Attitude

```
# import data
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.inspection import permutation_importance
from sklearn.tree import plot_tree
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import export_graphviz
import graphviz
from sklearn.ensemble import RandomForestRegressor
from yellowbrick.model_selection import FeatureImportances
import seaborn as sns

# Remove the cases with value 8 in 'attitude_before'
data = data[data['attitude_before'] != 8]

# Replace '#NULL!' with NaN in the relevant columns
data['gender'].replace('#NULL!', np.nan, inplace=True)
data['experience_recoded'].replace('#NULL!', np.nan, inplace=True)

# Remove the rows with NaN in the relevant columns
data = data.dropna(subset=['gender', 'experience_recoded'])

# read data
data = pd.read_csv('Dataset.csv', delimiter=';')

# define target variable and predictor variables
y = data['attitude_before']
X = data[['country', 'age_categories', 'knowledge', 'interest_in_technology', 'gender',
'experience_recoded', 'residence', 'income']]

# One-Hot-Encoding of categorical variables
X = pd.get_dummies(X, columns=['country', 'gender', 'income'], drop_first=False)

# step 1: cross-validation and model performance
rf = RandomForestRegressor(n_estimators=1000, random_state=42)
kf = KFold(n_splits=10, shuffle=True, random_state=42)
scores_rf = -cross_val_score(rf, X, y, cv=kf, scoring='neg_mean_squared_error')
rmse_rf = np.sqrt(np.mean(scores_rf))
print(f'RMSE für Random Forest (Cross-Validation): {rmse_rf}')

# step 2: hyperparameter-optimization
param_grid_rf = {
    'n_estimators': [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
    'max_depth': list(range(2, 11)),  # Test max_depth from 2 to 10
    'min_samples_split': [30, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750,
800, 850, 900, 950, 1000]
}
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=kf,
scoring='neg_mean_squared_error')
grid_search_rf.fit(X, y)
best_rf = grid_search_rf.best_estimator_
best_alpha_rf = best_rf.ccp_alpha

print(f'best combination: ccp_alpha={best_alpha_rf}, n_estimators={best_rf.n_estimators},
max_depth={best_rf.max_depth}, min_samples_split={best_rf.min_samples_split}')
y_pred_best_rf = best_rf.predict(X)
rmse_best_rf = np.sqrt(mean_squared_error(y, y_pred_best_rf))
print(f'RMSE for random forest (best combination): {rmse_best_rf}')
```

# step 3: test robustness (feature importance)

```
feature_importance_rf = best_rf.feature_importances_
```

# step 4: feature importance (adapted for dummy-coded variables)

```
def plot_feature_importance(model, feature_names):
    # group dummy-coded variables "gender", "country" and “income”
    dummy_feature_groups = {"gender": [], "country": [], "income": []}
    other_features = []

    feature_names = list(feature_names)  # concert index in a list

    for feature_name in feature_names:
        if feature_name.startswith("gender_"):
            dummy_feature_groups["gender"].append(feature_name)
        elif feature_name.startswith("country_"):
            dummy_feature_groups["country"].append(feature_name)
        elif feature_name.startswith("income_"):
            dummy_feature_groups["income"].append(feature_name)
        else:
            other_features.append(feature_name)

    grouped_feature_importance = {}

    for group_name, group_features in dummy_feature_groups.items():
        # calculate average importance of dummy-variables
        group_importance = np.mean([model.feature_importances_[feature_names.index(feature)] for
feature in group_features])
        grouped_feature_importance[group_name] = group_importance

    # add importance of other features
    for feature_name in other_features:
        grouped_feature_importance[feature_name] =
model.feature_importances_[feature_names.index(feature_name)]

    # sort features according to their importance
    sorted_features = sorted(grouped_feature_importance.items(), key=lambda x: x[1], reverse=True)
    # extract feature names and their importance
    feature_names, feature_importance = zip(*sorted_features)

    # adjust plot size
    plt.figure(figsize=(16, 12))
    plt.bar(range(len(feature_names)), feature_importance)
    plt.xticks(range(len(feature_names)), feature_names, rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Relative importance')
    plt.title('Importance of features (attitude_before)')

    # store plot as png-file
    plt.savefig('feature_importance_plot.png', bbox_inches='tight')

# visualize feature importance
plot_feature_importance(best_rf, X.columns)

# step 5: permutation feature importance
perm_importance_rf = permutation_importance(best_rf, X, y, n_repeats=30, random_state=42)

# step 6: visualize permutated feature importance (adapted for dummy-coded variables)
def plot_permutation_importance(perm_importance, feature_names, plot_title):
    # group dummy-coded variables "gender", "country" and “income”
    dummy_feature_groups = {"gender": [], "country": [], "income": []}
    other_features = []

    feature_names = list(feature_names)  # concert index in a list

    for feature_name in feature_names:
        if feature_name.startswith("gender_"):
            dummy_feature_groups["gender"].append(feature_name)
        elif feature_name.startswith("country_"):
            dummy_feature_groups["country"].append(feature_name)
        elif feature_name.startswith("income_"):
            dummy_feature_groups["income"].append(feature_name)
        else:
            other_features.append(feature_name)

    grouped_importance = {}

    for group_name, group_features in dummy_feature_groups.items():
        # calculate average importance of dummy-variables
        group_importance = np.mean([perm_importance[feature_names.index(feature)] for feature in
group_features])
        grouped_importance[group_name] = group_importance

    # add importance of other features
    for feature_name in other_features:
        grouped_importance[feature_name] = perm_importance[feature_names.index(feature_name)]

    # sort features according to their importance
    sorted_importance = sorted(grouped_importance.items(), key=lambda x: x[1], reverse=True)
    # extract feature names and their importance
    feature_names, importance_values = zip(*sorted_importance)

    # plot imputed feature importance
    plt.figure(figsize=(12, 8))
    plt.bar(range(len(feature_names)), importance_values)
    plt.xticks(range(len(feature_names)), feature_names, rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Imputed feature importance')
    plt.title(plot_title)
    plt.show()

# visualize permutated feature importance
plot_permutation_importance(perm_importance_rf.importances_mean, X.columns, 'Imputed
feature importance (attitude_before)')
```

