

# Intermediate Machine Learning

Tej Gorde



**Women in  
Data Science  
Worldwide**

# Index

- Missing Values
- Categorical Variables
- Pipelines
- Cross- Validation
- XGBoost
- Data Leakage

# Missing Values

## 3 Approaches :

### 1) A Simple Option: Drop Columns with Missing Values

Dropping columns with missing values can be wasteful - imagine losing an entire column of 10,000 rows just because of one missing value!

Bed	Bath	
1.0	1.0	
2.0	1.0	
3.0	2.0	
NaN	2.0	



Bath
1.0
1.0
2.0
2.0

### 2) A Better Option: Imputation

Imputation fills in the missing values with some number. For instance, we can fill in the mean value along each column. The imputed value won't be exactly right in most cases, but it usually leads to more accurate models than you would get from dropping the column entirely.

Bed	Bath	
1.0	1.0	
2.0	1.0	
3.0	2.0	
NaN	2.0	



Bed	Bath	
1.0	1.0	
2.0	1.0	
3.0	2.0	
2.0	2.0	

### 3) An Extension To Imputation

Beyond basic imputation, you can create indicator columns to track which values were originally missing - this can help if missing data itself is meaningful, though it doesn't always improve model performance. My Grad school professor, preferred this method and flagged the filled-in values.

Bed	Bath	Bed_was_missing
1.0	1.0	FALSE
2.0	1.0	FALSE
3.0	2.0	FALSE
NaN	2.0	TRUE



## Measure Quality of each approach

We define a function `score_dataset()` to compare different approaches to dealing with missing values. This function reports the mean absolute error (MAE) from a random forest model.

MAE =  $\text{sum}(|\text{predicted\_value} - \text{actual\_value}|) / \text{number\_of\_predictions}$

### 1) A Simple Option: Drop Columns with Missing Values

```
# Get names of columns with missing values
cols_with_missing = [col for col in X_train.columns
                     if X_train[col].isnull().any()]

# Drop columns in training and validation data
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)

print("MAE from Approach 1 (Drop columns with missing values):")
print(score_dataset(reduced_X_train, reduced_X_valid, y_train, y_valid))
```

MAE from Approach 1 (Drop columns with missing values):

183550.22137772635

- Lower MAE = Better predictions
- Easy to interpret (in same units as target variable)
- In this case, MAE of 183,550 means predictions are off by \$ 183,550 on average



## 2) A Better Option: Imputation

```
from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns

print("MAE from Approach 2 (Imputation):")
print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))
```

MAE from Approach 2 (Imputation):  
178166.46269899711

## 3) An Extension To Imputation

```
# Make copy to avoid changing original data (when imputing)
X_train_plus = X_train.copy()
X_valid_plus = X_valid.copy()

# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

# Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus))
imputed_X_valid_plus = pd.DataFrame(my_imputer.transform(X_valid_plus))

# Imputation removed column names; put them back
imputed_X_train_plus.columns = X_train_plus.columns
imputed_X_valid_plus.columns = X_valid_plus.columns

print("MAE from Approach 3 (An Extension to Imputation):")
print(score_dataset(imputed_X_train_plus, imputed_X_valid_plus, y_train, y_valid))
```

MAE from Approach 3 (An Extension to Imputation):  
178927.503183954

Approach 2 is better than Approach 1

**So, why did imputation perform better than dropping the columns?**

The training data has 10864 rows and 12 columns, where three columns contain missing data. For each column, less than half of the entries are missing. Thus, dropping the columns removes a lot of useful information, and so it makes sense that imputation would perform better.

**So, why did extension imputation perform worse imputation of columns?**

1. Added complexity: The additional indicator columns might introduce noise rather than signal, making the model work harder without providing meaningful patterns

2. Data characteristics: In this housing dataset, the fact that a value was missing may not actually correlate with the target (house price). For example, a missing bedroom size doesn't necessarily mean the house is more or less expensive

Remember: More complex doesn't always mean better - sometimes simpler approaches work best depending on your data's characteristics.



## Handling Categorical Variables

A categorical variable has a fixed set of possible string values.

Examples:

- Days of week: "Monday", "Tuesday", etc.
- Car brands: "Honda", "Toyota", "Ford"
- Survey responses: "Never", "Sometimes", "Always"

Most ML models require numerical input, so categorical data needs preprocessing.

There are 3 approaches to preprocess categorical data :

1. Drop Categorical Columns
2. Label Encoding
3. One-hot encoding

Note : ML models require numeric input, hence important to check data type of all the columns during data analysis.



