Intermediate Machine Learning Home Page (https://www.kaggle.com/learn/intermediate-machine-learning)

This exercise involves you writing code, and we check it automatically to tell you if it's right. We're having a temporary problem with out checking infrastructure, causing a bar that says `None` *in some cases when you have the right answer. We're sorry. We're fixing it. In the meantime, if you see a bar saying* `None` *that means you've done something good.*

By encoding **categorical variables**, you'll obtain your best results thus far!

Setup

The questions below will give you feedback on your work. Run the following cell to set up the feedback system.

In [1]:
```
# Set up code checking
import os
if not os.path.exists("../input/train.csv"):
    os.symlink("../input/home-data-for-ml-course/train.csv", "../inpu
t/train.csv")
    os.symlink("../input/home-data-for-ml-course/test.csv", "../input/
test.csv")
from learntools.core import binder
binder.bind(globals())
from learntools.ml_intermediate.ex3 import *
print("Setup Complete")
```

```
Setup Complete
```

In this exercise, you will work with data from the Housing Prices Competition for Kaggle Learn Users (https://www.kaggle.com/c/home-data-for-ml-course).

Run the next code cell without changes to load the training and validation sets in `X_train`, `X_valid`, `y_train`, and `y_valid`. The test set is loaded in `X_test`.

In [2]:
```
import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data
X = pd.read_csv('../input/train.csv', index_col='Id')
X_test = pd.read_csv('../input/test.csv', index_col='Id')
```

```
# Remove rows with missing target, separate target from predictors
X.dropna(axis=0, subset=['SalePrice'], inplace=True)
y = X.SalePrice
X.drop(['SalePrice'], axis=1, inplace=True)

# To keep things simple, we'll drop columns with missing values
cols_with_missing = [col for col in X.columns if X[col].isnull().any
()]
X.drop(cols_with_missing, axis=1, inplace=True)
X_test.drop(cols_with_missing, axis=1, inplace=True)

# Break off validation set from training data
X_train, X_valid, y_train, y_valid = train_test_split(X, y,
                                                      train_size=0.8,
test_size=0.2,
                                                      random_state=0)
```

Use the next code cell to print the first five rows of the data.

In [3]:

```
X_train.head()
```

Out[3]:

	MSSubClass	MSZoning	LotArea	Street	LotShape	LandContour	Utilities	LotConfig	Land
Id									
619	20	RL	11694	Pave	Reg	Lvl	AllPub	Inside	Gtl
871	20	RL	6600	Pave	Reg	Lvl	AllPub	Inside	Gtl
93	30	RL	13360	Pave	IR1	HLS	AllPub	Inside	Gtl
818	20	RL	13265	Pave	IR1	Lvl	AllPub	CulDSac	Gtl
303	20	RL	13704	Pave	IR1	Lvl	AllPub	Corner	Gtl

5 rows × 60 columns

Notice that the dataset contains both numerical and categorical variables. You'll need to encode the categorical data before training a model.

To compare different models, you'll use the same `score_dataset()` function from the tutorial. This function reports the mean absolute error (https://en.wikipedia.org/wiki/Mean_absolute_error) (MAE) from a random forest model.

In [4]:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=100, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Step 1: Drop columns with categorical data

You'll get started with the most straightforward approach. Use the code cell below to preprocess the data in `X_train` and `X_valid` to remove columns with categorical data. Set the preprocessed DataFrames to `drop_X_train` and `drop_X_valid`, respectively.

In [5]:
```
# Fill in the lines below: drop columns in training and validation data

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

# Check your answers
step_1.check()
```

Correct

In [6]:
```
# Lines below will give you a hint or solution code
#step_1.hint()
#step_1.solution()
```

Run the next code cell to get the MAE for this approach.

In [7]:
```
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):
17837.82570776256
```

Step 2: Label encoding

Before jumping into label encoding, we'll investigate the dataset. Specifically, we'll look at the `'Condition2'` column. The code cell below prints the unique entries in both the training and validation sets.

In [8]:
```
print("Unique values in 'Condition2' column in training data:", X_train['Condition2'].unique())
print("\nUnique values in 'Condition2' column in validation data:", X_valid['Condition2'].unique())
```

```
Unique values in 'Condition2' column in training data: ['Norm' 'PosA'
'Feedr' 'PosN' 'Artery' 'RRAe']

Unique values in 'Condition2' column in validation data: ['Norm' 'RRA
n' 'RRNn' 'Artery' 'Feedr' 'PosN']
```

If you now write code to:

- fit a label encoder to the training data, and then
- use it to transform both the training and validation data,

you'll get an error. Can you see why this is the case? (*You'll need to use the above output to answer this question.*)

In [9]:
```
#step_2.a.hint()
```

In [10]:
```
#step_2.a.solution()
```

This is a common problem that you'll encounter with real-world data, and there are many approaches to fixing this issue. For instance, you can write a custom label encoder to deal with new categories. The simplest approach, however, is to drop the problematic categorical columns.

