Motivation

- relation between variables
  - do variables in [[Multiple Regression]] affect each other?
- This example raises the question about the degree of relation
- We could formulate the question as a hypothesis test:

$$H_0 : \beta_j = 0 \text{ vs. } H_1 : \beta_j \neq 0$$

- This question has some important scientific and practical consequences:
  1. We learn something about relationships between variables
  2. Should we eliminate some variables for efficiency and a better prediction
  3. Correlation vs. causation

Underfitting vs. Overfitting

- The previous example illustrates a typical problem in multiple regression
- We have many features but do we want to include them all in the model?
- A smaller model with less features has two advantages:
  1. The prediction may be better
  2. Better understanding of the problem
- Generally, more features leads to less bias but a higher variance
- Too few features: **underfitting** results in high bias
- Too many features: **overfitting** results in high variance
- Good predictions result from achieving a good balance between bias and variance

- overfitting causes worse prediction accuracy on unseen data

Model Selection

- We apply **model selection** to achieve a good balance between bias and variance
- Model selection consists of two steps:
  1. Assign a *score* to each model, which measures how good model is
  2. Search trough (all) models to find the model with the best score
- It is a bad idea to score the model on the **training** data
- We will always underestimate the prediction error because we are using the data twice: to fit the model and to estimate the error
- A better estimation of the prediction error is on the **test** data
- Popular choices for scoring are SSE or R^2:

$$R^2 = 1 - \frac{\sum_i (Y_i - \hat{Y}_i)^2}{\sum_i (Y_i - \overline{Y}_n)^2}$$

-
- cross validation
  - Cross-validation is a computational method for (among others) estimating prediction error
  - A typical approach is **k-fold cross-validation**:
    1. We divide the data into k groups, e.g. $k = 10$
    2. We omit one group and fit the model on the remaining groups
    3. We use the fitted model to predict the data from the omitted group
    4. We estimate the error by e.g. SSE on the omitted group
    5. We repeat all the steps for all k groups and average the individual SSEs

Model Search

- Now that we know how to score the models we need a search strategy
- If there are k features how many possible models do we have?
- 2^k
- If k is not too large we can perform an exhaustive search
- Otherwise we need a heuristic and search only over a subset of all models
-
- greedy model search

- Two common methods are **forward and backward stepwise regression**
- In forward regression we start with no feature in the model
- We then greedily add the feature that gives the best score
- We continue adding one feature at a time until the score does not improve
- In backward regression we start with the full model and remove one feature at a time until we can not improve the score any more
- Both methods can not guarantee to reach the model with the globally best score
- In practice, they work quite well

- example
  - We have a video games dataset containing games release dates, price, sales, average and median playtime from Steam, critics rating, and the user rating from from Metacritic reviewing site. We are interested in relationship between user rating and other features. We fit the model and obtain the following results:
  - model search
    *
```
def get_X(df, features):
    X = df.loc[:, features]
    return preprocessing.scale(X)


def model_selection(df, features, y):
    reg = LinearRegression()
    scores = cross_val_score(reg, get_X(df, features), y, cv=5)
    full_score = np.mean(scores)

    feature_scores = {}
    for feature in features:
        remaining_features = [f for f in features if f != feature]
        reg = LinearRegression()
        scores = cross_val_score(reg, get_X(df, remaining_features), y, cv=5)
        feature_scores[feature] = np.mean(scores)

    weakest_feature = max(feature_scores, key=feature_scores.get)
    return full_score, feature_scores[weakest_feature], [f for f in features if f != weakest_feature]
```
    *
```
df = pd.read_csv('../data/games.csv')
all_features = ['release_date', 'price', 'owners', 'average_playtime', 'median_playtime', 'metascore']
y = df.userscore

score = 0
features = list(all_features)
while True:
    full_score, score, new_features = model_selection(df, features, y)
    print('{}\nscore={}'.format(features, full_score))
    print('{}\nnew score={}'.format(new_features, score))
    print('********')
    if score >= full_score:
        features = new_features
    else:
        break

print('*************** Summary ****************')
X_selected = get_X(df, features)
reg = LinearRegression().fit(X_selected, y)
print('Best model:\n{}\ncv-score={}, training-score={}'.format(features, score, reg.score(X_selected, y)))

X = get_X(df, all_features)
reg = LinearRegression().fit(X, y)
scores = cross_val_score(LinearRegression(), X, y, cv=5)
print('Full model:\n{}\ncv-score={}, training-score={}'.format(all_features, np.mean(scores), reg.score(X, y)))
```
  - [[Konfidenzintervall]] with [[Bootstrap Principle]]
    * We bootstrap the linear regression by resampling the features and the target value. For each bootstrap sample we fit a linear regression model and record the feature coefficients. We obtain in this way bootstrap distributions of linear regression coefficients and we estimate the confidence intervals for the coefficients. All coeffienents with confidence intervals not including zero are siginificant.

```
df = pd.read_csv('../data/games.csv')
features = ['release_date', 'price', 'owners', 'average_playtime', 'median_playtime', 'metascore']
y = df.userscore

X = get_X(df, features)
reg = LinearRegression().fit(X, y)

print(features)
print('Regression coefficients:', reg.coef_)
print('Intercept', reg.intercept_)

b = 1000
n = len(y)
indices = range(len(y))
indices_star = np.random.choice(indices, (b, n))
delta_coef_stars = []
delta_intercept_stars = []
for index_star in indices_star:
    reg_star = LinearRegression().fit(X[index_star], y[index_star])
    delta_coef_star = reg_star.coef_ - reg.coef_
    delta_coef_stars.append(delta_coef_star)
    delta_intercept_star = reg_star.intercept_ - reg.intercept_
    delta_intercept_stars.append(delta_intercept_star)

alpha = 0.05
delta_coef_stars = np.array(delta_coef_stars)
for i in range(len(features)):
    print('{}% bootstrap confidence interval for {}: ({:8.6f}, {:8.6f})'.
          format((1 - alpha) * 100,
                 features[i],
                 reg.coef_[i] - np.quantile(delta_coef_stars[:, i], 1 - alpha/2),
                 reg.coef_[i] - np.quantile(delta_coef_stars[:, i], alpha/2)))

print('{}% bootstrap confidence interval for intercept: ({:8.6f}, {:8.6f})'.
```