## 1) Drop Categorical Variables

The easiest approach to dealing with categorical variables is to simply remove them from the dataset. This approach will only work well if the columns did not contain useful information.

## 2) Ordinal Encoding / Label Encoding

Ordinal encoding assigns each unique value to a different integer.

A diagram illustrating the process of ordinal encoding. On the left, a table titled "Breakfast" lists five categories: "Every day", "Never", "Rarely", "Most days", and "Never". An arrow points to the right, where another table with the same title "Breakfast" shows the corresponding numerical values: 3, 0, 1, 2, and 0 respectively.

Breakfast
Every day
Never
Rarely
Most days
Never

→

Breakfast
3
0
1
2
0

## 3) One-Hot Encoding

One-hot encoding is useful because it treats categories fairly - instead of saying "Red = 1, Blue = 2, Green = 3" (which suggests Green is somehow "more" than Red), it simply marks whether each category is present or not. This makes more sense for things like colors, car brands, or cities where there's no natural ordering between them.

Just don't use it when you have too many categories (like zip codes) as you'll end up with too many new columns!

A diagram illustrating the process of one-hot encoding. On the left, a table titled "Color" lists five categories: "Red", "Red", "Yellow", "Green", and "Yellow". An arrow points to the right, where a 5x3 matrix is shown. The matrix has columns labeled "Red", "Yellow", and "Green". The rows correspond to the color names. A "1" is placed in the cell at the intersection of "Red" and the first column, and so on for the other colors.

Color	Red	Yellow	Green
Red	1	0	0
Red	1	0	0
Yellow	0	1	0
Green	0	0	1
Yellow	0	1	0

### Why is it better ?

One-hot encoding simply tells the model "this color is present (1) or absent (0)", avoiding these misleading numerical relationships.



## 1) Drop Categorical Columns

```
drop_X_train = X_train.select_dtypes(exclude=['object'])
drop_X_valid = X_valid.select_dtypes(exclude=['object'])

print("MAE from Approach 1 (Drop categorical variables):")
print(score_dataset(drop_X_train, drop_X_valid, y_train, y_valid))
```

MAE from Approach 1 (Drop categorical variables):  
175703.48185157913

## 2) Ordinal Encoding

```
from sklearn.preprocessing import OrdinalEncoder

# Make copy to avoid changing original data
label_X_train = X_train.copy()
label_X_valid = X_valid.copy()

# Apply ordinal encoder to each column with categorical data
ordinal_encoder = OrdinalEncoder()
label_X_train[object_cols] = ordinal_encoder.fit_transform(X_train[object_cols])
label_X_valid[object_cols] = ordinal_encoder.transform(X_valid[object_cols])

print("MAE from Approach 2 (Ordinal Encoding):")
print(score_dataset(label_X_train, label_X_valid, y_train, y_valid))
```

MAE from Approach 2 (Ordinal Encoding):  
165936.40548390493

### 1. Drop Categorical (MAE: 175,703)

- Simply removes all categorical columns
- Performs worse because it throws away potentially useful information
- Like dropping 'Neighborhood' or 'House Style' which could affect house prices!

### 2. Ordinal Encoding (MAE: 165,936)

- Converts categories to numbers (e.g., 'Red'=0, 'Blue'=1, 'Green'=2)
- Performs better because it keeps the information
- But be careful! It assumes ordering between categories that might not make sense (like colors)

The ordinal encoding worked better here because:

1. Some categories might actually have a natural order (like quality ratings)
2. Even for unordered categories, keeping the information (even with random numbers) is better than throwing it away completely



### 3)One-Hot Encoding

```
from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[object_cols]))
OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[object_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

# Ensure all columns have string type
OH_X_train.columns = OH_X_train.columns.astype(str)
OH_X_valid.columns = OH_X_valid.columns.astype(str)

print("MAE from Approach 3 (One-Hot Encoding):")
print(score_dataset(OH_X_train, OH_X_valid, y_train, y_valid))
```

MAE from Approach 3 (One-Hot Encoding):  
166089.4893009678

Let's break down the results of all three approaches:

- 1.Drop Categorical: MAE = 175,703
- 2.Ordinal Encoding: MAE = 165,936
- 3.One-Hot Encoding: MAE = 166,089

Surprisingly, ordinal encoding performed slightly better than one-hot encoding here! This could be because:

- 1.Some categorical variables might actually have a natural order (like quality ratings)
- 2.Limited data: One-hot encoding creates more features, which might lead to overfitting with smaller datasets
- 3.Dataset specific: Some categorical features might have relationships that ordinal encoding captured by chance

Key Learning:

- Never assume one method is always best
- Test different approaches on your specific dataset
- Choose the method that gives best results in validation
- Consider the nature of your categorical variables (ordered vs unordered)

## Pipelines

**Pipelines** are a simple way to keep your data preprocessing and modeling code organized. Specifically, a pipeline bundles preprocessing and modeling steps so you can use the whole bundle as if it were a single step.

