Codeübersicht Data Scientist

1 Data Gathering

1.1 From Data Files

In [ ]:
```
import pandas as pd

# Import files in data frames
df = pd.read_csv('file_name.csv')
df = pd.read_excel('file_name.xlsx')
```

1.2 SQL Data Bases

In [ ]:
```
import sqlalchemy as sa

engine = sa.create_engine('sqlite:///DataBase.db')
inspector = sa.inspect(engine)

# Getting the names of the columns in the Data Base
inspector.get_table_names()

# Defining the connection to the Data Base
connection = engine.connect()
sql_query = ''' SELECT * FROM temperature_sensor_1 '''
df1 = pd.read_sql(sql_query, con=connection)
```

1.3 Web Scraping

In [ ]:
```
import sqlalchemy as sa

engine = sa.create_engine('sqlite:///DataBase.db')
inspector = sa.inspect(engine)

# Getting the names of the columns in the Data Base
inspector.get_table_names()

# Defining the connection to the Data Base
connection = engine.connect()
sql_query = ''' SELECT * FROM temperature_sensor_1 '''
df1 = pd.read_sql(sql_query, con=connection)
```

```
import requests
import lxml.html

# Creating a list of weblists to visit
website_url = 'https://homepage.org'
response = requests.get(website_url)


root = lxml.html.fromstring(response.text)
table = root.get_element_by_id('constituents')

links = []
for link in table.iter('a'):
    if not 'class' in link.attrib:
        links_wiki.append(link.attrib['href'])

# Visiting all websites from the list
base_url = 'https://homepage.org'
response = requests.get(base_url+links)
response.raise_for_status()

root = lxml.html.fromstring(response.text)
table = root.find_class('infobox vcard')[0]


table_names_all = []
table_values_all = []
```

```
from time import sleep # needed to add a delay between requests to avoid blocking by the web page

for link in links:
    response_tmp= requests.get(base_url+link)
    response_tmp.raise_for_status()
    table_names = [element.text_content() for element in table.iter('th')]
    table_values = [element.text_content() for element in table.iter('td') if not ('colspan' in element.attrib and element.
attrib['colspan']=='2')]
    table_names_all.append(table_names)
    table_values_all.append(table_values)
    delay = response.elapsed.total_seconds()
    sleep(5*delay)

dic_list = []  # create empty list
for table_names, table_values in zip(table_names_all, table_values_all):  # iterate through the data
    dic = dict(zip(table_names, table_values))  # use dict in combination with zip to get the key-value-pairs
    dic_list.append(dic)  # append the dictionary to the list

names = []

for i in range(len(links)):
    names.append(links[i].split('/')[-1])


# Storing the data in a Data Frame
import pandas as pd
df = pd.DataFrame(dic_list, index=names)
```

1.4 From PDF files

In [ ]:

```
import PyPDF2 #  Possibly needs to be installed manually, helps assessing content of PDFs

f_reader = open('FileName.pdf', 'rb') #  rb reads without interpretation (e.g. with utf 8)

pdf_reader = PyPDF2.PdfFileReader(f_reader)

# Getting number of pages of the document
pdf_reader.getNumPages()

# Selecting a page
pdf_page = pdf_reader.getPage(0)

# Extract text
pdf_str = pdf_page.extractText()

# Closing the file afterwards
f_reader.close()

# Remove new lines
pdf_str = pdf_str.replace('\n','')
```

1.5 Regular expressions

In [ ]:

```
import re

expression = 'String Name'  # define regular expression
re.findall(expression, pdf_str)  # look for regular expression in pdf_str

expr=r'\d\d\.\d\d\.\d\d\d\d'
re.findall(expr, pdf_str)

# Using Regular Expressions and pandas
df['Feature year']=df_company['Feature'].str.extract(r'(\d\d\d\d)',expand=False)

# Transforming number strings into numbers
col_list=['Feature1', 'Feature2']

for col in col_list:
    df[col+' year'] = df[col].str.extract(r'(\d{4})',expand=False)
    df[col+' value'] = df[col].str.extract(r'([\d.,]+\s?[a-zA-Z]*)',expand=False)
    df_company.loc[:, col+' value'] = df_company.loc[:, col+' value'].str.replace('\strillion','1e12')
    df_company.loc[:, col+' value'] = df_company.loc[:, col+' value'].str.replace('\sbillion','1e9')
    df_company.loc[:, col+' value'] = df_company.loc[:, col+' value'].str.replace('bn','1e9')
    df_company.loc[:, col+' value'] = df_company.loc[:, col+' value'].str.replace('\smillion','1e6')
    df_company.loc[:, col+' value'] = df_company.loc[:, col+' value'].str.replace(',','.')
    df_company.loc[:, col+' value'] = df_company.loc[:, col+' value'].astype(float)
```