# step 7: calculate out-of-bag-error

```
rf = RandomForestRegressor(n_estimators=best_rf.n_estimators, oob_score=True,
random_state=42)
rf.fit(X, y)
oob_predictions = rf.oob_prediction_
oob_error = 1 - rf.score(X, y)
print(f'Out-of-Bag (OOB) Error: {oob_error}')
```

# step 8: residual plots

```
residuals = y - y_pred_best_rf
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_best_rf, residuals, alpha=0.5)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('predictions')
plt.ylabel('residuals')
plt.title('residual plot')
plt.show()
```

# step 9: heatmap

```
correlation_matrix = X.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('correlation matrix of features')
plt.show()
```

# step 10: partial dependence plots (PDP) without aggregating dummy-variables

```
def plot_partial_dependence_without_aggregation(model, X, feature_names):
    # plot of all features
    fig, ax = plt.subplots(figsize=(12, 8))

    # colors
    colors = ['blue', 'red', 'green', 'purple']

    # partial dependence plots for age_categories', 'knowledge', 'experience_recoded', 'residence',
'interest_in_technology
```

```
    for feature_name, color in zip([age_categories', 'knowledge', 'experience_recoded', 'residence',
'interest_in_technology], colors):
        values = np.unique(X[feature_name])
        average_predictions = []
        for value in values:
            X_temp = X.copy()
            X_temp[feature_name] = value
            predictions = model.predict(X_temp)
            average_predictions.append(np.mean(predictions))
        ax.plot(values, average_predictions, label=feature_name, color=color, marker='o', linestyle='-')

    # aggregated PDP for 'country'
    # dummy for country
    countries = ["country_1", "country_2", "country_3", "country_4", "country_5", "country_6"]

    # empty lists for countries and average predictions
    country_names = []
    country_average_predictions = []

    # scroll through countries
    for i, country in enumerate(countries):
        country_data = X.copy()
        country_data[countries] = False  # set all countries to False
        country_data[country] = True  # set the referring country to True

        # calculate prediction for country
        predictions = model.predict(country_data)
        country_average_predictions.append(np.mean(predictions))
        country_names.append(i + 1)  # use i + 1 for the x-axis label

    # aggregated PDP for 'gender'
    # dummy for gender
    genders = ["gender_1", "gender_2"]

    # empty lists for genders and average predictions
    gender_names = []
    gender_average_predictions = []

    # scroll through genders
    for i, gender in enumerate(genders):
        gender_data = X.copy()
        gender_data[genders] = False  # set all genders to False
        gender_data[gender] = True  # set the referring gender to True

        # calculate prediction for gender
        predictions = model.predict(gender_data)
        gender_average_predictions.append(np.mean(predictions))
        gender_names.append(i + 1)  # use i + 1 for the x-axis label

    # aggregated PDP for 'income'
    # dummy for income
    incomes = ["income_0", "income_1", "income_2", "income_3"]
    # empty lists for incomes and average predictions
    income_names = []
    income_average_predictions = []

    # scroll through incomes
    for i, income in enumerate(incomes):
        income_data = X.copy()
        income_data[incomes] = False  # set all incomes to False
        income_data[income] = True  # set the referring income to True

        # calculate prediction for income
        predictions = model.predict(income_data)
        income_average_predictions.append(np.mean(predictions))
        income_names.append(i + 1)  # use i + 1 for the x-axis label

    # plot for aggregated PDPs
    ax.plot(country_names, country_average_predictions, label='country', color='orange', marker='o',
linestyle='-')
    ax.plot(gender_names, gender_average_predictions, label='gender', color='black', marker='o',
linestyle='-')
    ax.plot(income_names, income_average_predictions, label='income', color='brown', marker='o',
linestyle='-')

    ax.set_xlabel('Feature Values')
    ax.set_ylabel('Average Predictions')
    ax.set_title('Partial Dependence Plots (attitude_before)')
    ax.set_xticks(np.arange(1, 8))  # customized x-axis label from 1 to 7
    ax.set_yticks(np.arange(3, 8))  # customized x-axis label from 3 to 7
    ax.legend()

    # save plot as png-file and center it
    plt.savefig('partial_dependence_plot.png', bbox_inches='tight')

    # show plot
    plt.show()

# create partial dependence plots without aggregation of dummy-variables
plot_partial_dependence_without_aggregation(best_rf, X, X.columns)
```

