Mud card answers

- **For the example of seizure dataset, I understand if the kfold CV is using blindly, the some datapoint of one patient will be distributed into different place, but I don't understand why that is the the bad case of data leakage, what happens if the new patient data will be seen for the first time**
  - the model will perform poorly
  - you can rewrite my code to simulate that scenario
  - take data from two random patients and create a fourth set
  - train a model as we did in class with the stratified split, and apply that model to the fourth set
- **In the first example, seizure project data set. Is it a time series data? Because that we can see a time series pattern in the feature seizure_ID. If it is a time series data. Does it mean that we could apply group k-fold to deal with some time series data with groups**
  - the measurements are time series
  - but the features I calculated (mean, std, etc of 30 sec blocks) are not
- **What's the best way to check if iid fails and should you do it for every dataset before you start the preprocessing step?**
  - there is no statistical test for this
  - but yes, it needs to be determined for every dataset
  - you should do it before you start splitting because your splitting strategy depends on this
  - generally, if you have some sort of ID in your dataset, it's likely not iid
  - if you work with a time series data, it's not iid either
- **I'm kind of confused how non time dependent features are built into an autoregression like we did in the lectures. Im finding a little difficult to get the lag matrix straight, and that seems like an added complexity**
- **How do we reconcile a case where we have both time series and non time series features?**
  - work out the indices of the lag matrix to get a better understanding or take a simple time series data and figure out what the lag matrix and the target variable are
  - you'll have X1 (the non-time dependent feature matrix) and X2 (the time-dependent feature matrix) and you merge them into X
- **are there applications where one might use the time series approach along a dimension other than time?**
  - possibly?
  - I have no clue what you have in mind :)
- **Why the correlation with the date itself is one? I am sort of lost**
  - write down that the equation of the Pearson correlation coefficient is (e.g., Eq 3 here) and work out what happens if x_i = y_i

- **So after plotting the autocorrelation, we would find the lag i, that is not 0 but maximize the autocorrelation, and use it as the max shift? Or how do we use it?**
  - you use it to check for any sort of periodicity or seasonality
  - it also gives you an idea how many features you might need to use in the lag matrix

Missing data, part 2

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

- review simple approaches for handling missing values
- Apply XGBoost to a dataset with missing values
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

We continue working with the house price data set

- regression problem
- categorical, ordinal, continuous features
- missing data in all feature types

In [1]:
```
# 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 [2]:
```
perc_missing_per_ftr = df.isnull().sum(axis=0)/df.shape[0]
print('fraction of missing values in features:')
print(perc_missing_per_ftr[perc_missing_per_ftr > 0])
print('data types of the features with missing values:')
print(df[perc_missing_per_ftr[perc_missing_per_ftr > 0].index].dtypes)
frac_missing = sum(df.isnull().sum(axis=1)!=0)/df.shape[0]
print('fraction of points with missing values:',frac_missing)
```

```
fraction of missing values in features:
LotFrontage     0.177397
Alley           0.937671
MasVnrType      0.005479
```

```
MasVnrArea      0.005479
BsmtQual        0.025342
BsmtCond        0.025342
BsmtExposure    0.026027
BsmtFinType1    0.025342
BsmtFinType2    0.026027
Electrical      0.000685
FireplaceQu     0.472603
GarageType      0.055479
GarageYrBlt     0.055479
GarageFinish    0.055479
GarageQual      0.055479
GarageCond      0.055479
PoolQC          0.995205
Fence           0.807534
MiscFeature     0.963014
dtype: float64
data types of the features with missing values:
LotFrontage     float64
Alley            object
MasVnrType       object
MasVnrArea      float64
BsmtQual         object
BsmtCond         object
BsmtExposure     object
BsmtFinType1     object
BsmtFinType2     object
Electrical       object
FireplaceQu      object
GarageType       object
GarageYrBlt     float64
GarageFinish     object
GarageQual       object
GarageCond       object
PoolQC           object
Fence            object
MiscFeature      object
dtype: object
fraction of points with missing values: 1.0
```

Missing data, part 2

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

- **review simple approaches for handling missing values**
- Apply XGBoost to a dataset with missing values
- 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
- 4) multivariate imputation
  - 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 fine

In [3]:
```
# let's split to train, CV, and test
X_other, X_test, y_other, y_test = train_test_split(df, y, test_size=0.2, random
X_train, X_CV, y_train, y_CV = train_test_split(X_other, y_other, test_size=0.25