Run the code cell below to save the problematic columns to a Python list `bad_label_cols`. Likewise, columns that can be safely label encoded are stored in `good_label_cols`.

In [11]:
```
# All categorical columns
object_cols = [col for col in X_train.columns if X_train[col].dtype ==
"object"]

# Columns that can be safely label encoded
good_label_cols = [col for col in object_cols if
                   set(X_train[col]) == set(X_valid[col])]

# Problematic columns that will be dropped from the dataset
bad_label_cols = list(set(object_cols)-set(good_label_cols))

print('Categorical columns that will be label encoded:', good_label_co
ls)
print('\nCategorical columns that will be dropped from the dataset:',
bad_label_cols)
```

```
Categorical columns that will be label encoded: ['MSZoning', 'Street',
'LotShape', 'LandContour', 'LotConfig', 'BldgType', 'HouseStyle', 'Ext
erQual', 'CentralAir', 'KitchenQual', 'PavedDrive', 'SaleCondition']

Categorical columns that will be dropped from the dataset: ['Exterior1
st', 'Foundation', 'LandSlope', 'RoofStyle', 'Condition1', 'Heating',
'Exterior2nd', 'Functional', 'Utilities', 'ExterCond', 'SaleType', 'He
atingQC', 'Neighborhood', 'RoofMatl', 'Condition2']
```

Use the next code cell to label encode the data in `X_train` and `X_valid`. Set the preprocessed DataFrames to `label_X_train` and `label_X_valid`, respectively.

- We have provided code below to drop the categorical columns in `bad_label_cols` from the dataset.
- You should label encode the categorical columns in `good_label_cols`.

In [12]:
```
from sklearn.preprocessing import LabelEncoder

# Drop categorical columns that will not be encoded
label_X_train = X_train.drop(bad_label_cols, axis=1)
label_X_valid = X_valid.drop(bad_label_cols, axis=1)

# Apply label encoder
enc = LabelEncoder()
for col in good_label_cols:
    label_X_train[col] = enc.fit_transform(X_train[col])
    label_X_valid[col] = enc.transform(X_valid[col])
# Check your answer
step_2.b.check()
```

Correct

In [13]:
```
# Lines below will give you a hint or solution code
#step_2.b.hint()
#step_2.b.solution()
```

Run the next code cell to get the MAE for this approach.

In [14]:
```
print("MAE from Approach 2 (Label Encoding):")
print(score_dataset(label_X_train, label_X_valid, y_train, y_valid))
```

```
MAE from Approach 2 (Label Encoding):
17575.291883561644
```

Step 3: Investigating cardinality

So far, you've tried two different approaches to dealing with categorical variables. And, you've seen that encoding categorical data yields better results than removing columns from the dataset.

Soon, you'll try one-hot encoding. Before then, there's one additional topic we need to cover. Begin by running the next code cell without changes.

In [15]:

```
# Get number of unique entries in each column with categorical data
object_nunique = list(map(lambda col: X_train[col].nunique(), object_cols))
d = dict(zip(object_cols, object_nunique))

# Print number of unique entries by column, in ascending order
sorted(d.items(), key=lambda x: x[1])
```

Out[15]:

```
[('Street', 2),
 ('Utilities', 2),
 ('CentralAir', 2),
 ('LandSlope', 3),
 ('PavedDrive', 3),
 ('LotShape', 4),
 ('LandContour', 4),
 ('ExterQual', 4),
 ('KitchenQual', 4),
 ('MSZoning', 5),
 ('LotConfig', 5),
 ('BldgType', 5),
 ('ExterCond', 5),
 ('HeatingQC', 5),
 ('Condition2', 6),
 ('RoofStyle', 6),
 ('Foundation', 6),
 ('Heating', 6),
 ('Functional', 6),
 ('SaleCondition', 6),
 ('RoofMatl', 7),
 ('HouseStyle', 8),
 ('Condition1', 9),
 ('SaleType', 9),
 ('Exterior1st', 15),
 ('Exterior2nd', 16),
 ('Neighborhood', 25)]
```

The output above shows, for each column with categorical data, the number of unique values in the column. For instance, the `'Street'` column in the training data has two unique values: `'Grvl'` and `'Pave'`, corresponding to a gravel road and a paved road, respectively.

We refer to the number of unique entries of a categorical variable as the **cardinality** of that categorical variable. For instance, the `'Street'` variable has cardinality 2.

Use the output above to answer the questions below.

In [16]:

```
# Fill in the line below: How many categorical variables in the training data
# have cardinality greater than 10?
high_cardinality_numcols = 3

# Fill in the line below: How many columns are needed to one-hot encode the
# 'Neighborhood' variable in the training data?
```

```
num_cols_neighborhood = 25

# Check your answers
step_3.a.check()
```

Correct

In [17]:
```
# Lines below will give you a hint or solution code
#step_3.a.hint()
#step_3.a.solution()
```

For large datasets with many rows, one-hot encoding can greatly expand the size of the dataset. For this reason, we typically will only one-hot encode columns with relatively low cardinality. Then, high cardinality columns can either be dropped from the dataset, or we can use label encoding.

