Missing data in supervised ML

Andras Zsom

Lead Data Scientist and Adjunct Lecturer in Data Science
Center for Computation and Visualization
Brown University

About me

- Born and raised in Hungary
- Astrophysics PhD at MPIA, Heidelberg, Germany
- Postdoctoral researcher at MIT (still in astrophysics at the time)
- Started at Brown in December 2015 as a Data Scientist
- Promoted to Lead Data Scientist in 2017
- Adjunct Lecturer in Data Science last fall
    - Teaching the course *DATA1030: Hands-on data science* to the DS master students at Brown

Data Science at Brown

- Center for Computation and Visualization (CCV) - https://ccv.brown.edu/ (https://ccv.brown.edu/)
- Institutional Data group
    - Data-driven decision support and predictive modeling for Brown's administrative units
    - Academic research on data-intensive projects

Learning Objectives

By the end of this workshop, you will be able to

- Describe the three main types of missingness patterns
- Evaluate simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- Apply multivariate imputation
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

Before we start, a few words on our dataset: kaggle house price

- good for educational purposes
  - messy data that requires quite a bit of preprocessing
  - a nice mixture of continuous, ordinal, and categorical features, each feature type has missing values
- lots of excellent kernels on kaggle
  - check them out here (https://www.kaggle.com/c/house-prices-advanced-regression-techniques)
- dataset and description available in repo
  - let's take a look!

https://github.com/brown-ccv/ODSC_East_2020 (https://github.com/brown-ccv/ODSC_East_2020)

Missing values often occur in datasets

- survey data: not everyone answers all the questions
- medical data: not all tests/treatments/etc are performed on all patients
- sensor can be offline or malfunctioning

Missing values are an issue for multiple reasons

Concenptual reason

- missing values can introduce biases
  - bias: the samples (the data points) are not representative of the underlying distribution/population
  - any conclusion drawn from a biased dataset is also biased.
  - rich people tend to not fill out survey questions about their salaries and the mean salary estimated from survey data tend to be lower than true value

Practical reason

- missing values (NaN, NA, inf) are incompatible with sklearn
  - all values in an array need to be numerical otherwise sklearn will throw a *ValueError*
- there are a few supervised ML techniques that work with missing values (e.g., XGBoost, CatBoost)
  - we will cover those later today

Learning Objectives

By the end of this workshop, you will be able to

- **Describe the three main types of missingness patterns**
- Evaluate simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- Apply multivariate imputation
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

Missing data patterns

- **MCAR** - Missing Complete At Random
  - some people skip some survey questions by accident
- **MAR** - Missing At Random
  - males are less likely to fill out a survey on depression
  - this has nothing to do with their level of depression after accounting for maleness
- **MNAR** - Missing Not At Random
  - depressed people are less likely to fill out a survey on depression due to their level of depression

MCAR test

- MCAR can be diagnosed with a statistical test (Little, 1988 (https://www.tandfonline.com/doi/abs/10.1080/01621459.1988.10478722))
  - python implementation available in the pymice (https://github.com/RianneSchouten/pymice) package or in the skipped slide
- Caveat: it can differentiate between MCAR and MAR only, it misses MNAR