2 Data Preparation

```
# Converting Dates to useful data type
[df.loc[:, 'date'] = pd.to_datetime(df.loc[:, 'date'])]

# Setting datatypes

df.loc[:, 'Column'] = df_train.loc[:, 'Column'].astype('DataType')

#extract hours, minutes, etc.. from a datetine object
df.iloc[0:3, 0].dt.minute #  minute can be replaced by appropriate time unit


# Data aggregation
groups = df.groupby('Feature') #  Grouping data based on a category
diction = {'Feature1':'sum','Feature2':'nunique'} #  Creating a dictionary with features and functions to be applied
df = groups.agg(diction) #  Aggreagate groups based on choices in diction



# Merging DataFrame
df = pd.merge(left=df_1, right=df_2,left_index=True,right_index=True,how='left')

# Transforming categorical data to ints
df.loc[:,'Column'].factorize()[0] #  Returns tupel with old symbol and new int. Setting [0] specifies the new value

list_cat = ['Categorical Feature1', 'Categorical Feature2']

for cat in list_cat:
    df_train.loc[:, cat] = df_train.loc[:, cat].replace({'Yes': 1, 'No': 0})
    df_test.loc[:, cat] = df_test.loc[:, cat].replace({'Yes': 1, 'No': 0})
    df_aim.loc[:, cat] = df_aim.loc[:, cat].replace({'Yes': 1, 'No': 0})
```

2.1 Outliers

2.1.1 RANSAC

In [ ]:

```
from sklearn.linear_model import RANSACRegressor

# RANSAC Regression
model_ransac = RANSACRegressor()
model_ransac.fit(features_train,target_train)
pred_ransac = model_ransac.predict(features_test)

# Outliers
model_ransac = RANSACRegressor(residual_threshold=1) #  Residual_threshold defines distance to regression curve
model_ransac.inlier_mask_ #  Boolean Mask of Data Points not considered outliers
```

stackfuel

In [ ]:

```
# Distance in standard deviations

mean = df.loc[:,'Feature'].mean()
std = df.loc[:,'Feature'].std()
value= df.loc[:,'Feature'].max()

distance_sd = abs(value - mean) / std

print(distance_sd)


# Counting Outliers
df_outliers_count = df.loc[:, ['Feature']]


distance_to_mean = df.loc[:,'Feature'] - df.loc[:,'Feature'].mean()
absolute_distance_to_mean = distance_to_mean.abs()
std_distance_to_mean = absolute_distance_to_mean / df_.loc[:,'Feature'].std()

# identify outliers
mask_outliers = std_distance_to_mean >= 3

# count outliers and add to DataFrame
df_outliers_count.loc[:, 'N_outliers'] = mask_outliers.sum()

# Visualizing Outliers
# modul import (in case it wasn't done before)
import seaborn as sns

# initialise figure and axes
fig, ax = plt.subplots()

# draw scatter plot
sns.scatterplot(x=list(range(60)),
                y=df_temp4.iloc[144, :-1],
                hue=model_ransac.inlier_mask_,
                ax=ax)

# optimise plot
ax.set(xlabel='Time since cycle start [sec]',
       ylabel='Temperature [°C]',
      title='Temperature sensor 4 data of hydraulic pump')
ax.legend(title='Inlier')
```

2.2 Feature Engineering

2.2.1 Polynomial Features

In [ ]:

```
from sklearn.preprocessing import PolynomialFeatures
# Creating a pipeline with polynomial regression
poly_transformer= PolynomialFeatures(degree=2,include_bias=False)
from sklearn.pipeline import Pipeline

pipeline = Pipeline([('poly',poly_transformer),('model',model)])
pipeline.fit(features,target)

# Cross Validation
from sklearn.model_selection import cross_val_score
cv_results=cross_val_score(estimator=pipeline,X=features,y=target,cv=5, scoring='neg_mean_absolute_error')
print(cv_results.mean())
```

2.2.2 PCA

In [ ]:

```
from sklearn.decomposition import PCA
model = PCA(n_components=1)
model.fit(arr)

model.components_  #  Stores the learned n_components principal components

model.explained_variance_ratio_


# PCA as part of a pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
import numpy as np
poly_transformer = PolynomialFeatures(degree=2,include_bias=False)
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import validation_curve

standardizer = StandardScaler()
pca=PCA()

pipeline = Pipeline([('poly',poly_transformer),('scale',standardizer),('pca',pca),('reg',model)])
train_scores,valid_scores = validation_curve(estimator=pipeline, X=features,y=target,param_name='pca__n_components', param_
range=range(1,50),cv=5,scoring='neg_mean_absolute_error')

train_scores_mean=np.mean(train_scores,axis=1)
valid_scores_mean=np.mean(valid_scores,axis=1)
import matplotlib.pyplot as plt
%matplotlib inline

fig, ax = plt.subplots(figsize=(8,5))
ax.plot(range(1,50), train_scores_mean, label='train')
ax.plot(range(1,50), valid_scores_mean, label='valid')
ax.hlines(y=-0.57717, xmin=1, xmax=50)
ax.set_xlabel('Number of components')
ax.set_ylabel('Negative mean absolute error')
ax.legend()

# Selecting number of components by information content
pca=PCA()
pca.fit(arr)

import numpy as np

rolling_sum_ratios = np.cumsum(pca.explained_variance_ratio_) #  cumsum is the cummulative sum
fig, ax = plt.subplots(figsize=[8,5])
plt.plot(range(1,len(rolling_sum_ratios)+1),rolling_sum_ratios)

# We can also specify the target explained variance by specifying PCA(n_components=x), where x is a value between 0 and 1
```