Concerns

# import data

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.inspection import permutation_importance
from sklearn.tree import plot_tree
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import export_graphviz
import graphviz
from sklearn.ensemble import RandomForestRegressor
from yellowbrick.model_selection import FeatureImportances
import seaborn as sns

# Remove the cases with value 8 in ‘concerns’
data = data[data['concerns'] != 8]

# Replace '#NULL!' with NaN in the relevant columns
data['gender'].replace('#NULL!', np.nan, inplace=True)
data['experience_recoded'].replace('#NULL!', np.nan, inplace=True)

# Remove the rows with NaN in the relevant columns
data = data.dropna(subset=['gender', 'experience_recoded'])

# read data
data = pd.read_csv('Dataset.csv', delimiter=';')

# define target variable and predictor variables
y = data[‘concerns’]
X = data[['country', 'age_categories', 'knowledge', 'interest_in_technology', 'gender',
'experience_recoded', 'residence', 'income']]

# Convert the target variable to numeric values
y = y.str.replace(',', '.').astype(float)

# One-Hot-Encoding of categorical variables
X = pd.get_dummies(X, columns=['country', 'gender', 'income'], drop_first=False)

# step 1: cross-validation and model performance
rf = RandomForestRegressor(n_estimators=1000, random_state=42)
kf = KFold(n_splits=10, shuffle=True, random_state=42)
scores_rf = -cross_val_score(rf, X, y, cv=kf, scoring='neg_mean_squared_error')
rmse_rf = np.sqrt(np.mean(scores_rf))
print(f'RMSE für Random Forest (Cross-Validation): {rmse_rf}')

# step 2: hyperparameter-optimization
param_grid_rf = {
   'n_estimators': [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
   'max_depth': list(range(2, 11)),  # Test max_depth from 2 to 10
   'min_samples_split': [30, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750,
800, 850, 900, 950, 1000]
}
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=kf,
scoring='neg_mean_squared_error')
grid_search_rf.fit(X, y)
best_rf = grid_search_rf.best_estimator_
best_alpha_rf = best_rf.ccp_alpha
```

```
print(f'best combination: ccp_alpha={best_alpha_rf}, n_estimators={best_rf.n_estimators},
max_depth={best_rf.max_depth}, min_samples_split={best_rf.min_samples_split}')
y_pred_best_rf = best_rf.predict(X)
rmse_best_rf = np.sqrt(mean_squared_error(y, y_pred_best_rf))
print(f'RMSE for random forest (best combination): {rmse_best_rf}')
```

# step 3: test robustness (feature importance)

```
feature_importance_rf = best_rf.feature_importances_
```

# step 4: feature importance (adapted for dummy-coded variables)

```
def plot_feature_importance(model, feature_names):
   # group dummy-coded variables "gender", "country" and “income”
   dummy_feature_groups = {"gender": [], "country": [], "income": []}
   other_features = []

   feature_names = list(feature_names)  # concert index in a list

   for feature_name in feature_names:
      if feature_name.startswith("gender_"):
         dummy_feature_groups["gender"].append(feature_name)
      elif feature_name.startswith("country_"):
         dummy_feature_groups["country"].append(feature_name)
      elif feature_name.startswith("income_"):
         dummy_feature_groups["income"].append(feature_name)
      else:
         other_features.append(feature_name)

   grouped_feature_importance = {}

   for group_name, group_features in dummy_feature_groups.items():
      # calculate average importance of dummy-variables
      group_importance = np.mean([model.feature_importances_[feature_names.index(feature)] for
feature in group_features])
      grouped_feature_importance[group_name] = group_importance

   # add importance of other features
   for feature_name in other_features:
      grouped_feature_importance[feature_name] =
model.feature_importances_[feature_names.index(feature_name)]

   # sort features according to their importance
   sorted_features = sorted(grouped_feature_importance.items(), key=lambda x: x[1], reverse=True)

   # extract feature names and their importance
   feature_names, feature_importance = zip(*sorted_features)

   # adjust plot size
   plt.figure(figsize=(16, 12))
   plt.bar(range(len(feature_names)), feature_importance)
   plt.xticks(range(len(feature_names)), feature_names, rotation=45)
   plt.xlabel('Features')
   plt.ylabel('Relative importance')
```

```
    plt.title('Importance of features (concerns)')

    # store plot as png-file
    plt.savefig('feature_importance_plot.png', bbox_inches='tight')

# visualize feature importance
plot_feature_importance(best_rf, X.columns)
```

# step 5: permutation feature importance

```
perm_importance_rf = permutation_importance(best_rf, X, y, n_repeats=30, random_state=42)
```

# step 6: visualize permutated feature importance (adapted for dummy-coded variables)

```
def plot_permutation_importance(perm_importance, feature_names, plot_title):
    # group dummy-coded variables "gender", "country" and “income”
    dummy_feature_groups = {"gender": [], "country": [], "income": []}
    other_features = []

    feature_names = list(feature_names)  # concert index in a list

    for feature_name in feature_names:
        if feature_name.startswith("gender_"):
            dummy_feature_groups["gender"].append(feature_name)
        elif feature_name.startswith("country_"):
            dummy_feature_groups["country"].append(feature_name)
        elif feature_name.startswith("income_"):
            dummy_feature_groups["income"].append(feature_name)
        else:
            other_features.append(feature_name)

    grouped_importance = {}

    for group_name, group_features in dummy_feature_groups.items():
        # calculate average importance of dummy-variables
        group_importance = np.mean([perm_importance[feature_names.index(feature)] for feature in
group_features])
        grouped_importance[group_name] = group_importance