In [1]:
```python
# from the pymice package
# https://github.com/RianneSchouten/pymice

import numpy as np
import pandas as pd
import math as ma
import scipy.stats as st

def checks_input_mcar_tests(data):
    """ Checks whether the input parameter of class McarTests is correct
            Parameters
            ----------
            data:
                The input of McarTests specified as 'data'
            Returns
            -------
            bool
                True if input is correct
            """

    if not isinstance(data, pd.DataFrame):
        print("Error: Data should be a Pandas DataFrame")
        return False

    if not any(data.dtypes.values == np.float):
        if not any(data.dtypes.values == np.int):
            print("Error: Dataset cannot contain other value types than
 floats and/or integers")
            return False

    if not data.isnull().values.any():
        print("Error: No NaN's in given data")
        return False

    return True


def mcar_test(data):
    """ Implementation of Little's MCAR test
    Parameters
    ----------
    data: Pandas DataFrame
        An incomplete dataset with samples as index and variables as col
umns
    Returns
    -------
    p_value: Float
        This value is the outcome of a chi-square statistical test, test
ing whether the null hypothesis
        'the missingness mechanism of the incomplete dataset is MCAR' ca
n be rejected.
    """

    if not checks_input_mcar_tests(data):
        raise Exception("Input not correct")
```

```
    dataset = data.copy()
    vars = dataset.dtypes.index.values
    n_var = dataset.shape[1]

    # mean and covariance estimates
    # ideally, this is done with a maximum likelihood estimator
    gmean = dataset.mean()
    gcov = dataset.cov()

    # set up missing data patterns
    r = 1 * dataset.isnull()
    mdp = np.dot(r, list(map(lambda x: ma.pow(2, x), range(n_var))))
    sorted_mdp = sorted(np.unique(mdp))
    n_pat = len(sorted_mdp)
    correct_mdp = list(map(lambda x: sorted_mdp.index(x), mdp))
    dataset['mdp'] = pd.Series(correct_mdp, index=dataset.index)

    # calculate statistic and df
    pj = 0
    d2 = 0
    for i in range(n_pat):
        dataset_temp = dataset.loc[dataset['mdp'] == i, vars]
        select_vars = ~dataset_temp.isnull().any()
        pj += np.sum(select_vars)
        select_vars = vars[select_vars]
        means = dataset_temp[select_vars].mean() - gmean[select_vars]
        select_cov = gcov.loc[select_vars, select_vars]
        mj = len(dataset_temp)
        parta = np.dot(means.T, np.linalg.solve(select_cov, np.identity(
select_cov.shape[1])))
        d2 += mj * (np.dot(parta, means))

    df = pj - n_var

    # perform test and save output
    p_value = 1 - st.chi2.cdf(d2, df)

    return p_value
```

MCAR, MAR, MNAR are nice in theory, pretty useless in practice

- it can be challenging to infer the missingness pattern from an incomplete dataset
  - There is a statistical test to differentiate MCAR and MAR
  - MNAR is difficult/impossible to diagnose to the best of my knowledge
- multiple patterns can be present in the data
  - even worse, multiple patterns can be present in one feature!
  - missing values in a feature can occur due to a mix of MCAR, MAR, MNAR

Learning Objectives

By the end of this workshop, you will be able to

- Describe the three main types of missingness patterns
- **Evaluate simple approaches for handling missing values**
- Apply XGBoost to a dataset with missing values
- Apply multivariate imputation
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

Simple approaches for handling missing values

- 1) categorical/ordinal features: treat missing values as another category
  - missing values in categorical/ordinal features are not a big deal
- 2) continuous features: this is the tough part
  - sklearn's SimpleImputer
- 3) exclude points or features with missing values
  - might be OK

1a) Missing values in a categorical feature

- YAY - this is not an issue at all!
- Categorical feature needs to be one-hot encoded anyway
- Just replace the missing values with 'NA' or 'missing' and treat it as a separate category

1b) Missing values in a ordinal feature

- this can be a bit trickier but usually fine
- Ordinal encoder is applied to ordinal features
  - where does 'NA' or 'missing' fit into the order of the categories?
  - usually first or last
- if you can figure this out, you are done

In [2]:

```
# read the data
import pandas as pd
import numpy  as np
from sklearn.model_selection import train_test_split

# Let's load the data
df = pd.read_csv('data/train.csv')
# drop the ID
df.drop(columns=['Id'],inplace=True)

# the target variable
y = df['SalePrice']
df.drop(columns=['SalePrice'],inplace=True)
# the unprocessed feature matrix
X = df.values
print(X.shape)
# the feature names
ftrs = df.columns
```

```
(1460, 79)
```

In [3]:

```
# let's split to train, test, and holdout
X_other, X_holdout, y_other, y_holdout = train_test_split(df, y, test_si
ze=0.2, random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X_other, y_other, te
st_size=0.25, random_state=0)

print(X_train.shape)
print(X_test.shape)
print(X_holdout.shape)
```

```
(876, 79)
(292, 79)
(292, 79)
```