2.3 Natural Language Processing

In [ ]:

```
import string,re #  Used for string processing

import spacy,nltk #  NLP modules

# Doc
doc = nlp(df.loc[105, "msg"])

# Doc-Tokens
doc_tokens = [token.text for token in doc]
token_pos = [[token.text, token.pos_] for token in doc]

# Lemmatization
lemma_token = [token.lemma_ for token in doc]
doc = [token.lemma_ for token in doc if token.lemma_ != "-PRON-"] #  Removing pronouns

# Stopwords
stopWords = set(stopwords.words('english'))
doc = [token for token in doc if token not in stopWords ]

# Punctuation
punctuations = string.punctuation
doc = [token for token in doc if token not in punctuations]

# function for cleaning
# Definiere die Funktion `text_cleaner()` mit dem Parameter `sentence`
def text_cleaner(sentence):

    # Erstelle das Doc-Objekt `sentence` unter Verwendung von `nlp()`
    doc = nlp(sentence)
    # Lemmatisierung
    lemma_token = [token.lemma_ for token in doc]
    doc = [token.lemma_ for token in doc if token.lemma_ != "-PRON-"]

    # Stoppwort Entfernung
    stopWords = set(stopwords.words('english'))
    doc = [token for token in doc if token not in stopWords ]
    # Satzzeichen Entfernung
    punctuations = string.punctuation
    doc = [token for token in doc if token not in punctuations]

    # Wende die `my_str.join` Methode an, um die Liste zu einem string zusammenzufügen
    doc = " ".join(doc)
    # Verwende re.sub(), um multiple Punkte oder Leerzeichen '[\.\s]+' durch einzelne Leerzeichen ' ' zu ersetzen.
    doc  = re.sub('[\.\s]+', ' ', doc)
    # Ausgabe
    return doc

df['msg_clean'] = df['msg'].apply(text_cleaner)


# Vectorization
from sklearn.model_selection import train_test_split
features = df.loc[:,['msg_clean']]
target = df['status']
features_train, features_test, target_train, target_test = train_test_split(features, target, test_size = 0.3, random_state
= 1)
from sklearn.feature_extraction.text import CountVectorizer
count_vectorizer = CountVectorizer()
features_train_bow = count_vectorizer.fit_transform(features_train)
bow_features = count_vectorizer.get_feature_names()
bow_array = features_train_bow.toarray()
bow_vector = pd.DataFrame(bow_array, columns=bow_features)



# tfidf Vectorization
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()

features_train_tfidf = tfidf_vectorizer.fit_transform(features_train)
tfidf_features = tfidf_vectorizer.get_feature_names()

tfidf_vector = pd.DataFrame(features_train_tfidf.toarray(), columns = tfidf_features)
idf_values = tfidf_vec.idf_
```

stackfuel

2.4 Over- and Undersampling

In [ ]:
```
# Unbalanced data sets: undersampling

mask_minority_class = df_train.loc[:, 'Target'] == 1  # create mask to select minority class
len_minority = mask_minority_class.sum()  # count rows with minority class

df_train_minority = df_train.loc[mask_minority, :]  # select minority class with mask
df_train_majority = df_train.loc[~mask_minority, :]  # select majority class with inverted mask

df_train_majority_sample = df_train_majority.sample(n=len_minority, random_state=42)  # undersample majority class

df_train_balanced_by_undersampling = df_train_minority.append(df_train_majority_sample)  # combine minority class and under
sampled majority class


# Oversampling
mask_majority_class = df_train.loc[:, 'Target'] == 0  # create mask to select minority class
len_majority = mask_majority_class.sum()
df_train_minority_bootstrap = df_train_minority.sample(replace=True, random_state=42,n=len_majority)
df_train_balanced_by_oversampling = df_train_majority.append(df_train_minority_bootstrap)

pd.crosstab(df_train_balanced_by_oversampling['attrition'],'count') #  Check that set is balanced
```

3 Data Exploration

3.1 Seaborn

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Magic command to show the figures in the notebook

%matplotlib inline

# Scatter plots

sns.pairplot(df)
pd.plotting.scatter_matrix(df)

# Linear Regression plot
sns.regplot(x=df.loc[:,'Feature'],y=df.loc[:,'Target'])

# Strip, box and violin plots

fig,axs= plt.subplots(ncols=3,figsize=[12,6])
sns.stripplot(data=df_train, x='Feature1',y='Feature2',alpha=0.5, ax = axs[0])
sns.boxplot(x="Feature1", y="Feature2", data=df_train, ax=axs[1])
sns.violinplot(x="Feature1", y="Feature2", data=df_train, ax=axs[2])
fig.tight_layout()

# Regplots
import seaborn as sns
sns.regplot(x=features,
            y=target,
            scatter_kws={'color':'#17415f',  # dark blue dots
                         'alpha':1},  # no transparency for dots
            fit_reg=False)  # no regression line
# add bokeh
```

3.2 Interactive Plots with bokeh

In [ ]:
```
import bokeh
from bokeh.io import output_notebook
output_notebook(resources=bokeh.resources.INLINE)

from bokeh.models import ColumnDataSource

source_type = ColumnDataSource(groupby_type)

from bokeh.plotting import figure
p=figure(plot_height=500, # Desired figure height
       plot_width=500,  # Desired figure width
       title='Title',       # Desired title of your figure
       x_range=groupby_type      # Customize x-axis values
      )

p.vbar(x='Feature1', top='Feature2', source=source_type, width=0.7)

p.yaxis[0].formatter.use_scientific = False

p.xaxis[0].axis_label='x-Axis'
p.yaxis[0].axis_label='y-Axis'

from bokeh.io import show

show(p)

# Scatterplot

groupby_feature = df.groupby('Feature')
source_feature = ColumnDataSource(groupby_feature)
# 3. Create figure
p = figure(plot_width=500,
           plot_height=500,
           title="Title")

# 4. Add glyphs
# Use a glyph method for scatterplots
# Parameters are similar to `vbar`. This time,
# No need to specify `width` parameter and use `y` instead of `top`
p.circle(x="Feature1",
         y="Feature2",
         source=source_date)

# 5. Add styling preferences
# Supply x and y axis labels and disable scientific notation
p.xaxis[0].axis_label = 'Feature1'
p.yaxis[0].axis_label = 'Feature2'
p.xaxis[0].formatter.use_scientific = False

# 6. Display graph
show(p)


from bokeh.models.tools import HoverTool
hover_tool = HoverTool(tooltips=[("Feature1", "@Feature1"),("Feature2", "@Feature2")],mode="mouse")
p.add_tools(hover_tool)
show(p)
```

4 Machine Learning Models

4.1 Linear Regression

In [ ]:
```
from sklearn.linear_model import LinearRegression

# Create instance of the model
model = LinearRegression()

# Training the model
model.fit(features,target) # features is a DataFrame of the feature to be analysed target is a series of the output

# Return the learned coefficients
model.intercept_ #  Returns the learned y-intercept
model.coef_ #  Returns the learned slope

# Predict the output for given inputs
target_aim_pred = model.predict(features_aim) #  Here features_aim is a DataFrame with the features for which the output sh
ould be predicted
```

4.1.2 Ridge and Lasso

In [ ]:
```
from sklearn.linear_model import Ridge
#Ridge
# Initializing the model
model_ridge = Ridge(alpha=x) #  alpha interpolates between no (alpha=0) and strong (alpha>1) regularization

# Rescaling

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

features_train_standardized = scaler.fit_transform(features_train) #  features_train is data frame with features
model_ridge.fit(features_train_standardized, target_train)


# Rescaling test data to the same scale as the training data

scaler.fit(features_train)
features_test_standardized = scaler.transform(features_test)

# Lasso

from sklearn.linear_model import Lasso

model_lasso = Lasso(alpha=1)

model_lasso.fit(features_train,target_train)
```

4.2 Classification

4.2.1 k-Nearest-Neighbors

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier

# Initializing the model
model = KNeighborsClassifier(n_neighbors=3) #  n_neighbors sets number of nearest neighbors taken into account (the k valu
e)

#Standardizing the features
from sklearn.preprocessing import StandardScaler
standardizer = StandardScaler()
features_train_standardized = standardizer.fit_transform(features_train)

#Training the model
model.fit(features_train_standardized, target_train)

# Predicting
target_aim_pred = model.predict(features_aim_standardized)

# Evaluating the model
from sklearn.metrics import accuracy_score
accuracy_score(target_test, target_test_pred)

# Confusion matrix
from sklearn.metrics import confusion_matrix
print(confusion_matrix(target_test,target_test_pred))

# For classification (Precision, recall and F1 score)
from sklearn.metrics import precision_score, recall_score, f1_score
print(recall_score(target_test,target_test_pred))
print(precision_score(target_test,target_test_pred))
print(f1_score(target_test,target_test_pred))
```

4.2.2 Logistic Regression

In [ ]:
```python
from sklearn.linear_model import LogisticRegression
model_log = LogisticRegression(solver='lbfgs')

features_train = df.loc[:,['Feature']]

target = df['Target']
model_log.fit(features_train,target)

# Preparing categorical features /label coding
for cat in features_cat[1:]:
    df.loc[:, cat] = df.loc[:, cat].replace({'Yes': 1, 'No': 0})

# One hot encoding
df = pd.get_dummies(df, columns=["FeatureToBeEncoded"])

# Scaling the data
from sklearn.preprocessing import MinMaxScaler #  Assures the min and max of each feature is 0 and 1 repectively
scaler=MinMaxScaler()

features_train_scaled = scaler.fit_transform(features_train,target_train)
features_train_scaled = pd.DataFrame(features_train_scaled, columns=features_train.columns)

# Training the model
model_reg.fit(features_train_scaled, target_train)

# Tranforming the Data which should be classified
features_aim_scaled = scaler.transform(features_aim)

df_aim['fake_pred_reg'] = model_reg.predict(features_aim_scaled)

# Predicted probabilities
target_aim_pred_proba = model_log.predict_proba(features_aim)

# False Positive Rate
from sklearn.metrics import confusion_matrix
import numpy as np


cm_log = confusion_matrix(target_test,target_test_pred_log)
cm_reg = confusion_matrix(target_test,target_test_pred_reg)
FPR_log = cm_log[0,1]/(np.sum(cm_log[1,:]))
FPR_reg = cm_reg[0,1]/(np.sum(cm_reg[1,:]))
print('FPR for Log =' + str(FPR_log))
print('FPR for Reg =' + str(FPR_reg))

# ROC Curve
# calculate probability
target_test_pred_proba_log = model_log.predict_proba(features_test)  # model_log does not use regularization --> scaled fea
tures not needed

# module import
from sklearn.metrics import roc_curve

# calculate roc curve values
false_positive_rate_log, recall_log, threshold = roc_curve(target_test, target_test_pred_proba_log[:, 1], drop_intermediat
e=False)

target_test_pred_proba_reg = model_reg.predict_proba(features_test_scaled)

# calculate roc curve values
false_positive_rate_reg, recall_reg, threshold = roc_curve(target_test, target_test_pred_proba_reg[:, 1], drop_intermediat
e=False)

# figure and axes intialisation
fig, ax = plt.subplots()

# reference lines
ax.plot([0, 1], ls = "--", label='random model')  # blue diagonal
ax.plot([0, 0], [1, 0], c=".7", ls='--', label='ideal model')  # grey vertical
ax.plot([1, 1], c=".7", ls='--')  # grey horizontal

# roc curve
ax.plot(false_positive_rate_reg, recall_reg, label='model_reg')

# labels
ax.set_title("Title")
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("Recall")
ax.legend()
```

```
# ROC-AUC measure (Area Under Curve)

from sklearn.metrics import roc_auc_score

roc_auc_score(target_test, target_test_pred_proba_log[:,1])

# Logistic Regression with GridSearch in a pipeline

model_reg = LogisticRegression(solver = 'saga', max_iter = 1e4)


pipeline_log = Pipeline([('scaling',scaler),('classifier',model_reg)])

import numpy as np
C_values = np.geomspace(0.001,1000,14) #  Create geometric list of parameters for C parameter grid search
search_space_grid = [{'classifier__penalty':['l1','l2'],'classifier__C':C_values}]
model_grid = GridSearchCV(estimator=pipeline_log, param_grid=search_space_grid,scoring='roc_auc',cv=5)

from sklearn.exceptions import DataConversionWarning
import warnings
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

print(model_grid.best_estimator_)
print(model_grid.best_score_)

features_test = df_test.iloc[:, 1:]
target_test = df_test['fake']

target_test_pred_proba = model_grid.predict_proba(features_test)
from sklearn.metrics import roc_auc_score

roc_auc_score(target_test,target_test_pred_proba[:, 1])
```

4.2.3 Decision Tree

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
model  = DecisionTreeClassifier(max_depth=1,random_state=0)
model.fit(features_train,target_train)
```

4.2.4 Random Forests

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

model_rf = RandomForestClassifier(n_estimators=100,max_depth=12, class_weight = 'balanced', random_state=42)

features_train = df_train.drop('Target', axis=1)
target_train = df_train['Target']
model_rf.fit(features_train,target_train)

target_test_pred = model_rf.predict(features_test)

print('Precision: ', precision_score(target_test, target_test_pred))

print('\nRecall: ', recall_score(target_test, target_test_pred))
```

stackfuel

4.3 Clustering

4.3.1 k-Means

In [ ]:
```
from sklearn.cluster import KMeans

# Creating an instance of the model
model = KMeans(n_clusters = 4) #  n_clusters is the number of clusters used

# Training the model
model.fit(df_features)

# Getting the predictions
model.labels_ #  Stores the labels output by the model

# Adding the labels to a data frame
df['lables'] = model.labels_

# Visualization of data points with cluster centers
ax_cluster = sns.scatterplot(data=df,x='x',y='y',hue='labels')

sns.scatterplot(x=model.cluster_centers_[:,0],y=model.cluster_centers_[:,1],ax=ax_cluster, marker='X', s=200)

# Predicting labels
labels_predicted = model.predict(df.loc[:,['x','y']])

# Checking if predictions coincide with learned labels
check = (labels_predicted!=model.labels_)
print(check.sum())

# Obtaining distance to the cluster centers
model.transform(df.loc[:, ['x','y']])

# Assessing the model: within-cluster sum of squares
model.score(data)

# Elbow method: Varying cluster number
cluster_scores = []
for i in range(1,31):
    model_i = KMeans(n_clusters = i)
    model_i.fit(arr_customers_std)
    cluster_scores.append(model_i.score(arr_customers_std))

# Elbow method: Plotting the within-cluster sum of squares
import matplotlib.pyplot as plt
%matplotlib inline

ax = plt.subplots() # Pick number of clusters in the "Elbow"
plt.plot(range(1,31),cluster_scores)
plt.style.use('fivethirtyeight')


# Ensuring comparable results by setting random state
model = KMeans(n_clusters = int1, random_state=int2) #  the actual value is not important, it acts as seed

# Silhouette score and outliers
from sklearn.metrics import silhouette_score
silhouette_score(X=arr, labels=model.labels_) #  Returns average silhouette score, measure of the model

# Silhouette score of individual data points
from sklearn.metrics import silhouette_samples

arr_sil = silhouette_samples(X=arr, labels=model.labels_)
```

4.3.2 DBSCAN

In [ ]:
```
from sklearn.cluster import DBSCAN

model_db= DBSCAN(eps=0.12,min_samples=4) #  eps is radius around data points, min_samples is number of data points to form
new cluster
model_db.fit(other_cluster)

# Scatterplot of data with obtained labels
sns.scatterplot(data=other_cluster,x='x',y='y', hue=model_db.labels_)
```

4.3.3 (linear) Support Vector Machines

In [ ]:
```
from sklearn.svm import SVC

model_svm = SVC(kernel='linear',C=10)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

features = scaler.fit_transform(df.loc[:,['feature_1', 'feature_2']])

model_svm.fit(features,df['class'])

features = scaler.fit_transform(df.loc[:, ['feature_1', 'feature_2']])
```