    # add importance of other features
    for feature_name in other_features:
        grouped_importance[feature_name] = perm_importance[feature_names.index(feature_name)]

    # sort features according to their importance
    sorted_importance = sorted(grouped_importance.items(), key=lambda x: x[1], reverse=True)

    # extract feature names and their importance
    feature_names, importance_values = zip(*sorted_importance)

    # plot imputed feature importance
    plt.figure(figsize=(12, 8))
    plt.bar(range(len(feature_names)), importance_values)
    plt.xticks(range(len(feature_names)), feature_names, rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Imputed feature importance')
    plt.title(plot_title)
    plt.show()

# visualize permutated feature importance
plot_permutation_importance(perm_importance_rf.importances_mean, X.columns, 'Imputed
feature importance (concerns)')

# step 7: calculate out-of-bag-error
rf = RandomForestRegressor(n_estimators=best_rf.n_estimators, oob_score=True,
random_state=42)
rf.fit(X, y)
oob_predictions = rf.oob_prediction_
oob_error = 1 - rf.score(X, y)
print(f'Out-of-Bag (OOB) Error: {oob_error}')

# step 8: residual plots
residuals = y - y_pred_best_rf
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_best_rf, residuals, alpha=0.5)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('predictions')
plt.ylabel('residuals')
plt.title('residual plot')
plt.show()

# step 9: heatmap
correlation_matrix = X.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('correlation matrix of features')
plt.show()

# step 10: partial dependence plots (PDP) without aggregating dummy-variables
def plot_partial_dependence_without_aggregation(model, X, feature_names):
    # plot of all features
    fig, ax = plt.subplots(figsize=(12, 8))

    # colors
    colors = ['blue', 'red', 'green', 'purple']

    # partial dependence plots for age_categories', 'knowledge', 'experience_recoded', 'residence',
'interest_in_technology
    for feature_name, color in zip([age_categories', 'knowledge', 'experience_recoded', 'residence',
'interest_in_technology], colors):
        values = np.unique(X[feature_name])
        average_predictions = []
        for value in values:
            X_temp = X.copy()
            X_temp[feature_name] = value
            predictions = model.predict(X_temp)
            average_predictions.append(np.mean(predictions))
        ax.plot(values, average_predictions, label=feature_name, color=color, marker='o', linestyle='-')
```

```
# aggregated PDP for 'country'
# dummy for country
countries = ["country_1", "country_2", "country_3", "country_4", "country_5", "country_6"]

# empty lists for countries and average predictions
country_names = []
country_average_predictions = []

# scroll through countries
for i, country in enumerate(countries):
    country_data = X.copy()
    country_data[countries] = False  # set all countries to False
    country_data[country] = True  # set the referring country to True

    # calculate prediction for country
    predictions = model.predict(country_data)
    country_average_predictions.append(np.mean(predictions))
    country_names.append(i + 1)  # use i + 1 for the x-axis label

# aggregated PDP for 'gender'
# dummy for gender
genders = ["gender_1", "gender_2"]

# empty lists for genders and average predictions
gender_names = []
gender_average_predictions = []

# scroll through genders
for i, gender in enumerate(genders):
    gender_data = X.copy()
    gender_data[genders] = False  # set all genders to False
    gender_data[gender] = True  # set the referring gender to True

    # calculate prediction for gender
    predictions = model.predict(gender_data)
    gender_average_predictions.append(np.mean(predictions))
    gender_names.append(i + 1)  # use i + 1 for the x-axis label

# aggregated PDP for 'income'
# dummy for income
incomes = ["income_0", "income_1", "income_2", "income_3"]

# empty lists for incomes and average predictions
income_names = []
income_average_predictions = []

# scroll through incomes
for i, income in enumerate(incomes):
    income_data = X.copy()
    income_data[incomes] = False  # set all incomes to False
    income_data[income] = True  # set the referring income to True

    # calculate prediction for income
```

```
        predictions = model.predict(income_data)
        income_average_predictions.append(np.mean(predictions))
        income_names.append(i + 1)  # use i + 1 for the x-axis label

    # plot for aggregated PDPs
    ax.plot(country_names, country_average_predictions, label='country', color='orange', marker='o',
linestyle='-')
    ax.plot(gender_names, gender_average_predictions, label='gender', color='black', marker='o',
linestyle='-')
    ax.plot(income_names, income_average_predictions, label='income', color='brown', marker='o',
linestyle='-')

    ax.set_xlabel('Feature Values')
    ax.set_ylabel('Average Predictions')
    ax.set_title('Partial Dependence Plots (concerns)')
    ax.set_xticks(np.arange(1, 8))  # customized x-axis label from 1 to 7
    ax.set_yticks(np.arange(3, 8))  # customized x-axis label from 3 to 7
    ax.legend()

