Mudcard answers

- **Can all models we learned be used to impute missing values? Should we compare different models when trying to impute the missing values?**
    - do you mean in the iterative imputer?
    - I recommend using non-deterministic ML algorithms in there like a random forest
- **can you elaborate "early_stopping_rounds"**
- **"What's the eval_set=[(df_CV, y_CV)] in the XGB do? Sort of misunderstanding here**
    - check out here and here
- **How can missingness affect the correlation with the target variable, and how is this beneficial in the XGBoost model?**
    - here is a hypothetical example:
    - you have a balanced classification problem (50% in class 0 and 50% in class 1)
    - one feature contains missing values
    - you see in the training set that points with 90% of the missing values belong to class 0, and 10% of points with missing values belong to class 1
    - when you see a missing value in that feature in the test set for example, XGBoost will know that the chances of that happening is 90% so that point is more likely to belong to class 0 than to class 1
- **For quiz 1, I rerun the LinearRegression estimator several times, I get the answers all the same. But the result of mine is not the same as my classmate's, a little bit different, I am a little confused.**
    - something must be different, you would need to investigate this more closely
- **I was a bit confused with the first quiz. My two imputed values in LinearRegression did not match the RFR as you mentioned it should.**
    - you misunderstood, the LinReg results should not match the RFR results
    - the LinReg results should remain the same if you rerun the same cell
    - the RFR results will be different each time you rerun the cell
- **if multivariate imputation with random forest shows low uncertainty, is it ok to continue multivariate imputation with a different model?**
    - sure but by the time you figure that out, you are already done developing the ML pipeline
- **Can we use XGBoost directly on the time series (non-iid) data? Or we need to remove the dependency among data points first?**
    - if you create autoregressive features, you can use XGB
    - XGB is not a neural network, you can't feed it one time series
- **With multivar imputation, I'm unsure why its valuable to have a model that returns an imputation with differing values!**
    - imputation is another source of uncertainty in your test score
    - by creating several imputed datasets and test scores, you'll be able to measure the uncertainty
- **can we use XGBoost with other models or are we restricted to using it when we have complicated missing value scenarios.**

    - you can use it on any structured dataset, it doesn't need to contain missing values
    - XGB is just another ML algorithm
- **So, XGBoost is a regression model using random forest?**
    - nope
    - it can be both regression and classification
    - it does not use a random forest, it uses decision trees

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

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]:

```
# 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 [3]:

```
# 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 [4]:

```
# 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 [5]:

```
# 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)
```

Reduced-features model (or pattern submodel approach)

- first described in 2007 in a JMLR article as the reduced features model
- in 2018, "rediscovered" as the pattern submodel approach in Biostatistics

My test 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 CV points that are complete in feature 2.

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

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

Etc. We will train as many models as the number of patterns in test/deployment.

How to determine the patterns?

In [6]:

```
mask = df_test[['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 [7]:

```
import xgboost
```

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

def xgb_model(X_train, Y_train, X_CV, y_CV, X_test, y_test, verbose=1):

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

    XGB = xgboost.XGBRegressor(n_jobs=1)

    # find the best parameter set
    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]}

    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_CV, y_CV)]
        XGB.fit(X_train, Y_train,
                early_stopping_rounds=50, eval_set=eval_set, verbose=False)# wit
        y_CV_pred = XGB.predict(X_CV, ntree_limit=XGB.best_ntree_limit)
        scores[i] = mean_squared_error(y_CV,y_CV_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 test 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_test_pred = XGB.predict(X_test, ntree_limit=XGB.best_ntree_limit)

    if verbose >= 1:
        print ('The MSE is:',mean_squared_error(y_test,y_test_pred))
    if verbose >= 2:
        print ('The predictions are:')
        print (y_test_pred)
    if verbose >= 3:
        print("Feature importances:")
        print(XGB.feature_importances_)

    return (mean_squared_error(y_test,y_test_pred), y_test_pred, XGB.feature_imp
# Function: Reduced-feature XGB model
# all the inputs need to be pandas DataFrame
def reduced_feature_xgb(X_train, Y_train, X_CV, y_CV, X_test, y_test):

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

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

    # divide test sets into subgroups according to the unique patterns
    for i in range(len(unique_rows)):
        print ('working on unique pattern', i)
        ## generate X_test subset that matches the unique pattern i
        sub_X_test = pd.DataFrame()
        sub_y_test = pd.Series(dtype=float)
        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 pattern matche
                sub_X_test = sub_X_test.append(X_test.iloc[j])# append the accor
                sub_y_test = sub_y_test.append(y_test.iloc[[j]])# append the acc
        sub_X_test = sub_X_test[X_test.columns[~unique_rows[i]]]

