W4995 Applied Machine Learning
Fall 2021

Lecture 2

Dr. Vijay Pappu

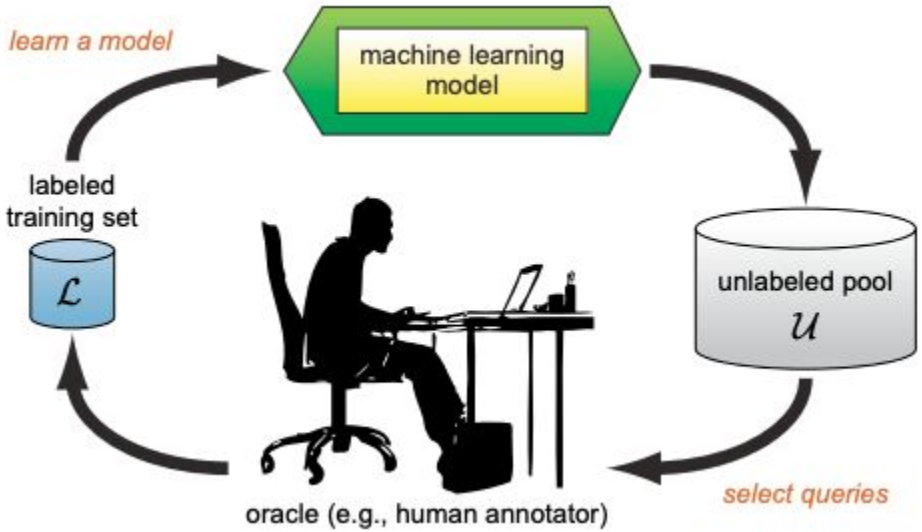
Announcements

- Class roster has been finalized - there are 164 students!
- Classroom change - 501 Northwest Corner building
- Assignment 1 has been posted:
  - https://classroom.github.com/a/84l3qGwp
  - Due date: **09/29** at midnight ET.
- Project description has been posted and students are assigned groups:
  - Deliverable 1 (Project Proposal) - **10/06**
  - Deliverable 2 (Data Analysis & Visualization) - **11/10**
  - Deliverable 3 (Presentation) - **12/15**

Follow-ups to questions from Lecture 1

Active Learning Example

- Active Learning frameworks have been applied in NLP tasks to significantly reduce the amount of labeled data needed.



http://burrsettles.com/pub/settles.activelearning.pdf

Self-Supervised Learning Example

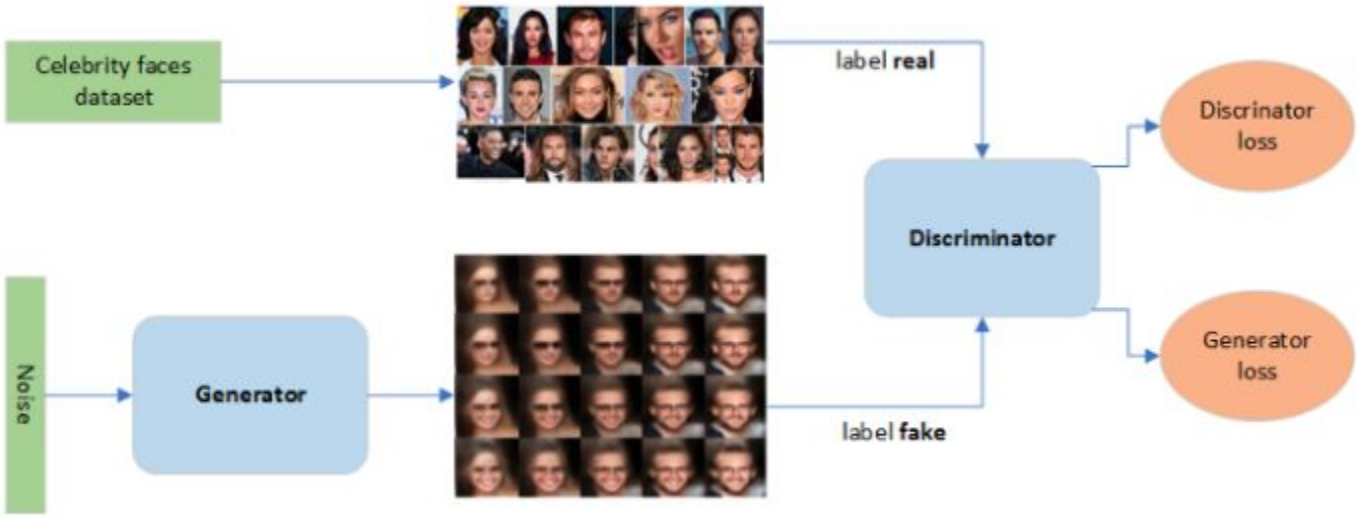
- Self-supervised learning techniques are generally used as pre-train tasks to generate labels for other downstream tasks.

Center-word prediction

Neighbor sentence prediction

Deep Learning application to Unsupervised Learning

- Generative Adversarial Networks (GANs) set up a supervised learning problem in order to do unsupervised learning (i.e. density estimation)



generative adversarial network for celebrity faces (self-created)

https://bit.ly/3CDt7Qi

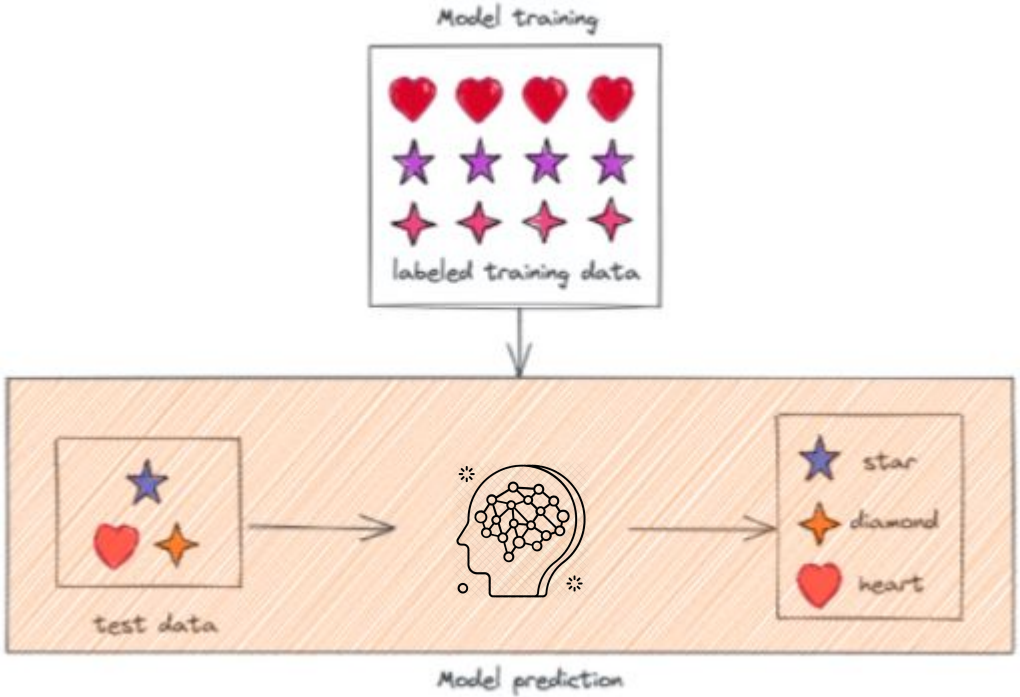
In today’s lecture, we will cover...

- Supervised learning
- Data preprocessing

Supervised Learning

Supervised Learning

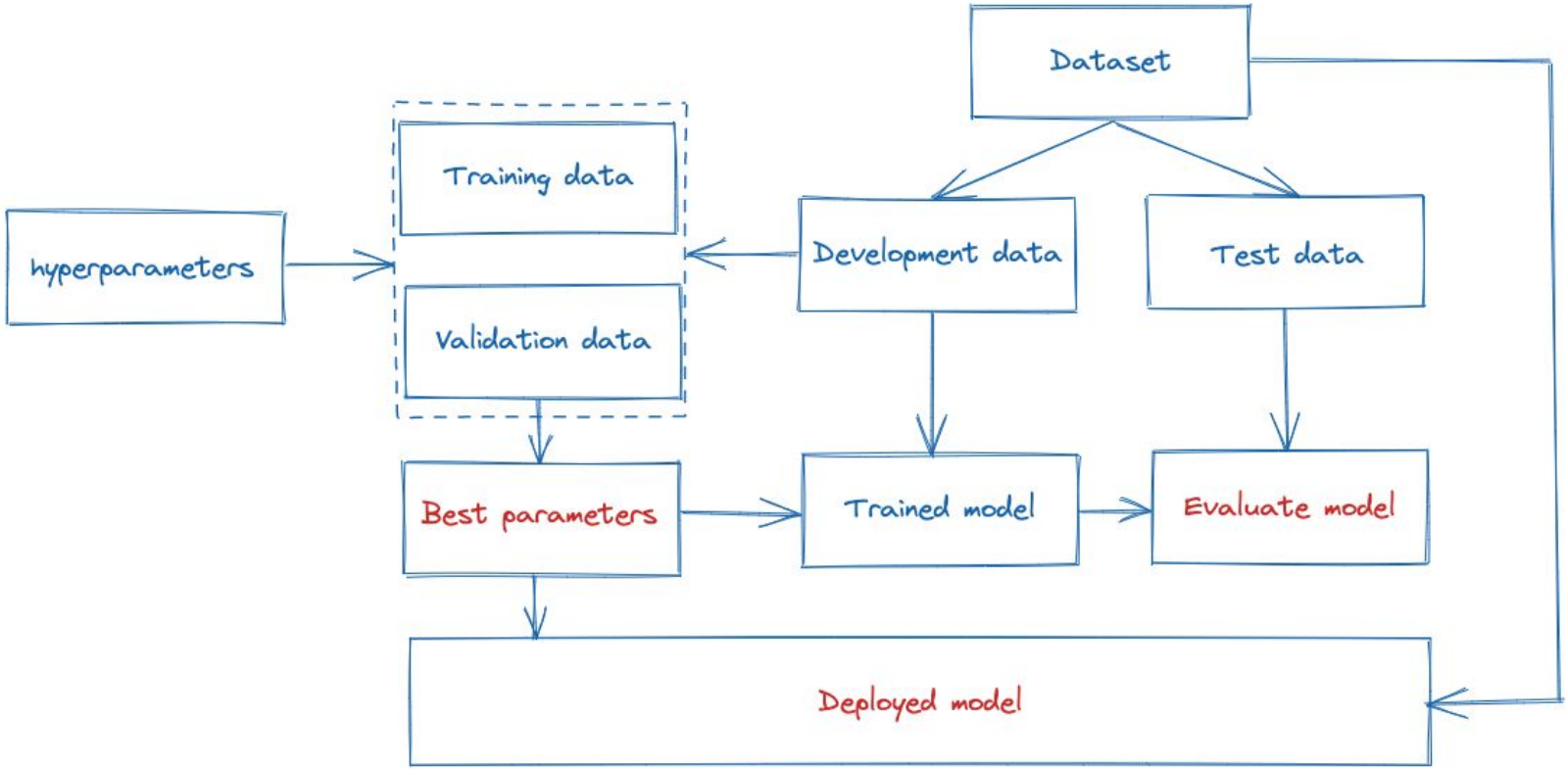
- Supervised learning algorithms learn a function that maps inputs to an output from a set of **labeled** training data.