    # save plot as png-file and center it
    plt.savefig('partial_dependence_plot.png', bbox_inches='tight')

    # show plot
    plt.show()

# create partial dependence plots without aggregation of dummy-variables
plot_partial_dependence_without_aggregation(best_rf, X, X.columns)
```

Approval for private and commercial use cases

# import data

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.inspection import permutation_importance
from sklearn.tree import plot_tree
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import export_graphviz
import graphviz
from sklearn.ensemble import RandomForestRegressor
from yellowbrick.model_selection import FeatureImportances
import seaborn as sns
```

```
# Remove the cases with value 8 in ‘use_case_private_commercial’
data = data[data[‘use_case_private_commercial’] != 8]

# Replace '#NULL!' with NaN in the relevant columns
data['gender'].replace('#NULL!', np.nan, inplace=True)
data['experience_recoded'].replace('#NULL!', np.nan, inplace=True)

# Remove the rows with NaN in the relevant columns
data = data.dropna(subset=['gender', 'experience_recoded'])

# read data
data = pd.read_csv('Dataset.csv', delimiter=';')

# define target variable and predictor variables
y = data[‘use_case_private_commercial’]
X = data[['country', 'age_categories', 'knowledge', 'interest_in_technology', 'gender',
'experience_recoded', 'residence', 'income']]

# Convert the target variable to numeric values
y = y.str.replace(',', '.').astype(float)

# One-Hot-Encoding of categorical variables
X = pd.get_dummies(X, columns=['country', 'gender', 'income'], drop_first=False)

# step 1: cross-validation and model performance
rf = RandomForestRegressor(n_estimators=1000, random_state=42)
kf = KFold(n_splits=10, shuffle=True, random_state=42)
scores_rf = -cross_val_score(rf, X, y, cv=kf, scoring='neg_mean_squared_error')
rmse_rf = np.sqrt(np.mean(scores_rf))
print(f'RMSE für Random Forest (Cross-Validation): {rmse_rf}')

# step 2: hyperparameter-optimization
param_grid_rf = {
   'n_estimators': [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
   'max_depth': list(range(2, 11)),  # Test max_depth from 2 to 10
   'min_samples_split': [30, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750,
800, 850, 900, 950, 1000]
}
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=kf,
scoring='neg_mean_squared_error')
grid_search_rf.fit(X, y)
best_rf = grid_search_rf.best_estimator_
best_alpha_rf = best_rf.ccp_alpha

print(f'best combination: ccp_alpha={best_alpha_rf}, n_estimators={best_rf.n_estimators},
max_depth={best_rf.max_depth}, min_samples_split={best_rf.min_samples_split}')
y_pred_best_rf = best_rf.predict(X)
rmse_best_rf = np.sqrt(mean_squared_error(y, y_pred_best_rf))
print(f'RMSE for random forest (best combination): {rmse_best_rf}')

# step 3: test robustness (feature importance)
feature_importance_rf = best_rf.feature_importances_
```

```
# step 4: feature importance (adapted for dummy-coded variables)
def plot_feature_importance(model, feature_names):
   # group dummy-coded variables "gender", "country" and “income”
   dummy_feature_groups = {"gender": [], "country": [], "income": []}
   other_features = []

   feature_names = list(feature_names)  # concert index in a list

   for feature_name in feature_names:
      if feature_name.startswith("gender_"):
         dummy_feature_groups["gender"].append(feature_name)
      elif feature_name.startswith("country_"):
         dummy_feature_groups["country"].append(feature_name)
      elif feature_name.startswith("income_"):
         dummy_feature_groups["income"].append(feature_name)
      else:
         other_features.append(feature_name)

   grouped_feature_importance = {}

   for group_name, group_features in dummy_feature_groups.items():
      # calculate average importance of dummy-variables
      group_importance = np.mean([model.feature_importances_[feature_names.index(feature)] for
feature in group_features])
      grouped_feature_importance[group_name] = group_importance

   # add importance of other features
   for feature_name in other_features:
      grouped_feature_importance[feature_name] =
model.feature_importances_[feature_names.index(feature_name)]

   # sort features according to their importance
   sorted_features = sorted(grouped_feature_importance.items(), key=lambda x: x[1], reverse=True)

   # extract feature names and their importance
   feature_names, feature_importance = zip(*sorted_features)

   # adjust plot size
   plt.figure(figsize=(16, 12))
   plt.bar(range(len(feature_names)), feature_importance)
   plt.xticks(range(len(feature_names)), feature_names, rotation=45)
   plt.xlabel('Features')
   plt.ylabel('Relative importance')
   plt.title('Importance of features (use_case_private_commercial)')

   # store plot as png-file
   plt.savefig('feature_importance_plot.png', bbox_inches='tight')

# visualize feature importance
plot_feature_importance(best_rf, X.columns)

# step 5: permutation feature importance
perm_importance_rf = permutation_importance(best_rf, X, y, n_repeats=30, random_state=42)
```

```
# step 6: visualize permutated feature importance (adapted for dummy-coded variables)
def plot_permutation_importance(perm_importance, feature_names, plot_title):
    # group dummy-coded variables "gender", "country" and “income”
    dummy_feature_groups = {"gender": [], "country": [], "income": []}
    other_features = []

    feature_names = list(feature_names)  # concert index in a list

    for feature_name in feature_names:
        if feature_name.startswith("gender_"):
            dummy_feature_groups["gender"].append(feature_name)
        elif feature_name.startswith("country_"):
            dummy_feature_groups["country"].append(feature_name)
        elif feature_name.startswith("income_"):
            dummy_feature_groups["income"].append(feature_name)
        else:
            other_features.append(feature_name)

