# CS412 - Machine Learning - 2020

## Homework 2

100 pts

## Goal

The goal of this homework is to get familiar feature handling and cross validation.

## Dataset

German Credit Risk dataset, prepared by Prof. Hoffman, classifies each person as having a good or bad credit risk. The dataset that we use consists of both numerical and categorical features.

## Task

Build a k-NN classifier with scikit-learn library to classify people as bad or good risks for the german credit dataset.

## Software

Documentation for the necessary functions can be accessed from the link below.

[http://scikit-learn.org/stable/supervised_learning.html](http://scikit-learn.org/stable/supervised_learning.html)

## Submission

Follow the instructions at the end.

## 1) Initialize

First, make a copy of this notebook in your drive

```
# Mount to your drive, in this way you can reach files that are in your drive
# Run this cell
# Go through the link that will be showed below
# Select your google drive account and copy authorization code and paste here in output an
# You can also follow the steps from that link
# https://medium.com/ml-book/simplest-way-to-open-files-from-google-drive-in-google-colab-

from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.r
```

## 2) Load Dataset

To start working for your homework, take a copy of the folder, given in the below link to your own google drive. You find the train and test data under this folder.

https://drive.google.com/drive/folders/1DbW6VxLKZv2oqFn9SwxAnVadmn1_nPXi?usp=sharing

After copy the folder, copy the path of the train and test dataset to paste them in the below cell to load your data.

```
import pandas as pd

train_df = pd.read_csv('/content/drive/My Drive/Copy of german_credit_train.csv')
test_df = pd.read_csv('/content/drive/My Drive/Copy of german_credit_test.csv')
```

## 3) Optional - Analyze the Dataset

You can use the functions of the pandas library to analyze your train dataset in detail - **this part is OPTIONAL - look around the data as you wish**.

- Display the number of instances and features in the train ***(shape function can be used)**
- Display 5 random examples from the train ***(sample function can be used)**
- Display the information about each features ***(info method can be used)**

```
# Print shape
print("Train data dimensionality: ", train_df.shape)

# Print random 5 rows
print("Examples from train data: ")
train_df.sample(n = 5)
```

```
Train data dimensionality:  (800, 13)
Examples from train data:
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | Employmen |
|---|---|---|---|---|---|---|
| **142** | A12 | 6 | A33 | 1449 | A62 | |
| **318** | A12 | 20 | A33 | 7057 | A65 | |
| **714** | A13 | 12 | A34 | 1480 | A63 | |
| **37** | A11 | 12 | A32 | 1274 | A61 | |
| **595** | A14 | 11 | A34 | 1393 | A61 | |

```
# Print the information about the dataset
print("Information about train data ", train_df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   AccountStatus     800 non-null    object
 1   Duration          800 non-null    int64 
 2   CreditHistory     800 non-null    object
 3   CreditAmount      800 non-null    int64 
 4   SavingsAccount    800 non-null    object
 5   EmploymentSince   800 non-null    object
 6   PercentOfIncome   800 non-null    int64 
 7   PersonalStatus    800 non-null    object
 8   Property          800 non-null    object
 9   Age               800 non-null    int64 
 10  OtherInstallPlans  800 non-null    object
 11  Housing           720 non-null    object
 12  Risk              800 non-null    int64 
dtypes: int64(5), object(8)
memory usage: 81.4+ KB
Information about train data  None
```

## 4) Define your train and test labels

- Define labels for both train and test data in new arrays
- And remove the label column from both train and test sets do tht it is not used as a feature!

(**you can use pop method**)

```
# Define labels
train_label = train_df.pop("Risk")
test_label = test_df.pop("Risk")

train_label
```

```
0      1
1      1
2      1
3      2
4      2
      ..
795    1
796    1
797    2
798    1
799    2
Name: Risk, Length: 800, dtype: int64
```

## 5) Handle missing values if any

- Print the columns that have **NaN** values (**isnull** method can be used)

- You can impute missing values with mode of that feature or remove samples or attributes
- To impute the test set, you should use the mode values that you obtain from **train** set, as **you should not be looking at your test data to gain any information or advantage.**

```
# Print columns with NaN values
print(train_df.isnull().any())
print(train_df.isnull().sum())
```