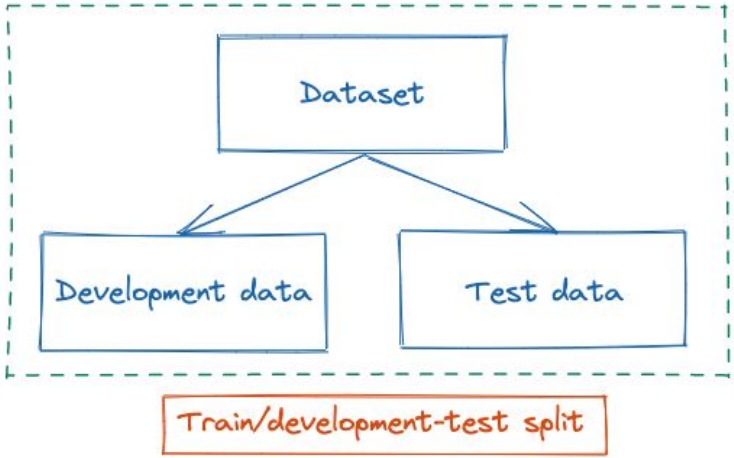
Supervised Learning goal

Deployed ML model **generalizes** well on unseen data

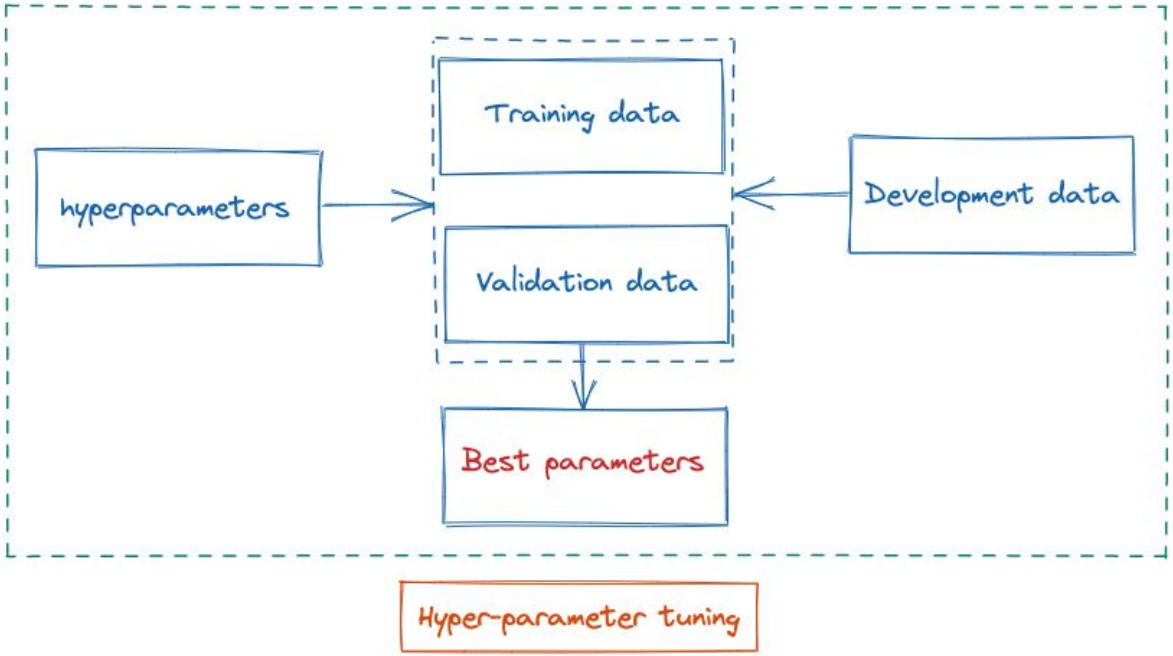
Supervised learning framework



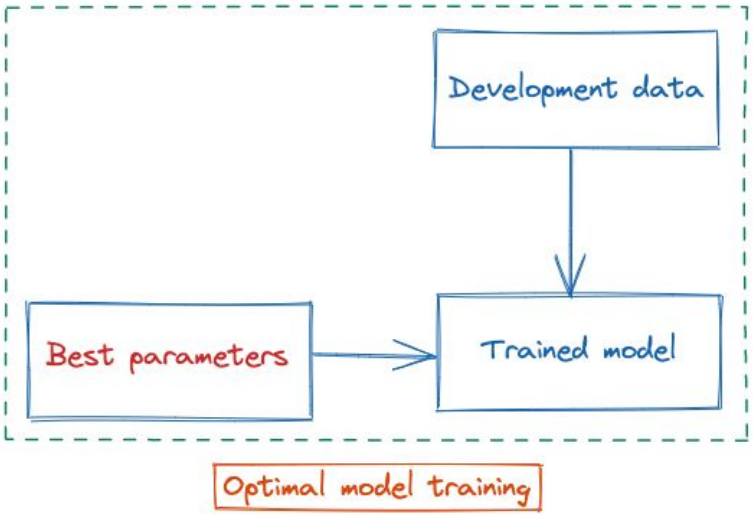
Supervised learning framework



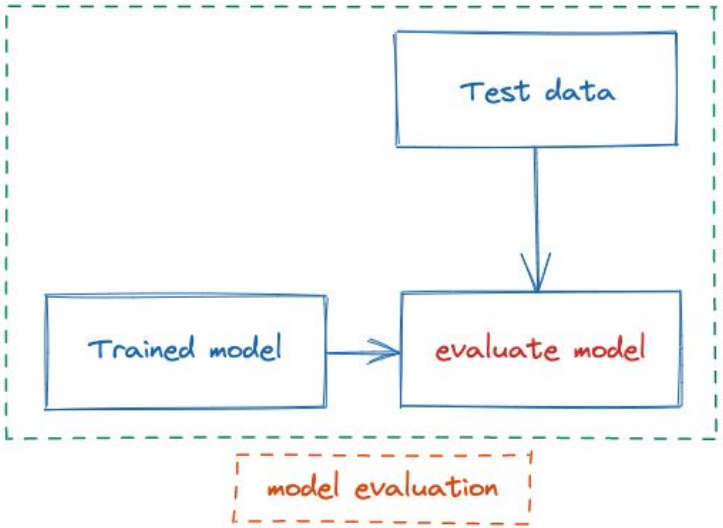
Supervised learning framework



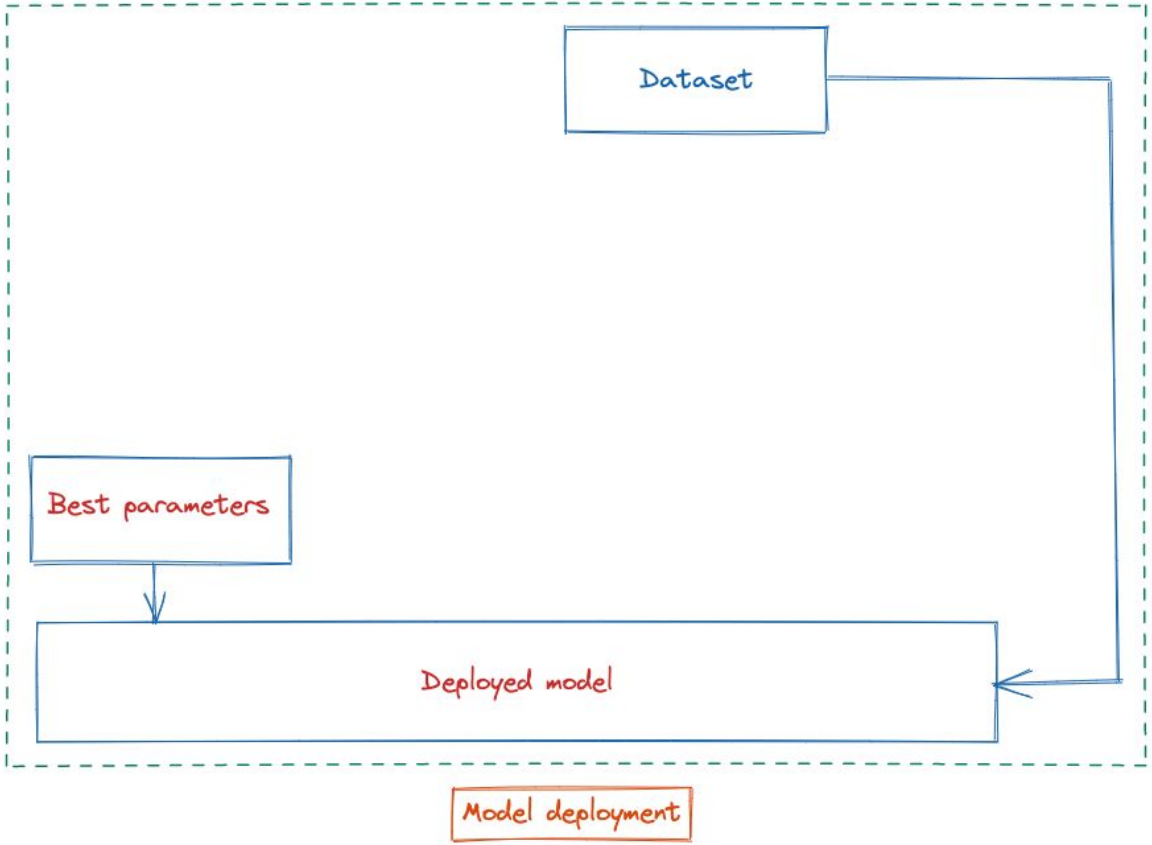
Supervised learning framework



Supervised learning framework



Supervised learning framework



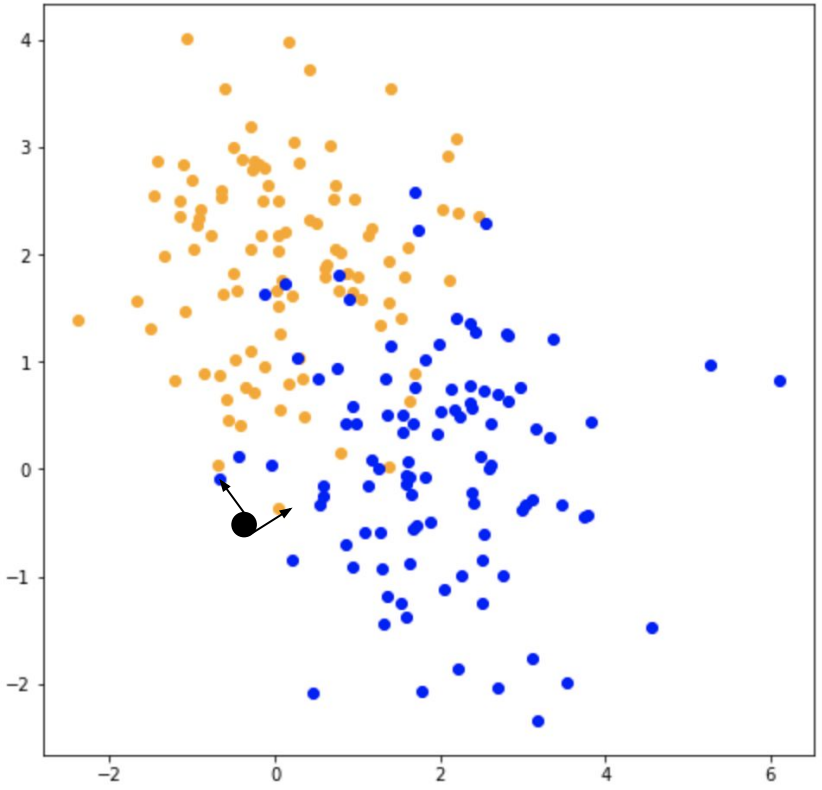
Supervised learning framework

- Development-test split
- Hyperparameter tuning
- Optimal model training
- Model evaluation
- Model deployment

k-nearest neighbors

- A simple non-parametric supervised learning method
- Assigns the value of the *nearest* neighbor(s) to the unseen data point
- Prediction is computationally expensive, while training is *trivial*
- Generally performs poorly at high dimensions

k-nearest neighbors



Example

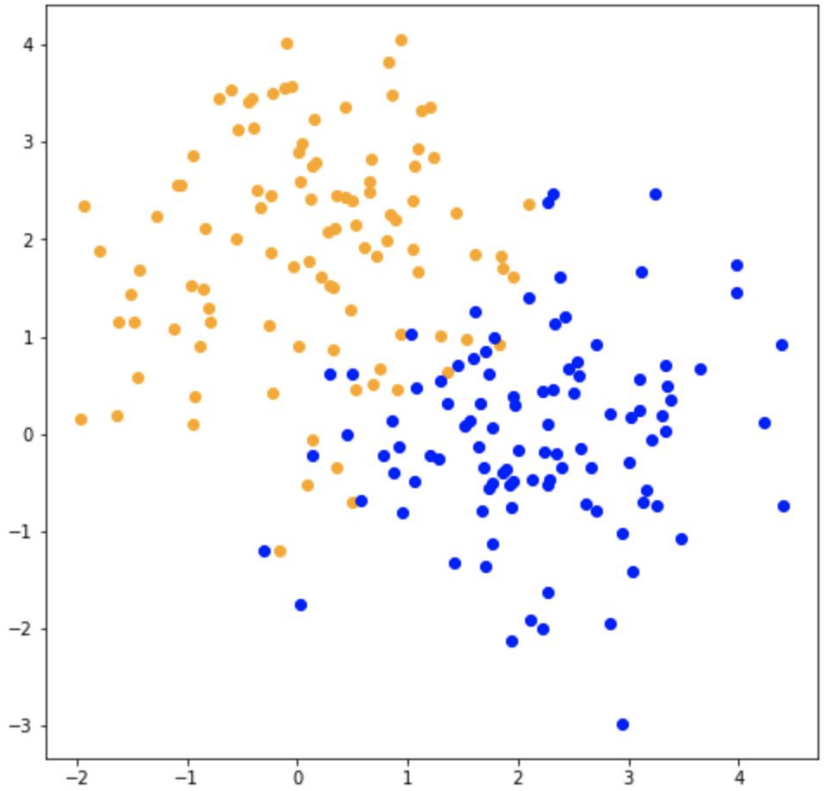
```
def generate_data(size, mean, cov, seed):
    m = multivariate_normal(mean=mean, cov=identity)
    samples = m.rvs(size=size, random_state = seed)
    return samples

def plot_data(orange_data, blue_data):
    axes.plot(orange_data[:, 0], orange_data[:, 1], 'o', color='orange')
    axes.plot(blue_data[:, 0], blue_data[:, 1], 'o', color='blue')

sample_size = 100
cov = np.identity(2)
blue_data = generate_data(sample_size, [2, 0], cov, 10)
orange_data = generate_data(sample_size, [0, 2], cov, 20)

# plotting
fig = plt.figure(figsize = (8, 8))
axes = fig.add_subplot(1, 1, 1)
plot_data(orange_data, blue_data)

plt.show()
```