    grouped_importance = {}

    for group_name, group_features in dummy_feature_groups.items():
        # calculate average importance of dummy-variables
        group_importance = np.mean([perm_importance[feature_names.index(feature)] for feature in
group_features])
        grouped_importance[group_name] = group_importance

    # add importance of other features
    for feature_name in other_features:
        grouped_importance[feature_name] = perm_importance[feature_names.index(feature_name)]

    # sort features according to their importance
    sorted_importance = sorted(grouped_importance.items(), key=lambda x: x[1], reverse=True)

    # extract feature names and their importance
    feature_names, importance_values = zip(*sorted_importance)

    # plot imputed feature importance
    plt.figure(figsize=(12, 8))
    plt.bar(range(len(feature_names)), importance_values)
    plt.xticks(range(len(feature_names)), feature_names, rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Imputed feature importance')
    plt.title(plot_title)
    plt.show()

# visualize permutated feature importance
plot_permutation_importance(perm_importance_rf.importances_mean, X.columns, 'Imputed
feature importance (use_case_private_commercial)')

# step 7: calculate out-of-bag-error
rf = RandomForestRegressor(n_estimators=best_rf.n_estimators, oob_score=True,
random_state=42)
```

```
rf.fit(X, y)
oob_predictions = rf.oob_prediction_
oob_error = 1 - rf.score(X, y)
print(f'Out-of-Bag (OOB) Error: {oob_error}')

# step 8: residual plots
residuals = y - y_pred_best_rf
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_best_rf, residuals, alpha=0.5)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('predictions')
plt.ylabel('residuals')
plt.title('residual plot')
plt.show()

# step 9: heatmap
correlation_matrix = X.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('correlation matrix of features')
plt.show()

# step 10: partial dependence plots (PDP) without aggregating dummy-variables
def plot_partial_dependence_without_aggregation(model, X, feature_names):
   # plot of all features
   fig, ax = plt.subplots(figsize=(12, 8))

   # colors
   colors = ['blue', 'red', 'green', 'purple']

   # partial dependence plots for age_categories', 'knowledge', 'experience_recoded', 'residence',
'interest_in_technology
   for feature_name, color in zip([age_categories', 'knowledge', 'experience_recoded', 'residence',
'interest_in_technology], colors):
      values = np.unique(X[feature_name])
      average_predictions = []
      for value in values:
         X_temp = X.copy()
         X_temp[feature_name] = value
         predictions = model.predict(X_temp)
         average_predictions.append(np.mean(predictions))
      ax.plot(values, average_predictions, label=feature_name, color=color, marker='o', linestyle='-')

   # aggregated PDP for 'country'
   # dummy for country
   countries = ["country_1", "country_2", "country_3", "country_4", "country_5", "country_6"]

   # empty lists for countries and average predictions
   country_names = []
   country_average_predictions = []

   # scroll through countries
   for i, country in enumerate(countries):
        country_data = X.copy()
        country_data[countries] = False  # set all countries to False
        country_data[country] = True  # set the referring country to True

        # calculate prediction for country
        predictions = model.predict(country_data)
        country_average_predictions.append(np.mean(predictions))
        country_names.append(i + 1)  # use i + 1 for the x-axis label

    # aggregated PDP for 'gender'
    # dummy for gender
    genders = ["gender_1", "gender_2"]

    # empty lists for genders and average predictions
    gender_names = []
    gender_average_predictions = []

    # scroll through genders
    for i, gender in enumerate(genders):
        gender_data = X.copy()
        gender_data[genders] = False  # set all genders to False
        gender_data[gender] = True  # set the referring gender to True

        # calculate prediction for gender
        predictions = model.predict(gender_data)
        gender_average_predictions.append(np.mean(predictions))
        gender_names.append(i + 1)  # use i + 1 for the x-axis label

    # aggregated PDP for 'income'
    # dummy for income
    incomes = ["income_0", "income_1", "income_2", "income_3"]

    # empty lists for incomes and average predictions
    income_names = []
    income_average_predictions = []

    # scroll through incomes
    for i, income in enumerate(incomes):
        income_data = X.copy()
        income_data[incomes] = False  # set all incomes to False
        income_data[income] = True  # set the referring income to True

        # calculate prediction for income
        predictions = model.predict(income_data)
        income_average_predictions.append(np.mean(predictions))
        income_names.append(i + 1)  # use i + 1 for the x-axis label

    # plot for aggregated PDPs
    ax.plot(country_names, country_average_predictions, label='country', color='orange', marker='o',
linestyle='-')
    ax.plot(gender_names, gender_average_predictions, label='gender', color='black', marker='o',
linestyle='-')
```

```
    ax.plot(income_names, income_average_predictions, label='income', color='brown', marker='o',
linestyle='-')

    ax.set_xlabel('Feature Values')
    ax.set_ylabel('Average Predictions')
    ax.set_title('Partial Dependence Plots (use_case_private_commercial)')
    ax.set_xticks(np.arange(1, 8))  # customized x-axis label from 1 to 7
    ax.set_yticks(np.arange(3, 8))  # customized x-axis label from 3 to 7
    ax.legend()

    # save plot as png-file and center it
    plt.savefig('partial_dependence_plot.png', bbox_inches='tight')

    # show plot
    plt.show()

# create partial dependence plots without aggregation of dummy-variables
plot_partial_dependence_without_aggregation(best_rf, X, X.columns)
```