```
AccountStatus       False
Duration            False
CreditHistory       False
CreditAmount        False
SavingsAccount      False
EmploymentSince     False
PercentOfIncome     False
PersonalStatus      False
Property            False
Age                 False
OtherInstallPlans   False
Housing              True
dtype: bool
AccountStatus        0
Duration             0
CreditHistory        0
CreditAmount         0
SavingsAccount       0
EmploymentSince      0
PercentOfIncome      0
PersonalStatus       0
Property             0
Age                  0
OtherInstallPlans    0
Housing             80
dtype: int64
```

```
print(train_df["Housing"])
train_df["Housing"].value_counts()
```

```
0      A152
1      A152
2      A153
3      A152
4      A152
       ...
795     NaN
796     NaN
797    A153
798    A152
799    A152
Name: Housing, Length: 800, dtype: object
A152    512
A151    130
A153     78
Name: Housing, dtype: int64
```

```
# Impute missing values by replacing with mode value
train_df["Housing"] = train_df["Housing"].fillna(train_df["Housing"].mode()[0])
```

```
test_df["Housing"] = test_df["Housing"].fillna(train_df["Housing"].mode()[0])
print(train_df.isnull().sum())
print(test_df.isnull().sum())
```

```
AccountStatus       0
Duration            0
CreditHistory       0
CreditAmount        0
SavingsAccount      0
EmploymentSince     0
PercentOfIncome     0
PersonalStatus      0
Property            0
Age                 0
OtherInstallPlans   0
Housing             0
dtype: int64
AccountStatus       0
Duration            0
CreditHistory       0
CreditAmount        0
SavingsAccount      0
EmploymentSince     0
PercentOfIncome     0
PersonalStatus      0
Property            0
Age                 0
OtherInstallPlans   0
Housing             0
dtype: int64
```

## 6) Transform categorical / ordinal features

- Transform all categorical / ordinal features using the methods that you have learnt in lectures and recitation 4 for both train and test data
- You saw the dictionary use for mapping in recitation. (You can use **replace function** to assign new values to the categories of a column).
- The class of the categorical attributes in the dataset are defined as follows:
    - Status of existing checking account
        - A11 : ... < 0 DM
        - A12 : 0 <= ... < 200 DM
        - A13 : ... >= 200 DM / salary assignments for at least 1 year
        - A14 : no checking account
    - Credit history
        - A30 : no credits taken/all credits paid back duly
        - A31 : all credits at this bank paid back duly

    - A32 : existing credits paid back duly till now
    - A33 : delay in paying off in the past
    - A34 : critical account/other credits existing (not at this bank)
  - Savings account
    - A61 : ... < 100 DM
    - A62 : 100 <= ... < 500 DM
    - A63 : 500 <= ... < 1000 DM
    - A64 : .. >= 1000 DM
    - A65 : unknown/ no savings account
  - Employment Since
    - A71 : unemployed
    - A72 : ... < 1 year
    - A73 : 1 <= ... < 4 years
    - A74 : 4 <= ... < 7 years
    - A75 : .. >= 7 years
  - Personal Status
    - A91 : male : divorced/separated
    - A92 : female : divorced/separated/married
    - A93 : male : single
    - A94 : male : married/widowed
    - A95 : female : single
  - Property
    - A121 : real estate
    - A122 : if not A121 : building society savings agreement/life insurance
    - A123 : if not A121/A122 : car or other, not in attribute 6
    - A124 : unknown / no property
  - OtherInstallPlans
    - A141 : bank
    - A142 : stores
    - A143 : none
  - Housing
    - A151 : rent
    - A152 : own
    - A153 : for free

```
train_df.head()
```

|   | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | A14 | 12 | A32 | 2859 | A65 | |
| **1** | A11 | 9 | A32 | 2136 | A61 | |
| **2** | A11 | 18 | A34 | 5302 | A61 | |
| **3** | A11 | 14 | A32 | 8978 | A61 | |

```
# Transform the categorical / ordinal attributes
print(train_df["AccountStatus"].unique())
AccountStatusMap = {"A14":0, "A11":1, "A12":2, "A13":3}
train_df["AccountStatus"] = train_df["AccountStatus"].replace(AccountStatusMap)
train_df.head()
```

```
['A14' 'A11' 'A12' 'A13']
```

|   | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | A32 | 2859 | A65 | |
| **1** | 1 | 9 | A32 | 2136 | A61 | |
| **2** | 1 | 18 | A34 | 5302 | A61 | |
| **3** | 1 | 14 | A32 | 8978 | A61 | |
| **4** | 0 | 15 | A32 | 4623 | A62 | |