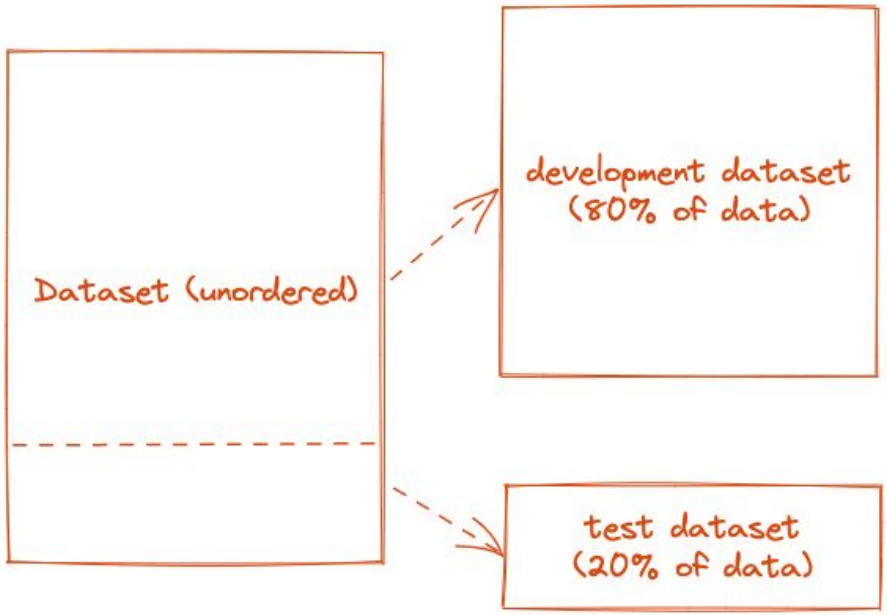
Development-test split

Development-test split

- Typically the dataset is split into development dataset and test dataset in the ratio of 4:1 (also called 80/20 split) or 3:1.
- The purpose of test dataset is to evaluate the performance of the final optimal model
- Model evaluation is supposed to give a pulse on how the model would perform in the *wild*.
- Splitting strategies:
    - Random splitting
    - Stratified splitting
    - Structured splitting

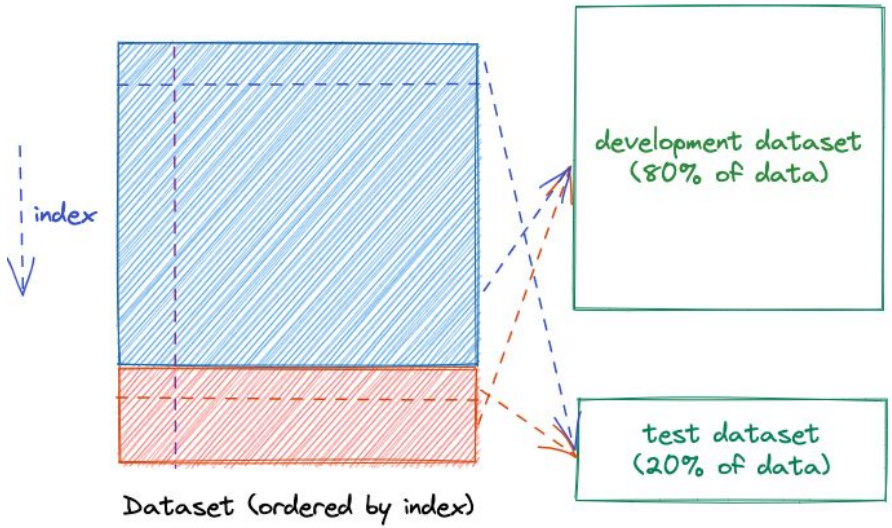
Random Splitting

- The dataset is split at random to create development & test datasets
- The size of test dataset is generally determined by the total number of samples.



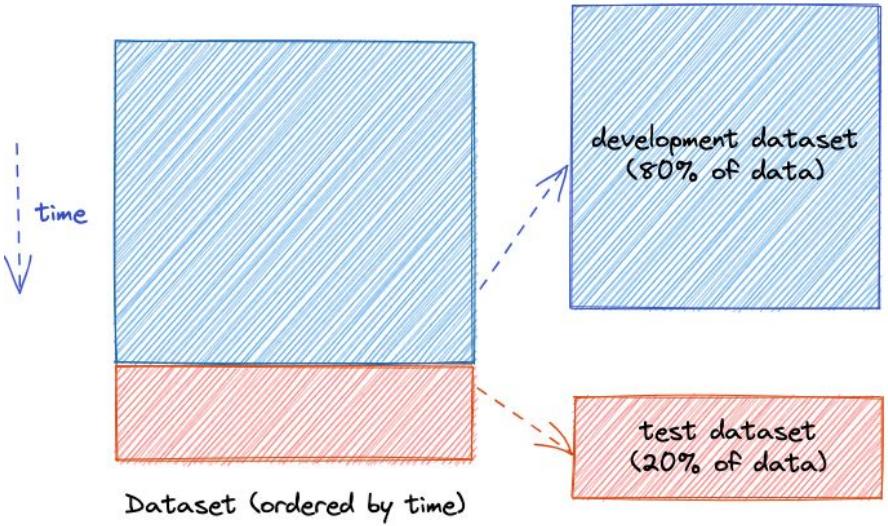
Stratified Splitting

- The stratified splitting ensures that the ratio of indices (classes) in development and test datasets equals that of the original dataset.
- Generally employed when performing classification tasks on highly imbalanced datasets.



Structured Splitting

- The structured splitting is generally employed to prevent data leakage.
- Examples:
  - Stock price predictions
  - Time-series predictions



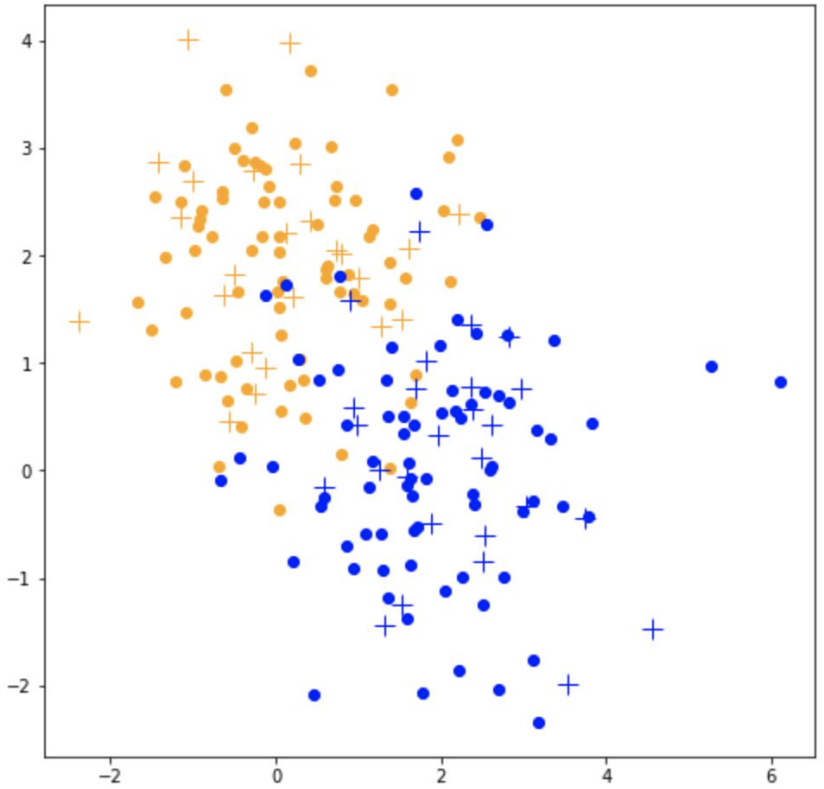
Development-test splitting

```
X = np.r_[blue_data, orange_data]
y = np.r_[np.zeros(sample_size), np.ones(sample_size)]
from sklearn.model_selection import train_test_split
X_dev, X_test, y_dev, y_test = train_test_split(X, y)

# create dev/test datasets
dev_orange_data = X_dev[y_dev == 1]
dev_blue_data = X_dev[y_dev == 0]
test_orange_data = X_test[y_test == 1]
test_blue_data = X_test[y_test == 0]

def plot_data(orange_data, blue_data, shape='o', markersize=6):
    axes.plot(orange_data[:, 0], orange_data[:, 1], shape, markersize=markersize, color='orange')
    axes.plot(blue_data[:, 0], blue_data[:, 1], shape, markersize=markersize, color='blue')

# plotting
fig = plt.figure(figsize = (8, 8))
axes = fig.add_subplot(1, 1, 1)
plot_data(dev_orange_data, dev_blue_data, shape='o')
plot_data(test_orange_data, test_blue_data, shape='+', markersize=12)
```

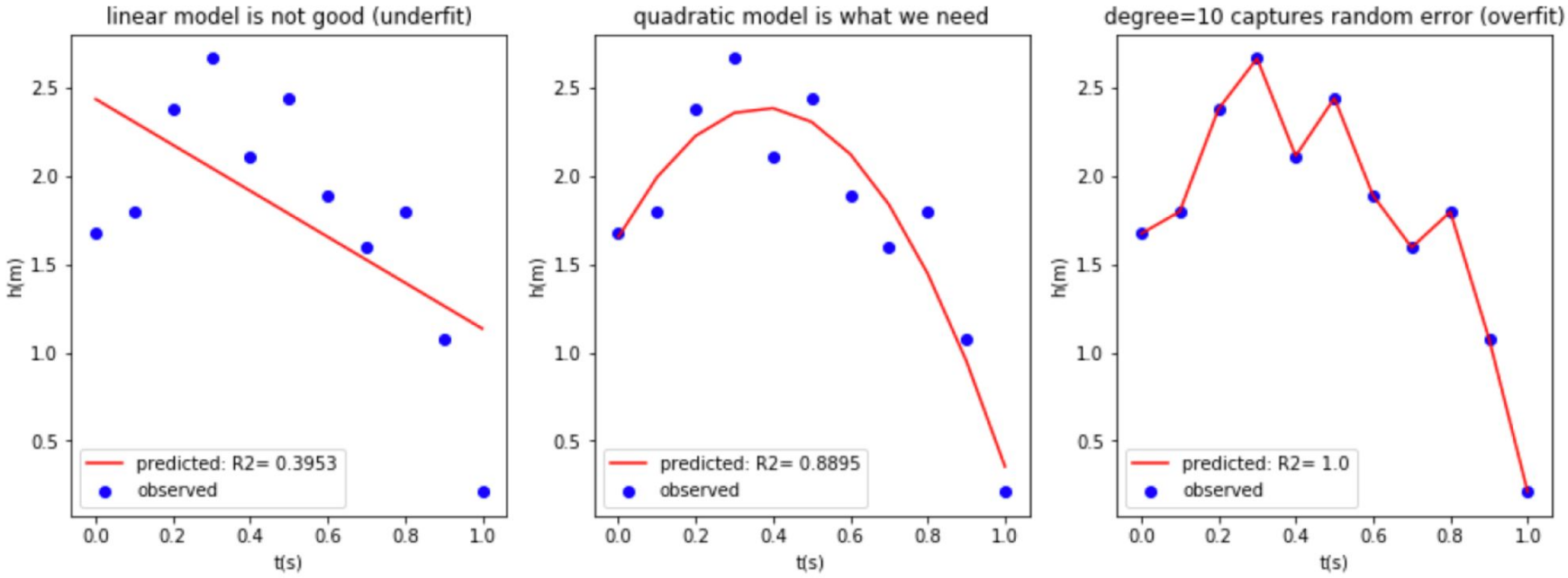


Default sklearn settings:

- Stratified splitting
- development/train-test split - 3:1

Hyperparameter tuning

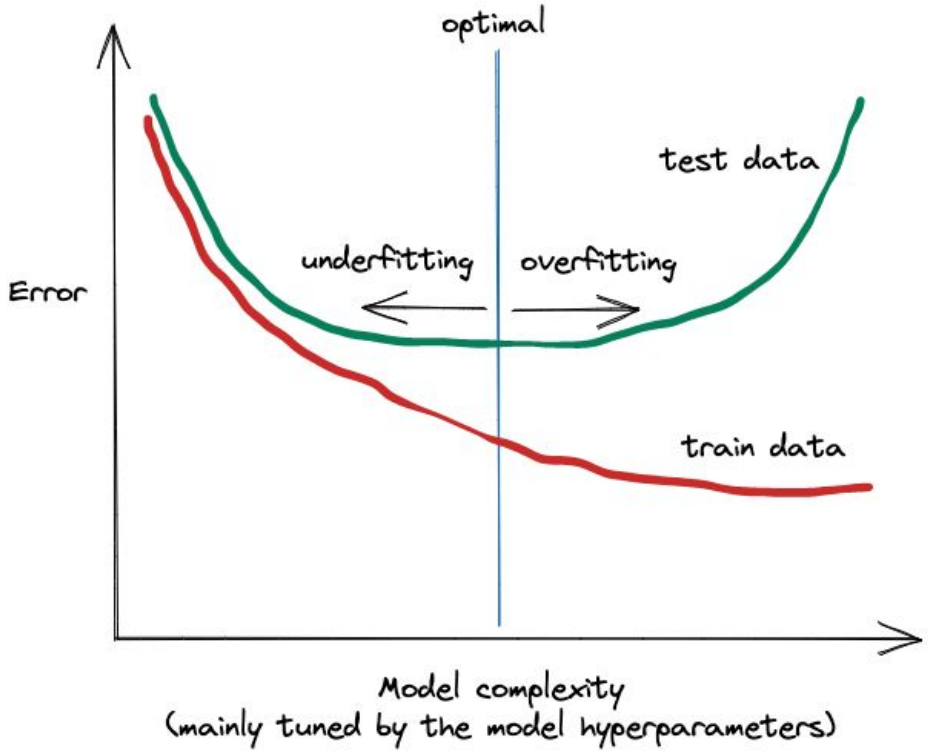
Model complexity



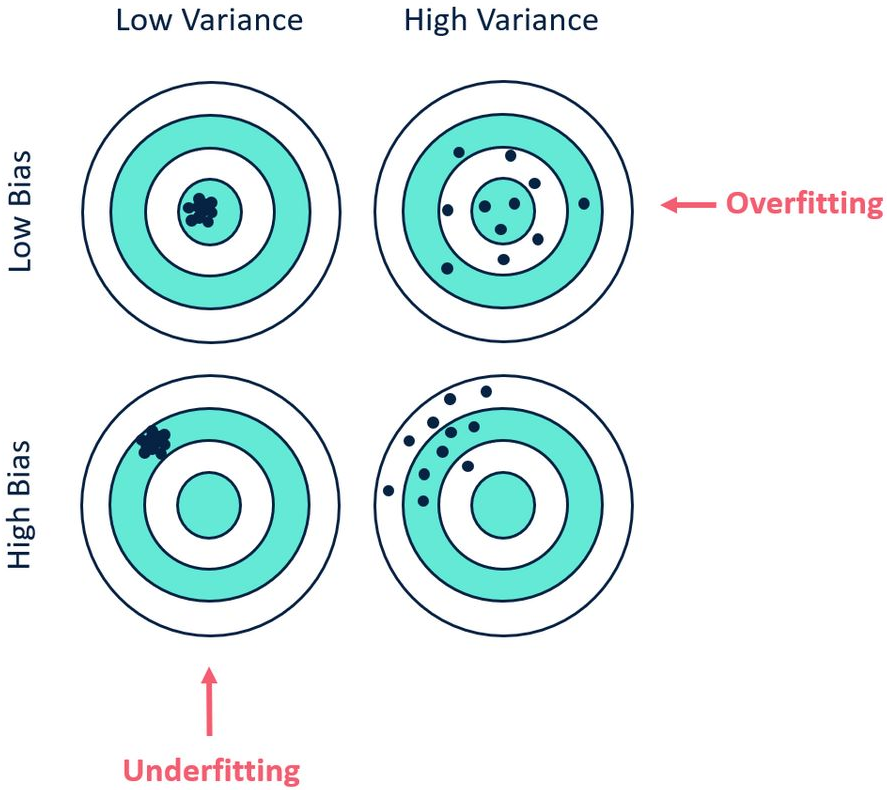
Model hyperparameters control the model complexity

[Model-complexity]

Model complexity v.s. Model performance



Bias-Variance Tradeoff



https://community.alteryx.com/t5/Data-Science/Bias-Versus-Variance/ba-p/351862

Model hyperparameters v.s. Model parameters

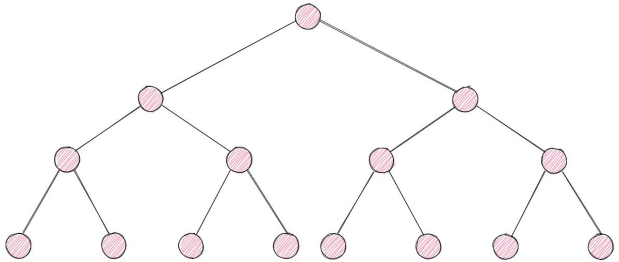
Model hyperparameter
(**tuned** during the training process)

(Lasso regression)

$$\underset{w}{Min} \; ||y - X^T w||_2^{\,2} + \lambda ||w||_1$$

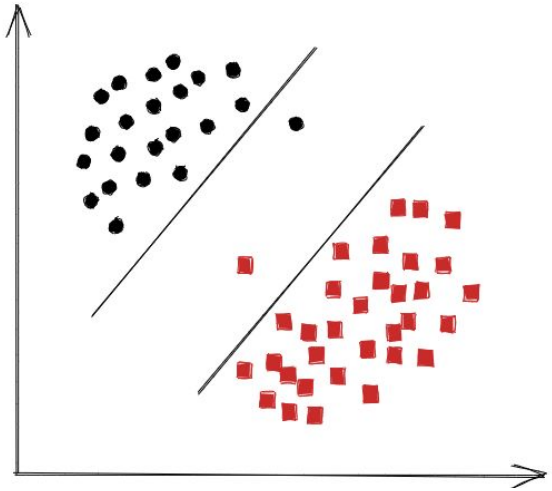
Model parameter
(**learned** during the training process)

Model hyperparameter examples



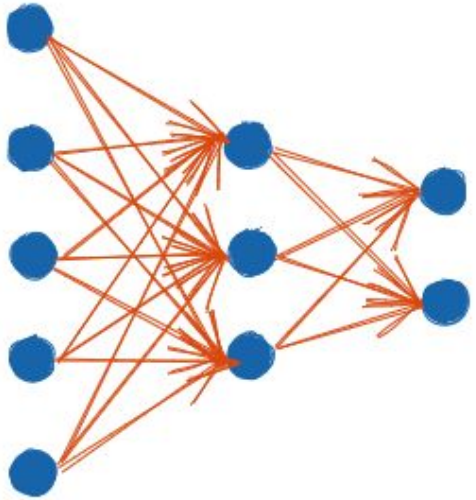
Decision trees

Depth, criterion, min samples split etc.



Support vector machines

Kernel width, penalty



Neural networks

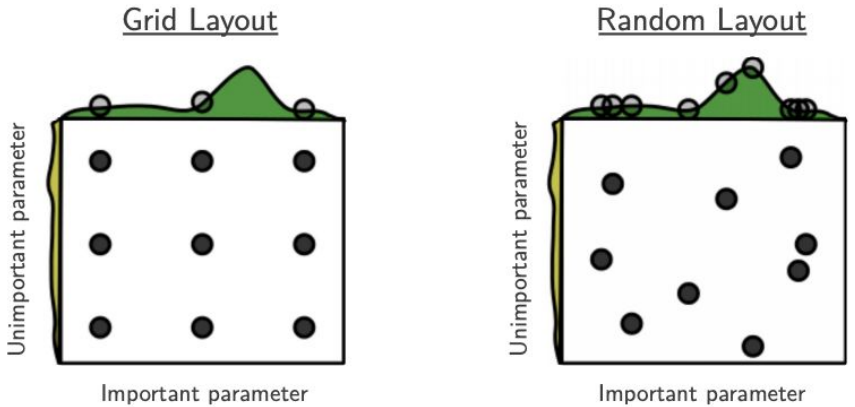
Network topology, learning rate etc.

Hyperparameter search

- Grid search
- Random search
- Bayesian optimization
- Evolutionary optimization

Grid search v.s. Random search

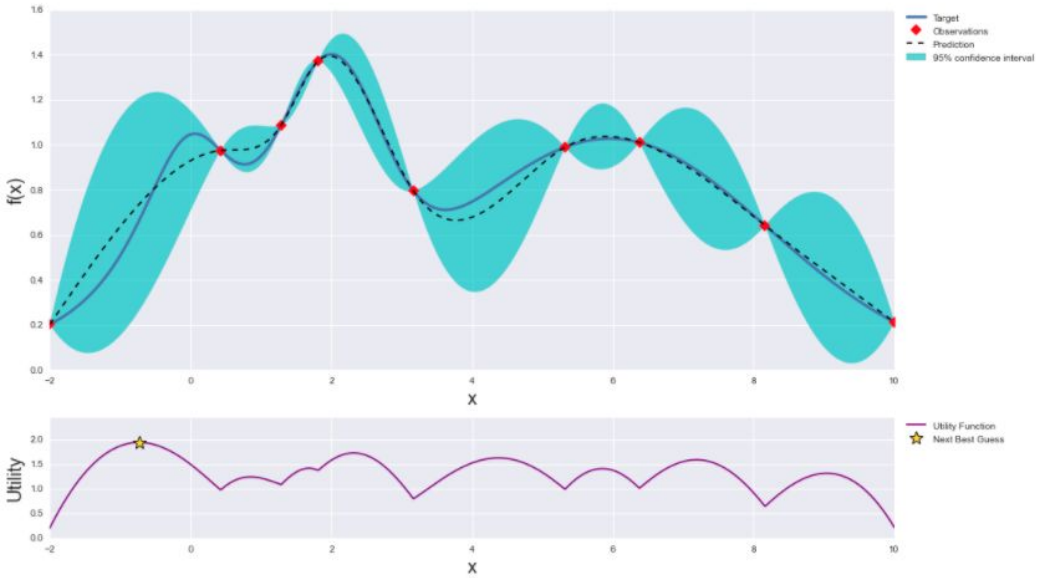
- Grid search and Random search are both considered as uninformed search strategies
- Possible to combine both strategies:
  - Search a larger space using random search
  - Find promising areas
  - Perform grid search in the smaller area
  - Continue until optimal score is obtained



Random Search for Hyper-Parameter Optimization

Bayesian optimization

- Bayesian optimization works by constructing a probability distribution of possible functions (gaussian process) that best describe the function you want to optimize.
- A utility function helps explore the parameter space by trading between exploration and exploitation.
- The probability distribution of functions is updated (bayesian) based on observations so far.



https://github.com/fmfn/BayesianOptimization

Model selection

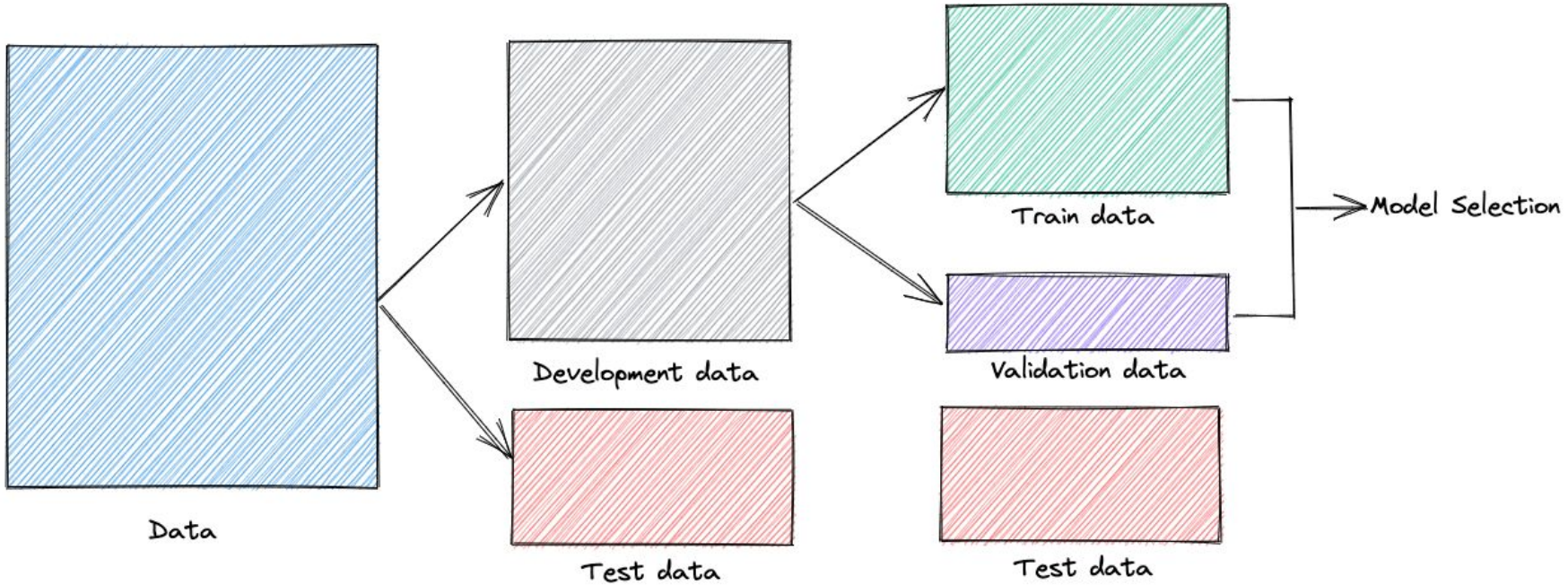
- Once a hyperparameter search strategy is chosen, we need a way to know how *good* a hyperparameter value is.
- Theoretically, we *could* use the test dataset to evaluate the performance of the model trained using the hyperparameter value.
- However, this could lead to overfitting and/or providing an overly optimistic model performance estimate.
- We will need a **validation** dataset to estimate the effectiveness of a hyperparameter value, thereby eventually helping with the model selection.

Model selection strategies

- Three-way holdout
- K-fold cross validation (CV)
  - Repeated K-fold CV
  - Leave-one-out CV
- Stratified K-fold CV
  - Repeated stratified K-fold CV
- Random permutation CV

Three-way holdout

- Model selection corresponds to the hyperparameter with the best performance on validation data.



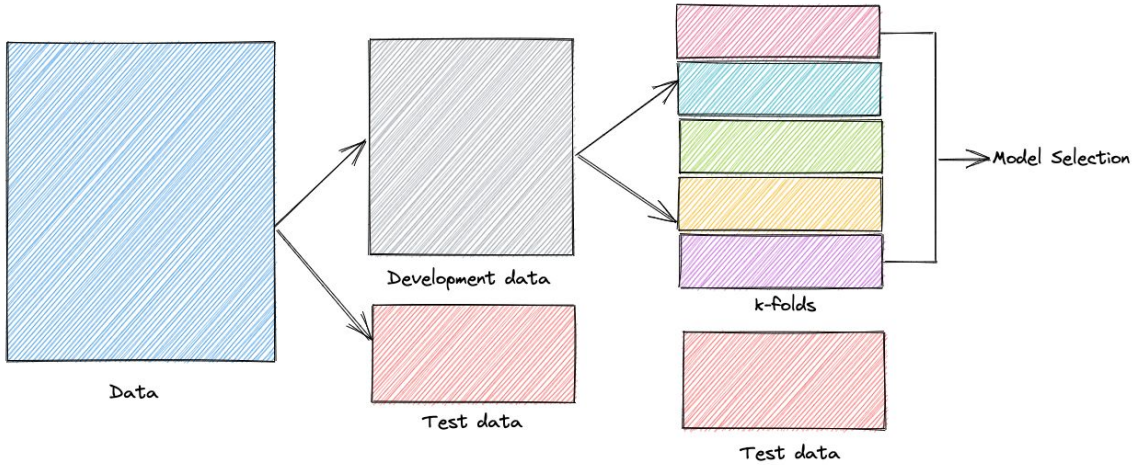
Grid search with three-way holdout

```
from sklearn.neighbors import KNeighborsClassifier
X_dev, X_test, y_dev, y_test = train_test_split(X, y, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_dev, y_dev, random_state=84)
val_scores = []
neighbors = np.arange(1, 20, 2)
for idx in neighbors:
    knn = KNeighborsClassifier(n_neighbors=idx)
    knn.fit(X_train, y_train)
    val_scores.append(knn.score(X_val, y_val))
print(f"Best validation score: {np.max(val_scores):.3f}")
best_n_neighbors = neighbors[np.argmax(val_scores)]
print("Best # of neighbors:", best_n_neighbors)
```

```
Best validation score: 0.947
Best # of neighbors: 11
```

(Repeated) K-fold CV & Leave-one-out CV

1. The development data is split into k-folds randomly.
2. For every hyperparameter, model is trained using k-1 folds and evaluated on the k^{th}-fold
3. The process is repeated for all k-folds and an average model performance is estimated.
4. Model selection corresponds to the hyperparameter with the best average performance.
5. In **repeated k-fold CV**, steps (1)-(3) are repeated for n-times and performance is estimated over k*n iterations.
6. In **leave-one-out CV**, we set k equal to the number of samples.



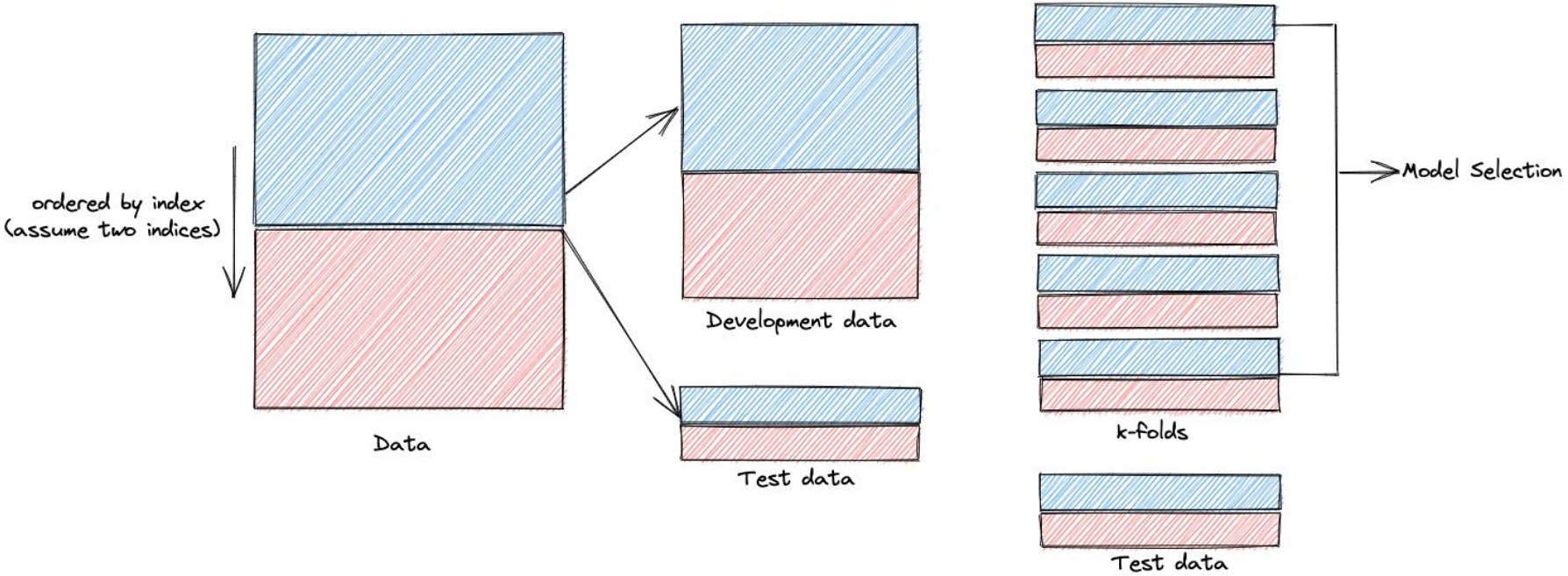
Grid search with K-fold CV

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
X_dev, X_test, y_dev, y_test = train_test_split(X, y, random_state=42)
kfold_val_scores = []
neighbors = np.arange(1, 20, 2)
for idx in neighbors:
    knn = KNeighborsClassifier(n_neighbors=idx)
    scores = cross_val_score(knn, X_dev, y_dev, cv=5)
    kfold_val_scores.append(np.mean(scores))
print(f"Best cross-validation score: {np.max(kfold_val_scores):.3f}")
best_n_neighbors = neighbors[np.argmax(kfold_val_scores)]
print("Best # of neighbors:", best_n_neighbors)
```

```
Best cross-validation score: 0.913
Best # of neighbors: 11
```

(Repeated) Stratified K-fold cross validation

- Similar to (repeated) k-fold cv, with one difference that each fold (and test data) is now a stratified sample.



Grid search with Stratified K-fold CV

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold
X_dev, X_test, y_dev, y_test = train_test_split(X, y, random_state=42)
sfold = StratifiedKFold(n_splits=5, shuffle=False)
sfold_val_scores = []
neighbors = np.arange(1, 20, 2)
for idx in neighbors:
    knn = KNeighborsClassifier(n_neighbors=idx)
    scores = cross_val_score(knn, X_dev, y_dev, cv=sfold)
    sfold_val_scores.append(np.mean(scores))
print(f"Best sfold cross-validation score: {np.max(sfold_val_scores):.3f}")
best_n_neighbors = neighbors[np.argmax(sfold_val_scores)]
print("Best # of neighbors:", best_n_neighbors)
```

```
Best sfold cross-validation score: 0.913
Best # of neighbors: 11
```

Random permutation CV

- A user specified number of train/validation datasets are created.
- For every pair, the development data is shuffled before creating the training & validation datasets
- The remaining steps are similar to K-fold CV.

Guidelines on model selection strategies

- **Three-way holdout** can give reasonable approximation of test performance on large balanced datasets
- **Leave-one-out CV** generally has high variance and is applied on small datasets
- **K-fold CV** works well in general, more stable and is used if model selection is not computationally expensive
- **Stratified** sampling is used when working with highly imbalanced datasets
- **Repeated** versions of K-fold CV and Stratified K-fold CV have lower variance, but are computationally expensive

Model evaluation

Evaluate model

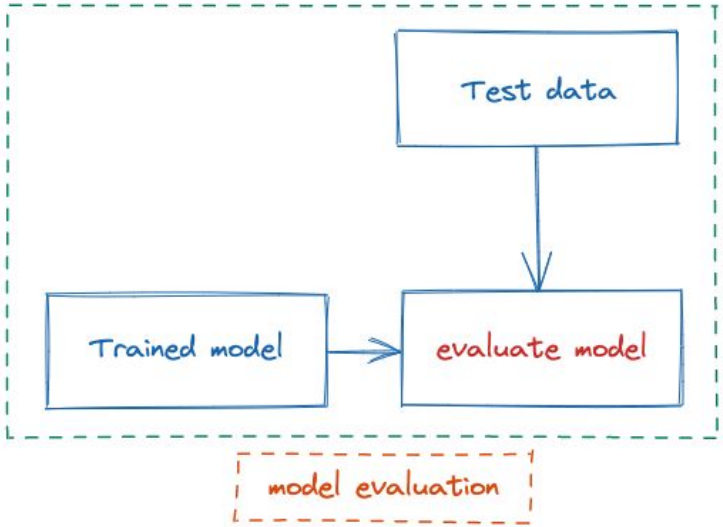
```python
from sklearn.neighbors import KNeighborsClassifier
X_dev, X_test, y_dev, y_test = train_test_split(X, y, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_dev, y_dev, random_state=84)
val_scores = []
neighbors = np.arange(1, 20, 2)
for idx in neighbors:
    knn = KNeighborsClassifier(n_neighbors=idx)
    knn.fit(X_train, y_train)
    val_scores.append(knn.score(X_val, y_val))
print(f"Best validation score: {np.max(val_scores):.3f}")
best_n_neighbors = neighbors[np.argmax(val_scores)]
print("Best # of neighbors:", best_n_neighbors)
knn.fit(X_dev, y_dev)
print(f"Test performance: {knn.score(X_test, y_test):.3f}")
```

```
Best validation score: 0.947
Best # of neighbors: 11
Test performance: 0.900
```

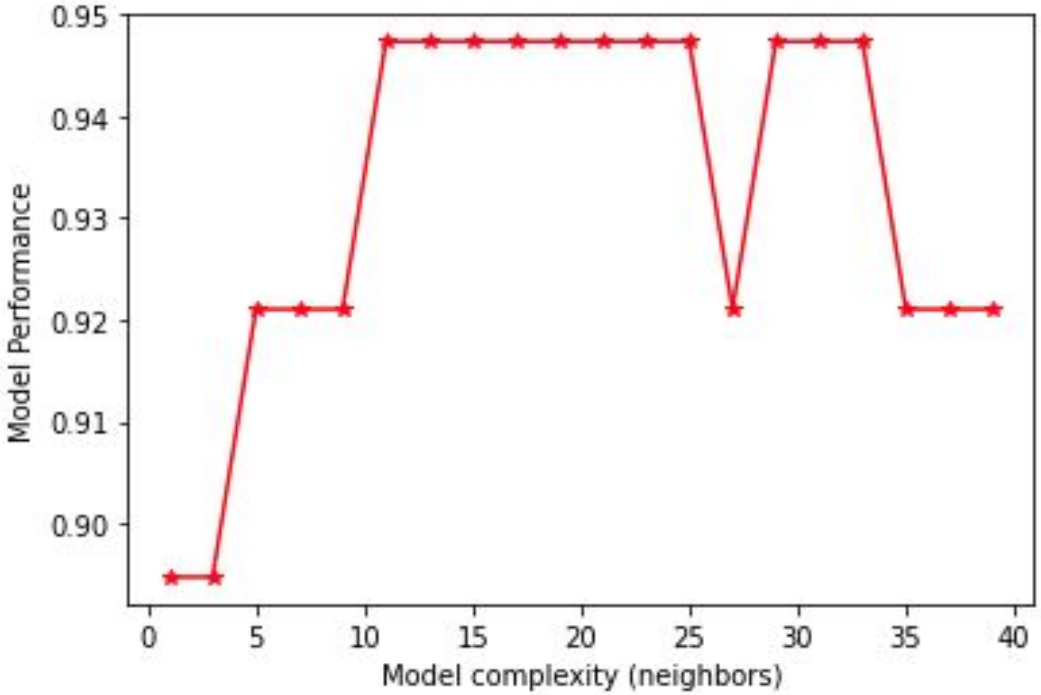
Three-way holdout

```
Best cross-validation score: 0.913
Best # of neighbors: 11
Test performance: 0.900
```

K-fold CV



Model complexity v.s. Model performance



Questions?

Let’s take a 10 min break!

