Intermediate Machine Learning

Based on the Kaggle Tutorial ["Intermediate Machine Learning"](#) by Alexis Cook.

1. Introduction

Goals:

- tackle data types often found in real-world datasets (missing values, categorical variables),
- design pipelines to improve the quality of your machine learning code,
- use advanced techniques for model validation (cross-validation),
- build state-of-the-art models that are widely used to win Kaggle competitions (XGBoost), and
- avoid common and important data science mistakes (leakage).

Competition submission steps reiterated:

1. save version with the `Save and Run all` version enabled.
2. once that is done, click on the number right next to the `Save Version` button.
3. This pulls up a list of versions on the right of the screen. Click on the ellipsis (...) to the right of the most recent version, and select Open in Viewer.
4. This brings you into view mode of the same page. You will need to scroll down to get back to these instructions.
5. Click on the Output tab on the right of the screen. Then, click on the file you would like to submit, and click on the blue Submit button to submit your results to the leaderboard.

2. Missing Values

There are many reasons why there would be missing values, some examples are: - The size of the 3rd bedroom for a 2 bedroom house - One survey respondant declining to share his income - etc.

Two ways to deal with this:

Method	notes
Drop columns with missing values	you will be ignoring lots of potentially important information.
Imputation	Fill in the missing blank with the average for that row. Maybe not exactly *the* best value for that spot, but this causes the model to be a lot more accurate.
Imputation Extention	add a column that notes if a given column had missing values that were imputed, might be more accurate than your average imputation (pun?).
IMPORTANT!	There is no universal rule on what is better in any case, and you certainly can't guess it without reading a huge part of the data manually. So just try 'em all and make an educated guess.

Method	notes
For more examples and techniques:	Have a look at this article

Tips to choose:

- A lot of missing data (think more than 50%) in a given column -- drop it.
- If not, try both the imputation methods and pick the better one.

Extention to imputation

- Some times, the imputed values could be systematically above or below their actual values, or really just be correctly and uniquely wrong (think accuracy v. precision)

- So a good idea would be to add another coloumn along the lines of "`bed 3 missing`" with a true or false value to let the model know that the value was artifically generated and is therefore different from the original model *someway*.

    - This may or may not help depending on your situation.
    - So be sure to try using it and then evaluate if it turns out to be better than plain old imputation.

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
NaN	2.0

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

Good starting point to optimize models:

- make an accuracy function such as this one.

```python
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=10, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Method 1: Drop columns

- Be sure to drop the same columns from the training and validation data sets.

```
# 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)
```

Method 2: Imputation

- use `sklearn.impute.SimpleImputer` class.
- Many different method have been researched for imputation, but once put into modern machine learning models, it doesn't make much of a difference to using average values.

```
from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
#### MUST fit transform training data
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
###### MUST only "regular" tranform validation data
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
```

QQ: I noticed that you called `fit_transform()` for the training dataset and `transform()` for the validation dataset. What gives?
Check out this answer from stack exachange.

> To center the data (make it have zero mean and unit standard error), you subtract the mean and then divide the result by the standard deviation:
>
> x'=x−μσ
>
> You do that on the training set of data. But then you have to apply the same transformation to your testing set (e.g. in cross-validation), or to newly obtained examples before forecast. But you have to use the exact same two parameters μ and σ (values) that you used for centering the training set.
>
> Hence, every sklearn's transform's fit() just calculates the parameters (e.g. μ and σ in case of StandardScaler) and saves them as an internal object's state. Afterwards, you can call its transform() method to apply the transformation to any particular set of examples.
>
> fit_transform() joins these two steps and is used for the initial fitting of parameters on the training set x, while also returning the transformed x'.

> Internally, the transformer object just calls first fit() and then transform() on the same data.

Whhhaaaaaattttt?

> Given that thre are so few missing values in the dataset, we'd expect imputation to perform better than dropping columns entirely. However, we see that dropping columns performs slightly better! While this can probably partially be attributed to noise in the dataset, another potential explanation is that the imputation method is not a great match to this dataset. That is, maybe instead of filling in the mean value, it makes more sense to set every missing value to a value of 0, to fill in the most frequently encountered value, or to use some other method. For instance, consider the GarageYrBlt column (which indicates the year that the garage was built). It's likely that in some cases, a missing value could indicate a house that does not have a garage. Does it make more sense to fill in the median value along each column in this case? Or could we get better results by filling in the minimum value along each column? It's not quite clear what's best in this case, but perhaps we can rule out some options immediately - for instance, setting missing values in this column to 0 is likely to yield horrible results! TLDR:

- There is no universal rule on what is better in which type of dataset, so you just gotta try 'em all.

Method 3: Extention 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))
```