In [4]:

```python
# collect the various features
cat_ftrs = ['MSZoning','Street','Alley','LandContour','LotConfig','Neighborhood','Condition1','Condition2',\
            'BldgType','HouseStyle','RoofStyle','RoofMatl','Exterior1st','Exterior2nd','MasVnrType','Foundation',\
           'Heating','CentralAir','Electrical','GarageType','PavedDrive','MiscFeature','SaleType','SaleCondition']
ordinal_ftrs = ['LotShape','Utilities','LandSlope','ExterQual','ExterCond','BsmtQual','BsmtCond','BsmtExposure',\
               'BsmtFinType1','BsmtFinType2','HeatingQC','KitchenQual','Functional','FireplaceQu','GarageFinish',\
               'GarageQual','GarageCond','PoolQC','Fence']
ordinal_cats = [['Reg','IR1','IR2','IR3'],['AllPub','NoSewr','NoSeWa','ELO'],['Gtl','Mod','Sev'],\
               ['Po','Fa','TA','Gd','Ex'],['Po','Fa','TA','Gd','Ex'],['NA','Po','Fa','TA','Gd','Ex'],\
               ['NA','Po','Fa','TA','Gd','Ex'],['NA','No','Mn','Av','Gd'],['NA','Unf','LwQ','Rec','BLQ','ALQ','GLQ'],\
               ['NA','Unf','LwQ','Rec','BLQ','ALQ','GLQ'],['Po','Fa','TA','Gd','Ex'],['Po','Fa','TA','Gd','Ex'],\
               ['Sal','Sev','Maj2','Maj1','Mod','Min2','Min1','Typ'],['NA','Po','Fa','TA','Gd','Ex'],\
               ['NA','Unf','RFn','Fin'],['NA','Po','Fa','TA','Gd','Ex'],['NA','Po','Fa','TA','Gd','Ex'],
               ['NA','Fa','TA','Gd','Ex'],['NA','MnWw','GdWo','MnPrv','GdPrv']]
num_ftrs = ['MSSubClass','LotFrontage','LotArea','OverallQual','OverallCond','YearBuilt','YearRemodAdd',\
             'MasVnrArea','BsmtFinSF1','BsmtFinSF2','BsmtUnfSF','TotalBsmtSF','1stFlrSF','2ndFlrSF',\
             'LowQualFinSF','GrLivArea','BsmtFullBath','BsmtHalfBath','FullBath','HalfBath','BedroomAbvGr',\
             'KitchenAbvGr','TotRmsAbvGrd','Fireplaces','GarageYrBlt','GarageCars','GarageArea','WoodDeckSF',\
             'OpenPorchSF','EnclosedPorch','3SsnPorch','ScreenPorch','PoolArea','MiscVal','MoSold','YrSold']
```

In [5]:

```
# preprocess with pipeline and columntransformer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

# one-hot encoder
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant',fill_value='missing'
)),
    ('onehot', OneHotEncoder(sparse=False,handle_unknown='ignore'))])

# ordinal encoder
ordinal_transformer = Pipeline(steps=[
    ('imputer2', SimpleImputer(strategy='constant',fill_value='NA')),
    ('ordinal', OrdinalEncoder(categories = ordinal_cats))])

# standard scaler
numeric_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())])

# collect the transformers
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, num_ftrs),
        ('cat', categorical_transformer, cat_ftrs),
        ('ord', ordinal_transformer, ordinal_ftrs)])
```

In [6]:

```
# fit_transform the training set
X_prep = preprocessor.fit_transform(X_train)
# little hacky, but collect feature names
feature_names = preprocessor.transformers_[0][-1] + \
                list(preprocessor.named_transformers_['cat'][1].get_feat
ure_names(cat_ftrs)) + \
                preprocessor.transformers_[2][-1]

df_train = pd.DataFrame(data=X_prep,columns=feature_names)
print(df_train.shape)

# transform the test
df_test = preprocessor.transform(X_test)
df_test = pd.DataFrame(data=df_test,columns = feature_names)
print(df_test.shape)

# transform  the holdout
df_holdout = preprocessor.transform(X_holdout)
df_holdout = pd.DataFrame(data=df_holdout,columns = feature_names)
print(df_holdout.shape)
```

```
(876, 221)
(292, 221)
(292, 221)
```