Data preprocessing

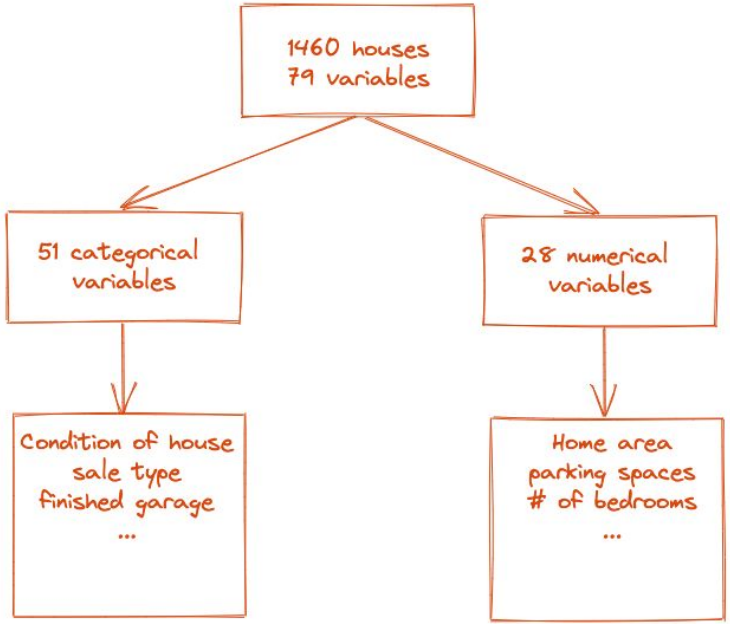
Data types

- Quantitative/numerical continuous - 1, 3.5, 100, 10^10, 3.14
- Quantitative/numerical discrete - 1, 2, 3, 4
- Qualitative/categorical unordered - cat, dog, whale
- Qualitative/categorical ordered - good, better, best
- Date or time - 09/15/2021, Jan 8th 2020 15:00:00
- Text - The quick brown fox jumps over the lazy dog

Numerical data

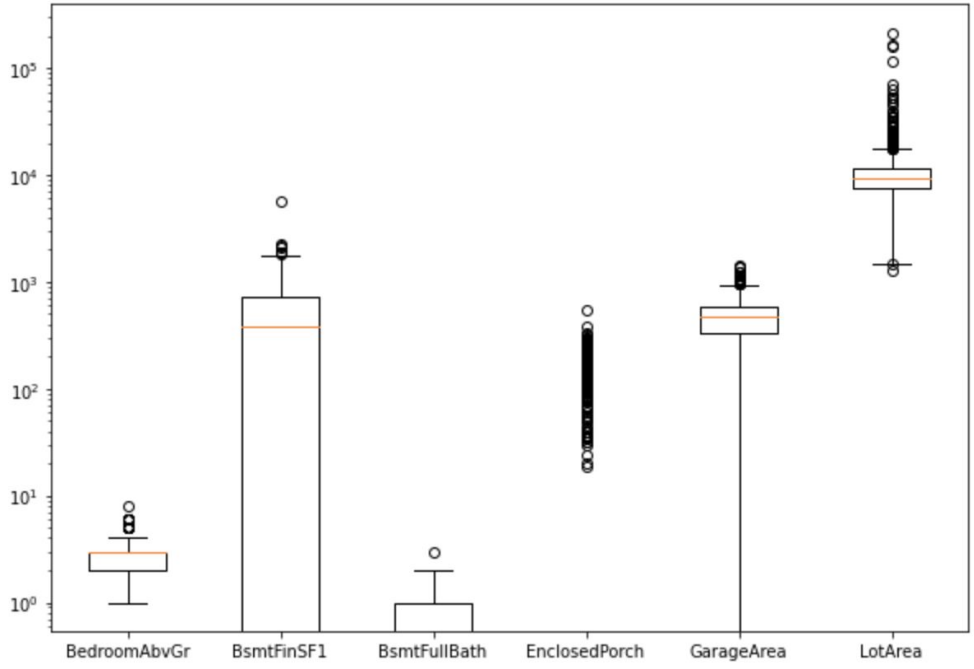
Example dataset

Predict sale price of houses in Ames, Iowa and identify factors that impact the sale price.

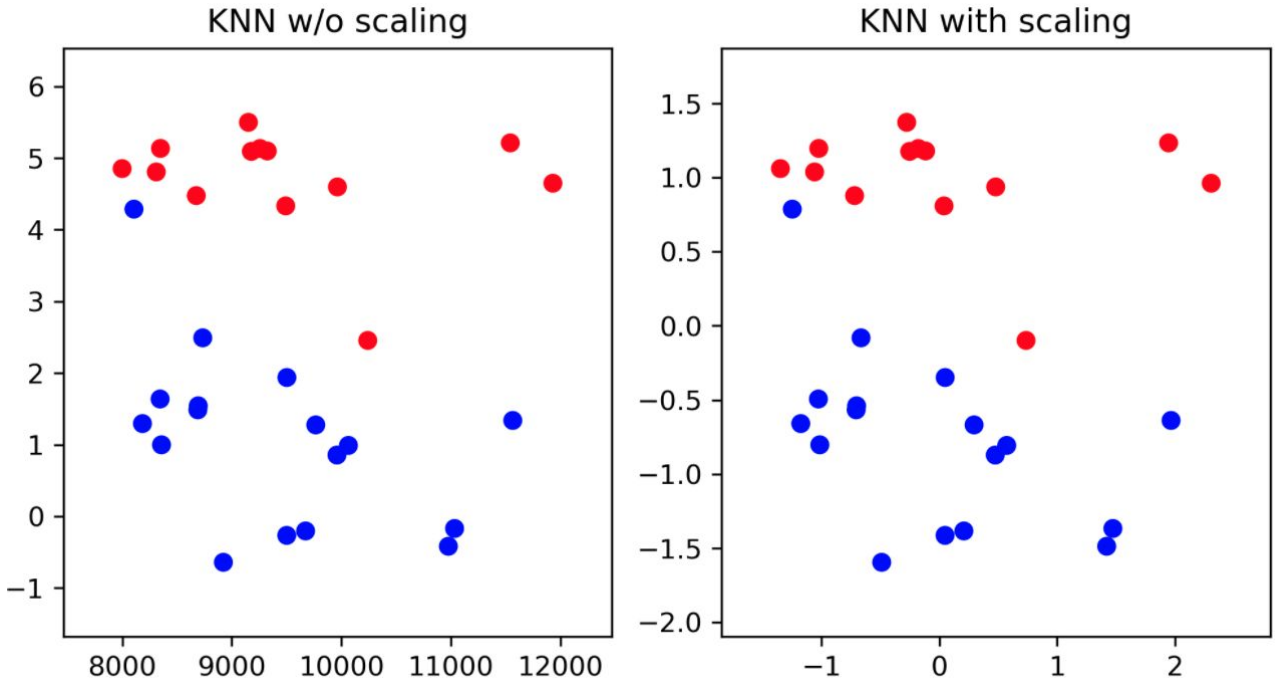


https://www.kaggle.com/c/house-prices-advanced-regression-techniques/overview/description

Numerical features

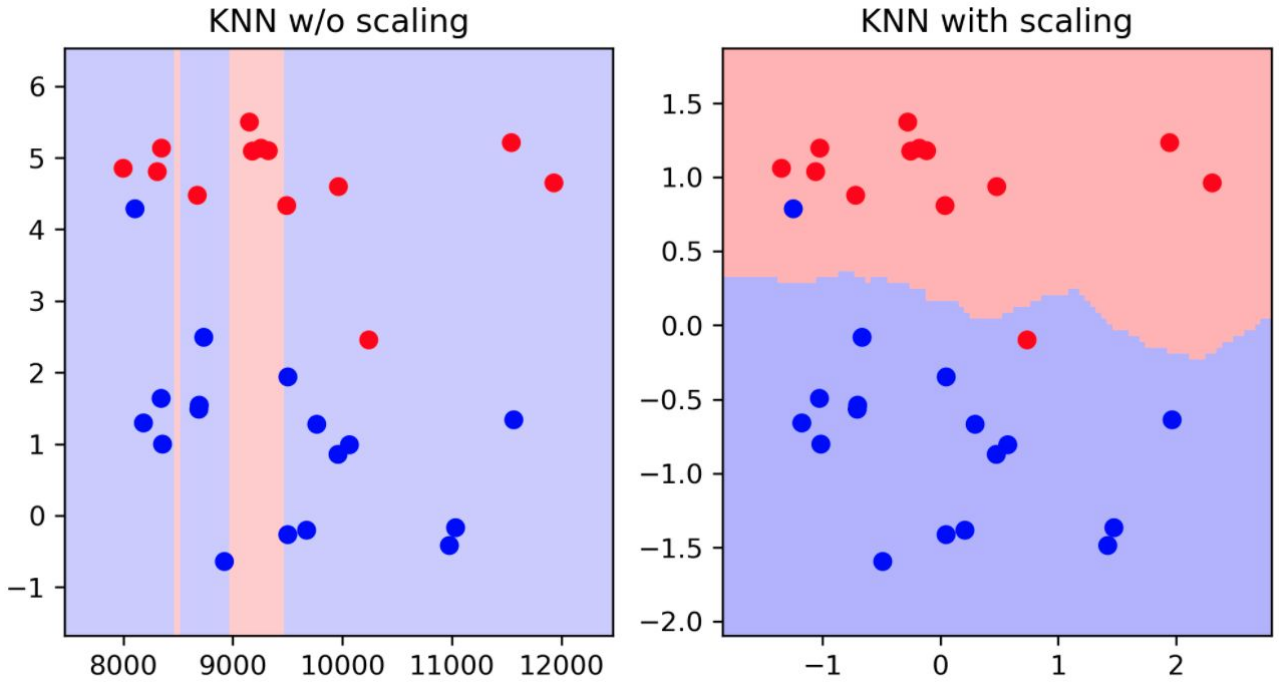


Why is scaling important?



https://amueller.github.io/COMS4995-s20/slides/aml-04-preprocessing/#6

Why is scaling important?



https://amueller.github.io/COMS4995-s20/slides/aml-04-preprocessing/#7

Scaling numerical features

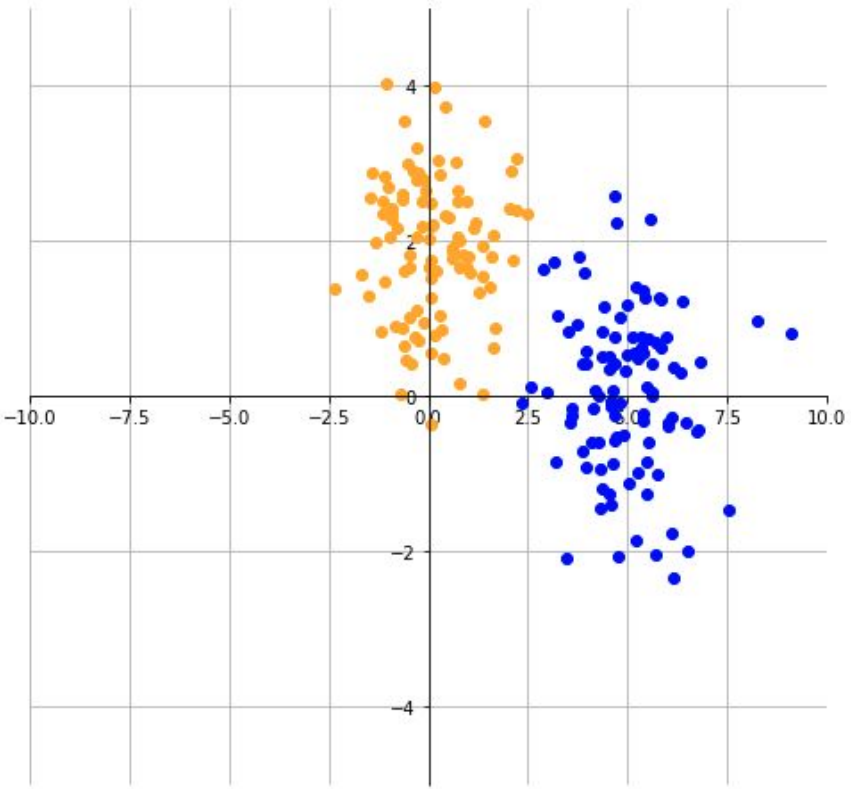
```
def generate_data(size, mean, cov, seed):
    m = multivariate_normal(mean=mean, cov=cov)
    samples = m.rvs(size=size, random_state = seed)
    return samples

sample_size = 100
cov = np.identity(2)
blue_data = generate_data(sample_size, [5, 0], cov, 2)
orange_data = generate_data(sample_size, [0, 2], cov, 4)

# plotting
fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.plot(orange_data[:, 0], orange_data[:, 1], 'o', color='orange')
ax.plot(blue_data[:, 0], blue_data[:, 1], 'o', color='blue')

ax.grid(True)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.set_xlim([-10, 10])
ax.set_ylim([-5, 5])

plt.show()
```



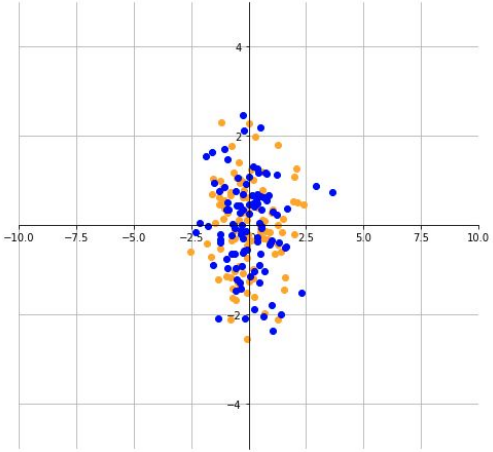
Standard Scaler

```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
blue_data_scaled = ss.fit_transform(blue_data)
orange_data_scaled = ss.fit_transform(orange_data)

fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.plot(orange_data_scaled[:, 0], orange_data_scaled[:, 1], 'o', color='orange')
ax.plot(blue_data_scaled[:, 0], blue_data_scaled[:, 1], 'o', color='blue')

ax.grid(True)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.set_xlim([-10, 10])
ax.set_ylim([-5, 5])

plt.show()
```



