



▼ Roland's Notebook for LND - RolandZsoltNagy ©

▶ Imports

▼ Feature Engineering

▶ Categorical and Text Features (DictVectorizer, CountVectorizer, TfidfVectorizer)

▶ Derived (Polynomial) Features, Imputation of Missing Data and Feature Pipeline

▼ ML Concepts

▶ Model validation (K-Fold Cross Validation, LOOCV)

▶ Selecting the best model (Bias-variance tradeoff, Validation and Learning Curves, Grid Search)

▼ Data Visualization

▶ Load dataset to visualize

▶ Matplotlib

▶ Seaborn

▼ Linear Regression

▶ Mathematics

▶ Mathematics behind the `sklearn.datasets.make_regression()` function



Linear Regression

- ▶ Mathematics
- ▶ Mathematics behind the `sklearn.datasets.make_regression()` function
- ▶ Creating model from scratch using NumPy (with mathematics in comments)
- ▶ Applying created model on data, visualizing Gradient Descent in action and performance benchmarking
- ▶ Model extensions (Polynomial and Gaussian basis functions, Regularization techniques)

Logistic Regression

- ▶ Mathematics
- ▶ Creating model from scratch using NumPy (code structure in line with Mathematics)
- ▶ Applying created model, analyzing the training process and comparing performance to sklearn's model

Evaluation metrics

- ▶ Creating and applying class to a classification task

Roland's Notebook for LND - RolandZsoltNagy ©

Imports

```
%pylab inline import dataiku
from dataiku import pandasutils as pdu

import ms.version

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns # seaborn.set() # plot formatting

import sklearn # for sklearn.linear_model.LogisticRegression()
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.pipeline import make_pipeline
from sklearn import datasets

from sklearn.linear_model import LinearRegression, LogisticRegression
```

Matplotlib created a temporary config/cache directory at /tmp/matplotlib-a0r56rvz because the default path (/ms/dist/wirectory, in particular to speed up the import of Matplotlib and to better support multiprocessing.
`configure_inline_support` is deprecated since IPython 7.23, directly use `matplotlib_inline.backend_inline.configure_

```
# # Example: Load a DSS dataset as a Pandas dataframe
# mydataset = dataiku.Dataset("mydataset")
# mydataset_df = mydataset.get_dataframe()
```

```
"""
cool! Thank you for establishing the totebook
"""

'\n cool! Thank you for establishing the totebook\n'
```

Feature Engineering

Categorical and Text Features (DictVectorizer, CountVectorizer, TfIdfVectorizer)

```
## Categorical Features

data = [
    {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
    {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
    {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
    {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
]

# For the neighborhood feature no straightforward numerical mapping by coding, instead: one-hot encoding
# data as list of dictionaries: DictVectorizer from scikit-Learn

from sklearn.feature_extraction import DictVectorizer
vectorizer = DictVectorizer(sparse=False, dtype=int)
# own research from sklearn documentation for the 'sparse' hyperparameter:
# default=True, Whether transform should produce scipy.sparse matrices

encodedData = vectorizer.fit_transform(data)
display(encodedData)
```

```

# In fact, many machine learning models produce empty sparse matrices
encodedData = vectorizer.fit_transform(data)
display(encodedData)

array([[ 0,  1,  0, 850000,  4],
       [ 1,  0,  0, 700000,  3],
       [ 0,  0,  1, 650000,  3],
       [ 1,  0,  0, 600000,  2]])

# at this point I noticed that the new columns are in an alphabetic order:
# like the first column is for the 'Fremont' value of the neighborhood feature
# we can also check this:
print(vectorizer.get_feature_names_out())

# If the categorical feature has many possible values, this would increase too much the dataset
# Instead, set the sparse hyperparameter to True, or:

# For small datasets with a small number of possible values for categorical variable, one-hot encoding is often
# a good option. However, for large datasets with a large number of possible values for categorical variables,
# embeddings are often the better choice
# For example Categorical Embedding:
# https://arxiv.org/abs/1604.06737
# https://arxiv.labs.arxiv.org/html/1604.06737
# https://www.kaggle.com/code/colinmorris/embedding-layers

vectorizer = DictVectorizer(sparse=True, dtype=int)
encodedData = vectorizer.fit_transform(data)
display(encodedData)
print(encodedData)

# Most ML models accept this type (sparse matrix) of input data too
# Additional similar encoders: sklearn.preprocessing.OneHotEncoder, sklearn.feature_extraction.FeatureHasher

['neighborhood=Fremont' 'neighborhood=Queen Anne'
 'neighborhood=Wallingford' 'price' 'rooms']
<4x5 sparse matrix of type '<class 'numpy.int64'>'>
      with 12 stored elements in Compressed Sparse Row format>
(0, 1)      1
(0, 3)  850000
(0, 4)      4
(1, 0)      1
(1, 3)  700000
(1, 4)      3
(2, 2)      1
(2, 3)  650000
(2, 4)      3
(3, 0)      1
(3, 3)  600000
(3, 4)      2

# Text Features
sample = ['problem of evil',
          'evil queen',
          'horizon problem']

# For example encoding text data by word counts:
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X = vec.fit_transform(sample)
display(vec.get_feature_names_out())

# Convert matrix to array and then to DF
df = pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
display(df, X)

# Own addition: sum the rows to get summed word counts
print(df.sum(axis='rows'))

array(['evil', 'horizon', 'of', 'problem', 'queen'], dtype=object)

```

```

# Convert matrix to array and then to DF
df = pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
display(df, X)

# Own addition: sum the rows to get summed word counts
print(df.sum(axis='rows'))

```

```
array(['evil', 'horizon', 'of', 'problem', 'queen'], dtype=object)
```

	evil	horizon	of	problem	queen
0	1	0	1	1	0
1	1	0	0	0	1
2	0	1	0	1	0

```

<3x5 sparse matrix of type '<class 'numpy.int64'>'  

  with 7 stored elements in Compressed Sparse Row format>  

evil      2  

horizon   1  

of         1  

problem   2  

queen     1  

dtype: int64

```

```

# Problem with this approach: features which put too much weight on words that appear very frequently, which can be solved by one fix: TF-IDF (term frequency-inverse document frequency), which 'weights the word counts by a measure of how often each term appears in the corpus'
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer()
X = vec.fit_transform(sample)
display(pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out()))

# Own addition: remove stop words
vec = TfidfVectorizer(stop_words = 'english')
X = vec.fit_transform(sample)
display(pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out()))

```

	evil	horizon	of	problem	queen
0	0.51785611616769744	0.00000000000000000000	0.680918560398684036	0.51785611616769744	0.00000000000000000000
1	0.605348508106291594	0.00000000000000000000	0.00000000000000000000	0.00000000000000000000	0.795960541568165203
2	0.00000000000000000000	0.795960541568165203	0.00000000000000000000	0.605348508106291594	0.00000000000000000000

	evil	horizon	problem	queen
0	0.707106781186547573	0.00000000000000000000	0.707106781186547573	0.00000000000000000000
1	0.605348508106291594	0.00000000000000000000	0.00000000000000000000	0.795960541568165203
2	0.00000000000000000000	0.795960541568165203	0.605348508106291594	0.00000000000000000000

```

# At this point I got very interested about how the displayed values of the TfidfVectorizer are calculated, so I did my own research and found these 2 articles helpful:
# https://medium.com/@cmukesh8688/tf-idf-vectorizer-scikit-learn-dbc0244a911a
# https://www.analyticsvidhya.com/blog/2021/11/how-sklearns-tfidfvectorizer-calculates-tf-idf-values/
# With these I could understand the mathematical process of TF-IDF and then applied it on my own data by myself
# I calculated by myself all the numbers, for this used my manual calculator and also an online matrix calculator
# Please see below:

```

```

# TF-IDF = TF(t, d) * IDF(t)
# TF(t, d): term frequency - number of times term t appears in a doc d
# IDF(t) = log_e[(1+n)/(1+DF(t))] + 1, n=no. of documents (3), DF(t): document frequency of the term t

print(sample)

# TF(t, d) =
# ...

```

```

# TF-IDF = TF(t, d) X IDF(t)
# TF(t, d): term frequency - number of times term t appears in a doc d
# IDF(t) = log_e[(1+n)/(1+DF(t))] + 1, n=no. of documents (3), DF(t): document frequency of the term t

print(sample)

# TF(t, d) =
# evil, horizon, problem, queen
# [1, 0, 1, 0] first doc
# 1, 0, 0, 1 second doc
# 0, 1, 1, 0] third doc
# let's check with code:
vec = CountVectorizer(stop_words = 'english')
X = vec.fit_transform(sample)
X.toarray()

['problem of evil', 'evil queen', 'horizon problem']
array([[1, 0, 1, 0],
       [1, 0, 0, 1],
       [0, 1, 1, 0]])

# IDF(t) =
# for example idf("evil") = log_e[(1+n)/(1+DF(t))] + 1 = log_e[(1+3)/(1+2)] + 1 = 1.287682
# for example idf("horizon") = log_e[(1+n)/(1+DF(t))] + 1 = log_e[(1+3)/(1+1)] + 1 = 1.69314
# so IDF as a matrix would be:
# IDF(t) =
# evil, horizon, problem, queen
# [1.287682, 0, 0, 0]
# 1, 1.69314, 0, 0           the corpus of all documents
# 0, 0, 1.287682, 0
# 0, 0, 0, 1.69314

# IDF(t) as a vector:
# IDF(t) = (1.287682, 1.69314, 1.287682, 1.69314)
# let's check with code:
vec = TfidfVectorizer(stop_words = 'english')
X = vec.fit_transform(sample)
display(vec.idf_)

array([1.28768207, 1.69314718, 1.28768207, 1.69314718])

# as TF-IDF = TF(t, d) X IDF(t) = (3x4) X (4x4) = (3x4)
# TF-IDF =
# evil, horizon, problem, queen
# [1.28768207, 0, 1.28768207, 0] first doc
# 1.28768207, 0, 0, 1.69314718 second doc
# 0, 1.69314718, 1.28768207, 0] third doc

# To avoid large documents in the corpus dominating smaller ones,
# we have to normalize each row in the sparse matrix to have the Euclidean norm
# for example 1.28768207 / sqrt( 1.28768207^2 + 0^2 + 1.28768207^2 + 0^2 ) = 0.7071067784
# TF-IDF normalized =
# evil, horizon, problem, queen
# [0.7071067784, 0, 0.7071067784, 0] first doc
# 0.6053485058, 0, 0, 0.7959605398 second doc
# 0, 0.7959605398, 0.6053485058, 0] third doc
# and this matrix equals with the matrix that we got by TfidfVectorizer(stop_words = 'english').fit_transformed on the
# let's check with code:
display(pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out()))

# Conclusion about TF-IDF:
# a high weight of the tf-idf calculation is reached when we have a high term frequency(tf) in the given document(Loc)
# If the word is common and appears in many documents, the idf value (normalized) will approach 0 or else approach 1
# The higher the tf-idf value is, the more relevant the term is in that document

```

	evil	horizon	problem	queen
0	0.707106781186547573	0.00000000000000000000	0.707106781186547573	0.00000000000000000000
1	0.605348508106291594	0.00000000000000000000	0.00000000000000000000	0.795960541568165203

```

# as TF-IDF = TF(t, d) X IDF(t) = (3x4) X (4x4) = (3x4)
# TF-IDF =
# evil, horizon, problem, queen
# [1.28768207, 0, 1.28768207, 0    first doc
# 1.28768207, 0, 0, 1.69314718    second doc
# 0, 1.69314718, 1.28768207, 0]  third doc

# To avoid large documents in the corpus dominating smaller ones,
# we have to normalize each row in the sparse matrix to have the Euclidean norm
# for example 1.28768207 / sqrt( 1.28768207^2 + 0^2 + 1.28768207^2 + 0^2 ) = 0.7071067784
# TF-IDF normalized =
# evil, horizon, problem, queen
# [0.7071067784, 0, 0.7071067784, 0    first doc
# 0.6053485058, 0, 0, 0.7959605398    second doc
# 0, 0.7959605398 , 0.6053485058, 0]  third doc
# and this matrix equals with the matrix that we got by TfidfVectorizer(stop_words = 'english' fit_transformed on the
# let's check with code:
display(pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out()))

# Conclusion about TF-IDF:
# 'a high weight of the tf-idf calculation is reached when we have a high term frequency(tf) in the given document(Loc
# If the word is common and appears in many documents, the idf value (normalized) will approach 0 or else approach 1
# The higher the tf-idf value is, the more relevant the term is in that document

```

	evil	horizon	problem	queen
0	0.707106781186547573	0.00000000000000000000	0.707106781186547573	0.00000000000000000000
1	0.605348508106291594	0.00000000000000000000	0.00000000000000000000	0.795960541568165203
2	0.00000000000000000000	0.795960541568165203	0.605348508106291594	0.00000000000000000000

Derived (Polynomial) Features, Imputation of Missing Data and Feature Pipeline

```

# Derived Features

# we can convert linear regression into a polynomial regression not by changing the model,
# but by transforming the input. This is sometimes known as basis function regression

# For example, this data clearly cannot be well described by a straight line:
x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y)
plt.show()

from sklearn.linear_model import LinearRegression
X = x[:, np.newaxis]
print(np.newaxis)
print(x)
print(X, "\n")

model = LinearRegression().fit(X, y) # y = a * x + b
yfit = model.predict(X)
print(yfit)

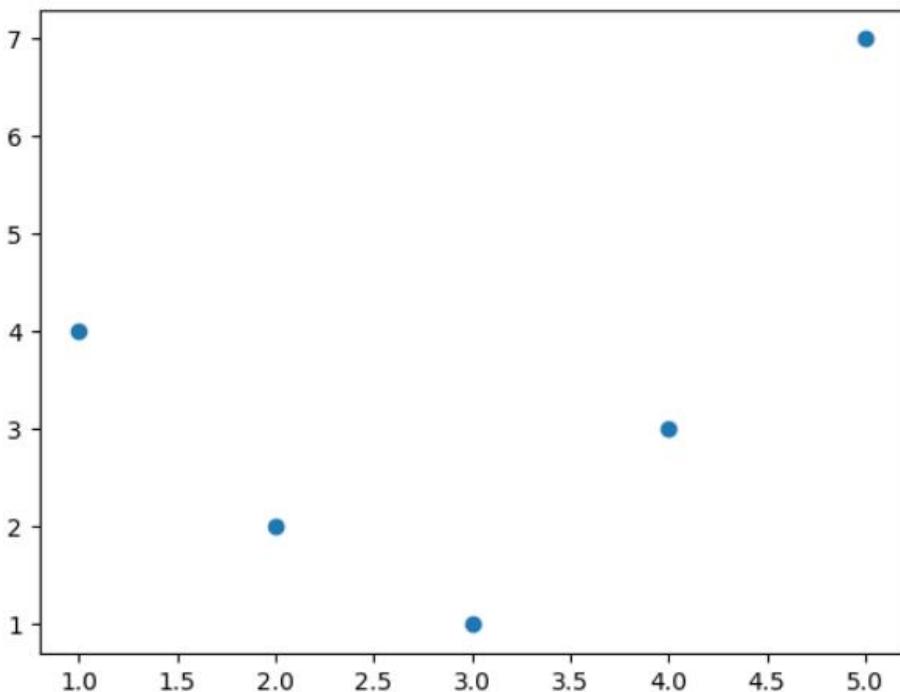
plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()

print(model.coef_)
# we need a more sophisticated model to describe the relationship between x and y

```

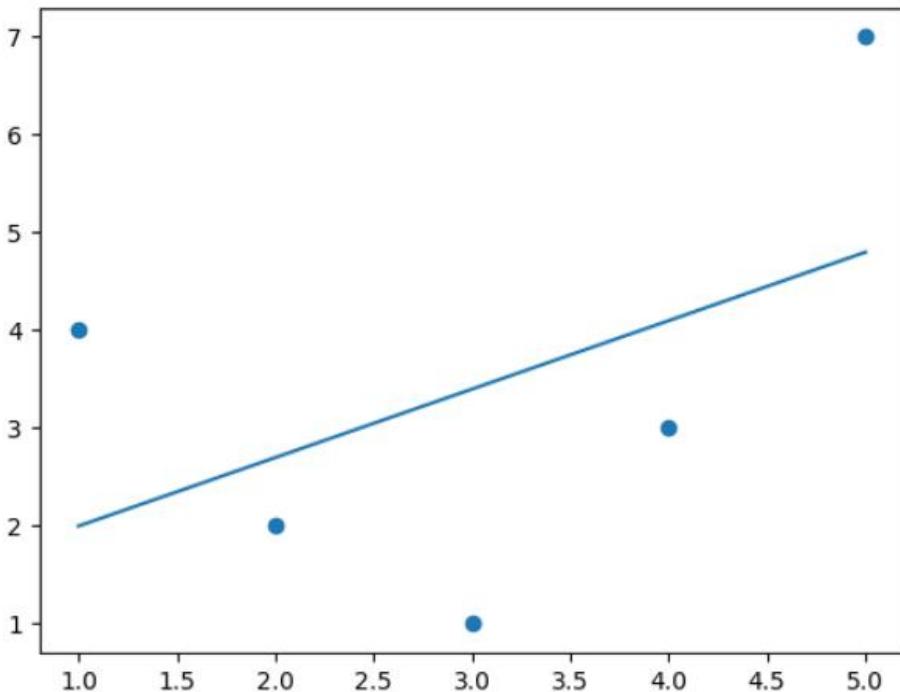


```
print(model.coef_)
# we need a more sophisticated model to describe the relationship between x and y
```



```
None
[1 2 3 4 5]
[[1]
 [2]
 [3]
 [4]
 [5]]
```

```
[2. 2.7 3.4 4.1 4.8]
```



```
[0.7]
```

```
# One approach to this is to transform the data, adding extra columns of features to drive more flexibility in the model
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=False)
# Own addition: include bias: If True (default), then include a bias column, the feature in which all polynomial power
```

```

# One approach to this is to transform the data, adding extra columns of features to drive more flexibility in the model
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=False)
# Own addition: include_bias: If True (default), then include a bias column, the feature in which all polynomial power
X2 = poly.fit_transform(X)
print(X)
print()
print("    x    x^2    x^3")
print(X2)
print()

# The derived feature matrix has one column representing x, and a second column representing x2, and a third column representing x3
# Computing a linear regression on this expanded input gives a much closer fit to our data

model = LinearRegression().fit(X2, y) # y = a*x3 + b*x2 + cx + d
yfit = model.predict(X2)
print(yfit)

plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()

print(model.intercept_)
print(model.coef_)
print(model.coef_[0] * 2 + model.coef_[1] * 4 + model.coef_[2] * 8 + model.intercept_, yfit[1])
# This idea of improving a model not by changing the model, but by transforming the inputs
# is fundamental to many of the more powerful machine learning methods. This is one motivational path to the powerful

```

```

[[1]
 [2]
 [3]
 [4]
 [5]]

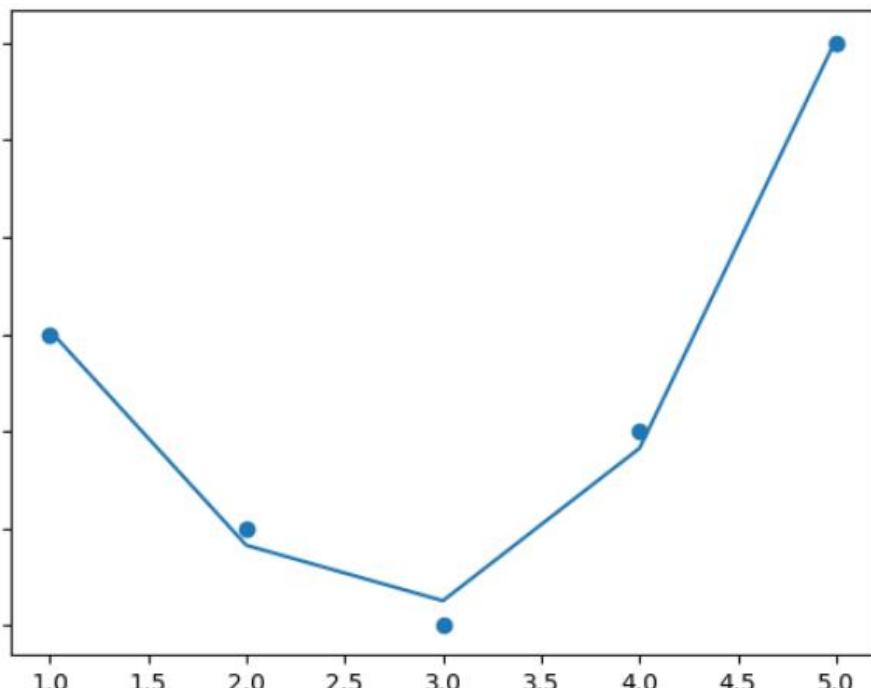
```

```

x    x^2    x^3
[[ 1.   1.   1.]
 [ 2.   4.   8.]
 [ 3.   9.  27.]
 [ 4.  16.  64.]
 [ 5.  25. 125.]]

```

```
[4.04285714 1.82857143 1.25714286 2.82857143 7.04285714]
```



```
7.399999999999769
[ 3.761904176  0.321402857  0.082222221]
```

```
7.399999999999769
[-3.76190476  0.32142857  0.08333333]
1.8285714285712622 1.8285714285712622
```

```
# Imputation of Missing Data

X = np.array([[ np.nan,  0,   3  ],
              [ 3,    7,   9  ],
              [ 3,    5,   2  ],
              [ 4,    np.nan, 6  ],
              [ 8,    8,   1  ]])
print(X)
print()

y = np.array([14, 16, -1,  8, -5])
print(y)
print()

try:
    model = LinearRegression().fit(X, y)
except ValueError as e:
    print("ML model cannot be fitted to this data, valueError::", e)
#except Exception as e:
#    print("ML model cannot be fitted to this data, Exception::", e)
print()

# Before fitting an ML model to such data we will need to first replace the missing data with some appropriate fill value.
# This is known as imputation of missing values, strategies range from simple (e.g., replacing missing values with the mean).
# For a baseline imputation approach, using the mean, median, or most frequent value, Scikit-Learn provides the Imputer class.
from sklearn.impute import SimpleImputer
imp = SimpleImputer(strategy='mean')
X2 = imp.fit_transform(X)
print(X2)
# the two missing values have been replaced with the mean of the remaining values in the column

# This imputed data can then be fed directly into, for example, a LinearRegression estimator:
model = LinearRegression().fit(X2, y)
y_pred = model.predict(X2)
print(y_pred)
```

```
[[nan  0.  3.]
 [ 3.  7.  9.]
 [ 3.  5.  2.]
 [ 4. nan  6.]
 [ 8.  8.  1.]]
```

```
[14 16 -1  8 -5]
```

```
ML model cannot be fitted to this data, valueError:: Input contains NaN, infinity or a value too large for dtype('float64')
```

```
[[4.5 0.  3. ]
 [3.  7.  9. ]
 [3.  5.  2. ]
 [4.  5.  6. ]
 [8.  8.  1. ]]
[13.14869292 14.3784627 -1.15539732 10.96606197 -5.33782027]
```

```
# Feature Pipelines
```

```
# For example, we might want a processing pipeline that looks something like this:
# 1. Impute missing values using the mean
# 2. Transform features to quadratic
# 3. Fit a linear regression
# for this type of processing pipeline Scikit-Learn provides a Pipeline object
from sklearn.pipeline import make_pipeline

model = make_pipeline(SimpleImputer(strategy='mean'),
                      PolynomialFeatures(degree=2),
                      LinearRegression())
```

```

# 3. Fit a Linear regression
# for this type of processing pipeline Scikit-Learn provides a Pipeline object
from sklearn.pipeline import make_pipeline

model = make_pipeline(SimpleImputer(strategy='mean'),
                      PolynomialFeatures(degree=2),
                      LinearRegression())
# This pipeline looks and acts like a standard Scikit-Learn object
model.fit(X, y)
print(model.predict(X))
print(y)

print()

print(X)
imp = SimpleImputer(strategy='mean')
X2 = imp.fit_transform(X)
poly = PolynomialFeatures(degree=2, include_bias=True)
X2 = poly.fit_transform(X2)
print(X2)

print(model.steps[2][1].coef_)
print(model.steps[2][1].intercept_)

[14. 16. -1. 8. -5.]
[14 16 -1 8 -5]

[[nan 0. 3.]
 [ 3. 7. 9.]
 [ 3. 5. 2.]
 [ 4. nan 6.]
 [ 8. 8. 1.]]
[[ 1. 4.5 0. 3. 20.25 0. 13.5 0. 0. 9. ]
 [ 1. 3. 7. 9. 9. 21. 27. 49. 63. 81. ]
 [ 1. 3. 5. 2. 9. 15. 6. 25. 10. 4. ]
 [ 1. 4. 5. 6. 16. 20. 24. 25. 30. 36. ]
 [ 1. 8. 8. 1. 64. 64. 8. 64. 8. 1. ]]
[-6.38378239e-16 3.14294638e-02 -1.13545415e-01 -2.58101282e-03
 3.56096859e-01 -3.66324628e-01 -1.15167185e-01 -1.09315710e-01
 -1.06212696e-01 3.91096967e-01]
4.690233341938034

```

ML Concepts

Model validation (K-Fold Cross Validation, LOOCV)

```

from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
print(X.shape, '\n', X[:5])

y = iris.target
print(y.shape, '\n', y[:5])

from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)

(150, 4)
[[5.1 3.5 1.4 0.2]
 [4.9 3. 1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5. 3.6 1.4 0.2]]
(150,)
[0 0 0 0 0]

# Here we do two validation trials, alternately using each half of the data as a holdout set
X1 X2 y1 y2 = train_test_split(X, y, test_size=0.5)

```

```
# Here we do two validation trials, alternately using each half of the data as a holdout set
X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                   train_size=0.5)

y2_model = model.fit(X1, y1).predict(X2)
print(y2.shape, '\n', y2[10:20])
print(y2_model.shape, '\n', y2_model[10:20])

y1_model = model.fit(X2, y2).predict(X1)

print(accuracy_score(y1, y1_model), accuracy_score(y2, y2_model))
# We could combine the two accuracy scores(by, say, taking the mean) to get a better measure of the global model performance
# This particular form of cross-validation is a two-fold cross-validation
np.mean([accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)])
```

```
# Five-fold cross-validation
from sklearn.model_selection import cross_val_score
print(cross_val_score(model, X, y, cv=5))
```

```
[0.96666667 0.96666667 0.93333333 0.93333333 1. ]
```

```
# Scikit-Learn implements a number of useful cross-validation schemes
# For example: our number of folds is equal to the number of data points: that is, we train on all points but one in each trial
# This type of cross-validation is known as leave-one-out cross validation
from sklearn.model_selection import LeaveOneOut
print(len(X))

scores = cross_val_score(model, X, y, cv=LeaveOneOut())
print(scores)

#Taking the mean of these gives an estimate of the error rate:
scores.mean()

# Other cross-validation schemes can be used similarly
```

Selecting the best model (Bias-variance tradeoff, Validation and Learning Curves, Grid Search)

```
# The Validation Curve

## This code snippet is only for illustration purposes
x = np.linspace(0, 1, 1000)
y1 = -(x - 0.5) ** 2
y2 = y1 - 0.33 + np.exp(x - 1)

fig, ax = plt.subplots()
ax.plot(x, y2, lw=10, alpha=0.5, color='blue')
ax.plot(x, y1, lw=10, alpha=0.5, color='red')

ax.text(0.25, 0.15, "training score", rotation=35, size=12, color='blue')
ax.text(0.75, 0.15, "validation score", rotation=-35, size=12, color='red')
```

```

fig, ax = plt.subplots()
ax.plot(x, y2, lw=10, alpha=0.5, color='blue')
ax.plot(x, y1, lw=10, alpha=0.5, color='red')

ax.text(0.25, 0.15, "training score", rotation=35, size=12, color='blue')
ax.text(0.2, -0.025, "validation score", rotation=15, size=12, color='red')

ax.text(0.02, 0.1, r'$\longleftarrow$ High Bias', size=12, rotation=90, va='center')
ax.text(0.98, 0.1, r'$\longleftarrow$ High Variance $\longrightarrow$', size=12, rotation=90, ha='right', va='center')
ax.text(0.47, -0.12, 'Best$\longrightarrow$\nModel', size=12, rotation=90, va='center')

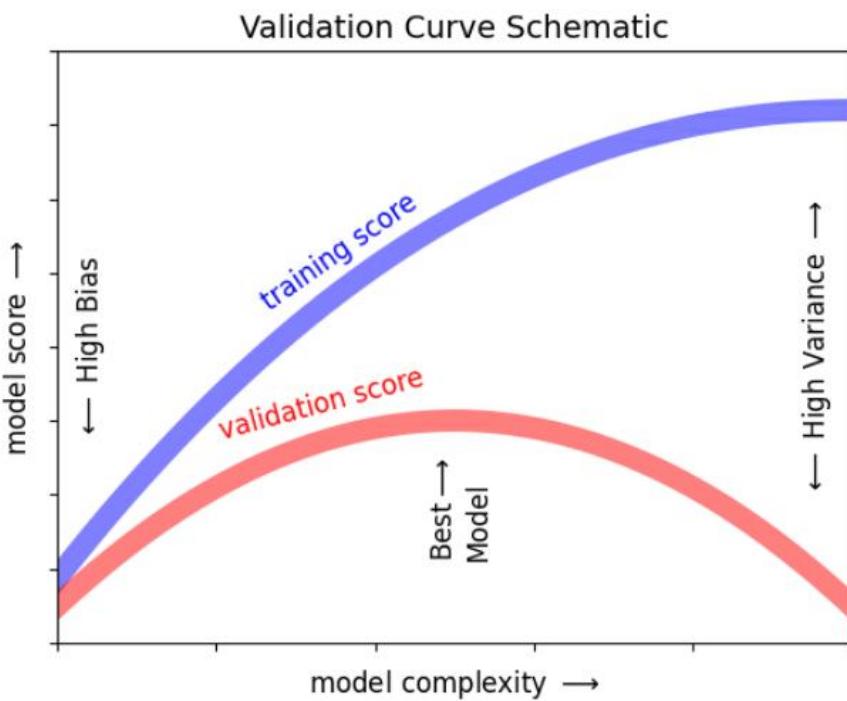
ax.set_xlim(0, 1)
ax.set_ylim(-0.3, 0.5)

ax.set_xlabel(r'model complexity $\longrightarrow$', size=12)
ax.set_ylabel(r'model score $\longrightarrow$', size=12)

ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.yaxis.set_major_formatter(plt.NullFormatter())

ax.set_title("Validation Curve Schematic", size=14)
plt.show()

```



```

# Validation curves in Scikit-Learn

## pipeline to combine a simple linear regression with the polynomial preprocessor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
                         LinearRegression(**kwargs))

```

```

# for the understanding of the next cell

display(np.random.RandomState(1), np.random.RandomState(1), np.random.RandomState(2), np.random.RandomState(3))
# seed parameter: Random seed used to initialize the pseudo-random number generator

x = np.random.RandomState(1).rand(8, 2)

```

```

# for the understanding of the next cell

display(np.random.RandomState(1), np.random.RandomState(1), np.random.RandomState(2), np.random.RandomState(3))
# seed parameter: Random seed used to initialize the pseudo-random number generator

x = np.random.RandomState(1).rand(8,2)
print(x, "\n")
# Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

# ravel: Return a contiguous flattened array
xravel = x.ravel() + 0.1
print(xravel, "\n")

print(1. / xravel)
print(1 / xravel)
print()

print(10 - 1. / xravel)

```

```

RandomState(MT19937) at 0x2B56F7E24E20
RandomState(MT19937) at 0x2B57366088D0
RandomState(MT19937) at 0x2B57366089E0
RandomState(MT19937) at 0x2B5736608AF0
[[4.17022005e-01 7.20324493e-01]
 [1.14374817e-04 3.02332573e-01]
 [1.46755891e-01 9.23385948e-02]
 [1.86260211e-01 3.45560727e-01]
 [3.96767474e-01 5.38816734e-01]
 [4.19194514e-01 6.85219500e-01]
 [2.04452250e-01 8.78117436e-01]
 [2.73875932e-02 6.70467510e-01]]

```

```

[0.517022 0.82032449 0.10011437 0.40233257 0.24675589 0.19233859
 0.28626021 0.44556073 0.49676747 0.63881673 0.51919451 0.7852195
 0.30445225 0.97811744 0.12738759 0.77046751]

```

```

[1.93415365 1.2190298 9.98857558 2.48550594 4.05258815 5.19916453
 3.49332516 2.24436298 2.01301424 1.56539418 1.92606041 1.27352925
 3.28458732 1.02237212 7.85005804 1.29791326]
[1.93415365 1.2190298 9.98857558 2.48550594 4.05258815 5.19916453
 3.49332516 2.24436298 2.01301424 1.56539418 1.92606041 1.27352925
 3.28458732 1.02237212 7.85005804 1.29791326]

```

```

[8.06584635 8.7809702 0.01142442 7.51449406 5.94741185 4.80083547
 6.50667484 7.75563702 7.98698576 8.43460582 8.07393959 8.72647075
 6.71541268 8.97762788 2.14994196 8.70208674]

```

```

# let's create some data to which we will fit our model
def make_data(N, err=1.0, rseed=1):

```

```

    # randomly sample the data
    rng = np.random.RandomState(rseed)

    X = rng.rand(N, 1) ** 2
    y = 10 - 1. / (X.ravel() + 0.1)

    if err > 0:
        y += err * rng.randn(N)

    return X, y

```

```

X, y = make_data(40)
print(X[:5])
print(y[:5])

```

```

[[1.73907352e-01]
 [5.18867376e-01]
 [1.30815988e-08]
 [9.14049845e-02]
 [2.15372915e-02]]
[ 7.24009644  7.70041705  0.12299993  2.92070673  1.50419151]

```

```

x, y = make_data(40)
print(X[:5])
print(y[:5])

[[1.73907352e-01]
 [5.18867376e-01]
 [1.30815988e-08]
 [9.14049845e-02]
 [2.15372915e-02]]
[ 7.24998644  7.70041705 -0.12288892  3.83970623  1.50418461]

```

```

# for the understanding of the next cell

print(np.linspace(-0.1, 1.1, 10))
print()
print(np.linspace(-0.1, 1.1, 10)[:, None])
print(np.linspace(-0.1, 1.1, 10)[:, np.newaxis])

[-0.1      0.03333333  0.16666667  0.3      0.43333333  0.56666667
 0.7      0.83333333  0.96666667  1.1      ]
[[[-0.1      ]
 [ 0.03333333]
 [ 0.16666667]
 [ 0.3      ]
 [ 0.43333333]
 [ 0.56666667]
 [ 0.7      ]
 [ 0.83333333]
 [ 0.96666667]
 [ 1.1      ]]
[[[-0.1      ]
 [ 0.03333333]
 [ 0.16666667]
 [ 0.3      ]
 [ 0.43333333]
 [ 0.56666667]
 [ 0.7      ]
 [ 0.83333333]
 [ 0.96666667]
 [ 1.1      ]]]
```

```

# Pipeline to combine a simple linear regression with the polynomial preprocessor

def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
                         LinearRegression(**kwargs))

X, y = make_data(40)

# We can now visualize our data, along with polynomial fits of several degrees
X_test = np.linspace(-0.1, 1.1, 500)[:, None]

plt.figure(figsize=(15,8))
plt.scatter(X.ravel(), y, color='black')

for degree in [0, 1, 3, 5]:
    y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)

    plt.plot(X_test.ravel(), y_test, label=f'degree={degree}', linewidth=4, alpha=0.8)

# Own addition: checking R2 on the training data
model = PolynomialRegression(degree).fit(X, y)
print(degree, model.score(X, y))

plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best', fontsize="10")
plt.show()
```

```

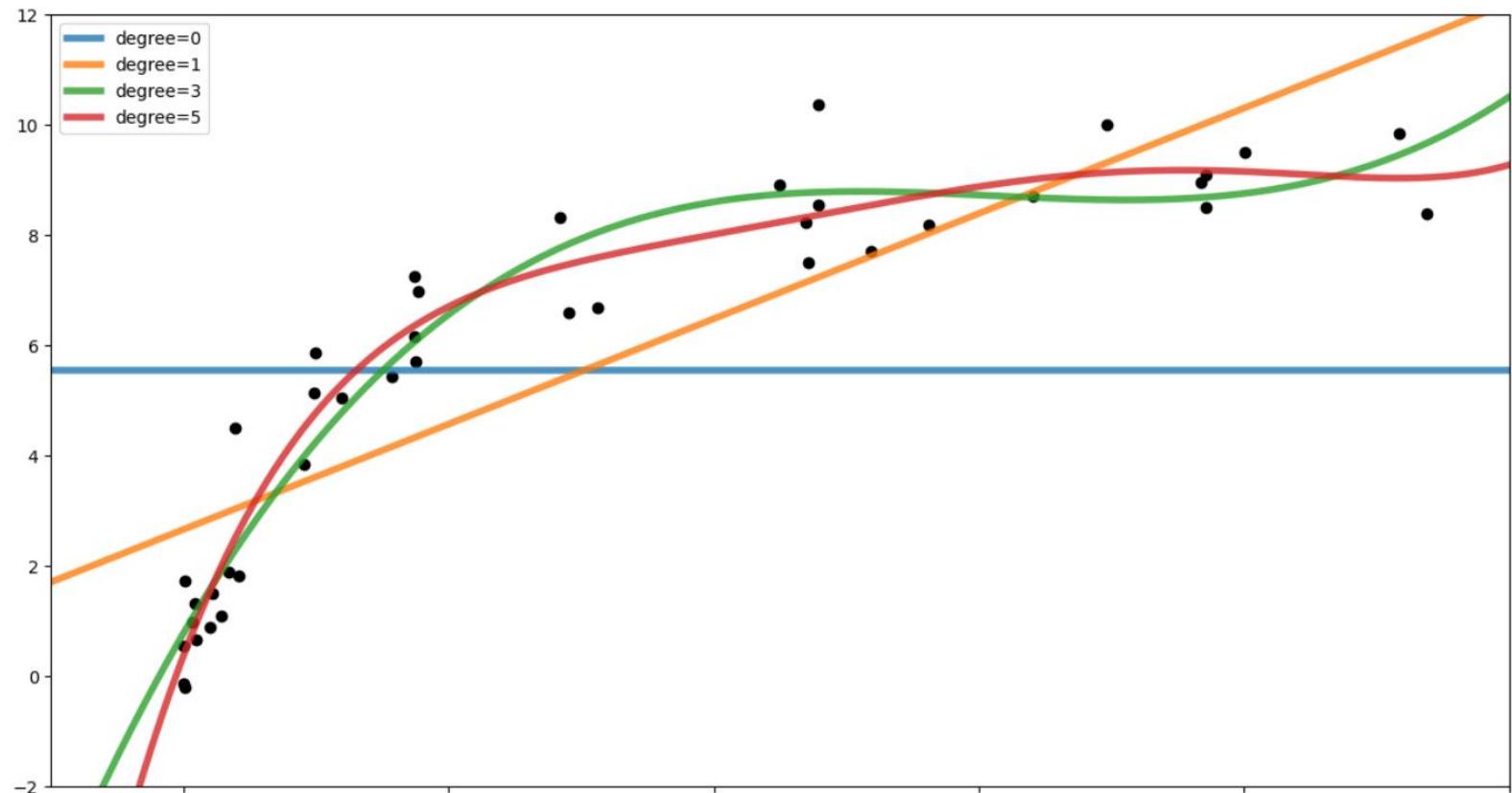
# Own addition: checking R2 on the training data
model = PolynomialRegression(degree).fit(X, y)
print(degree, model.score(X, y))

plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best', fontsize="10")
plt.show()

poly = PolynomialFeatures(degree=0, include_bias=True)
X = poly.fit_transform(X)
print(X.shape)
print(y.mean())

```

0 0.0
1 0.7243243045065821
3 0.9328514024067204
5 0.9459786646758395



(40, 1)
5.554274294786664

```

# Visualizing the validation curve for this particular data and model
X, y = make_data(40)

from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)

```

```

# Visualizing the validation curve for this particular data and model

X, y = make_data(40)

from sklearn.model_selection import validation_curve

degree = np.arange(0, 21)

train_score, val_score = validation_curve(PolynomialRegression(), X, y,
                                         param_name='polynomialfeatures_degree', param_range=degree, cv=7)
# validation_curve(estimator, X, y, *, param_name, param_range, groups=None, cv=None, etc...)

print(train_score[:10], train_score.shape, "\n")
print(val_score[:10], val_score.shape)
# Own question: Which metric is used for the score value?

plt.figure(figsize=(20,10))
plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')

plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score')
plt.xticks(np.linspace(0, 21, 22))
plt.show()

print(np.median(val_score, 1).max(), np.argmax(np.median(val_score, 1)))

```

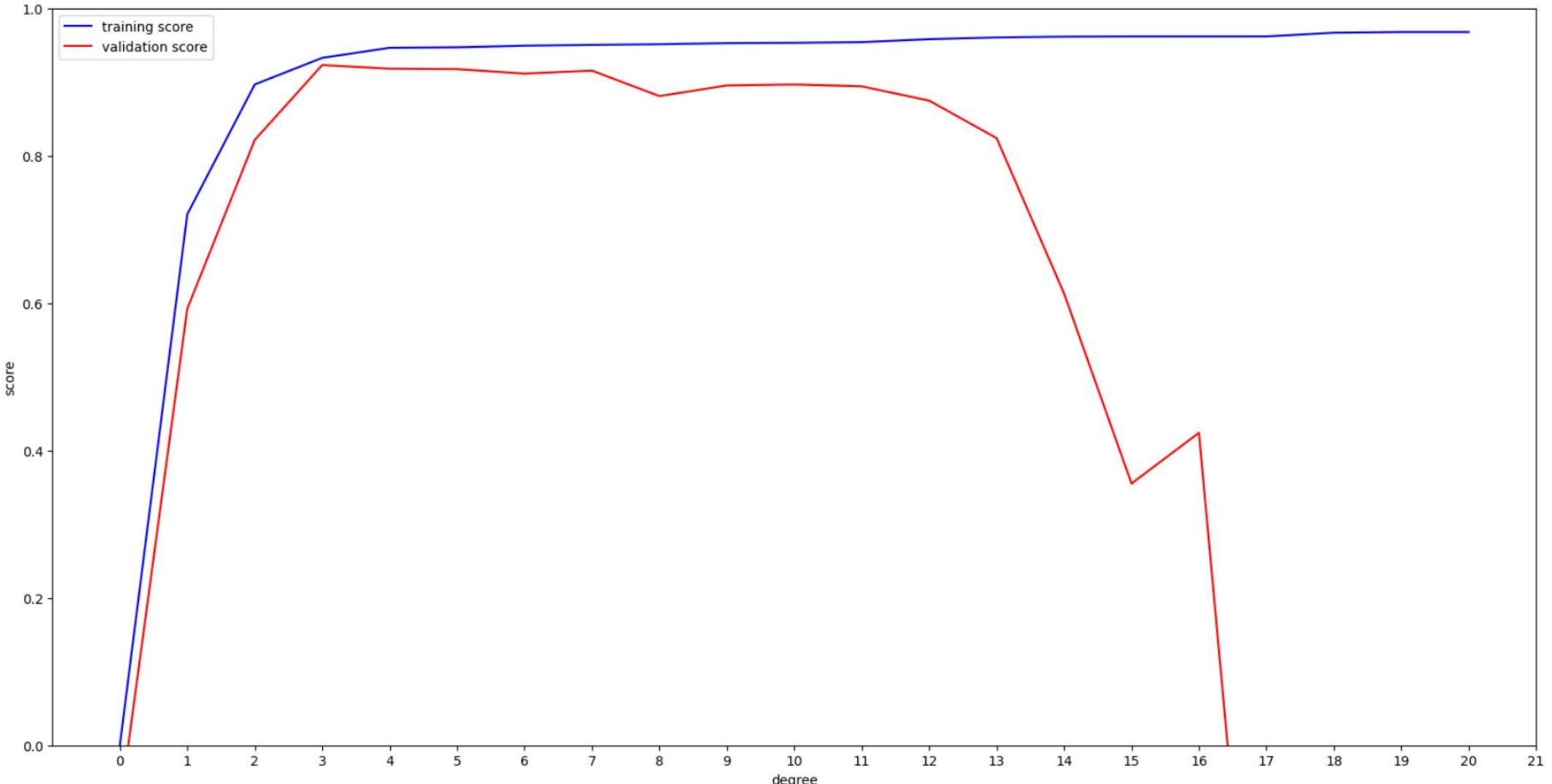
```

[[0.          0.          0.          0.          0.          0.
 0.          ]
[0.72060505 0.73235327 0.71615237 0.75517775 0.6979951  0.74351307
 0.71104367]
[0.8966661  0.89427274 0.88460403 0.90482859 0.87685741 0.90106431
 0.90420325]
[0.92830604 0.93586649 0.93273606 0.95501615 0.91919623 0.92673491
 0.9402878 ]
[0.9386132  0.94947024 0.94775706 0.9632512  0.93425141 0.94148127
 0.94641559]
[0.94121068 0.94988755 0.94891078 0.9637873  0.93481636 0.94464031
 0.94706837]
[0.94280027 0.95011515 0.95028793 0.96555698 0.93622172 0.94504929
 0.94928043]
[0.94465044 0.95166164 0.95034312 0.96584535 0.93731514 0.94571392
 0.95274503]
[0.94465097 0.95169173 0.95126473 0.96709791 0.93735454 0.9484231
 0.95274521]
[0.94935834 0.9537075  0.95317786 0.96799601 0.93964532 0.9490145
 0.9527462 ]] (21, 7)

[[-0.60246643 -0.00204615 -0.00514137 -0.08448036 -0.04342093 -0.21918928
 -0.09456538]
[ 0.5924168   0.52955915  0.78017784  0.49009912  0.76508479  0.52353779
 0.79740092]
[ 0.82135627  0.89189957  0.96373583  0.81295458  0.93423775  0.8077944
 0.80337792]
[ 0.92298132  0.83916959  0.92509741  0.7186875   0.96782936  0.95496284
 0.86925102]
[ 0.95481913  0.79122363  0.91532177  0.79557168  0.96633248  0.93954673
 0.91808179]
[ 0.94566936  0.80611696  0.91738597  0.79549816  0.97270522  0.91750742
 0.92275223]
[ 0.9381225   0.81633602  0.9113703   0.7191907   0.96464074  0.92203214
 0.90379357]
[ 0.93621355  0.80554842  0.91542385  0.7535823   0.96605322  0.93453324
 0.88097397]
[ 0.93624908  0.80567665  0.90156766  0.48096379  0.9643832   0.73304398
 0.8808854 ]
[ 0.92096749  0.81695135  0.90496397  0.80149852  0.97568253  0.89531322
 0.92096749]] (21, 7)

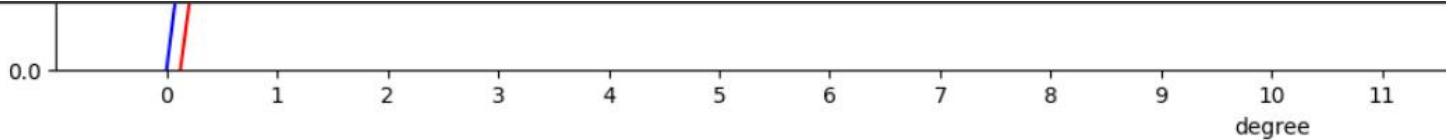
```

```
[ 0.92298132  0.83916959  0.92509741  0.7186875   0.96782936  0.95496284
 0.86925102]
[ 0.95481913  0.79122363  0.91532177  0.79557168  0.96633248  0.93954673
 0.91808179]
[ 0.94566936  0.80611696  0.91738597  0.79549816  0.97270522  0.91750742
 0.92275223]
[ 0.9381225   0.81633602  0.9113703   0.7191907   0.96464074  0.92203214
 0.90379357]
[ 0.93621355  0.80554842  0.91542385  0.7535823   0.96605322  0.93453324
 0.88097397]
[ 0.93624908  0.80567665  0.90156766  0.48096379  0.9643832   0.73304398
 0.8808854 ]
[ 0.92096749  0.81695135  0.90496397  0.80149852  0.97568253  0.89531322
 0.88200911]] (21, 7)
```



0.922981319057214 3

```
# We can see that the optimal trade-off between bias and variance is found for the third-order polynomial
# we can compute and display this fit over the original data
```



0.922981319057214 3

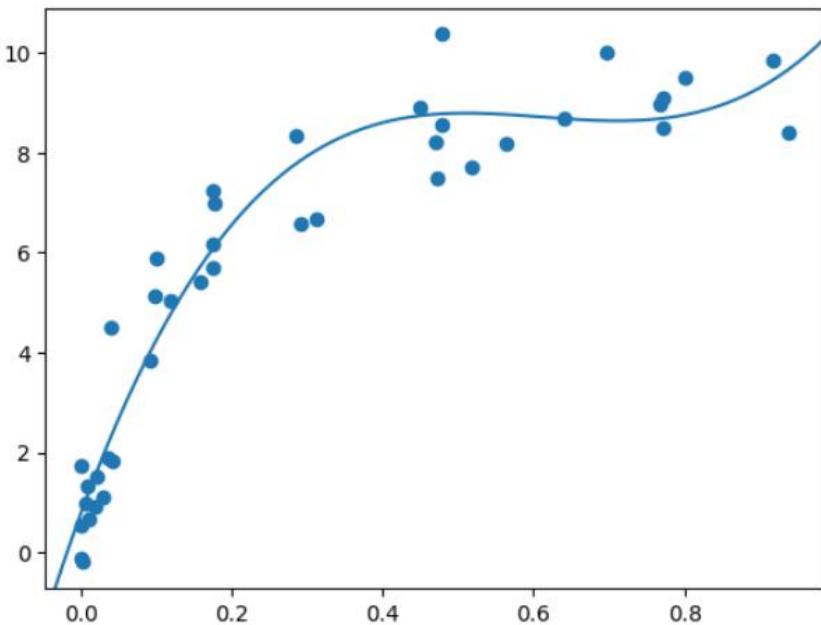
```
# We can see that the optimal trade-off between bias and variance is found for the third-order polynomial
# we can compute and display this fit over the original data
plt.scatter(X.ravel(), y)
lim = plt.axis()

X_test = np.linspace(-0.1, 1.1, 500)[:, None]
print(X_test.shape)

y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
print(y_test.shape)

plt.plot(X_test.ravel(), y_test);
plt.axis(lim)
plt.show()
```