2) Continuous features: mean or median imputation

- Imputation means you infer the missing values from the known part of the data
- sklearn's SimpleImputer can do mean and median imputation
- USUALLY A BAD IDEA!
  - MCAR: mean/median of non-missing values is the same as the mean/median of the true underlying distribution, but the variances are different
  - not MCAR: the mean/median and the variance of the completed dataset will be off
  - supervised ML model is too confident (MCAR) or systematically off (not MCAR)

3) Exclude points or features with missing values

- easy to do with pandas
- it is an ACCEPTABLE approach under two conditions:
  - Little's test supports MCAR ($p > 0.05$)
  - only small fraction of points contain missing values (maybe a few percent?) OR the missing values are limited to one or a few features that can be dropped
- if the MCAR assumption is justified, dropping points will not introduce biases to your model
- due to the smaller sample size, the confidence of your model might suffer.
- what will you do with missing values when you deploy the model?

In [7]:
```
print('data dimensions:',df_train.shape)
print('the p value of the mcar test:',mcar_test(df_train))
perc_missing_per_ftr = df_train.isnull().sum(axis=0)/df_train.shape[0]
print('fraction of missing values in features:')
print(perc_missing_per_ftr[perc_missing_per_ftr > 0])
frac_missing = sum(df_train.isnull().sum(axis=1)!=0)/df_train.shape[0]
print('fraction of points with missing values:',frac_missing)
```

```
data dimensions: (876, 221)
the p value of the mcar test: 1.0
fraction of missing values in features:
LotFrontage    0.173516
MasVnrArea     0.004566
GarageYrBlt    0.050228
dtype: float64
fraction of points with missing values: 0.2237442922374429
```

In [8]:
```
print(df_train.shape)
# by default, rows/points are dropped
df_r = df_train.dropna()
print(df_r.shape)
# drop features with missing values
df_c = df_train.dropna(axis=1)
print(df_c.shape)
```

```
(876, 221)
(680, 221)
(876, 218)
```

Learning Objectives

By the end of this workshop, you will be able to

- Describe the three main types of missingness patterns
- Evaluate simple approaches for handling missing values
- **Apply XGBoost to a dataset with missing values**
- Apply multivariate imputation
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

XGBoost and missing values

- sklearn raises an error if the feature matrix (X) contains nans.
- XGBoost doesn't!
- If a feature with missing values is split:
  - XGBoost tries to put the points with missing values to the left and right
  - calculates the impurity measure for both options
  - puts the points with missing values to the side with the lower impurity
- if missingness correlates with the target variable, XGBoost extracts this info!

In [9]:

```
import xgboost
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error

param_grid = {"learning_rate": [0.03],
              "n_estimators": [2000],
              "seed": [0],
              #"n_jobs": [-1],
              #"reg_alpha": [0e0,0.1,0.31622777,1.,3.16227766,10.],
              #"reg_lambda": [0e0,0.1,0.31622777,1.,3.16227766,10.],
              "missing": [np.nan],
              #"max_depth": [1,2,3,4,5],
              "colsample_bytree": [0.9],
              "subsample": [0.66]}

XGB = xgboost.XGBRegressor()
XGB.set_params(**ParameterGrid(param_grid)[0])
XGB.fit(df_train,y_train,early_stopping_rounds=50,eval_set=[(df_test, y_
test)], verbose=False)
print('the test RMSE:',XGB.evals_result()['validation_0']['rmse'][-1])
y_holdout_pred = XGB.predict(df_holdout)
print('the holdout RMSE:',np.sqrt(mean_squared_error(y_holdout,y_holdout
_pred)))
```

```
the test RMSE: 23486.925781
the holdout RMSE: 31748.96283078089
```

Learning Objectives

By the end of this workshop, you will be able to

- Describe the three main types of missingness patterns
- Evaluate simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- **Apply multivariate imputation**
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