REGARDLESS OF THE METHOD USED FOR REMOVING MISSING VALUES:

- To transform the actual dataset on which you will use your model to predict the answers for, use the following tool.

○ *Do not try to do this yourself manually... like how would you even do that? The final dataset will have different details missing from different columns than the dataset you used for training if you used the drop coloumns method for example.*

```
# NOTE: this only works for imputation... I will add a method for dropping
the coloumns later.
# Use the following method to transform the final dataset so that it alligns
with the training dataset.
final_X_test = pd.DataFrame(final_imputer.transform(X_test))
```

3. Categorial Variables

- some variables only have a limited number of possible values
  ○ such as the brands of the cars owned by a set of people--it really only can be `['ford', 'tesla', 'honda', ... , 'and so on']`.
- these variables are called ***Categorial Variables***.

There are 3 main approaches to deal with categorical variables:

Sno | Method | Usefulness | description --- | ---- | ---- 1 | Drop Categorical Variables | Easiest [and worst] way | just remove them off the dataset. 2 | Label Encoding | Usually the 2nd best, but maybe better | Assign each value to an unique integer. 3 | One-Hot Encoding | Usaually the best, use only with < 15 different possibilities | Create a true/false column for each possibility to denote if the item belongs to that class or not.

Method 1: Drop Categorial Variables

- Use this method IFF those variables have no useful information.

```
########## Example code to do this
# Get list of categorical variables
s = (X_train.dtypes == 'object') # get a dictionary with each column name followed
by a boolean that says if the data
                              # type that each column stores is "object" or
not.
object_cols = list(s[s].index) #

print("Categorical variables:")
print(object_cols)

# drop 'em
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))
```

Method 2: Label Encoding

Breakfast
Every day
Never
Rarely
Most days
Never

➡️

Breakfast
3
0
1
2
0

- Assumes an ordering/relative worth to each of these categories.

    ○ In this case, `Never (0) < Rarely (1) < Most Days (2) < Every Day (3)`

- If your categories can't really be ranked so clearly, they are called `ordinal variables`.

    ○ This method works well enough for these variables in tree based models
    ○ so be aware of this when looking for baises in data when using other types of models.

- **Error Source:** The validation dataset might have unique values not seen before in the training dataset, so the model will crash.

    ○ Easiest way to solve it: Remove those problematic columns.

    ○
    ```python
    # 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_valid[col]).issubset(set(X_train[col]))]

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

Implementation:

- Scikit-learn has a `LabelEncoder` class that can be used to get label encodings. We loop over the categorical variables and apply the label encoder separately to each column.
- In this approach, we assign random unique integers for each possiblity of the categorical variables for a column, which is a common (and easier) approach.
- To get a better model, you should manually assign better-informed numerical variables for each of these classes.

```
# Get a list of columns that store categorical variables
s = (X_train.dtypes == 'object')
object_cols = list(s[s].index)

# import the LabelEncoder class
from sklearn.preprocessing import LabelEncoder
# make a copy of the training and validation datasets
label_X_train = X_train.copy()
label_X_valid = X_valid.copy()
# Create a LabelEncoder object
encoder = LabelEncoder()
# Apply the label encoder to each column with categorical data
for col in object_cols:
    label_X_train[col] = label_encoder.fit_transform(X_train[col]) # fit and
transform for training dataset
    label_X_valid[col] = label_encoder.transform(X_valid[col]) # plain old
transform for validation dataset

#### At this point, feel free to train the model and see how well it works
```

Method 3: One-Hot encoding

Color
Red
Red
Yellow
Green
Yellow

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

- Create new columns to indicate if the given entry belongs to a given class in the form of true/false values.
- Use only if the number of possible values for the given categorical variable is less than 15 elements.
    - does not work well if the number of possibilites are greater than that 😣
- Works better than `label encoding` for nominal variables--i.e. variables whose possiblities do not have a clear ordering as per human logic.
- *Cardinality of a categorical variable:* The number of unique values that exist in the respective column.
- To save storage space, we only use `one-hot` encoding when the number of possible values for a given column is small
    - If you have a column with 100 unique values, and have a total of 10,000 rows, then you will add `99*10,000` extra data entries!!!
    - Drop or label-encode other columns with a large number of possibilites.