(500, 1)
(500,)



```
# Learning Curves

# The optimal model will generally depend on the size of your training data
X2, y2 = make_data(200)

degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
                                             param_name='polynomialfeatures_degree', param_range=degree, cv=7)

plt.figure(figsize=(15,6))
plt.plot(degree, np.median(train_score2, 1), color='blue', label='training score with 200 obs data')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score with 200 obs data')

plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3, linestyle='dashed', label='training score with 50 obs data')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3, linestyle='dashed', label='validation score with 50 obs data')

plt.legend(loc='lower center')
plt.ylim(0, 1)
```

```

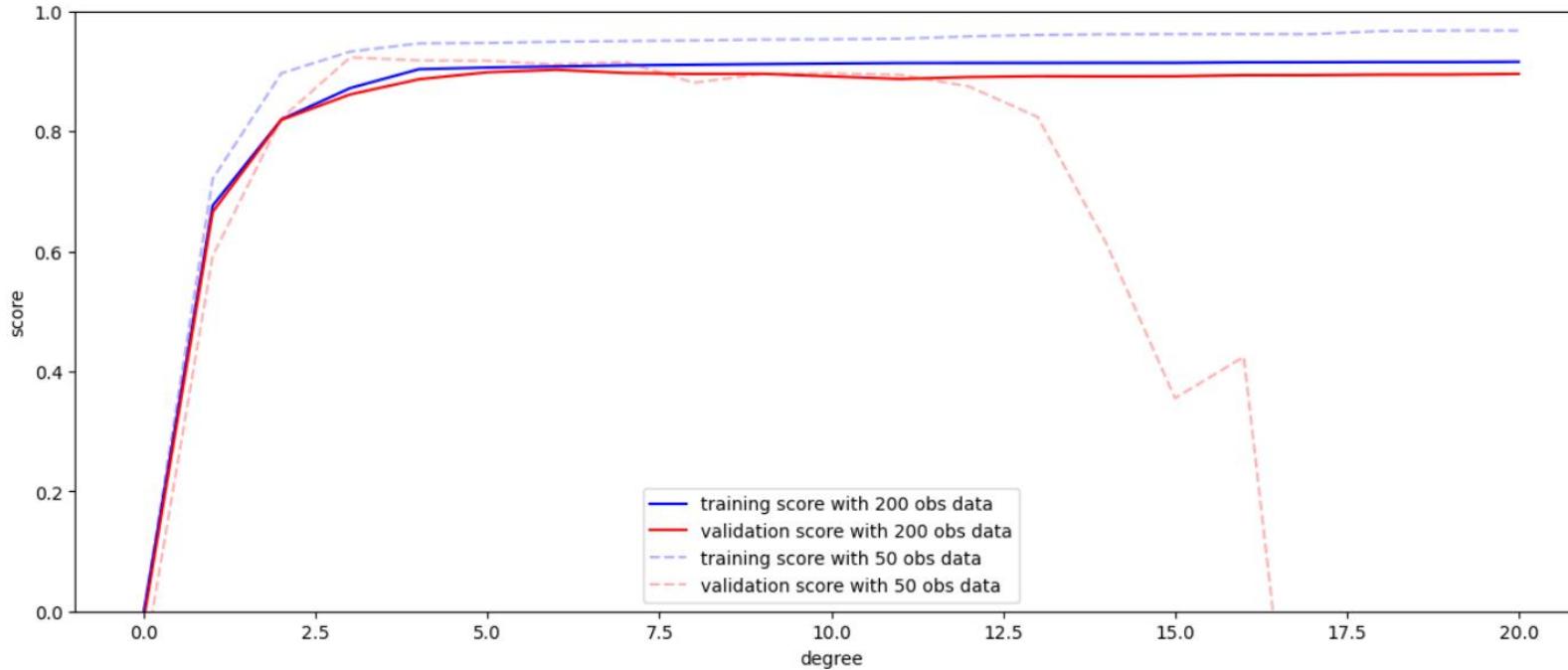
plt.plot(degree, np.median(train_score2, 1), color='blue', label='training score with 200 obs data')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score with 200 obs data')

plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3, linestyle='dashed', label='training score with 50 obs data')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3, linestyle='dashed', label='validation score with 50 obs data')

plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score')

plt.show()
# The larger dataset can support a much more complicated model
# The peak here is probably around a degree of 6, but even a degree-20 model is not seriously over-fitting the data

```



```

# It is often useful to explore the behavior of the model as a function of the number of training points,
# which we can do by using increasingly larger subsets of the data to fit our model

# The general behavior we would expect from a learning curve:

# A model of a given complexity will overfit a small dataset
# the training score will be relatively high, while the validation score will be relatively low

# A model of a given complexity will underfit a large dataset
# the training score will decrease, but the validation score will increase

# A model will never, except by chance, give a better score to the validation set than the training set
# the curves should keep getting closer together but never cross

N = np.linspace(0, 1, 1000)
x = N
y1 = 0.75 + 0.2 * np.exp(-4 * N)
y2 = 0.7 - 0.6 * np.exp(-4 * N)

fig, ax = plt.subplots()
ax.set_title('Learning Curve')

```

```

# the curves should keep getting closer together but never cross

N = np.linspace(0, 1, 1000)
x = N
y1 = 0.75 + 0.2 * np.exp(-4 * N)
y2 = 0.7 - 0.6 * np.exp(-4 * N)

fig, ax = plt.subplots()
fig.set_figheight(6)
fig.set_figwidth(12)
ax.plot(x, y1, lw=10, alpha=0.5, color='blue')
ax.plot(x, y2, lw=10, alpha=0.5, color='red')

ax.text(0.2, 0.83, "training score", rotation=-8, size=16, color='blue')
ax.text(0.2, 0.5, "validation score", rotation=20, size=16, color='red')

ax.text(0.98, 0.45, r'Good Fit $\rightarrow$', size=18, rotation=90, ha='right', va='center')
ax.text(0.02, 0.57, r'$\leftarrow$ High Variance $\rightarrow$', size=18, rotation=90, va='center')

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

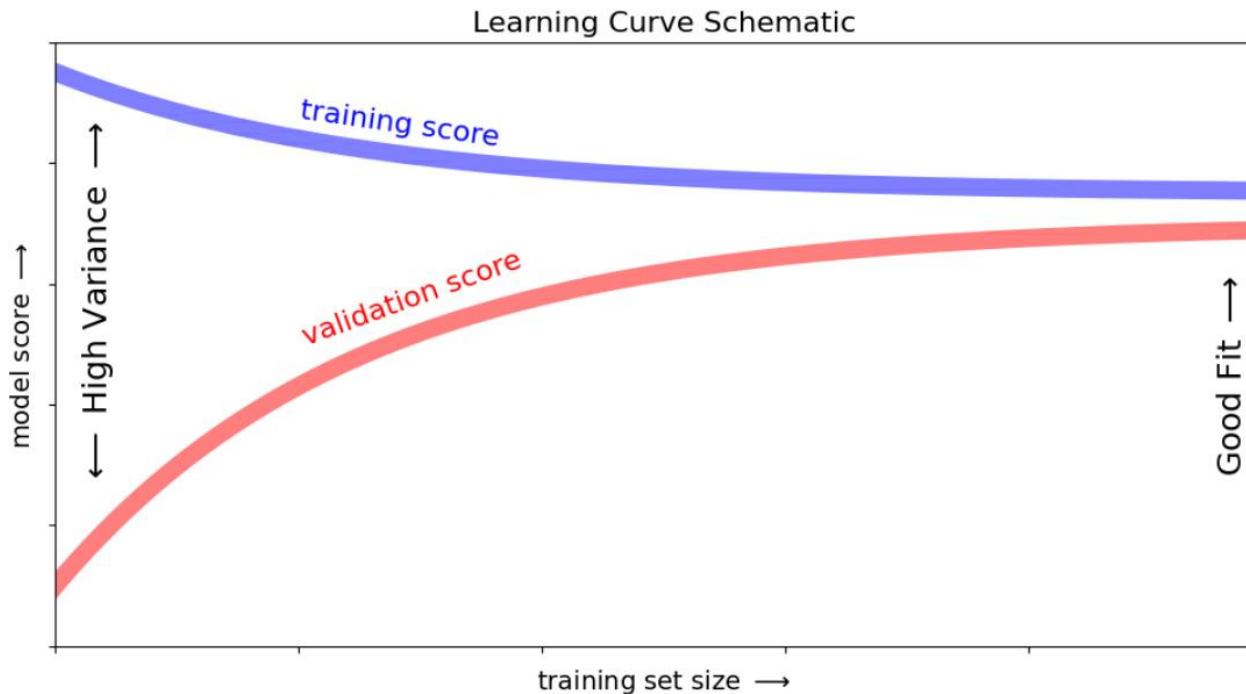
ax.set_xlabel(r'training set size $\rightarrow$', size=14)
ax.set_ylabel(r'model score $\rightarrow$', size=14)

ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.yaxis.set_major_formatter(plt.NullFormatter())

ax.set_title("Learning Curve Schematic", size=16)
plt.show()

# There is a convergence to a particular score as the number of training samples grows
# Once you have enough points that a particular model has converged, adding more training data will not help you
# The only way to increase model performance in this case is to use another (often more complex) model

```



training set size →

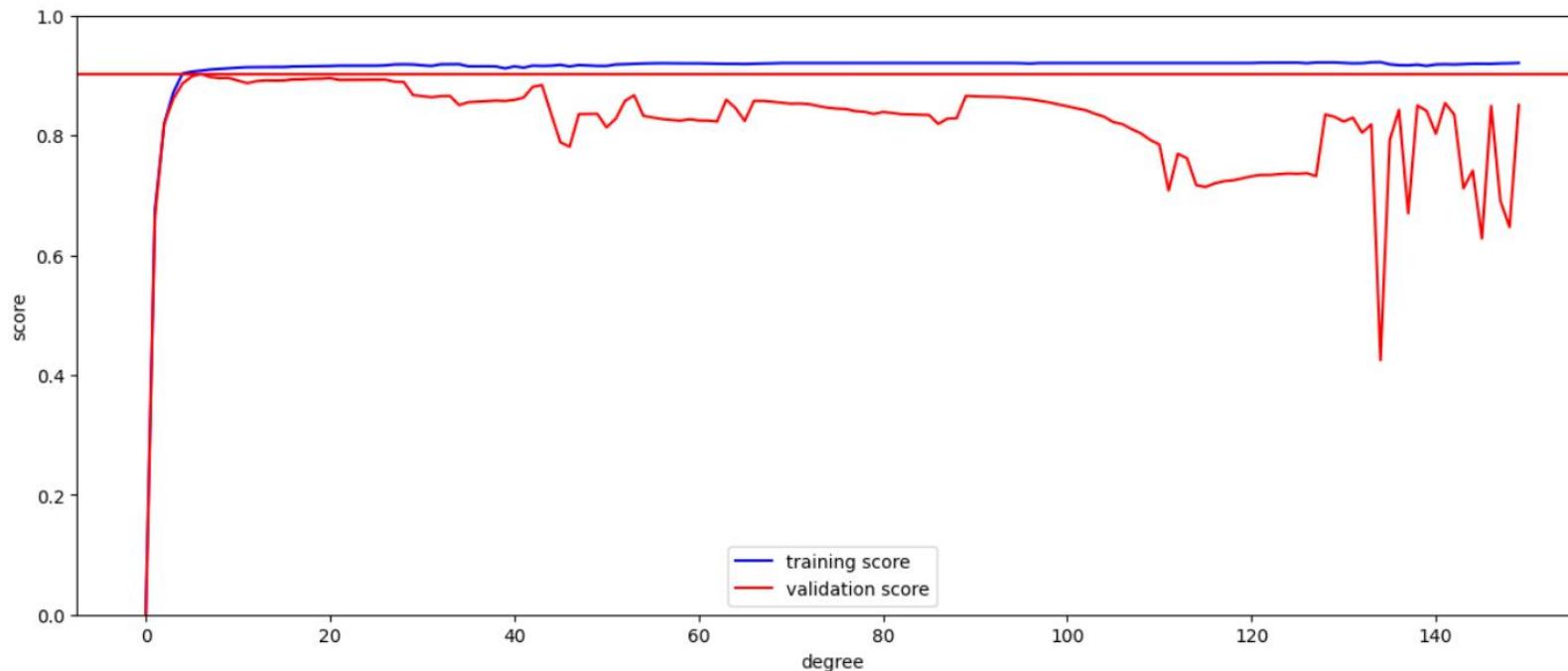
```
# Own research
X2, y2 = make_data(200)

degree = np.arange(150)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
                                             param_name='polynomialfeatures_degree', param_range=degree, cv=7)

plt.figure(figsize=(15,6))
plt.plot(degree, np.median(train_score2, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.axhline(y = np.median(val_score2, 1).max(), color = 'r', linestyle = '-')

plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score')
plt.show()

print(np.median(val_score2, 1).max(), np.argmax(np.median(val_score2, 1)))
print(np.median(train_score2, 1).max(), np.argmax(np.median(train_score2, 1)))
```



0.9024627943209514 6
0.9221360259163539 134

```
# Learning curves in Scikit-Learn

from sklearn.model_selection import learning_curve

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve(PolynomialRegression(degree),
                                         X, y, cv=7)
```

```

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve(PolynomialRegression(degree),
                                         X, y, cv=7,
                                         train_sizes=np.linspace(0.3, 1, 25))
    print(N, N.shape)
    print(train_lc.shape, val_lc.shape)

    ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
    ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')

    print([train_lc[-1], val_lc[-1]])
    print(np.mean(val_lc[-1]), np.median(val_lc[-1]))
    print()

    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1],
                 color='gray', linestyle='dashed')

    ax[i].set_xlim(0, 1)
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_yticks(np.linspace(0, 1, 11))
    ax[i].set_title('degree = {}'.format(degree), size=14)
    ax[i].legend(loc='best')

plt.show()
# When your Learning curve has already converged (i.e., when the training and validation curves are already close to each other)
# adding more training data will not significantly improve the fit

# The only way to increase the converged score is to use a different (usually more complicated) model
# In the right panel by moving to a much more complicated model, we increase the score of convergence (dashed line),
# But at the expense of higher model variance (the difference between the training and validation scores).
# If we were to add more data points, the Learning curve for the more complicated model would eventually converge

```

```

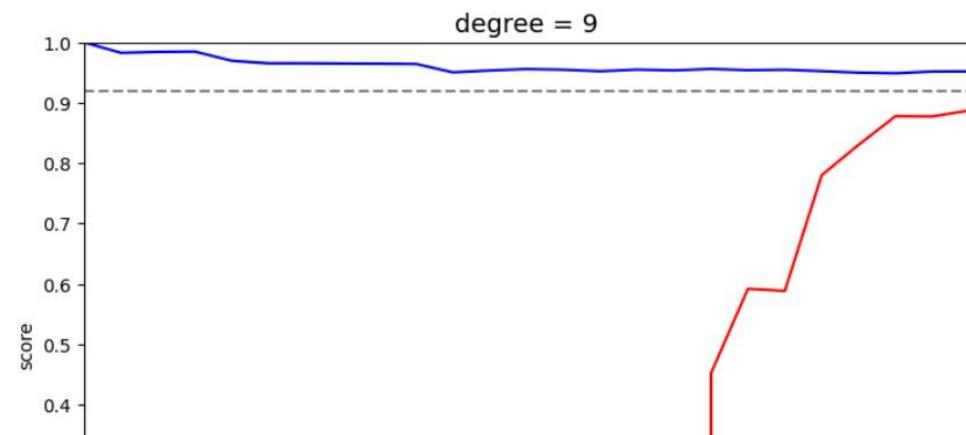
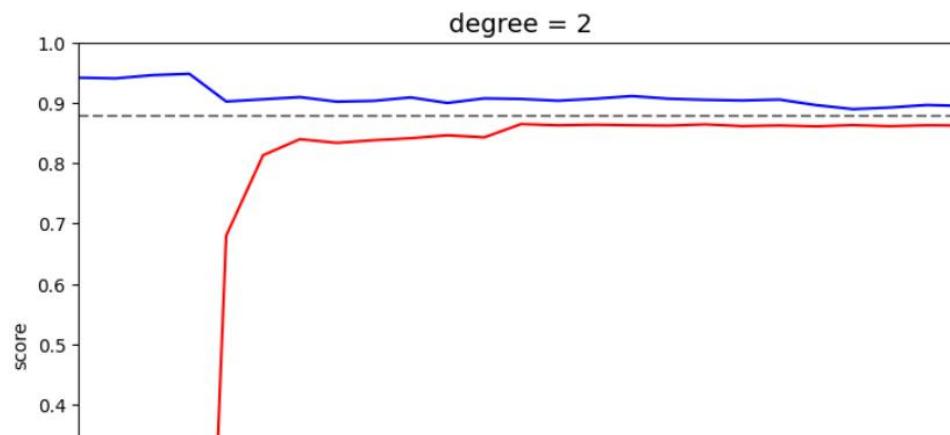
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34] (25,)
(25, 7) (25, 7)
[array([0.8966661 , 0.89427274, 0.88460403, 0.90482859, 0.87685741,
       0.90271579, 0.9029431 ], array([0.82135627, 0.89189957, 0.96373583, 0.81295458, 0.93423775,
       0.81451818, 0.79718561])]
0.8622696815670261 0.8213562669413159

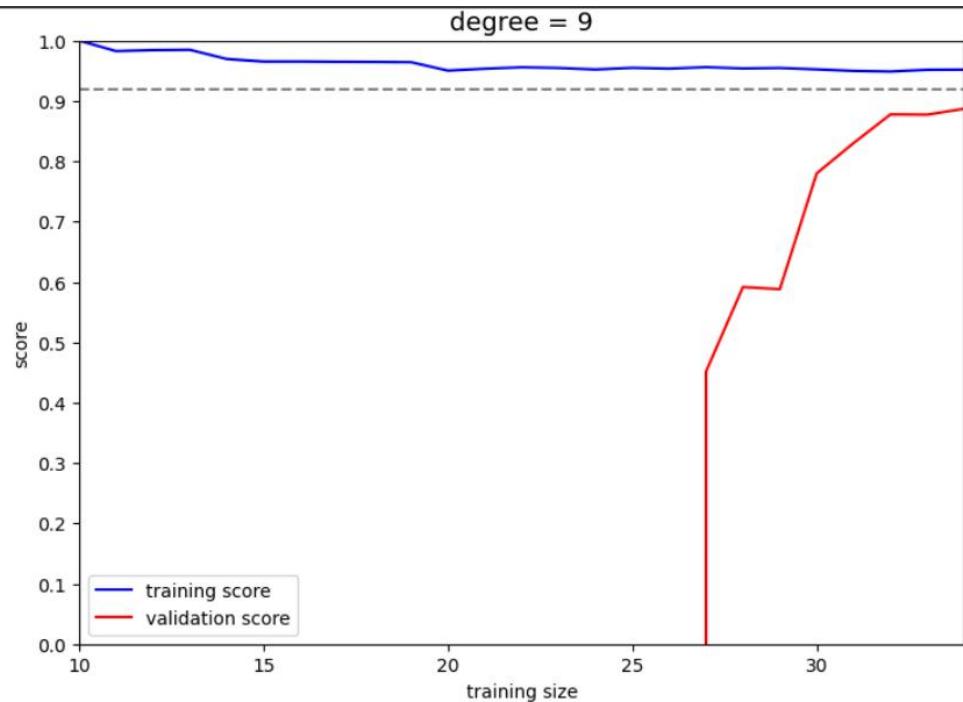
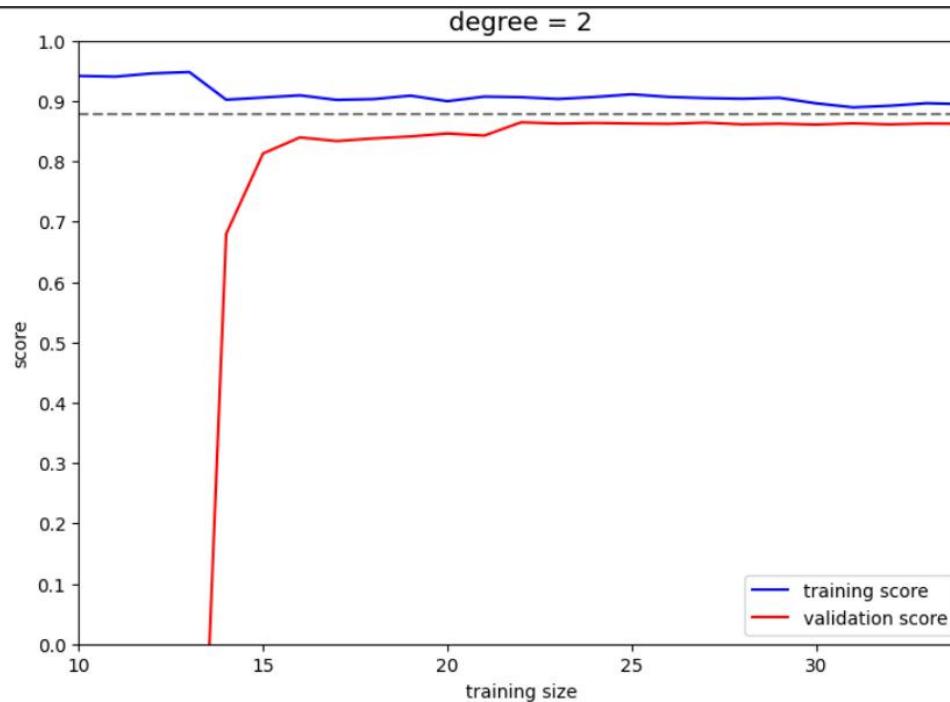
```

```

[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34] (25,)
(25, 7) (25, 7)
[array([0.94935834, 0.9537075 , 0.95317786, 0.96799601, 0.93964532,
       0.94828235, 0.95163327], array([0.92096749, 0.81695135, 0.90496397, 0.80149852, 0.97568253,
       0.91258214, 0.87726809])]
0.8871305834633204 0.9049639722085828

```





```
# Validation in Practice: Grid Search

# models generally have more than one knob to turn,
# and thus plots of validation and learning curves change from Lines to multi-dimensional surfaces
# In these cases we would rather simply find the particular model that maximizes the validation score

from sklearn.model_selection import GridSearchCV

param_grid = {'polynomialfeatures_degree': np.arange(21),
              'linearregression_fit_intercept': [True, False],
              'linearregression_positive': [True, False]}

grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)

grid.fit(X, y)

grid.best_params_
```

```
{'linearregression_fit_intercept': False,
 'linearregression_positive': False,
 'polynomialfeatures_degree': 4}

X_test = np.linspace(-0.1, 1.1, 500)[:, None]

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

model3 = PolynomialRegression(3, fit_intercept=True, positive=False)
ax[0].scatter(X.ravel(), y)

model1 = PolynomialRegression(1)
y_test = model1.fit(X, y).predict(X_test)
ax[0].plot(X_test.ravel(), y_test)

model4 = grid.best_estimator_

```

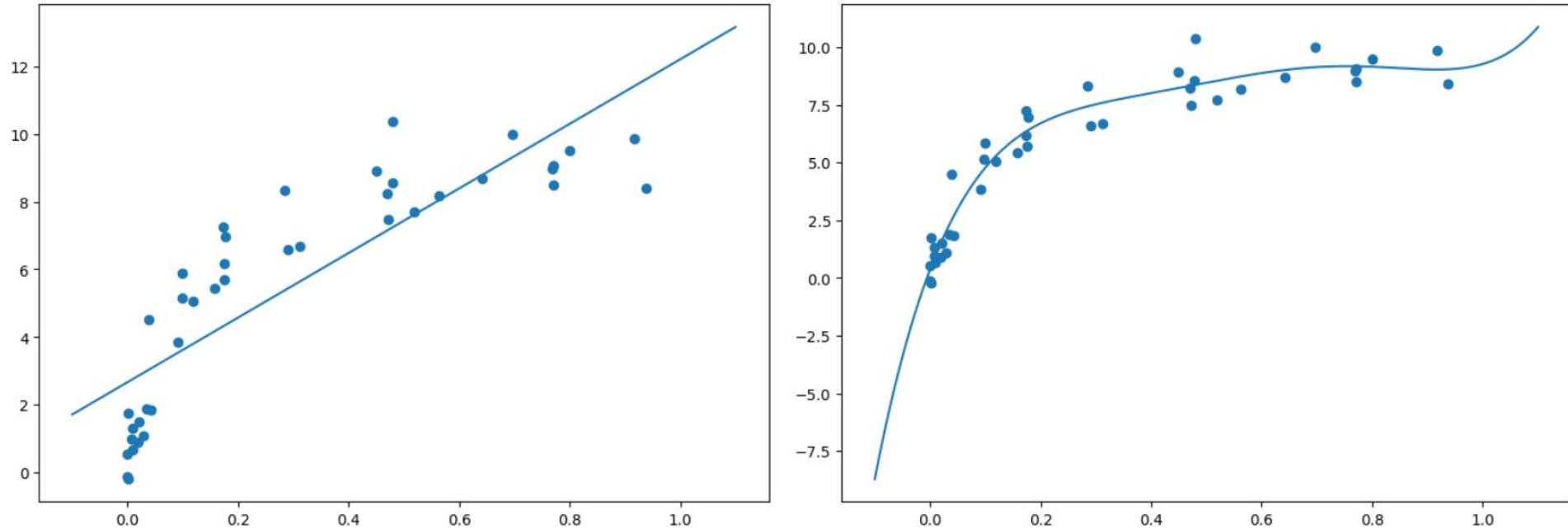
```

model1 = PolynomialRegression(1)
y_test = model1.fit(X, y).predict(X_test)
ax[0].plot(X_test.ravel(), y_test)

model4 = grid.best_estimator_
ax[1].scatter(X.ravel(), y)
y_test = model4.fit(X, y).predict(X_test)
ax[1].plot(X_test.ravel(), y_test)

plt.show()

```



```

# Own addition - Finding out why GridSearchCV says degree=4 is the best when we found it to be degree=3

for i in range(1,6):
    print(i, np.mean(cross_val_score(PolynomialRegression(i), X, y, cv=7)),
          np.median(cross_val_score(PolynomialRegression(i), X, y, cv=7)))

degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(), X, y,
                                           param_name='polynomialfeatures_degree', param_range=degree, cv=7)

plt.figure(figsize=(15,5))
ax = plt.axes()
ax.set_facecolor('#DEDEDE')

plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score - median (Graphical method)')
plt.plot(degree, np.mean(val_score, 1), color='orange', label='validation score - mean (GridSearchCV)')

plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score')
plt.xticks(np.linspace(0, 21, 22))
plt.grid()

plt.show()

```

```

    plt.show()
plt.xticks(np.linspace(0, 21, 22))
plt.grid()

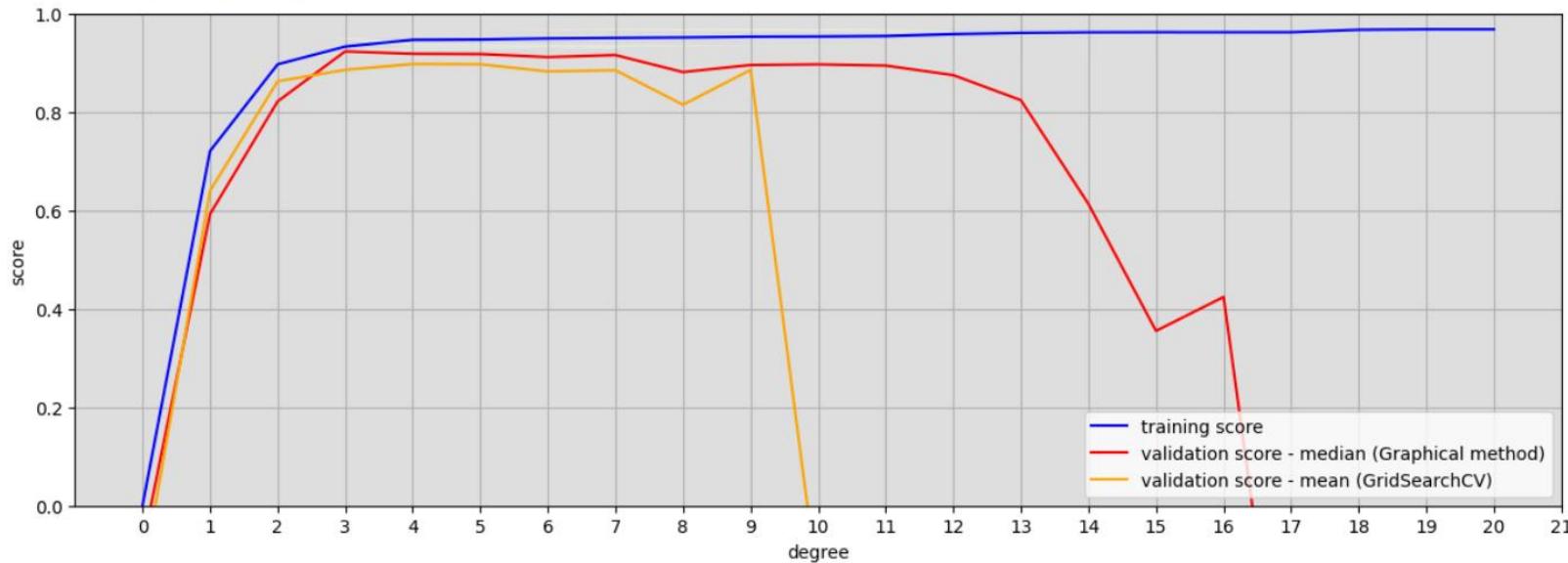
print(np.median(val_score, 1).max(), np.argmax(np.median(val_score, 1)))
print(np.mean(val_score, 1).max(), np.argmax(np.mean(val_score, 1)))

```

```

1 0.6397537735805129 0.5924167994880056
2 0.8621937570650806 0.8213562669413159
3 0.8854255761479877 0.922981319057214
4 0.8972710305736512 0.9188817885689393
5 0.8968050454664926 0.9175074208775591

```



```

0.922981319057214 3
0.8972710305736512 4

```

```

# ----- Some addition from another article -----

# While going through this article
# https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6
# Based on my prior knowledge I assumed that the following part in this article is a incorrect:
# "It is six times as many points as the original plot because I used cv=6."
# I wanted to check my assumption, below is the code where I found out that it's right and this part of the article is wrong

```

```

from sklearn import datasets, linear_model
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn import metrics

# Load the Diabetes dataset
columns = 'age sex bmi map tc ldl hdl tch ltg glu'.split() # Declare the columns names
diabetes = datasets.load_diabetes() # Call the diabetes dataset from sklearn
df = pd.DataFrame(diabetes.data, columns=columns) # Load the dataset as a pandas data frame
y = diabetes.target # define the target variable (dependent variable) as y
print(df.shape)
print(y.shape)

X_train, X_test, y_train, y_test = train_test_split(df, y, test_size=0.2)
print(X_test.shape)

# fit a model
lm = linear_model.LinearRegression()
model = lm.fit(X_train, y_train)

```

```
print(X_test.shape)

# fit a model
lm = linear_model.LinearRegression()
model = lm.fit(X_train, y_train)
predictions = lm.predict(X_test)
print(predictions.shape)

# Make cross validated predictions
predictions = cross_val_predict(model, df, y, cv=10)
predictions.shape

(442, 10)
(442,)
(89, 10)
(89,)
(442,)
```

Data Visualization

Load dataset to visualize

```
# Tips dataset
# More info here: https://www.kaggle.com/code/sanjanabasu/tips-dataset/comment
```

```
data = [[16.99,1.01,"Female","No","Sun","Dinner",2],
[10.34,1.66,"Male","No","Sun","Dinner",3],
[21.01,3.5,"Male","No","Sun","Dinner",3],
[23.68,3.31,"Male","No","Sun","Dinner",2],
[24.59,3.61,"Female","No","Sun","Dinner",4],
[25.29,4.71,"Male","No","Sun","Dinner",4],
[8.77,2,"Male","No","Sun","Dinner",2],
[26.88,3.12,"Male","No","Sun","Dinner",4],
[15.04,1.96,"Male","No","Sun","Dinner",2],
[14.78,3.23,"Male","No","Sun","Dinner",2],
[10.27,1.71,"Male","No","Sun","Dinner",2],
[35.26,5,"Female","No","Sun","Dinner",4],
[15.42,1.57,"Male","No","Sun","Dinner",2],
[18.43,3,"Male","No","Sun","Dinner",4],
[14.83,3.02,"Female","No","Sun","Dinner",2],
[21.58,3.92,"Male","No","Sun","Dinner",2],
[10.33,1.67,"Female","No","Sun","Dinner",3],
[16.29,3.71,"Male","No","Sun","Dinner",3],
[16.97,3.5,"Female","No","Sun","Dinner",3],
[20.65,3.35,"Male","No","Sat","Dinner",3],
[17.92,4.08,"Male","No","Sat","Dinner",2],
[20.29,2.75,"Female","No","Sat","Dinner",2],
[15.77,2.23,"Female","No","Sat","Dinner",2],
[39.42,7.58,"Male","No","Sat","Dinner",4],
[19.82,3.18,"Male","No","Sat","Dinner",2],
[17.81,2.34,"Male","No","Sat","Dinner",4],
[13.37,2,"Male","No","Sat","Dinner",2],
[12.69,2,"Male","No","Sat","Dinner",2],
[21.7,4.3,"Male","No","Sat","Dinner",2],
[19.65,3,"Female","No","Sat","Dinner",2],
[9.55,1.45,"Male","No","Sat","Dinner",2],
[18.35,2.5,"Male","No","Sat","Dinner",4],
[15.06,3,"Female","No","Sat","Dinner",2],
[20.69,2.45,"Female","No","Sat","Dinner",4],
[17.78,3.27,"Male","No","Sat","Dinner",2],
[24.06,3.6,"Male","No","Sat","Dinner",3],
[16.31,2,"Male","No","Sat","Dinner",3],
[16.93,3.07,"Female","No","Sat","Dinner",3],
[18.69,2.31,"Male","No","Sat","Dinner",3],
[31.27,5,"Male","No","Sat","Dinner",3],
[16.04,2.24,"Male","No","Sat","Dinner",3],
[17.46,2.54,"Male","No","Sun","Dinner",2],
```

[16.04,2.24,"Male","No","Sat","Dinner",3],
[17.46,2.54,"Male","No","Sun","Dinner",2],
[13.94,3.06,"Male","No","Sun","Dinner",2],
[9.68,1.32,"Male","No","Sun","Dinner",2],
[30.4,5.6,"Male","No","Sun","Dinner",4],
[18.29,3,"Male","No","Sun","Dinner",2],
[22.23,5,"Male","No","Sun","Dinner",2],
[32.4,6,"Male","No","Sun","Dinner",4],
[28.55,2.05,"Male","No","Sun","Dinner",3],
[18.04,3,"Male","No","Sun","Dinner",2],
[12.54,2.5,"Male","No","Sun","Dinner",2],
[10.29,2.6,"Female","No","Sun","Dinner",2],
[34.81,5.2,"Female","No","Sun","Dinner",4],
[9.94,1.56,"Male","No","Sun","Dinner",2],
[25.56,4.34,"Male","No","Sun","Dinner",4],
[19.49,3.51,"Male","No","Sun","Dinner",2],
[38.01,3,"Male","Yes","Sat","Dinner",4],
[26.41,1.5,"Female","No","Sat","Dinner",2],
[11.24,1.76,"Male","Yes","Sat","Dinner",2],
[48.27,6.73,"Male","No","Sat","Dinner",4],
[20.29,3.21,"Male","Yes","Sat","Dinner",2],
[13.81,2,"Male","Yes","Sat","Dinner",2],
[11.02,1.98,"Male","Yes","Sat","Dinner",2],
[18.29,3.76,"Male","Yes","Sat","Dinner",4],
[17.59,2.64,"Male","No","Sat","Dinner",3],
[20.08,3.15,"Male","No","Sat","Dinner",3],
[16.45,2.47,"Female","No","Sat","Dinner",2],
[3.07,1,"Female","Yes","Sat","Dinner",1],
[20.23,2.01,"Male","No","Sat","Dinner",2],
[15.01,2.09,"Male","Yes","Sat","Dinner",2],
[12.02,1.97,"Male","No","Sat","Dinner",2],
[17.07,3,"Female","No","Sat","Dinner",3],
[26.86,3.14,"Female","Yes","Sat","Dinner",2],
[25.28,5,"Female","Yes","Sat","Dinner",2],
[14.73,2.2,"Female","No","Sat","Dinner",2],
[10.51,1.25,"Male","No","Sat","Dinner",2],
[17.92,3.08,"Male","Yes","Sat","Dinner",2],
[27.2,4,"Male","No","Thur","Lunch",4],
[22.76,3,"Male","No","Thur","Lunch",2],
[17.29,2.71,"Male","No","Thur","Lunch",2],
[19.44,3,"Male","Yes","Thur","Lunch",2],
[16.66,3.4,"Male","No","Thur","Lunch",2],
[10.07,1.83,"Female","No","Thur","Lunch",1],
[32.68,5,"Male","Yes","Thur","Lunch",2],
[15.98,2.03,"Male","No","Thur","Lunch",2],
[34.83,5.17,"Female","No","Thur","Lunch",4],
[13.03,2,"Male","No","Thur","Lunch",2],
[18.28,4,"Male","No","Thur","Lunch",2],
[24.71,5.85,"Male","No","Thur","Lunch",2],
[21.16,3,"Male","No","Thur","Lunch",2],
[28.97,3,"Male","Yes","Fri","Dinner",2],
[22.49,3.5,"Male","No","Fri","Dinner",2],
[5.75,1,"Female","Yes","Fri","Dinner",2],
[16.32,4.3,"Female","Yes","Fri","Dinner",2],
[22.75,3.25,"Female","No","Fri","Dinner",2],
[40.17,4.73,"Male","Yes","Fri","Dinner",4],
[27.28,4,"Male","Yes","Fri","Dinner",2],
[12.03,1.5,"Male","Yes","Fri","Dinner",2],
[21.01,3,"Male","Yes","Fri","Dinner",2],
[12.46,1.5,"Male","No","Fri","Dinner",2],
[11.35,2.5,"Female","Yes","Fri","Dinner",2],
[15.38,3,"Female","Yes","Fri","Dinner",2],
[44.3,2.5,"Female","Yes","Sat","Dinner",3],
[22.42,3.48,"Female","Yes","Sat","Dinner",2],
[20.92,4.08,"Female","No","Sat","Dinner",2],
[15.36,1.64,"Male","Yes","Sat","Dinner",2],
[20.49,4.06,"Male","Yes","Sat","Dinner",2],
[25.21,4.29,"Male","Yes","Sat","Dinner",2],
[18.24,3.76,"Male","No","Sat","Dinner",2],
[14.31,4,"Female","Yes","Sat","Dinner",2],
[14.3,"Male","No","Sat","Dinner",2].

[25.21,4.29,"Male","Yes","Sat","Dinner",2],
[18.24,3.76,"Male","No","Sat","Dinner",2],
[14.31,4,"Female","Yes","Sat","Dinner",2],
[14,3,"Male","No","Sat","Dinner",2],
[7.25,1,"Female","No","Sat","Dinner",1],
[38.07,4,"Male","No","Sun","Dinner",3],
[23.95,2.55,"Male","No","Sun","Dinner",2],
[25.71,4,"Female","No","Sun","Dinner",3],
[17.31,3.5,"Female","No","Sun","Dinner",2],
[29.93,5.07,"Male","No","Sun","Dinner",4],
[10.65,1.5,"Female","No","Thur","Lunch",2],
[12.43,1.8,"Female","No","Thur","Lunch",2],
[24.08,2.92,"Female","No","Thur","Lunch",4],
[11.69,2.31,"Male","No","Thur","Lunch",2],
[13.42,1.68,"Female","No","Thur","Lunch",2],
[14.26,2.5,"Male","No","Thur","Lunch",2],
[15.95,2,"Male","No","Thur","Lunch",2],
[12.48,2.52,"Female","No","Thur","Lunch",2],
[29.8,4.2,"Female","No","Thur","Lunch",6],
[8.52,1.48,"Male","No","Thur","Lunch",2],
[14.52,2,"Female","No","Thur","Lunch",2],
[11.38,2,"Female","No","Thur","Lunch",2],
[22.82,2.18,"Male","No","Thur","Lunch",3],
[19.08,1.5,"Male","No","Thur","Lunch",2],
[20.27,2.83,"Female","No","Thur","Lunch",2],
[11.17,1.5,"Female","No","Thur","Lunch",2],
[12.26,2,"Female","No","Thur","Lunch",2],
[18.26,3.25,"Female","No","Thur","Lunch",2],
[8.51,1.25,"Female","No","Thur","Lunch",2],
[10.33,2,"Female","No","Thur","Lunch",2],
[14.15,2,"Female","No","Thur","Lunch",2],
[16,2,"Male","Yes","Thur","Lunch",2],
[13.16,2.75,"Female","No","Thur","Lunch",2],
[17.47,3.5,"Female","No","Thur","Lunch",2],
[34.3,6.7,"Male","No","Thur","Lunch",6],
[41.19,5,"Male","No","Thur","Lunch",5],
[27.05,5,"Female","No","Thur","Lunch",6],
[16.43,2.3,"Female","No","Thur","Lunch",2],
[8.35,1.5,"Female","No","Thur","Lunch",2],
[18.64,1.36,"Female","No","Thur","Lunch",3],
[11.87,1.63,"Female","No","Thur","Lunch",2],
[9.78,1.73,"Male","No","Thur","Lunch",2],
[7.51,2,"Male","No","Thur","Lunch",2],
[14.07,2.5,"Male","No","Sun","Dinner",2],
[13.13,2,"Male","No","Sun","Dinner",2],
[17.26,2.74,"Male","No","Sun","Dinner",3],
[24.55,2,"Male","No","Sun","Dinner",4],
[19.77,2,"Male","No","Sun","Dinner",4],
[29.85,5.14,"Female","No","Sun","Dinner",5],
[48.17,5,"Male","No","Sun","Dinner",6],
[25,3.75,"Female","No","Sun","Dinner",4],
[13.39,2.61,"Female","No","Sun","Dinner",2],
[16.49,2,"Male","No","Sun","Dinner",4],
[21.5,3.5,"Male","No","Sun","Dinner",4],
[12.66,2.5,"Male","No","Sun","Dinner",2],
[16.21,2,"Female","No","Sun","Dinner",3],
[13.81,2,"Male","No","Sun","Dinner",2],
[17.51,3,"Female","Yes","Sun","Dinner",2],
[24.52,3.48,"Male","No","Sun","Dinner",3],
[20.76,2.24,"Male","No","Sun","Dinner",2],
[31.71,4.5,"Male","No","Sun","Dinner",4],
[10.59,1.61,"Female","Yes","Sat","Dinner",2],
[10.63,2,"Female","Yes","Sat","Dinner",2],
[50.81,10,"Male","Yes","Sat","Dinner",3],
[15.81,3.16,"Male","Yes","Sat","Dinner",2],
[7.25,5.15,"Male","Yes","Sun","Dinner",2],
[31.85,3.18,"Male","Yes","Sun","Dinner",2],
[16.82,4,"Male","Yes","Sun","Dinner",2],
[32.9,3.11,"Male","Yes","Sun","Dinner",2],
[17.89,2,"Male","Yes","Sun","Dinner",2],
[8.42,2,"Male","Yes","Sun","Dinner",2]

[32.9,3.11,"Male","Yes","Sun","Dinner",2],
[17.89,2,"Male","Yes","Sun","Dinner",2],
[14.48,2,"Male","Yes","Sun","Dinner",2],
[9.6,4,"Female","Yes","Sun","Dinner",2],
[34.63,3.55,"Male","Yes","Sun","Dinner",2],
[34.65,3.68,"Male","Yes","Sun","Dinner",4],
[23.33,5.65,"Male","Yes","Sun","Dinner",2],
[45.35,3.5,"Male","Yes","Sun","Dinner",3],
[23.17,6.5,"Male","Yes","Sun","Dinner",4],
[40.55,3,"Male","Yes","Sun","Dinner",2],
[20.69,5,"Male","No","Sun","Dinner",5],
[20.9,3.5,"Female","Yes","Sun","Dinner",3],
[30.46,2,"Male","Yes","Sun","Dinner",5],
[18.15,3.5,"Female","Yes","Sun","Dinner",3],
[23.1,4,"Male","Yes","Sun","Dinner",3],
[15.69,1.5,"Male","Yes","Sun","Dinner",2],
[19.81,4.19,"Female","Yes","Thur","Lunch",2],
[28.44,2.56,"Male","Yes","Thur","Lunch",2],
[15.48,2.02,"Male","Yes","Thur","Lunch",2],
[16.58,4,"Male","Yes","Thur","Lunch",2],
[7.56,1.44,"Male","No","Thur","Lunch",2],
[10.34,2,"Male","Yes","Thur","Lunch",2],
[43.11,5,"Female","Yes","Thur","Lunch",4],
[13,2,"Female","Yes","Thur","Lunch",2],
[13.51,2,"Male","Yes","Thur","Lunch",2],
[18.71,4,"Male","Yes","Thur","Lunch",3],
[12.74,2.01,"Female","Yes","Thur","Lunch",2],
[13,2,"Female","Yes","Thur","Lunch",2],
[16.4,2.5,"Female","Yes","Thur","Lunch",2],
[20.53,4,"Male","Yes","Thur","Lunch",4],
[16.47,3.23,"Female","Yes","Thur","Lunch",3],
[26.59,3.41,"Male","Yes","Sat","Dinner",3],
[38.73,3,"Male","Yes","Sat","Dinner",4],
[24.27,2.03,"Male","Yes","Sat","Dinner",2],
[12.76,2.23,"Female","Yes","Sat","Dinner",2],
[30.06,2,"Male","Yes","Sat","Dinner",3],
[25.89,5.16,"Male","Yes","Sat","Dinner",4],
[48.33,9,"Male","No","Sat","Dinner",4],
[13.27,2.5,"Female","Yes","Sat","Dinner",2],
[28.17,6.5,"Female","Yes","Sat","Dinner",3],
[12.9,1.1,"Female","Yes","Sat","Dinner",2],
[28.15,3,"Male","Yes","Sat","Dinner",5],
[11.59,1.5,"Male","Yes","Sat","Dinner",2],
[7.74,1.44,"Male","Yes","Sat","Dinner",2],
[30.14,3.09,"Female","Yes","Sat","Dinner",4],
[12.16,2.2,"Male","Yes","Fri","Lunch",2],
[13.42,3.48,"Female","Yes","Fri","Lunch",2],
[8.58,1.92,"Male","Yes","Fri","Lunch",1],
[15.98,3,"Female","No","Fri","Lunch",3],
[13.42,1.58,"Male","Yes","Fri","Lunch",2],
[16.27,2.5,"Female","Yes","Fri","Lunch",2],
[10.09,2,"Female","Yes","Fri","Lunch",2],
[20.45,3,"Male","No","Sat","Dinner",4],
[13.28,2.72,"Male","No","Sat","Dinner",2],
[22.12,2.88,"Female","Yes","Sat","Dinner",2],
[24.01,2,"Male","Yes","Sat","Dinner",4],
[15.69,3,"Male","Yes","Sat","Dinner",3],
[11.61,3.39,"Male","No","Sat","Dinner",2],
[10.77,1.47,"Male","No","Sat","Dinner",2],
[15.53,3,"Male","Yes","Sat","Dinner",2],
[10.07,1.25,"Male","No","Sat","Dinner",2],
[12.6,1,"Male","Yes","Sat","Dinner",2],
[32.83,1.17,"Male","Yes","Sat","Dinner",2],
[35.83,4.67,"Female","No","Sat","Dinner",3],
[29.03,5.92,"Male","No","Sat","Dinner",3],
[27.18,2,"Female","Yes","Sat","Dinner",2],
[22.67,2,"Male","Yes","Sat","Dinner",2],
[17.82,1.75,"Male","No","Sat","Dinner",2],
[18.78,3,"Female","No","Thur","Dinner",2]]

```

[16.58,4,"Male","Yes","Thur","Lunch",2],
[7.56,1.44,"Male","No","Thur","Lunch",2],
[10.34,2,"Male","Yes","Thur","Lunch",2],
[43.11,5,"Female","Yes","Thur","Lunch",4],
[13,2,"Female","Yes","Thur","Lunch",2],
[13.51,2,"Male","Yes","Thur","Lunch",2],
[18.71,4,"Male","Yes","Thur","Lunch",3],
[12.74,2.01,"Female","Yes","Thur","Lunch",2],
[13,2,"Female","Yes","Thur","Lunch",2],
[16.4,2.5,"Female","Yes","Thur","Lunch",2],
[20.53,4,"Male","Yes","Thur","Lunch",4],
[16.47,3.23,"Female","Yes","Thur","Lunch",3],
[26.59,3.41,"Male","Yes","Sat","Dinner",3],
[38.73,3,"Male","Yes","Sat","Dinner",4],
[24.27,2.03,"Male","Yes","Sat","Dinner",2],
[12.76,2.23,"Female","Yes","Sat","Dinner",2],
[30.06,2,"Male","Yes","Sat","Dinner",3],
[25.89,5.16,"Male","Yes","Sat","Dinner",4],
[48.33,9,"Male","No","Sat","Dinner",4],
[13.27,2.5,"Female","Yes","Sat","Dinner",2],
[28.17,6.5,"Female","Yes","Sat","Dinner",3],
[12.9,1.1,"Female","Yes","Sat","Dinner",2],
[28.15,3,"Male","Yes","Sat","Dinner",5],
[11.59,1.5,"Male","Yes","Sat","Dinner",2],
[7.74,1.44,"Male","Yes","Sat","Dinner",2],
[30.14,3.09,"Female","Yes","Sat","Dinner",4],
[12.16,2.2,"Male","Yes","Fri","Lunch",2],
[13.42,3.48,"Female","Yes","Fri","Lunch",2],
[8.58,1.92,"Male","Yes","Fri","Lunch",1],
[15.98,3,"Female","No","Fri","Lunch",3],
[13.42,1.58,"Male","Yes","Fri","Lunch",2],
[16.27,2.5,"Female","Yes","Fri","Lunch",2],
[10.09,2,"Female","Yes","Fri","Lunch",2],
[20.45,3,"Male","No","Sat","Dinner",4],
[13.28,2.72,"Male","No","Sat","Dinner",2],
[22.12,2.88,"Female","Yes","Sat","Dinner",2],
[24.01,2,"Male","Yes","Sat","Dinner",4],
[15.69,3,"Male","Yes","Sat","Dinner",3],
[11.61,3.39,"Male","No","Sat","Dinner",2],
[10.77,1.47,"Male","No","Sat","Dinner",2],
[15.53,3,"Male","Yes","Sat","Dinner",2],
[10.07,1.25,"Male","No","Sat","Dinner",2],
[12.6,1,"Male","Yes","Sat","Dinner",2],
[32.83,1.17,"Male","Yes","Sat","Dinner",2],
[35.83,4.67,"Female","No","Sat","Dinner",3],
[29.03,5.92,"Male","No","Sat","Dinner",3],
[27.18,2,"Female","Yes","Sat","Dinner",2],
[22.67,2,"Male","Yes","Sat","Dinner",2],
[17.82,1.75,"Male","No","Sat","Dinner",2],
[18.78,3,"Female","No","Thur","Dinner",2]]
```

df = pd.DataFrame(data, columns=["total_bill", "tip", "sex", "smoker", "day", "time", "size"])

display(df.describe())

display(df)

	total_bill	tip	size
count	244.0000000000000000	244.0000000000000000	244.0000000000000000
mean	19.785942622950820180	2.998278688524589786	2.569672131147541005
std	8.902411954856855658	1.383638189001182184	0.951099804732234433
min	3.06999999999999840	1.0000000000000000	1.0000000000000000
25%	13.34750000000000142	2.0000000000000000	2.0000000000000000
50%	17.795000000000001705	2.89999999999999911	2.0000000000000000
75%	24.1274999999999726	3.5625000000000000	3.0000000000000000
99.5%	60.0000000000000000	6.0000000000000000	6.0000000000000000

	total_bill	tip	size
count	244.00000000000000000000	244.00000000000000000000	244.00000000000000000000
mean	19.785942622950820180	2.998278688524589786	2.569672131147541005
std	8.902411954856855658	1.383638189001182184	0.951099804732234433
min	3.06999999999999840	1.0000000000000000	1.0000000000000000
25%	13.3475000000000142	2.0000000000000000	2.0000000000000000
50%	17.79500000000001705	2.8999999999999911	2.0000000000000000
75%	24.1274999999999726	3.5625000000000000	3.0000000000000000
max	50.81000000000002274	10.0000000000000000	6.0000000000000000

	total_bill	tip	sex	smoker	day	time	size
0	16.9899999999998437	1.0100000000000009	Female	No	Sun	Dinner	2
1	10.339999999999858	1.6599999999999920	Male	No	Sun	Dinner	3
2	21.01000000000001563	3.5000000000000000	Male	No	Sun	Dinner	3
3	23.6799999999999716	3.3100000000000053	Male	No	Sun	Dinner	2
4	24.589999999999858	3.6099999999999876	Female	No	Sun	Dinner	4
...
239	29.03000000000001137	5.9199999999999929	Male	No	Sat	Dinner	3
240	27.1799999999999716	2.0000000000000000	Female	Yes	Sat	Dinner	2
241	22.67000000000001705	2.0000000000000000	Male	Yes	Sat	Dinner	2
242	17.8200000000000284	1.7500000000000000	Male	No	Sat	Dinner	2
243	18.78000000000001137	3.0000000000000000	Female	No	Thur	Dinner	2