Approval for public and civil use cases

```
# import data
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.inspection import permutation_importance
from sklearn.tree import plot_tree
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import export_graphviz
import graphviz
from sklearn.ensemble import RandomForestRegressor
from yellowbrick.model_selection import FeatureImportances
import seaborn as sns

# Remove the cases with value 8 in ‘use_case_private_commercial’
data = data[data[‘use_case_public_civil’] != 8]

# Replace '#NULL!' with NaN in the relevant columns
data['gender'].replace('#NULL!', np.nan, inplace=True)
data['experience_recoded'].replace('#NULL!', np.nan, inplace=True)

# Remove the rows with NaN in the relevant columns
data = data.dropna(subset=['gender', 'experience_recoded'])
```

```
# read data
data = pd.read_csv('Dataset.csv', delimiter=';')

# define target variable and predictor variables
y = data[‘use_case_public_civil’]
X = data[['country', 'age_categories', 'knowledge', 'interest_in_technology', 'gender',
'experience_recoded', 'residence', 'income']]

# Convert the target variable to numeric values
y = y.str.replace(',', '.').astype(float)

# One-Hot-Encoding of categorical variables
X = pd.get_dummies(X, columns=['country', 'gender', 'income'], drop_first=False)

# step 1: cross-validation and model performance
rf = RandomForestRegressor(n_estimators=1000, random_state=42)
kf = KFold(n_splits=10, shuffle=True, random_state=42)
scores_rf = -cross_val_score(rf, X, y, cv=kf, scoring='neg_mean_squared_error')
rmse_rf = np.sqrt(np.mean(scores_rf))
print(f'RMSE für Random Forest (Cross-Validation): {rmse_rf}')

# step 2: hyperparameter-optimization
param_grid_rf = {
   'n_estimators': [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
   'max_depth': list(range(2, 11)),  # Test max_depth from 2 to 10
   'min_samples_split': [30, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750,
800, 850, 900, 950, 1000]
}
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=kf,
scoring='neg_mean_squared_error')
grid_search_rf.fit(X, y)
best_rf = grid_search_rf.best_estimator_
best_alpha_rf = best_rf.ccp_alpha

print(f'best combination: ccp_alpha={best_alpha_rf}, n_estimators={best_rf.n_estimators},
max_depth={best_rf.max_depth}, min_samples_split={best_rf.min_samples_split}')
y_pred_best_rf = best_rf.predict(X)
rmse_best_rf = np.sqrt(mean_squared_error(y, y_pred_best_rf))
print(f'RMSE for random forest (best combination): {rmse_best_rf}')

# step 3: test robustness (feature importance)
feature_importance_rf = best_rf.feature_importances_

# step 4: feature importance (adapted for dummy-coded variables)
def plot_feature_importance(model, feature_names):
   # group dummy-coded variables "gender", "country" and “income”
   dummy_feature_groups = {"gender": [], "country": [], "income": []}
   other_features = []

   feature_names = list(feature_names)  # concert index in a list

   for feature_name in feature_names:
        if feature_name.startswith("gender_"):
            dummy_feature_groups["gender"].append(feature_name)
        elif feature_name.startswith("country_"):
            dummy_feature_groups["country"].append(feature_name)
        elif feature_name.startswith("income_"):
            dummy_feature_groups["income"].append(feature_name)
        else:
            other_features.append(feature_name)

    grouped_feature_importance = {}

    for group_name, group_features in dummy_feature_groups.items():
        # calculate average importance of dummy-variables
        group_importance = np.mean([model.feature_importances_[feature_names.index(feature)] for
feature in group_features])
        grouped_feature_importance[group_name] = group_importance

    # add importance of other features
    for feature_name in other_features:
        grouped_feature_importance[feature_name] =
model.feature_importances_[feature_names.index(feature_name)]

    # sort features according to their importance
    sorted_features = sorted(grouped_feature_importance.items(), key=lambda x: x[1], reverse=True)

    # extract feature names and their importance
    feature_names, feature_importance = zip(*sorted_features)

    # adjust plot size
    plt.figure(figsize=(16, 12))
    plt.bar(range(len(feature_names)), feature_importance)
    plt.xticks(range(len(feature_names)), feature_names, rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Relative importance')
    plt.title('Importance of features (use_case_public_civil)')

    # store plot as png-file
    plt.savefig('feature_importance_plot.png', bbox_inches='tight')

# visualize feature importance
plot_feature_importance(best_rf, X.columns)

# step 5: permutation feature importance
perm_importance_rf = permutation_importance(best_rf, X, y, n_repeats=30, random_state=42)

# step 6: visualize permutated feature importance (adapted for dummy-coded variables)
def plot_permutation_importance(perm_importance, feature_names, plot_title):
    # group dummy-coded variables "gender", "country" and “income”
    dummy_feature_groups = {"gender": [], "country": [], "income": []}
    other_features = []

    feature_names = list(feature_names)  # concert index in a list
    for feature_name in feature_names:
        if feature_name.startswith("gender_"):
            dummy_feature_groups["gender"].append(feature_name)
        elif feature_name.startswith("country_"):
            dummy_feature_groups["country"].append(feature_name)
        elif feature_name.startswith("income_"):
            dummy_feature_groups["income"].append(feature_name)
        else:
            other_features.append(feature_name)

    grouped_importance = {}

    for group_name, group_features in dummy_feature_groups.items():
        # calculate average importance of dummy-variables
        group_importance = np.mean([perm_importance[feature_names.index(feature)] for feature in
group_features])
        grouped_importance[group_name] = group_importance

    # add importance of other features
    for feature_name in other_features:
        grouped_importance[feature_name] = perm_importance[feature_names.index(feature_name)]

    # sort features according to their importance
    sorted_importance = sorted(grouped_importance.items(), key=lambda x: x[1], reverse=True)

    # extract feature names and their importance
    feature_names, importance_values = zip(*sorted_importance)

    # plot imputed feature importance
    plt.figure(figsize=(12, 8))
    plt.bar(range(len(feature_names)), importance_values)
    plt.xticks(range(len(feature_names)), feature_names, rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Imputed feature importance')
    plt.title(plot_title)
    plt.show()

# visualize permutated feature importance
plot_permutation_importance(perm_importance_rf.importances_mean, X.columns, 'Imputed
feature importance (use_case_public_civil)')

# step 7: calculate out-of-bag-error
rf = RandomForestRegressor(n_estimators=best_rf.n_estimators, oob_score=True,
random_state=42)
rf.fit(X, y)
oob_predictions = rf.oob_prediction_
oob_error = 1 - rf.score(X, y)
print(f'Out-of-Bag (OOB) Error: {oob_error}')

# step 8: residual plots
residuals = y - y_pred_best_rf
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_best_rf, residuals, alpha=0.5)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('predictions')
plt.ylabel('residuals')
plt.title('residual plot')
plt.show()
```