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

        # run XGB
        sub_y_test_pred = xgb_model(sub_X_train, sub_Y_train, sub_X_CV,
                                    sub_y_CV, sub_X_test, sub_y_test, verbose
        sub_y_test_pred = pd.DataFrame(sub_y_test_pred[1],columns=['sub_y_test_p
                                       index=sub_y_test.index)
        print('    RMSE:',np.sqrt(mean_squared_error(sub_y_test,sub_y_test_pred))
        # collect the test predictions
        all_y_test_pred = all_y_test_pred.append(sub_y_test_pred)

    # rank the final y_test_pred according to original y_test index
    all_y_test_pred = all_y_test_pred.sort_index()
    y_test = y_test.sort_index()

    # get global RMSE
    total_RMSE = np.sqrt(mean_squared_error(y_test,all_y_test_pred))
    total_R2 =  r2_score(y_test,all_y_test_pred)
    return total_RMSE, total_R2
```

In [8]:

```
RMSE, R2 = reduced_feature_xgb(df_train, y_train, df_CV, y_CV, df_test, y_test)
```

```
print('final RMSE:', RMSE)
print('final R2:', R2)
```

```
there are 6 unique missing value patterns.
working on unique pattern 0
   RMSE: 35277.53667892742
working on unique pattern 1
   RMSE: 11607.856743646213
working on unique pattern 2
   RMSE: 1134.5625
working on unique pattern 3
   RMSE: 18366.394043603428
working on unique pattern 4
   RMSE: 18521.340554971906
working on unique pattern 5
   RMSE: 65343.46875
final RMSE: 32061.238747816282
final R2: 0.8511518443924384
```

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**

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

A note on imbalanced datasets

- we learnt that a classification problem is imbalanced if more than 90-95% of the points belong to one class (class 0) and only a small fraction of the points belong to the other class (class 1)
  - fraud detection
  - sick or not sick (usually by far most people are not sick)
- we learnt to not use a metric that relies on the True Negatives in the confusion matrix
  - no accuracy or ROC
  - use f_beta or the precision-recall curve instead

What else can I do if I have an imbalanced dataset?

- most (but not all) classification algorithms we covered have a parameter called `class_weight` which allows you to assign more weight to the class 1 point
  - a misclassified class 1 point will contribute more to the cost function than a misclassified class 0 point
  - read the manual on `class_weight` because the different algorithms have slightly different definitions for this parameter
  - usually you can use `None`, `balanced`, or manually define what the class weight should be
  - it is worthwhile to tune this parameter if you have an imbalanced dataset
- resample/augment the dataset
  - SMOTE (Synthetic Minority Over-sampling Technique), see the paper
  - to improve the balance of the problem, new class 1 examples are synthesized from the existing examples
  - be careful though!
    - while resampling improves the balance of the dataset, the results of the model can be misleading
    - when you deploy the model, the incoming data will be as imbalanced as the original data

Misleading results with resampling

- let's assume you have an imbalanced dataset with 99% of points in class 0 and 1% of points in class 1
- you resample it such that the improved class balance is 50-50
- here the confusion matrix of the trained model:

		Predicted class	
		Predicted Negative (0)	Predicted Positive (1)
Actual class	Condition Negative (0)	**True Negative (TN): 45%**	**False Positive (FP): 5%**
	Condition Positive (1)	**False Negative (FN): 5%**	**True Positive (TP): 45%**

- 90% accuracy which is well above the 50% baseline accuracy!
- the precision, recall, and f1 scores are all 0.9.
- it looks great, doesn't it?
- let's rewrite the confusion matrix to reflect rates on the Condition Negative and Condition Positive points!

		Predicted class	
		Predicted Negative (0)	Predicted Positive (1)
Actual class	Condition Negative (0) 50% of the points	**90% of CNs are correctly classified**	**10% of CNs are incorrectly classified**
	Condition Positive (1) 50% of	**10% of CPs are incorrectly**	**90% of CPs are correctly**

	the points	**classified**	**classified**

Let's deploy this model

- the incoming data has the same balance as the original dataset (99% to 1%)
- let's assume we have 1e5 new points, 9.9e4 belongs to class 0, 1000 belongs to class 1
- what will be the numbers in the confusion matrix?

		Predicted class	
		Predicted Negative (0)	Predicted Positive (1)
Actual class	Condition Negative (0) 99000 points	**True Negative (TN): 99000 * 0.9 = 89100**	**False Positive (FP): 99000 * 0.1 = 9900**
	Condition Positive (1) 1000 points	**False Negative (FN): 1000 * 0.1 = 100**	**True Positive (TP): 1000 * 0.9 = 900**

- the accuracy of this model is still 0.90 but now it is well below the baseline of 0.99!
- recall is good (0.90) but the precision is not great (~0.083)
- the f1 score is ~0.15
- the false positives are overwhelming
- this is why you need to be careful with resampling

Quiz

Mudcard