244 rows × 7 columns

Matplotlib

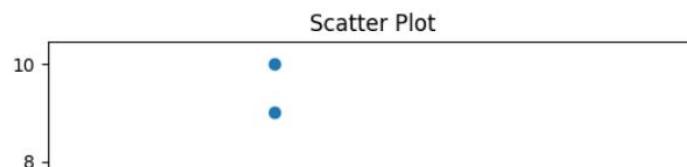
```
# Most commonly used plots for Matplotlib: scatter plot, line plot, histogram, etc.
```

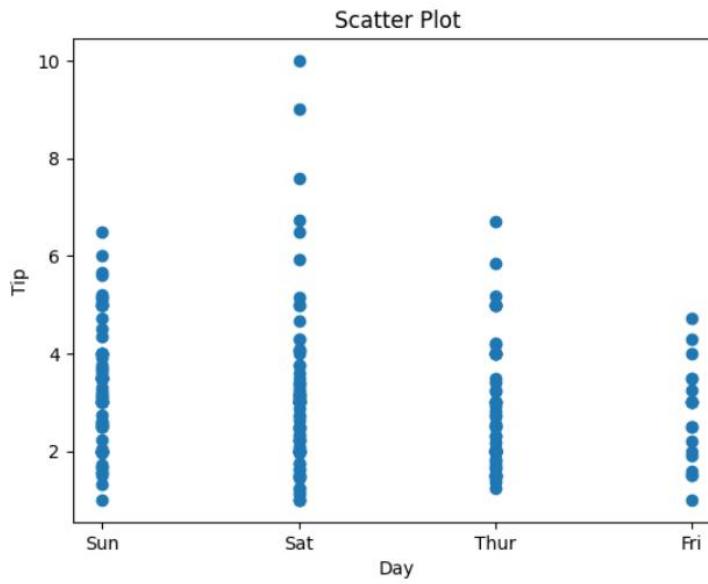
```
# Scatter Plot
# Day against tip
plt.scatter(df['day'], df['tip'])

# Adding Title to the Plot
plt.title("Scatter Plot")

# Setting the X and Y Labels
plt.xlabel('Day')
plt.ylabel('Tip')

plt.show()
```



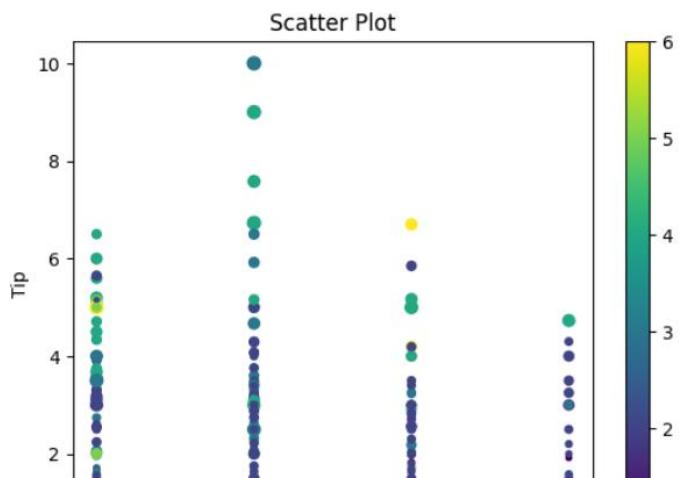


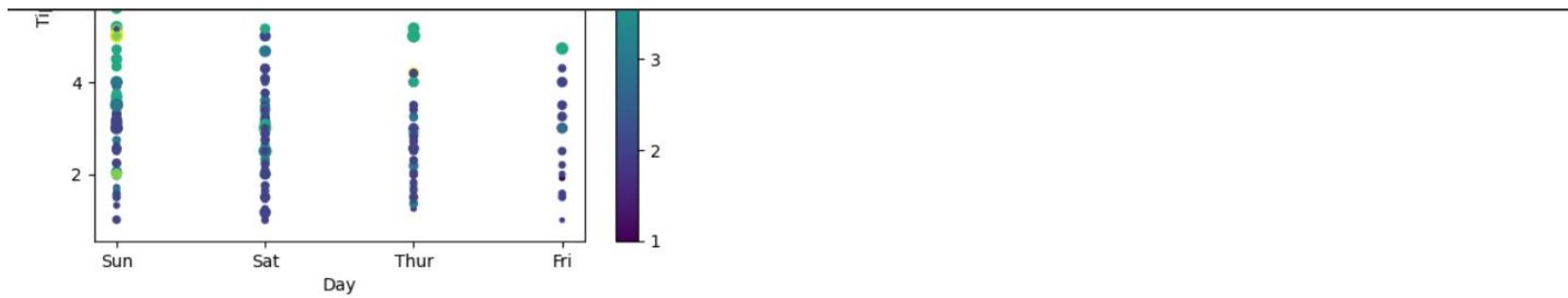
```
# Scatter plot with some settings
# D day against tip
plt.scatter(df['day'], df['tip'], c=df['size'],
           s=df['total_bill'])

# Adding Title to the Plot
plt.title("Scatter Plot")

# Setting the X and Y Labels
plt.xlabel('Day')
plt.ylabel('Tip')

plt.colorbar()
plt.show()
```





```
# Line Chart

plt.plot(df['tip'], label = 'Tip')
plt.plot(df['size'], label = 'Size')

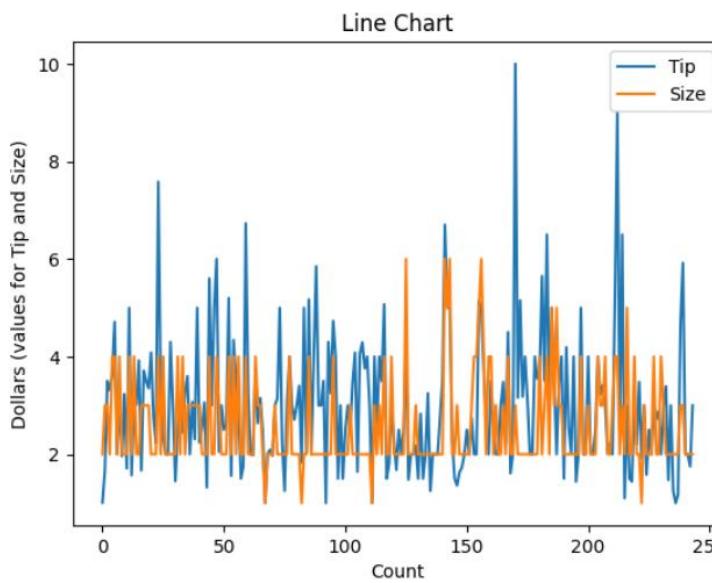
# Adding Title to the Plot
plt.title("Line Chart")

# Setting the X and Y Labels
plt.xlabel('Count')
plt.ylabel('Dollars (values for Tip and Size)')

plt.legend()

plt.show()

df[['tip','size']].describe()
```



	tip	size
count	244.00000000000000	244.00000000000000
mean	2.998278688524589786	2.569672131147541005
std	1.383638189001182184	0.951099804732234433

	Count	
	tip	size
count	244.00000000000000000000	244.00000000000000000000
mean	2.998278688524589786	2.569672131147541005
std	1.383638189001182184	0.951099804732234433
min	1.0000000000000000	1.0000000000000000
25%	2.0000000000000000	2.0000000000000000
50%	2.8999999999999911	2.0000000000000000
75%	3.5625000000000000	3.0000000000000000
max	10.0000000000000000	6.0000000000000000

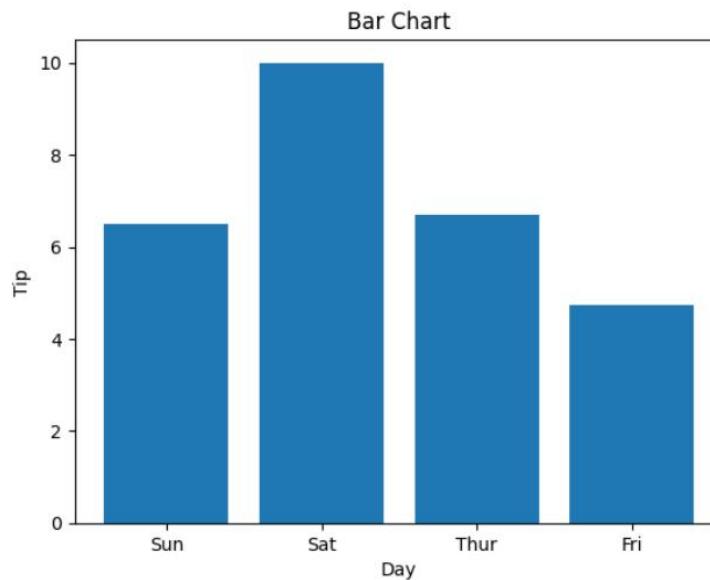
```
# Bar chart

# Bar chart with day against tip
plt.bar(df['day'], df['tip'])

plt.title("Bar Chart")

# Setting the X and Y Labels
plt.xlabel('Day')
plt.ylabel('Tip')

# Adding the Legends
plt.show()
```



```
# Histogram

plt.figure(figsize=(15,6))

# histogram of total_bill
plt.hist(df['total_bill'])
```

```

# Histogram

plt.figure(figsize=(15,6))

# histogram of total_bills
plt.hist(df['total_bill'])

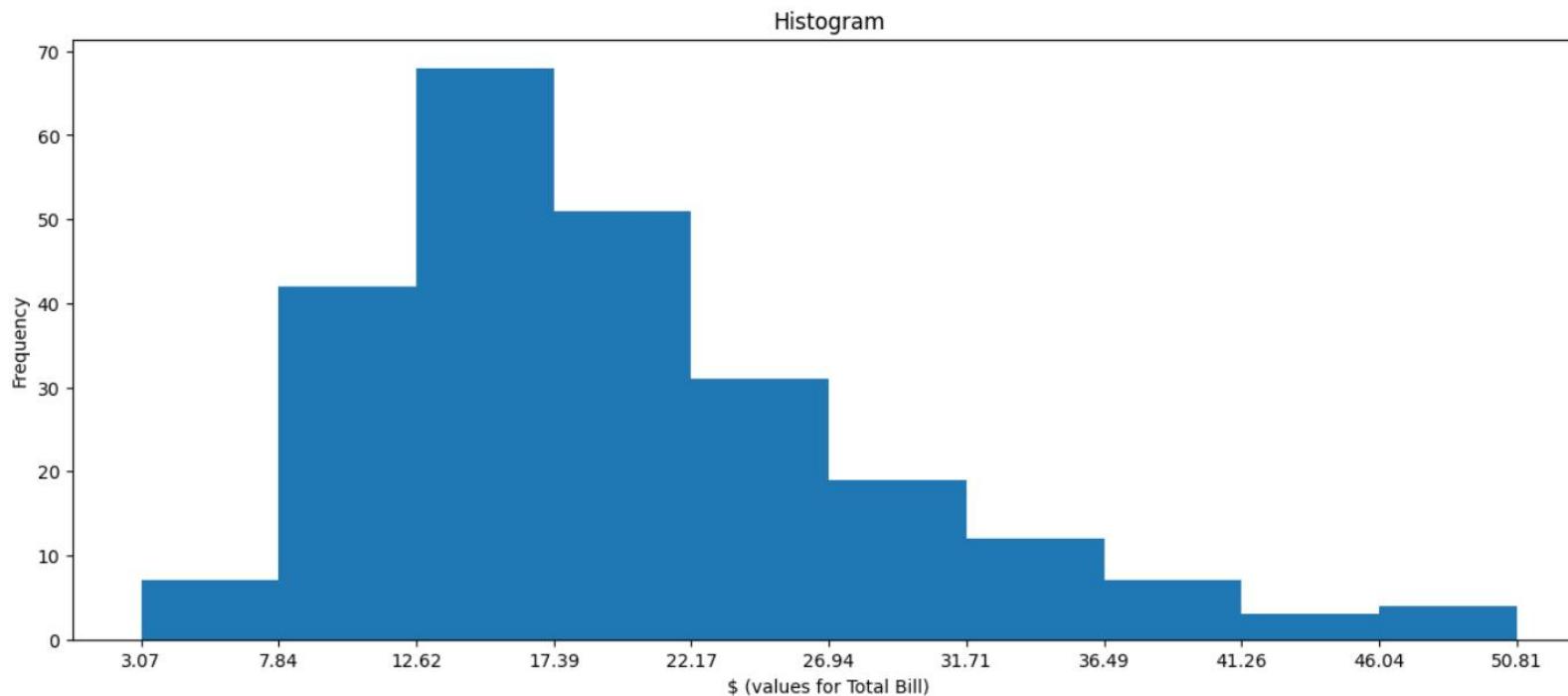
plt.title("Histogram")

# Setting the X and Y labels
plt.xlabel('$ (values for Total Bill)')
plt.xticks(np.linspace(df['total_bill'].min(), df['total_bill'].max(), 11))
plt.ylabel('Frequency')

plt.show()

df['total_bill'].describe()

```



```

count    244.00000000000000
mean     19.785942622950820180
std      8.902411954856855658
min      3.06999999999999840
25%     13.34750000000000142
50%     17.795000000000001705
75%     24.12749999999997726
max      50.8100000000002274
Name: total_bill, dtype: float64

```

Seaborn

Seaborn is a high level interface

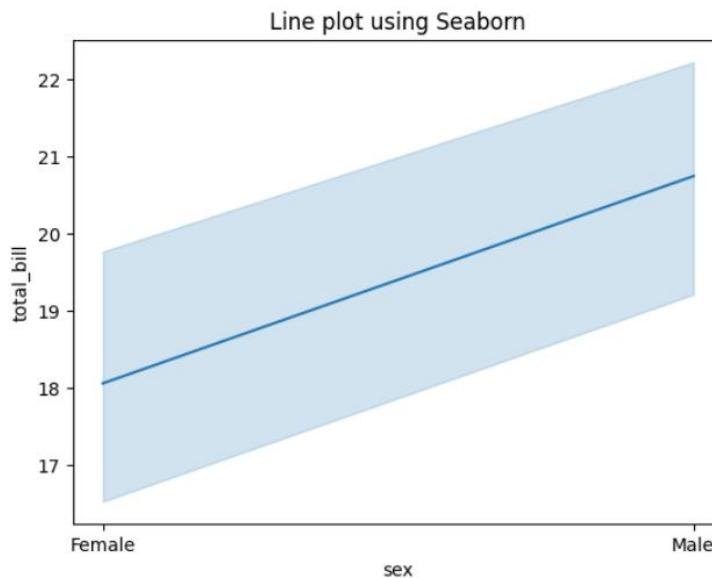
```
Name: total_bill, dtype: float64
```

Seaborn

```
# Seaborn is a high-level interface  
# It is built on top of Matplotlib, therefore it can be used with Matplotlib as well
```

```
# Using Seaborn and Matplotlib together  
  
# Draw Lineplot using Seaborn  
sns.lineplot(x="sex", y="total_bill", data=df)  
  
# Setting the title using Matplotlib  
plt.title('Line plot using Seaborn')  
  
plt.show()
```

```
df["total_bill"].groupby(df['sex']).mean()
```



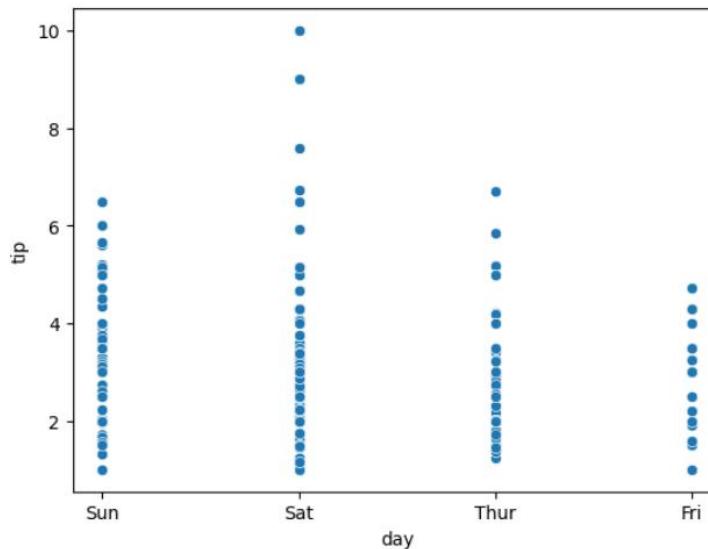
```
sex  
Female    18.056896551724140210  
Male     20.744076433121019676  
Name: total_bill, dtype: float64
```

```
# Scatter plot  
  
sns.scatterplot(x='day', y='tip', data=df)  
plt.show()
```

```
df["tip"].groupby(df['day']).describe()
```

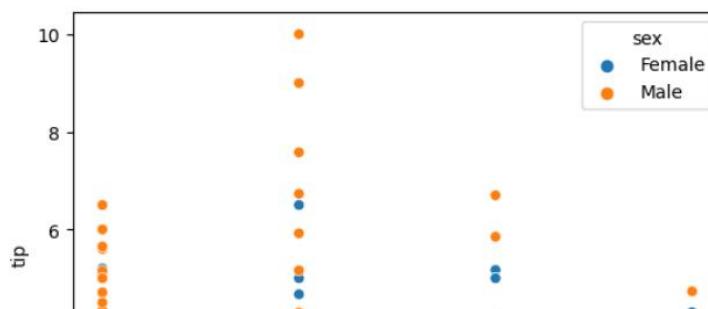
```
# Scatter plot  
sns.scatterplot(x='day', y='tip', data=df)  
plt.show()
```

```
df[“tip”].groupby(df[‘day’]).describe()
```



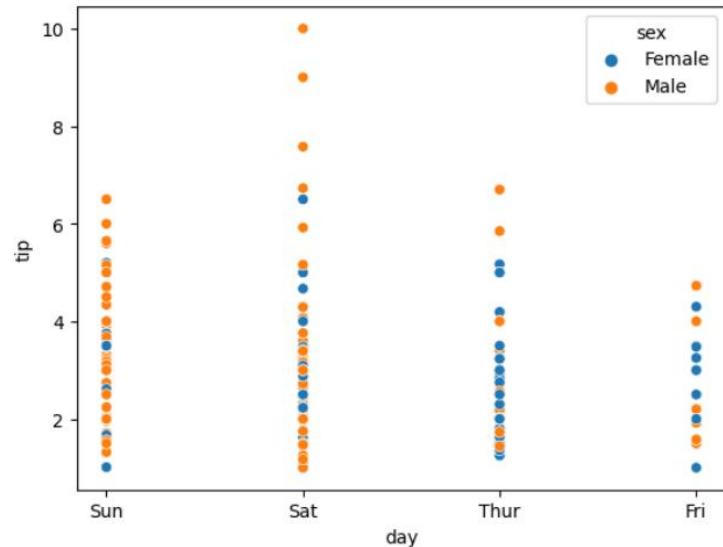
day	count	mean	std	min	25%	50%	75%	max
Fri	19.0	2.734736842105262689	1.019577082373169352	1.0000000000000000	1.9599999999999964	3.0000000000000000	3.3650000000000213	4.7300000000000426
Sat	87.0	2.993103448275861833	1.631014315840404683	1.0000000000000000	2.0000000000000000	2.7500000000000000	3.3700000000000107	10.0000000000000000
Sun	76.0	3.255131578947368709	1.234880283990911209	1.0100000000000009	2.03749999999999645	3.1500000000000355	4.0000000000000000	6.5000000000000000
Thur	62.0	2.771451612903224948	1.240223204097134602	1.2500000000000000	2.0000000000000000	2.30499999999999716	3.36249999999999822	6.7000000000000178

```
# Scatter plot with coloring based on the ‘sex’ feature  
sns.scatterplot(x='day', y='tip', data=df,  
hue='sex')  
plt.show()
```



```
# Scatter plot with coloring based on the 'sex' feature
```

```
sns.scatterplot(x='day', y='tip', data=df,
                 hue='sex')
plt.show()
```



```
# Own addition: Trying to recreate the same scatter plot with Matplotlib
```

```
fig, ax = plt.subplots(figsize=(16, 6))

color = pd.get_dummies(df['sex'], drop_first=True).squeeze()

scatter = plt.scatter(df['day'], df['tip'], c=color)

legend = ax.legend(*scatter.legend_elements(),
                   loc="best", title="Sex")

# color change - not working
legend.legendHandles[0].set_color('blue')
legend.legendHandles[1].set_color('red')

ax.add_artist(legend)

plt.xlabel('Day')
plt.ylabel('Tip')

plt.show()
```



```

# Own addition: Trying to recreate the same scatter plot with Matplotlib

fig, ax = plt.subplots(figsize=(16, 6))

color = pd.get_dummies(df['sex'], drop_first=True).squeeze()

scatter = plt.scatter(df['day'], df['tip'], c=color)

legend = ax.legend(*scatter.legend_elements(),
                   loc="best", title="Sex")

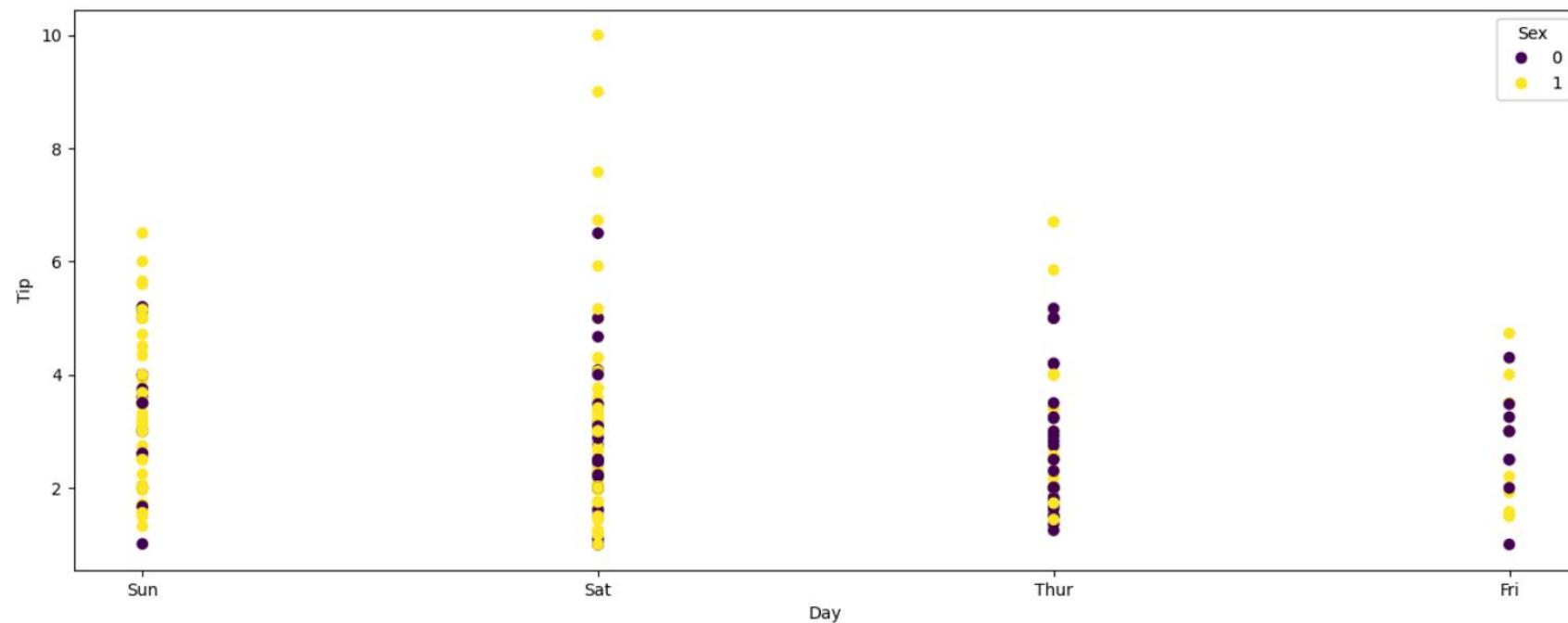
# color change - not working
legend.legendHandles[0].set_color('blue')
legend.legendHandles[1].set_color('red')

ax.add_artist(legend)

plt.xlabel('Day')
plt.ylabel('Tip')

plt.show()

```



```

# Line plot

sns.lineplot(x='day', y='tip', data=df)
plt.show()

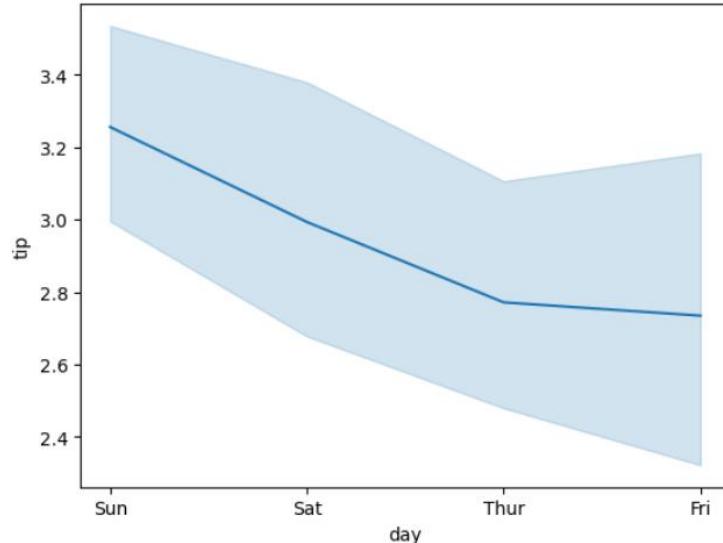
df["tip"].groupby(df['day']).mean()

```



Day

```
# Line plot  
  
sns.lineplot(x='day', y='tip', data=df)  
plt.show()  
  
df["tip"].groupby(df['day']).mean()
```



```
day  
Fri      2.734736842105263133  
Sat      2.993103448275861833  
Sun      3.255131578947368265  
Thur     2.771451612903225392  
Name: tip, dtype: float64
```

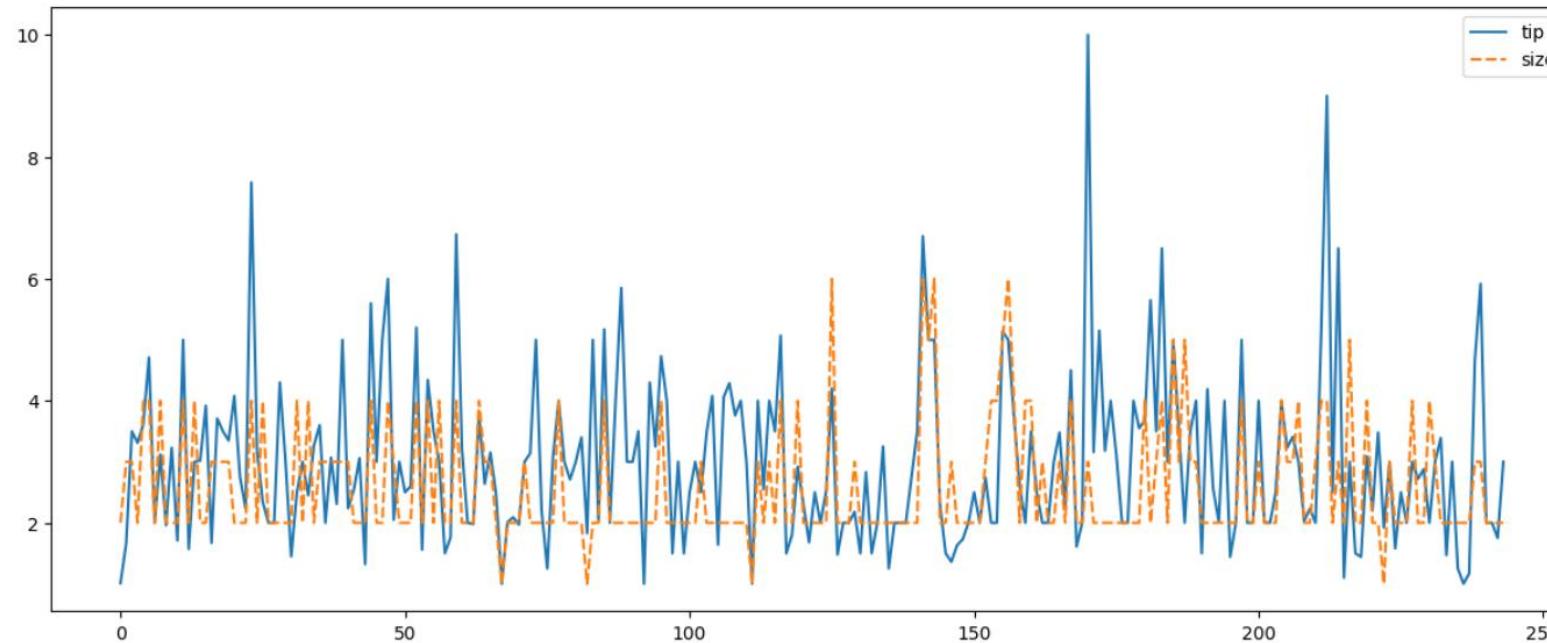
```
# Line plot 2  
  
plt.figure(figsize=(15,6))  
  
sns.lineplot(data=df.drop(['total_bill'], axis=1))  
# df.drop(): Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').  
plt.show()  
  
# Own addition: VS with Matplotlib  
plt.figure(figsize=(15,6))  
plt.plot(df['tip'], label = 'Tip')  
plt.plot(df['size'], label = 'Size')  
plt.legend()  
plt.show()  
  
df[['tip','size']].describe()
```

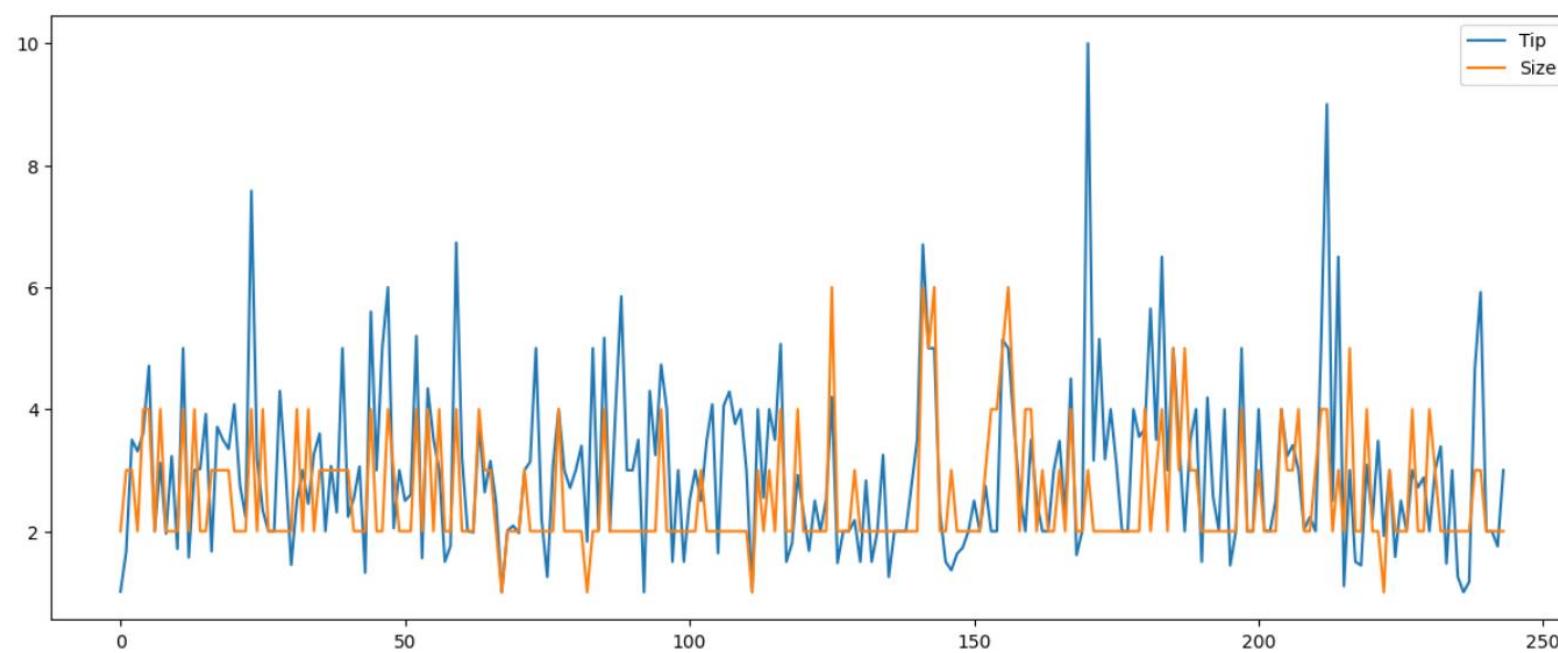
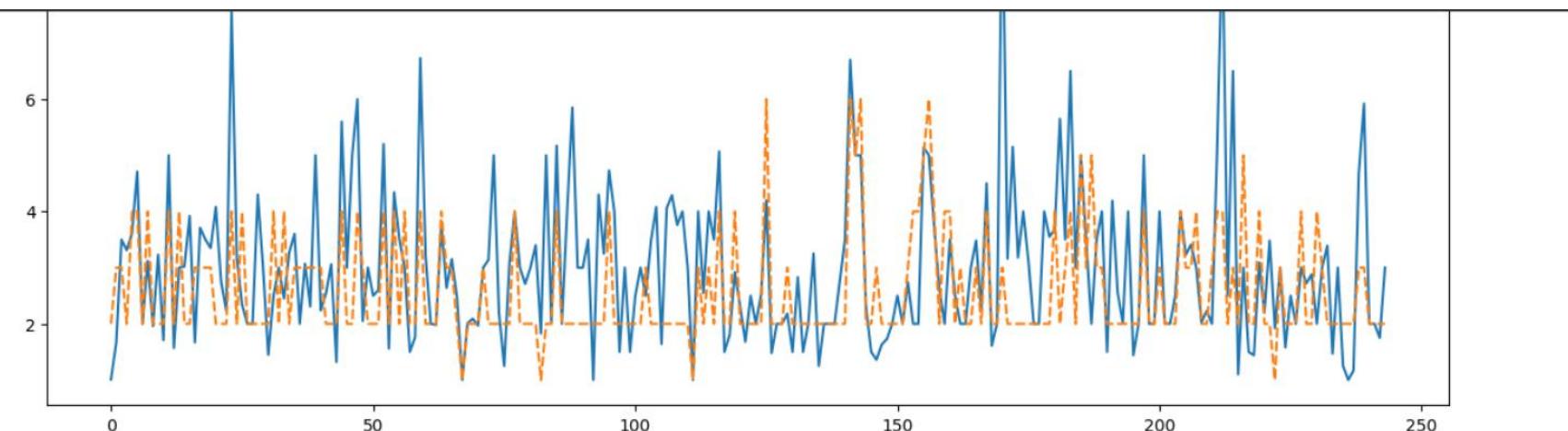
```
Name: tip, dtype: float64
```

```
# Line plot 2  
  
plt.figure(figsize=(15,6))  
  
sns.lineplot(data=df.drop(['total_bill'], axis=1))  
# df.drop(): Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').  
  
plt.show()
```

```
# Own addition: VS with Matplotlib  
plt.figure(figsize=(15,6))  
plt.plot(df['tip'], label = 'Tip')  
plt.plot(df['size'], label = 'Size')  
plt.legend()  
plt.show()
```

```
df[['tip','size']].describe()
```





	tip	size
count	244.00000000000000	244.00000000000000
mean	2.998278688524589786	2.569672131147541005
std	1.383638189001182184	0.951099804732234433
min	1.0000000000000000	1.0000000000000000
25%	2.0000000000000000	2.0000000000000000
50%	2.8999999999999991	2.0000000000000000
75%	3.5625000000000000	3.0000000000000000
max	10.0000000000000000	6.0000000000000000

```

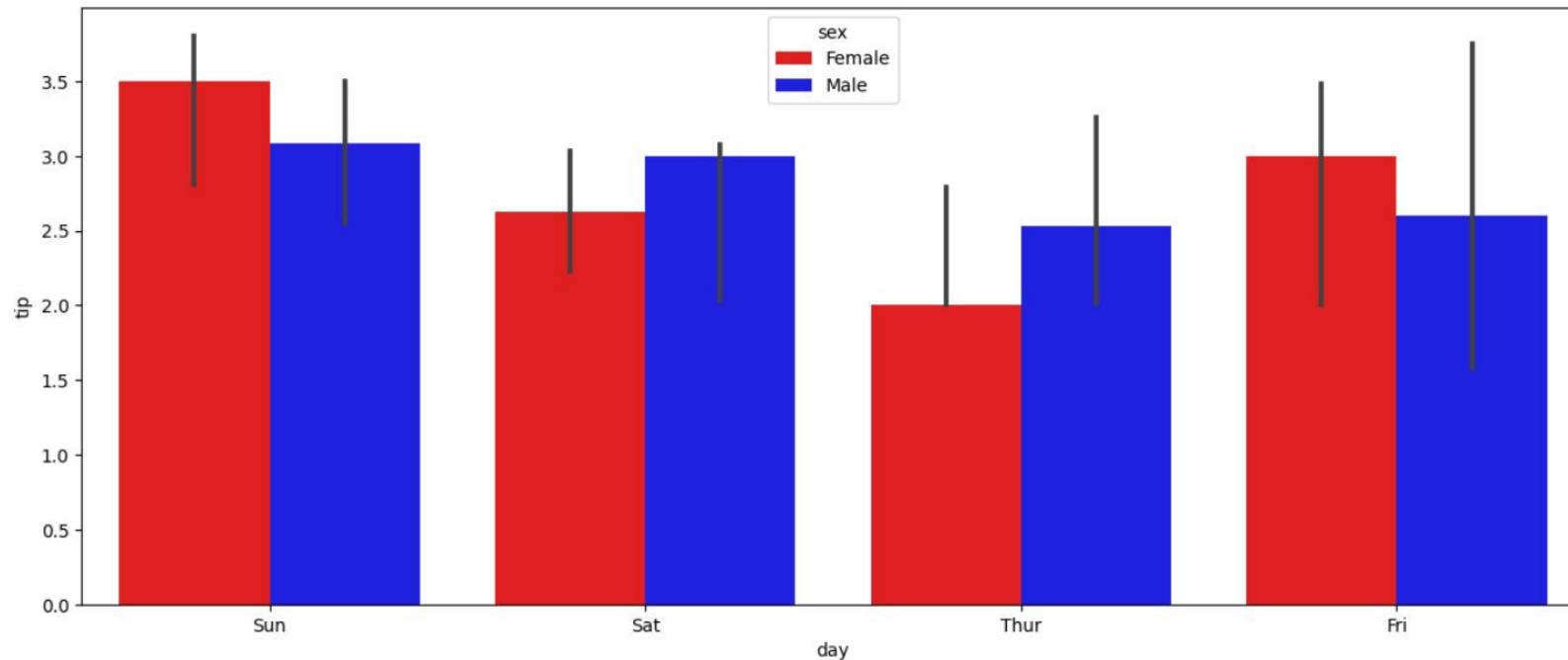
# Bar plot
plt.figure(figsize=(15,6))

sns.barplot(data=df, x='day',y='tip',
            hue='sex', estimator = np.median, palette=['r', 'b'])

plt.show()

df["tip"].groupby([df['day'], df['sex']]).mean()

```



```

day   sex
Fri  Female  2.78111111111111089
      Male    2.69300000000000060
Sat  Female  2.801785714285714324
      Male    3.083898305084745495
Sun  Female  3.36722222222222339
      Male    3.220344827586206993
Thur Female  2.57562500000000053
      Male    2.98033333333333279
Name: tip, dtype: float64

```

```

# Histogram

plt.figure(figsize=(12,3))
sns.histplot(x='total_bill', data=df, kde=True, hue='sex', palette=['r', 'b'], bins=14)
# kde: If True, compute a kernel density estimate to smooth the distribution
# and show on the plot as (one or more) line(s). Only relevant with univariate data.

plt.xlabel('$ (values for Total Bill)')

TotBill_Female_min = df["total_bill"].groupby(df['sex']).min()['Female']
TotBill_Female_max = df["total_bill"].groupby(df['sex']).max()['Female']
plt.xticks(np.linspace(TotBill_Female_min, TotBill_Female_max, 13))
plt.ylabel('Frequency')
plt.show()

```

```

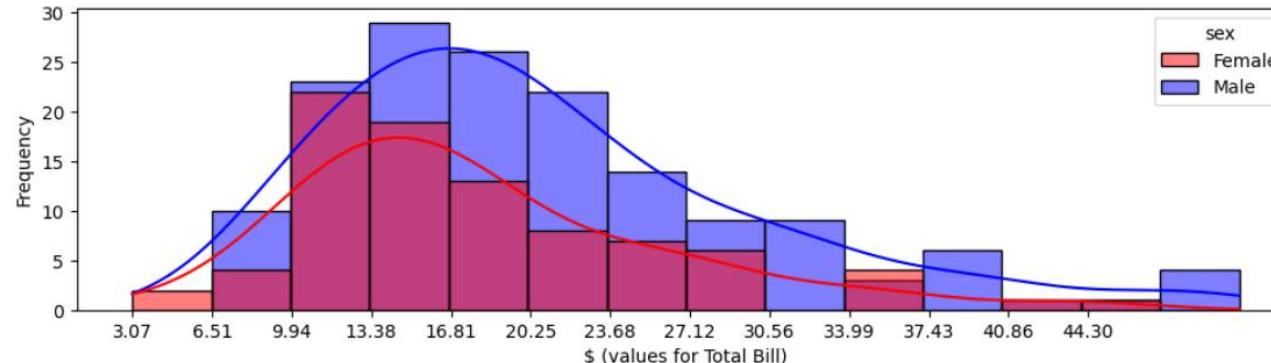
# histogram
plt.figure(figsize=(12,3))
sns.histplot(x='total_bill', data=df, kde=True, hue='sex', palette=['r', 'b'], bins=14)
# kde: If True, compute a kernel density estimate to smooth the distribution
# and show on the plot as (one or more) line(s). Only relevant with univariate data.

plt.xlabel('$ (values for Total Bill)')

TotBill_Female_min = df["total_bill"].groupby(df['sex']).min()['Female']
TotBill_Female_max = df["total_bill"].groupby(df['sex']).max()['Female']
plt.xticks(np.linspace(TotBill_Female_min, TotBill_Female_max, 13))
plt.ylabel('Frequency')
plt.show()

display(df["total_bill"].groupby(df['sex']).describe())
df["total_bill"].groupby(df['sex']).value_counts(bins=12).sort_index()

```



	count	mean	std	min	25%	50%	75%	max
sex								
Female	87.0	18.056896551724133104	8.009208988951961317	3.0699999999999984	12.75	16.3999999999998579	21.520000000000003126	44.2999999999997158
Male	157.0	20.744076433121016123	9.246468761544759829	7.250000000000000	14.00	18.35000000000001421	24.71000000000000853	50.81000000000002274

sex	Female	Male
Female	(3.028, 6.506]	(7.205, 10.88]
	(6.506, 9.942]	(10.88, 14.51]
	(9.942, 13.378]	(14.51, 18.14]
	(13.378, 16.813]	(18.14, 21.77]
	(16.813, 20.249]	(21.77, 25.4]
	(20.249, 23.685]	(25.4, 29.03]
	(23.685, 27.121]	(29.03, 32.66]
	(27.121, 30.557]	(32.66, 36.29]
	(30.557, 33.992]	(36.29, 39.92]
	(33.992, 37.428]	(39.92, 43.55]
	(37.428, 40.864]	(43.55, 47.18]
	(40.864, 44.3]	(47.18, 50.81]

Name: total_bill, dtype: int64

Linear Regression

Mathematics

```
# I've created the content of the cell below in a Word document in a much more readable format
# Like I could use proper formatting for equations
# If you're interested please reach out to me and I'll send the .docx file form of the content below
```

Notation
Bold variables are vectors **and/or** matrices, non-bold variables are scalars

'The Residual Error'
Typically: We assume ϵ **is** Gaussian Distributed (i.e. Gaussian noise)
 $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$
 $p(\epsilon) = \mathcal{N}(\epsilon|\theta, \sigma^2)$
But other distributions are also possible

'The distribution of y'
the error **is** gaussian, thus y **is** gaussian too - a distribution derived from a gaussian distribution **is** also gaussian
conditional probability density **for** the target variable y :
 $p(y|x, w, \sigma) = \mathcal{N}(y|w^T x, \sigma^2)$
Notation: $p(y|x, w, \sigma) = p(y|x, \theta)$

'General Linear Model'
 $f(x) = w^T x + \epsilon = y$,
where $x_0 = 1$ **and** $w_0 = b$ (bias) when $y = mx + b$, m (slope/coefficient) **=** w (weight))
Reminder: bold variables are vectors and/or matrices, non-bold variables are scalars

Note: It **is** assumed that the noise **is** Gaussian distributed! i.e. $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$
-> We assume $\mu = 0$: That means we only have to estimate σ !

The model parameters are:
 $\theta = \{ w_0, w_1, \dots, w_n, \sigma \} = \{ w, \sigma \}$

'Optimal parameters'
We search over all possible θ **and** select the θ which most likely generated our training dataset D
 $\theta^* = \underset{\theta}{\operatorname{argmax}} p(D|\theta)$, θ : mean **and** standard deviation of the data

This process **is** called Maximum Likelihood Estimation (MLE)

Data has a certain mean value **and** a sigma value, **as** well **as** parameters have a certain mean value **and** sigma value
Model: $D \sim \mathcal{N}(\mu, \sigma^2)$, parameters: $\theta = \{\mu, \sigma\}$

$p(D|\theta) = l(\theta)$ **and** **is** called Likelihood: the likelihood of observing our data given a certain theta certain parametrization

'The Likelihood Function'
We assume each sample (each individual (x, y) combination like a vector) **is** independent, identically distributed (i.i.d.)!
 $L(\theta) = p(D|\theta)$
 $= p(s_0, s_1, \dots, s_n|\theta)$
 $= p(s_0|\theta) * p(s_1|\theta) * p(s_n|\theta)$
 $= \prod_{i=1}^n p(s_i|\theta)$

'The Log-Likelihood'
The product of small numbers **in** a computer **is** unstable
We transform the likelihood into the quasi-equivalent Log-Likelihood:
 $L(\theta) = \prod_{i=1}^n p(s_i|\theta) \Leftrightarrow \sum_{i=1}^n \log[p(s_i|\theta)] = l(\theta)$

The logarithm „just“ scales the problem. Likelihood **and** Log-Likelihood both have to be maximized!