# step 9: heatmap

```
correlation_matrix = X.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('correlation matrix of features')
plt.show()
```

# step 10: partial dependence plots (PDP) without aggregating dummy-variables

```
def plot_partial_dependence_without_aggregation(model, X, feature_names):
   # plot of all features
   fig, ax = plt.subplots(figsize=(12, 8))

   # colors
   colors = ['blue', 'red', 'green', 'purple']

   # partial dependence plots for age_categories', 'knowledge', 'experience_recoded', 'residence', 
'interest_in_technology
   for feature_name, color in zip([age_categories', 'knowledge', 'experience_recoded', 'residence', 
'interest_in_technology], colors):
      values = np.unique(X[feature_name])
      average_predictions = []
      for value in values:
         X_temp = X.copy()
         X_temp[feature_name] = value
         predictions = model.predict(X_temp)
         average_predictions.append(np.mean(predictions))
      ax.plot(values, average_predictions, label=feature_name, color=color, marker='o', linestyle='-')

   # aggregated PDP for 'country'
   # dummy for country
   countries = ["country_1", "country_2", "country_3", "country_4", "country_5", "country_6"]

   # empty lists for countries and average predictions
   country_names = []
   country_average_predictions = []

   # scroll through countries
   for i, country in enumerate(countries):
      country_data = X.copy()
      country_data[countries] = False  # set all countries to False
      country_data[country] = True  # set the referring country to True

      # calculate prediction for country
      predictions = model.predict(country_data)
      country_average_predictions.append(np.mean(predictions))
      country_names.append(i + 1)  # use i + 1 for the x-axis label
```

```
    # aggregated PDP for 'gender'
    # dummy for gender
    genders = ["gender_1", "gender_2"]

    # empty lists for genders and average predictions
    gender_names = []
    gender_average_predictions = []

    # scroll through genders
    for i, gender in enumerate(genders):
        gender_data = X.copy()
        gender_data[genders] = False  # set all genders to False
        gender_data[gender] = True  # set the referring gender to True

        # calculate prediction for gender
        predictions = model.predict(gender_data)
        gender_average_predictions.append(np.mean(predictions))
        gender_names.append(i + 1)  # use i + 1 for the x-axis label

    # aggregated PDP for 'income'
    # dummy for income
    incomes = ["income_0", "income_1", "income_2", "income_3"]

    # empty lists for incomes and average predictions
    income_names = []
    income_average_predictions = []

    # scroll through incomes
    for i, income in enumerate(incomes):
        income_data = X.copy()
        income_data[incomes] = False  # set all incomes to False
        income_data[income] = True  # set the referring income to True

        # calculate prediction for income
        predictions = model.predict(income_data)
        income_average_predictions.append(np.mean(predictions))
        income_names.append(i + 1)  # use i + 1 for the x-axis label

    # plot for aggregated PDPs
    ax.plot(country_names, country_average_predictions, label='country', color='orange', marker='o',
linestyle='-')
    ax.plot(gender_names, gender_average_predictions, label='gender', color='black', marker='o',
linestyle='-')
    ax.plot(income_names, income_average_predictions, label='income', color='brown', marker='o',
linestyle='-')

    ax.set_xlabel('Feature Values')
    ax.set_ylabel('Average Predictions')
    ax.set_title('Partial Dependence Plots (use_case_public_civil)')
    ax.set_xticks(np.arange(1, 8))  # customized x-axis label from 1 to 7
    ax.set_yticks(np.arange(3, 8))  # customized x-axis label from 3 to 7
    ax.legend()
    # save plot as png-file and center it
    plt.savefig('partial_dependence_plot.png', bbox_inches='tight')

    # show plot
    plt.show()

# create partial dependence plots without aggregation of dummy-variables
plot_partial_dependence_without_aggregation(best_rf, X, X.columns)
```