```
print(train_df["CreditHistory"].unique())
CreditHistoryMap = {"A34":0, "A33":1, "A32":2, "A31":3, "A30":4}
train_df["CreditHistory"] = train_df["CreditHistory"].replace(CreditHistoryMap)
train_df.head()
```

```
['A32' 'A34' 'A33' 'A31' 'A30']
```

|   | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | A65 | |
| **1** | 1 | 9 | 2 | 2136 | A61 | |
| **2** | 1 | 18 | 0 | 5302 | A61 | |
| **3** | 1 | 14 | 2 | 8978 | A61 | |
| **4** | 0 | 15 | 2 | 4623 | A62 | |

```
print(train_df["SavingsAccount"].unique())
SavingsAccountMap = {"A65":0,"A61":1,"A62":2,"A63":3, "A64":4}
train_df["SavingsAccount"] = train_df["SavingsAccount"].replace(SavingsAccountMap)
train_df.head()
```

```
['A65' 'A61' 'A62' 'A63' 'A64']
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |
| **1** | 1 | 9 | 2 | 2136 | 1 | |
| **2** | 1 | 18 | 0 | 5302 | 1 | |

```
print(train_df["EmploymentSince"].unique())
EmploymentSinceMap = {"A71":0,"A72":1,"A73":2,"A74":3, "A75":4}
train_df["EmploymentSince"] = train_df["EmploymentSince"].replace(EmploymentSinceMap)
train_df.head()
```

```
['A71' 'A73' 'A75' 'A74' 'A72']
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |
| **1** | 1 | 9 | 2 | 2136 | 1 | |
| **2** | 1 | 18 | 0 | 5302 | 1 | |
| **3** | 1 | 14 | 2 | 8978 | 1 | |
| **4** | 0 | 15 | 2 | 4623 | 2 | |

```
print(train_df["PersonalStatus"].unique())
dummies_status = pd.get_dummies(train_df['PersonalStatus'],prefix='status')
dummies_status.head()
```

```
['A93' 'A91' 'A92' 'A94']
```

| | status_A91 | status_A92 | status_A93 | status_A94 |
|---|---|---|---|---|
| **0** | 0 | 0 | 1 | 0 |
| **1** | 0 | 0 | 1 | 0 |
| **2** | 0 | 0 | 1 | 0 |
| **3** | 1 | 0 | 0 | 0 |
| **4** | 0 | 0 | 1 | 0 |

```
train_df = pd.merge(train_df,dummies_status,left_index=True,right_index=True)
train_df = train_df.drop(columns="PersonalStatus")
train_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |

```
print(train_df["Property"].unique())
dummies_property = pd.get_dummies(train_df["Property"],prefix='property')
dummies_property.head()
```

```
['A124' 'A121' 'A122' 'A123']
```

| | property_A121 | property_A122 | property_A123 | property_A124 |
|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 1 |
| **1** | 1 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 1 |
| **3** | 0 | 1 | 0 | 0 |
| **4** | 0 | 1 | 0 | 0 |

```
train_df = pd.merge(train_df,dummies_property,left_index=True,right_index=True)
train_df = train_df.drop(columns="Property")
train_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |
| **1** | 1 | 9 | 2 | 2136 | 1 | |
| **2** | 1 | 18 | 0 | 5302 | 1 | |
| **3** | 1 | 14 | 2 | 8978 | 1 | |
| **4** | 0 | 15 | 2 | 4623 | 2 | |

```
print(train_df["OtherInstallPlans"].unique())
dummies_plans = pd.get_dummies(train_df["OtherInstallPlans"],prefix='plans')
dummies_plans.head()
```

```
['A143' 'A141' 'A142']
```

| | plans_A141 | plans_A142 | plans_A143 |
|---|---|---|---|
| **0** | 0 | 0 | 1 |
| **1** | 0 | 0 | 1 |
| **2** | 0 | 0 | 1 |
| **3** | 0 | 0 | 1 |
| **4** | 0 | 0 | 1 |

```
train_df = pd.merge(train_df,dummies_plans,left_index=True,right_index=True)
train_df = train_df.drop(columns="OtherInstallPlans")
```