The logarithm „just“ scales the problem. Likelihood **and** Log-Likelihood both have to be maximized!

```
'The conditional distribution of p(si|θ)'  
The conditional is the Gaussian distribution of y  
 $p(s_i|\theta) = p(y_i|x_i, w, \sigma) = N(y_i|w^T x_i, \sigma^2)$ , where  $w_0 = \text{intercept}/\text{bias}$  and  $x_{0,i} = 1$ 
```

After inserting this into the log-likelihood,
simplifying **and** removing the constant parts like **for** example sigma, we get the Loss:
(sigma **is** variable of the input, but **in** the sake of modelling it **is** fixed (constant))
 $\mathcal{L}(\theta) = \sum_{i=1}^N [\log[p(s_i|\theta)]]$
= $\sum_{i=1}^N [\log[N(y_i|w^T x_i, \sigma^2)]]$
= simplifying **and** omitting (removing) the constant parts like **for** example sigma
= $-\sum_{i=1}^N [(y_i - w^T x_i)^2]$, where $w_0 = \text{intercept}/\text{bias}$ **and** $x_{0,i} = 1$

this (without the minus sign) **is** the Residual Sum of Squares
we only have to minimize this

```
'The Residual Sum of Squares'  
We minimize the Loss term:  
 $\mathcal{L}(\theta) = \sum_{i=1}^N [(y_i - w^T x_i)^2]$ , where  $w_0 = \text{intercept}/\text{bias}$  and  $x_{0,i} = 1$ 
```

For linear regression: The loss **is** shaped like a bowl **with** a unique minimum

Multiple ways to find the minimum:

- Analytical Solution - In practice too time consuming to compute
- Gradient Descend
- Newtons Method
- Particle Swarm Optimization
- Genetic Algorithms
- Many more methods

```
'MSE'  
 $MSE = J(w,b) = \frac{1}{N} \sum_{i=1}^N [(y_i - (w^T x_i + b))^2]$  (common notation)  
=  $\frac{1}{N} \sum_{i=1}^N [(y_i - w^T x_i)^2]$  (vector notation)  
where  $w_0 = b = \text{intercept} = \text{bias}$   
 $x_1 = (1, 5), x_2 = (1, 10), x_{0,i} = 1$  always for the intercept,  
 $x_{1,1} = 5$  and  $x_{2,1} = 10$  are just examples
```

```
'Quality of the fit: R^2'  
A measure for the percentage of variability of the dependent variable (y) in the data explained by the model
```

```
Total sum of squares: SS_tot =  $\sum_{i=1}^N [(y_i - y_{\text{mean}})^2]$   
Residual sum of squares (unexplained variability): SS_res =  $\sum_{i=1}^N [(y_i - f(x_i))^2]$   
Sum of squares regression (explained variability): SS_regr =  $\sum_{i=1}^N [(f(x_i) - y_{\text{mean}})^2]$ 
```

Coefficient of determination: $R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = \frac{SS_{\text{regr}}}{SS_{\text{tot}}}$

```
'Basis Function expansion'
```

Polynomial regression fits a nonlinear relationship between input of x **and** the corresponding conditional mean of y

Example of a polynomial model:

$y = w_0 + w_1 x + w_2 x^2 + \dots + w_m x^m + \epsilon$

model **is** still linear **in** weights, quadratic **or** polynomic **in** x

How can we use polynomials **in** linear regression?

With Basis function expansion: apply a function before we predict our “linear line”:

$f(x) = \sum_{j=1}^D [w_j \Phi_j(x) + \epsilon] = y$

So we transform our Input-Space by using the function Φ

Example **for** the Basis Function:

$\Phi(x) = (1 \ x \ x^2 \ x^3 \ x^4 \ x^5 \ x^6)^T$

The model parameters are:

$\theta = \{w_0, w_1, w_2, w_3, w_4, w_5, w_6, \sigma\}$

σ : The standard deviation of the gaussian noise

Mathematics behind the `sklearn.datasets.make_regression()` function

```
# Investigating how the underlying mathematics of Linear Regression work within
# sklearn's built-in make_regression() function

biasOrIntercept = 10
StDevOfNoise = 0

from sklearn import datasets
X, y, coefficients = datasets.make_regression(
    n_samples=500, n_features=2, bias = biasOrIntercept, noise=StDevOfNoise, random_state=4, coef = True)
# make_regression:
# generates random Linear regression datasets, with added Gaussian noise to add some randomness.
# The input set (X) can either be well conditioned ('kinda Gaus Distr') (by default) or have a low rank-fat tail singular profile
# The output (Y) is generated by applying a (potentially biased) random Linear regression model with
# n_informative nonzero regressors to the previously generated input and some gaussian centered noise with some adjustable scale

# n_features: single/multiple features/variables for X |
# Note: to display more than one feature the plot is not properly set
# noise: float, default=0.0 The standard deviation of the gaussian noise applied to the output
# In other words: apply some (gaussian centered) noise to the linear model
# coef: ndarray of shape (n_features,) The coefficient (the weights) of the underlying linear model

print(X.shape)
print(y.shape)

plt.grid()
plt.scatter(X[:, 0], y)
plt.title('Dummy dataset generated generated by applying a linear regression model without noise')
plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('Y')
plt.show()

# The underlying Linear model behind the generated data:
#  $f(x) = w^T x + \epsilon = y$ ,  $x_0=1$ 
#  $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ , where we assume  $\mu = 0$ : that means we only have to estimate  $\sigma$ 
# The parameters of the model are:
#  $\theta = \{w_0, w_1, \dots, w_n, \sigma\} = \{w, \sigma\}$ 

print("w0 =", biasOrIntercept) # = b (bias) when  $y = mx + b$ ,  $w_0 = b$ 
print("w1 =", coefficients[0].astype(float))
print("w2 =", coefficients[1].astype(float))
print("x0 = 1")
print("sigma = ", StDevOfNoise) # =>  $\epsilon = 0$  because  $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ , assumption  $\mu = 0$ 
print("epsilon = 0 <= As  $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$  and  $\sigma = 0$ ,  $\mu = 0$ ")
print()

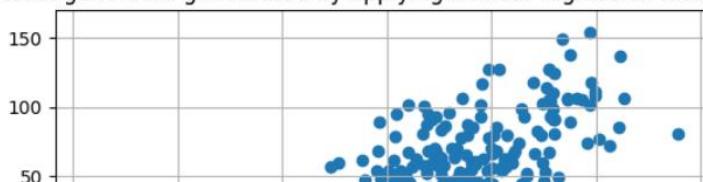
for i in range(0, 2):
    print(f"x1 = {X[i, 0]}, x2 = {X[i, 1]}, y = {y[i]}")

# With calculator
#  $x_1 = 1.0910373925705656, x_2 = -1.1104262019784206, y = 30.10364701683811$ 
#  $f(x) = w^T x + \epsilon = w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + \theta(\epsilon) = 10 * 1 + 48.59557438453652 * 1.0910373925705656 + 29.642619826054194 * -1.1104262019784206 + 0 = 30.1036 = y$ 

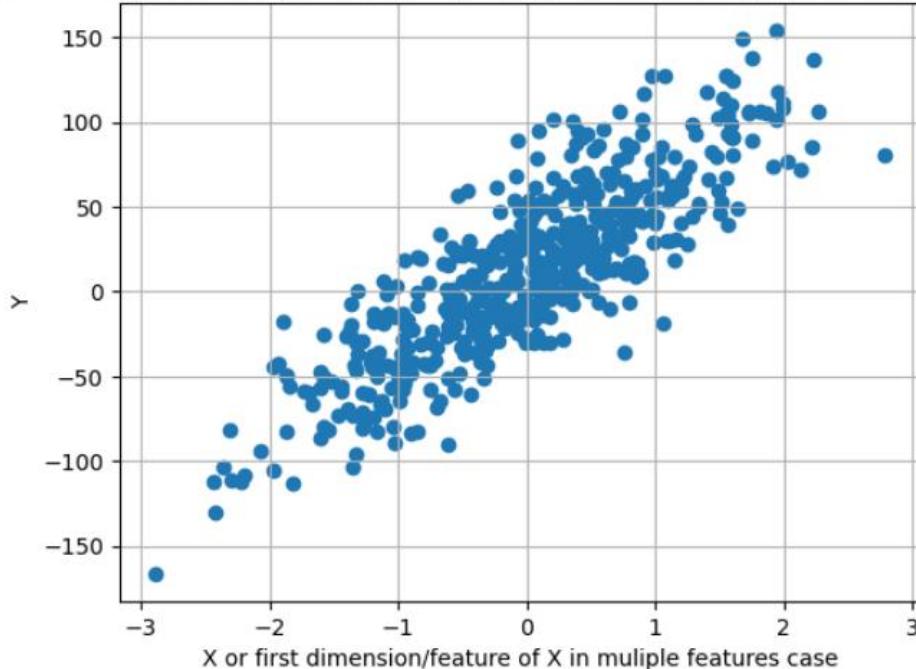
#  $x_{-1} = 0.010833796049367062, x_{-2} = -0.6803566054338689, y = -9.641077659237016$ 
#  $f(x) = w^T x + \epsilon = w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + \theta(\epsilon) = 10 * 1 + 48.59557438453652 * 0.010833796049367062 + 29.642619826054194 * -0.6803566054338689 + 0 = -9.6410 = y$ 
```

(500, 2)
(500,)

Dummy dataset generated generated by applying a linear regression model without noise



Dummy dataset generated by applying a linear regression model without noise



```
w0 = 10
w1 = 48.59557438453652
w2 = 29.642619826054194
x0 = 1
σ = 0
ε = 0 <= As ε ~ N(μ, σ²) and σ = 0, μ = 0
```

```
x1 = 1.0910373925705656, x2 = -1.1104262019784206, y = 30.10364701683811
x1 = 0.010833796049367062, x2 = -0.6803566054338689, y = -9.641077659237016
```

Creating model from scratch using NumPy (with mathematics in comments)

```
# Introduction

# Even though Linear Regression only works with simple straight-line patterns (in the default case)
# understanding the math behind it helps us understand Gradient Descent and Loss Minimization methods,
# which are important for models used in machine learning and deep Learning tasks

# General Linear Model

#  $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + \epsilon = y$ , where  $x_0 = 1$  and  $w_0$ : intercept ( $w_0 = b$  (bias) when  $y = mx + b$ ,  $m$  (slope/coefficients) =  $w$  (weights))
# Bold variables are vectors and/or matrices, non-bold variables are scalars

# It is assumed that the noise is Gaussian distributed i.e.  $\epsilon \sim N(\mu, \sigma^2)$ 
# -> We assume  $\mu = 0$ : That means we only have to estimate  $\sigma$ 
# The model parameters are:
#  $\theta = \{ w_0, w_1, \dots, w_n, \sigma \} = \{ w, \sigma \}$ 

# Note: sigma is variable of the input, but in the sake of modelling and optimization it is fixed (constant),
# therefore when we computed the Residual Sum of Squares (in the above Mathematics cell) it got removed,
# thus for further calculations like MSE (loss) we don't have to compute it

class LinRegNumPy:

    # Class initialization
    # Setting the the Learning rate and number of iterations hyperparameters to a default value for the Gradient Descent
    # The number of weight parameters depends on the input features within the data,
    # and as we do not have access to data yet, initializing them to None for now
    def __init__(self,
                 learning_rate = 0.1,
                 n_iters = 1000):
```

```

class LinRegNumPy:

# Class initialization
# Setting the the learning rate and number of iterations hyperparameters to a default value for the Gradient Descent
# The number of weight parameters depends on the input features within the data,
# and as we do not have access to data yet, initializing them to None for now
    def __init__(self,
                 learning_rate = 0.1,
                 n_iters = 1000):

        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None # =intercept

# Normalization of the independent variables
# As the implementation is general, the model can deal with any number of features
# Actually normalization is not needed for linear regression, as the
# calculated coefficients account for the appropriate scaling
# However, when using gradient descent-based optimization algorithms, feature scaling has some advantages,
# for example it can help speed up convergence, help improve model performance, and
# ensures that the coefficients are on the same scale, making it possible to compare
# the impact of different predictors on the dependent variable
# So I decided to include normalization
# (it did actually improve performance as I first didn't include normalization and then compared results using it)
# The other reason why I decided to include besides advantages is that other models do require normalization
# (like Lasso, Ridge, Elastic Net regressions) because of the penalty coefficients,
# and I want my model implementation to be as general as possible
    def _normalizeX(self, X):
        for i in range(X.shape[1]):
            X[:, i] = ((X[:, i] - np.mean(X[:, i]))/np.std(X[:, i]))
            # different method for averaging over columns of X:
            # X = ((X - np.mean(X, axis=0)) / np.std(X, axis=0))
            # computes less accurate results, please see the cell below for comparison

        return X

# Cost/loss function
# Let's take the Mean Squared Error (MSE) as the cost/loss function
# (MSE is the mean of the Residual Sum of Squares, for further information please see the Mathematics cell above)
# (another common cost function is the half MSE - division by two is for the sake of cleaner MSE derivation calculus)
# We will find its minimum with Gradient Descent
#  $J(w,b) = \frac{1}{N} \sum_{i=1}^N [(y_i - (w \cdot x_i + b))^2]$  (common notation)
    def _loss_function(self, X, y_pred, y):
        num_samples = X.shape[0]
        return (1 / num_samples) * np.sum((y_pred-y)**2)
        # same: return np.mean(np.square(y_pred-y))
        # same: return mean_squared_error(y_pred, y)

# Gradient Descent
# It's the training process to find optimal weights and bias
# We aim to find optimal values of the parameters which generate a line
# that minimizes the loss (converging towards minimum of the loss function),
# so the squared difference between predicted and actual y values (Residual Sum of Squares)

# There are multiple ways to find the minimum of the loss function (now MSE), for example Gradient Descent:
# We take the partial derivatives (gradients) of the MSE with respect to weights and bias
# Note: When we have two or more derivatives of the same function, they are called a Gradient
#  $\frac{\partial J}{\partial w} = \frac{1}{N} \sum_{i=1}^N [2x_i \cdot (y_i - (w \cdot x_i + b))]$ 

    def _GradDesc_computeWeightDeriv(self, X, y_pred, y):
        num_samples = X.shape[0]
        #  $\frac{\partial J}{\partial w} = \frac{1}{N} \sum_{i=1}^N [2x_i \cdot (y_i - (w \cdot x_i + b))]$ 
        # =  $\frac{1}{N} \sum_{i=1}^N [2x_i \cdot (y_i - (w \cdot x_i + b))] = \frac{1}{N} \sum_{i=1}^N [-2x_i \cdot (y_i - y_{i\_pred})] =$ 
        # =  $\frac{1}{N} \sum_{i=1}^N [2x_i \cdot (y_{i\_pred} - y_i)]$ 
        # We omit the constant 2
        return (1 / num_samples) * np.dot(X.T, y_pred - y)
        # X.T (num_features, num_samples) multiply by (y_pred - y) (num_samples, ) = dw (num_features, )
        # So we have a separate derivate value for each weight, and we optimize them separately

```

```

# We omit the constant 2
return (1 / num_samples) * np.dot(X.T, y_pred - y)
# X.T (num_features, num_samples) multiply by (y_pred - y) (num_samples, ) = dw (num_features, )
# So we have a separate derivate value for each weight, and we optimize them separately

def _GradDesc_computeBiasDeriv(self, X, y_pred, y):
    num_samples = X.shape[0]
    # J'(w,b) = dJ/db =
    # = db (bias derivative) =
    # = 1/N *  $\sum_{i=1}^N$  [ -2 * (y_i - (w*x_i + b)) ] = 1/N *  $\sum_{i=1}^N$  [-2 * (y_i - y_i_pred)] =
    # = 1/N *  $\sum_{i=1}^N$  [ 2 * (y_i_pred - y_i) ]
    # We omit the constant 2
    return (1 / num_samples) * np.sum(y_pred - y)
    # (y_pred - y).shape: (num_samples, ), thus db.shape: (num_samples, )
    # So the derivate value for the bias is a single value just as the bias is

# We took the gradient of the loss with respect to each weight
# We need to push J(w) towards its minimum (converge towards minimum)
# The gradient of MSE is usually initially positive (but depends on the initial weights),
# so we decrease the weight to move towards the minimum
# (moving backwards on the x-axis or 'moving the values in the opposite direction of the gradient, because the gradient of MSE usually points in the direction of increase not decrease, but this depends on the initial weights'):
# weights = weights - alpha (Learning rate) * dw (weights partial derivative of the loss)
# bias = bias - alpha (learning rate) * db (bias partial derivative of the loss)
# The learning rate (or alpha) controls (scales) the size of the steps
# We start with random weight values and change them slightly for multiple steps
# We only make a small change in the value for each step for stable movement towards the minimum
def _GradDesc_updateWeightValues(self, dw):
    return self.weights - self.learning_rate * dw

def _GradDesc_updateBiasValue(self, db):
    return self.bias - self.learning_rate * db

# Limitations of the Gradient Descent
# There is no guarantee that the algorithm converges to the global optimum, for example:
# The learning rate can lead to slow convergence if not properly configured
# Or it will never converge because the update value is too large (just jumping around minimum value)
# The starting parameters can lead to a solution stuck in a local minimum (see
# there are ways to adapt gradually during learning the learning rate:
# dynamic Learning rate update, adaptive Learning rate

# Plotting the loss/cost in the function of num_iters
def _plot_cost(self, MSELossFuncValues):
    plt.grid()
    plt.xlabel('Epochs (Number of iterations)')
    plt.ylabel('MSE (Loss/Cost function)')
    plt.plot(range(self.n_iters), MSELossFuncValues, 'm', linewidth = "5")
    plt.xticks(np.linspace(0, self.n_iters, 11))
    plt.yticks(np.linspace(0, int(round(MSELossFuncValues.max(),-3)), 20))
    plt.show()

def fit(self, X, y, GradDescPlot = False):

    # Training initialization
    # Normalizing data of independent variables
    X = self._normalizeX(X)

    # Reading data of independent variables shape
    num_samples, num_features = X.shape # X.shape: (num_samples, num_features)

    # Assigning random initial values for weights and 0 for bias,
    self.weights = np.random.rand(num_features) # weights shape: (num_features, )
    # np.random.rand: Create an array of the given shape and populate it with random samples
    # from a uniform distribution over [0, 1)
    self.bias = 0

    # Creating collector for MSE values of each iteration
    MSELossFuncValues = np.empty(0)

```

```

# from a uniform distribution over [0, 1)
self.bias = 0

# Creating collector for MSE values of each iteration
MSELossFuncValues = np.empty(0)

# Training process
# Incrementally performing the steps to find optimal values for the parameters
# by converging towards minimum of the loss function
for i in range(self.n_iters):

    # 1. step: Get predictions with current weights
    # so that in the next step we are able to compute the partial derivatives (gradient) of the Loss function
    # X (num_samples, num_features) multiply by weights (num_features, ) = y_pred (num_samples, )
    # The bias value is added at the end of each prediction
    y_pred = np.dot(X, self.weights) + self.bias # could create a '_predict' function instead and
                                                # use here and for the 'predict' function,
                                                # but this approach seems to be worse for me here... ?

    # ----- just for investigating the Gradient Descent in action -----
    # If GradDescPlot is set to True,
    # then compute and collect Loss (MSE) values in each iteration during the training process
    if GradDescPlot == True:
        MSELossFunc = self._loss_function(X, y_pred, y)
        MSELossFuncValues = np.append(MSELossFuncValues, MSELossFunc)

        if i % 100 == 0:
            print(f"\n_n_iters: {i}, MSE LinRegNumPy: {MSELossFunc}")
            print(f"Current value(s) for weight(s) in this iteration: {np.float64(self.weights)}\n")
            # for checking: print(f"\n_n_iters: {i} || MSE BuiltIn: {mean_squared_error(y_pred,y)}")

    # ----- just for investigating the Gradient Descent in action -----


    # 2. step: Take the weights and bias partial derivatives (gradient) of the Loss function (MSE)
    # so that in the next step we are able to update the parameters
    # Partial derivative values in the current iteration of GradDesc:
    weightDeriv = self._GradDesc_computeWeightDeriv(X, y_pred, y)
    biasDeriv = self._GradDesc_computeBiasDeriv(X, y_pred, y)

    # 3. step: Update weights and bias
    # so that with the updated parameters predictions have (produce) a smaller Loss
    self.weights = self._GradDesc_updateWeightValues(weightDeriv)
    self.bias = self._GradDesc_updateBiasValue(biasDeriv)

    # ----- just for investigating the Gradient Descent in action -----
    # Plotting the loss/cost in the function of num_iters
    if GradDescPlot == True:
        print(f"Final value(s) for weight(s) set by {self.n_iters} training interations (iterations of minimizing the Loss): {np.float64(self.weights)}")
        self._plot_cost(MSELossFuncValues)
    # ----- just for investigating the Gradient Descent in action -----


return self

# Predicting with trained model
# After training (updating the parameters in incremental steps) now we have the optimal set of weights and bias
# The predicted values should now be close to the original values
# X (num_samples, num_features) multiply by weights (num_features, ) = y_pred (num_samples, )
# The bias value is added at the end of each prediction
def predict(self, X):
    X = self._normalizeX(X)
    return np.dot(X, self.weights) + self.bias

# Comparing different column averaging methods for normalization:
# np.mean/std(X, axis=0) vs. np.mean/std(X[:, columnIndex])

```

```

# Comparing different column averaging methods for normalization:
# np.mean/std(X, axis=0) vs. np.mean/std(X[:, columnIndex])

X, y = datasets.make_regression(n_samples=600, bias = 30, n_features=2, noise = 15, random_state=4)
X2 = X.copy()

X2 = ((X2 - np.mean(X2, axis=0)) / np.std(X2, axis=0))

X3 = X.copy()
for i in range(0, X3.shape[1]):
    X3[:, i] = ((X3[:, i] - np.mean(X3[:, i]))/np.std(X3[:, i]))

comparison = {
    'Mean of X columns using axis': [np.mean(X, axis=0), np.mean(X2, axis=0), np.mean(X3, axis=0)],
    'Mean of X columns with indexing': [[np.mean(X[:, 0]), np.mean(X[:, 1])], [np.mean(X2[:, 0]), np.mean(X2[:, 1])], [np.mean(X3[:, 0]), np.mean(X3[:, 1])]],
    'Mean of whole X': [X.mean(), X2.mean(), X3.mean()],
    '': ['.', '.', '.'],
    'Std of X columns using axis': [np.std(X, axis=0), np.std(X2, axis=0), np.std(X3, axis=0)],
    'Std of X columns with indexing': [[np.std(X[:, 0]), np.std(X[:, 1])], [np.std(X2[:, 0]), np.std(X2[:, 1])], [np.std(X3[:, 0]), np.std(X3[:, 1])]],
    'Std of whole X': [X.std(), X2.std(), X3.std()]
}

df = pd.DataFrame.from_dict(comparison, orient='index', columns=['Original X', 'X normalized using axis', 'X normalized with indexing'])
df.index.name = 'Measure on (normalized) X'
pd.set_option("display.precision", 18)
display(df)

```

	Original X	X normalized using axis	X normalized with indexing
Measure on (normalized) X			
Mean of X columns using axis	[0.0007967313319007003, 0.09053283843981025]	[4.838722015657974e-17, -2.312964634635743e-17]	[3.839521293495333e-17, 5.736152293896642e-18]
Mean of X columns with indexing	[0.0007967313319007137, 0.0905328384398102]	[2.960594732333751e-17, -6.513308411134252e-17]	[2.8125649957170635e-17, -2.0724163126336256e-17]
Mean of whole X	0.045664784885855458	-0.000000000000000015	0.000000000000000006
Std of X columns using axis	[0.9757849010980479, 0.9540903315309516]	[0.9999999999999998, 1.0]	[0.9999999999999994, 1.000000000000002]
Std of X columns with indexing	[0.9757849010980482, 0.9540903315309516]	[1.0000000000000002, 1.0]	[1.0, 1.0]
Std of whole X	0.96604110119373543	1.0000000000000022	1.0

Applying created model on data, visualizing Gradient Descent in action and performance benchmarking

```

# Using the created model on data

# Dataset (dummy) generating method with Scikit-Learn by applying a Linear regression model with some noise to X
# X: well conditioned data ('kinda' Gaus Distr)
# Y: generated by applying a (potentially biased) Linear regression model
# For the interpretation of this sklearn function and for more information please see the
# '### Mathematics behind the sklearn.datasets.make_regression() function' Notebook heading
X, y = datasets.make_regression(n_samples=600, bias = 30, n_features=1, noise=15, random_state=4)
# Note: to display multiple features the plots below are not properly set

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples)
print("Number of generated features:", num_features)

# Dataset plotting
plt.figure(figsize=(6,6))
plt.grid()
plt.title('Dummy dataset generated by applying a linear regression model with some noise', fontsize = 10)
plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('y')

```

```

plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('y')
plt.scatter(X[:, 0], y)
plt.show()

# Dataset splitting
X_train, X_test, y_test = train_test_split(X, y, test_size=0.2, random_state=1235)
print("Train shape:", X_train.shape)
print("Test shape:", X_test.shape)

# Performance of NumPy model without training (random values for weights and 0 for bias)
regressorNoTrain = LinRegNumPy(n_iters=0)
y_pred_noTrain = regressorNoTrain.fit(X_train, y_train).predict(X_test)

MSE_ModelsOwnFunc = regressorNoTrain._loss_function(X_test, y_pred_noTrain, y_test)
MSE_NumPy = np.mean(np.square(y_pred_noTrain - y_test))
MSE_scikitlearn = mean_squared_error(y_pred_noTrain, y_test)

plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('LinRegNumPy without training (random weights, bias=30)')
plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
plt.scatter(X_test[:, 0], y_test, color = 'green', label = 'Test set')
# plt.scatter(X[:, 1], y, label='Whole dataset')
plt.plot(X_test[:, 0], y_pred_noTrain, linewidth=0.8, color='black', label='Predictions on test set without training')
plt.legend(loc='upper left')
plt.text(1.1, 0, f'\nMSE ModelsFunc: {round(MSE_ModelsOwnFunc,5)}\nMSE NumPy: {round(MSE_NumPy,5)}\nMSE sklearn: {round(MSE_scikitlearn,5)}', fontsize = '9', ha='left', va='top')
plt.show()


# Performance of NumPy model with training
regressor = LinRegNumPy(n_iters=1001)
y_pred = regressor.fit(X_train, y_train).predict(X_test)

MSE_ModelsOwnFunc = regressor._loss_function(X_test, y_pred, y_test)
MSE_NumPy = np.mean(np.square(y_pred - y_test))
MSE_scikitlearn = mean_squared_error(y_pred, y_test)

plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('LinRegNumPy with training')
plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
plt.scatter(X_test[:, 0], y_test, color = 'green', label = 'Test set')
# plt.scatter(X[:, 1], y, label='Whole dataset')
plt.plot(X_test[:, 0], y_pred, linewidth=0.8, color='black', label='Predictions on test set after training')
plt.legend(loc='upper left')
plt.text(1.2, 0, f'\nMSE ModelsFunc: {round(MSE_ModelsOwnFunc,5)}\nMSE NumPy: {round(MSE_NumPy,5)}\nMSE sklearn: {round(MSE_scikitlearn,5)}', fontsize = '9', ha='left', va='top')
plt.show()


# Performance of Scikit-Learn's built-in LinearRegression model (with training)
regressorSklearn = sklearn.linear_model.LinearRegression()
y_pred = regressorSklearn.fit(X_train, y_train).predict(X_test)

sklearnModel_MSE_NumPy = np.mean(np.square(y_pred - y_test))
sklearnModel_MSE_scikitlearn = mean_squared_error(y_pred, y_test)

plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('LinearRegression() from sklearn with training')
plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')

```

```

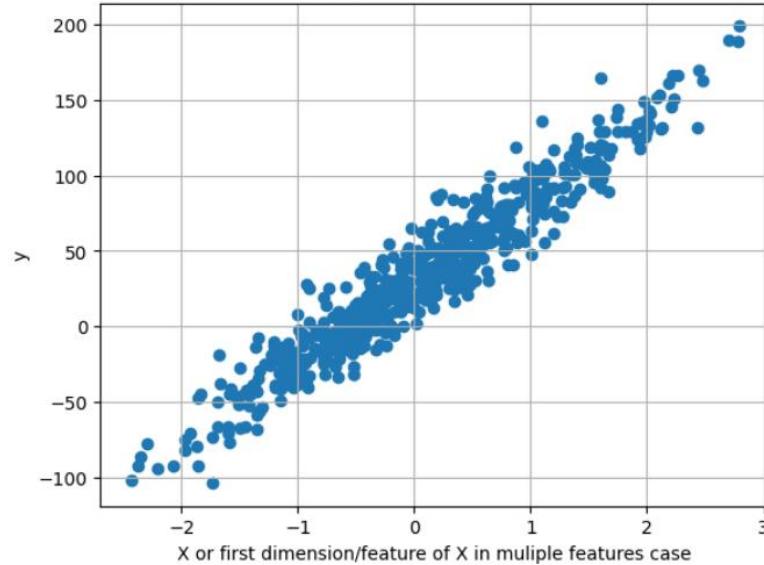
plt.figure(figsize=(10, 6))
plt.grid()
plt.title('LinearRegression() from sklearn with training')
plt.xlabel('X or first dimension/feature of X in multiple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
plt.scatter(X_test[:, 0], y_test, color = 'green', label = 'Test set')
# plt.scatter(X[:, 1], y, label='Whole dataset')
plt.plot(X_test[:, 0], y_pred, linewidth=0.8, color='black', label='Predictions on test set after training')
plt.legend(loc='upper left')
plt.text(1.5, 0, f'\nMSE NumPy: {round(sklearnModel_MSE_Numpy,5)}\nMSE sklearn: {round(sklearnModel_MSE_scikitlearn,5)}', fontsize = '9', ha='left', va='top')
plt.show()

```

Number of generated samples: 600

Number of generated features: 1

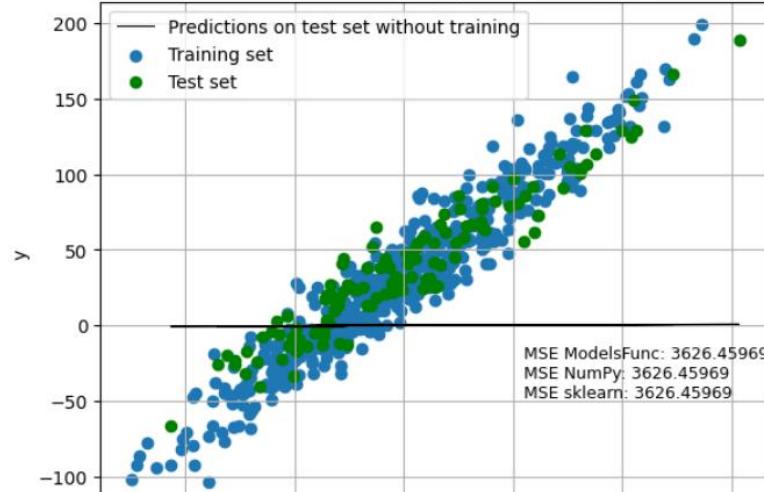
Dummy dataset generated by applying a linear regression model with some noise



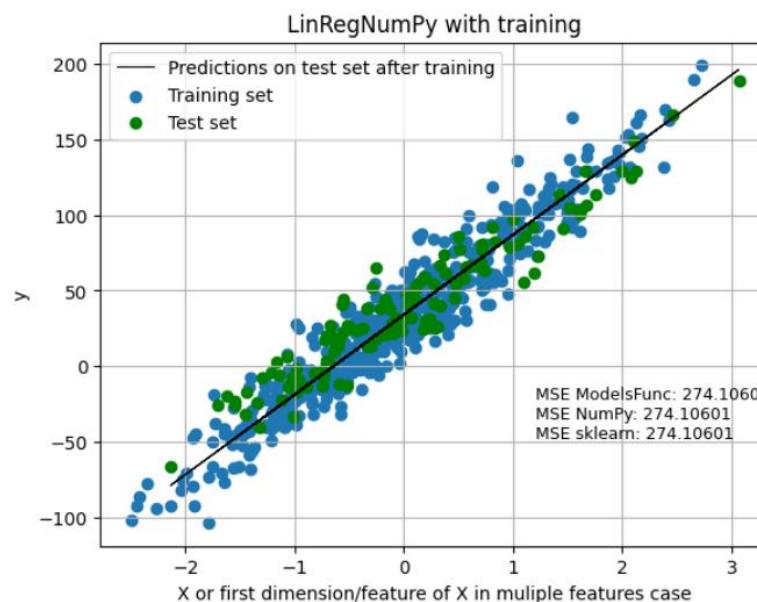
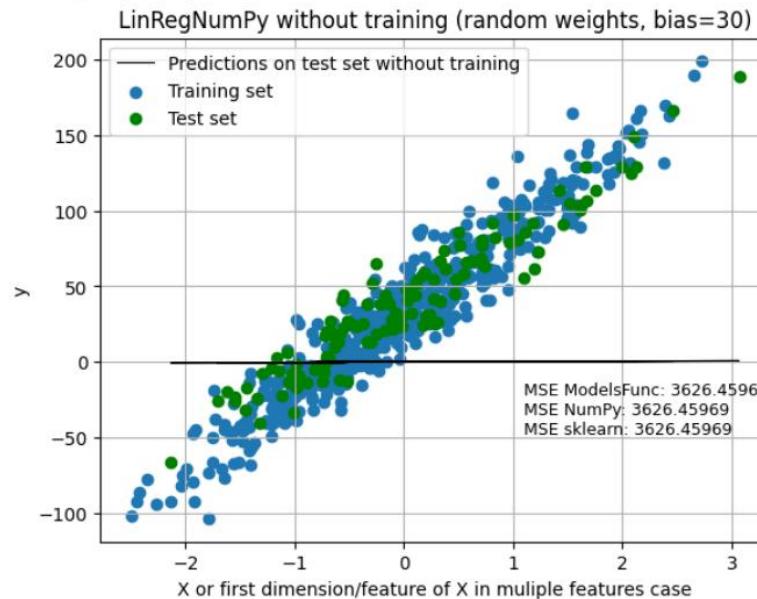
Train shape: (480, 1)

Test shape: (120, 1)

LinRegNumPy without training (random weights, bias=30)

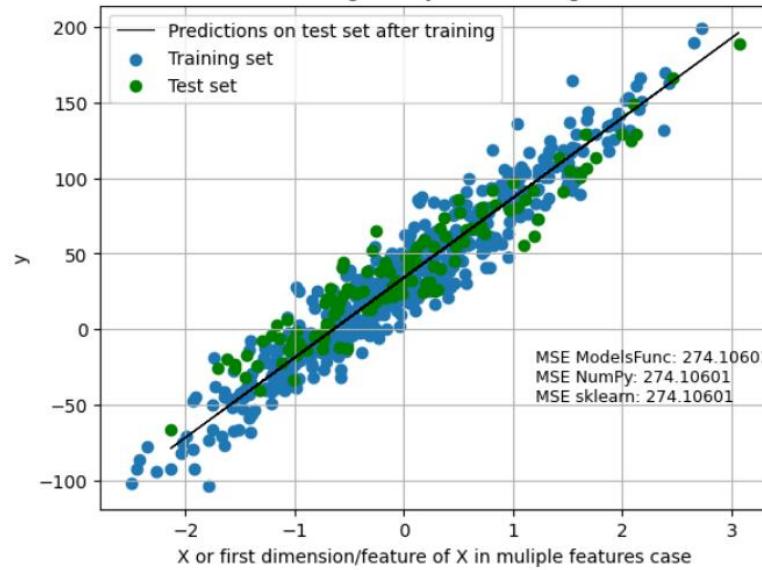


Train shape: (480, 1)
Test shape: (120, 1)

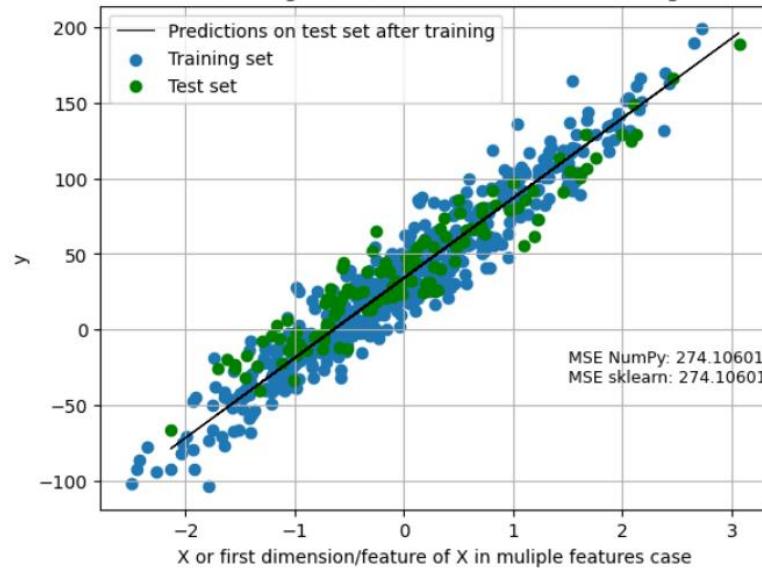


X or first dimension/feature of X in multiple features case

LinRegNumPy with training



LinearRegression() from sklearn with training



```
# Visualizing Gradient Descent in work during the training process:  
# Plot of the loss/cost in the function of num_iters  
# Two different Learning rates for comparison  
  
# In the above cell we already trained, predicted with and calculated MSE with the  
# learning_rate = 0.1 model (default setting for the LinRegNumPy class)  
# Here we only call it (and not assign it to a variable) just for visualization (GradDescplot = True)
```

```

# Visualizing Gradient Descent in work during the training process:
# Plot of the loss/cost in the function of num_iters
# Two different learning rates for comparison

# In the above cell we already trained, predicted with and calculated MSE with the
# learning_rate = 0.1 model (default setting for the LinRegNumPy class)
# Here we only call it (and not assign it to a variable) just for visualization (GradDescplot = True)
LinRegNumPy(n_iters=1001, learning_rate = 0.1).fit(X_train, y_train, GradDescPlot = True)

# Different Learning rate to do comparison in the next cell
lowerLRRegressor = LinRegNumPy(n_iters=1001, learning_rate=0.01)
# So far we don't have MSE score for the learning_rate = 0.01 model, therefore we predict with it and calculate MSE
y_pred = lowerLRRegressor.fit(X_train, y_train, GradDescPlot=True).predict(X_test)
lowerLR_MSE_NumPy = np.mean(np.square(y_pred - y_test))

```

n_iters: 0, MSE LinRegNumPy: 4124.079402046711
 Current value(s) for weight(s) in this iteration: 0.6356580358223799

n_iters: 100, MSE LinRegNumPy: 239.13626897670375
 Current value(s) for weight(s) in this iteration: 52.79759768708586

n_iters: 200, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.79898318117169

n_iters: 300, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.79898321797235

n_iters: 400, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 500, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 600, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

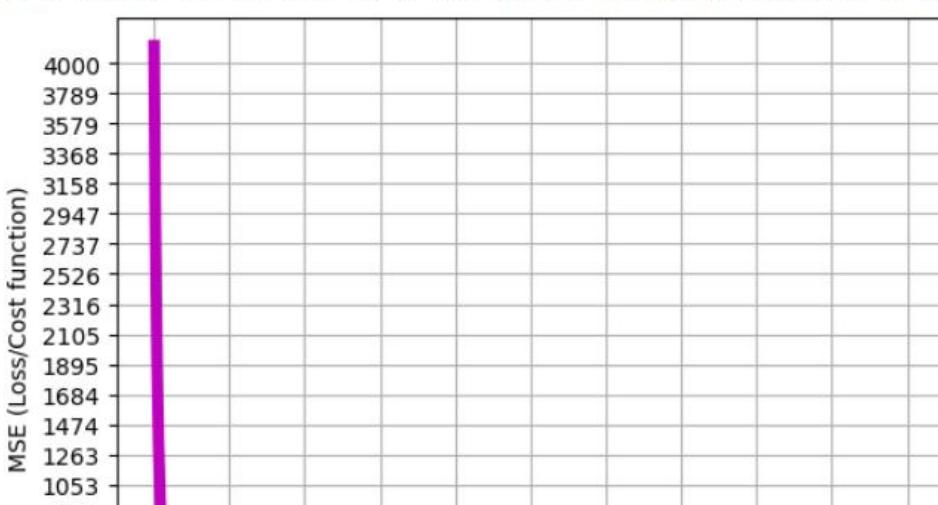
n_iters: 700, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 800, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 900, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 1000, MSE LinRegNumPy: 239.13626623584562
 Current value(s) for weight(s) in this iteration: 52.798983217973294

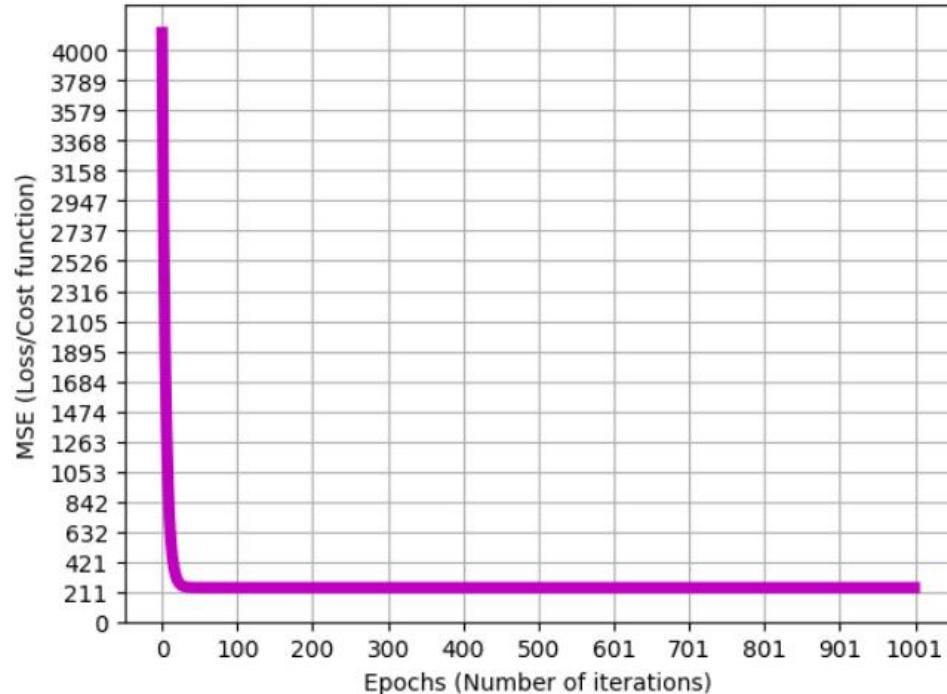
Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.798983217973294



```
n_iters: 1000, MSE LinRegNumPy: 239.13626623584562
```

```
Current value(s) for weight(s) in this iteration: 52.798983217973294
```

```
Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.798983217973294
```



```
n_iters: 0, MSE LinRegNumPy: 4166.358333635428
```

```
Current value(s) for weight(s) in this iteration: 0.23196479722265828
```

```
n_iters: 100, MSE LinRegNumPy: 765.304201921055
```

```
Current value(s) for weight(s) in this iteration: 33.55775439167297
```

```
n_iters: 200, MSE LinRegNumPy: 309.6320751796347
```

```
Current value(s) for weight(s) in this iteration: 45.75607118170867
```

```
n_iters: 300, MSE LinRegNumPy: 248.5812717969835
```

```
Current value(s) for weight(s) in this iteration: 50.22104963595796
```

```
n_iters: 400, MSE LinRegNumPy: 240.40170500995845
```

```
Current value(s) for weight(s) in this iteration: 51.85537615330138
```

```
n_iters: 500, MSE LinRegNumPy: 239.3058093113539
```

```
Current value(s) for weight(s) in this iteration: 52.45359251484948
```

```
n_iters: 600, MSE LinRegNumPy: 239.15898156197667
```

```
Current value(s) for weight(s) in this iteration: 52.672559050254904
```

```
n_iters: 700, MSE LinRegNumPy: 239.13930962785494
```

```
Current value(s) for weight(s) in this iteration: 52.752707883869846
```

```
n_iters: 800, MSE LinRegNumPy: 239.13667398851751
```

```
Current value(s) for weight(s) in this iteration: 52.78204494908824
```

```
n_iters: 900, MSE LinRegNumPy: 239.13632086641604
```

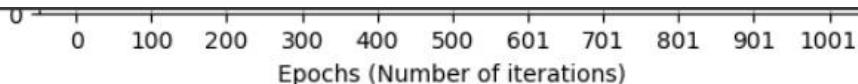
```
Current value(s) for weight(s) in this iteration: 52.79278326375621
```

```
n_iters: 1000, MSE LinRegNumPy: 239.13627355523167
```

```
Current value(s) for weight(s) in this iteration: 52.79671383421545
```

```
Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.79673652805303
```





```

n_iters: 0, MSE LinRegNumPy: 4166.358333635428
Current value(s) for weight(s) in this iteration: 0.23196479722265828

n_iters: 100, MSE LinRegNumPy: 765.304201921055
Current value(s) for weight(s) in this iteration: 33.55775439167297

n_iters: 200, MSE LinRegNumPy: 309.6320751796347
Current value(s) for weight(s) in this iteration: 45.75607118170867

n_iters: 300, MSE LinRegNumPy: 248.5812717969835
Current value(s) for weight(s) in this iteration: 50.22104963595796

n_iters: 400, MSE LinRegNumPy: 240.40170500995845
Current value(s) for weight(s) in this iteration: 51.85537615330138

n_iters: 500, MSE LinRegNumPy: 239.3058093113539
Current value(s) for weight(s) in this iteration: 52.45359251484948

n_iters: 600, MSE LinRegNumPy: 239.15898156197667
Current value(s) for weight(s) in this iteration: 52.672559050254904

n_iters: 700, MSE LinRegNumPy: 239.13930962785494
Current value(s) for weight(s) in this iteration: 52.752707883869846

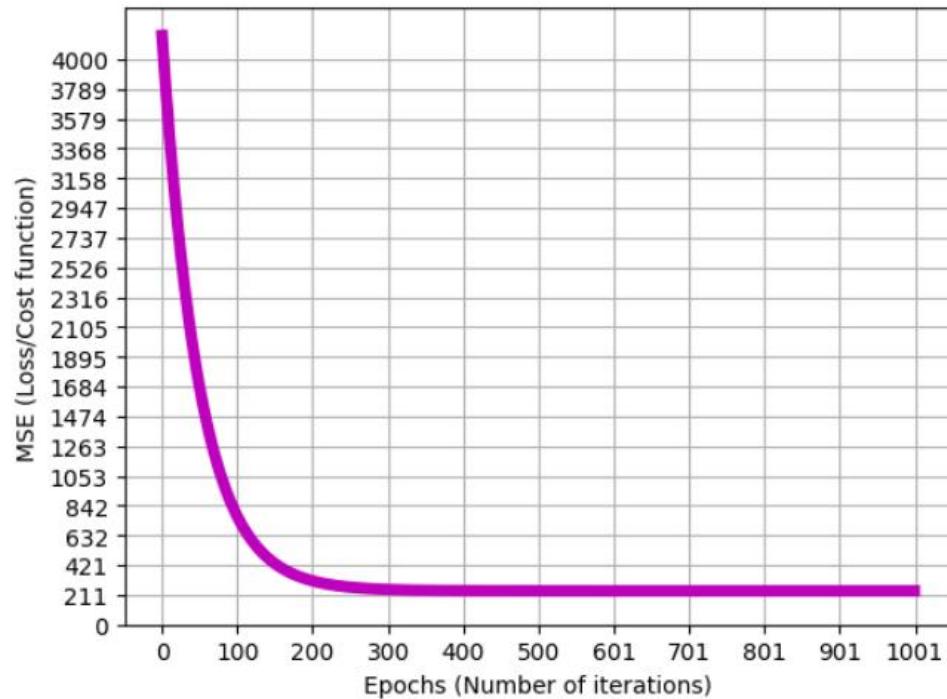
n_iters: 800, MSE LinRegNumPy: 239.13667398851751
Current value(s) for weight(s) in this iteration: 52.78204494908824

n_iters: 900, MSE LinRegNumPy: 239.13632086641604
Current value(s) for weight(s) in this iteration: 52.79278326375621

n_iters: 1000, MSE LinRegNumPy: 239.13627355523167
Current value(s) for weight(s) in this iteration: 52.79671383421545

```

Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.79673652805303

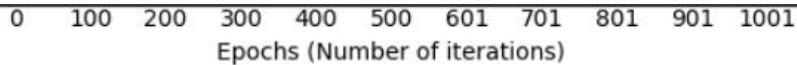


```

# Comparing parameter values of NumPy from scratch model vs Scikit-Learn's model after training:
# Comparison of parameter values set by training and the MSE scored by predictions on the Test set

comparison_11Decimals = {
    'LinearRegression()' sklearn ': [regressorSklearn.coef_.round(11), regressorSklearn.intercept_, sklearnModel_MSE_Numpy],

```



```

# Comparing parameter values of NumPy from scratch model vs Scikit-Learn's model after training:
# Comparison of parameter values set by training and the MSE scored by predictions on the Test set

comparison_11Decimals = {
    'LinearRegression() sklearn': [regressorSklearn.coef_.round(11), regressorSklearn.intercept_, sklearnModel_MSE_Numpy],
    'LinRegNumPy lr=0.1': [regressor.weights.round(11), regressor.bias, MSE_Numpy],
    'LinRegNumPy lr=0.01': [lowerLRegressor.weights.round(11), lowerLRegressor.bias, lowerLR_MSE_Numpy]
}

df = pd.DataFrame.from_dict(comparison_11Decimals, orient='index', columns=['Weight(s)/Coefficient(s) (param)', 'Bias/Intercept (param)', 'MSE (metric of preds on Test set)'])
pd.set_option("display.precision", 11)
display(df)

comparison_15Decimals = {
    'LinearRegression() sklearn': [[np.float64(regressorSklearn.coef_)], regressorSklearn.intercept_, sklearnModel_MSE_Numpy],
    'LinRegNumPy lr=0.1': [[np.float64(regressor.weights)], regressor.bias, MSE_Numpy],
    'LinRegNumPy lr=0.01': [[np.float64(lowerLRegressor.weights)], lowerLRegressor.bias, lowerLR_MSE_Numpy]
}

df = pd.DataFrame.from_dict(comparison_15Decimals, orient='index', columns=['Weight(s)/Coefficient(s) (param)', 'Bias/Intercept (param)', 'MSE (metric of preds on Test set)'])
pd.set_option("display.precision", 15)
display(df)

# Interpretation of results
# Note: after the current Kernel has been restarted the values I pasted here won't be exactly the same,
# but the relation (smaller/bigger) between them will be the same, so the interpretation is the same

# Comparing the two dataframes:
# The first df suggests that LinRegNumPy lr=0.1 model's parameter values are equal (or at least very close)
# to the values of the imported sklearn model, which validates the created NumPy class.
# On the second more precise df we can see that although the values are very close to each other,
# they're not exactly the same. This might be because sklearn uses not exactly the same methods
# for creating their LinearRegression() class

# Comparing different Learning rates:
# With a Lower Learning rate we get a better (smaller) Mean Squared Error score on the predictions,
# whereas with this Lower Learning rate we reached a worse Loss function value at the 1000.th iteration of
# Gradient Descent: 239.13627355841672. With a higher Learning rate already just at the 100.th iteration GradDesc
# reached a smaller loss function value (239.1362689603656)
# This could mean that with a higher Learning rate we overfitted (just a little bit, but still did) the model
# IN COMPARISON to training with lower learning rate, that might be why we get better prediction MSE score with Lower LR

```

	Weight(s)/Coefficient(s) (param)	Bias/Intercept (param)	MSE (metric of preds on Test set)
LinearRegression() sklearn	[52.79898321797]	34.11642774020	274.10601178077
LinRegNumPy lr=0.1	[52.79898321797]	34.11642774020	274.10601178077
LinRegNumPy lr=0.01	[52.79673652805]	34.11496961994	274.08152930247