Many data scientists hack together models without pipelines, but pipelines have some important benefits.

Those include:

1. **Cleaner Code:** Accounting for data at each step of preprocessing can get messy. With a pipeline, you won't need to manually keep track of your training and validation data at each step.
2. **Fewer Bugs:** There are fewer opportunities to misapply a step or forget a preprocessing step.
3. **Easier to Productionize:** It can be surprisingly hard to transition a model from a prototype to something deployable at scale. We won't go into the many related concerns here, but pipelines can help.

Let me explain Pipelines with a real-world analogy:

Think of making a sandwich:

Without Pipeline (Messy way):

# Like making a sandwich the hard way: 1. Take bread out 2. Put bread back while finding butter 3. Take bread out again 4. Butter it 5. Put it back while finding cheese 6. Take bread out...

With Pipeline (Organized way):

python

Copy

```
sandwich_pipeline = [ ('get_bread', bread), ('add_butter', butter), ('add_cheese', cheese) ]
```



# Using pipelines

## 1. Cleaning Data

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy='constant')

# Preprocessing for categorical data
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

## 2. Define a Model

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=0)

from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model)
                             ])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

MAE: 160679.18917034855

Think of it like a car assembly line:

- Numbers go down one line (get cleaned)
- Categories go down another line (get cleaned and encoded)
- Both parts come together
- Final model makes predictions

The beauty is: You just put raw data in and get predictions out - all the messy middle steps happen automatically in the right order!



Tej Gorde

## Cross Validation

Imagine testing a recipe:

- **Regular Validation:** You let one person taste it and give feedback. Eg - Bob might hate spicy food, giving a biased review Or Bob might be sick that day, affecting his taste. You only get ONE perspective!
- **Cross-Validation:** You let 5 different people taste it and average their feedback. Eg – Bob tastes it (likes spicy food) Alice tastes it (prefers mild food) Charlie tastes it (loves all food) Danny tastes it (picky eater) Eve tastes it (neutral preference)

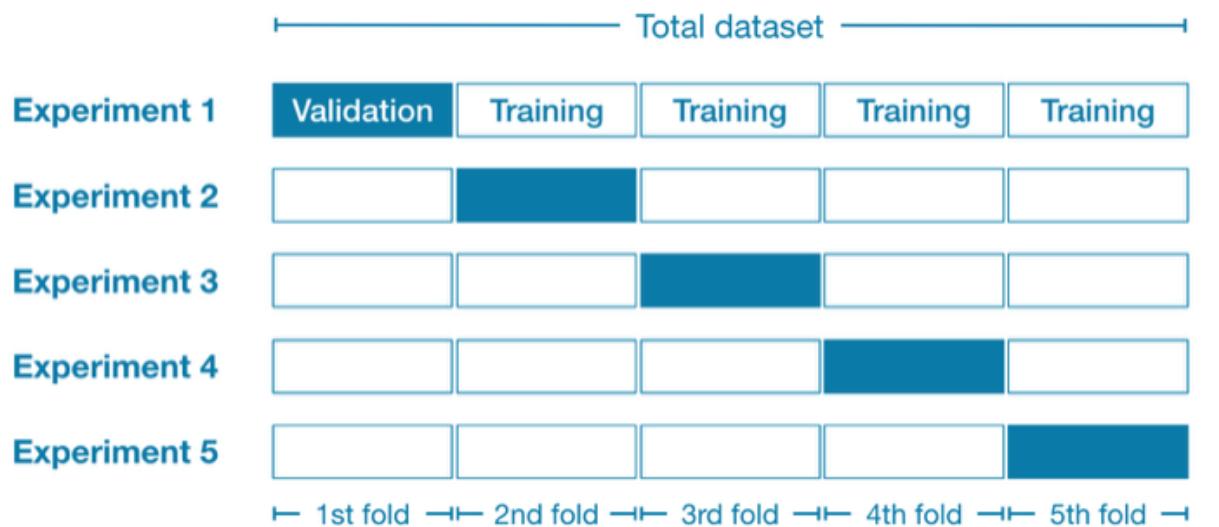
In ML terms: Instead of trusting how your model performs on one subset of data, you test it on multiple different subsets and average the results for a more reliable score!