```
train_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |
| **1** | 1 | 9 | 2 | 2136 | 1 | |
| **2** | 1 | 18 | 0 | 5302 | 1 | |
| **3** | 1 | 14 | 2 | 8978 | 1 | |
| **4** | 0 | 15 | 2 | 4623 | 2 | |

```
print(train_df["Housing"].unique())
dummies_housing = pd.get_dummies(train_df["Housing"],prefix='plans')
dummies_housing.head()
```

```
['A152' 'A153' 'A151']
```

| | plans_A151 | plans_A152 | plans_A153 |
|---|---|---|---|
| **0** | 0 | 1 | 0 |
| **1** | 0 | 1 | 0 |
| **2** | 0 | 0 | 1 |
| **3** | 0 | 1 | 0 |
| **4** | 0 | 1 | 0 |

```
train_df = pd.merge(train_df,dummies_housing,left_index=True,right_index=True)
train_df = train_df.drop(columns="Housing")
train_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |
| **1** | 1 | 9 | 2 | 2136 | 1 | |
| **2** | 1 | 18 | 0 | 5302 | 1 | |
| **3** | 1 | 14 | 2 | 8978 | 1 | |
| **4** | 0 | 15 | 2 | 4623 | 2 | |

```
print(test_df["AccountStatus"].unique())
AccountStatusMap = {"A14":0, "A11":1, "A12":2, "A13":3}
test_df["AccountStatus"] = test_df["AccountStatus"].replace(AccountStatusMap)
```

```
['A12' 'A11' 'A13' 'A14']
```

```
print(test_df["CreditHistory"].unique())
CreditHistoryMap = {"A34":0, "A33":1, "A32":2, "A31":3, "A30":4}
test_df["CreditHistory"] = test_df["CreditHistory"].replace(CreditHistoryMap)
```

```
['A31' 'A34' 'A32' 'A30' 'A33']
```

```
print(test_df["SavingsAccount"].unique())
SavingsAccountMap = {"A65":0,"A61":1,"A62":2,"A63":3, "A64":4}
test_df["SavingsAccount"] = test_df["SavingsAccount"].replace(SavingsAccountMap)
print(test_df["EmploymentSince"].unique())
EmploymentSinceMap = {"A71":0,"A72":1,"A73":2,"A74":3, "A75":4}
test_df["EmploymentSince"] = test_df["EmploymentSince"].replace(EmploymentSinceMap)
```

```
['A64' 'A61' 'A65' 'A63' 'A62']
['A73' 'A71' 'A72' 'A75' 'A74']
```

```
print(test_df["PersonalStatus"].unique())
dummies_status = pd.get_dummies(test_df['PersonalStatus'],prefix='status')
test_df = pd.merge(test_df,dummies_status,left_index=True,right_index=True)
test_df = test_df.drop(columns="PersonalStatus")
```

```
['A93' 'A92' 'A91' 'A94']
```

```
print(test_df["Property"].unique())
dummies_property = pd.get_dummies(test_df["Property"],prefix='property')
test_df = pd.merge(test_df,dummies_property,left_index=True,right_index=True)
test_df = test_df.drop(columns="Property")
```

```
['A123' 'A121' 'A124' 'A122']
```

```
print(test_df["OtherInstallPlans"].unique())
dummies_plans = pd.get_dummies(test_df["OtherInstallPlans"],prefix='plans')
test_df = pd.merge(test_df,dummies_plans,left_index=True,right_index=True)
test_df = test_df.drop(columns="OtherInstallPlans")
```

```
['A142' 'A143' 'A141']
```

```
print(test_df["Housing"].unique())
dummies_housing = pd.get_dummies(test_df["Housing"],prefix='plans')
test_df = pd.merge(test_df,dummies_housing,left_index=True,right_index=True)
test_df = test_df.drop(columns="Housing")
```

```
['A152' 'A151' 'A153']
```

```
test_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 2 | 30 | 3 | 3496 | 4 | |
| **1** | 1 | 12 | 0 | 3499 | 1 | |

## 7) Build a k-NN classifier on training data and perform models selection using 5 fold cross validation

- Initialize k-NN classifiers with **k= 5, 10, 15**
- Calculate the cross validation scores using cross_al_score method, number of folds is 5.
- Note: Xval is performed on training data! Do not use test data in any way and do not separate a hold-out validation set, rather use cross-validation.