	Weight(s)/Coefficient(s) (param)	Bias/Intercept (param)	MSE (metric of preds on Test set)
LinearRegression() sklearn	[52.79898321797334]	34.116427740196400	274.106011780774736
LinRegNumPy lr=0.1	[52.798983217973294]	34.116427740196372	274.106011780774281
LinRegNumPy lr=0.01	[52.79673652805303]	34.114969619941085	274.081529302467459

LinReg presentacio kerdesek docxbol (fájlok között) a matekos reszeket bemasolni a cellakba

further considerations cella a class létrehozás alá közvetlenül

reviewed notebookokbol a static fv-ek es egyeb dolgok (pl átnvezések) áthozása az eles notebookba

```
# Further considerations
```

```

# Further considerations

# Rename normalization to standardization and use mean and std of train set to standardize test set
# Use num_features inside 'fit' function for calling other _functions instead of calculating num_features inside the other _functions - didn't work?

# Proper plot for multiple features
# Create '_predict' func and use inside 'predict' and 'fit' instead of current approach - would be better or not?
# Interpretation of the results of the cell above - correct?

# for the LinRegNumPy class use static methods instead where possible, for example for def _GradDesc_updateWeightValues

# Put mathematical formulas into LaTeX form: $\sigma(x)$

# Add R2 to the metrics used

# Notes for myself
# Amikor lr 0.1 es modell MSE (metric of preds on Test set) kisebb, ezért lehet:
# a training alatt az 1000. iteracioban a 0.1 es modell MSEje (graddesces) kisebb mint a 0.01-esé,
# tehát a 0.1-esek optimalisabb paramtereket sikerült találnia a training alatt, így a test seten optimalisabbak a predíciók
# ha a jobb training ellenére a predíkciós MSE magasabb lenne (legutóbbi output esetben), az azt jelentené sztem h már túl optimalis/minimális értékét találták meg a lossnak (MSEnek) a training alatt, tehát olyan paraméter értékeket amik a training setre már túl jók (túltanítás)

```

Model extensions (Polynomial and Gaussian basis functions, Regularization techniques)

```

# Linear Regression with single feature data and with multidimensional data

# Data which is scattered about a line with a slope of 2 and an intercept of -5

rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)

plt.scatter(x, y)
plt.show()

print(x.shape, x.reshape(-1, 1).shape)

# Estimator to fit data and construct the best-fit line
model = LinearRegression(fit_intercept=True)
model.fit(x.reshape(-1, 1), y)

xfit = np.linspace(0, 10, 1000)
yfit = model.predict(xfit.reshape(-1, 1))

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.show()

# Parameters of the model
print("Model slope: ", model.coef_[0])
print("Model intercept: ", model.intercept_)
# they're not perfectly 2 and -5 because of the noise that we added to y: + rng.randn(50)

# Handling multidimensional data
rng = np.random.RandomState(1)
X = 10 * rng.rand(100, 3)
y = 0.5 + np.dot(X, [1.5, -2., 1.])
print()
print(X.shape, y.shape)
print(X[:, 2])
print(y[:, 2])

model.fit(X, y)

```

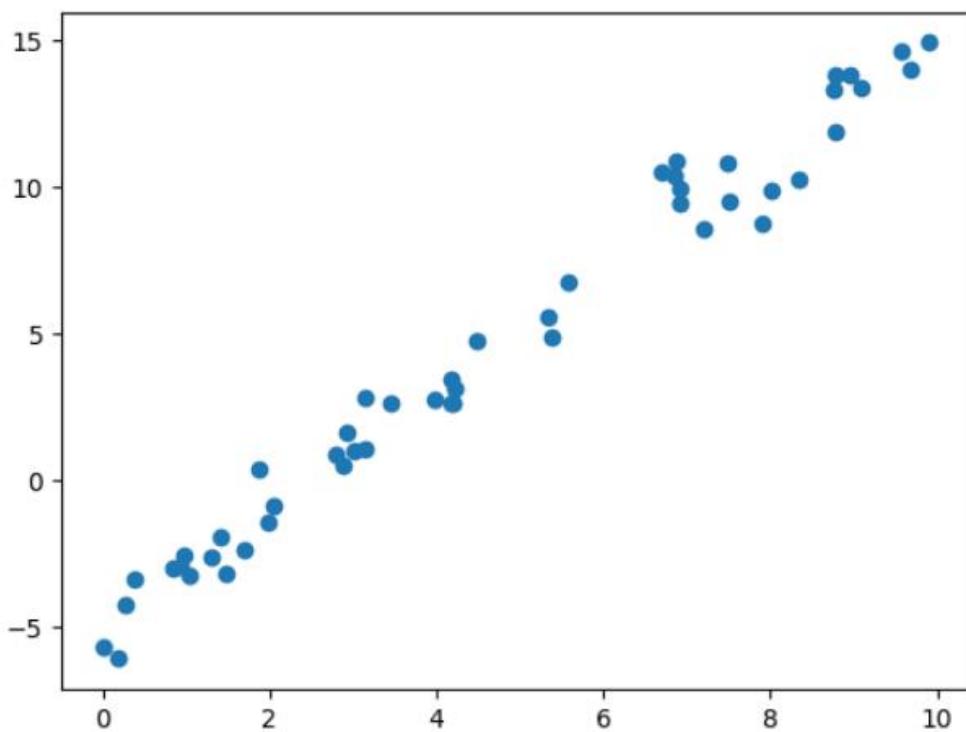
```

print(X.shape, y.shape)
print(X[:2])
print(y[:2])

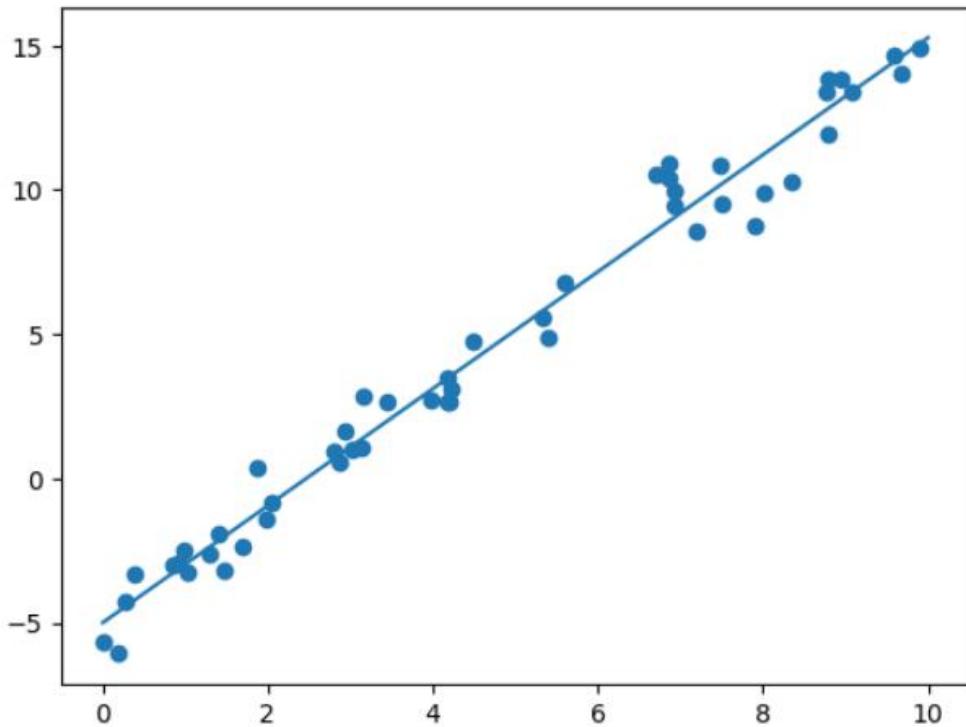
model.fit(X, y)
# Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a
# hyper-plane to points in higher dimensions
print()
print("Model weights/coefficients:    ", model.coef_)
print("Model intercept:", model.intercept_)

# In this way, we can use the single LinearRegression estimator to fit lines, planes, or hyperplanes to our data

```



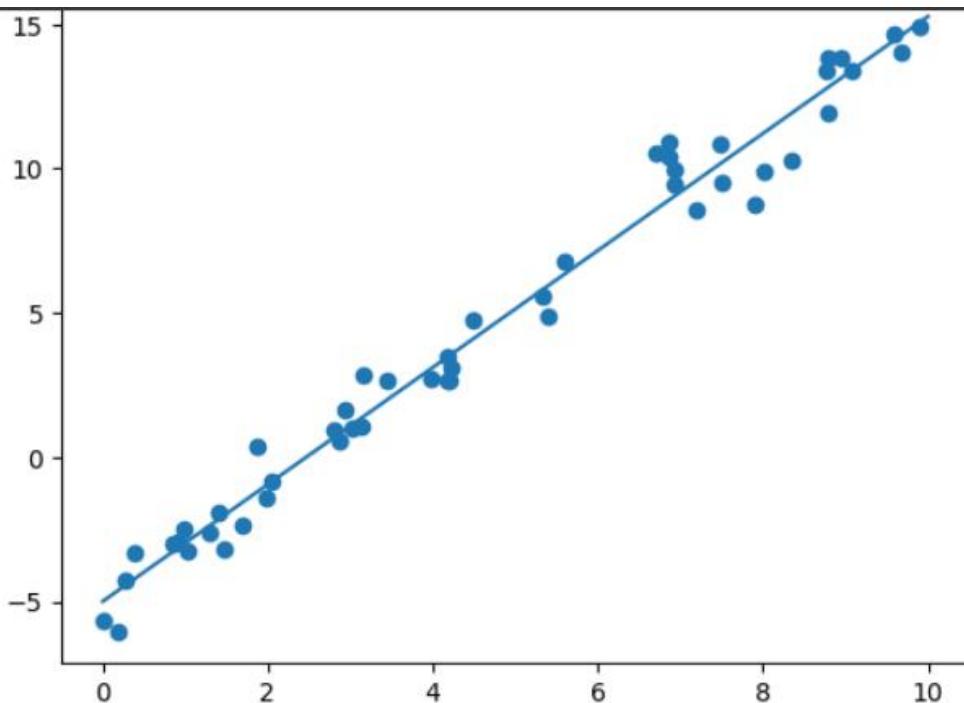
(50,) (50, 1)



Model slope: 2.0272088103606953

Model intercept: -4.998577085553204

(100, 3) (100,)
[[4.17022005e+00 7.20324493e+00 1.14374817e-03]
 1 167558910100 9 222859180_0111]



Model slope: 2.0272088103606953

Model intercept: -4.998577085553204

```
(100, 3) (100, )
[[4.17022005e+00 7.20324493e+00 1.14374817e-03]
 [3.02332573e+00 1.46755891e+00 9.23385948e-01]]
[-7.65001605 3.02325672]
```

Model weights/coefficients: [1.5 -2. 1.]

Model intercept: 0.5000000000000033

Basis Function Regression

One trick you can use to adapt linear regression to nonlinear relationships between variables is
to transform the data according to basis functions

With a single-dimensional input x:
We let $x_n = fn(x)$, where $fn()$ is some function that transforms our data

Polynomial basis functions

if $fn(x) = x^n$, our model becomes a polynomial regression:

$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$

This is still a linear model—the linearity refers to the fact that the coefficients are never multiply/divide each other.
We have projected our one-dimensional x values into a higher dimension,
so that a linear fit can fit more complicated relationships between x and y

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
```

```
# Higher-dimensional data representation
poly = PolynomialFeatures(degree = 3, include_bias=False)
print(poly.fit_transform(x.reshape(-1,1)))
```

```
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

With this transform in place, we can use the Linear model to fit much more complicated relationships between x

E.g. sine wave with noise:

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
```

```

poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
# With this transform in place, we can use the linear model to fit much more complicated relationships between x and y
# E.g. sine wave with noise:
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

poly_model.fit(x.reshape(-1,1), y)

xfit = np.linspace(0, 10, 1000)
yfit = poly_model.predict(xfit.reshape(-1,1))

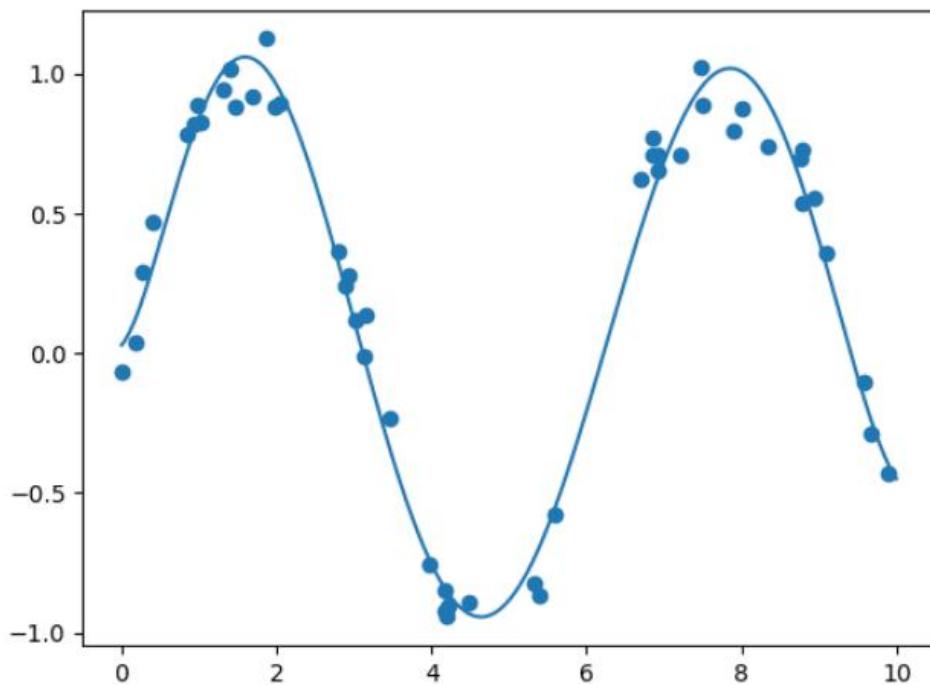
plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.show()

```

```

[[ 2.  4.  8.]
 [ 3.  9. 27.]
 [ 4. 16. 64.]]

```



```

# Gaussian basis functions

# Another basis function is for example to fit a model that is not a sum of polynomial bases,
# but a sum of Gaussian bases. The shaded regions in the plot are the scaled basis functions,
# and when added together they reproduce the smooth curve through the data

from sklearn.base import BaseEstimator, TransformerMixin

# BaseEstimator is a base class in the scikit-learn machine learning library that defines the basic
# interface for all estimators in scikit-learn. It provides a set of methods and attributes that
# any custom estimator needs to implement or inherit to work properly with the scikit-learn framework.
# This base class enables to set and get parameters of the estimator
# implements get_params and set_params for you. These two methods allow your custom transformer to be cloned
# and hyperparameter-tuned using GridSearchCV or RandomizedSearchCV, it also contains some common
# input validation methods, and more recently, feature name handling.

# TransformerMixin: mixin class for all transformers in scikit-learn

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

```

```

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # creating N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return __class__._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                      self.width_, axis=1)

rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
xfit = np.linspace(0, 10, 1000)

gauss_model = make_pipeline(GaussianFeatures(N = 10, width_factor = 1.0),
                             LinearRegression())

gauss_model.fit(x.reshape(-1,1), y)
yfit = gauss_model.predict(xfit.reshape(-1,1))

gf = gauss_model.named_steps['gaussianfeatures']
lm = gauss_model.named_steps['linearregression']

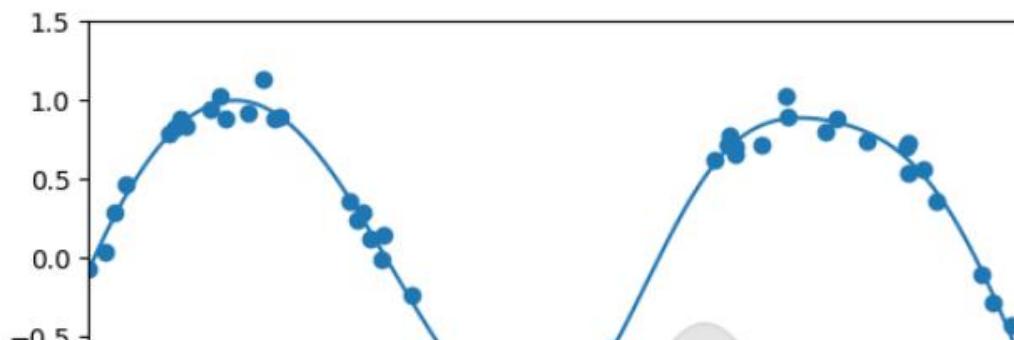
fig, ax = plt.subplots()

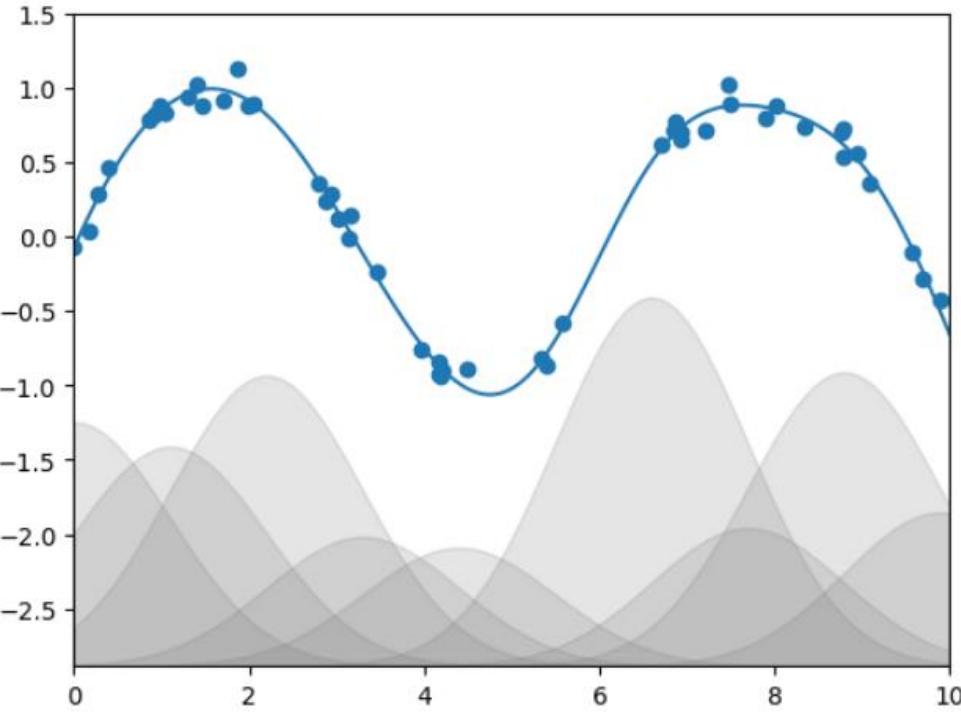
for i in range(10):
    selector = np.zeros(10)
    selector[i] = 1
    Xfit = gf.transform(xfit.reshape(-1,1)) * selector
    yfit = lm.predict(Xfit)
    ax.fill_between(xfit, yfit.min(), yfit, color='gray', alpha=0.2)

ax.scatter(x, y)
ax.plot(xfit, gauss_model.predict(xfit.reshape(-1,1)))
ax.set_xlim(0, 10)
ax.set_ylim(yfit.min(), 1.5)
plt.show()

print(lm.coef_)

```





```
[ 1.64110388  1.47667984  1.9534691   0.8702509   0.79767371 -0.08567809
 2.47809194  0.92998259  1.9757953   1.03547107]
```

```
# Regularization
```

```
# Increasing complexity of models can quickly lead to overfitting,
# for example too many Gaussian basis functions:
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
xfit = np.linspace(0, 10, 1000)

model = make_pipeline(GaussianFeatures(30),
                      LinearRegression())

model.fit(x.reshape(-1,1), y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit.reshape(-1,1)))

plt.xlim(0, 10)
plt.ylim(-5, 1.5)
plt.title("Gaussian basis functions with 30-dimensional basis")
plt.show()

# With the data projected to the 30-dimensional basis, the model has far too much flexibility
# We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:
        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))
```

```

# We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:
        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))
    #ax[1].set_yticks(np.linspace(round(model.steps[1][1].coef_.min(), 1), round(model.steps[1][1].coef_.max(), 1), 8))

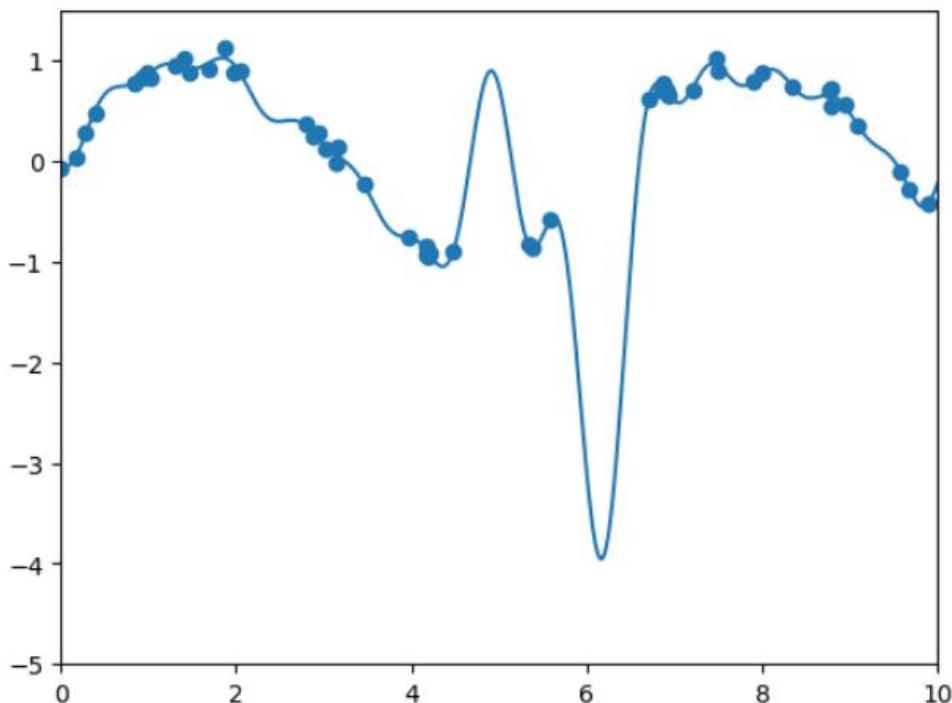
    plt.show()

model = make_pipeline(GaussianFeatures(30, width_factor = 2.0), LinearRegression())
basis_plot(model, title="Gaussian basis functions with 30-dimensional basis")

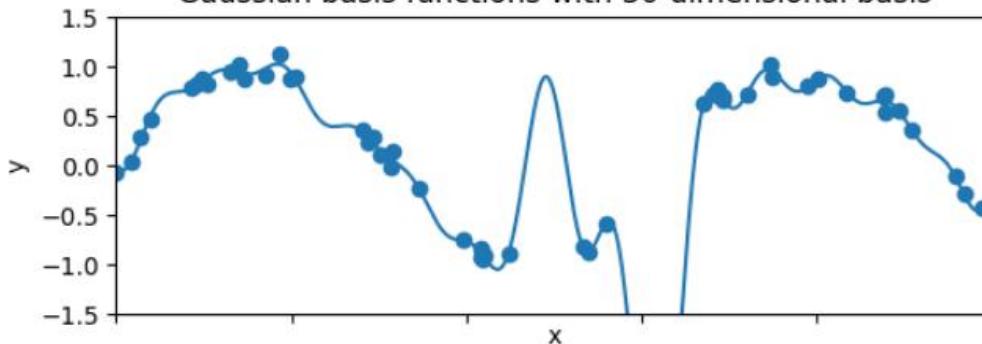
# On the plot we see a typical overfitting behavior: the basis functions overlap, the coefficients of adjacent
# basis functions blow up and cancel each other out.
# It would be nice if we could limit such spikes explicitly in the model by penalizing large values of
# the model parameters: Regularization

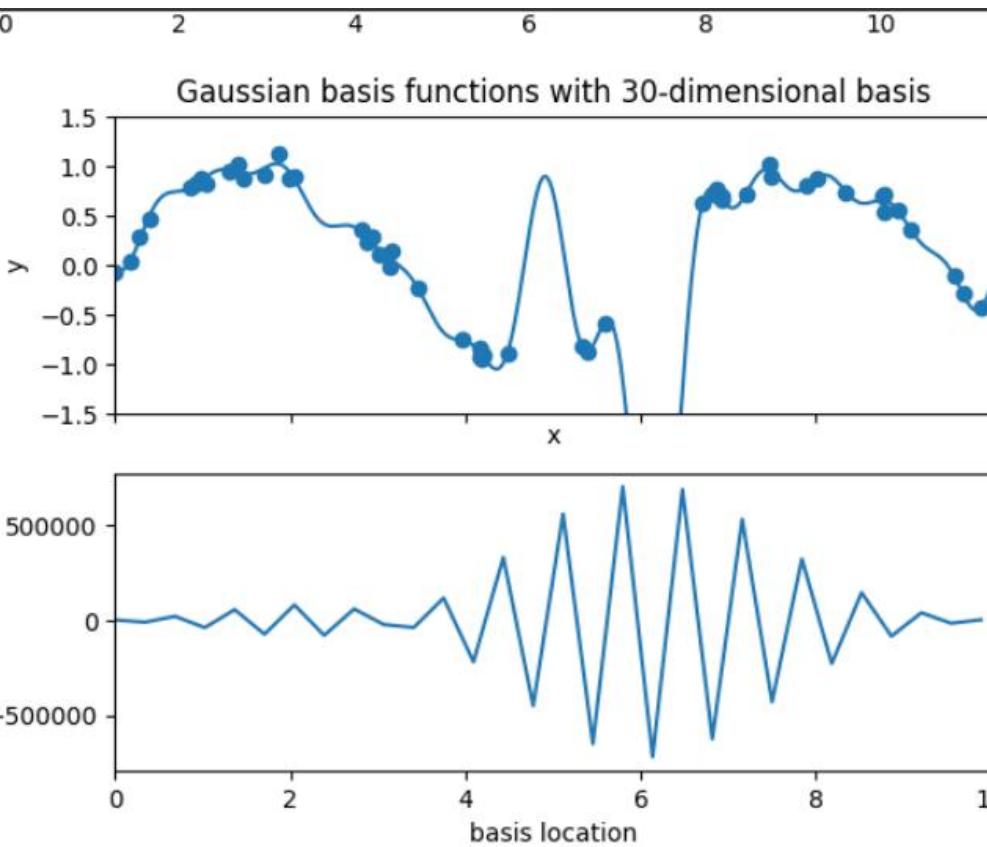
```

Gaussian basis functions with 30-dimensional basis



Gaussian basis functions with 30-dimensional basis





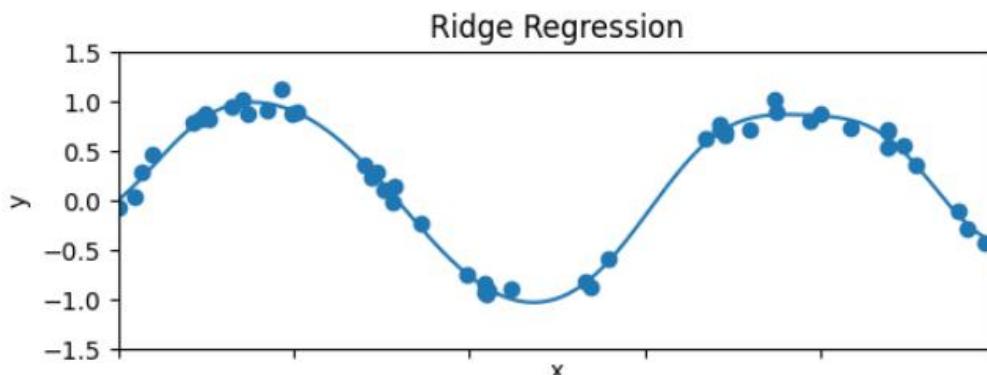
```
# Ridge regression (L2 Regularization)
# Sometimes also called Tikhonov regularization
# Penalizes the sum of squares (2-norms) of the model coefficients/weights

# Residual Sum of Squares in this case:
#  $\sum_{i=1}^N [(y_i - w^T \cdot x_i)^2] + \alpha * \sum_{j=1}^D [w_j^2]$ ,
# where  $\alpha$  is a free parameter that controls the strength of the penalty,
# D: number of dimensions,
# w0 = intercept/bias and x0_i = 1
# In this formula the first sum is to fit the training data well,
# and the second is to keep weights small (so to prevent overfitting)
# Thus there's a trade-off between these 2 goals:
# as we increase weights (increase overfitting) the prediction errors (on the training set) decrease, and vice-versa

from sklearn.linear_model import Ridge
RidgeModel = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
basis_plot(RidgeModel, title='Ridge Regression')

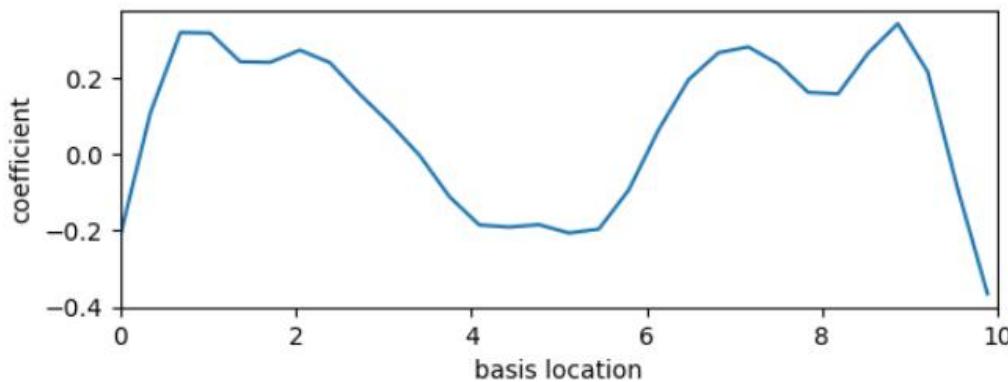
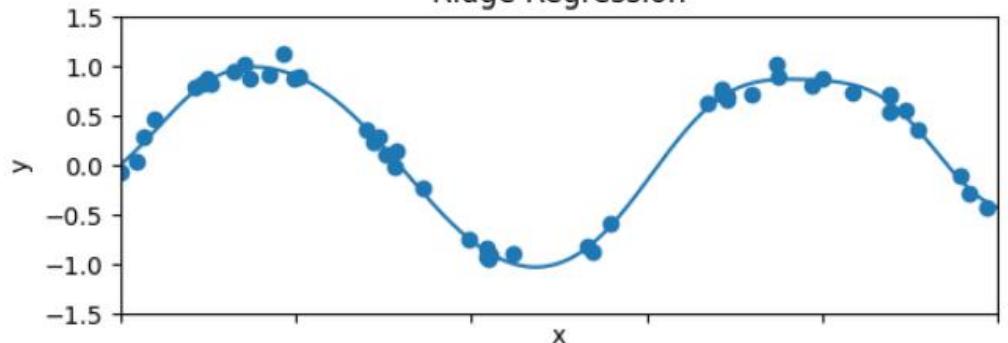
# The  $\alpha$  parameter is essentially a knob controlling the complexity of the resulting model
# In the limit  $\alpha \rightarrow 0$ , we recover the standard Linear regression result (the weights won't be penalized for any size)
# in the limit  $\alpha \rightarrow \infty$  all model responses will be suppressed (the penalty is so strong that  $w \rightarrow 0$ )

# One advantage of Ridge regression is that it can be computed very efficiently,
# at hardly more computational cost than the original linear regression model
```



```
# One advantage of Ridge regression is that it can be computed very efficiently,
# at hardly more computational cost than the original linear regression model
```

Ridge Regression



```
# Lasso regression (L1 regularization, Least absolute shrinkage and selection operator)
```

```
# Penalizes the sum of absolute values (1-norms) of the model coefficients/weights
```

```
# Residual Sum of Squares in this case:
```

```
#  $\sum_{i=1}^N [(y_i - w^T x_i)^2] + \alpha * \sum_{j=1}^D [|w_j|]$ ,
```

```
# where  $\alpha$  is a free parameter that controls the strength of the penalty,
```

```
# D: number of dimensions,
```

```
#  $w_0$  = intercept/bias and  $x_0_i = 1$ 
```

```
# In this formula the first sum is to fit the training data well,
```

```
# and the second is to keep weights small (so to prevent overfitting)
```

```
# Thus there's a trade-off between these 2 goals:
```

```
# as we increase weights (increase overfitting) the prediction errors (on the training set) decrease, and vice-versa
```

```
from sklearn.linear_model import Lasso
```

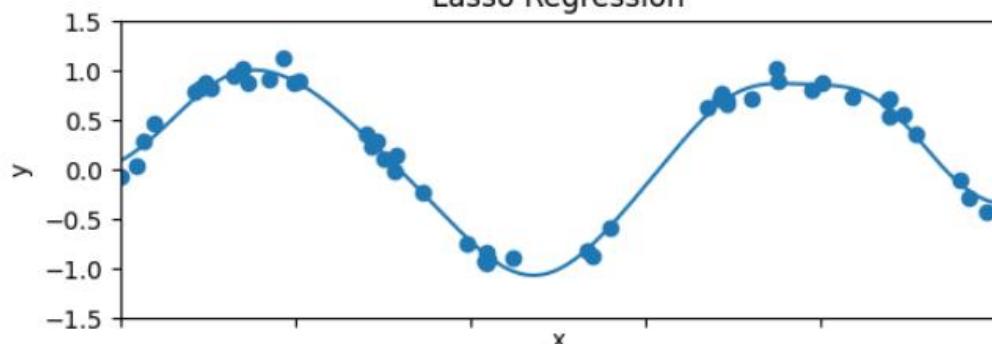
```
LassoModel = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.002))
```

```
basis_plot(LassoModel, title='Lasso Regression')
```

```
# One advantage of Lasso regression is that it is able to perform 'embedded feature selection'
```

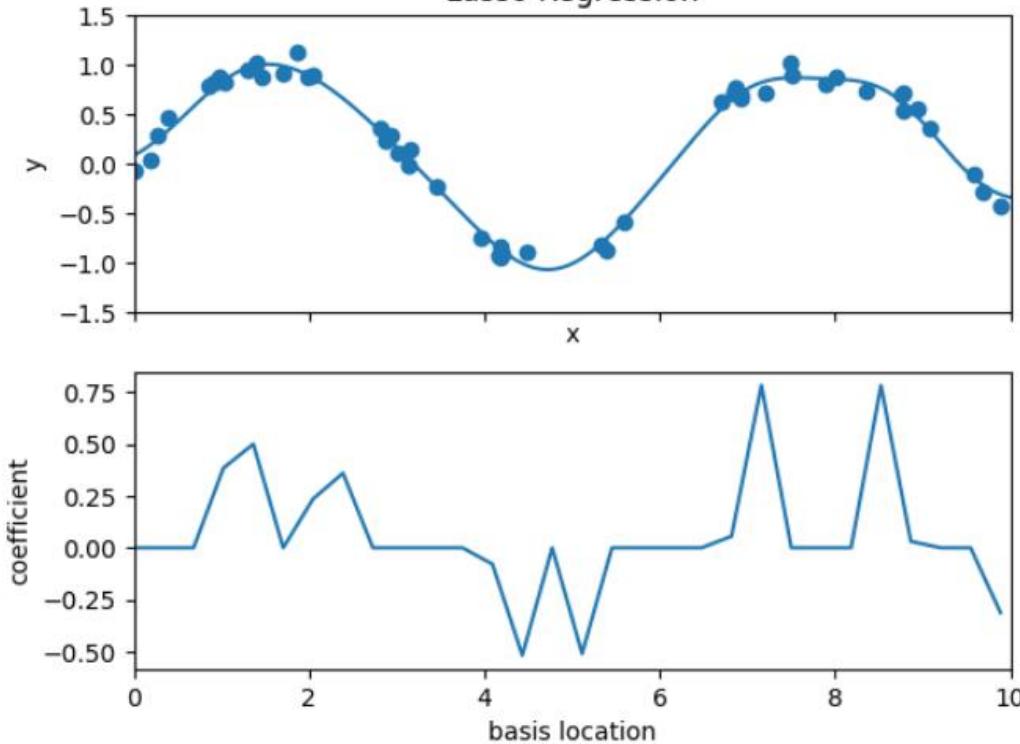
```
# by setting coefficients to exactly zero (Ridge can only set them close to zero)
```

Lasso Regression



" by setting coefficients to exactly zero (Ridge can only set them close to zero)

Lasso Regression



```
# Lasso vs. Ridge Regression
```

```
# Unlike Ridge, due to geometric reasons (diamond vs circle problem) Lasso is able to perform
# 'embedded feature selection' by setting coefficients to exactly zero, while Ridge can only
# shrink them close to zero, but never equal to zero. So Lasso tends to favor sparse models where possible.

# On the plot above we can see that the majority of the coefficients are exactly zero,
# so only a small subset of the basis functions are 'available'/'active'
print(LassoModel.steps[1][1].coef_)
print()
print(RidgeModel.steps[1][1].coef_)

# Problem with the 'automated feature selection' of Lasso:
# When we have correlated features/variables, it usually retains only one of feature/variable
# which is found to be the most important, and sets other correlated variables to zero,
# so sometimes it cancels too many features and thus increases bias too much (error by simplifying assumptions of the

# One 'solution' to this problem:
# Elastic Net Regression, which combines both Ridge and Lasso Regression techniques
# by learning from their shortcomings to improve regularization
# So the penalty term is a combination of the l1-norm (absolute value) and the l2-norm (square) of the coefficients,
# scaled by a parameter called alpha.
# Like Lasso, Elastic Net can also generate reduced models by zero-valued coefficients,
# but it can outperform Lasso on data with highly correlated predictors/variables/features
# Can be found here: sklearn.linear_model.ElasticNet
```

```
[-0.          -0.          0.          0.38496444  0.500335   0.
 0.23609036  0.35994884  0.          0.          -0.          -0.
 -0.07887826 -0.5184722   -0.         -0.51073554  -0.          -0.
 0.          0.          0.05560327  0.78410977  0.          0.
 0.          0.78318771  0.0308607   0.          -0.         -0.31307227]
```

```
[-0.21105794  0.10862732  0.31824548  0.31676161  0.2418113   0.24003021
 0.27210133  0.23947825  0.15675484  0.08094844  -0.00307864 -0.11134067
 -0.18560261 -0.19090305 -0.18482812 -0.20657632 -0.19638464 -0.09387141
 0.06369053  0.19454448  0.26527119  0.28056559  0.23593141  0.16159984
 0.15753808  0.26372747  0.34134295  0.21541902 -0.09008851 -0.36548691]
```

```
# A bit off-topic:
```

```

# A bit off-topic:
#Cell to show how prone to outliers Linear Regression is

x = np.array([1, 2, 3, 4, 5, 6, 7])
y = np.array([2, 4, 6, 8, 10, 12, 14])

model = LinearRegression().fit(x.reshape(-1,1), y)
yfit = model.predict(x.reshape(-1,1))

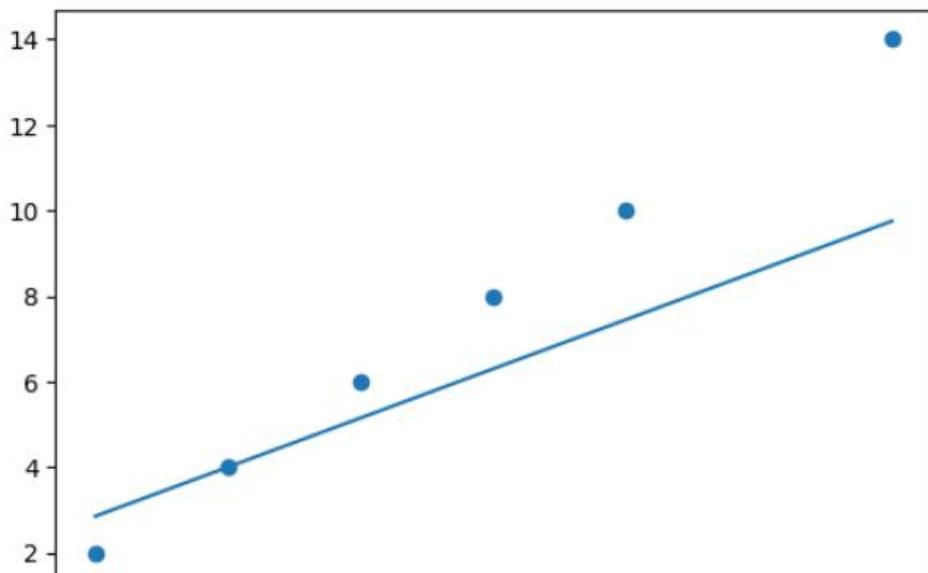
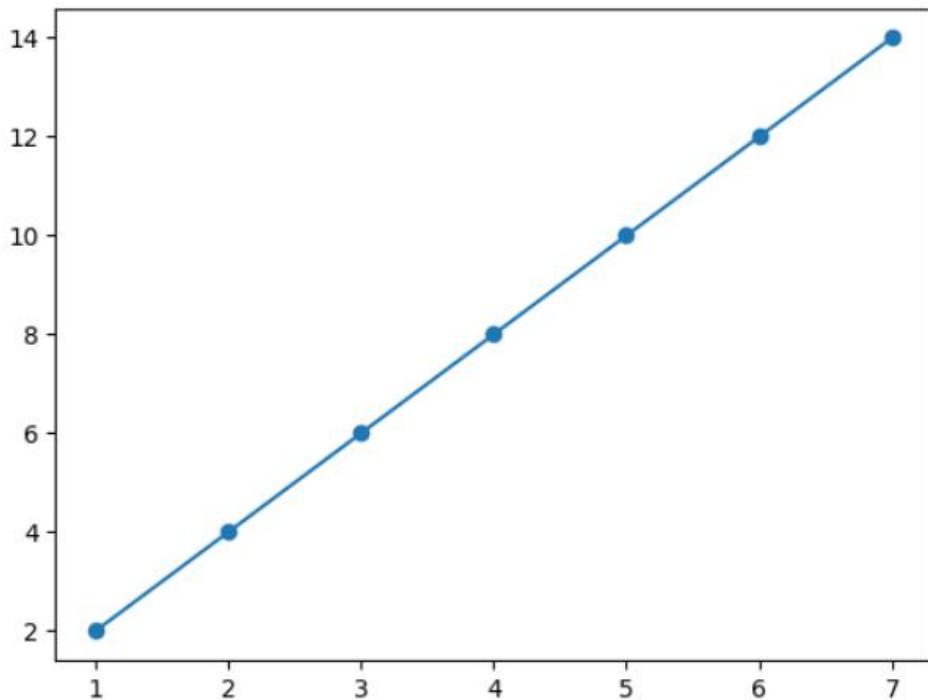
plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()

x = np.array([1, 2, 3, 4, 5, 6, 7])
y = np.array([2, 4, 6, 8, 10, 0.1, 14])

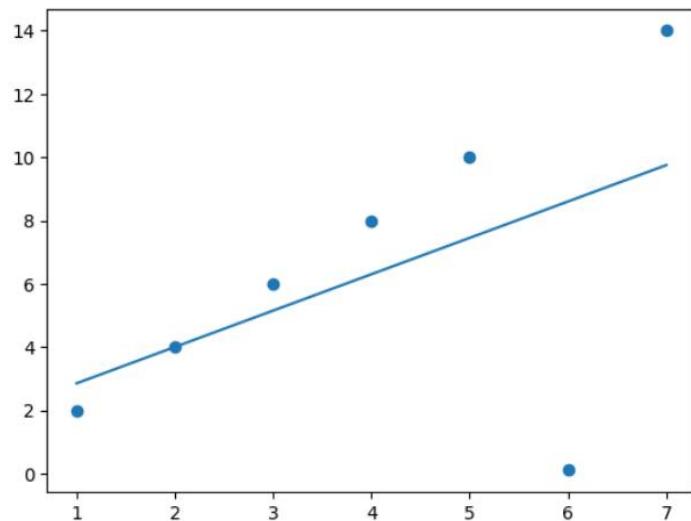
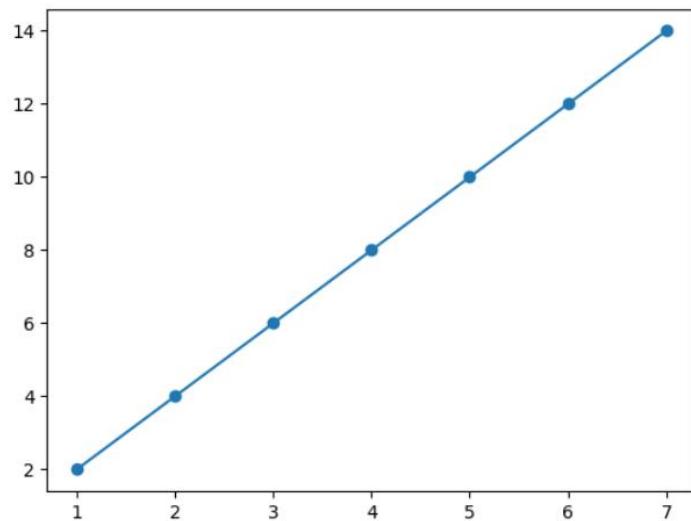
model = LinearRegression().fit(x.reshape(-1,1), y)
yfit = model.predict(x.reshape(-1,1))

plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()

```



```
plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()
```



Logistic Regression

Mathematics

Introduction

Logistic Regression is a widely-used method in Machine Learning for binary classification, where the goal is to categorize data into one of two classes. It falls under the umbrella of Supervised Learning, requiring labeled training data.

The Logistic Model

Logistic Regression

Mathematics

Introduction

Logistic Regression is a widely-used method in Machine Learning for binary classification, where the goal is to categorize data into one of two classes. It falls under the umbrella of Supervised Learning, requiring labeled training data.

The Logistic Model

1. Linear equation: Calculating the distance

$$f(x) = \sum_{j=1}^D w_j x_j$$

Explanation:

- Calculates a signed distance between the input and the linear model (a linear line as the decision boundary between the 2 classes)
- Returns positive value if input belongs to the class on the right side of the linear decision boundary, and negative if input belongs to the class on the left side of the linear decision boundary
- The larger the distance of the input to the decision boundary, the more certain it belongs to either class 1 or class 2

2. Sigmoid Function

The key component in Logistic Regression is the Sigmoid function, denoted as σ , mapping any real number to a value between 0 and 1. It is expressed as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Explanation:

- The Sigmoid function maps any real number to a value between 0 and 1.
- It is crucial in logistic regression as it transforms the output of the linear equation into probabilities.

Why Sigmoid?

The Sigmoid function is chosen for logistic regression due to its properties:

- Maps to values between 0 and 1, ideal for binary classification probabilities.
- Smooth and differentiable, facilitating optimization using methods like gradient descent.
- Simple mathematical form for efficient computation

Why Sigmoid?

The Sigmoid function is chosen for logistic regression due to its properties:

- Maps to values between 0 and 1, ideal for binary classification probabilities.
- Smooth and differentiable, facilitating optimization using methods like gradient descent.
- Simple mathematical form for efficient computation.
- Symmetric around the origin, ensuring model stability and interpretability.

Alternative Functions

For logistic regression, other functions can replace the Sigmoid:

- **Softmax Function:** Generalization for multi-class classification.
- **Probit Function:** Based on the cumulative distribution function of the standard normal distribution.
- **Hyperbolic Tangent Function (tanh):** Maps to values between -1 and 1.

3. From distance to probability: Inserting the function $f(x)$ into Sigmoid

$$\mu(x, w) = \frac{1}{1 + e^{-w^T x}}$$

Explanation:

- This is the model now, calculates probabilities instead of distances
- Returns the probability of input x belonging only to the positive class.

Modelling both events: Bernoulli distribution

$$p(y|x, w) = Ber(y|x, w) = \mu(x, w)^y (1 - \mu(x, w))^{1-y}$$

Explanation:

- Returns the probability of input x belonging to either the positive class or negative class (based on y).

Optimization

Likelihood function

The objective function: Maximum Likelihood Estimation

Logistic Regression employs Maximum Likelihood Estimation (MLE) to find the optimal weights. MLE is a statistical method used to estimate the parameters of a model that maximize the likelihood of observing our data given a certain parametrization (w) of the observed data under the given model.

Likelihood function

The objective function: Maximum Likelihood Estimation

Logistic Regression employs Maximum Likelihood Estimation (MLE) to find the optimal weights. MLE is a statistical method used to estimate the parameters of a model that maximize the likelihood of observing our data given a certain parametrization (w) of the observed data under the given model.

$$L(w|X, y) = \prod_{i=1}^N p(y_i = 1|x_i, w)^{y_i} (1 - p(y_i = 1|x_i, w))^{1-y_i}$$

With Bernoulli

$$L(\theta) = \prod_{i=1}^N \mu(x_i, w)^{y_i} (1 - \mu(x_i, w))^{1-y_i}$$

Explanation:

- $L(w|X, y)$ is the likelihood function representing the joint probability of observed labels y given input data X and model parameters w .
- It is used in Maximum Likelihood Estimation (MLE) to find the set of parameters that maximize the likelihood of observing our data given a certain parametrization (w) of the observed data under the given model.

Logarithmic Likelihood/Log-Likelihood

$$\log L(w|X, y) = \sum_{i=1}^N y_i \log(p(y_i = 1|x_i, w)) + (1 - y_i) \log(1 - p(y_i = 1|x_i, w))$$

With Bernoulli

$$\begin{aligned} \ell(\theta) &= \sum_{i=1}^N \log[p(y_i|x_i, \theta)] \\ &= \sum_{i=1}^N \log(\mu(x_i, w)^{y_i} (1 - \mu(x_i, w))^{1-y_i}) \\ &= \sum_{i=1}^N y_i \log(\mu(x_i, w)) + (1 - y_i) \log(1 - \mu(x_i, w)) \end{aligned}$$

Why to take the logarithm of the Likelihood function?

Some of the main reasons:

- The product of small numbers in a computer is unstable, for example when $w^T x \rightarrow -\infty \Rightarrow e^{-w^T x} \rightarrow +\infty \Rightarrow \frac{1}{1+e^{-w^T x}} \rightarrow 0$, which means

Why to take the logarithm of the Likelihood function?

Some of the main reasons:

- The product of small numbers in a computer is unstable, for example when $w^T x \rightarrow -\infty \Rightarrow e^{-w^T x} \rightarrow +\infty \Rightarrow \frac{1}{1+e^{-w^T x}} \rightarrow 0$, which means $p(y=1|x, w)^y \rightarrow 0$ and then this factor (factor for this (x_i, y_i)) in the product might turn to exactly 0 (instead of very close to zero), which results in the whole product turning to 0 (compare `print(1e-323)` vs. `print(1e-324)`)
- Often times the logarithm of a non-concave Likelihood function is concave (concave down, and monotonically increasing in x), which is a change from multiple local maximums to one unique (global) maximum (after this it's much easier to optimize i.e. find the global optimum)
- Easier to take the derivative of the log of the function than the function itself as the log of a product is the sum of the logs of its factors

Loss function (Binary Cross Entropy)

Instead of Maximizing the Log-Likelihood, we minimize the Negative Log-Likelihood: Binary Cross Entropy Loss

$$J(w) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p(x_i, w)) + (1 - y_i) \log(1 - p(x_i, w))$$

With Bernoulli

$$J(\theta) = -\sum_{i=1}^N y_i \log(\mu(x_i, w)) + (1 - y_i) \log(1 - \mu(x_i, w))$$

Explanation:

- $J(w) = J(\theta)$ is the objective function (Loss function) to be minimized during training
- N is the number of training samples, y_i is the actual label (0 or 1), $p(x_i, w)$ is the predicted probability of x_i belonging to the positive class, and w is the vector of weights
- Measures the difference between values for predicted probability and true values (true labels): the logarithmic terms penalize a lot more when the predicted probability is far away from the true value compared to when it's closer (e.g. compare the Loss value difference ($y_i = 0$ or 1) between $p(x_i, w) = \mu(x_i, w) = 0.8$ and $p(x_i, w) = \mu(x_i, w) = 0.9$ with the difference between $p(x_i, w) = \mu(x_i, w) = 0.1$ and $p(x_i, w) = \mu(x_i, w) = 0.2$)
- The model's objective is to find optimal weights w that minimize $J(w) = J(\theta)$.
- It has a unique minimum, thus can be optimized with for example Gradient Descent

Training Process

Repeat for each training iteration:

1. Get predictions with current weights

So that we are able to compute the next step

(the derivative formula needs predictions as input)

$$\hat{y}_i = p(x_i, w)$$

Training Process

Repeat for each training iteration:

1. Get predictions with current weights

So that we are able to compute the next step
(the derivative formula needs predictions as input)

$$\hat{y}_i = p(x_i, w)$$

2. Compute gradient (derivative) of the Loss Function

So that in the next step we are able to update the weights

We use Gradient Descent as the optimization method to minimize the Loss Function

The gradient of the Loss is computed as:

$$\frac{\partial J(w)}{\partial w_j} = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i) x_{ij}$$

Explanation:

- Calculates the gradient (derivative) of the Loss Function with respect to the weights.
- Visually the Loss looks like a mountainous landscape, we find the minimum by “walking down” the slope of the mountain
- The derivative (gradient) represents the direction and magnitude of the steepest ascent/descent (depends on initial weights) of the corresponding function (now the Loss Function).

3. Update weights

So that with the updated parameters predictions produce a smaller Loss

$$w_j = w_j - \alpha \frac{\partial J(w)}{\partial w_j}$$

Explanation:

- Updates the weights in the direction of the minimum of the Loss Function/in the direction that reduces the Loss
- The learning rate α controls (scales) the size of the update

Regularization

Large values of the weights (parameters) make the model follow small changes in the input too closely in the output (overfitting)
We can control overfitting explicitly in the model by penalizing these large weights: Regularization

Regularization

Large values of the weights (parameters) make the model follow small changes in the input too closely in the output (overfitting). We can control overfitting explicitly in the model by penalizing these large weights: Regularization.

L1 Regularization

Penalizes the sum of absolute values (1-norms) of the model weights.

Modifies the Loss Function (more precisely the Likelihood Function, from which the Loss is derived) by adding the penalty term $\lambda \sum_{j=1}^n |w_j|$

Loss Function with L1 Regularization:

$$J(w) = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p(x_i, w)) + (1 - y_i) \log(1 - p(x_i, w))] + \lambda \sum_{j=1}^n |w_j|$$

Explanation:

- λ is a free parameter that controls the strength of the regularization (of the penalty)
- The first \sum is to fit the training data well, and the second is to keep weights small (to prevent overfitting)
- So there's a trade-off between these 2 goals: as we increase weights (i.e. increase overfitting, so the right \sum increases), the prediction errors on the training set decrease (so the negative of the left \sum decreases), and vice-versa
- In the limit $\lambda \rightarrow 0$ we recover the standard Logistic Regression without penalty (weights won't be penalized for any size)
- In the limit $\lambda \rightarrow \infty$ all model responses will be suppressed (the penalty is so strong that $w \rightarrow 0$)

L2 Regularization

Penalizes the sum of squares (2-norms) of the model weights. Modifies the Loss Function (more precisely the Likelihood Function, from which the Loss is derived) by adding the penalty term $\lambda \sum_{j=1}^n w_j^2$

Loss Function with L2 Regularization:

$$J(w) = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p(x_i, w)) + (1 - y_i) \log(1 - p(x_i, w))] + \lambda \sum_{j=1}^n w_j^2$$

Comparing L1 vs. L2 techniques for regularization

Due to geometric reasons (diamond vs circle problem), L1 can perform 'embedded feature selection' by being able to drive some weights to exactly zero, while L2 can only shrink them close to zero, but never equal to zero.

- Thus L1 penalty term encourages sparsity in the model and is often used when feature selection is desirable

Problem with the 'automated feature selection' of L1:

Problem with the 'automated feature selection' of L1:

When we have correlated features/variables, it usually retains only one of feature/variable which is found to be the most important, and sets other correlated variables to zero

- Thus sometimes L1 cancels too many features and therefore increases bias too much (error induced by simplifying assumptions of the model)

One suggestion to this problem:

- L2 technique discourages the model from relying too much on any particular feature

It can not perform feature selection because of not being able to set weights to exactly zero (only to shrink close to zero), due to the same geometric reasons

Another suggestion:

- Elastic Net Regression, which combines both L1 and L2 techniques by learning from their shortcomings to improve regularization

For Elastic Net the penalty term is a combination of the l1-norm (absolute value) and the l2-norm (square) of the coefficients scaled by a parameter.

Like L1, it can also generate reduced models by zero-valued weights, but empirical studies have suggested that it can outperform L1 on data with highly correlated features.

Creating model from scratch using NumPy (code structure in line with Mathematics) [...]

Applying created model, analyzing the training process and comparing performance to sklearn's model [...]

Evaluation metrics

Creating and applying class to a classification task

Creating model from scratch using NumPy (code structure in line with Mathematics)

