# SEN P5: Regressionsanalyse

### Vorbemerkungen

- Dieser Praktikumsversuch ist für zwei Vorlesungsblöcke angelegt (insgesamt also 180 Minuten). Ihnen steht für die einzelnen Bearbeitungsschritte daher ausreichend viel Zeit zur Verfügung.
- Im Rahmen dieser Anleitung werden Ihnen einige Codezeilen als Beispiele und Tipps vorgegeben, die Sie direkt in Ihren Code übernehmen können. Es wird dennoch empfohlen, die jeweiligen Codezeilen nicht mit Copy-Paste zu übernehmen, sondern abzutippen und sich mit jeder Codezeile ausführlich zu beschäftigen.

### Einleitung

- In diesem Praktikumsversuch soll es darum gehen, basierend auf einem vorgegebenen Datensatz[1], Preise für Häuser im Rahmen einer auf maschinellem Lernen basierenden Regressionsanalyse zu schätzen.
- Ein sekundäres Lernziel dieses Praktikumsversuchs stellt der Umgang mit Pandas Dataframes dar, die speziell für Machine-Learning-Aufgaben besonders vorteilhafte Eigenschaften aufweisen, auf die bislang jedoch nur im Rahmen einiger Spezialfälle detaillierter eingegangen wurde.

### Laden und Kennenlernen des Datensatzes

- Laden Sie den Datensatz als Pandas Dataframe mit dem Namen `housing`.
- Lassen Sie mit der Befehlszeile `print(housing.keys())` die *Keys* (d.h. die Bezeichner der einzelnen Spalten im Datensatz) auf der Konsole ausgeben.
  Die im Rahmen der Regressionsanalyse zu schätzende Zielgröße trägt den Bezeichner `median_house_value`.
- **Tipp**: Mit dem Befehl `print(housing.head(10))` werden die ersten zehn Einträge des Datensatzes strukturiert auf der Konsole dargestellt.
- Mit dem Befehl `housing.info()` werden einige Informationen aus dem Datensatz auf der Konsole ausgegeben. Hieraus ergeben sich folgende Erkenntnisse:
  - Das Merkmal `total_bedrooms` hat im Gegensatz zu den anderen Merkmalen nur 20.433 Datenpunkte.
  - Das Merkmal `ocean_proximity` ist vom Typ Object (und nicht Floating-Point).

  Diesen beiden Punkten werden wir uns später noch widmen.

[1] Den auch auf Moodle zur Verfügung gestellten Datensatz `housing.csv` finden Sie im Internet bspw. unter https://github.com/ageron/handson-ml/blob/master/datasets/housing/housing.csv.

- Um herauszufinden, welche Werte das kategorische Merkmal `ocean_proximity` aufweist und wie häufig die jeweiligen Werte im Datensatz vorkommen, kann die Codezeile `print(housing["ocean_proximity"].value_counts())` dienen. Hier wird der als Pandas Dataframe vorliegende Datensatz `housing` mit dem Key `ocean_proximity` adressiert, es wird also nur dieses Merkmal betrachtet.

- Mit der Codezeile `print(housing.describe())` werden einige Statistiken zum Datensatz auf der Konsole ausgegeben.

### Daten visualisieren

- Erstellen Sie Histogramme für alle Merkmale. Neben der bekannten Möglichkeit via `plt.hist(...)` können Plot-Funktionen wie etwa `hist()` auch direkt über das Pandas Dataframe aufgerufen werden: `housing.hist(bins=50, figsize=(15,8))`
  **Tipp**: Durch den Parameter `bins=50` werden die Merkmale jeweils in 50 Zahlenwertbereiche unterteilt. Der zweite Parameter `figsize=(15,8)` dient der größeren Darstellung der Diagramme.

- Stellen Sie nun die Koordinatenmerkmale `longitude` (x-Achse) und `latitude` (y-Achse) als Scatterplot dar.
  **Tipp**: Dies kann bspw. über die Codezeile `plt.scatter(housing["longitude"], housing["latitude"])` geschehen, syntaktisch praktischer ist jedoch `housing.plot(kind="scatter", x="longitude", y="latitude")`.
  **Erkenntnis**: Deutlich ist die geographische Struktur von Kalifornien erkennbar.