Regularization (Ridge Regression)

- example

  Using measures of cognitive-motor, attentional, and perceptual processing extracted from game data from 3360 Real-Time Strategy players (StarCraft 2 players) at different levels of expertise, identify variables most relevant to expertise.

- show bias-variance trade-off

Evaluating Mean Squared Error, Bias and Variance

We randomly split the dataset to train/test, fit the models on the train and predict on the test dataset. We measure the mean squared error, bias and variance of the prediction. To obtain more robust results we create $1,000$ random splits.

```
def evaluate(df, features, target, reg):
    X = get_data(df, features)
    y = get_data(df, target)

    b = 1000
    mse = []
    bias = []
    var = []
    for i in range(b):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
        reg.fit(X_train, y_train)
        y_hat = reg.predict(X_test)
        mse.append(mean_squared_error(y_test, y_hat))
        bias.append(np.mean(y_hat - y_test) ** 2)
        var.append(np.var(y_hat))

    return np.mean(mse), np.mean(bias), np.mean(var)
```

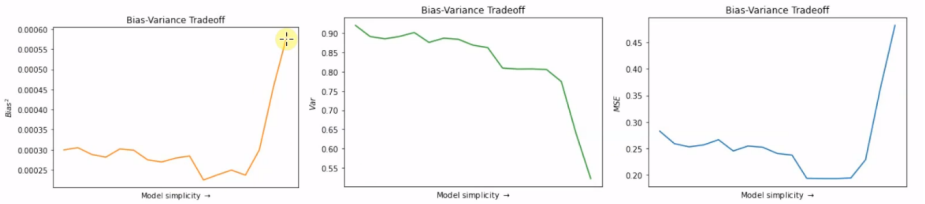
Evaluating Ordinary Least Squares

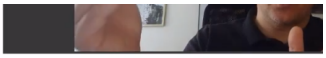
In the ordinary least squares we control for the complexity of the model by adding/removing features.

```
def ls_performance(df, all_features, target):
    ms = []
    bias = []
    var = []

    for i in range(len(all_features)):
        features = all_features[i:]
        m, b, v = evaluate(df, features, target,  LinearRegression())
        ms.append(m)
        bias.append(b)
        var.append(v)
        print(i)

    x = np.arange(len(all_features))
    plot(x, ms, bias, var)
```



- ridge regression
  - OLS is unbiased w.r.t. feature coefficients 
  - Each feature is equally important as others
  - However, in practice some features are more important than others
  - We would like to give more weight to important features
  - Also, less weight to less important features
  - We can achieve this by shrinking the less important coefficients
  - We regularize the SSE and add a coefficient shrinking term:

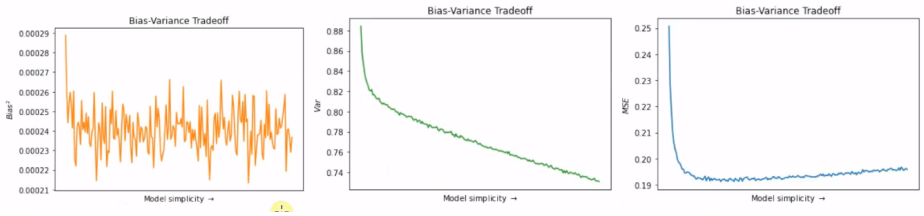
$$J(\hat{\boldsymbol{\beta}}) = ||\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}||_2^2 + \lambda||\hat{\boldsymbol{\beta}}||_2^2$$

  - The contribution of a less important feature to SSE term is low
  - Its contribution to the regularization term should be also low
  - Hence, we move its coefficient towards zero
  - The parameter λ: SSE vs. regularization
  - Higher λ more coefficients moving towards zero
- apply to example

```python
def ridge_performance(df, all_features, target):
    ms = []
    bias = []
    var = []
    alpha_max = 200

    for alpha in range(1, alpha_max):
        m, b, v = evaluate(df, all_features, target, Ridge(alpha=alpha))
        ms.append(m)
        bias.append(b)
        var.append(v)
        print(alpha)

    x = np.arange(1, alpha_max)
    plot(x, ms, bias, var)
    return np.argmin(ms)
```




```
Optimal value of regularization parameter:  62
Features:  ['HoursPerWeek', 'AssignToHotkeys', 'ComplexUnitsMade', 'UniqueHotkeys', 'MinimapAttacks', 'UniqueUnit
sMade', 'ActionsInPAC', 'Age', 'MinimapRightClicks', 'TotalHours', 'TotalMapExplored', 'WorkersMade', 'ComplexAbi
litiesUsed', 'GapBetweenPACs', 'NumberOfPACs', 'SelectByHotkeys', 'APM']
Ridge regression coefficients: [ 0.00438386 -0.01267361 -0.01090427  0.01153025  0.01062275  0.01604049
 -0.04553774  0.05199558  0.02615809  0.00384143 -0.03039133  0.01602896
 -0.02400709  0.25456661 -0.50963935  0.16389016 -0.35606091]
Ridge intercept:  8.793596146518764e-17
```