```
# Get sklearn model's default parameter values
regressorSklearn = sklearn.linear_model.LogisticRegression()
DEFAULTSKPARAMS = regressorSklearn.__dict__


class LogRegNumPy:

    def __init__(self,
                 learning_rate = 0.1,
                 penalty = DEFAULTSKPARAMS['penalty'], #='L2', the technique to use for regularization
                 lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0, the  $\lambda$  (Lambda) a free parameter that controls the
                                                 #strength of the regularization (of the penalty)
                                                 # the 'C' parameter in sklearn's LogReg class is the inverse of this parameter)
                 max_iter = DEFAULTSKPARAMS['max_iter'], #=100, number of iterations for the training process (for Gradient Descent)
                 tol = DEFAULTSKPARAMS['tol'], #=0.0001=1e-4, when the Loss Function decreases so slightly that
                                                 # the difference between the Loss in the current iteration and in the previous
                                                 # iteration is below this tolerance boundary, then we can say the Loss Function
                                                 # has converged into its minimum and there's no need to take more gradient steps
                 patience=20, # Stop training iterations after this number (patience) of such iterations in which the
                             # Loss was not lower than the lowest Loss that Gradient Descent achieved so far in this training
                 threshold=0.5, # Cut-off value for the predicted probabilities (of  $x_i$  belonging to the positive class)
                               # based on which value  $x_i$  classified into the positive or negative class
                 initialweights_lowerlimit = 0, # Initialize weights for the first training iteration from a uniformly
                 initialweights_upperlimit = 1): # distributed interval [initialweights_lowerlimit, initialweights_upperlimit]
                                                 # Useful for fixed results (not different results when cell rerun) for comparison,
                                                 # or to check performance of the model without training

        self.learning_rate = learning_rate
        self.penalty = penalty
        self.lambda_or_inverseC = lambda_or_inverseC
        self.max_iter = max_iter
        self.tol = tol
        self.patience = patience
        self.threshold = threshold
        self.initialweights_lowerlimit = initialweights_lowerlimit
        self.initialweights_upperlimit = initialweights_upperlimit
        self.loss_history = [] # Collector for the Loss value in each iteration
        self.weights_history = [] # Collector for the weight(s) in each iteration
        self.weights = None # Initializing values to weights later during fit
        self.X_train_means = [] # For saving during fit() the mean and standard deviation of the train set
        self.X_train_stds = [] # with respect to each feature in order to be able to standardize the test set with them during predict()

    # ----- Standardize the input features -----
    @staticmethod
    def _standardizeX(X, trainX_means, trainX_stds):

        for feature in range(X.shape[1]):
            X[:, feature] = ((X[:, feature] - trainX_means[feature])/trainX_stds[feature])

        return X

    # ----- The Logistic Model -----
    # 1. Calculate distance
    @staticmethod
    def _calculate_distance(weights, X):
        return np.dot(weights, X.T) # transpose of X instead of weights as in the Mathematics

    @staticmethod
    # 2. Sigmoid Function
    def _sigmoid_function(z):
        return 1 / (1 + np.exp(-z))

    @staticmethod
```

```

@staticmethod
# 3. From distance to probability
def _predict_probability(wTx_distance):
    return __class__._sigmoid_function(wTx_distance)

# Computing Loss is not part of the core training process (3 steps) and is not required for training,
# here it is computed only to report (print) and visualize Loss values during training
@staticmethod
def _compute_loss(num_samples, y_pred, y_true, weights, regularization_technique, regularization_strength):

    # Binary Cross Entropy as the Loss Function
    cross_entropy = - (1 / num_samples) * np.sum(y_true*np.log(y_pred) + (1-y_true)*np.log(1-y_pred))
    # Same results:
    # epsilon = 1e-5
    # cross_entropy = (-y_true * np.log(y_pred + epsilon) - (1 - y_true) * np.log(1 - y_pred + epsilon)).mean()

    # The penalty term depends on the Regularization technique
    if regularization_technique == 'l1':
        penalty_term = regularization_strength * np.sum(np.abs(weights))

    elif regularization_technique == 'l2':
        penalty_term = regularization_strength * np.sum(np.square(weights))

    return cross_entropy + penalty_term

@staticmethod
def _visualize_graddesc(loss_values, weight_values, lowest_loss, number_of_dimensions):

    if number_of_dimensions == 1:
        fig, ax = plt.subplots(nrows=1, ncols=2, sharey=True)
        fig.set_figheight(4)
        fig.set_figwidth(15)
        ax[0].grid()
        ax[0].set_xlabel('Epochs (Number of iterations)', ylabel='Loss/Cost function')
        ax[0].plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
        ax[0].axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "1.5")
        ax[0].set_xticks = np.linspace(0, len(loss_values), 11)

        ax[1].grid()
        ax[1].set_xlabel('Weight')
        ax[1].axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "1.5")
        ax[1].set_xticks = np.linspace(weight_values.min(), weight_values.max(), 15)

        ax[1].scatter(weight_values, loss_values)
        for iterator in range(len(weight_values[:10])-1):
            ax[1].arrow(weight_values[iterator],
                        loss_values[iterator],
                        weight_values[iterator+1]-weight_values[iterator],
                        loss_values[iterator+1]-loss_values[iterator],
                        color = 'm', shape='full', width = 0.0003, head_width = 0.003, length_includes_head=True)

        # ax[1].plot(range(1,len(weight_values)+1), weight_values, 'm', linewidth = "3")
        # loss_values = np.array(loss_values)
        # weight_values = np.array(weight_values)
        # SortedIndexes = np.argsort(weight_values)
        # ax[1].plot(weight_values[SortedListes], loss_values[SortedListes], 'm', linewidth = "3")

    plt.show()

    else:
        plt.grid()
        plt.xlabel('Epochs (Number of iterations)')
        plt.ylabel('Loss/Cost function')
        plt.plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
        plt.axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "2")
        plt.xticks(np.linspace(0, len(loss_values), 11))
        plt.show()

```

```

else:
    plt.grid()
    plt.xlabel('Epochs (Number of iterations)')
    plt.ylabel('Loss/Cost function')
    plt.plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
    plt.axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "2")
    plt.xticks(np.linspace(0, len(loss_values), 11))
    plt.show()

# ----- Functions for the training process -----
@staticmethod
def _compute_gradient(X, y_pred, y_true, weights, regularization_technique, regularization_strength):
    num_samples = X.shape[0]

    # Derivative of the Cross Entropy Loss Function
    loss_derivative = 1 / num_samples * np.dot(X.T, y_pred - y_true)

    # Derivative of the penalty term
    if regularization_technique == 'l1':
        penalty_derivative = regularization_strength * np.sign(weights)
        # np.sign: returns -1 if x < 0, 0 if x==0, 1 if x > 0 as the derivative of |x| = x/|x| = sign(x)

    elif regularization_technique == 'l2':
        penalty_derivative = 2 * regularization_strength * weights

    return loss_derivative + penalty_derivative

@staticmethod
def _update_weights(gradient, weights, regularization_technique, learning_rate):

    if regularization_technique == 'l1':
        updated_weights = weights - learning_rate * gradient

    elif regularization_technique == 'l2':
        updated_weights = weights - learning_rate * gradient

    return updated_weights

# ----- Main functions of the class: fit and predict -----
def fit(self, X, y, training_report = False):

    num_samples, num_features = X.shape

    for feature in range(num_features):
        self.X_train_means.append(np.mean(X[:, feature]))
        self.X_train_stds.append(np.std(X[:, feature]))
    X = __class__._standardizeX(X, self.X_train_means, self.X_train_stds)
    # using np.mean(X, axis=0) and np.std(X, axis=0) to get the mean and the std for each feature
    # computes less accurate results, for details please see the last cell about comparing
    # the two methods in this heading: Linear Regression - Creating from scratch

    self.weights = np.random.uniform(self.initialweights_lowerlimit, self.initialweights_upperlimit, num_features)

    weights_of_lowest_loss = None
    lowest_loss = float('inf')
    loss_not_lower_than_lowest_counter = 0
    previous_loss = None
    number_of_iteration = 0

    # ----- Training process -----
    for number_of_iteration in range(1, self.max_iter+1):

        # 1. Get predictions with current weights
        distances = __class__._calculate_distance(self.weights, X)
        y_pred = __class__._predict_probability(distances)

        if training_report:
            plt.grid()
            plt.xlabel('Epochs (Number of iterations)')
            plt.ylabel('Loss/Cost function')
            plt.plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
            plt.axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "2")
            plt.xticks(np.linspace(0, len(loss_values), 11))
            plt.show()

        if previous_loss is not None and previous_loss >= lowest_loss:
            loss_not_lower_than_lowest_counter += 1
        else:
            loss_not_lower_than_lowest_counter = 0
        if loss_not_lower_than_lowest_counter >= self.convergence_threshold:
            break

        previous_loss = lowest_loss
        lowest_loss = None
        for i in range(len(loss_values)):
            if loss_values[i] < lowest_loss or lowest_loss is None:
                lowest_loss = loss_values[i]
        if lowest_loss <= self.convergence_threshold:
            break

```

```

    for number_of_iteration in range(1, self.max_iter+1):

        # 1. Get predictions with current weights
        distances = __class__._calculate_distance(self.weights, X)
        y_pred = __class__._predict_probability(distances)

# ----- Functionalities outside of the core training (3 steps) process -----
        current_loss = __class__._compute_loss(num_samples, y_pred, y, self.weights,
                                              self.penalty, self.lambda_or_inverseC)
        gradient_for_printing = __class__._compute_gradient(X, y_pred, y, self.weights,
                                              self.penalty, self.lambda_or_inverseC)

        if number_of_iteration == 1:
            lowest_loss = current_loss
            weights_of_lowest_loss = self.weights.copy()
            iteration_of_lowest_loss = 1
            loss_not_lower_than_lowest_counter = 0

        from decimal import Decimal
        decimalplaces_for_rounding = -Decimal(str(self.tol)).as_tuple().exponent + 2
        # for example when tol = 1e-13, then decimalplaces_for_rounding = -(-13) + 2 = 15
        # this is for when investigating the printed out Losses:
        # to be able to compare current vs. previous Loss using the computed difference

        if training_report == True:
            print(f"Training initialized with the following hyperparameter settings:\nLearning rate: {self.learning_rate}, "
                  f"inverseC: {self.lambda_or_inverseC}, Maximum iterations: {self.max_iter}, "
                  f"Tolerance for convergence: {self.tol}, Patience for early stop: {self.patience},\n"
                  f"Initial weight(s) randomly from the interval: [{self.initialweights_lowerlimit}, "
                  f"{self.initialweights_upperlimit}], Initial Loss: {current_loss.round(decimalplaces_for_rounding)}"
                  f" with the {self.weights} initial weight(s)\n")

            print(f"It: {number_of_iteration}. Loss: {current_loss.round(9)}, Weight(s): {self.weights}, "
                  f"Gradient: {gradient_for_printing.round(7)}")
        else:
            print(f"Training initialized with {self.max_iter} maximum possible iterations")

        if number_of_iteration > 1:
            # Compare current iteration's Loss to the previous iteration's
            loss_difference_from_prevloss = (previous_loss - current_loss)
            difference_in_science_format = "{:.1e}".format(np.abs(loss_difference_from_prevloss))

            if current_loss < previous_loss:
                loss_status = "lower"
            elif current_loss > previous_loss:
                loss_status = "higher"
                # when the Loss has not decreased compared to the previous iteration but increased
                # then probably the Learning rate is too high and the Loss function is increasing
            else:
                loss_status = "equal"

            if training_report == True:
                # Print information of current iteration
                if number_of_iteration <= 5 or number_of_iteration % np.round(self.max_iter*0.1, -1) == 0:

                    print(f"It: {number_of_iteration}. Loss: {current_loss.round(9)} {loss_status} "
                          f"by {difference_in_science_format} than in previous iteration, Weight(s): {self.weights}, "
                          f"Gradient: {gradient_for_printing.round(7)})")

                    if number_of_iteration == 5:
                        print()

            # Compare current iteration's Loss to the lowest Loss achieved so far
            if current_loss < lowest_loss:

```



```

previous_loss = current_loss

# ----- End of Functionalities outside of the core training (3 steps) process -----


# 2. Compute gradient (derivative) of the Loss Function
current_gradient = __class__._compute_gradient(X, y_pred, y, self.weights, self.penalty,
                                                self.lambda_or_inverseC)

# 3. Update weights
self.weights = __class__._update_weights(current_gradient, self.weights, self.penalty,
                                          self.learning_rate)


# Visualize Gradient Descent in work
if training_report == True:
    print(f"\nTraining finished with the following results:\nLast iteration: {number_of_iteration}., "
          f"Weight(s): {self.weights}, Loss: {current_loss} which was {loss_status} than in previous iteration\n"
          f"The lowest Loss {lowest_loss} was achieved in the {iteration_of_lowest_loss}. iteration"
          f"with the weight(s) {weights_of_lowest_loss}\n")

    __class__._visualize_graddesc(np.array(self.loss_history), np.array(self.weights_history),
                                 lowest_loss, num_features)

else:
    if self.max_iter == 0: # same: number_of_iteration == 0:
        print("There was no training performed as maximum iterations for the training are set to 0\n")

    else:
        print(f"\nTraining finished after {number_of_iteration} iterations\n")


# In case the training process was either stopped with early stopping by reaching the patience limit,
# or wasn't stopped before the maximum iterations, set the weight(s)
# to those weight(s) with whom the Lowest Loss was achieved (if training was stopped with convergence, then
# the last iteration's weights produce the Lowest Loss and thus this below is useless)
if number_of_iteration > 1:
    self.weights = weights_of_lowest_loss


return self


def predict_proba(self, X):
    X = __class__._standardizeX(X, self.X_train_means, self.X_train_stds)

    distances = __class__._calculate_distance(self.weights, X)

    return __class__._predict_probability(distances)

def predict(self, X):
    y_pred_proba = self.predict_proba(X)

    return y_pred_proba >= self.threshold

```

Applying created model, analyzing the training process and comparing performance to sklearn's model

```

X, y = datasets.make_classification(n_samples=10000, n_features=1, n_clusters_per_class= 1, n_informative = 1,
                                    n_redundant = 0, n_classes = 2, random_state=11)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n\n")

print("----- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight "
      "\n\n")

```

Applying created model, analyzing the training process and comparing performance to sklearn's model

```
X, y = datasets.make_classification(n_samples=10000, n_features=1, n_clusters_per_class= 1, n_informative = 1,
                                    n_redundant = 0, n_classes = 2, random_state=11)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n\n")

print("----- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight "
      "\n\n")

model1 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'l2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
                      initialweights_lowerlimit = 2, # on this particular data the optimal weight is around 0.2
                      initialweights_upperlimit = 10 # that's why these limits are given)
                     .fit(X_train, y_train, training_report = True)

y_train_pred = model1.predict(X_train)
y_pred = model1.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model2 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'l2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
                      initialweights_lowerlimit = 0.2, # on this particular data the optimal weight is around 0.2
                      initialweights_upperlimit = 2 # that's why these limits are given
                     .fit(X_train, y_train, training_report = True)

y_train_pred = model2.predict(X_train)
y_pred = model2.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model3 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'l2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
                      initialweights_lowerlimit = 0, # on this particular data the optimal weight is around 0.2
                      initialweights_upperlimit = 0.2 # that's why these limits are given
                     .fit(X_train, y_train, training_report = True)

y_train_pred = model3.predict(X_train)
y_pred = model3.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

print("----- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate "
```

```

print("----- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate "
      "\n\n")

model4 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'],#=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.1,
                      initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different Learning rates
                      initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model4.predict(X_train)
y_pred = model4.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model5 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'],#=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.4,
                      initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different Learning rates
                      initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model5.predict(X_train)
y_pred = model5.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model6 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'],#=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.5,
                      initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different Learning rates
                      initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model6.predict(X_train)
y_pred = model6.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model7 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'],#=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.8,
                      initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different Learning rates
                      initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model7.predict(X_train)

```

```

y_train_pred = model7.predict(X_train)
y_pred = model7.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model8 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                     lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                     max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                     tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                     learning_rate = 0.9,
                     initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
                     initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model8.predict(X_train)
y_pred = model8.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

```

Number of generated samples: 10000 Number of generated features: 1

----- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight -----

Training initialized with the following hyperparameter settings:

Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [2, 10], Initial Loss: 81.45833969862673 with the [9.01916148] initial weight(s)

```

It: 1. Loss: 81.458339699, Weight(s): [9.01916148], Gradient: [18.0461857]
It: 2. Loss: 52.150078823 lower by 2.9e+01 than in previous iteration, Weight(s): [7.2145429], Gradient: [14.4348743]
It: 3. Loss: 33.399821632 lower by 1.9e+01 than in previous iteration, Weight(s): [5.77105548], Gradient: [11.5435534]
It: 4. Loss: 21.411227002 lower by 1.2e+01 than in previous iteration, Weight(s): [4.61670014], Gradient: [9.2263133]
It: 5. Loss: 13.756035351 lower by 7.7e+00 than in previous iteration, Weight(s): [3.69406881], Gradient: [7.366173]

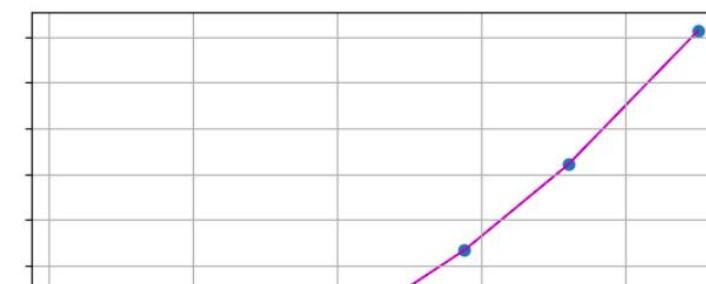
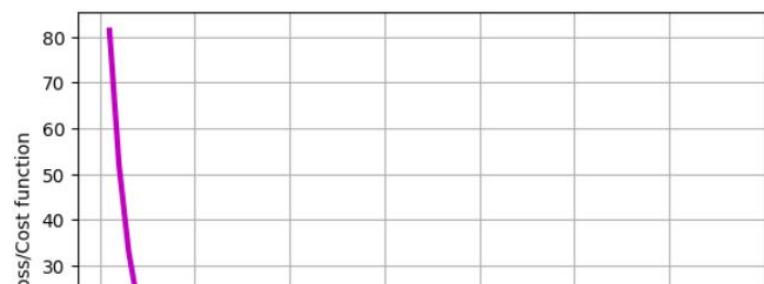
It: 10. Loss: 1.852708364 lower by 7.6e-01 than in previous iteration, Weight(s): [1.24248932], Gradient: [2.2945471]
It: 20. Loss: 0.655843041 lower by 5.2e-03 than in previous iteration, Weight(s): [0.28396894], Gradient: [0.1874847]
It: 30. Loss: 0.64805589 lower by 3.2e-05 than in previous iteration, Weight(s): [0.20698242], Gradient: [0.0147641]
It: 40. Loss: 0.648007653 lower by 2.0e-07 than in previous iteration, Weight(s): [0.2009244], Gradient: [0.0011607]
It: 50. Loss: 0.648007355 lower by 1.2e-09 than in previous iteration, Weight(s): [0.20044817], Gradient: [9.12e-05]
It: 60. Loss: 0.648007353 lower by 7.6e-12 than in previous iteration, Weight(s): [0.20041074], Gradient: [7.2e-06]

```

Converged at iteration 69., where the Loss (0.648007353207076) was lower by 7.8e-14 than
in the prev it(0.648007353207154), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 69., Weight(s): [0.20040787], Loss: 0.6480073532070763 which was lower than in previous iteration
The lowest Loss 0.6480073532070763 was achieved in the 69. iteration with the weight(s) [0.20040787]

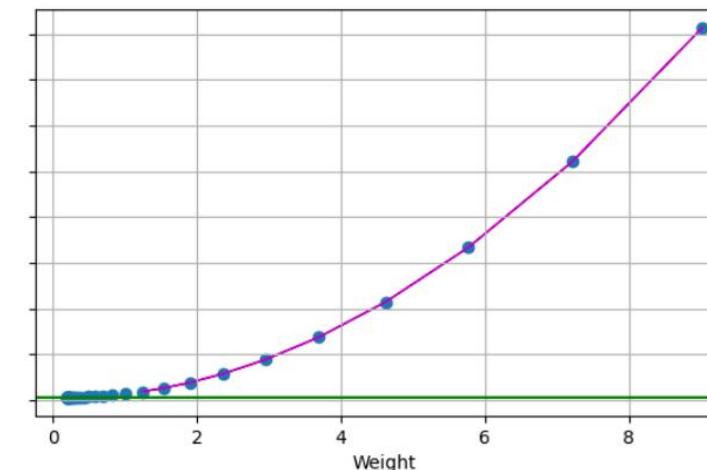
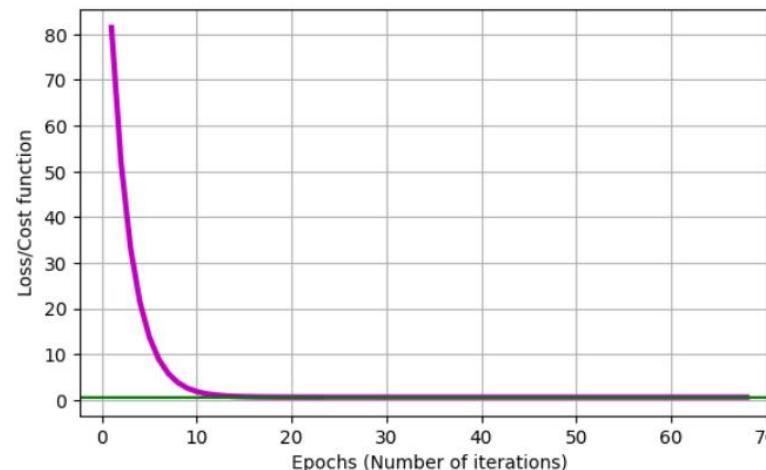


It: 60. Loss: 0.648007353 lower by 7.6e-12 than in previous iteration, Weight(s): [0.200410/4], Gradient: [/.2e-06]

Converged at iteration 69., where the Loss (0.648007353207076) was lower by 7.8e-14 than
in the prev it(0.648007353207154), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 69., Weight(s): [0.20040787], Loss: 0.6480073532070763 which was lower than in previous iteration
The lowest Loss 0.6480073532070763 was achieved in the 69. iteration with the weight(s) [0.20040787]



Accuracy on Train set: 0.969875

Accuracy on Test set: 0.963

Training initialized with the following hyperparameter settings:

Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0.2, 2), Initial Loss: 2.402061731524572 with the [1.4598525] initial weight(s)

It: 1. Loss: 2.402061732, Weight(s): [1.4598525], Gradient: [2.7594064]

It: 2. Loss: 1.721990221 lower by 6.8e-01 than in previous iteration, Weight(s): [1.18391186], Gradient: [2.1684821]

It: 3. Loss: 1.302603773 lower by 4.2e-01 than in previous iteration, Weight(s): [0.96706365], Gradient: [1.6987145]

It: 4. Loss: 1.045549692 lower by 2.6e-01 than in previous iteration, Weight(s): [0.7971922], Gradient: [1.3272254]

It: 5. Loss: 0.88877907 lower by 1.6e-01 than in previous iteration, Weight(s): [0.66446967], Gradient: [1.0348506]

It: 10. Loss: 0.667221669 lower by 1.3e-02 than in previous iteration, Weight(s): [0.33128343], Gradient: [0.2934708]

It: 20. Loss: 0.648126565 lower by 7.9e-05 than in previous iteration, Weight(s): [0.21071179], Gradient: [0.0231376]

It: 30. Loss: 0.64800809 lower by 4.9e-07 than in previous iteration, Weight(s): [0.20121759], Gradient: [0.0018191]

It: 40. Loss: 0.648007358 lower by 3.0e-09 than in previous iteration, Weight(s): [0.20047122], Gradient: [0.000143]

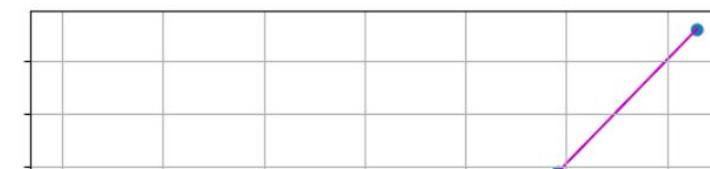
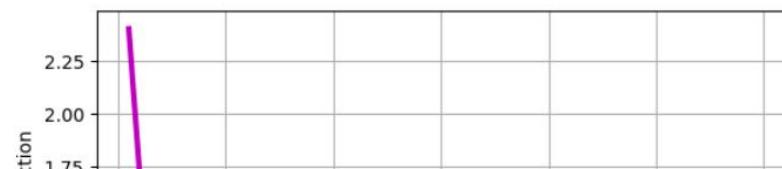
It: 50. Loss: 0.648007353 lower by 1.9e-11 than in previous iteration, Weight(s): [0.20041255], Gradient: [1.12e-05]

It: 60. Loss: 0.648007353 lower by 1.2e-13 than in previous iteration, Weight(s): [0.20040794], Gradient: [9.e-07]

Converged at iteration 61., where the Loss (0.648007353207063) was lower by 6.9e-14 than
in the prev it(0.648007353207132), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 61., Weight(s): [0.20040785], Loss: 0.6480073532070632 which was lower than in previous iteration
The lowest Loss 0.6480073532070632 was achieved in the 61. iteration with the weight(s) [0.20040785]



```

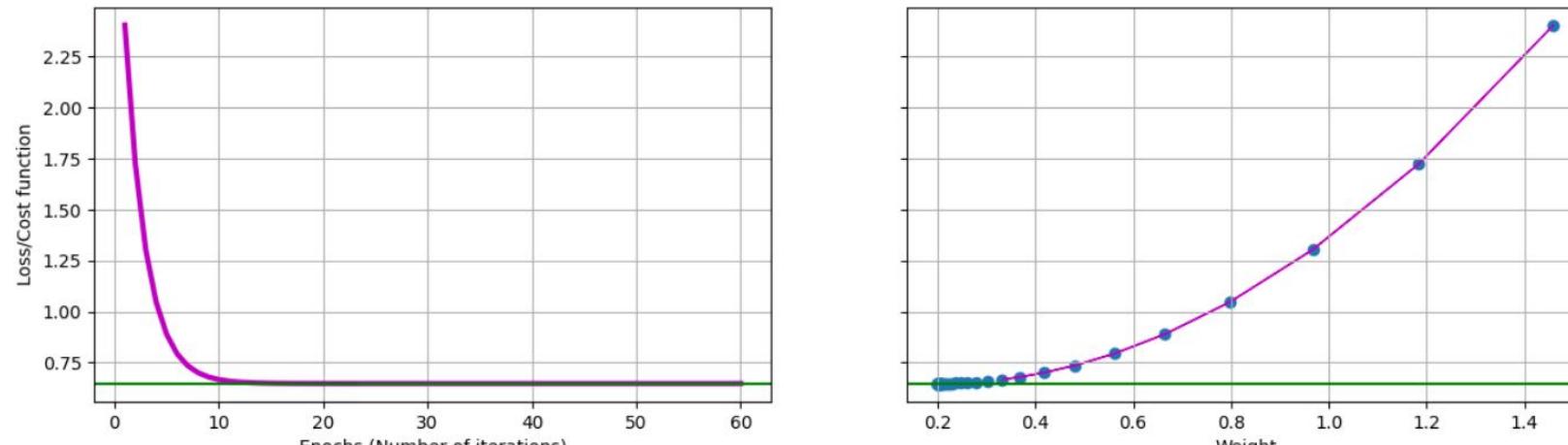
It: 1. Loss: 2.402061732, Weight(s): [1.4598525], Gradient: [2.7594064]
It: 2. Loss: 1.721990221 lower by 6.8e-01 than in previous iteration, Weight(s): [1.18391186], Gradient: [2.1684821]
It: 3. Loss: 1.302603773 lower by 4.2e-01 than in previous iteration, Weight(s): [0.96706365], Gradient: [1.6987145]
It: 4. Loss: 1.045549692 lower by 2.6e-01 than in previous iteration, Weight(s): [0.7971922], Gradient: [1.3272254]
It: 5. Loss: 0.88877907 lower by 1.6e-01 than in previous iteration, Weight(s): [0.66446967], Gradient: [1.0348506]

It: 10. Loss: 0.667221669 lower by 1.3e-02 than in previous iteration, Weight(s): [0.33128343], Gradient: [0.2934708]
It: 20. Loss: 0.648126565 lower by 7.9e-05 than in previous iteration, Weight(s): [0.21071179], Gradient: [0.0231376]
It: 30. Loss: 0.64800809 lower by 4.9e-07 than in previous iteration, Weight(s): [0.20121759], Gradient: [0.0018191]
It: 40. Loss: 0.648007358 lower by 3.0e-09 than in previous iteration, Weight(s): [0.20047122], Gradient: [0.000143]
It: 50. Loss: 0.648007353 lower by 1.9e-11 than in previous iteration, Weight(s): [0.20041255], Gradient: [1.12e-05]
It: 60. Loss: 0.648007353 lower by 1.2e-13 than in previous iteration, Weight(s): [0.20040794], Gradient: [9.e-07]

```

Converged at iteration 61., where the Loss (0.648007353207063) was lower by 6.9e-14 than
in the prev it(0.648007353207132), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last iteration: 61., Weight(s): [0.20040785], Loss: 0.6480073532070632 which was lower than in previous iteration
The lowest Loss 0.6480073532070632 was achieved in the 61. iteration with the weight(s) [0.20040785]



Accuracy on Train set: 0.96825
Accuracy on Test set: 0.963

Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0.2), Initial Loss: 0.685828398991805 with the [0.01695944] initial weight(s)

```

It: 1. Loss: 0.685828399, Weight(s): [0.01695944], Gradient: [-0.4124669]
It: 2. Loss: 0.670729356 lower by 1.5e-02 than in previous iteration, Weight(s): [0.05820613], Gradient: [-0.3196689]
It: 3. Loss: 0.661659883 lower by 9.1e-03 than in previous iteration, Weight(s): [0.09017303], Gradient: [-0.247763]
It: 4. Loss: 0.656211505 lower by 5.4e-03 than in previous iteration, Weight(s): [0.11494932], Gradient: [-0.1920449]
It: 5. Loss: 0.652937996 lower by 3.3e-03 than in previous iteration, Weight(s): [0.13415382], Gradient: [-0.1488674]

It: 10. Loss: 0.648394399 lower by 2.6e-04 than in previous iteration, Weight(s): [0.18184239], Gradient: [-0.0416983]
It: 20. Loss: 0.648007473 lower by 1.6e-06 than in previous iteration, Weight(s): [0.19894856], Gradient: [-0.0032765]
It: 30. Loss: 0.648007368 lower by 9.8e-09 than in previous iteration, Weight(s): [0.20029287], Gradient: [-0.0002575]
It: 40. Loss: 0.648007353 lower by 6.0e-11 than in previous iteration, Weight(s): [0.20039853], Gradient: [-2.02e-05]
It: 50. Loss: 0.648007353 lower by 3.7e-13 than in previous iteration, Weight(s): [0.20040684], Gradient: [-1.6e-06]

```

Converged at iteration 53., where the Loss (0.648007353207081) was lower by 8.1e-14 than
in the prev it(0.648007353207163), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

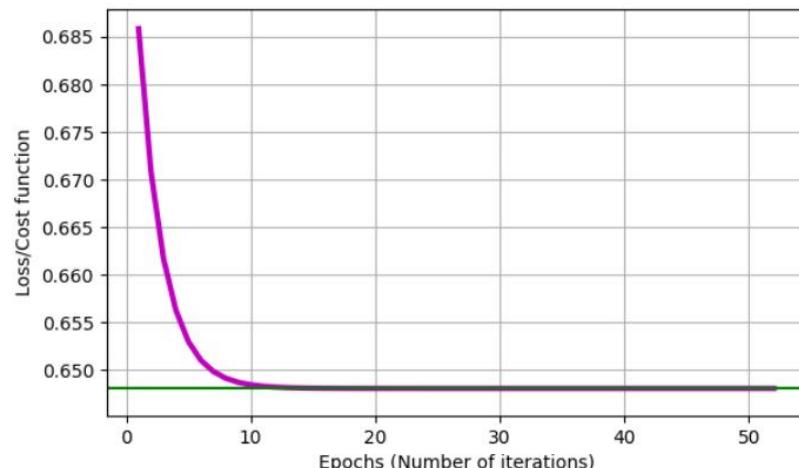
Training initialized with the following hyperparameter settings:
 Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
 Initial weight(s) randomly from the interval: [0, 0.2), Initial Loss: 0.685828398991805 with the [0.01695944] initial weight(s)

```
It: 1. Loss: 0.685828399, Weight(s): [0.01695944], Gradient: [-0.4124669]
It: 2. Loss: 0.670729356 lower by 1.5e-02 than in previous iteration, Weight(s): [0.05820613], Gradient: [-0.3196689]
It: 3. Loss: 0.661659883 lower by 9.1e-03 than in previous iteration, Weight(s): [0.09017303], Gradient: [-0.247763]
It: 4. Loss: 0.6562111505 lower by 5.4e-03 than in previous iteration, Weight(s): [0.11494932], Gradient: [-0.1920449]
It: 5. Loss: 0.652937996 lower by 3.3e-03 than in previous iteration, Weight(s): [0.13415382], Gradient: [-0.1488674]

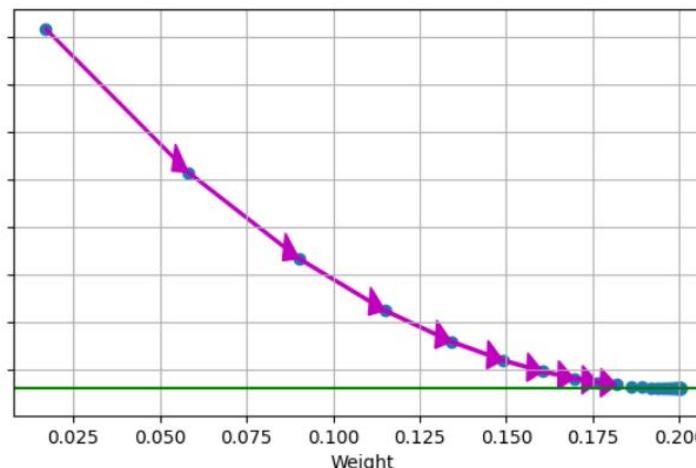
It: 10. Loss: 0.648394399 lower by 2.6e-04 than in previous iteration, Weight(s): [0.18184239], Gradient: [-0.0416983]
It: 20. Loss: 0.648009743 lower by 1.6e-06 than in previous iteration, Weight(s): [0.19894856], Gradient: [-0.0032765]
It: 30. Loss: 0.648007368 lower by 9.8e-09 than in previous iteration, Weight(s): [0.20029287], Gradient: [-0.0002575]
It: 40. Loss: 0.648007353 lower by 6.0e-11 than in previous iteration, Weight(s): [0.20039853], Gradient: [-2.02e-05]
It: 50. Loss: 0.648007353 lower by 3.7e-13 than in previous iteration, Weight(s): [0.20040684], Gradient: [-1.6e-06]
```

Converged at iteration 53., where the Loss (0.648007353207081) was lower by 8.1e-14 than
 in the prev it(0.648007353207163), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
 Last iteration: 53., Weight(s): [0.20040722], Loss: 0.6480073532070812 which was lower than in previous iteration
 The lowest Loss 0.6480073532070812 was achieved in the 53. iteration with the weight(s) [0.20040722]



Accuracy on Train set: 0.969875
 Accuracy on Test set: 0.963



----- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate -----

Training initialized with the following hyperparameter settings:
 Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
 Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

```
It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.675125271 lower by 1.8e-02 than in previous iteration, Weight(s): [0.04506255], Gradient: [-0.3492381]
It: 3. Loss: 0.664300458 lower by 1.1e-02 than in previous iteration, Weight(s): [0.07998636], Gradient: [-0.2706749]
It: 4. Loss: 0.657797878 lower by 6.5e-03 than in previous iteration, Weight(s): [0.10705385], Gradient: [-0.2097991]
It: 5. Loss: 0.653891174 lower by 3.9e-03 than in previous iteration, Weight(s): [0.12803376], Gradient: [-0.1626261]