Documentation of the cross_val_score method:

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.cross_val_score

- Stores the average accuracies of these folds
- Select the value of k using the cross validation results.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from statistics import mean

# k values
kVals = [5,10,15]

# Save the accuracies of each value of kVal in [accuracies] variable
accuracies = []

# Loop over values of k for the k-Nearest Neighbor classifier
for k in kVals:
  # Initialize a k-NN classifier with k neighbors
  my_classifier = KNeighborsClassifier(n_neighbors= k)

  # Calculate the 5 fold cross validation scores using cross_val_score
  # cv parameter: number of folds, in our case it must be 5
  scores = cross_val_score(my_classifier, train_df, train_label, cv = 5)
  # Stores the average accuracies of the scores in accuracies variable, you can use mean m
  accuracies.append(mean(scores))

print(accuracies)
```

```
     [0.675, 0.7075, 0.71]
```

## 8) Retrain using all training data and test on test set

- Train a classifier with the chosen k value of the best classifier using **all training data**.

Note: k-NN training involves no explicit training, but this is what we would do after model selection with decision trees or any other ML approach (we had 5 diff. models -one for each fold - for each k in the previous step - dont know which one to submit. Even if we picked the best one, it does not use all training samples.

- Predict the labels of testing data
- Report the accuracy

```
from sklearn.metrics import accuracy_score

# Train the best classifier using all training set
my_best_classifier = KNeighborsClassifier(n_neighbors= 15)
my_best_classifier.fit(train_df, train_label)
# Estimate the prediction of the test data
pred = my_best_classifier.predict(test_df)

# Print accuracy of test data
accuracy = accuracy_score(test_label, pred)
print(accuracy)
```

```
     0.665
```

## 9) Bonus (5pts)

There is a limited bonus for any extra work that you may use and improve the above results.

You may try a larger k values, scale input features, remove some features, .... Please **do not overdo**, maybe spend another 30-60min on this. The idea is not do an exhaustive search (which wont help your understanding of ML process), but just to give some extra points to those who may look at the problem a little more comprehensively.

**If you obtain better results than the above, please indicate the best model you have found and the corresponding accuracy.**

E.g. using feature normalization ..... and removing .... features and using a value k=...., I have obtained ....% accuracy.

```
train_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0 | 12 | 2 | 2859 | 0 | |
| **1** | 1 | 9 | 2 | 2136 | 1 | |

```
train_label.head()
```

```
0    1
1    1
2    1
3    2
4    2
Name: Risk, dtype: int64
```

```
#Make label as binary zeros and ones
Label = {1:1, 2:0}
train_label = train_label.replace(Label)
test_label = test_label.replace(Label)
train_label.head()
```

```
0    1
1    1
2    1
3    0
4    0
Name: Risk, dtype: int64
```

```
#Normalization of datasets
train_df = (train_df - train_df.min())/(train_df.max()-train_df.min())
test_df = (test_df - test_df.min())/(test_df.max()-test_df.min())
train_df.head()
```

| | AccountStatus | Duration | CreditHistory | CreditAmount | SavingsAccount | EmploymentS |
|---|---|---|---|---|---|---|
| **0** | 0.000000 | 0.117647 | 0.5 | 0.143557 | 0.00 | |
| **1** | 0.333333 | 0.073529 | 0.5 | 0.103775 | 0.25 | |
| **2** | 0.333333 | 0.205882 | 0.0 | 0.277980 | 0.25 | |
| **3** | 0.333333 | 0.147059 | 0.5 | 0.480247 | 0.25 | |
| **4** | 0.000000 | 0.161765 | 0.5 | 0.240618 | 0.50 | |

```
#Drop PERSONAL STATUS
train_df.pop("status_A91")
train_df.pop("status_A92")
train_df.pop("status_A93")
train_df.pop("status_A94")

test_df.pop("status_A91")
test_df.pop("status_A92")
test_df.pop("status_A93")
test_df.pop("status_A94")

     0      0.0
     1      0.0
     2      0.0
     3      0.0
     4      0.0
           ...
     195    0.0
     196    0.0
     197    1.0
     198    0.0
     199    0.0
     Name: status_A94, Length: 200, dtype: float64
```

```
#Drop OtherInstallPlans
train_df.pop("plans_A141")
train_df.pop("plans_A142")
train_df.pop("plans_A143")

test_df.pop("plans_A141")
test_df.pop("plans_A142")
test_df.pop("plans_A143")

     0      0.0
     1      1.0
     2      1.0
     3      1.0
     4      1.0
           ...
     195    1.0
     196    1.0
     197    1.0
     198    1.0
     199    1.0
     Name: plans_A143, Length: 200, dtype: float64
```

```
# k values
kVals = [5,10,15, 20]

# Save the accuracies of each value of kVal in [accuracies] variable
accuracies = []

# Loop over values of k for the k-Nearest Neighbor classifier
for k in kVals:
  # Initialize a k-NN classifier with k neighbors
  my_classifier = KNeighborsClassifier(n_neighbors= k)

  # Calculate the 5 fold cross validation scores using cross_val_score
  # cv parameter: number of folds, in our case it must be 5
  scores = cross_val_score(my_classifier, train_df, train_label, cv = 5)
  # Stores the average accuracies of the scores in accuracies variable, you can use mean m
  accuracies.append(mean(scores))

print(accuracies)
```

```
     [0.7075, 0.705, 0.7225, 0.7162499999999999]
```

```
deneme = KNeighborsClassifier(n_neighbors= 5)
deneme.fit(train_df, train_label)
# Estimate the prediction of the test data
new_pred = deneme.predict(test_df)

# Print accuracy of test data
accuracy = accuracy_score(test_label, new_pred)
print(accuracy)
```

```
     0.705
```