Multivariate Imputation

- models each feature with missing values as a function of other features
  - at each step, a feature with nans is designated as target variable y and the other features are treated as feature matrix X
  - a regressor is trained on (X, y) for known y
  - then, the regressor is used to predict the missing values of y
- in the ML pipeline:
  - create n imputed datasets
  - run all of them through the ML pipeline
  - generate n holdout scores
  - the uncertainty in the holdout scores is due to the uncertainty in imputation
- works on MCAR and MAR, fails on MNAR
- paper [here (https://www.jstatsoft.org/article/view/v045i03)](https://www.jstatsoft.org/article/view/v045i03)

sklearn's IterativeImputer

In [10]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor

print(df_train[['LotFrontage','MasVnrArea','GarageYrBlt']].head())

imputer = IterativeImputer(estimator = RandomForestRegressor(n_estimator
s=10), random_state=0)
X_impute = imputer.fit_transform(df_train)
df_train_imp = pd.DataFrame(data=X_impute, columns = df_train.columns)

print(df_train_imp[['LotFrontage','MasVnrArea','GarageYrBlt']].head())

df_test_imp = pd.DataFrame(data=imputer.transform(df_test), columns = df
_train.columns)
df_holdout_imp = pd.DataFrame(data=imputer.transform(df_holdout), column
s = df_train.columns)
```

```
   LotFrontage  MasVnrArea  GarageYrBlt
0     0.424926   -0.573303     0.979398
1          NaN    0.492835     1.018748
2          NaN   -0.573303     0.192399
3    -0.049970    0.810076    -0.476551
4    -1.474659   -0.022031     0.979398
   LotFrontage  MasVnrArea  GarageYrBlt
0     0.424926   -0.573303     0.979398
1    -1.258797    0.492835     1.018748
2    -0.516232   -0.573303     0.192399
3    -0.049970    0.810076    -0.476551
4    -1.474659   -0.022031     0.979398

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/sklea
rn/impute/_iterative.py:638: ConvergenceWarning: [IterativeImputer] Ear
ly stopping criterion not reached.
  " reached.", ConvergenceWarning)
```

In [11]:
```
XGB.fit(df_train_imp,y_train,early_stopping_rounds=50,eval_set=[(df_test
_imp, y_test)], verbose=False)
print('the test RMSE:',XGB.evals_result()['validation_0']['rmse'][-1])
y_holdout_pred = XGB.predict(df_holdout_imp)
print('the holdout RMSE:',np.sqrt(mean_squared_error(y_holdout,y_holdout
_pred)))
```

```
the test RMSE: 23531.046875
the holdout RMSE: 33248.31021868423
```

Learning Objectives

By the end of this workshop, you will be able to

- Describe the three main types of missingness patterns
- Evaluate simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- Apply multivariate imputation
- **Apply the reduced-features model (also called the pattern submodel approach)**
- Decide which approach is best for your dataset

Reduced-features model (or pattern submodel approach)

- first described in 2007 in a [JMLR article (http://www.jmlr.org/papers/v8/saar-tsechansky07a.html)](http://www.jmlr.org/papers/v8/saar-tsechansky07a.html) as the reduced features model
- in 2018, "rediscovered" as the pattern submodel approach in [Biostatistics (https://www.ncbi.nlm.nih.gov/pubmed/30203058)](https://www.ncbi.nlm.nih.gov/pubmed/30203058)

My holdout set:

index	feature 1	feature 2	feature 3	target var
0	NA	45	NA	0
1	NA	NA	8	1
2	12	6	34	0
3	1	89	NA	0
4	0	NA	47	1
5	687	24	67	1
6	NA	23	NA	1

To predict points 0 and 6, I will use train and test points that are complete in feature 2.

To predict point 1, I will use train and test points that are complete in feature 3.

To predict point 2 and 5, I will use train and test points that are complete in features 1-3.

Etc. We will train as many models as the number of patterns in holdout.

How to determine the patterns?

In [12]:
```python
mask = df_holdout[['LotFrontage','MasVnrArea','GarageYrBlt']].isnull()
unique_rows, counts = np.unique(mask, axis=0,return_counts=True)
print(unique_rows.shape) # 6 patterns, we will train 6 models
for i in range(len(counts)):
    print(unique_rows[i],counts[i])
```