### Benefits:

- More balanced feedback
- Less chance of bias
- More confidence in the recipe
- Better understanding of how different people like it

When to use it:

- ✓ Small datasets (need every data point to count)
- ✓ Important decisions (need more reliable scores)
- ✗ Large datasets (single validation is enough)
- ✗ Slow models (might take too long)



# How to use it

```
: from sklearn.model_selection import cross_val_score

# Multiply by -1 since sklearn calculates *negative* MAE
scores = -1 * cross_val_score(my_pipeline, X, y,
                             cv=5,
                             scoring='neg_mean_absolute_error')

print("MAE scores:\n", scores)
```

## What's happening:

```
MAE scores:
[301628.7893587 303164.4782723 287298.331666 236061.84754543
260383.45111427]
```

1. Data is split into 5 parts

2. Each fold acts as test set once:

1. Fold 1: Error = \$301,628
2. Fold 2: Error = \$303,164
3. Fold 3: Error = \$287,298
4. Fold 4: Error = \$236,061
5. Fold 5: Error = \$260,383

## Key insights:

- Wide range of errors (\$236K - \$303K) shows model performance varies based on which data it's tested on
- Average (\$277K) gives more reliable estimate than single validation set
- Negative used by sklearn (ignore this detail - just convention)

Think of it like having 5 different reviewers score your model instead of just one!

```
: print("Average MAE score (across experiments):")
print(scores.mean())
```

```
Average MAE score (across experiments):
277707.3795913405
```

## XGBoost

So far we have learned, how to make predictions with the random forest method, which achieves better performance than a single decision tree simply by averaging the predictions of many decision trees.

We refer to the random forest method as an "ensemble method". By definition, **ensemble methods** combine the predictions of several models (e.g., several trees, in the case of random forests).

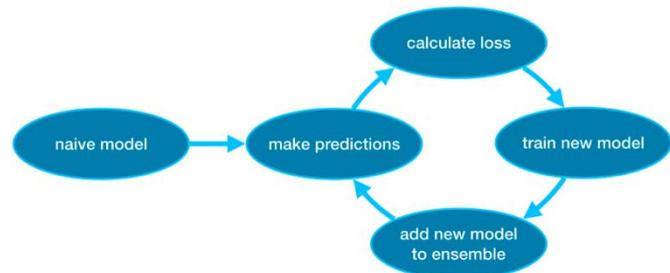
Next, we'll learn about another ensemble method called gradient boosting. **Gradient boosting** is a method that goes through cycles to iteratively add models into an ensemble.

Think of building a sports team:

**Random Forest** = Having 100 decent players and taking their average performance

**Gradient Boosting** = Starting with one player, then:

1. See what they're doing wrong
2. Add another player who's good at fixing those mistakes
3. Repeat!



```
from xgboost import XGBRegressor  
  
my_model = XGBRegressor()  
my_model.fit(X_train, y_train)
```

```
from sklearn.metrics import mean_absolute_error  
  
predictions = my_model.predict(X_valid)  
print("Mean Absolute Error: " + str(mean_absolute_error(predictions, y_valid)))
```

Mean Absolute Error: 241041.5160392121

## Parameter Tuning

**1.n\_estimators** (number of models) *Like deciding how many experts to hire: Hire 500 experts*

- Too few (100): Not enough expertise - Underfitting
- Too many (2000): Start copying each other's mistakes – Overfitting
- Sweet spot: Usually 100-1000

```
my_model = XGBRegressor(n_estimators=500)  
my_model.fit(X_train, y_train)
```

**2.early\_stopping\_rounds** (when to stop) - *Stop if no improvement for 5 rounds*

Think of it like: "Stop hiring if the team hasn't improved in 5 attempts"

```
my_model = XGBRegressor(n_estimators=500)  
my_model.fit(X_train, y_train,  
             early_stopping_rounds=5,  
             eval_set=[(X_valid, y_valid)],  
             verbose=False)
```

**3.learning\_rate** (how much to trust each new model) - *Trust each expert 5%*

- High (1.0): Trust each expert completely
- Low (0.01): Take their advice with a grain of salt
- Default (0.1): Balanced trust

```
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05)  
my_model.fit(X_train, y_train,  
             early_stopping_rounds=5,  
             eval_set=[(X_valid, y_valid)],  
             verbose=False)
```

**4.n\_jobs** (parallel processing) *Use 4 CPU cores*

Like having multiple interviewers working simultaneously to hire experts - doesn't make better choices, just faster!

```
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=4)  
my_model.fit(X_train, y_train,  
             early_stopping_rounds=5,  
             eval_set=[(X_valid, y_valid)],  
             verbose=False)
```

Remember: Lower learning\_rate needs more n\_estimators (like trusting each expert less means you need more experts)!