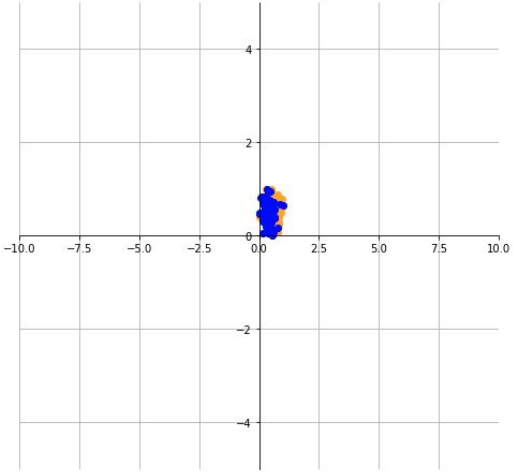
Min-max Scaler

```
mms = MinMaxScaler()
blue_data_scaled = mms.fit_transform(blue_data)
orange_data_scaled = mms.fit_transform(orange_data)

fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.plot(orange_data_scaled[:, 0], orange_data_scaled[:, 1], 'o', color='orange')
ax.plot(blue_data_scaled[:, 0], blue_data_scaled[:, 1], 'o', color='blue')

ax.grid(True)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.set_xlim([-10, 10])
ax.set_ylim([-5, 5])

plt.show()
```



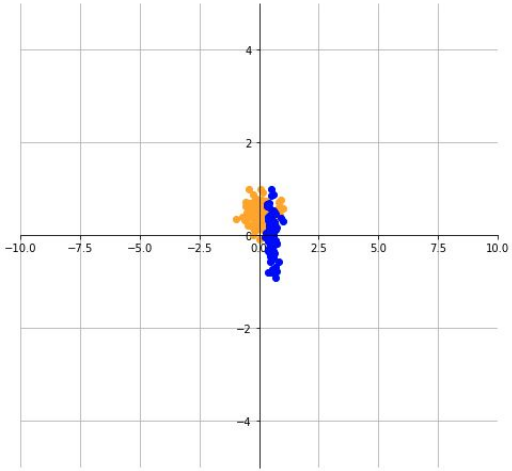
Max absolute Scaler

```
mas = MaxAbsScaler()
blue_data_scaled = mas.fit_transform(blue_data)
orange_data_scaled = mas.fit_transform(orange_data)

fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.plot(orange_data_scaled[:, 0], orange_data_scaled[:, 1], 'o', color='orange')
ax.plot(blue_data_scaled[:, 0], blue_data_scaled[:, 1], 'o', color='blue')

ax.grid(True)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.set_xlim([-10, 10])
ax.set_ylim([-5, 5])

plt.show()
```



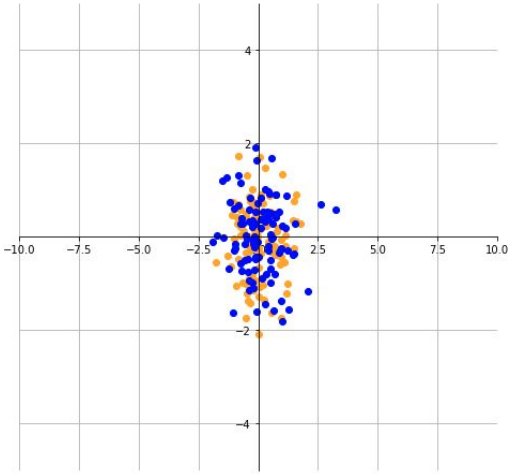
Robust Scaler

```
rs = RobustScaler()
blue_data_scaled = rs.fit_transform(blue_data)
orange_data_scaled = rs.fit_transform(orange_data)

fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.plot(orange_data_scaled[:, 0], orange_data_scaled[:, 1], 'o', color='orange')
ax.plot(blue_data_scaled[:, 0], blue_data_scaled[:, 1], 'o', color='blue')

ax.grid(True)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.set_xlim([-10, 10])
ax.set_ylim([-5, 5])

plt.show()
```



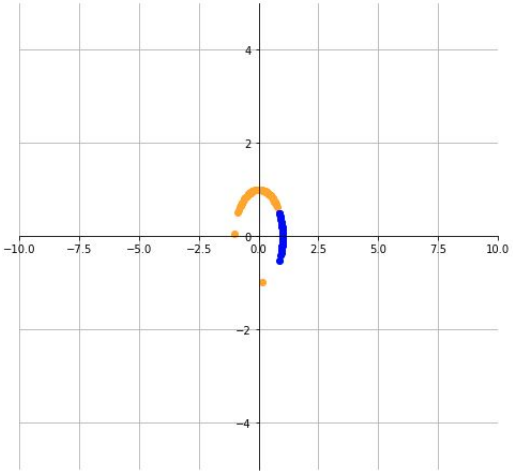
Normalizer

```
nz = Normalizer()
blue_data_scaled = nz.fit_transform(blue_data)
orange_data_scaled = nz.fit_transform(orange_data)

fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.plot(orange_data_scaled[:, 0], orange_data_scaled[:, 1], 'o', color='orange')
ax.plot(blue_data_scaled[:, 0], blue_data_scaled[:, 1], 'o', color='blue')

ax.grid(True)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')
ax.set_xlim([-10, 10])
ax.set_ylim([-5, 5])

plt.show()
```



Does scaling help?

```
from sklearn.linear_model import LinearRegression
less_numerical_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
target = data_df["SalePrice"]
num_df = data_df[less_numerical_features]
X_dev, X_test, y_dev, y_test = train_test_split(num_df, target, random_state = 42)
scaler = StandardScaler()
X_dev_scaled = scaler.fit_transform(X_dev)
X_test_scaled = scaler.transform(X_test)
lr_scaled = LinearRegression().fit(X_dev_scaled, y_dev)
lr_not_scaled = LinearRegression().fit(X_dev, y_dev)
print(f"R^2 coefficient (no scaling):", lr_not_scaled.score(X_test, y_test))
print(f"R^2 coefficient (with scaling):", lr_scaled.score(X_test_scaled, y_test))
```

```
R^2 coefficient (no scaling): 0.7239855269706834
R^2 coefficient (with scaling): 0.7239855269706912
```

→ **Not in this case!**

(and this is expected)

Does scaling help?

```
from sklearn.linear_model import Ridge
less_numerical_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
target = data_df["SalePrice"]
num_df = data_df[less_numerical_features]
X_dev, X_test, y_dev, y_test = train_test_split(num_df, target, random_state = 42)
scaler = StandardScaler()
X_dev_scaled = scaler.fit_transform(X_dev)
X_test_scaled = scaler.transform(X_test)
lr_scaled = Ridge().fit(X_dev_scaled, y_dev)
lr_not_scaled = Ridge().fit(X_dev, y_dev)
print(f"R^2 coefficient (no scaling):", lr_not_scaled.score(X_test, y_test))
print(f"R^2 coefficient (with scaling):", lr_scaled.score(X_test_scaled, y_test))
```

```
R^2 coefficient (no scaling): 0.7241056465716417
R^2 coefficient (with scaling): 0.7238704267774814
```

→ **Not in this case!**

(it generally does)

Does scaling help?

```
from sklearn.neighbors import KNeighborsRegressor
less_numerical_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
target = data_df["SalePrice"]
num_df = data_df[less_numerical_features]
X_dev, X_test, y_dev, y_test = train_test_split(num_df, target, random_state = 42)
scaler = StandardScaler()
X_dev_scaled = scaler.fit_transform(X_dev)
X_test_scaled = scaler.fit_transform(X_test)
lr_scaled = KNeighborsRegressor().fit(X_dev_scaled, y_dev)
lr_not_scaled = KNeighborsRegressor().fit(X_dev, y_dev)
print(f"R^2 coefficient (no scaling):", lr_not_scaled.score(X_test, y_test))
print(f"R^2 coefficient (with scaling):", lr_scaled.score(X_test_scaled, y_test))
```

```
R^2 coefficient (no scaling): 0.6336385002736059
R^2 coefficient (with scaling): 0.7517321165750432
```

→ It helps!

One common mistake

```
from sklearn.linear_model import Ridge
less_numerical_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
target = data_df["SalePrice"]
num_df = data_df[less_numerical_features]
X_dev, X_test, y_dev, y_test = train_test_split(num_df, target, random_state = 42)
scaler = StandardScaler()
X_dev_scaled = scaler.fit_transform(X_dev)
X_test_scaled = scaler.fit_transform(X_test)
lr_scaled = Ridge().fit(X_dev_scaled, y_dev)
lr_not_scaled = Ridge().fit(X_dev, y_dev)
print(f"R^2 coefficient (no scaling):", lr_not_scaled.score(X_test, y_test))
print(f"R^2 coefficient (with scaling):", lr_scaled.score(X_test_scaled, y_test))
```

The scaler from training step should be used to transform the test data

Pipeline API in sklearn

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import cross_val_score
less_numerical_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
target = data_df["SalePrice"]
num_df = data_df[less_numerical_features]
X_dev, X_test, y_dev, y_test = train_test_split(num_df, target, random_state = 42)
pipe = make_pipeline(StandardScaler(), Ridge())
pipe.fit(X_dev, y_dev)
print(f"Score on test set:", pipe.score(X_test, y_test))
```

```
Score on test set: 0.7238704267774814
```


This allows you to apply the same transformation on the test data

Scaling with cross-validation

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import cross_val_score
less_numerical_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
target = data_df["SalePrice"]
num_df = data_df[less_numerical_features]
X_dev, X_test, y_dev, y_test = train_test_split(num_df, target, random_state = 42)
scaler = StandardScaler()
X_dev_scaled = scaler.fit_transform(X_dev)
scores = cross_val_score(KNeighborsRegressor(), X_dev, y_dev, cv=10)
scores_scaled = cross_val_score(KNeighborsRegressor(), X_dev_scaled, y_dev, cv=10)
print(f"Mean score, std (no scaling):", np.mean(scores), np.std(scores))
print(f"Mean score, std (scaling):", np.mean(scores_scaled), np.std(scores_scaled))
```


Is this the correct?

```
Mean score, std (no scaling): 0.5725534970854226 0.06761409753263867
Mean score, std (scaling): 0.68318928861422 0.09084772090687601
```

Scaling with cross-validation - the right way

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=10)
scaler = StandardScaler()
scores = []
for train, val in kf.split(X_dev):
    X_train, y_train = X_dev.iloc[train], y_dev.iloc[train]
    X_val, y_val = X_dev.iloc[val], y_dev.iloc[val]
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)
    knn_regressor = KNeighborsRegressor()
    knn_regressor.fit(X_train_scaled, y_train)
    scores.append(knn_regressor.score(X_val_scaled, y_val))
print(np.mean(scores))
```

```
0.6855263281467252
```

Scaling with cross-validation - An easier way

```
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV
pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
params = {'kneighborsregressor__n_neighbors': [5]}
grid = GridSearchCV(pipe, params, cv=10)
grid.fit(X_dev, y_dev)
print(grid.cv_results_['mean_test_score'][0])
```

```
0.6855263281467252
```

Handling missing data

- It is important to know how missing values are encoded in dataset
- Often missingness is informative (often captured by adding missing indicator columns)
- Several ways to handle missing values:
  - Drop column (typically used as baseline)
  - Drop rows (if there are only a few with missing values)
  - Impute using mean or median (SimpleImputer in sklearn API)
  - kNN (neighbors are found using `nan_euclidean_distance` metric)
  - Regression models
  - Matrix factorization

Categorical data

Ways to handle categorical data

- Ordinal encoding
- One-hot encoding
- Target encoding

Example

```
less_cat_variables = [
    "LotShape",
    "Street",
    "Alley",
    "Neighborhood",
    "ExterCond"
]
cat_df = data_df[less_cat_variables]
```

```
cat_df.head()
```

	LotShape	Street	Alley	Neighborhood	ExterCond
0	Reg	Pave	NaN	CollgCr	TA
1	Reg	Pave	NaN	Veenker	TA
2	IR1	Pave	NaN	CollgCr	TA
3	IR1	Pave	NaN	Crawfor	TA
4	IR1	Pave	NaN	NoRidge	TA

ExterCond:

Ex	Excellent
Gd	Good
TA	Average/Typical
Fa	Fair
Po	Poor

Alley:

Grvl	Gravel
Pave	Paved
NA	No alley access

LotShape:

Reg	Regular
IR1	Slightly irregular
IR2	Moderately Irregular
IR3	Irregular

Neighborhood: Physical locations within Ames city limits

Blmngtn	Bloomington Heights
Blueste	Bluestem
BrDale	Briardale
BrkSide	Brookside
ClearCr	Clear Creek
CollgCr	College Creek
Crawfor	Crawford
Edwards	Edwards
Gilbert	Gilbert
IDOTRR	Iowa DOT and Rail Road
MeadowV	Meadow Village
Mitchel	Mitchell
Names	North Ames
NoRidge	Northridge
NPkVill	Northpark Villa
NridgHt	Northridge Heights
NWAmes	Northwest Ames
OldTown	Old Town
SWISU	South & West of Iowa State University
Sawyer	Sawyer
SawyerW	Sawyer West
Somerst	Somerset
StoneBr	Stone Brook
Timber	Timberland
Veenker	Veenker

Ordinal encoding