- Um noch mehr Informationen aus dem letzten Diagramm zu erlangen, soll nun die Zielgröße (`median_house_value`) farblich codiert, sowie das Merkmal `population` als Größe der Punkte im Scatterplot dargestellt werden. Dies kann durch folgenden Code bewerkstelligt werden:

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
             s=housing["population"]/100, c="median_house_value",
             cmap=plt.get_cmap("jet"), colorbar=True,
             figsize=(10,7), label="Population")
plt.legend()
```

  Darin dient der Parameter `alpha` der Transparenz der Datenpunkte im Diagramm, `s` der Größe der Datenpunkte und `c` der Farbcodierung.

### Korrelationen suchen und visualisieren

- Um sich weiter mit den Daten vertraut zu machen bietet es sich an, zunächst zu prüfen, welche Merkmale besonders stark mit der Zielgröße korrelieren. Pandas ermöglicht die Ermittlung einer Korrelationsmatrix (= sämtliche paarweisen Korrelationskoeffizienten aller Spalten im Dataframe) mittels `corr_matrix = housing.corr()`.

- Anschließend erlaubt Ihnen folgende Befehlszeile die Ausgabe sämtlicher Korrelationskoeffizienten mit der Größe `median_house_value` (und damit die Korrelation der Zielgröße mit den einzelnen Merkmalen) in absteigender Reihenfolge:
  `print(corr_matrix["median_house_value"].sort_values(ascending=False))`

- Die Zielgröße scheint stark mit dem Merkmal `median_income` zu korrelieren. Stellen Sie daher die Zielgröße als Funktion von `median_income` in einem Scatterplot dar. Was fällt Ihnen auf?

- **Anmerkung**: Auf eine Merkmalsauswahl wird an dieser Stelle der Einfachheit halber verzichtet.

### Unvollständiges Merkmal entfernen

- Für das Merkmal `total_bedrooms` fehlen einige Datenpunkte. Im Rahmen der Korrelationsanalyse zeigte sich, dass dieses Merkmal kaum mit der Zielgröße korreliert. Die naheliegende Vorgehensweise ist daher, dieses Merkmal einfach aus dem Datensatz zu löschen. Dies lässt sich im Pandas Dataframe am einfachsten mit dem `drop`-Befehl bewerkstelligen: `housing = housing.drop(["total_bedrooms"], axis=1)`

- Lassen Sie sich die Keys erneut auf der Konsole ausgeben und überprüfen Sie, dass das Merkmal `total_bedrooms` auch tatsächlich gelöscht wurde: `print(housing.keys())`

### Train-Test-Split

- Unterteilen Sie nun die Daten in Trainings- und Testdaten. Hierzu bietet es sich an, aus dem Pandas Dataframe zunächst sämtliche Merkmale in eine Merkmalsmatrix `X` zu überführen und die Zielgröße in einem Labelvektor `y` abzuspeichern:

  ```
  X = housing.drop(["median_house_value"], axis=1)
  y = housing["median_house_value"].copy()
  ```

- **Anmerkung**: Es sei angemerkt, dass sowohl die Merkmalsmatrix `X` als auch der Labelvektor `y` weiterhin als Pandas Dataframe vorliegen.

- Trotzdem kann der Train-Test-Split mit der Scikit-Learn Funktion `train_test_split` durchgeführt werden. Unterteilen Sie die Daten so, dass 20 % des Datensatzes als Testdaten vorgehalten werden. Nennen Sie die Trainingsdaten `X_train` und `y_train`, die Testdaten `X_test` und `y_test`.

- **Hinweis**: Der `stratify`-Paramter der Train-Test-Split-Funktion ist hier obsolet, da es sich bei der Zielgröße `y` nicht um diskrete Klassenlabels sondern um eine kontinuierliche Variable handelt.

### Merkmale One-Hot codieren und skalieren

- Eingangs hat sich gezeigt, dass es sich bei `ocean_proximity` um ein kategorisches Merkmal handelt. Führen Sie daher für den Trainingsdatensatz eine One-Hot-Codierung durch.
- Skalieren Sie anschließend den Trainingsdatenbestand mit dem MinMax-Scaler.
- **Anmerkung**: Bei einer Skalierung der Daten nach einer One-Hot Codierung hat der MinMax-Scaler den Vorteil, dass er an den Merkmalswerten der One-Hot codierten Merkmale (0 bzw. 1) nichts ändert.
- **Tipp**: Nach der Transformation der Daten mit dem MinMax-Scaler liegt der Datensatz als NumPy-Array vor. Um aus einer Merkmalsmatrix `X_train_np` in Form eines np-Arrays wieder ein Pandas Dataframe zu machen, kann folgende Befehlszeile verwendet werden:

  ```
  X_train_pandas = pd.DataFrame(X_train_np, columns=X_train_1hot.columns)
  ```

  Hierbei stellt in diesem Beispiel `X_train_1hot` den als Pandas Dataframe vorliegenden und One-Hot codierten Traingsdatensatz vor der Skalierung dar. Der Parameter `columns` dient nun dazu, die Keys des One-Hot codierten Pandas Dataframes wieder in das neue, nun auch skalierte Pandas Dataframe zu übernehmen.

### Erstes (lineares) Regressionsmodell trainieren und testen

- Als ein erstes Regressionsmodell soll der in Scikit-Learn verfügbare lineare Regressionsalgorithmus `LinearRegression` verwendet werden. Importieren Sie das Modell aus Scikit-Learn, instanziieren Sie es unter dem Namen `lin_reg` und trainieren Sie das Modell mit den One-Hot codierten und skalierten Pandas-Trainingsdaten.
- **Tipp**: Zur Überprüfung der Vorhersagequalität des Regressionsmodells macht die Ermittlung einer Erfolgsquote wie bei Klassifikationsaufgaben keinen Sinn. Um einen ersten Überblick über die Performance des trainierten Modells zu erlangen, können bspw. durch folgende Codezeilen die ersten paar Ergebnisse der Regressionsanalyse mit den korrekten Daten der Zielgröße verglichen werden:

  ```
  print(y_train[0:4].values)
  print(lin_reg.predict(X_train_1hot_scaled[0:4]))
  ```

  **Anmerkung**: Hier sei angenommen, dass das Pandas Dataframe mit den One-Hot codierten und skalierten Merkmals-Trainingsdaten in der Variablen `X_train_1hot_scaled` vorliegt.
- Als Erfolgsmetrik bei der Regressionsanalyse wird häufig der *Root Mean Squared Error* (RMSE), also die Wurzel des mittleren quadratischen Fehlers verwendet. Die Ermittlung des RMSE mittels Scikit-Learn ist in den folgenden Codezeilen dargestellt.