5 Model Tuning and Interpretation

5.1 Useful Metrics

In [ ]:
```
# Generating predictions on the training set
target_pred = model_category.predict(features)

# Mean squared error
from sklearn.metrics import mean_squared_error
mean_squared_error(target, target_pred) #  here target is a numpy array with the actual output and target_pred is a numpy a
rray with the output of the model

# r squared (r2) score
from sklearn.metrics import r2_score
r2_score(target, target_pred_)

#Residualplot
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
sns.scatterplot(x=target_pred,
                y=residuals,ax=ax)

# labels
ax.set(xlabel='Predicted y-values',
           ylabel='Residuals',
           title='Dataset: I')

# zero line
ax.hlines(y=0,
          xmin=target_pred.min(),
          xmax=target_pred.max())

# For classification(accuracy)
from sklearn.metrics import accuracy_score
accuracy_score(target_test, target_test_pred)

# For classification (Confusion matrix)
from sklearn.metrics import confusion_matrix
print(confusion_matrix(target_test,target_test_pred))

# For classification (Precision, recall and F1 score)
from sklearn.metrics import precision_score, recall_score, f1_score
print(recall_score(target_test,target_test_pred))
print(precision_score(target_test,target_test_pred))
print(f1_score(target_test,target_test_pred))
```

5.2 Cross-validation

In [ ]:
```
from sklearn.model_selection import KFold

# Initializing
kf = KFold(n_splits=2)

# Create a generator object
kf.split(df_train) #  Generates array pairs


# Full Cross-validation
step = 0   # set counter to 0

for train_index, val_index in kf.split(df_train):  # for each fold
    step = step + 1  # update counter

    print('Step ', step)
    standardizer = StandardScaler()
    features_fold_train = df_train.iloc[train_index, [4, 5]]  # features matrix of training data (of this step)
    features_fold_val = df_train.iloc[val_index, [4, 5]]  # features matrix of validation data (of this step)

    standardizer.fit(features_fold_train)
    features_fold_train_standardized = standardizer.transform(features_fold_train)
    features_fold_val_standardized = standardizer.transform(features_fold_val)

    target_fold_train = df_train.iloc[train_index, 6]  # target vector of training data (of this step)
    target_fold_val = df_train.iloc[val_index, 6]  # target vector of validation data (of this step)

    model.fit(features_fold_train_standardized, target_fold_train)

    target_fold_val_pred = model.predict(features_fold_val_standardized)

    print("Recall: "+ str(recall_score(target_fold_val,target_fold_val_pred)))
    print("Precision: "+ str(precision_score(target_fold_val,target_fold_val_pred)))
    print("F1: "+ str(f1_score(target_fold_val,target_fold_val_pred)))

# Cross Validation with cross_val_score
from sklearn.model_selection import cross_val_score
cv_results=cross_val_score(estimator=model,X=features,y=target,cv=5, scoring='neg_mean_absolute_error')
```

5.3 Grid Search

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# Create a dict with the Hyperparameters to be varied
search_space = [{'knn__n_neighbors': k}] #  One parameter
search_space_grid = [{'knn__n_neighbors': k, 'knn__weights': ['uniform', 'distance']}] #  Two parameter

# Creating an instance of grid search
grid_search_for_k = GridSearchCV(estimator = pipeline_std_knn,  # estimator
             param_grid = search_space,  # the grid of the grid search
             scoring='f1',  # which measure to optimise on
             cv=2)  # number of folds during cross_validation

# Getting the best score achieved by grid search
grid_search_for_k.best_score_

# Getting the best parameter
grid_search_for_k.best_estimator_

# Fitting the model
model.fit(features_train,target_train)

# Optional: Combine grid search and feature selection
import itertools #  module with helpful tools to generate iterations

itertools.combinations(col_of_interest, 2) #  Returns a list of tupels with all iterations of 2 elements from col_of_intere
st
```

stackfuel

5.4 Interpreting Models

5.4.1 Interpreting Decision Trees

In [ ]:

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(random_state=0,class_weight='balanced',max_depth=3)
model.fit(features_train,target_train)

from sklearn.tree import export_graphviz

tree_string = export_graphviz(decision_tree=model,filled=True,impurity=True,feature_names=features_train.columns)

from pydotplus import graph_from_dot_data

graph = graph_from_dot_data(tree_string)

graph.write_png('pic.png')

from IPython.display import Image
Image('decision_tree.png')
```

In [ ]:

```
# For Trees:

imp_series = pd.Series(data=model.feature_importances_, index=features_train.columns)
imp_series.sort_values() #  Creates Series with name of feature and Gini-importance of the feature


import matplotlib.pyplot as plt
%matplotlib inline
colors=['#7570b3', '#7570b3', '#7570b3', '#1b9e77','#1b9e77','#d95f02']
fig,ax = plt.subplots(figsize=[12,5])
mask = imp_series>0
x= imp_series[mask].sort_values()
ax = x.plot(kind='barh',color=colors,width=0.9)
ax.set_title(label='Title', family='serif', color='#d95f02', weight='semibold', size=14)
ax.set_xlabel('Relative Importance', size=12, position=[0, 0], horizontalalignment='left')
ax.set_ylabel('Features', size=12, position=[0, 1], horizontalalignment='right')
ax.set_yticklabels(ax.get_yticklabels(), size=12)

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

for idx in range(len(x.index)):
    ax.text(s='{}%'.format(int(100*x.iloc[idx])), x=x.iloc[idx]+0.005, y=idx, size=12, color=colors[idx])
```