## 10) Notebook & Report

**Notebook:** We may just look at your notebook results; so make sure each cell is run and outputs are there.

**Report:** Write an at most 1/2 page summary of your approach to this problem at the end of your notebook; this should be like an abstract of a paper or the executive summary.

**Must include statements such as:**

( Include the problem definition: 1-2 lines )

(Talk about any preprocessing you do, How you handle missing values and categorical features)

( Give the average validation accuracies for different k values and standard deviations between 5 folds of each k values, state which one you selected)

( State what your test results are with the chosen method, parameters: e.g. "We have obtained the best results with the ….. classifier (parameters=....) , giving classification accuracy of …% on test data….""

State if there is any **bonus** work...

You will get full points from here as long as you have a good (enough) summary of your work, regardless of your best performance or what you have decided to talk about in the last few lines.

## 11) Submission

Please submit your **"share link" INLINE in Sucourse submissions**. That is we should be able to click on the link and go there and run (and possibly also modify) your code.

For us to be able to modify, in case of errors etc, **you should get your "share link" as **share with anyone in edit mode**

**Also submit your notebook as pdf as attachment**, choose print and save as PDF, save with hw2-lastname-firstname.pdf to facilitate grading.

# REPORT

Our task is building a k-NN classifier to classify people as bad or good risks for the german credit dataset in this assignment. **Firstly**, I loaded the datasets as training and test data. Then, I analyze the dataset by using ".shape(), .info() functions". I realized that there are both **numerical and categorical features**. Thus, I need to play these features to make them proper for our k-NN classifier. Before going that **in part 4** I need to extract these labels from training and test data before training our classifier. **In part 5**, I checked our training data if it includes any missing value and I realized that there is a column **"Housing"** with missing values. I fill the missing values of training and test set with **mode of training data for both of them**. **In part 6**, I transformed categorical / ordinal features for appropriate way. I chose "Status of existing checking account", "Credit history", "Savings account", "Employment Since" as **ordinal features** so I gave them "orders" by using .replace() and dictionaries. On the other hand, I chose "Personal Status", "Property", "OtherInstallPlans", "Housing" as **Categorical features** and I used .getdummies() function. **In part 7,** I trained my model by using 5 fold cross validation with different k values and I recorded accuracies for each different k value. **The result table is below**:

## Accuracy table for different k values

| k value | Accuracy |
|---|---|
| 5 | 0.675 |
| 10 | 0.7075 |
| **15** | **0.71** |

I chose the **k = 15** for testing my model. Before testing I trained my model again with my entire training set with k = 15. Then, I tested my model. The result:

| k value | Accuracy |
|---|---|
| 15 | 0.665 |

**In Bonus Part**, Firstly, I decided to "Personal Status" and "OtherInstallingPlans" are not related to credit card risk so, I removed these features from my training and test data. I also modified my label sets as binary classification "1" and "0" by using .replace() function. Then, I used **normalization** on my real numeric values to fit between 0 and 1. Then, I tried several different k values and I chose **k as 5. The result:**

| k value | Accuracy |
|---|---|
| 5 | 0.705 |