# Data Leakage

Think of data leakage like accidentally seeing the answers before a test:

## 1.What is Data Leakage?

- It's when your model "cheats" by using information it shouldn't have
- Like a student who scores 100% on practice tests (because they saw the answers)
- But fails the real test (because they didn't actually learn)

- Two Types:

- Target leakage** occurs when your predictors include data that will not be available at the time you make predictions.

```
# Like using tomorrow's weather to predict today's  
predict_rain(  
    temperature=20,  
    people_with_umbrellas=True) ❌ This is cheating!
```

Think of it like:

- Goal: Predict tomorrow's rain
- Bad Feature: "People carrying umbrellas"
- Why Bad?: People grab umbrellas AFTER knowing it's going to rain!

- Train-Test Contamination** is a different type of leak occurs when you aren't careful to distinguish training data from validation data.

```
# Like practicing with the actual test questions  
test_data = get_test_data()  
prepare_for_test(test_data) ❌ Shouldn't see test data yet!
```



## Remove Target Leakage

We will use a dataset about credit card applications and skip the basic data set-up code. Our (X) independent variables / Feature variables are to Reports, Age, Income, Expenditure..etc. Target variable is – Will the application be accepted “Yes/No”

Here is a summary of the data, which you can also find under the data tab:

•**card**: 1 if credit card application accepted, 0 if not

•**reports**: Number of major derogatory reports

•**age**: Age n years plus twelfths of a year

•**income**: Yearly income (divided by 10,000)

•**share**: Ratio of monthly credit card expenditure to yearly income

•**expenditure**: Average monthly credit card expenditure

•**owner**: 1 if owns home, 0 if rents

•**selfempl**: 1 if self-employed, 0 if not

•**dependents**: 1 + number of dependents

•**months**: Months living at current address

•**majorcards**: Number of major credit cards held

•**active**: Number of active credit accounts

A few variables look suspicious. For example, does **expenditure** mean expenditure on this card or on cards used before applying?

	reports	age	income	share	expenditure	owner	selfemp	dependents	months	majorcards	active
0	0	37.66667	4.5200	0.033270	124.983300	True	False	3	54	1	12
1	0	33.25000	2.4200	0.005217	9.854167	False	False	3	34	1	13
2	0	33.66667	4.5000	0.004156	15.000000	True	False	4	58	1	5
3	0	30.50000	2.5400	0.065214	137.869200	False	False	0	25	1	7
4	0	32.16667	9.7867	0.067051	546.503300	True	False	2	64	1	5

```

from sklearn.pipeline import make_pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

# Since there is no preprocessing, we don't need a pipeline (used anyway as best practice!)
my_pipeline = make_pipeline(RandomForestClassifier(n_estimators=100))
cv_scores = cross_val_score(my_pipeline, X, y,
                            cv=5,
                            scoring='accuracy')

print("Cross-validation accuracy: %f" % cv_scores.mean())

```

Cross-validation accuracy: 0.981052

### ⚠️ Why This is a Red Flag:

- 1.ALL rejected applicants had \$0 expenditure
- 2.Almost ALL approved applicants had some expenditure
- 3.This is too perfect to be real!

With experience, you'll find that it's very rare to find models that are accurate 98% of the time. It happens, but it's uncommon enough that we should inspect the data more closely for target leakage.

```

: expenditures_cardholders = X.expenditure[y]
expenditures_noncardholders = X.expenditure[~y]

print('Fraction of those who did not receive a card and had no expenditures: %.2f' \
      %((expenditures_noncardholders == 0).mean()))
print('Fraction of those who received a card and had no expenditures: %.2f' \
      %((expenditures_cardholders == 0).mean()))

```

**Share column includes expenditure too. So, it's best to drop it out --- share = expenditure / income**

```

# Drop leaky predictors from dataset
potential_leaks = ['expenditure', 'share', 'active', 'majorcards']
X2 = X.drop(potential_leaks, axis=1)

# Evaluate the model with leaky predictors removed
cv_scores = cross_val_score(my_pipeline, X2, y,
                            cv=5,
                            scoring='accuracy')

print("Cross-val accuracy: %f" % cv_scores.mean())

```

Cross-val accuracy: 0.830919

This accuracy is quite a bit lower, which might be disappointing. However, we can expect it to be right about 80% of the time when used on new applications, whereas the leaky model would likely do much worse than that (in spite of its higher apparent score in cross-validation).

Data leakage can be multi-million dollar mistake in many data science applications. Careful separation of training and validation data can prevent train-test contamination, and pipelines can help implement this separation. Likewise, a combination of caution, common sense, and data exploration can help identify target leakage.