5.4.2 Feature Importance

In [ ]:

```
# Model agnostic methods: Permutation Feature Importance

from sklearn.metrics import accuracy_score

acc_orig = accuracy_score(target_test,target_test_pred) # Original accuracy

features_test_perm = features_test.copy()

age_series_perm = features_test_perm['age'].sample(random_state=0,frac=1,replace=False) #  Mixing the entries in column 'ag
e'

age_series_perm = age_series_perm.reset_index(drop=True) #  Resetting the indexes of the mixed entries

features_test_perm['age'] = age_series_perm #  Overwritting the previous column with the mixed column

target_test_pred_perm = model_rf.predict(features_test_perm) # Create prediction based on features with mixed column

acc_age= accuracy_score(target_test,target_test_pred_perm) # Calculate the accuracy of the prediction with mixed column

print(acc_orig-acc_age)  # Difference of the two accuracy as a measure of the importance of the feature

# Feature Permutation for all features

perm_importances=[]
for col in features_test_perm.columns:
    features_test_perm_col = features_test.copy()
    age_series_perm_col = features_test_perm_col[col].sample(random_state=0,frac=1,replace=False)
    age_series_perm_col = age_series_perm_col.reset_index(drop=True)
    features_test_perm_col[col] = age_series_perm_col
    target_test_pred_perm_col = model_rf.predict(features_test_perm_col)
    acc_col= accuracy_score(target_test,target_test_pred_perm_col)
    perm_importances.append(acc_orig-acc_col)
    features_test_perm.loc[:, col] = features_test.loc[:, col]
perm_importances

plot_series = pd.Series(data=perm_importances,index=features_test_perm.columns).sort_values()
plot_series.plot(kind='barh')
```

In [ ]:

```
# PDP and ICE plots
from pdpbox import pdp

pdp_monthlyincome = pdp.pdp_isolate(model=model_rf,           # a fitted sklearn model
                dataset=features_train,     # the dataset on which the model was trained
                model_features=features_train.columns,  # names of all the features the model uses
                feature='Feature'         # feature's column name in `dataset`
               )

pdp.pdp_plot(pdp_isolate_out=pdp_monthlyincome,  # output of pdp.isolate()
             feature_name='Feature',                          # name of feature (for title)
             center=bool,                              # center the plot
             plot_pts_dist=bool                        # display real feature values
            )

# ICE

fig, ax_dict=pdp.pdp_plot(pdp_isolate_out=pdp_monthlyincome,
             feature_name="Feature",
             plot_lines=True,
             plot_pts_dist=True,
             center=True)
ax_dict['pdp_ax']['_pdp_ax'].set_ylim(-0.5,0.5)

PDP Interact

pdp_income_joblevel = pdp.pdp_interact(model=model_rf, dataset=features_train, model_features=features_train.columns, featu
res=['Feature1', 'Feature2'])

pdp.pdp_interact_plot(pdp_income_joblevel,
                    ['Feature1', 'Feature2'],
                    plot_type='grid')
```

stackfuel

6 Further Frameworks

6.1 Keras and Neural Networks

In [ ]:

```
from keras.models import Sequential
from keras.layers import Dense

model_ann = Sequential()  # define the model type
model_ann.add(Dense(1, activation='sigmoid', input_dim=features_train_scaled.shape[1]))  # add one layer
model_ann.compile(optimizer = "adam", loss = 'binary_crossentropy', metrics = ['accuracy'])  # compile the model

model_ann.fit(features_train_scaled, target_train, epochs=5) # Fit the model

target_val_pred_ann = model_ann.predict(features_val_scaled)

target_val_pred_ann = target_val_pred_ann.flatten() #  flattening the nested output

target_val_pred_ann = target_val_pred_ann > 0.5 #  applying threshold of 0.5 for classification
```