It: 10. Loss: 0.64846917 lower by 3.1e-04 than in previous iteration, Weight(s): [0.18012839], Gradient: [-0.0455487]
It: 20. Loss: 0.648010205 lower by 1.9e-06 than in previous iteration, Weight(s): [0.19881389], Gradient: [-0.0035789]
It: 30. Loss: 0.648007371 lower by 1.2e-08 than in previous iteration, Weight(s): [0.20028228], Gradient: [-0.0002813]
```

----- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate -----

Training initialized with the following hyperparameter settings:

Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

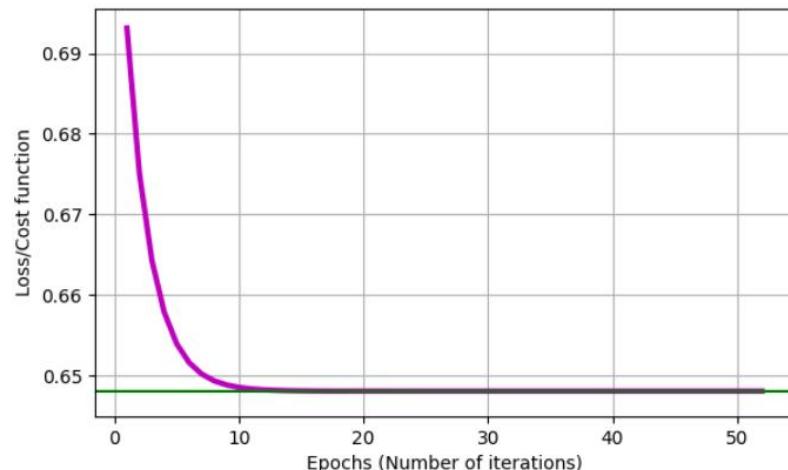
It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.675125271 lower by 1.8e-02 than in previous iteration, Weight(s): [0.04506255], Gradient: [-0.3492381]
It: 3. Loss: 0.664300458 lower by 1.1e-02 than in previous iteration, Weight(s): [0.07998636], Gradient: [-0.2706749]
It: 4. Loss: 0.657797878 lower by 6.5e-03 than in previous iteration, Weight(s): [0.10705385], Gradient: [-0.2097991]
It: 5. Loss: 0.653891174 lower by 3.9e-03 than in previous iteration, Weight(s): [0.12803376], Gradient: [-0.1626261]

It: 10. Loss: 0.64846917 lower by 3.1e-04 than in previous iteration, Weight(s): [0.18012839], Gradient: [-0.0455487]
It: 20. Loss: 0.648010205 lower by 1.9e-06 than in previous iteration, Weight(s): [0.19881389], Gradient: [-0.0035789]
It: 30. Loss: 0.648007371 lower by 1.2e-08 than in previous iteration, Weight(s): [0.20028228], Gradient: [-0.0002813]
It: 40. Loss: 0.648007353 lower by 7.2e-11 than in previous iteration, Weight(s): [0.2003977], Gradient: [-2.21e-05]
It: 50. Loss: 0.648007353 lower by 4.5e-13 than in previous iteration, Weight(s): [0.20040677], Gradient: [-1.7e-06]

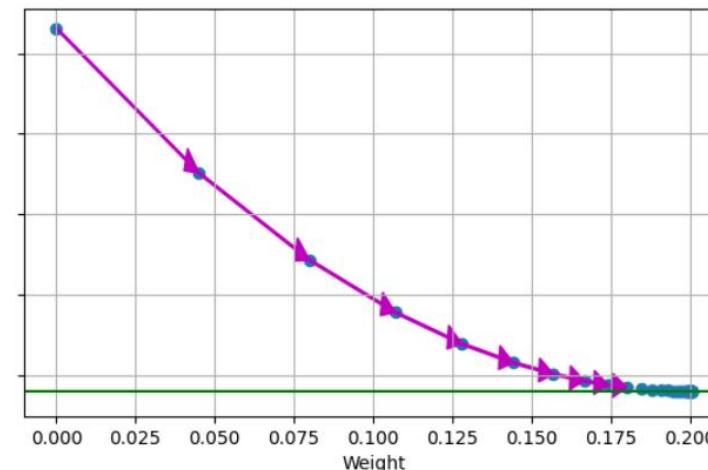
Converged at iteration 53., where the Loss (0.648007353207105) was lower by 9.7e-14 than
in the prev it(0.648007353207202), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 53., Weight(s): [0.20040719], Loss: 0.6480073532071049 which was lower than in previous iteration
The lowest Loss 0.6480073532071049 was achieved in the 53. iteration with the weight(s) [0.20040719]



Accuracy on Train set: 0.96825
Accuracy on Test set: 0.963



Training initialized with the following hyperparameter settings:

Learning rate: 0.4, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

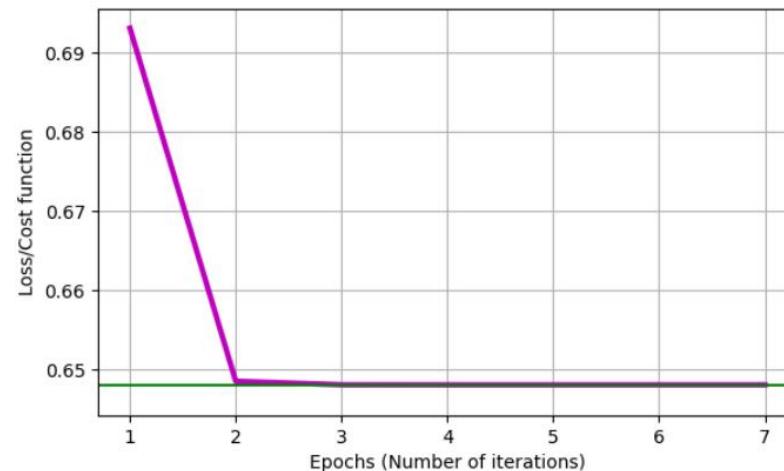
It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.648463639 lower by 4.5e-02 than in previous iteration, Weight(s): [0.1802502], Gradient: [-0.0452751]
It: 3. Loss: 0.64801206 lower by 4.5e-04 than in previous iteration, Weight(s): [0.19836024], Gradient: [-0.0045977]
It: 4. Loss: 0.648007402 lower by 4.7e-06 than in previous iteration, Weight(s): [0.20019931], Gradient: [-0.0004676]
It: 5. Loss: 0.648007354 lower by 4.8e-08 than in previous iteration, Weight(s): [0.20038636], Gradient: [-4.76e-05]

Converged at iteration 8., where the Loss (0.648007353206959) was lower by 5.3e-14 than
in the prev it(0.648007353207012), which is below the tolerance for convergence: 1.0e-13

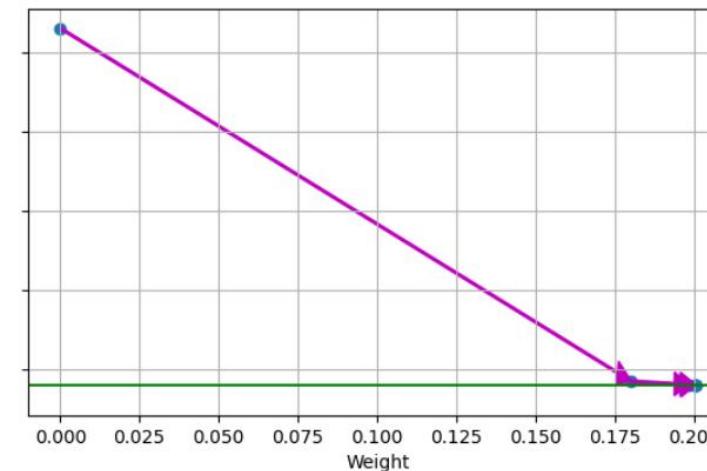
Converged at iteration 8., where the Loss (0.648007353206959) was lower by 5.3e-14 than in the prev it(0.648007353207012), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 8., Weight(s): [0.20040753], Loss: 0.6480073532069592 which was lower than in previous iteration
The lowest Loss 0.6480073532069592 was achieved in the 8. iteration with the weight(s) [0.20040753]



Accuracy on Train set: 0.969875
Accuracy on Test set: 0.963



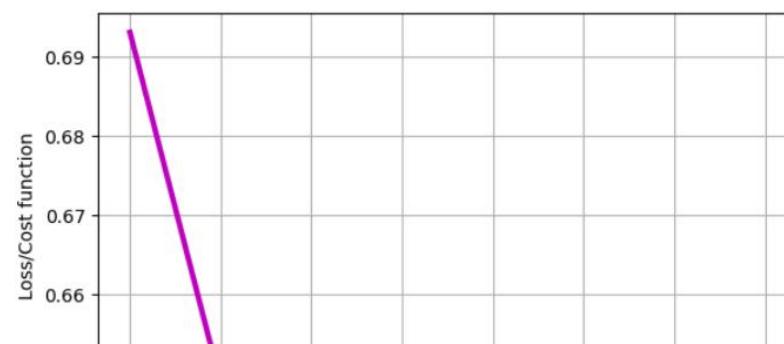
Training initialized with the following hyperparameter settings:
Learning rate: 0.5, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.648703701 lower by 4.4e-02 than in previous iteration, Weight(s): [0.22531275], Gradient: [0.0559152]
It: 3. Loss: 0.648017815 lower by 6.9e-04 than in previous iteration, Weight(s): [0.19735515], Gradient: [-0.0068549]
It: 4. Loss: 0.648007511 lower by 1.0e-05 than in previous iteration, Weight(s): [0.20078259], Gradient: [0.0008422]
It: 5. Loss: 0.648007356 lower by 1.6e-07 than in previous iteration, Weight(s): [0.20036148], Gradient: [-0.0001034]

Converged at iteration 9., where the Loss (0.648007353206959) was lower by 8.0e-15 than in the prev it(0.648007353206967), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

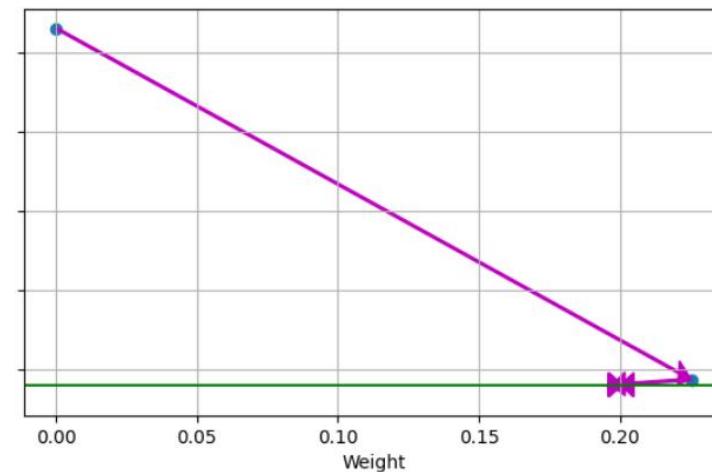
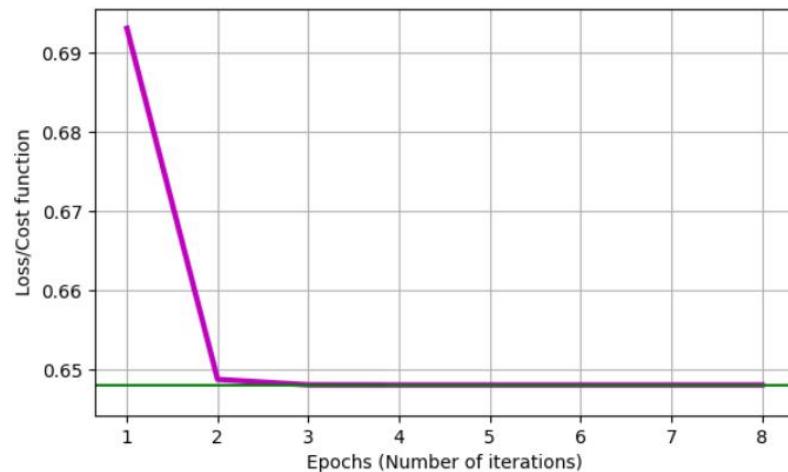
Last iteration: 9., Weight(s): [0.20040754], Loss: 0.6480073532069588 which was lower than in previous iteration
The lowest Loss 0.6480073532069588 was achieved in the 9. iteration with the weight(s) [0.20040754]



Converged at iteration 9., where the Loss (0.648007353206959) was lower by 8.0e-15 than in the prev it(0.648007353206967), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 9., Weight(s): [0.20040754], Loss: 0.648007353206958 which was lower than in previous iteration
The lowest Loss 0.6480073532069588 was achieved in the 9. iterationwith the weight(s) [0.20040754]



Accuracy on Train set: 0.96825

Accuracy on Test set: 0.963

Training initialized with the following hyperparameter settings:

Learning rate: 0.8, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

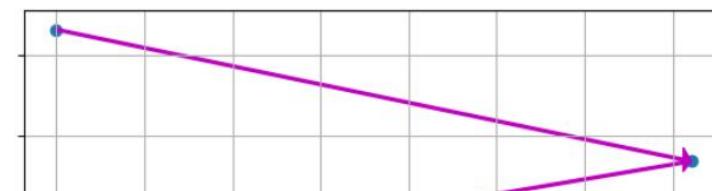
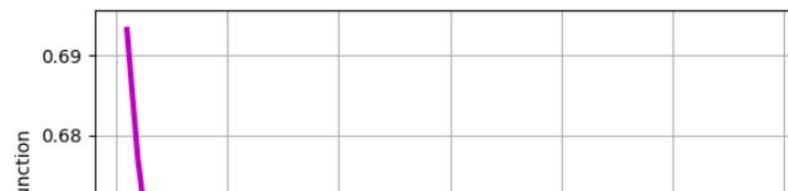
It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.676751161 lower by 1.6e-02 than in previous iteration, Weight(s): [0.3605004], Gradient: [0.3588454]
It: 3. Loss: 0.666125128 lower by 1.1e-02 than in previous iteration, Weight(s): [0.0734241], Gradient: [-0.2854357]
It: 4. Loss: 0.659536162 lower by 6.6e-03 than in previous iteration, Weight(s): [0.3017727], Gradient: [0.2273819]
It: 5. Loss: 0.655294248 lower by 4.2e-03 than in previous iteration, Weight(s): [0.11986717], Gradient: [-0.1809872]

It: 10. Loss: 0.648757173 lower by 4.3e-04 than in previous iteration, Weight(s): [0.22625137], Gradient: [0.058022]
It: 20. Loss: 0.648015275 lower by 4.6e-06 than in previous iteration, Weight(s): [0.20306377], Gradient: [0.0059648]
It: 30. Loss: 0.648007437 lower by 4.8e-08 than in previous iteration, Weight(s): [0.20068064], Gradient: [0.0006133]
It: 40. Loss: 0.648007354 lower by 5.1e-10 than in previous iteration, Weight(s): [0.20043563], Gradient: [6.31e-05]
It: 50. Loss: 0.648007353 lower by 5.4e-12 than in previous iteration, Weight(s): [0.20040718], Gradient: [6.5e-06]

Converged at iteration 59., where the Loss (0.648007353207114) was lower by 9.0e-14 than in the prev it(0.648007353207204), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

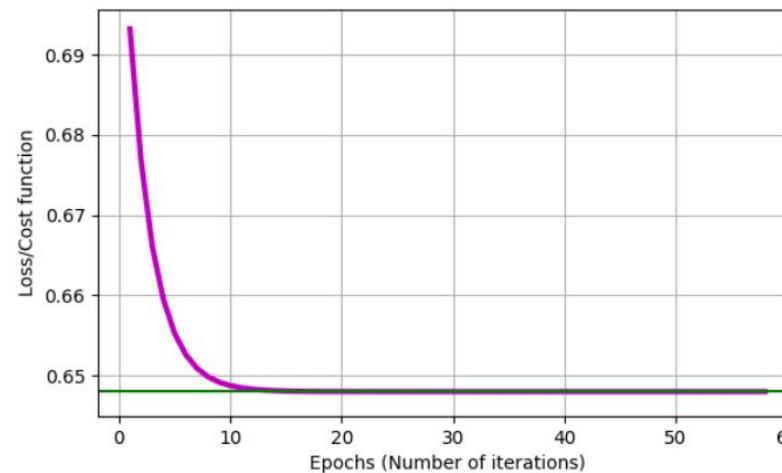
Last iteration: 59., Weight(s): [0.20040718], Loss: 0.6480073532071146 which was lower than in previous iteration
The lowest Loss 0.6480073532071146 was achieved in the 59. iterationwith the weight(s) [0.20040718]



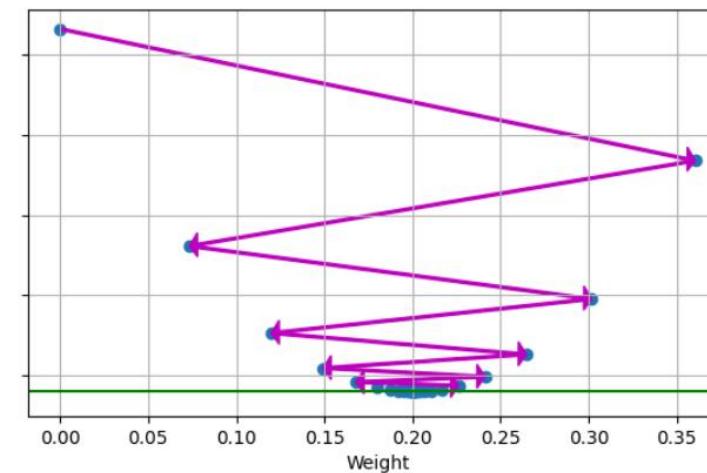
in the prev it(0.648007353207204), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 59., Weight(s): [0.20040718], Loss: 0.6480073532071146 which was lower than in previous iteration
The lowest Loss 0.6480073532071146 was achieved in the 59. iteration with the weight(s) [0.20040718]



Accuracy on Train set: 0.969875
Accuracy on Test set: 0.963



Training initialized with the following hyperparameter settings:

Learning rate: 0.9, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.695191357 higher by 2.0e-03 than in previous iteration, Weight(s): [0.40556295], Gradient: [0.4595564]
It: 3. Loss: 0.696841896 higher by 1.7e-03 than in previous iteration, Weight(s): [-0.00803779], Gradient: [-0.4687105]
It: 4. Loss: 0.699053298 higher by 2.2e-03 than in previous iteration, Weight(s): [0.41380167], Gradient: [0.4779527]
It: 5. Loss: 0.700818471 higher by 1.8e-03 than in previous iteration, Weight(s): [-0.0163558], Gradient: [-0.4874259]

It: 10. Loss: 0.712479494 higher by 2.8e-03 than in previous iteration, Weight(s): [0.44025842], Gradient: [0.5369923]
It: 20. Loss: 0.742173499 higher by 4.0e-03 than in previous iteration, Weight(s): [0.49034785], Gradient: [0.6486118]

Early stopped at iteration 21., where the patience limit (20) was reached

The lowest Loss 0.693147180559945 was achieved in the 1. iteration with the weight(s) [0.], and after that there were 20 iterations where the Loss was not lower than that

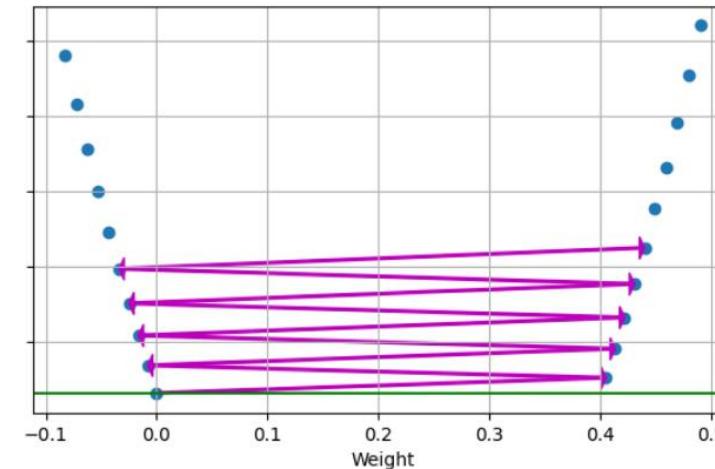
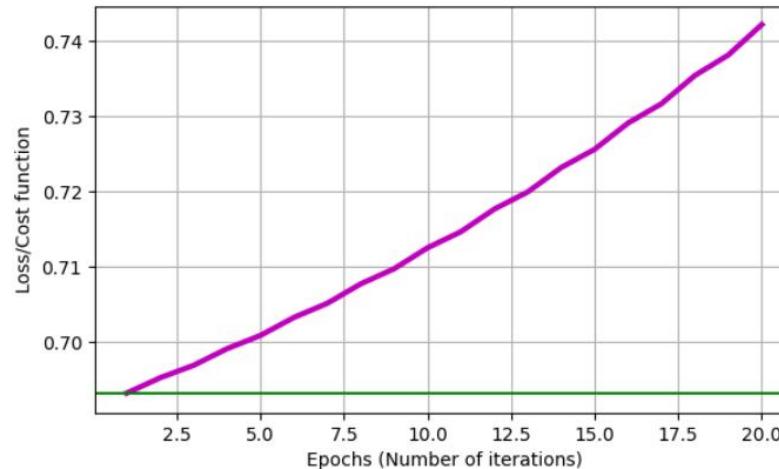
Training finished with the following results:

Last iteration: 21., Weight(s): [-0.09340276], Loss: 0.7450507371802867 which was higher than in previous iteration
The lowest Loss 0.693147180559945 was achieved in the 1. iteration with the weight(s) [0.]



Early stopped at iteration 21., where the patience limit (20) was reached
The lowest Loss 0.693147180559945 was achieved in the 1. iteration with the weight(s) [0.], and after that there were 20 iterations where the Loss was not lower than that

Training finished with the following results:
Last iteration: 21., Weight(s): [-0.09340276], Loss: 0.7450507371802867 which was higher than in previous iteration
The lowest Loss 0.693147180559945 was achieved in the 1. iteration with the weight(s) [0.]



Accuracy on Train set: 0.494875
Accuracy on Test set: 0.5245

```
X, y = datasets.make_classification(n_samples=10000, n_features=2, n_clusters_per_class= 1, n_informative = 1,
                                    n_redundant = 0, n_classes = 2, random_state=11)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n\n")

print("----- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight "
      "\n\n")

model1 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
                      initialweights_lowerlimit = 2,
                      initialweights_upperlimit = 6
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model1.predict(X_train)
y_pred = model1.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model2 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                      lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                      max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                      tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                      learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
                      initialweights_lowerlimit = 2,
                      initialweights_upperlimit = 6
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model2.predict(X_train)
y_pred = model2.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")
```

```

model2 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
    lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
    max_iter = DEFAULTSKPARAMS['max_iter'], #=100
    tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
    learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
    initialweights_lowerlimit = 0.2,
    initialweights_upperlimit = 2
    ).fit(X_train, y_train, training_report = True)

y_train_pred = model2.predict(X_train)
y_pred = model2.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model3 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
    lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
    max_iter = DEFAULTSKPARAMS['max_iter'], #=100
    tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
    learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
    initialweights_lowerlimit = 0,
    initialweights_upperlimit = 0.2
    ).fit(X_train, y_train, training_report = True)

y_train_pred = model3.predict(X_train)
y_pred = model3.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

print("----- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate "
      "\n\n")
model4 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
    lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
    max_iter = DEFAULTSKPARAMS['max_iter'], #=100
    tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
    learning_rate = 0.1,
    initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
    initialweights_upperlimit = 0
    ).fit(X_train, y_train, training_report = True)

y_train_pred = model4.predict(X_train)
y_pred = model4.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model5 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
    lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
    max_iter = DEFAULTSKPARAMS['max_iter'], #=100
    tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
    learning_rate = 0.4,
    initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
    initialweights_upperlimit = 0
    ).fit(X_train, y_train, training_report = True)

y_train_pred = model5.predict(X_train)
y_pred = model5.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)

```

```

).fit(X_train, y_train, training_report = True)

y_train_pred = model5.predict(X_train)
y_pred = model5.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model6 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                     lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                     max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                     tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                     learning_rate = 0.5,
                     initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
                     initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model6.predict(X_train)
y_pred = model6.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model7 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                     lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                     max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                     tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                     learning_rate = 0.8,
                     initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
                     initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model7.predict(X_train)
y_pred = model7.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

model8 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #'L2'
                     lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
                     max_iter = DEFAULTSKPARAMS['max_iter'], #=100
                     tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
                     learning_rate = 0.9,
                     initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
                     initialweights_upperlimit = 0
                     ).fit(X_train, y_train, training_report = True)

y_train_pred = model8.predict(X_train)
y_pred = model8.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")

```

Number of generated samples: 10000 Number of generated features: 2

----- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight -----

Number of generated samples: 10000 Number of generated features: 2

----- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight -----

Training initialized with the following hyperparameter settings:

Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [2, 6), Initial Loss: 33.50612445884154 with the [2.89818808 4.98351743] initial weight(s)

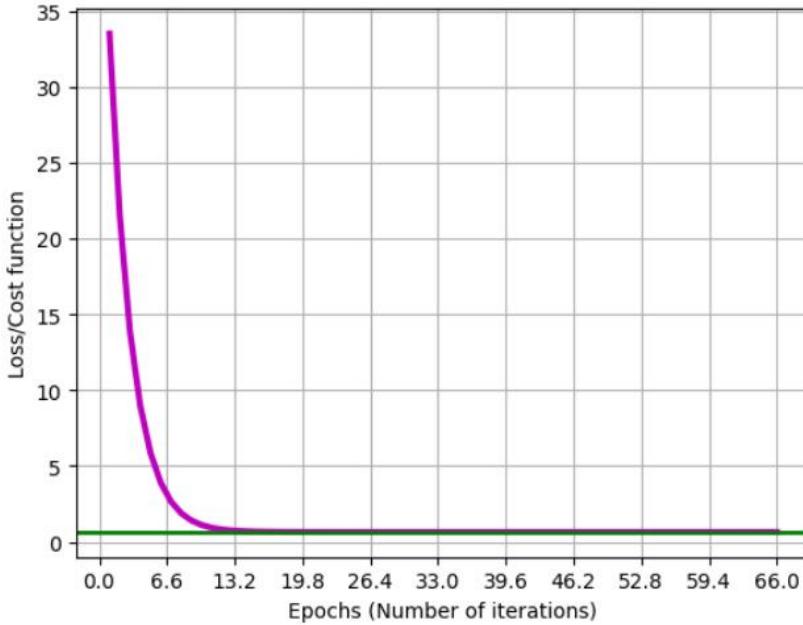
It: 1. Loss: 33.506124459, Weight(s): [2.89818808 4.98351743], Gradient: [5.9355506 9.9071946]
It: 2. Loss: 21.505991554 lower by 1.2e+01 than in previous iteration, Weight(s): [2.30463302 3.99279798], Gradient: [4.7491078 7.9154874]
It: 3. Loss: 13.842442023 lower by 7.7e+00 than in previous iteration, Weight(s): [1.82972224 3.20124924], Gradient: [3.7988291 6.3178954]
It: 4. Loss: 8.957239129 lower by 4.9e+00 than in previous iteration, Weight(s): [1.44983933 2.56945969], Gradient: [3.0364183 5.0349542]
It: 5. Loss: 5.852332528 lower by 3.1e+00 than in previous iteration, Weight(s): [1.1461975 2.06596427], Gradient: [2.4232886 4.0036814]

It: 10. Loss: 1.105823629 lower by 2.9e-01 than in previous iteration, Weight(s): [0.34197211 0.7447842], Gradient: [0.7449147 1.2123816]
It: 20. Loss: 0.651182719 lower by 1.9e-03 than in previous iteration, Weight(s): [0.03357508 0.24264454], Gradient: [0.0591315 0.0969486]
It: 30. Loss: 0.648336759 lower by 1.2e-05 than in previous iteration, Weight(s): [0.00942578 0.2029147], Gradient: [0.0045614 0.0075794]
It: 40. Loss: 0.648319501 lower by 7.0e-08 than in previous iteration, Weight(s): [0.00756396 0.19981003], Gradient: [0.0003514 0.000592]
It: 50. Loss: 0.648319397 lower by 4.2e-10 than in previous iteration, Weight(s): [0.00742056 0.19956755], Gradient: [2.71e-05 4.62e-05]
It: 60. Loss: 0.648319396 lower by 2.6e-12 than in previous iteration, Weight(s): [0.00740952 0.19954862], Gradient: [2.1e-06 3.6e-06]

Converged at iteration 67., where the Loss (0.648319396111103) was lower by 7.2e-14 than
in the prev it(0.648319396111175), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 67., Weight(s): [0.00740875 0.19954728], Loss: 0.6483193961111027 which was lower than in previous iteration
The lowest Loss 0.6483193961111027 was achieved in the 67. iteration with the weight(s) [0.00740875 0.19954728]



Accuracy on Train set: 0.96625

Accuracy on Test set: 0.962

Training initialized with the following hyperparameter settings:

Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [0.2, 2), Initial Loss: 4.72373869589876 with the [1.6994989 1.13235245] initial weight(s)

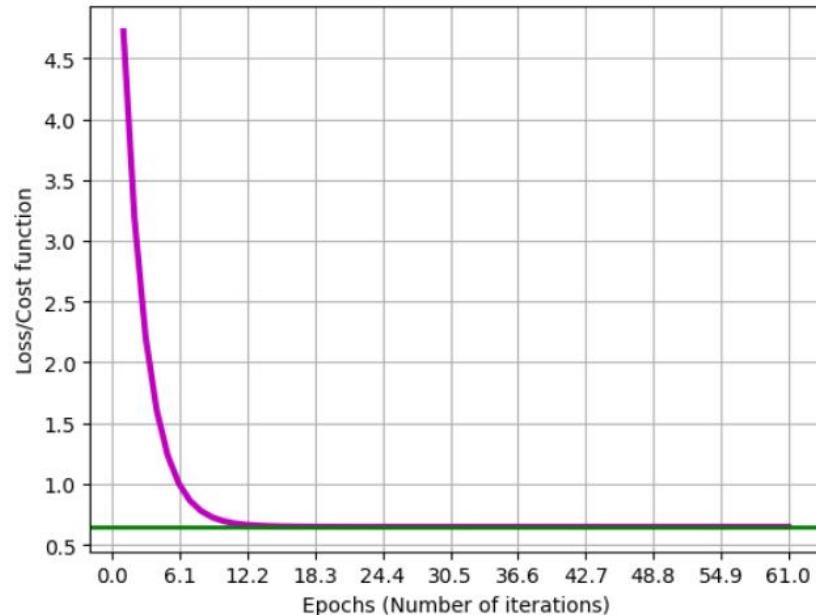
It: 1. Loss: 4.723738696, Weight(s): [1.6994989 1.13235245], Gradient: [3.6353942 2.0006157]
It: 2. Loss: 3.17983031 lower by 1.5e+00 than in previous iteration, Weight(s): [1.33595948 0.93229089], Gradient: [2.8818782 1.5872726]

Training initialized with the following hyperparameter settings:
 Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
 Initial weight(s) randomly from the interval: [0.2, 2], Initial Loss: 4.72373869589876 with the [1.6994989 1.13235245] initial weight(s)

It: 1. Loss: 4.723738696, Weight(s): [1.6994989 1.13235245], Gradient: [3.6353942 2.0006157]
 It: 2. Loss: 3.17983031 lower by 1.5e+00 than in previous iteration, Weight(s): [1.33595948 0.93229089], Gradient: [2.8818782 1.5872726]
 It: 3. Loss: 2.210685714 lower by 9.7e-01 than in previous iteration, Weight(s): [1.04777166 0.77356363], Gradient: [2.276655 1.254969]
 It: 4. Loss: 1.606615752 lower by 6.0e-01 than in previous iteration, Weight(s): [0.82010616 0.64806673], Gradient: [1.7918589 0.9884953]
 It: 5. Loss: 1.232889744 lower by 3.7e-01 than in previous iteration, Weight(s): [0.64092027 0.5492172], Gradient: [1.4052178 0.7757172]
 It: 10. Loss: 0.69533007 lower by 3.1e-02 than in previous iteration, Weight(s): [0.18643102 0.29819371], Gradient: [0.4022017 0.2221722]
 It: 20. Loss: 0.648605632 lower by 1.9e-04 than in previous iteration, Weight(s): [0.02139807 0.20718176], Gradient: [0.0315012 0.0172579]
 It: 30. Loss: 0.648321127 lower by 1.2e-06 than in previous iteration, Weight(s): [0.00849941 0.2001355], Gradient: [0.0024565 0.0013306]
 It: 40. Loss: 0.648319407 lower by 7.0e-09 than in previous iteration, Weight(s): [0.00749365 0.19959235], Gradient: [0.0001915 0.0001025]
 It: 50. Loss: 0.648319396 lower by 4.2e-11 than in previous iteration, Weight(s): [0.00741523 0.1995505], Gradient: [1.49e-05 7.90e-06]
 It: 60. Loss: 0.648319396 lower by 2.6e-13 than in previous iteration, Weight(s): [0.00740911 0.19954728], Gradient: [1.2e-06 6.0e-07]

Converged at iteration 62., where the Loss (0.648319396111132) was lower by 9.2e-14 than
 in the prev it(0.648319396111224), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
 Last iteration: 62., Weight(s): [0.00740891 0.19954717], Loss: 0.6483193961111323 which was lower than in previous iteration
 The lowest Loss 0.6483193961111323 was achieved in the 62. iteration with the weight(s) [0.00740891 0.19954717]



Accuracy on Train set: 0.9665
 Accuracy on Test set: 0.9615

Training initialized with the following hyperparameter settings:
 Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
 Initial weight(s) randomly from the interval: [0, 0.2), Initial Loss: 0.700201731952336 with the [0.13328449 0.02497659] initial weight(s)

It: 1. Loss: 0.700201732, Weight(s): [0.13328449 0.02497659], Gradient: [0.2815861 -0.3914447]
 It: 2. Loss: 0.679554759 lower by 2.1e-02 than in previous iteration, Weight(s): [0.10512588 0.06412106], Gradient: [0.2186186 -0.3036367]
 It: 3. Loss: 0.667124523 lower by 1.2e-02 than in previous iteration, Weight(s): [0.08326402 0.09448472], Gradient: [0.1697081 -0.2355242]
 It: 4. Loss: 0.65964161 lower by 7.5e-03 than in previous iteration, Weight(s): [0.0662932 0.11803714], Gradient: [0.1317335 -0.1826979]
 It: 5. Loss: 0.655136819 lower by 4.5e-03 than in previous iteration, Weight(s): [0.05311986 0.13630693], Gradient: [0.1022557 -0.1417279]
 It: 10. Loss: 0.648859546 lower by 3.6e-04 than in previous iteration, Weight(s): [0.02029863 0.18175931], Gradient: [0.0288268 -0.039847]
 It: 20. Loss: 0.648322794 lower by 2.2e-06 than in previous iteration, Weight(s): [0.00843426 0.19813834], Gradient: [0.0022935 -0.003155]
 It: 30. Loss: 0.648319418 lower by 1.4e-08 than in previous iteration, Weight(s): [0.00749022 0.19943542], Gradient: [0.0001825 -0.0002499]
 It: 40. Loss: 0.648319396 lower by 8.9e-11 than in previous iteration, Weight(s): [0.00741500 0.19952817], Gradient: [1.15e-05 1.08e-05]

```

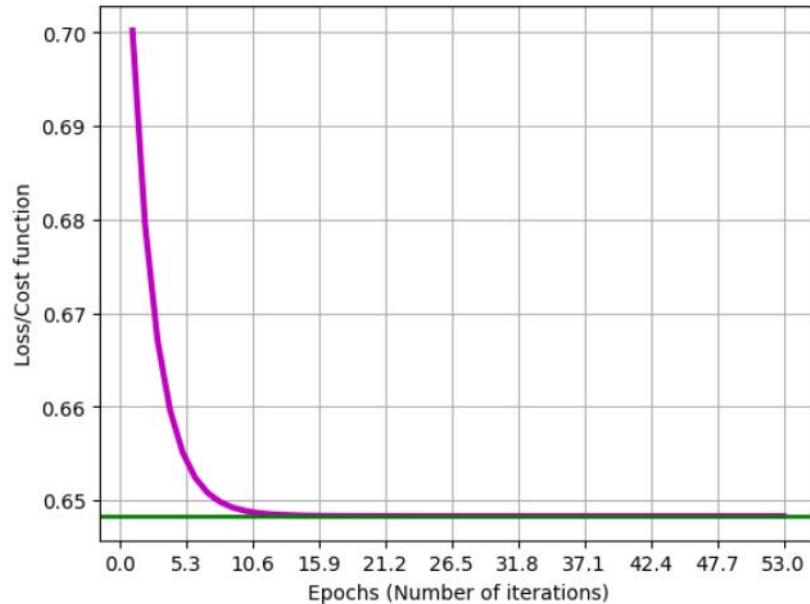
It: 30. Loss: 0.648319418 lower by 1.4e-08 than in previous iteration, Weight(s): [0.00749022 0.19943542], Gradient: [ 0.0001825 -0.0002499]
It: 40. Loss: 0.648319396 lower by 8.9e-11 than in previous iteration, Weight(s): [0.00741509 0.19953817], Gradient: [ 1.45e-05 -1.98e-05]
It: 50. Loss: 0.648319396 lower by 5.6e-13 than in previous iteration, Weight(s): [0.00740911 0.19954631], Gradient: [ 1.2e-06 -1.6e-06]

```

Converged at iteration 54., where the Loss (0.648319396111106) was lower by 7.4e-14 than in the prev it(0.64831939611118), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 54., Weight(s): [0.00740878 0.19954676], Loss: 0.6483193961111063 which was lower than in previous iteration
The lowest Loss 0.6483193961111063 was achieved in the 54. iteration with the weight(s) [0.00740878 0.19954676]



Accuracy on Train set: 0.96625

Accuracy on Test set: 0.962

----- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate -----

Training initialized with the following hyperparameter settings:

Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

```

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.675245175 lower by 1.8e-02 than in previous iteration, Weight(s): [0.00183623 0.04487573], Gradient: [-0.0138445 -0.3477744]
It: 3. Loss: 0.664494313 lower by 1.1e-02 than in previous iteration, Weight(s): [0.00322068 0.07965317], Gradient: [-0.0104313 -0.2695284]
It: 4. Loss: 0.658037291 lower by 6.5e-03 than in previous iteration, Weight(s): [0.00426381 0.10660601], Gradient: [-0.0078545 -0.2089012]
It: 5. Loss: 0.654158622 lower by 3.9e-03 than in previous iteration, Weight(s): [0.00504925 0.12749614], Gradient: [-0.0059102 -0.161923 ]

```

```

It: 10. Loss: 0.64877737 lower by 3.0e-04 than in previous iteration, Weight(s): [0.00685711 0.17936223], Gradient: [-0.001408 -0.0453424]
It: 20. Loss: 0.648322221 lower by 1.9e-06 than in previous iteration, Weight(s): [0.00738237 0.19796126], Gradient: [-7.2100e-05 -3.5614e-03]
It: 30. Loss: 0.648319414 lower by 1.2e-08 than in previous iteration, Weight(s): [0.00740788 0.19942239], Gradient: [-2.7000e-06 -2.799e-04]
It: 40. Loss: 0.648319396 lower by 7.1e-11 than in previous iteration, Weight(s): [0.00740865 0.19953722], Gradient: [ 0.0e+00 -2.2e-05]
It: 50. Loss: 0.648319396 lower by 4.4e-13 than in previous iteration, Weight(s): [0.00740861 0.19954624], Gradient: [ 0.0e+00 -1.7e-06]

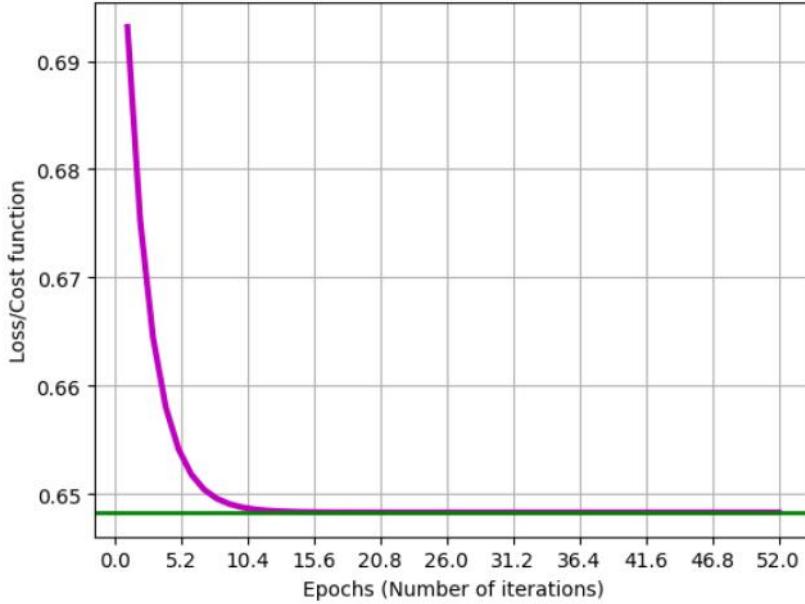
```

Converged at iteration 53., where the Loss (0.648319396111139) was lower by 9.6e-14 than in the prev it(0.648319396111135), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 53., Weight(s): [0.0074086 0.19954665], Loss: 0.6483193961111394 which was lower than in previous iteration
The lowest Loss 0.6483193961111394 was achieved in the 53. iteration with the weight(s) [0.0074086 0.19954665]

Training finished with the following results:
 Last iteration: 53., Weight(s): [0.0074086 0.19954665], Loss: 0.6483193961111394 which was lower than in previous iteration
 The lowest Loss 0.6483193961111394 was achieved in the 53. iteration with the weight(s) [0.0074086 0.19954665]



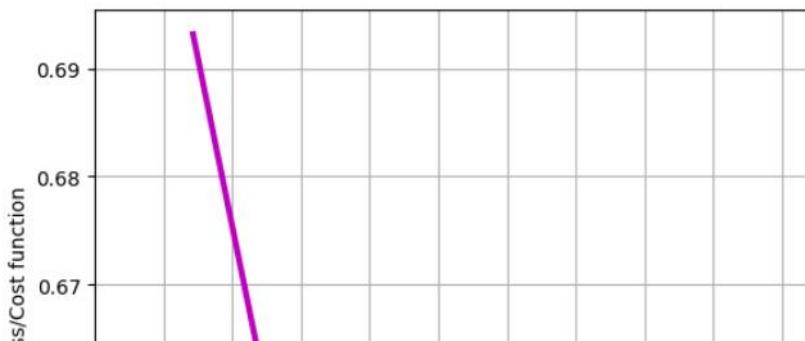
Accuracy on Train set: 0.9665
 Accuracy on Test set: 0.9615

Training initialized with the following hyperparameter settings:
 Learning rate: 0.4, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
 Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

```
It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.648770593 lower by 4.4e-02 than in previous iteration, Weight(s): [0.00734493 0.17950293], Gradient: [-0.0003102 -0.0450222]
It: 3. Loss: 0.64832405 lower by 4.5e-04 than in previous iteration, Weight(s): [0.00746901 0.1975118], Gradient: [ 0.0001189 -0.0045701]
It: 4. Loss: 0.648319444 lower by 4.6e-06 than in previous iteration, Weight(s): [0.00742147 0.19933984], Gradient: [ 2.720e-05 -4.651e-04]
It: 5. Loss: 0.648319397 lower by 4.8e-08 than in previous iteration, Weight(s): [0.00741059 0.1995259], Gradient: [ 4.30e-06 -4.74e-05]
```

Converged at iteration 8., where the Loss (0.648319396110995) was lower by 5.5e-14 than
 in the prev it(0.64831939611105), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
 Last iteration: 8., Weight(s): [0.0074086 0.19954699], Loss: 0.6483193961109951 which was lower than in previous iteration
 The lowest Loss 0.6483193961109951 was achieved in the 8. iteration with the weight(s) [0.0074086 0.19954699]

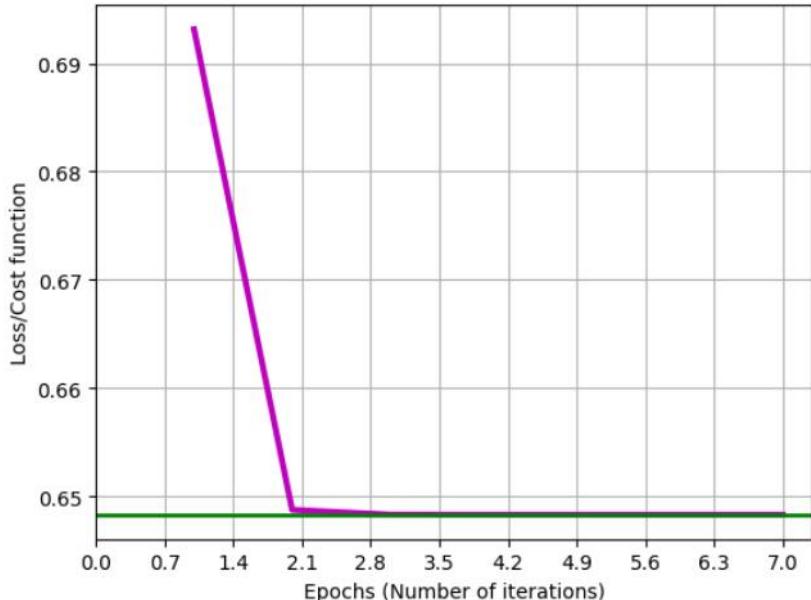


Converged at iteration 8., where the Loss (0.648319396110995) was lower by 5.5e-14 than in the prev it(0.64831939611105), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 8., Weight(s): [0.0074086 0.19954699], Loss: 0.6483193961109951 which was lower than in previous iteration

The lowest Loss 0.6483193961109951 was achieved in the 8. iteration with the weight(s) [0.0074086 0.19954699]



Accuracy on Train set: 0.96625

Accuracy on Test set: 0.962

Training initialized with the following hyperparameter settings:

Learning rate: 0.5, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]

It: 2. Loss: 0.64901555 lower by 4.4e-02 than in previous iteration, Weight(s): [0.00918117 0.22437866], Gradient: [0.0041885 0.0557662]

It: 3. Loss: 0.648329976 lower by 6.9e-04 than in previous iteration, Weight(s): [0.00708691 0.19649556], Gradient: [-0.0007484 -0.0068556]

It: 4. Loss: 0.648319558 lower by 1.0e-05 than in previous iteration, Weight(s): [0.00746111 0.19992336], Gradient: [0.0001211 0.0008456]

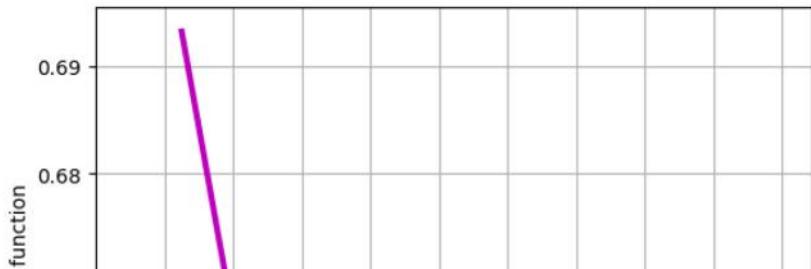
It: 5. Loss: 0.648319399 lower by 1.6e-07 than in previous iteration, Weight(s): [0.00740053 0.19950056], Gradient: [-1.850e-05 -1.044e-04]

Converged at iteration 9., where the Loss (0.648319396110995) was lower by 9.1e-15 than in the prev it(0.648319396111004), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 9., Weight(s): [0.00740859 0.199547], Loss: 0.6483193961109949 which was lower than in previous iteration

The lowest Loss 0.6483193961109949 was achieved in the 9. iteration with the weight(s) [0.00740859 0.199547]

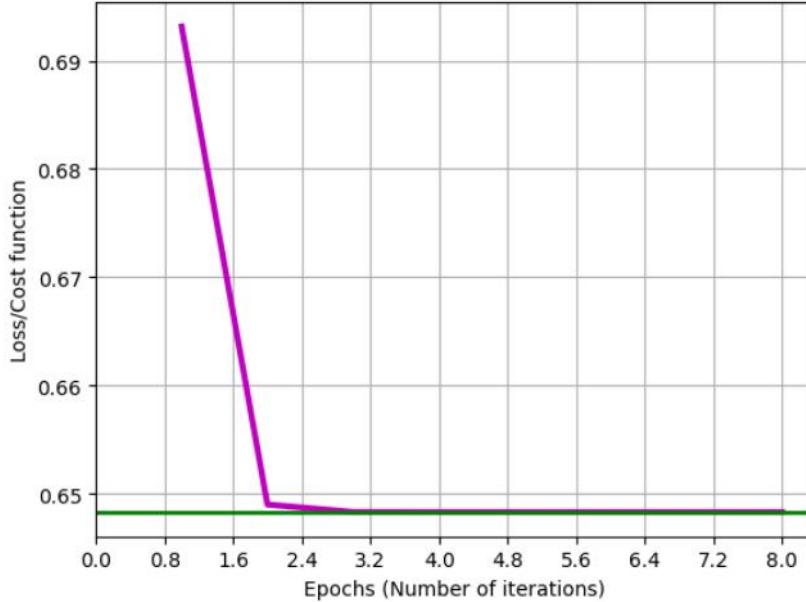


It: 5. Loss: 0.648319399 lower by 1.6e-07 than in previous iteration, Weight(s): [0.00740053 0.19950056], Gradient: [-1.850e-05 -1.044e-04]

Converged at iteration 9., where the Loss (0.648319396110995) was lower by 9.1e-15 than in the prev it(0.648319396111004), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 9., Weight(s): [0.00740859 0.199547], Loss: 0.6483193961109949 which was lower than in previous iteration
The lowest Loss 0.6483193961109949 was achieved in the 9. iteration with the weight(s) [0.00740859 0.199547]



Accuracy on Train set: 0.9665

Accuracy on Test set: 0.9615

Training initialized with the following hyperparameter settings:

Learning rate: 0.8, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]

It: 2. Loss: 0.676906019 lower by 1.6e-02 than in previous iteration, Weight(s): [0.01468987 0.35900586], Gradient: [0.0176221 0.3574985]
It: 3. Loss: 0.666370645 lower by 1.1e-02 than in previous iteration, Weight(s): [0.00059219 0.07300707], Gradient: [-0.0164014 -0.2845005]
It: 4. Loss: 0.659829209 lower by 6.5e-03 than in previous iteration, Weight(s): [0.01371333 0.30060744], Gradient: [0.0149793 0.2267572]
It: 5. Loss: 0.655611274 lower by 4.2e-03 than in previous iteration, Weight(s): [0.00172986 0.11920165], Gradient: [-0.0134463 -0.1806002]

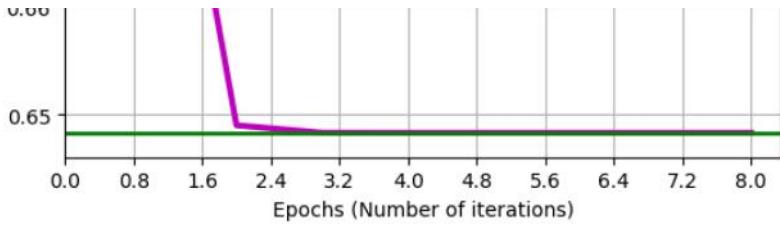
It: 10. Loss: 0.649081701 lower by 4.3e-04 than in previous iteration, Weight(s): [0.0103301 0.22542959], Gradient: [0.0067786 0.0581345]
It: 20. Loss: 0.648327882 lower by 4.8e-06 than in previous iteration, Weight(s): [0.00793952 0.20224237], Gradient: [0.0012156 0.0060573]
It: 30. Loss: 0.648319493 lower by 5.5e-08 than in previous iteration, Weight(s): [0.00748772 0.19982975], Gradient: [0.0001802 0.0006356]
It: 40. Loss: 0.648319397 lower by 6.3e-10 than in previous iteration, Weight(s): [0.00741936 0.19957687], Gradient: [2.44e-05 6.72e-05]
It: 50. Loss: 0.648319396 lower by 7.5e-12 than in previous iteration, Weight(s): [0.00740999 0.19955019], Gradient: [3.1e-06 7.1e-06]
It: 60. Loss: 0.648319396 lower by 9.1e-14 than in previous iteration, Weight(s): [0.00740877 0.19954735], Gradient: [4.e-07 8.e-07]

Converged at iteration 60., where the Loss (0.648319396111158) was lower by 9.1e-14 than in the prev it(0.648319396111249), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:

Last iteration: 60., Weight(s): [0.00740877 0.19954735], Loss: 0.6483193961111584 which was lower than in previous iteration
The lowest Loss 0.6483193961111584 was achieved in the 60. iteration with the weight(s) [0.00740877 0.19954735]





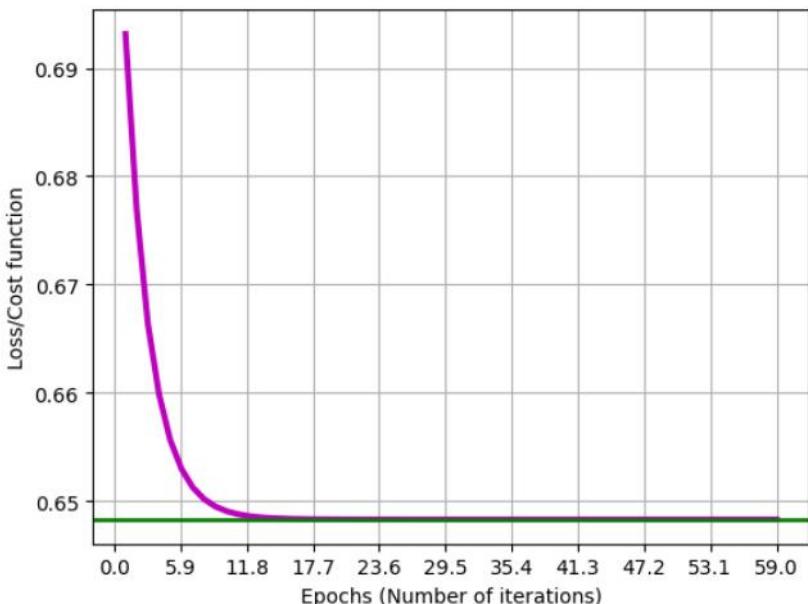
Accuracy on Train set: 0.9665
 Accuracy on Test set: 0.9615

Training initialized with the following hyperparameter settings:
 Learning rate: 0.8, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
 Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

```
It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.676906019 lower by 1.6e-02 than in previous iteration, Weight(s): [0.01468987 0.35900586], Gradient: [0.0176221 0.3574985]
It: 3. Loss: 0.666370645 lower by 1.1e-02 than in previous iteration, Weight(s): [0.00059219 0.07300707], Gradient: [-0.0164014 -0.2845005]
It: 4. Loss: 0.659829209 lower by 6.5e-03 than in previous iteration, Weight(s): [0.01371333 0.30060744], Gradient: [0.0149793 0.2267572]
It: 5. Loss: 0.655611274 lower by 4.2e-03 than in previous iteration, Weight(s): [0.00172986 0.11920165], Gradient: [-0.0134463 -0.1806002]
It: 10. Loss: 0.649081701 lower by 4.3e-04 than in previous iteration, Weight(s): [0.0103301 0.22542959], Gradient: [0.0067786 0.0581345]
It: 20. Loss: 0.648327882 lower by 4.8e-06 than in previous iteration, Weight(s): [0.00793952 0.20224237], Gradient: [0.0012156 0.0060573]
It: 30. Loss: 0.648319493 lower by 5.5e-08 than in previous iteration, Weight(s): [0.00748772 0.19982975], Gradient: [0.0001802 0.0006356]
It: 40. Loss: 0.648319397 lower by 6.3e-10 than in previous iteration, Weight(s): [0.00741936 0.19957687], Gradient: [2.44e-05 6.72e-05]
It: 50. Loss: 0.648319396 lower by 7.5e-12 than in previous iteration, Weight(s): [0.00740999 0.19955019], Gradient: [3.1e-06 7.1e-06]
It: 60. Loss: 0.648319396 lower by 9.1e-14 than in previous iteration, Weight(s): [0.00740877 0.19954735], Gradient: [4.e-07 8.e-07]
```

Converged at iteration 60., where the Loss (0.648319396111158) was lower by 9.1e-14 than
 in the prev it(0.648319396111249), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
 Last iteration: 60., Weight(s): [0.00740877 0.19954735], Loss: 0.6483193961111584 which was lower than in previous iteration
 The lowest Loss 0.6483193961111584 was achieved in the 60. iteration with the weight(s) [0.00740877 0.19954735]



Accuracy on Train set: 0.96625
 Accuracy on Test set: 0.962

Training initialized with the following hyperparameter settings:

Learning rate: 0.9, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20, Initial weight(s) randomly from the interval: [0, 0], Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]

It: 2. Loss: 0.695236876 higher by 2.1e-03 than in previous iteration, Weight(s): [0.0165261 0.40388159], Gradient: [0.022074 0.4578133]

It: 3. Loss: 0.696953572 higher by 1.7e-03 than in previous iteration, Weight(s): [-0.00334049 -0.00815036], Gradient: [-0.0259487 -0.4671244]

It: 4. Loss: 0.699242572 higher by 2.3e-03 than in previous iteration, Weight(s): [0.0200133 0.41226159], Gradient: [0.0299457 0.476552]

It: 5. Loss: 0.701106195 higher by 1.9e-03 than in previous iteration, Weight(s): [-0.00693784 -0.01663517], Gradient: [-0.0341157 -0.486246]

It: 10. Loss: 0.713583516 higher by 3.0e-03 than in previous iteration, Weight(s): [0.03211921 0.43950658], Gradient: [0.0572353 0.5374363]

It: 20. Loss: 0.747414831 higher by 4.7e-03 than in previous iteration, Weight(s): [0.05848048 0.49238784], Gradient: [0.1164904 0.6554187]

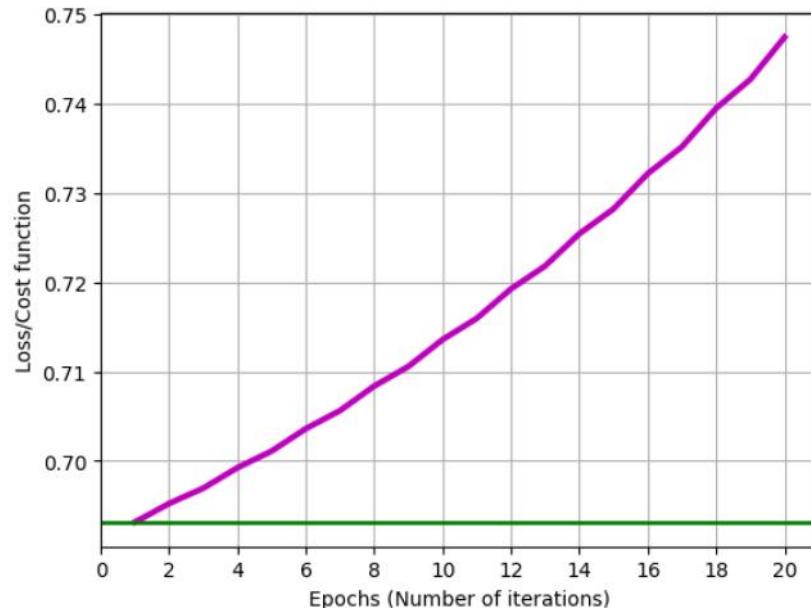
Early stopped at iteration 21., where the patience limit (20) was reached

The lowest Loss 0.693147180559945 was achieved in the 1. iteration with the weight(s) [0. 0.], and after that there were 20 iterations where th

Training finished with the following results:

Last iteration: 21., Weight(s): [-0.04636087 -0.09748896], Loss: 0.7508948125722171 which was higher than in previous iteration

The lowest Loss 0.693147180559945 was achieved in the 1. iteration with the weight(s) [0. 0.]



Accuracy on Train set: 0.499875

Accuracy on Test set: 0.504