As an example, consider a dataset with 10,000 rows, and containing one categorical column with 100 unique entries.

- If this column is replaced with the corresponding one-hot encoding, how many entries are added to the dataset?
- If we instead replace the column with the label encoding, how many entries are added?

Use your answers to fill in the lines below.

In [18]:
```
# Fill in the line below: How many entries are added to the dataset by
# replacing the column with a one-hot encoding?
OH_entries_added = 10000*100-10000

# Fill in the line below: How many entries are added to the dataset by
# replacing the column with a label encoding?
label_entries_added = 0

# Check your answers
step_3.b.check()
```

Correct

In [19]:
```
# Lines below will give you a hint or solution code
#step_3.b.hint()
#step_3.b.solution()
```

Step 4: One-hot encoding

In this step, you'll experiment with one-hot encoding. But, instead of encoding all of the categorical variables in the dataset, you'll only create a one-hot encoding for columns with cardinality less than 10.

Run the code cell below without changes to set `low_cardinality_cols` to a Python list containing the columns that will be one-hot encoded. Likewise, `high_cardinality_cols` contains a list of categorical columns that will be dropped from the dataset.

In [20]:
```
# Columns that will be one-hot encoded
low_cardinality_cols = [col for col in object_cols if X_train[col].nun
ique() < 10]

# Columns that will be dropped from the dataset
high_cardinality_cols = list(set(object_cols)-set(low_cardinality_cols
))

print('Categorical columns that will be one-hot encoded:', low_cardina
lity_cols)
```

Exercise: Categorical Variables

Python notebook using data from Housing Prices Competition for Kaggle Learn Users · 1 views · 1m ago · Edit tags

0 | Access | Edit

```
Categorical columns that will be one-hot encoded: ['MSZoning', 'Stree
t', 'LotShape', 'LandContour', 'Utilities', 'LotConfig', 'LandSlope',
'Condition1', 'Condition2', 'BldgType', 'HouseStyle', 'RoofStyle', 'Ro
ofMatl', 'ExterQual', 'ExterCond', 'Foundation', 'Heating', 'HeatingQ
C', 'CentralAir', 'KitchenQual', 'Functional', 'PavedDrive', 'SaleTyp
e', 'SaleCondition']

Categorical columns that will be dropped from the dataset: ['Exterior1
st', 'Exterior2nd', 'Neighborhood']
```

Ver
1

fork
Cat

Use the next code cell to one-hot encode the data in `X_train` and `X_valid`. Set the preprocessed DataFrames to `OH_X_train` and `OH_X_valid`, respectively.

- The full list of categorical columns in the dataset can be found in the Python list `object_cols`.
- You should only one-hot encode the categorical columns in `low_cardinality_cols`. All other categorical columns should be dropped from the dataset.

In [21]:
```
from sklearn.preprocessing import OneHotEncoder

oh_enc = OneHotEncoder(handle_unknown='ignore', sparse=False)

OH_train = pd.DataFrame(oh_enc.fit_transform(X_train[low_cardinality_c
ols]))
OH_valid = pd.DataFrame(oh_enc.transform(X_valid[low_cardinality_cols
]))

OH_train.index = X_train.index
OH_valid.index = X_valid.index
#print(OH_train.head(3))

num_X_train = X_train.drop(object_cols, axis =1)
num_X_valid = X_valid.drop(object_cols, axis=1)
```

```
OH_X_train = pd.concat([num_X_train, OH_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_valid], axis=1)

# Check your answer
step_4.check()
```

Correct

In [22]:

```
# Lines below will give you a hint or solution code
#step_4.hint()
#step_4.solution()
```

Run the next code cell to get the MAE for this approach.


```
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):
17525.345719178084
```

Step 5: Generate test predictions and submit your results

Data

Data Sources

- Housing Prices Competition for Kaggle Learn Users
  - sample_submission.csv 1459 x 2
  - test.csv 1459 x 80
  - train.csv 1460 x 81

Housing Prices Competition for Kaggle Learn Users

Apply what you learned in the Machine Learning course on Kaggle Learn alongside others in the course.

Last Updated: a year ago

About this Competition

File descriptions

- **train.csv** - the training set
- **test.csv** - the test set
- **data_description.txt** - full description of each column, originally prepared by Dean De Cock but lightly edited to match the column names used here
- **sample_submission.csv** - a benchmark submission from a linear regression on year and month of sale, lot square footage, and number of bedrooms

Data fields

Here's a brief version of what you'll find in the data description file.

- **SalePrice** - the property's sale price in dollars. This is the target variable that you're trying to predict.
- **MSSubClass**: The building class
- **MSZoning**: The general zoning classification
- **LotFrontage**: Linear feet of street connected to property
- **LotArea**: Lot size in square feet
- **Street**: Type of road access
- **Alley**: Type of alley access
- **LotShape**: General shape of property

Comments (0)

Click here to comment...