In [ ]:

```
# ensure reproducible results

# set environment-variable
import os
os.environ['PYTHONHASHSEED'] = '0'

# set seed of random number generators
import random as rn
rn.seed(0)

import numpy as np
np.random.seed(0)

import tensorflow as tf
tf.set_random_seed(0)

# disable parallel computation
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,
                              inter_op_parallelism_threads=1)
from keras import backend as K
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# turn off warnings
tf.logging.set_verbosity(tf.logging.ERROR)

from keras.models import Sequential
from keras.layers import Dense

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
features_train_scaled = scaler.fit_transform(features_train)
features_val_scaled = scaler.transform(features_val)

from keras.models import Sequential

model_ann = Sequential()

from keras.layers import Dense

model_ann.add(Dense(units=50, activation='relu',input_dim=features_train_scaled.shape[1]))
model_ann.add(Dense(units=50, activation='relu'))
model_ann.add(Dense(units=50, activation='relu'))
model_ann.add(Dense(units=50, activation='relu'))
model_ann.add(Dense(units=50, activation='relu'))

model_ann.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])

hist_ann = model_ann.fit(features_train_scaled, target_train, epochs=20, batch_size=64, validation_data=(features_val_scale
d, target_val))

%matplotlib inline
import matplotlib.pyplot as plt

# define figure and axes
fig, axs = plt.subplots(ncols=2, figsize=(12,6))
```

```
# plot training & validation loss values
axs[0].plot(hist_ann.history['loss'])
axs[0].plot(hist_ann.history['val_loss'])
axs[0].set(title='Model loss', ylabel='Loss', xlabel='Epoch')
axs[0].legend(['train', 'val'])

# plot training & validation accuracy values
axs[1].plot(hist_ann.history['acc'])
axs[1].plot(hist_ann.history['val_acc'])
axs[1].set(title='Model accuracy', ylabel='Accuracy', xlabel='Epoch')
axs[1].legend(['train', 'val'], loc='upper left');

fig.savefig('picname.png')

target_val_pred = model_ann.predict(features_val_scaled)

from sklearn.metrics import accuracy_score


target_val_pred = target_val_pred.flatten()  # make the array 1 dimensional
target_val_pred_binary = target_val_pred > 0.5  # use the comparison to get True (=1) and False (=0) values
accuracy_score(target_val, target_val_pred_binary)  # calculate accuracy

# Early stopping
from keras.callbacks import EarlyStopping

early_stop = EarlyStopping()

# train model using early stopping
model_ann.fit(features_train_scaled, target_train,
              epochs=200, batch_size=64,
              callbacks=[early_stop],
              validation_data=(features_val_scaled, target_val))

# Saving a model
model_ann.save('model_ann.h5')

# Loading a saved model
from keras.models import load_model

model = load_model('path_to_model.h5')
```

6.2 Spark and PySpark

In [ ]:

```
data_dir = "HDD_logs/"

import os
file_list = sorted(os.listdir(data_dir))
print("Number of Files:" ,len(file_list))


from pyspark.sql import SparkSession

# connect to Spark
spark = (SparkSession
        .builder
        .appName("Name")
        .getOrCreate()
        )

df1 = spark.read.csv(data_dir+file_list[0], header=True)

# Load huge amount of data into spark

df = spark.read.csv(data_dir+file_list[0], header=True)
for file in file_list[1:]:
    print('Processing: '+str(file)) #  This may take a long time. Printing filename can help to see if program is still run
ning
    df_tmp = spark.read.csv(data_dir+file, header=True)
    df = df.union(df_tmp)
print('Number of rows :' + str(df.count()))
```

```
# Ending a spark session

spark.stop()

# looking at a pyspark data frame
my_spark_df.show()

# Column names
df_spark.columns

# Select Columns in PySpark
my_spark[list]

df_spark = df_spark.toDF(*renamed_cols) #  here renamed_cols is a list of the new column names

#Stat data about Spark DataFrames
df[metacols].describe().show()

# Register as SQL Table
df.registerTempTable('meta_data')


# User-defined functions in spark

import pyspark.sql.functions as F
clean_name_udf = F.udf(clean_name)

# Datatypes of data in spark data frames

df.printSchema()

# One-hot-encoding

from pyspark.ml.feature import StringIndexer

brand_indexer = StringIndexer(inputCol="ColumnName", outputCol="OutputColumnName")  # initialize indexer
brand_indexer = brand_indexer.fit(df)  # fit indexer to dataframe
df = brand_indexer.transform(df)  # encode brand

df=df.drop('ColumnNametodrop1','ColumnNametodrop1')

from pyspark.ml.feature import OneHotEncoderEstimator

encoder = OneHotEncoderEstimator(inputCols= ['InputCol1','InputCol2'],  # list with names of categorical columns
                       outputCols= ['Output1', 'Output2'],  # list with names of new columns
                       )

encoder=encoder.fit(df)

df=encoder.transform(df)


# rename target col to label -> spark default for target
df = df.withColumnRenamed("ColumnName", "label")

# select all columns that we want to use as features
feature_cols = [col for col in df.columns if col not in ['ColNam1','ColName2']]

# import and initialize VectorAssembler
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols = feature_cols,
                            outputCol = "features")

# Now let us use the transform method to transform our dataset
df = assembler.transform(df)

# Create training/test split

df_train,df_test=df.randomSplit(weights=[0.9,0.1],seed=42)
```

```
# register training set table for use in SQL queries.
df_train.registerTempTable("train_set")

spark.sql("""SELECT label, COUNT(label)
                FROM train_set
                GROUP BY label""").show()

df_train_classes =  spark.sql("""SELECT label, COUNT(label)
                                FROM train_set
                                GROUP BY label""").toPandas()

df_train_count = df_train.count()

df_train_classes.index = df_train_classes.loc[:, 'label']

weights = df_train_count / df_train_classes.loc[:, 'count(label)']

from pyspark.sql.functions import when
df_train = (df_train.withColumn("weights",
              when(df_train["label"] == 0, weights.loc[0]).otherwise(weights.loc[1])))

# Logistic Regression with PySpark

from pyspark.ml.classification import LogisticRegression

model = LogisticRegression(weightCol='weights')

model = model.fit(df_train)
df_test_pred = model.transform(df_test)

pred_summary = model.evaluate(df_train)

print('accuracy :' +str(pred_summary.accuracy))
print('Recall by Label :' +str(pred_summary.recallByLabel))
print('AUROC :' +str(pred_summary.areaUnderROC))
```