  ```
  from sklearn.metrics import mean_squared_error
  import numpy as np
  ```

```
y_train_predict = lin_reg.predict(X_train_1hot_scaled)
mse = mean_squared_error(y_train, y_train_predict)
rmse = np.sqrt(mse)
print(rmse)
```

Nach dem Import der entsprechenden Bibliotheken wird das trainierte Modell (`lin_reg`) verwendet, um basierend auf den Trainingsdaten Prognosen zu erstellen (`predict()`). Anschließend wird mittels `mean_squared_error()` der mittlere quadratische Fehler zwischen den prognostizierten und den tatsächlichen Werten berechnet. In der vorletzten Codezeile wird die Wurzel des mittleren quadratischen Fehlers ermittelt.

- Lassen Sie den RMSE auf der Konsole ausgeben und interpretieren Sie das Ergebnis. Sind Sie zufrieden mit der Vorhersagequalität des Regressionsmodells?

**Weiteres Regressionsmodell (Decision Tree)**

- Nun soll ein weiteres Regressionsmodell in Form eines `DecisionTreeRegressor`s trainiert werden.
- Nennen Sie das Modell `tree_reg` und gehen Sie bzgl. Training und Verifikation zunächst so vor, wie zuvor für das lineare Regressionsmodell. Auf eine Feinabstimmung der Regularisierungsparamter kann an dieser Stelle verzichtet werden. Verwenden Sie stattdessen einfach die Default-Einstellungen des DecisionTreeRegressors. Was fällt Ihnen bzgl. des RMSE basierend auf den Trainingsdaten auf und was lässt sich hierdurch über das Regressionsmodell aussagen?
- Um die Qualität des Decision Tree Regressionsmodells besser einschätzen zu können, soll nun eine zehnfache Kreuzvalidierung durchgeführt werden. Hierbei wird der Trainingsdatenbestand in zehn gleich große Teile (sog. *Folds*) zerlegt und anschließend zehn mal mit unterschiedlichen neun Teilen des Datensatzes trainiert und mit dem jeweils zehnten Teil getestet. Dies hat den Vorteil, dass das Modell hier bereits im Rahmen der Trainingsphase mit bis dato unbekannten Daten getestet wird, wodurch sich bereits jetzt bessere Aussagen über die finale Zuverlässigkeit der Regressionsanalyse (in diesem Fall also über den finalen RMSE) treffen lassen. Die Kreuzvalidierung kann mittels der Scikit-Learn Funktion `cross_val_score` durchgeführt werden:

```
negative_mse = cross_val_score(tree_reg, X_train_1hot_scaled, y_train,
            scoring="neg_mean_squared_error", cv=10)
```

- Wie der Parameter `scoring="neg_mean_squared_error"` schon vermuten lässt, wird im Rahmen der Kreuzvalidierung der negative mittlere quadratische Fehler berechnet. Um wieder auf den RMSE als Erfolgsmetrik zu kommen, kann folgende Codezeile verwendet werden: `tree_rmse_scores = np.sqrt(-negative_mse)`
- Lassen Sie die Ergebnisse auf der Konsole ausgeben und interpretieren Sie die Ergebnisse. Wie erklären Sie sich den Unterschied dieser Ergebnisse zum Ergebnis vor der Kreuzvalidierung?

- Da das Ergebnis der zehnfachen Kreuzvalidierung aus zehn RMSE-Werten besteht, bietet es sich an, zusätzlich den Mittelwert (`tree_rmse_scores.mean()`) und die Standardabweichung (`tree_rmse_scores.std()`) der Ergebnisse zu betrachten.

**Weiteres Regressionsmodell (Random Forest mit Gittersuche)**

- Nun soll ein weiteres Regressionsmodell in Form eines `RandomForestRegressor` trainiert werden, wobei geeignete Werte für die Regularisierungsparameter mittels einer Gittersuche mit Kreuzvalidierung gefunden werden sollen.
- Hierzu sind zunächst die entsprechenden Bibliotheken zu importieren:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
```

- Anschließend ist ein Parameter-Grid zu erstellen. In diesem Beispiel wird sich auf die zwei wichtigsten Parameter `n_estimators` und `max_features` konzentriert:

```
param_grid = {'n_estimators': [3,10,30], 'max_features': [2,4,6,8,10,12]}
```

- Nun kann das Modell instanziiert werden:

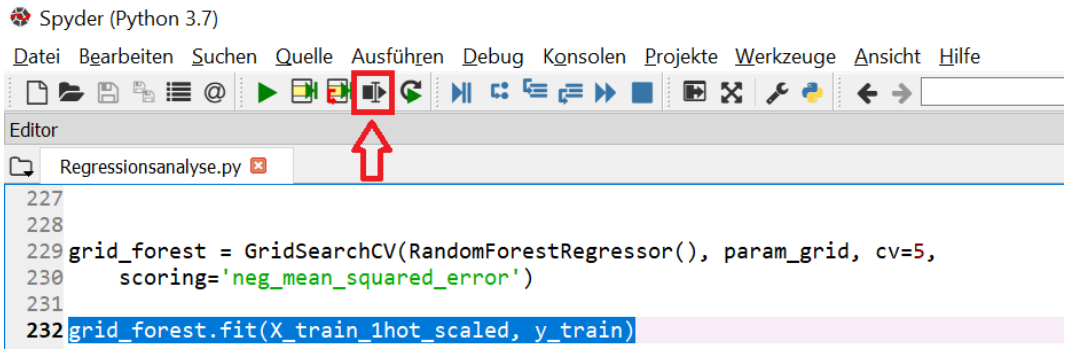
```
grid_forest = GridSearchCV(RandomForestRegressor(), param_grid, cv=5,
            scoring='neg_mean_squared_error')
```

  **Anmerkung**: Mit dem Parameter `scoring` wird der Gittersuche mitgeteilt, dass der negative mittlere quadratische Fehler als Optimierungskriterium dienen soll.

- Nun ist das Modell zu trainieren:

```
grid_forest.fit(X_train_1hot_scaled, y_train)
```

- **Tipp**: Aufgrund der doch recht beachtlichen Größe des Datensatzes nimmt das Training aufwendiger Systeme (insbesondere im Rahmen einer Gittersuche mit Kreuzvalidierung) einiges an Rechenzeit in Anspruch. Statt also das gesamte Python-Skript jedes mal komplett auszuführen, bietet es sich an, jeweils nur die zuletzt einprogrammierten Codezeilen ausführen zu lassen. Dies kann in Spyder dadurch bewerkstelligt werden, dass die auszuführenden Codezeilen im Editorfenster markiert werden und anschließend die Taste F9 (oder der in der nachfolgenden Abbildung rot markierte Button) gedrückt wird.



- **Anmerkung**: Wenn GridSearchCV mit `refit=True` parametrisiert wird (was defaultmäßig der Fall ist) wird im Anschluss an die Gittersuche mit Kreuzvalidierung (also dann, wenn die optimalen Werte für die Regularisierungsparameter vorliegen) das Modell automatisch erneut mit dem gesamten Trainingsdatensatz trainiert (im Gegensatz zum Training im Rahmen der Gittesuche, wo aufgrund der impliziten Kreuzvalidierung jeweils nur ein Teil des Trainingdatenbestandes für das Training verwendet wird).

- Die gefundenen optimalen Parameter können mit folgender Codezeile ausgegeben werden: `print(grid_forest.best_params_)`

- Der Befehl `print(grid_forest.best_estimator_)` liefert weitere Informationen zum trainierten Modell.

- Mittels `print(np.sqrt(-grid_forest.best_score_))` kann der RMSE des Modells mit den besten gefundenen Parametern ausgegeben werden. Interpretieren Sie die Ergebnisse.
  **Anmerkung**: Da im Rahmen der Gittersuche mit `GridSearchCV()` als Score der `neg_mean_squared_error` definiert wurde, steht hier in `best_score_` auch der beste (d.h. betragsmäßig kleinste) negative mittlere quadratische Fehler der gesamten Gittersuche.

- **Wichtig zu wissen**: Der ausgegebene „Score“ entspricht dem Test-Score (und nicht dem Train-Score) innerhalb der Kreuzvalidierung (wobei die beste gefundene Parameterkombination zugrunde gelegt wird). Das Ergebnis ist daher durchaus realistisch für die finale Performance des Modells. Deutlich unrealistischere Ergebnisse erhält man hingegen, wenn man das nun (via Gittersuche) trainierte Modell auf den kompletten Trainingsdatensatz anwendet.
  Wie bereits oben erwähnt, wurde das `grid_forest`-Modell implizit am Ende der Gittersuche mit dem besten gefundenen Parametersatz erneut mit dem kompletten Trainingsdatensatz trainiert. Eine Vorhersage der Zielgröße basierend auf den Trainingsdaten führt daher zu unrealistisch guten Scores, da das Modell diesen Datensatz schon komplett kennt. Überprüfen Sie dies, indem Sie das nun trainierte `grid_forest`-Modell auf den Trainingsdatensatz anwenden und daraus den RMSE berechnen.
  **Anmerkung**: Allgemein lässt sich sagen, dass die Differenz zwischen Score aus Test- und Trainingsdaten bei der Regressionsanalyse kein wirklich gutes Maß für die Optimierung bzw. Generalisierung eines Systems darstellt, da aufgrund der kontinuierlichen Zielgröße die Ergebnisse auf den Trainingsdaten in den allermeinsten Fällen zwangsläufig deutlich besser ausfallen, als auf den Testdaten.

- Um zu eruieren, welche Hyperparameterwerte zu welchem RSME führen, können mittels
  `mean_scores = grid_forest.cv_results_["mean_test_score"]` zunächst die durchschnittlichen RMSE-Werte, die sich pro Parameterkombination aus den einzelnen Durchläufen der Kreuzvalidierung ergeben, ermittelt werden. Die dazugehörigen Parameter

lassen sich mit `params = grid_forest.cv_results_["params"]` ermitteln. Anschließend können die Einzelergebnisse im Rahmen einer for-Schleife ausgegeben werden:

```
for i in np.arange(len(params)):
    print(np.sqrt(-mean_scores[i]), params[i])
```

- Der folgende Code ermöglicht die Ausgabe der „Wichtigkeit“ der einzelnen Merkmale, wobei ein großer Wert auf ein gutes Merkmal hindeutet:

```
feature_importances = grid_forest.best_estimator_.feature_importances_
feature_names = X_train_1hot_scaled.keys()
for i in np.arange(len(feature_importances)):
    print(feature_importances[i], feature_names[i])
```

**Anwenden des Modells auf die Testdaten**

- Wenden Sie das `grid_forest`-Modell auf den Testdatensatz an.
  **Tipp**: Bedenken Sie, dass `grid_forest` nach wie vor sämtliche Regressionsmodelle aller Parameterkombinationen der Gittersuche beinhaltet. Statt nun also jedesmal auf jenes Modell mit den bestem Parametersatz zuzugreifen (im Sinne von `grid_forest.best_estimator_.predict(...)`), bietet es sich an, das beste Regressiondmodell in einer neuen Instanz abzulegen:
  `final_model = grid_forest.best_estimator_`
  **Hinweis**: Achten Sie darauf, dass Sie die Testdaten noch für die Prädiktion vorbereiten müssen (One-Hot Codierung und Skalierung).

- Welchen RMSE liefert Ihr Modell? Sind Sie mit diesem Ergebnis zufrieden?

- Häufig ist bezüglich der Beurteilung der Zuverlässigkeit eines Regressionsmodells nicht nur der finale Score (bzw. RMSE) auf den Testdaten von Interesse. Vielmehr ist es zudem wichtig abzuschätzen, in welchem Werteintervall der im Live-Betrieb tatsächlich zu erwartende RMSE liegen wird. Hierfür kann ein *Konfidenzintervall* berechnet werden, in dem der tatsächliche Score für neue Daten des Regressionsmodells mit einer vorgegebenen Wahrscheinlichkeit zu erwarten ist. Der nachfolgende Code ermöglicht die Berechnung und Ausgabe des 95 %-Konfidenzintervalls (wobei davon ausgegangen wird, dass die Vorhersagen des finalen Regressionsmodells auf den Testdaten in der Variablen `final_predictions` vorliegen). Es kann also davon ausgegangen werden, dass der Score (bzw. RMSE) des finalen Modells mit 95 %-iger Wahrscheinlichkeit in diesem Wertebereich liegt, wenn das Modell auf bis dato ungekannte Daten angewendet wird.

```
from scipy import stats
confidence = 0.95
squared_errors = (final_predictions - y_test)**2
conf_intervall_95 = np.sqrt(stats.t.interval(confidence,
    len(squared_errors) - 1, loc=squared_errors.mean(),
```

```
        scale=stats.sem(squared_errors)))
    print(conf_intervall_95)
```