Implementation:

- Use the class `OneHotEncoder` from `scikit-learn`.
- A lot of parameters can be used to customize the behaviour of the class, a few examples are:

- - handle_unknown = 'ignore' --> If the validation dataset contains classes not seen before in the training dataset, ignore that data entry.
    - sparse = False --> return the encoded columns as a numpy array instead of a sparse matrix.
- supply only the categorical columns that we want to be one-hot encoded.
    - For instance, to encode the training data, we supply X_train[object_cols].
    - where object_cols is a list of all columns with categorical variables.

```python
from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data, store them in their
own columns
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)

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

4. Pipelines

QQ	Answer
What?	A simple way to keep code organized - bundle preprocessing and modeling into one single step.
Why?	Cleaner code, Fewer bugs, Easier to deploy to production, More options for model validation (example: Cross-Validation)

Three Steps to create a pipeline

Step 1: Define Preprocessing Steps

- Bundle all the preprocessing steps using a sklearn.compute.ColumnTransfer

```python
# Import the ColumnTransfer class
from sklearn.compute import ColumnTransfer
# Import the Pipeline class because that is just how you combine multiple steps...
from sklearn.pipeline import Pipeline
```

```python
# Import the libraries needed for the actual preprocessing
from sklearn.impute import SimpleImputer # imputer to fill in missing values using
the average of each column
from sklearn.preprocessing import OneHotEncoder # encoder to convert Categorical
variables into numbers

# Preprocessing for numerical data -- Replace missing values with column averages
numerical_transformer = SimpleImputer(strategy='constant')

# ----------------------- pipeline syntax example -----------------
# Preprocessing for categorical data -- Replace missing values with the most
frequently occuring value, and then OneHotEncode it.
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)
    ])
```

Step 2: Define the Model

- Nothing special here

```python
from sklearn.ensemble import RandomForestRegressor

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

Step 3: Create and Evaluate the Pipeline

- Just like we did for preprocessing, we will now create a pipline for the whole thing
- Notice:
    - Only a single line of code needed to do *everything* - imputation, OneHotEncoding, model training, etc..
    - Now, just directly pass in the unprocessed dataset to the pipeline line in the `.predict(data_set)` command, and it takes care of all the preprocessing for you!
- Pipeline syntax:
    - `Pipeline(steps=list(tuples))`, where `tuples = 'label', method`.

```python
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)
```

5. Cross-Validation

Commands Bitbucket

```
########### Drop columns with missing values.
# Run this once you have found the names of all the coulumns with missing values.
reduced_X_train = X_train.drop(cols_with_missing, axis=1)

########## Use a SimpleImputer to replace missing values with their averages in
the column.
# Import the library
from sklearn.impute import SimpleImputer
# Create the imputer object.
my_imputer = SimpleImputer()
# call it on both the training and testing datasets.
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

########## To implement the extention to imputation method, do some along the line
of:
# 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()

######### Get a count of missing values by coloumn
# Number of missing values in each column of training data
missing_val_count_by_column = (X_train.isnull().sum())
### side note: WOW, python can do this?? the array access works in mysterious ways
in python--at least to my c++ mind.
print(missing_val_count_by_column[missing_val_count_by_column > 0])

######## Use the following method to transform the final dataset so that it
```

```
alligns with the training dataset.
final_X_test = pd.DataFrame(final_imputer.transform(X_test))

######## Drop Categorical variables
drop_X_train = X_train.select_dtypes(exclude=['object'])
drop_X_valid = X_valid.select_dtypes(exclude=['object'])

####### 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])

####### Subtract two lists
bad_label_cols = list(set(object_cols)-set(good_label_cols))
```