```
cat_df["Neighborhood_ord"] = cat_df.Neighborhood.astype("category").cat.codes
cat_df.head()
```

	LotShape	Street	Alley	Neighborhood	ExterCond	Neighborhood_ord
0	Reg	Pave	NaN	CollgCr	TA	5
1	Reg	Pave	NaN	Veenker	TA	24
2	IR1	Pave	NaN	CollgCr	TA	5
3	IR1	Pave	NaN	Crawfor	TA	6
4	IR1	Pave	NaN	NoRidge	TA	15

→ **What is wrong with this?**

Ordinal encoding

```
def map_ExterCond(val):
    if val == "Ex":
        return 4
    elif val == "Gd":
        return 3
    elif val == "TA":
        return 2
    elif val == "Fa":
        return 1
    elif val == "Po":
        return 0
    else:
        return -1
cat_df["ExterCond_ord"] = cat_df["ExterCond"].map(lambda x: map_ExterCond(x))
cat_df.head()
```

	LotShape	Street	Alley	Neighborhood	ExterCond	Neighborhood_ord	ExterCond_ord
0	Reg	Pave	NaN	CollgCr	TA	5	2
1	Reg	Pave	NaN	Veenker	TA	24	2
2	IR1	Pave	NaN	CollgCr	TA	5	2
3	IR1	Pave	NaN	Crawfor	TA	6	2
4	IR1	Pave	NaN	NoRidge	TA	15	2

→ **How about this one?**

Ordinal encoding - another way

```
from sklearn.preprocessing import OrdinalEncoder
enc = OrdinalEncoder(categories = [["Po", "Fa", "TA", "Gd", "Ex"]])
cat_df["ExterCond_ord"] = enc.fit_transform(cat_df["ExterCond"].to_numpy().reshape(-1, 1))
cat_df.head()
```

	LotShape	Street	Alley	Neighborhood	ExterCond	Neighborhood_ord	ExterCond_ord
0	Reg	Pave	NaN	CollgCr	TA	5	2.0
1	Reg	Pave	NaN	Veenker	TA	24	2.0
2	IR1	Pave	NaN	CollgCr	TA	5	2.0
3	IR1	Pave	NaN	Crawfor	TA	6	2.0
4	IR1	Pave	NaN	NoRidge	TA	15	2.0

One-hot encoding

```
cat_df["LotShape"].head()
```

```
0    Reg
1    Reg
2    IR1
3    IR1
4    IR1
Name: LotShape, dtype: object
```

```
pd.get_dummies(cat_df["LotShape"]).head()
```

	IR1	IR2	IR3	Reg
0	0	0	0	1
1	0	0	0	1
2	1	0	0	0
3	1	0	0	0
4	1	0	0	0

One-hot encoding

```
cat_df_transformed = pd.get_dummies(cat_df, dummy_na=True)
print(f"Before transformation:", cat_df.shape)
print(f"After transformation:", cat_df_transformed.shape)
cat_df_transformed.head()
```

```
Before transformation: (1460, 5)
After transformation: (1460, 43)
```

	LotShape_IR1	LotShape_IR2	LotShape_IR3	LotShape_Reg	LotShape_nan	Street_Grvl	Street_Pave	Street_nan	Alley_Grvl	Alley_Pave	...
0	0	0	0	1	0	0	1	0	0	0	...
1	0	0	0	1	0	0	1	0	0	0	...
2	1	0	0	0	0	0	1	0	0	0	...
3	1	0	0	0	0	0	1	0	0	0	...
4	1	0	0	0	0	0	1	0	0	0	...

One-hot encoding

```
cat_df["LotShape"].head()
```

```
0    Reg
1    Reg
2    IR1
3    IR1
4    IR1
Name: LotShape, dtype: object
```

```
pd.get_dummies(cat_df["LotShape"]).head()
```

	IR1	IR2	IR3	Reg
0	0	0	0	1
1	0	0	0	1
2	1	0	0	0
3	1	0	0	0
4	1	0	0	0

```
test_df = pd.DataFrame(
    ["Reg", "IR5", "IR1", "IR2", "IR4"],
    columns=["LotShape"])
test_df
```

	LotShape
0	Reg
1	IR5
2	IR1
3	IR2
4	IR4

How do we handle unseen categorical data in test data?

One-hot encoding

```
cat_feature = pd.Categorical(cat_df["LotShape"],
                             categories = ["Reg", "IR1", "IR2", "IR3", "IR4", "IR5"])
train_df = pd.get_dummies(cat_train_df)
train_df.head()
```

	Reg	IR1	IR2	IR3	IR4	IR5
0	1	0	0	0	0	0
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	1	0	0	0	0
4	0	1	0	0	0	0

Add category in training phase

One-hot encoding - another way

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
cat_df_transformed = ohe.fit_transform(cat_df)
print(type(cat_df_transformed))
print(cat_df_transformed.shape)
print(ohe.get_feature_names())
print(cat_df_transformed.toarray())
```

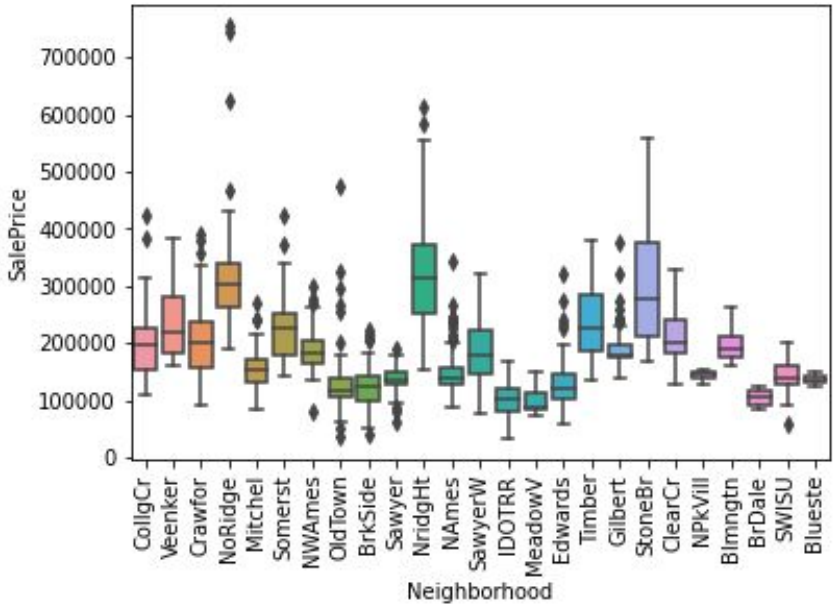
```
<class 'scipy.sparse.csr.csr_matrix'>
(1460, 39)
['x0_IR1' 'x0_IR2' 'x0_IR3' 'x0_Reg' 'x1_Grvl' 'x1_Pave' 'x2_Grvl'
 'x2_Pave' 'x2_nan' 'x3_Blmngtn' 'x3_Blueste' 'x3_BrDale' 'x3_BrkSide'
 'x3_ClearCr' 'x3_CollgCr' 'x3_Crawfor' 'x3_Edwards' 'x3_Gilbert'
 'x3_IDOTRR' 'x3_MeadowV' 'x3_Mitchel' 'x3_NAmes' 'x3_NPkVill' 'x3_NWAmes'
 'x3_NoRidge' 'x3_NridgHt' 'x3_OldTown' 'x3_SWISU' 'x3_Sawyer'
 'x3_SawyerW' 'x3_Somerst' 'x3_StoneBr' 'x3_Timber' 'x3_Veenker' 'x4_Ex'
 'x4_Fa' 'x4_Gd' 'x4_Po' 'x4_TA']
[[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]
 [1. 0. 0. ... 0. 0. 1.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]]
```

One-hot encoding considerations

- One-hot encoding introduces multi-collinearity
  - For e.g., x3 = 1 - x1 - x2 (in case when we have three categories)
  - Possible to remove one feature
  - Has implications on model interpretation
- Could be problematic for some models
  - Non-regularized regression techniques
- Some modeling techniques handle categorical features as-is
  - Tree-based models
  - Naive Bayes models
- Leads to high-dimensional datasets

Target encoding

- Generally applicable for high cardinality categorical features
- The encoding is specific to the problem type.
- **Regression:**
  - Average target value for each category
- **Binary classification:**
  - Probability of being in class 1
- **Multiclass classification**:
  - One feature per class that gives the probability distribution



Target encoding

```
from category_encoders import TargetEncoder
te = TargetEncoder(cols=["Neighborhood"]).fit(cat_df, target)
te.transform(cat_df).head()
```

	LotShape	Street	Alley	Neighborhood	ExterCond
0	Reg	Pave	NaN	197965.773333	TA
1	Reg	Pave	NaN	238770.100937	TA
2	IR1	Pave	NaN	197965.773333	TA
3	IR1	Pave	NaN	210624.725490	TA
4	IR1	Pave	NaN	335295.317073	TA

Average Sale price in neighborhood

Combining numerical & categorical data

```
from sklearn.compose import make_column_transformer

ohe_features = [
    "LotShape",
    "Street",
    "Alley",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(), ohe_features),
                        (TargetEncoder(), te_features)
                       )
preprocess.fit_transform(mixed_df, target)
print(preprocess.named_transformers_["onehotencoder"].get_feature_names())
print(preprocess.named_transformers_["targetencoder"].get_feature_names())
```

```
['x0_IR1' 'x0_IR2' 'x0_IR3' 'x0_Reg' 'x1_Grvl' 'x1_Pave' 'x2_Grvl'
 'x2_Pave' 'x2_nan' 'x3_Ex' 'x3_Fa' 'x3_Gd' 'x3_Po' 'x3_TA']
['Neighborhood']
```

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(), ohe_features),
                        (TargetEncoder(), te_features)
                       )
pipe = make_pipeline(preprocess, Ridge(alpha=5))
cross_val_score(pipe, mixed_df, target, cv=5)
```

```
array([       nan, 0.79740524, 0.77990227, 0.75366768, 0.64396256])
```

Why do we get a NaN?

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(), ohe_features),
                        (TargetEncoder(), te_features)
                       )
# mixed_df = preprocess.fit_transform(mixed_df, target)
pipe = make_pipeline(preprocess, Ridge(alpha=5))
cross_val_score(pipe, mixed_df, target, cv=5, error_score="raise")
```

```
ValueError: Found unknown categories ['Po'] in column 2 during transform
```

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                                     (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                                     (TargetEncoder(), te_features)
                                    )
pipe = make_pipeline(preprocess, Ridge(alpha=5))
cross_val_score(pipe, mixed_df, target, cv=5, error_score="raise")
```

```
array([0.82335453, 0.79740524, 0.77990227, 0.75366768, 0.64396256])
```

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (TargetEncoder(), te_features)
                       )
pipe = make_pipeline(preprocess, Ridge(alpha=5))
scores = cross_val_score(pipe, mixed_df, target, cv=5, error_score="raise")
print(scores)
print(np.mean(scores))
```

```
[0.82335453 0.79740524 0.77990227 0.75366768 0.64396256]
0.7596584551073746
```

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (OneHotEncoder(handle_unknown="ignore"), te_features),
                                remainder="passthrough"
                        )
pipe = make_pipeline(preprocess, Ridge(alpha=5))
scores = cross_val_score(pipe, mixed_df, target, cv=5, error_score="raise")
print(scores)
print(np.mean(scores))
```

```
[0.84043271 0.80967747 0.79038469 0.75975791 0.62000934]
0.7640524227182345
```

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import Ridge

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (OneHotEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                       )
pipe = make_pipeline(preprocess, Ridge(alpha=0.1))
scores = cross_val_score(pipe, mixed_df, target, cv=5, error_score="raise")
print(scores)
print(np.mean(scores))
```

```
[0.83124556 0.8118426  0.79143222 0.7530232  0.63461183]
0.76443108015182
```

End-to-end example

```
from sklearn.compose import make_column_transformer
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import GridSearchCV

ohe_features = [
    "LotShape",
    "Street",
    "ExterCond"
]

te_features = ["Neighborhood"]

num_features = [
    "BedroomAbvGr",
    "BsmtFinSF1",
    "BsmtFullBath",
    "EnclosedPorch",
    "GarageArea",
    "LotArea",
    "1stFlrSF",
    "2ndFlrSF",
]
mixed_df = data_df[ohe_features + te_features + num_features]
target = data_df["SalePrice"]

preprocess = make_column_transformer((StandardScaler(), num_features),
                        (OneHotEncoder(handle_unknown="ignore"), ohe_features),
                        (OneHotEncoder(handle_unknown="ignore"), te_features),
                                    remainder="passthrough"
                                   )
pipe = make_pipeline(preprocess, GridSearchCV(Ridge(), param_grid = [{"alpha":np.logspace(-3, 1, 20)}]))
pipe.fit(mixed_df, target)
print(f"Best score:", pipe.named_steps["gridsearchcv"].best_score_)
print(f"Best alpha:", pipe.named_steps["gridsearchcv"].best_params_)
```

```
Best score: 0.7649635829917203
Best alpha: {'alpha': 2.3357214690901213}
```

Textual data

We will look at preprocessing techniques related to textual data in lecture 10

Questions?