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

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

In [4]:
```
# collect the various features
cat_ftrs = ['MSZoning','Street','Alley','LandContour','LotConfig','Neighborhood'
            'BldgType','HouseStyle','RoofStyle','RoofMatl','Exterior1st','Exteri
           'Heating','CentralAir','Electrical','GarageType','PavedDrive','MiscFe
ordinal_ftrs = ['LotShape','Utilities','LandSlope','ExterQual','ExterCond','Bsmt
                'BsmtFinType1','BsmtFinType2','HeatingQC','KitchenQual','Function
                'GarageQual','GarageCond','PoolQC','Fence']
ordinal_cats = [['Reg','IR1','IR2','IR3'],['AllPub','NoSewr','NoSeWa','ELO'],['G
                ['Po','Fa','TA','Gd','Ex'],['Po','Fa','TA','Gd','Ex'],['NA','Po',
                ['NA','Po','Fa','TA','Gd','Ex'],['NA','No','Mn','Av','Gd'],['NA',
                ['NA','Unf','LwQ','Rec','BLQ','ALQ','GLQ'],['Po','Fa','TA','Gd','
                ['Sal','Sev','Maj2','Maj1','Mod','Min2','Min1','Typ'],['NA','Po',
                ['NA','Unf','RFn','Fin'],['NA','Po','Fa','TA','Gd','Ex'],['NA','P
                ['NA','Fa','TA','Gd','Ex'],['NA','MnWw','GdWo','MnPrv','GdPrv']]
num_ftrs = ['MSSubClass','LotFrontage','LotArea','OverallQual','OverallCond','Ye
            'MasVnrArea','BsmtFinSF1','BsmtFinSF2','BsmtUnfSF','TotalBsmtSF','1
            'LowQualFinSF','GrLivArea','BsmtFullBath','BsmtHalfBath','FullBath'
            'KitchenAbvGr','TotRmsAbvGrd','Fireplaces','GarageYrBlt','GarageCar
            'OpenPorchSF','EnclosedPorch','3SsnPorch','ScreenPorch','PoolArea',
```

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_feature_name
                preprocessor.transformers_[2][-1]

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

# transform the CV
df_CV = preprocessor.transform(X_CV)
df_CV = pd.DataFrame(data=df_CV,columns = feature_names)
print(df_CV.shape)

# transform the test
df_test = preprocessor.transform(X_test)
df_test = pd.DataFrame(data=df_test,columns = feature_names)
print(df_test.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 sometimes:
  - only small fraction of points contain missing values (maybe a few percent?)
  - or the missing values are limited to one or a few features and a large fraction of values are missing from those features (maybe up to 90%?)
- 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)
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)
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)
```

4) Multivariate Imputation

- **Does it make sense to impute the values?**
  - GarageYearBuilt should not be imputed because a missing value indicates no garage on the property
- models each feature with missing values as a function of other features, and uses that estimate for imputation

    - at each step, a feature is designated as target variable and the other feature columns 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 test scores
    - the uncertainty in the test scores is due to the uncertainty in imputation
- works on MCAR and MAR, fails on MNAR
- paper here

sklearn's IterativeImputer

In [9]:
```
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_estimators=10), r
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_CV_imp = pd.DataFrame(data=imputer.transform(df_CV), columns = df_train.colum
df_test_imp = pd.DataFrame(data=imputer.transform(df_test), columns = df_train.c
```

```
   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    -1.405584    0.492835     1.018748
2    -0.304687   -0.573303     0.192399
3    -0.049970    0.810076    -0.476551
4    -1.474659   -0.022031     0.979398
/Users/azsom/opt/anaconda3/envs/data1030/lib/python3.9/site-packages/sklearn/imp
ute/_iterative.py:685: ConvergenceWarning: [IterativeImputer] Early stopping cri
terion not reached.
  warnings.warn("[IterativeImputer] Early stopping criterion not"
```

Quiz

Missing data, part 2

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

- review simple approaches for handling missing values
- **Apply XGBoost to a dataset with missing values**
- Apply the reduced-features model (also called the pattern submodel approach)
- Decide which approach is best for your dataset

XGBoost

- eXtreme Gradient Boosting - a popular tree-based method
- blog post and paper
- more advanced than random forest
  - it has l1 and l2 regularization while random forest does not
  - trees are not independent
    - the next tree is built to improve the previous tree
    - less trees are necessary to achieve same accuracy
    - but XGBoost trees can overfit - more on this in the problem set
  - handles missing values well

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 [10]:

```
import xgboost
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

param_grid = {"learning_rate": [0.03],
              "n_estimators": [10000],
              "seed": [0],
              #"reg_alpha": [0e0, 1e-2, 1e-1, 1e0, 1e1, 1e2],
              #"reg_lambda": [0e0, 1e-2, 1e-1, 1e0, 1e1, 1e2],
              "missing": [np.nan],
              #"max_depth": [1,3,10,30,100],
              "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_CV, y_CV)], verb
y_CV_pred = XGB.predict(df_CV)
print('the CV RMSE:',np.sqrt(mean_squared_error(y_CV,y_CV_pred)))
y_test_pred = XGB.predict(df_test)
print('the test RMSE:',np.sqrt(mean_squared_error(y_test,y_test_pred)))
print('the test R2:',r2_score(y_test,y_test_pred))
```

```
the CV RMSE: 23470.132687324658
the test RMSE: 31748.96283078089
the test R2: 0.8540372805542484
```

XGBoost with the imputed data:

In [11]:

```
XGB.fit(df_train_imp,y_train,early_stopping_rounds=50,eval_set=[(df_CV_imp, y_CV
y_CV_pred = XGB.predict(df_CV_imp)
print('the CV RMSE:',np.sqrt(mean_squared_error(y_CV,y_CV_pred)))
y_test_pred = XGB.predict(df_test_imp)
print('the test RMSE:',np.sqrt(mean_squared_error(y_test,y_test_pred)))
print('the test R2:',r2_score(y_test,y_test_pred))
```

```
the CV RMSE: 23524.287408821077
the test RMSE: 32745.130769398726
the test R2: 0.8447340161321133
```

Quiz

Mudcard

In [ ]: