1337 29141.360300

1338 13270.422265

1339 13270.422265

Name: charges, Length: 1340, dtype: float64

Features:

|  | **age** | **bmi** | **children** | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 19.0 | 27.900000 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| **1** | 18.0 | 33.770000 | 1.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **2** | 28.0 | 33.000000 | 3.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **3** | 33.0 | 22.705000 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| **4** | 32.0 | 28.880000 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **1335** | 18.0 | 36.850000 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| **1336** | 21.0 | 25.800000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **1337** | 61.0 | 29.070000 | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| **1338** | 39.0 | 30.663397 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| **1339** | 39.0 | 30.663397 | 3.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |

1340 rows × 8 columns

1. Then split the data to train and validation. Increase by increments of 10% the number of values for the test\_size argument, ie taking 10%, 20%,30%...90% of the data for the validation set. Check the performance of different splitting in linear regression NE (as you did in practical 7) and the performance measure MSE and Plot the data with Plotly [scatter](https://plotly.com/python/line-and-scatter/).

**The outcome should look like this:**

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

You should see in the graphs that when the validation data size is small, its loss is small.  
One explanation to this is that for a small group of samples, it is easier to match a linear hypothesis.

You should see a low range of percentages of the validation set size - between 0.1 to 0.3, where the validation loss is smaller than the train loss, and a high range – between 0.4 to 0.9. where the validation loss is smaller than the test loss.  
In such case, choosing a validation group percentage between these ranges, say 35% of the dataset, is about the “right point” where the validation loss is equal to the train loss.

1. For the next steps do now splitting with test\_size=0.35.

**Bias and Variance**

The data is now prepared. Use it to train the NE regression model and print the MSE and R2 graphs of the train and test.

To check and visualize the effect of bias and variance, we’ll now play with the model:

Start in adding more features to the data set, thus increasing the **degree** of the model.

For example, increasing to a second degree, means we’re creating polynomial features of the data. This can be performed using Scikit-learn [PolynomialFeatures](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html" \l "sklearn.preprocessing.PolynomialFeatures),

Explanation of the procedure: If we have two original features [a, b] the increase to the 2nd degree will yield the features [1, a, b, a^2, ab, b^2].  
Note: we can choose not to include the intercept (the “1”) with the include\_bias option, and stay with [a, b, a^2, ab, b^2].

The advantage of this procedure is that instead of training a more complex polynomial model (which we haven’t learned yet) on the original features, we can now train a simpler linear model on these new features which are polynomial.

1. Try it how on 2 features from the data: age and bmi. Print the 2 original features and then use preprocessing.PolynomialFeatures and print the transformed features. You should get the age^2, the age\*bmi and the bmi^2 added columns.

**The outcome should look like this:**

|  | **age** | **bmi** |
| --- | --- | --- |
| **275** | 47.0 | 26.600 |
| **719** | 58.0 | 33.440 |
| **345** | 34.0 | 29.260 |
| **979** | 36.0 | 29.920 |
| **966** | 51.0 | 24.795 |
| **...** | ... | ... |
| **715** | 60.0 | 28.900 |
| **905** | 26.0 | 29.355 |
| **1096** | 51.0 | 34.960 |
| **235** | 40.0 | 22.220 |
| **1061** | 57.0 | 27.940 |

1. Let’s check which degree is best on the age feature: Take model degrees from 1 to 19, and for each degree train NE linear regression model. Plot a graph where the degrees are on the ‘x’ axis and the loss or score (MSE or R2) on the ‘y’ axis.

**The outcome should look like this:**

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

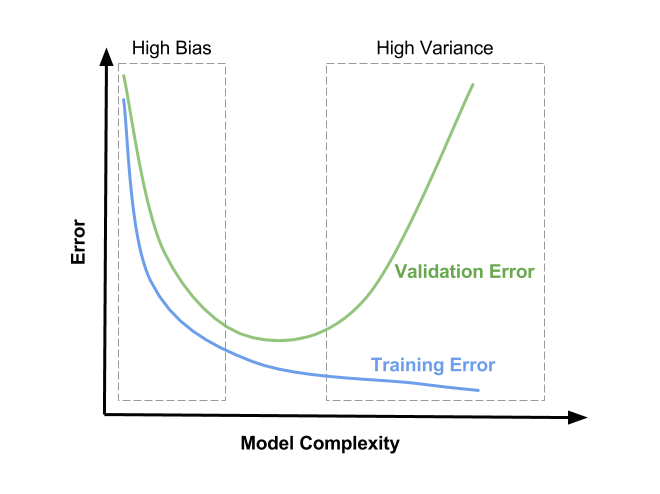
Chart, line chart

Description automatically generated

You should see that in the MSE loss of the train is smaller than the MSE loss of the validation – which means that the model is performing better on the train dataset.  
This is what happen most often (but not always).  
The train loss decreases as the complexity of the model increases.  
The validation loss decreases initially, until it reaches a minimum in degree 3, and then it starts increasing as the complexity of the model increases.