```
# Comparing performance to sklearn's model
```

```
X, y = datasets.make_classification(n_samples=350, n_features=2, n_redundant = 0, n_classes = 2, random_state=11)
# Note: to display multiple features the plots below are not properly set
```

```
num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n")
```

```
# Dataset plotting
```

```
plt.figure(figsize=(6,5))
```

```

# Comparing performance to sklearn's model

X, y = datasets.make_classification(n_samples=350, n_features=2, n_redundant = 0, n_classes = 2, random_state=11)
# Note: to display multiple features the plots below are not properly set

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n")

# Dataset plotting
plt.figure(figsize=(6,5))
plt.grid()
plt.title('Generated dataset', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X[y==1][:, 0], X[y==1][:, 1], c = "gold", label="1")
plt.scatter(X[y==0][:, 0], X[y==0][:, 1], c = "indigo", label="0")
plt.legend(loc="upper left")
plt.show()

# Dataset splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
print("After splitting the dataset:")
print("Shape of x_train:", X_train.shape, "Shape of y_train:", y_train.shape, "Shape of x_test:", X_test.shape, "Shape of y_test:", y_test.shape)

print("\n\n\n")
print("----- Created LogRegNumPy without training -----")
# Performance of NumPy model without training (0 as initial=final value(s) for the weight(s))

regressorNoTrain = LogRegNumPy(max_iter=0,
                               initialweights_lowerlimit = 0,
                               initialweights_upperlimit = 0)

fittedRegressorNoTrain = regressorNoTrain.fit(X_train, y_train)
y_pred = fittedRegressorNoTrain.predict(X_test)
y_TrainPred = fittedRegressorNoTrain.predict(X_train)

TrainAccuracy_LogRegNumPyNoTrain = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegNumPyNoTrain = accuracy_score(y_pred, y_test)
print("Number of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])
print()
print("Ratio of y=1 in y_train:", len(y_train[y_train==1])/(len(y_train)))
print("Ratio of y=1 in y_test:", len(y_test[y_test==1])/len(y_test))

print("\nAs the weight(s) are initialized to zero(s) and there is no training, weight(s) = 0 is/are used to make predictions")
print("With zero weights(s) the Sigmoid function returns exactly 0.5, thus all predictions are y = 1")

fig, ax = plt.subplots()
fig.set_figheight(4)
fig.set_figwidth(15)
ax.set_facecolor("lightgray")

ax.scatter(X_test[y_pred==y_test][:,0], y_pred[y_pred==y_test], c = 'green', label = 'Correct predictions on the test set')
ax.scatter(X_test[y_pred!=y_test][:,0], y_pred[y_pred!=y_test], c = 'red', label = 'Wrong predictions on the test set')

ax.set_title('LogRegNumPy without training', fontsize = 15)
ax.legend(loc = 'center left', fontsize = '11')
ax.set_yticks(np.linspace(0, 1, 2))
ax.set_xlabel('x1 (First feature)', fontsize = 13)
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.43, 0.5, f'Train accuracy: {round(TrainAccuracy_LogRegNumPyNoTrain,5)}\n Test accuracy: {round(TestAccuracy_LogRegNumPyNoTrain,5)}', )
plt.show()

# pred_0 = X_test[y_pred==0]
# pred_0_set = set([tuple(x) for x in A])

```

```

#-----#
#-----#
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.43, 0.5, f'Train accuracy: {round(TrainAccuracy_LogRegNumPyNoTrain,5)}\n Test accuracy: {round(TestAccuracy_LogRegNumPyNoTrain,5)}', fontsize = '11.5', ha='left', va='center')
plt.show()

# pred_0 = X_test[y_pred==0]
# pred_0_set = set([tuple(x) for x in A])

# correct_pred = X_test[y_pred == y_test]
# correct_pred_set = set([tuple(x) for x in correct_pred ])

# pred_0_correct = np.array([x for x in pred_0_set & correct_pred_set])
# .....

plt.figure(figsize=(6,5))
plt.grid()
plt.title('LogRegNumPy without training', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X_test[y_pred==y_test][:,0], X_test[y_pred==y_test][:,1], c = 'green', label = 'Correct predictions on test set')
plt.scatter(X_test[y_pred!=y_test][:,0], X_test[y_pred!=y_test][:,1], c = 'red', label = 'Wrong predictions on test set')
plt.legend(loc="lower right", fontsize = "small")
plt.show()

print("\n\n\n")
print("----- Created LogRegNumPy with training "
      "\n-----\n")

# Performance of LogRegNumPy with training
# Set LogRegNumPy regressor's parameters to sklearn regressor's default values for the parameters
regressorSklearn = sklearn.linear_model.LogisticRegression()
defaultSkParams = regressorSklearn.__dict__

# For a proper comparison with sklearn's built-in model the weights are initialized to 0 here (for the created) class
# as well
# With weights initialized between 0 and 1 sometimes (depends on the final weights set at the end of the training)
# we get better results: 0.92857 Test Accuracy
regressor = LogRegNumPy(penalty = defaultSkParams['penalty'],
                       lambda_or_inverseC = defaultSkParams['C'],
                       max_iter = defaultSkParams['max_iter'],
                       tol = defaultSkParams['tol'],
                       initialweights_lowerlimit = 0,
                       initialweights_upperlimit = 0)

fittedRegressor = regressor.fit(X_train, y_train)
y_pred = fittedRegressor.predict(X_test)
y_TrainPred = fittedRegressor.predict(X_train)

TrainAccuracy_LogRegNumPy = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegNumPy = accuracy_score(y_pred, y_test)
print("Number of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])

fig, ax = plt.subplots()
fig.set_figheight(4)
fig.set_figwidth(15)
ax.set_facecolor("lightgray")

ax.scatter(X_test[y_pred==y_test][:,0], y_pred[y_pred==y_test], c = 'green', label = 'Correct predictions on the test set')
ax.scatter(X_test[y_pred!=y_test][:,0], y_pred[y_pred!=y_test], c = 'red', label = 'Wrong predictions on the test set')

# Why the approach below (mean of features on the X-axis) can sometimes be problematic:
# when for two (or more) samples the mean of the features are close to each other, then these samples overlap
# so much on the X-axis that we see both of them as one point in the scatter plot:
# print(X_test[:,0].std())
# # vs
# print(np.mean(X_test, axis = 1).std())
# Normalization of X_test happened when the regressor was fitted, so we can take the mean of different features
# ax.scatter(np.mean(X_test[y_pred==y_test], axis = 1), y_pred[y_pred==y_test], c = 'green', label = "Correct")
# ax.scatter(np.mean(X_test[y_pred!=y_test], axis = 1), y_pred[y_pred!=y_test], c = 'red', label = "Wrong")
# print(np.sort(np.mean(X_test[y_pred!=y_test], axis = 1)))

```

```

# so much on the X-axis that we see both of them as one point in the scatter plot:
# print(X_test[:,0].std())
# # vs
# print(np.mean(X_test, axis = 1).std())
# Normalization of X_test happened when the regressor was fitted, so we can take the mean of different features
# ax.scatter(np.mean(X_test[y_pred==y_test], axis = 1), y_pred[y_pred==y_test], c = 'green', label = "Correct")
# ax.scatter(np.mean(X_test[y_pred!=y_test], axis = 1), y_pred[y_pred!=y_test], c = 'red', label = "Wrong")
# print(np.sort(np.mean(X_test[y_pred!=y_test], axis = 1)))

# Problem with the approach below: As both correctly and incorrectly predicted samples are plotted
# at the same time, incorrect predictions won't be in the foreground and thus sometimes they can't be
# seen because of the much more correctly predicted green data points hiding the red points
# from matplotlib.colors import ListedColormap
# ax.scatter(X_test[:,0], y_pred, c = (y_pred == y_test), cmap=ListedColormap(['red','green']))
# import matplotlib.patches as mpatches
# NotCorrectPatch = mpatches.Patch(color='red', label='Wrong test set predictions')
# CorrectPatch = mpatches.Patch(color='green', label='Correct test set predictions')
# ax.legend(handles=[CorrectPatch, NotCorrectPatch], loc='center left', frameon=False)
# print(np.sort(X_test[y_pred!=y_test][:,0]))

ax.set_title('LogRegNumPy with training', fontsize = 15)
ax.legend(loc = 'center left', fontsize = '11')
ax.set_yticks(np.linspace(0, 1, 2))
ax.set_xlabel('x1 (First feature)', fontsize = 13)
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.43, 0.5, f'Train accuracy: {round(TrainAccuracy_LogRegNumPy,5)}\n Test accuracy: {round(TestAccuracy_LogRegNumPy,5)}', fontsize = '11.5', ha='left', va='center')
plt.show()

plt.figure(figsize=(6,5))
plt.grid()
plt.title('LogRegNumPy with training', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X_test[y_pred==y_test][:,0], X_test[y_pred==y_test][:,1], c = 'green', label = 'Correct predictions on test set')
plt.scatter(X_test[y_pred!=y_test][:,0], X_test[y_pred!=y_test][:,1], c = 'red', label = 'Wrong predictions on test set')
plt.legend(loc="lower right", fontsize = "small")
plt.show()

print("\n\n\n\n")
print("----- Scikit-Learn's built-in model -----")
# Performance of Scikit-Learn's built-in model (with training)
# Changing the solver (optimization method of the Loss function) to Stochastic Average Gradient (SAG), as
# this is the most similar method to Gradient Descent implemented in the LogRegNumPy class
# Stochastic Gradient Descent (SGD) is a variation of GD that updates the model parameters based on the gradient of the
# cost function for a single training example, rather than the average gradient over the entire training set
# Stochastic naming: in each training step the gradient is only an approximation of the actual gradient as it is
# calculated based on only one training sample
# Stochastic Average Gradient (SAG) updates the parameters based on the average of the gradients for each data point

# Based on ChatGPT (nowhere else I could find information about this) the initial weight(s) for
# the SAG solver are either zero(s) or (a) close to zero value(s)
# This leads to proper comparison as for the created class zero weights were initialized as well

# fit_intercept set to False as with the created class no intercept is used either
regressorSklearn = sklearn.linear_model.LogisticRegression(solver = 'sag',
                                                          fit_intercept = False,
                                                          n_jobs = 1,
                                                          verbose = True)

y_pred = regressorSklearn.fit(X_train, y_train).predict(X_test)
y_TrainPred = regressorSklearn.fit(X_train, y_train).predict(X_train)

TrainAccuracy_LogRegSklearn = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegSklearn = accuracy_score(y_pred, y_test)
print("\nNumber of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])

fig, ax = plt.subplots()
fig.set_figheight(4)

```

```

y_TrainPred = regressorSklearn.fit(X_train, y_train).predict(X_train)

TrainAccuracy_LogRegSklearn = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegSklearn = accuracy_score(y_pred, y_test)
print("\nNumber of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])

fig, ax = plt.subplots()
fig.set_figheight(4)
fig.set_figwidth(15)
ax.set_facecolor("lightgray")

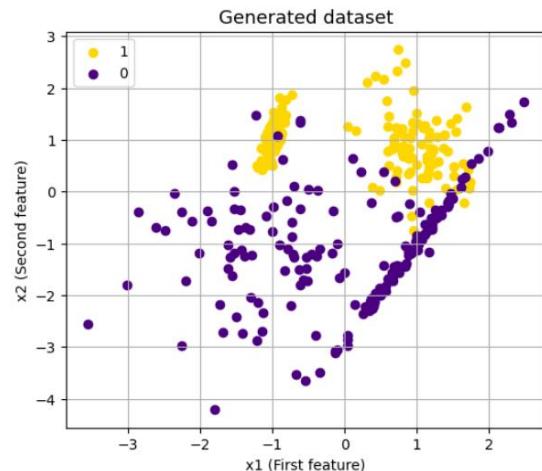
ax.scatter(X_test[y_pred==y_test][:,0], y_pred[y_pred==y_test], c = 'green', label = 'Correct predictions on the test set')
ax.scatter(X_test[y_pred!=y_test][:,0], y_pred[y_pred!=y_test], c = 'red', label = 'Wrong predictions on the test set')

ax.set_title('LogisticRegression from sklearn', fontsize = 15)
ax.legend(loc = 'center left', fontsize = 11)
ax.set_yticks(np.linspace(0, 1, 2))
ax.set_xlabel('x1 (First feature)', fontsize = 13)
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.18, 0.5, f'      Train accuracy: {round(TrainAccuracy_LogRegSklearn,5)}\nTrain built-in accuracy: {round(regressorSklearn.score(X_train, y_train), 5)}\n', color='red', fontweight='bold')
ax.text(1.18, 0.6, f'      Test accuracy: {round(TestAccuracy_LogRegSklearn,5)}\nTest built-in accuracy: {round(regressorSklearn.score(X_test, y_test), 5)}\n', color='blue', fontweight='bold')
plt.show()

plt.figure(figsize=(6,5))
plt.grid()
plt.title('LogisticRegression from sklearn', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X_test[y_pred==y_test][:,0], X_test[y_pred==y_test][:,1], c = 'green', label = 'Correct predictions on test set')
plt.scatter(X_test[y_pred!=y_test][:,0], X_test[y_pred!=y_test][:,1], c = 'red', label = 'Wrong predictions on test set')
plt.legend(loc="lower right", fontsize = "small")
plt.show()

```

Number of generated samples: 350 Number of generated features: 2



After splitting the dataset:
Shape of x_train: (280, 2) Shape of y_train: (280,) Shape of x_test: (70, 2) Shape of y_test: (70,)

x1 (First feature)

```
After splitting the dataset:  
Shape of x_train: (280, 2) Shape of y_train: (280,) Shape of x_test: (70, 2) Shape of y_test: (70,)
```

```
----- Created LogRegNumPy without training -----
```

```
There was no training performed as maximum iterations for the training are set to 0
```

```
Number of wrong predictions on the test set: 29
```

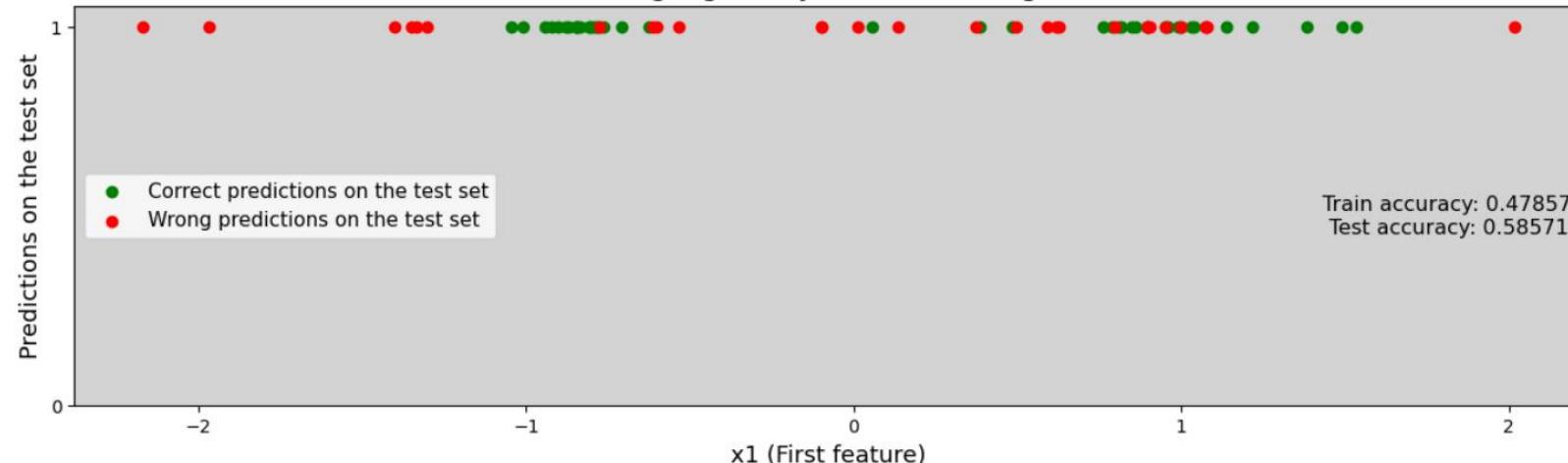
```
Ratio of y=1 in y_train: 0.4785714285714286
```

```
Ratio of y=1 in y_test: 0.5857142857142857
```

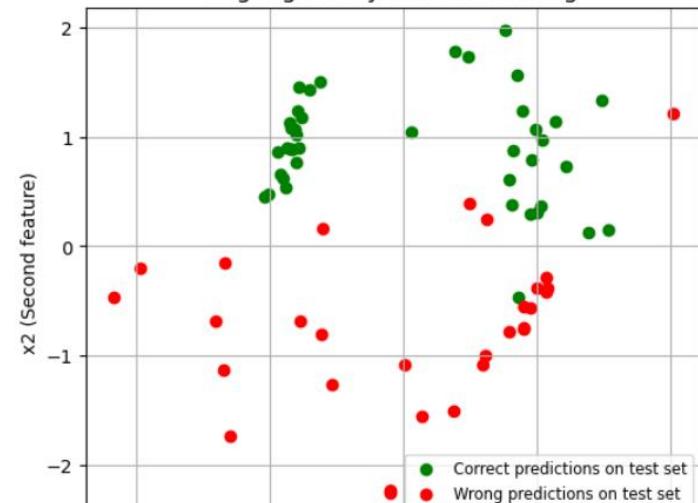
```
As the weight(s) are initialized to zero(s) and there is no training, weight(s) = 0 is/are used to make predictions
```

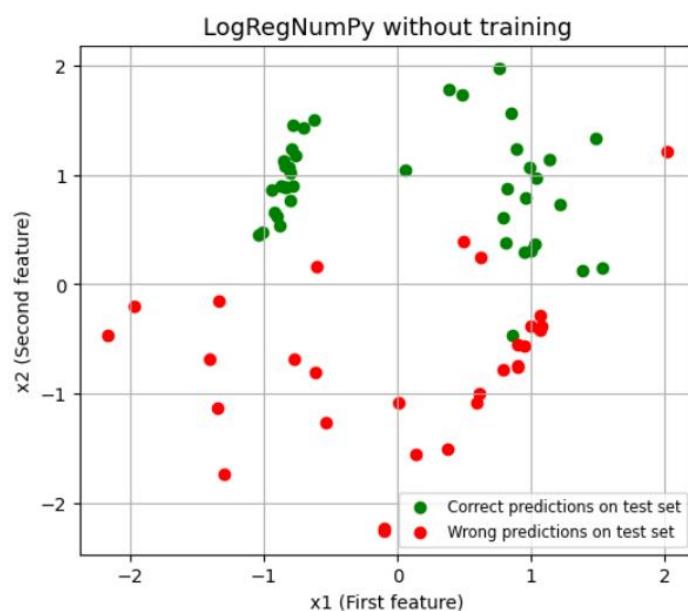
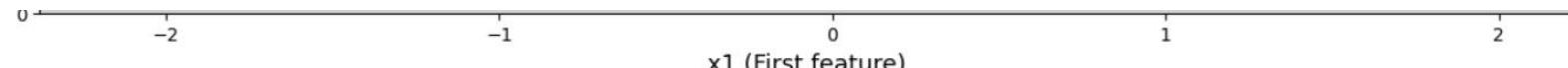
```
With zero weights(s) the Sigmoid function returns exactly 0.5, thus all predictions are y = 1
```

LogRegNumPy without training



LogRegNumPy without training





----- Created LogRegNumPy with training -----

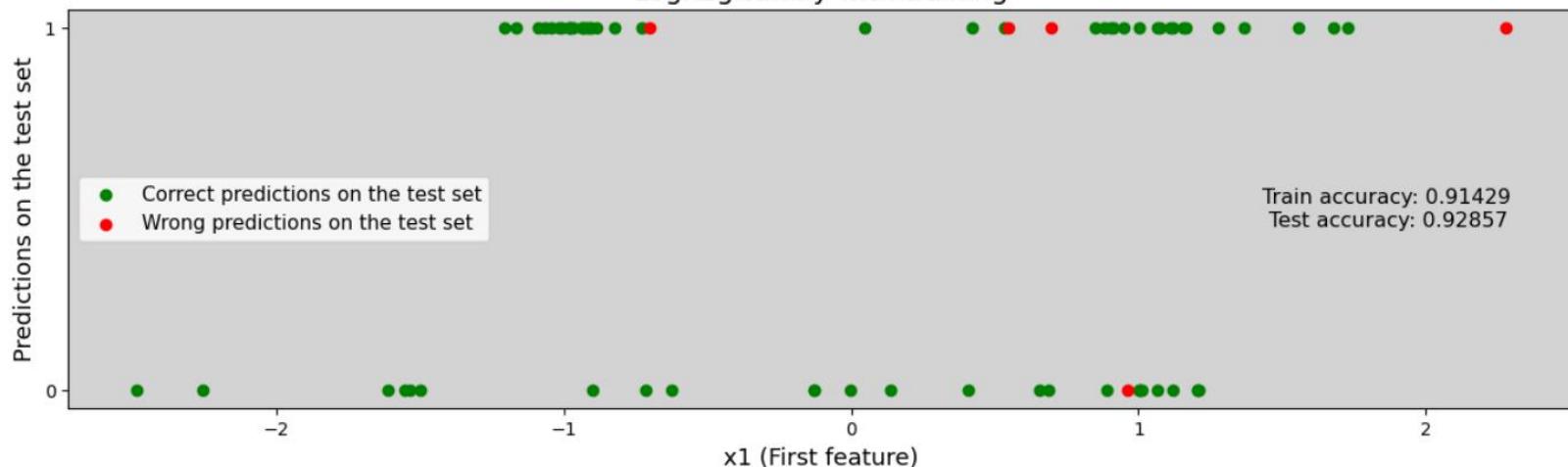
Training initialized with 100 maximum possible iterations

Converged at iteration 12, where the drop in Loss was below the tolerance set for convergence

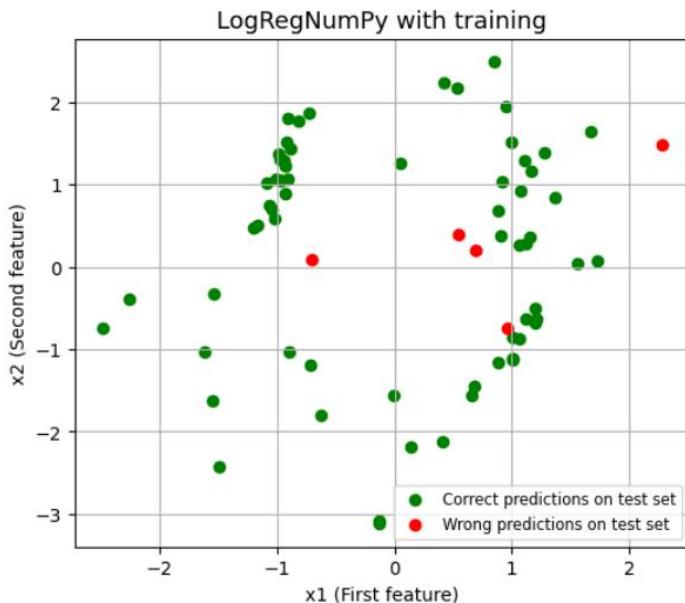
Training finished after 12 iterations

Number of wrong predictions on the test set: 5

LogRegNumPy with training



x1 (First feature)



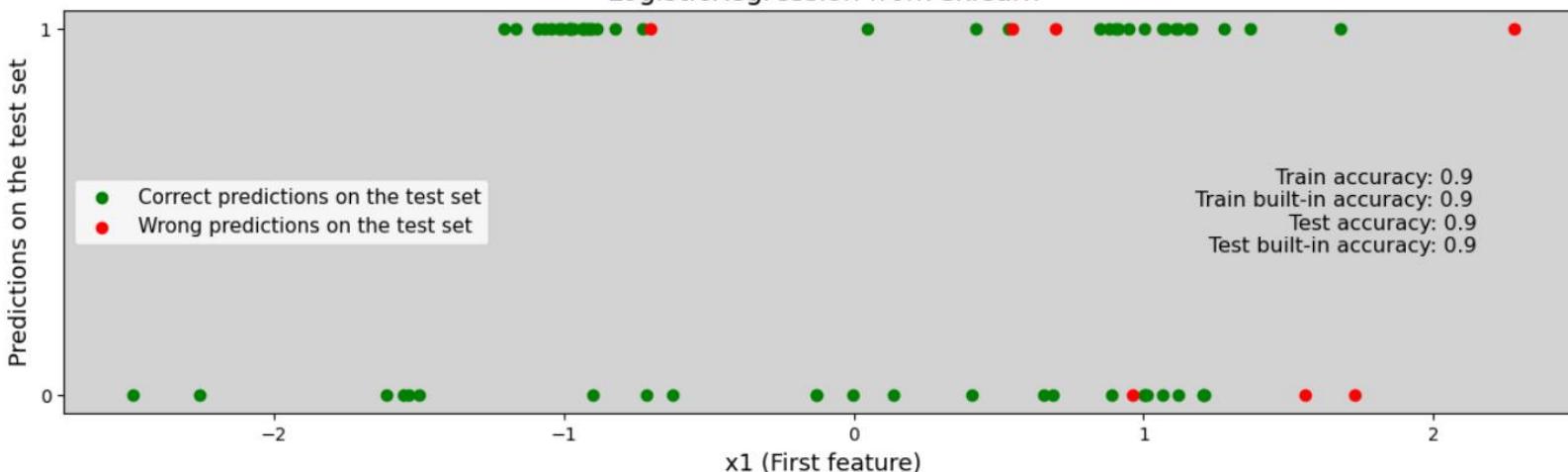
----- Scikit-Learn's built-in model -----

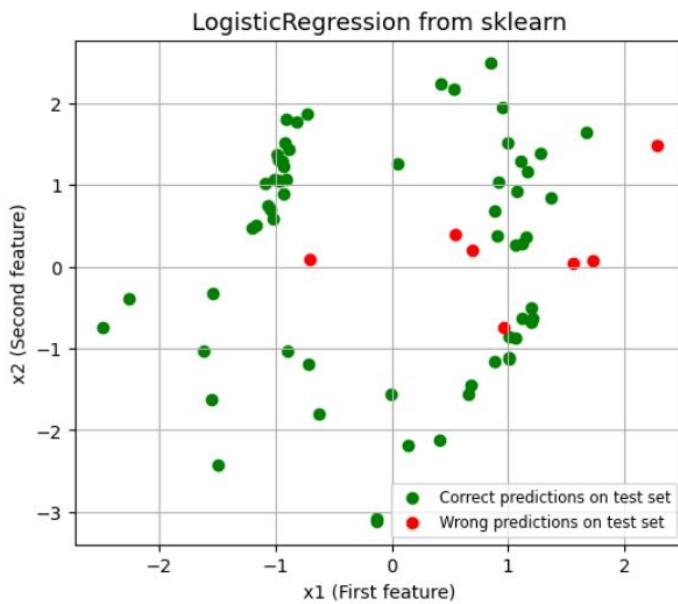
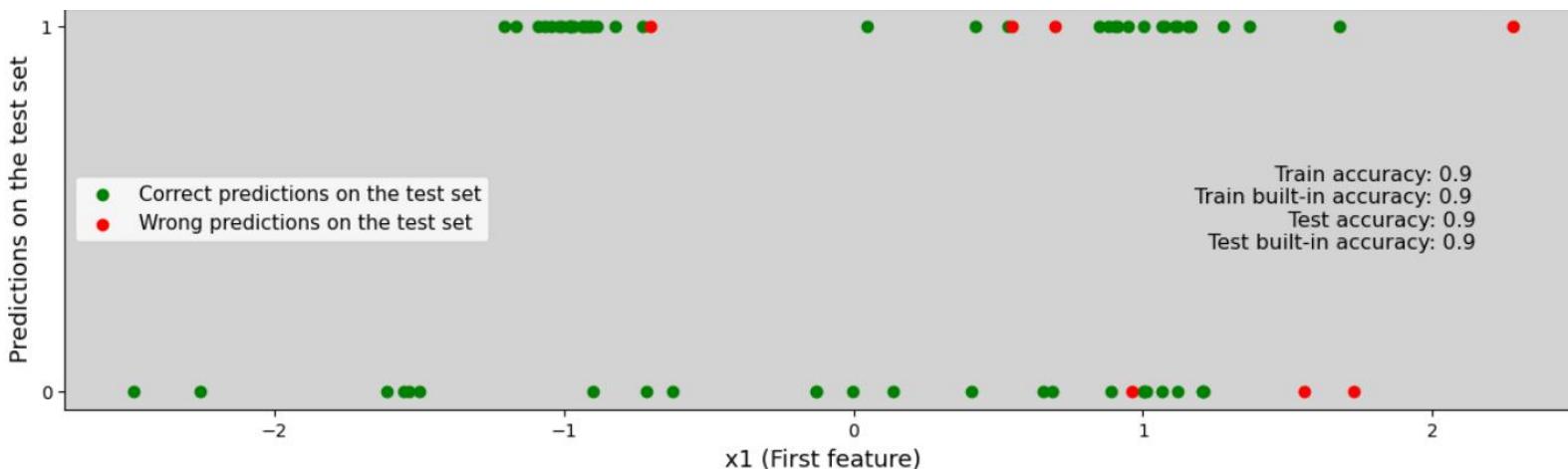
```
convergence after 19 epochs took 0 seconds
convergence after 22 epochs took 0 seconds
```

Number of wrong predictions on the test set: 7

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:  0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:  0.0s finished
```

LogisticRegression from sklearn





```
# Further considerations and todos
# Use functions instead of many duplicated lines of code in cells where possible
# Local minimumba érést összehozni (nagyon nagy kezdő súlyok -> sok és optimálistól messze kezdődő iteráció ami alatt egyszer lehet local minimumba ér)
# és ez a local minimumba érés jelenten meg platon is, lehet h sok feature (súly) kell hozzá hogy elég komplex
# legyen a loss function hogy legyen local minimuma is
# nyilak javítása a plotokon
# Mindegyik reviewed notebookbol minden ami más mint itt átrakni
```

Evaluation metrics

Evaluation metrics

Creating and applying class to a classification task

```
class ClassificationMetrics(BaseEstimator, TransformerMixin):
    """
    A class to calculate and plot evaluation metrics for a given model.

    Parameters
    -----
    best_threshold : float, optional
        The threshold value to be used for converting probabilities to class labels, by default None
    best_threshold_func : callable, optional
        The function to be used for calculating the best threshold, by default f1_score
    model_training : object
        The trained model object.

    Attributes
    -----
    best_threshold : float
        The best threshold value calculated based on the given function.
    best_threshold_func : callable
        The function to be used for calculating the best threshold.
    model : object
        The trained model object.
    model_spec : str
        The specifications of the trained model.
    """

    def __init__(self,
                 best_threshold: Optional[float] = None,
                 best_threshold_func: Callable = f1_score,
                 model_training: Optional[Any] = None,
                 ):
        #self.best_threshold = best_threshold
        self.best_threshold_func = best_threshold_func
        #self.model = model_training.model
        #self.model_spec = model_training.model_spec
        self.cumulative_lift_values = None
        self.cumulative_gain_values = None
        self.lift_gain_dict = None
        self.y_true = None
        self.y_pred = None

    def calculate_metrics(self,
                         y_true: np.ndarray,
                         y_pred: np.ndarray,
                         model_name: str,
                         dataset_name: str,
                         num_bins: int = 10):

        self.best_threshold = threshold_finder(y_true, y_pred, self.best_threshold_func)
        self.prec_rec_f1_auc = self.precision_recall_f1_auc(y_true, y_pred, model_name, dataset_name)
        self.cumulative_lift_values, self.cumulative_gain_values, self.lift_gain_table = self.lift_gain(y_true, y_pred, num_bins)
        self.classification_report = self.calc_classification_report(y_true, y_pred, dataset_name)
        self.confusion_matrix = self.calc_confusion_matrix(y_true, y_pred)
        self.y_true = y_true
        self.y_pred = y_pred

    return self #self.prec_rec_f1_auc, self.cumulative_lift_values, self.cumulative_gain_values, self.lift_gain_table, self.classification_report, self.confusion_matrix

def calc_classification_report(self,
                               y_true: np.ndarray,
                               y_pred: np.ndarray,
                               dataset_name: str):
    y_pred_bin = to_labels(y_pred, self.best_threshold)
    cr = pd.DataFrame(classification_report(y_true, y_pred_bin,
                                             target_names=['Class 0', 'Class 1'],
                                             output_dict=True))
    self.classification_report = cr
```

```

    return self #self.prec_rec_f1_auc, self.cumulative_lift_values, self.cumulative_gain_values, self.lift_gain_table, self.classification_report, self.confusion_matrix

def calc_classification_report(self,
                               y_true: np.ndarray,
                               y_pred: np.ndarray,
                               dataset_name):
    y_pred_bin = to_labels(y_pred, self.best_threshold)
    cr = pd.DataFrame(classification_report(y_true, y_pred_bin,
                                              target_names=['Class 0', 'Class 1'],
                                              output_dict=True)).transpose()
    df_name = pd.DataFrame(columns=cr.columns)
    df_name.loc['Dataset'] = [dataset_name for i in df_name.columns]
    cr = pd.concat([df_name, cr])
    return cr

def calc_confusion_matrix(self,
                          y_true: np.ndarray,
                          y_pred: np.ndarray):
    y_pred_bin = to_labels(y_pred, self.best_threshold)
    conf_matrix = confusion_matrix(y_true, y_pred_bin, labels=[0, 1]) #labels=self.model.classes_) # -----
    conf_df = pd.DataFrame(conf_matrix, index=['Actual Negative', 'Actual Positive'],
                           columns=['Predicted Negative', 'Predicted Positive'])

    return conf_df

def precision_recall_f1_auc(self,
                            y_true: np.ndarray,
                            y_pred: np.ndarray,
                            model_name: str,
                            dataset_name: str) -> pd.DataFrame:
"""
Calculate precision, recall, f1 score and auc for the given true and predicted values.

Parameters
-----
y_true : np.ndarray
    The true values.
y_pred : np.ndarray
    The predicted values.
dataset_name : str
    The name of the dataset.

Returns
-----
pd.DataFrame
    A dataframe containing the calculated metrics.
"""

    labels = to_labels(y_pred, self.best_threshold)
    pr = precision_score(y_true, labels)
    rec = recall_score(y_true, labels)
    f1 = f1_score(y_true, labels)
    auc = roc_auc_score(y_true, y_pred)
    current_metrics = [round(pr, 3), round(rec, 3), round(f1, 3), round(auc, 3)]
    metrics_names = ['Model', 'Dataset', 'precision', 'recall', 'f1-score', 'AUC']
    prec_rec_f1_auc = pd.DataFrame([[model_name] + [dataset_name] + current_metrics], columns=metrics_names)
    return prec_rec_f1_auc

def lift_gain(self,
              y_true: np.ndarray,
              y_pred: np.ndarray,
              num_bins: int) -> (List[float], List[float], dict):
"""
Calculate lift and gain for the given true and predicted values.

Parameters
-----
y_true : np.ndarray
    The true values.

```

```

def lift_gain(self,
             y_true: np.ndarray,
             y_pred: np.ndarray,
             num_bins: int) -> (List[float], List[float], dict):
"""
Calculate lift and gain for the given true and predicted values.

Parameters
-----
y_true : np.ndarray
    The true values.
y_pred : np.ndarray
    The predicted values.
num_bins : int
    The number of bins to be used for calculation.

Returns
-----
cumulative_lift_values, cumulative_gain_values, lift_gain_dict: Tuple[List[float], List[float]]
"""

data = np.column_stack((y_true, y_pred))
data = data[data[:, 1].argsort()[:-1]]
total_positives = np.sum(y_true)
bin_positives = np.zeros(num_bins).astype('int')
cumulative_lift_values = []
cumulative_gain_values = []

lift_gain_dict = {"Bin (Quantile)": [],
                 "No. of observations in bin": [],
                 "Probability upper limit": [],
                 "Probability lower limit": [],
                 "No. of all positive obs. divided equally into bins (A)": [],
                 "No. of positive observations in bin (B)": [],
                 "Cumulative of A (Cum. A)": [],
                 "Cumulative of B (Cum. B)": [],
                 "Lift (cumulative) (Cum. B/Cum. A)": [],
                 "Gain (Cum. B/ $\Sigma$ (A))": []}

lift_gain_dict["Bin (Quantile)"].append(str(i+1))
lift_gain_dict["No. of observations in bin"].append(len(bin_data))
lift_gain_dict["Probability upper limit"].append(data[start, 1])
lift_gain_dict["Probability lower limit"].append(data[end-1, 1])
lift_gain_dict["No. of all positive obs. divided equally into bins (A)"].append(bin_equal_positives)
lift_gain_dict["No. of positive observations in bin (B)"].append(bin_positives[i])
lift_gain_dict["Cumulative of A (Cum. A)"].append(bin_equal_positives_cum)
lift_gain_dict["Cumulative of B (Cum. B)"].append(bin_positives_cum)
lift_gain_dict["Lift (cumulative) (Cum. B/Cum. A)"].append(lift_cum)
lift_gain_dict["Gain (Cum. B/ $\Sigma$ (A))"].append(gain)

lift_gain_df = pd.DataFrame.from_dict(lift_gain_dict)

return cumulative_lift_values, cumulative_gain_values, lift_gain_df

```

```

lift_gain_df = pd.DataFrame.from_dict(lift_gain_dict)

return cumulative_lift_values, cumulative_gain_values, lift_gain_df

class VisualizeMetrics(BaseEstimator, TransformerMixin):

    def __init__(self,
                 best_threshold: Optional[float] = None,
                 func: Callable = f1_score,
                 model_training: Any = None,
                 if_plot: bool = True):
        self.best_threshold = best_threshold
        self.func = func
    #     self.model = model_training.model
    #     self.model_spec = model_training.model_spec

    def roc_auc_plot(self, y_true, y_pred, ax):
        # ROC AUC
        fpr, tpr, thresholds = roc_curve(y_true, y_pred)
        ax[0].plot([0,0,1], [0,1,1], color = 'darkgreen', linestyle='--', linewidth = "1.5", label = 'ROC (Perfect)')
        ax[0].plot([0, 1], [0, 1], color = 'red', linestyle='--', linewidth = "1.5", label = 'ROC (No Skill)') #Baseline/No Skill/Random
        sns.lineplot(x=fpr, y=tpr, label='ROC (Model)', linewidth = "3", ax=ax[0])
        auc = roc_auc_score(y_true, y_pred)
        ax[0].text(0.565, 0.28, f'Area Under Curve: {round(auc,3)}', bbox={'facecolor': 'white', 'edgecolor': (31/255,119/255,180/255), 'alpha': 0.8, 'pad': 0.5, 'linewidth': 1.5, 'boxstyle': 'roundtooth'}, color='black', ha='left', va='center')
        ax[0].set_ylim(0-0.04, 1.0 + 0.06)
        ax[0].set_title('ROC AUC Curve')
        ax[0].set_xlabel('False Positive Rate (FPR)')
        ax[0].set_ylabel('True Positive Rate (TPR)')
        ax[0].legend(loc='lower right')
        ax[0].grid()

    def precision_recall_plot(self, y_true, y_pred, ax):
        precision, recall, thresholds = precision_recall_curve(y_true, y_pred)
        class_balance = len(y_true[y_true == 1])/len(y_true)
        ax[1].plot([0,1,1], [1,1,class_balance], color = 'darkgreen', linestyle='--', linewidth = "1.5", label = 'PR (Perfect)')
        ax[1].plot([0,1], [class_balance, class_balance], color = 'red', linestyle = '--', linewidth = "1.5", label = 'PR (No Skill) = Class balance')
        ax[1].plot(recall, precision, linewidth = "3", label = 'PR (Model)')
        ax[1].set_yticks(np.arange(0, 1.0+0.01, 0.2)) # step:0.1
        ax[1].set_xlim(0-0.04, 1.0 + 0.06)
        ax_secondary_y = ax[1].twinx()
        self._set_PR_noskill_tick(class_balance, ax_secondary_y)
        ax[1].set_xlabel('Recall (Sensitivity)')
        ax[1].set_ylabel('Precision')
        ax[1].set_title('Precision-Recall Curve')
        ax[1].legend(loc='best')
        ax[1].grid()
        plt.show()

    def lift_plot(self, lift, ax):
        ax[0].plot([1,len(lift)], [1.0, 1.0], color='red', linestyle='--', linewidth="1.5", label='Lift (No Skill)') #Baseline/No Skill/Random
        sns.lineplot(x=np.arange(1, len(lift) + 1), y=lift, marker='o', ax=ax[0], label = 'Lift (Model)')
        ax[0].set_xticks(np.linspace(1, len(lift), 10))
        self._set_lift_ticks(lift, ax[0])
        ticks = sorted(list(ax[0].get_yticks()))
        spacing_between_ticks = ticks[1]-ticks[0]
        ax[0].set_ylim(min(min(lift)-spacing_between_ticks*0.2, 1-0.04), max(lift) + 0.27*spacing_between_ticks)
        ax[0].set_title('Lift (cumulative) Chart')
        ax[0].set_xlabel('Bins (Quantiles)')
        ax[0].set_ylabel('Lift')
        ax[0].legend(loc='best')
        ax[0].grid()

    def gain_plot(self, gain, ax):
        ax[1].plot([0, len(gain)], [0, 1], color = 'red', linestyle='--', linewidth = "1.5", label = 'Gain (No Skill)') #Baseline/No Skill/Random
        sns.lineplot(x=np.arange(0, len([0] + gain)), y=[0] + gain, marker='o', ax=ax[1], label = 'Gain (Model)')
        ax[1].set_xticks(np.linspace(0, len(gain), 11)) # (1, len(gain), 10)
        ax[1].set_yticks(np.arange(0, 1.0+0.01, 0.2)) # step:0.1
        ax[1].set_xlim(0-0.04, 1+0.06)
        ax[1].set_title('Gain Chart')
        ax[1].set_xlabel('Bins (Quantiles)')

```

```

ax[0].grid()

def gain_plot(self, gain, ax):
    ax[1].plot([0, len(gain)], [0, 1], color = 'red', linestyle='--', linewidth = "1.5", label = 'Gain (No Skill)') #Baseline/No Skill/Random
    sns.lineplot(x=np.arange(0, len([0] + gain)), y=[0] + gain, marker='o', ax=ax[1], label = 'Gain (Model)')
    ax[1].set_xticks(np.linspace(0, len(gain), 11)) #(0, len(gain), 10)
    ax[1].set_yticks(np.arange(0, 1.0+0.01, 0.2)) # step:0.1
    ax[1].set_xlim(0-0.04, 1+0.06)
    ax[1].set_title('Gain Chart')
    ax[1].set_xlabel('Bins (Quantiles)')
    ax[1].set_ylabel('Gain (cumulative)')
    ax[1].legend(loc='lower right')
    ax[1].grid()
    plt.show()

def _set_PR_noskill_tick(self,
                         class_balance: float,
                         axis: Axes) -> None:
    axis.set_yticks([class_balance])
    plt.draw()

    class_balance_tick = axis.get_yticklabels()[0]
    class_balance_tick.set_color('red')

def _set_lift_ticks(self, lift_values: List[float], axis: Axes) -> None:
    ticks = axis.get_yticks()
    max_lift = round(max(lift_values),2)

    if 1 in ticks:
        ticks = sorted(list(ticks))
    else:
        ticks = sorted(list(ticks) + [1])

    del ticks[0]
    del ticks[-1]
    tick_distance = ticks[-1] - ticks[-2]
    if (max_lift - ticks[-1]) >= tick_distance/3:
        ticks+=[max_lift]
    else:
        ticks[-1] = max_lift

    axis.set_yticks(ticks)
    plt.draw()

    for tick in axis.get_yticklabels():
        if (tick.get_text() == f'{max_lift:.3f}' or tick.get_text() == f'{max_lift:.2f}' or tick.get_text() == str(max_lift)):
            tick.set_color('darkgreen')

def plot_charts(self, y_true, y_pred, lift, gain):
    self.best_threshold = threshold_finder(y_true, y_pred, self.func)
    y_pred_bin = to_labels(y_pred, self.best_threshold)
    fig, ax = plt.subplots(1, 2, figsize=(12, 4))
    self.roc_auc_plot(y_true, y_pred, ax)
    self.precision_recall_plot(y_true, y_pred_bin, ax)
    plt.show()

    fig, ax = plt.subplots(1, 2, figsize=(12, 4))
    self.lift_plot(lift, ax)
    self.gain_plot(gain, ax)
    plt.show()

y_true=y_test
y_pred=y_pred_test
dataset_name='test'

get_metrics = ClassificationMetrics(best_threshold_func=f1_score)
test_metrics = get_metrics.calculate_metrics(y_true=y_true, y_pred=y_pred, model_name='model', dataset_name=dataset_name)

```

```

plt.show()

y_true=y_test
y_pred=y_pred_test
dataset_name='test'

get_metrics = ClassificationMetrics(best_threshold_func=f1_score)
test_metrics = get_metrics.calculate_metrics(y_true=y_true, y_pred=y_pred, model_name='model', dataset_name=dataset_name)
display(test_metrics.prec_rec_f1_auc)
display(test_metrics.confusion_matrix)
display(test_metrics.classification_report)
display(test_metrics.lift_gain_table)

vm = VisualizeMetrics(func=f1_score) #, model_training=model_training)
vm.plot_charts(test_metrics.y_true, test_metrics.y_pred, test_metrics.cumulative_lift_values, test_metrics.cumulative_gain_values)

```

Best Threshold=0.420000, F-Score=0.916

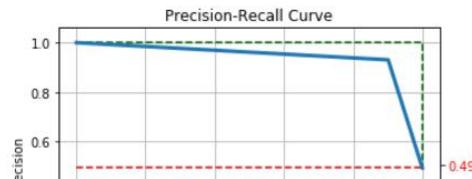
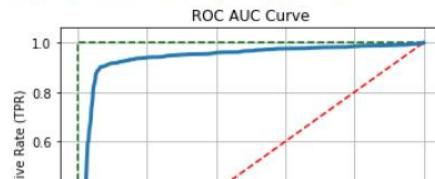
Model	Dataset	precision	recall	f1-score	AUC
0	model	test	0.93	0.901	0.916

	Predicted Negative	Predicted Positive
Actual Negative	1419	100
Actual Positive	146	1335

	f1-score	precision	recall	support
Dataset	test	test	test	test
Class 0	0.920233	0.906709	0.934167	1519
Class 1	0.915638	0.930314	0.901418	1481
micro avg	0.918	0.918	0.918	3000
macro avg	0.917936	0.918511	0.917793	3000
weighted avg	0.917965	0.918362	0.918	3000

Bin (Quantile)	No. of observations in bin	Probability upper limit	Probability lower limit	No. of all positive obs. divided equally into bins (A)	No. of positive observations in bin (B)	Cumulative of A (Cum. A)	Cumulative of B (Cum. B)	Lift (cumulative) (Cum. B/Cum. A)	Gain (Cum. B/Σ(A))
0	1	300	0.999966	0.992300	148.1	287	148.1	287	1.937880
1	2	300	0.992284	0.970449	148.1	281	296.2	568	1.917623
2	3	300	0.970272	0.913874	148.1	289	444.3	857	1.928877
3	4	300	0.913661	0.758299	148.1	278	592.4	1135	1.915935
4	5	300	0.758297	0.335426	148.1	221	740.5	1356	1.831195
5	6	300	0.334957	0.167566	148.1	46	888.6	1402	1.577763
6	7	300	0.167340	0.113082	148.1	22	1036.7	1424	1.373589
7	8	300	0.113027	0.090046	148.1	22	1184.8	1446	1.220459
8	9	300	0.090021	0.074310	148.1	14	1332.9	1460	1.095356
9	10	300	0.074225	0.004422	148.1	21	1481.0	1481	1.000000

Best Threshold=0.420000, F-Score=0.916



	f1-score	precision	recall	support						
Dataset	test	test	test	test						
Class 0	0.920233	0.906709	0.934167	1519						
Class 1	0.915638	0.930314	0.901418	1481						
micro avg	0.918	0.918	0.918	3000						
macro avg	0.917936	0.918511	0.917793	3000						
weighted avg	0.917965	0.918362	0.918	3000						

Bin (Quantile)	No. of observations in bin	Probability upper limit	Probability lower limit	No. of all positive obs. divided equally into bins (A)	No. of positive observations in bin (B)	Cumulative of A (Cum. A)	Cumulative of B (Cum. B)	Lift (cumulative) (Cum. B/Cum. A)	Gain (Cum. B/ $\Sigma(A)$)	
0	1	300	0.999966	0.992300	148.1	287	148.1	287	1.937880	0.193788
1	2	300	0.992284	0.970449	148.1	281	296.2	568	1.917623	0.383525
2	3	300	0.970272	0.913874	148.1	289	444.3	857	1.928877	0.578663
3	4	300	0.913661	0.758299	148.1	278	592.4	1135	1.915935	0.766374
4	5	300	0.758297	0.335426	148.1	221	740.5	1356	1.831195	0.915598
5	6	300	0.334957	0.167566	148.1	46	888.6	1402	1.577763	0.946658
6	7	300	0.167340	0.113082	148.1	22	1036.7	1424	1.373589	0.961512
7	8	300	0.113027	0.090046	148.1	22	1184.8	1446	1.220459	0.976367
8	9	300	0.090021	0.074310	148.1	14	1332.9	1460	1.095356	0.985820
9	10	300	0.074225	0.004422	148.1	21	1481.0	1481	1.000000	1.000000

Best Threshold=0.420000, F-Score=0.916