```
(6, 3)
[False False False] 223
[False False  True] 21
[False  True False] 1
[ True False False] 44
[ True False  True] 2
[ True  True False] 1
```

In [13]:

```
def xgb_model(X_train, Y_train, X_test, Y_test, X_holdout, Y_holdout, ve
rbose=1):

    # make into row vectors to avoid an obnoxious sklearn/xgb warning
    Y_train = np.reshape(np.array(Y_train), (1, -1)).ravel()
    Y_test = np.reshape(np.array(Y_test), (1, -1)).ravel()
    Y_holdout = np.reshape(np.array(Y_holdout), (1, -1)).ravel()

    XGB = xgboost.XGBRegressor(n_jobs=1)

    # find the best parameter set
    param_grid = {"learning_rate": [0.03],
                  "n_estimators": [2000],
                  "seed": [0],
                  #"n_jobs": [6],
                  #"reg_alpha": [0e0,0.1,0.31622777,1.,3.16227766,10.],
                  #"reg_lambda": [0e0,0.1,0.31622777,1.,3.16227766,10.],
                  "missing": [np.nan],
                  #"max_depth": [1,2,3,4,5],
                  "colsample_bytree": [0.9],
                  "subsample": [0.66]}

    pg = ParameterGrid(param_grid)

    scores = np.zeros(len(pg))

    for i in range(len(pg)):
        if verbose >= 5:
            print("Param set " + str(i + 1) + " / " + str(len(pg)))
        params = pg[i]
        XGB.set_params(**params)
        eval_set = [(X_test, Y_test)]
        XGB.fit(X_train, Y_train,
                early_stopping_rounds=50, eval_set=eval_set, verbose=Fal
se)# with early stopping
        Y_test_pred = XGB.predict(X_test, ntree_limit=XGB.best_ntree_lim
it)
        scores[i] = mean_squared_error(Y_test,Y_test_pred)

    best_params = np.array(pg)[scores == np.max(scores)]
    if verbose >= 4:
        print('Test set max score and best parameters are:')
        print(np.max(scores))
        print(best_params)

    # test the model on the holdout set with best parameter set
    XGB.set_params(**best_params[0])
    XGB.fit(X_train,Y_train,
            early_stopping_rounds=50,eval_set=eval_set, verbose=False)
    Y_holdout_pred = XGB.predict(X_holdout, ntree_limit=XGB.best_ntree_l
imit)

    if verbose >= 1:
        print ('The MSE is:',mean_squared_error(Y_holdout,Y_holdout_pred
))
    if verbose >= 2:
```

```
        print ('The predictions are:')
        print (Y_holdout_pred)
    if verbose >= 3:
        print("Feature importances:")
        print(XGB.feature_importances_)

    return (mean_squared_error(Y_holdout,Y_holdout_pred), Y_holdout_pred
, XGB.feature_importances_)

# Function: Reduced-feature XGB model
# all the inputs need to be pandas DataFrame
def reduced_feature_xgb(X_train, Y_train, X_test, Y_test, X_holdout, Y_h
oldout):

    # find all unique patterns of missing value in holdout set
    mask = X_holdout.isnull()
    unique_rows = np.array(np.unique(mask, axis=0))
    all_Y_holdout_pred = pd.DataFrame()

    print('there are', len(unique_rows), 'unique missing value pattern
s.')

    # divide holdout sets into subgroups according to the unique pattern
s
    for i in range(len(unique_rows)):
        print ('working on unique pattern', i)
        ## generate X_holdout subset that matches the unique pattern i
        sub_X_holdout = pd.DataFrame()
        sub_Y_holdout = pd.Series()
        for j in range(len(mask)): # check each row in mask
            row_mask = np.array(mask.iloc[j])
            if np.array_equal(row_mask, unique_rows[i]): # if the patter
n matches the ith unique pattern
                sub_X_holdout = sub_X_holdout.append(X_holdout.iloc[j])#
append the according X_holdout row j to the subset
                sub_Y_holdout = sub_Y_holdout.append(Y_holdout.iloc[[j
]])# append the according Y_holdout row j
        sub_X_holdout = sub_X_holdout[X_holdout.columns[~unique_rows[i
]]]