The behaviour where the train and validation/test losses are decreasing together indicates **High Bias**.  
High Bias means that the model is not robust enough and we need to increase its complexity to help it learn better on the data.

The behaviour where the train loss decreases and the validation/test loss increases indicates **High Variance**.  
High Variance means that the model is fitted too closely to the train data, and we need to make the model less complex (lower the degree).



We can see the same phenomena, but in the opposite direction, in the R2 score graphs. On **High Bias** the R2 score graph of the train and validation/test will increase together.  
On **High Variance** the R2 score graph of the train will increase, and the R2 graph of the validation/test will decrease.

Another phenomenon in the line graphs of the age feature, is that the R2 score is very low.  
This indicates that this feature alone is not enough to predict the charges target on its own.

**Visualization of the Regression Graph**

We can observe a regression hypothesis in a graph, choosing 1 feature for a 2D graph or 2 features for a 3D graph.

1. Plot 2D graph of the datapoints, using Plotly [Scatter](https://plotly.com/python/line-and-scatter/)  with the bmi feature on the x axis, and the target charges will be on the y axis. Add the regression line (with NumPy [linspace](https://numpy.org/doc/stable/reference/generated/numpy.linspace.html)). You can add a ‘slider’ – which slides using a cursor, over the different degrees to show the resulting line for each , using Plotly [Slider](https://plotly.com/python/sliders/).

**The outcome should look like this:**

Chart, scatter chart

Description automatically generated

1. Now plot a 3D graph with Plotly [Scatter3d](https://plotly.com/python/3d-scatter-plots/) samples with the age feature on the x axis, and the bmi feature on the y axis. The target charges will be on the z axis.  
   To add the regression surface , use NumPy [meshgrid](https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html), to add a slider for different degrees use Plotly [Slider](https://plotly.com/python/sliders/).

**The outcome should look like this:**

Chart, radar chart

Description automatically generated

**More Information**

Explanation on the difference between Matplotlib, Seaborn and Plotly:  
[Matplotlib vs. Seaborn vs. Plotly](https://towardsdatascience.com/matplotlib-vs-seaborn-vs-plotly-f2b79f5bddb)

Post on how to clean datasets using Pandas:  
[How To Clean Machine Learning Datasets Using Pandas](https://www.activestate.com/blog/how-to-clean-machine-learning-datasets-using-pandas/)

Explanation on the difference between scatterplot and dotplot:  
[Difference between scatter-plot and a dotplot](https://math.stackexchange.com/a/691754)

Tutorial on how to use bar charts with Plotly Express:  
[Step by step bar-charts using Plotly Express](https://towardsdatascience.com/step-by-step-bar-charts-using-plotly-express-bb13a1264a8b)

Explanation on the differences between Categorical, Ordinal and Numerical variables:  
[What is the Difference Between Categorical Ordinal and Numerical Variables?](https://stats.idre.ucla.edu/other/mult-pkg/whatstat/what-is-the-difference-between-categorical-ordinal-and-numerical-variables/)

Explanation on why it is important to define correctly categorical and ordinal features:  
[Categorical and ordinal feature data representation in regression analysis?](https://datascience.stackexchange.com/a/9211)

A package for regression tasks on ordinal target:  
[mord: Ordinal Regression in Python](https://pythonhosted.org/mord/)

Explanation on how to predict empty values:  
[Predict Missing Values in the Dataset](https://towardsdatascience.com/predict-missing-values-in-the-dataset-897912a54b7b)

Explanation on the differences between label encoding, one-hot encoding and dummy encoding:  
[One-Hot Encoding vs. Label Encoding using Scikit-Learn](https://www.analyticsvidhya.com/blog/2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/)

Wikipedia on multicollinearity:  
[Multicollinearity](https://en.wikipedia.org/wiki/Multicollinearity)

Tutorial on how to use label encoding and one-hot encoding:  
[Categorical encoding using Label-Encoding and One-Hot-Encoder](https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd)

A post on the Bias-Variance Decomposition:  
[Bias-Variance Decomposition](http://rasbt.github.io/mlxtend/user_guide/evaluate/bias_variance_decomp/)

Examples of plots in Plotly that are best for ML Regression:  
[ML Regression in Python](https://plotly.com/python/ml-regression/)

Documentation of Plotly sliders:  
[Python Figure Reference: layout.sliders](https://plotly.com/python/reference/layout/sliders/)

How to change default control values in Plotly sliders:  
[Python: Change Custom Control Values in Plotly](https://stackoverflow.com/a/58976725)

An explentation on some rare train/test scenerios:  
[How is it possible to obtain better results on the test set than on the training set?](https://www.researchgate.net/post/How_is_it_possible_to_obtain_better_results_on_the_test_set_than_on_the_training_set)