        ## choose the according reduced features for subgroups
        sub_X_train = pd.DataFrame()
        sub_Y_train = pd.DataFrame()
        sub_X_test = pd.DataFrame()
        sub_Y_test = pd.DataFrame()
        # 1.cut the feature columns that have nans in the according sub_
X_holdout
        sub_X_train = X_train[X_train.columns[~unique_rows[i]]]
        sub_X_test = X_test[X_test.columns[~unique_rows[i]]]
        # 2.cut the rows in the sub_X_train and sub_X_test that have any
nans
        sub_X_train = sub_X_train.dropna()
        sub_X_test = sub_X_test.dropna()
        # 3.cut the sub_Y_train and sub_Y_test accordingly
        sub_Y_train = Y_train.iloc[sub_X_train.index]
        sub_Y_test = Y_test.iloc[sub_X_test.index]
```

```
        # run XGB
        sub_Y_holdout_pred = xgb_model(sub_X_train, sub_Y_train, sub_X_t
est,
                                       sub_Y_test, sub_X_holdout, sub_Y_
holdout, verbose=0)
        sub_Y_holdout_pred = pd.DataFrame(sub_Y_holdout_pred[1],columns=
['sub_Y_holdout_pred'],
                                          index=sub_Y_holdout.index)
        print('    RMSE:',np.sqrt(mean_squared_error(sub_Y_holdout,sub_Y_
holdout_pred)))
        # collect the holdout predictions
        all_Y_holdout_pred = all_Y_holdout_pred.append(sub_Y_holdout_pre
d)

    # rank the final Y_holdout_pred according to original Y_holdout inde
x
    all_Y_holdout_pred = all_Y_holdout_pred.sort_index()
    Y_holdout = Y_holdout.sort_index()

    # get global RMSE
    total_RMSE = np.sqrt(mean_squared_error(Y_holdout,all_Y_holdout_pred
))

    return total_RMSE
```

A python implementation is available on the skipped slide

In [14]:
```
print('final RMSE:',reduced_feature_xgb(df_train, y_train, df_test, y_te
st, df_holdout, y_holdout))
```

```
there are 6 unique missing value patterns.
working on unique pattern 0

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/ipyke
rnel_launcher.py:76: DeprecationWarning: The default dtype for empty Se
ries will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.

   RMSE: 35277.53669207676
working on unique pattern 1

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/ipyke
rnel_launcher.py:76: DeprecationWarning: The default dtype for empty Se
ries will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.

   RMSE: 11607.857261825593
working on unique pattern 2

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/ipyke
rnel_launcher.py:76: DeprecationWarning: The default dtype for empty Se
ries will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.

   RMSE: 1134.5625
working on unique pattern 3

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/ipyke
rnel_launcher.py:76: DeprecationWarning: The default dtype for empty Se
ries will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.

   RMSE: 18366.394043603428
working on unique pattern 4

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/ipyke
rnel_launcher.py:76: DeprecationWarning: The default dtype for empty Se
ries will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.

   RMSE: 18521.340554971906
working on unique pattern 5

/anaconda3/envs/datasci_v0.0.2_local4/lib/python3.6/site-packages/ipyke
rnel_launcher.py:76: DeprecationWarning: The default dtype for empty Se
ries will be 'object' instead of 'float64' in a future version. Specify
a dtype explicitly to silence this warning.

   RMSE: 65343.46875
final RMSE: 32061.23877235819
```

Learning Objectives

By the end of this workshop, you will be able to

- Describe the three main types of missingness patterns
- Evaluate simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- Apply multivariate imputation
- Apply the reduced-features model (also called the pattern submodel approach)
- **Decide which approach is best for your dataset**

Which approach is best for my data?

- **XGB**: run n XGB models with n different seeds
- **imputation**: prepare n different imputations and run n XGB models on them
- **reduced-features**: run n reduced-features model with n different seeds
- rank the three methods based on how significantly different the corresponding mean scores are

Now you can

- Describe the three main types of missingness patterns
- Evaluate simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- Apply multivariate imputation
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

Thanks for your attention!