CODING

Clarinda

August 2024

1 Introduction

```
    import pandas as pd
    import numpy as np

    import matplotlib.pyplot as plt


# save filepath to variable for easier access
melbourne_file_path = '../input/melbourne-housing-snapshot/melb_data.csv'
# read the data and store data in DataFrame titled melbourne_data
melbourne_data = pd.read_csv(melbourne_file_path)
# print a summary of the data in Melbourne data
melbourne_data.describe()

average_lot_size = df['lot_size'].mean()

print(df.columns.tolist())

df.shape gives row and columns

df.describe()
df.info()

so you dont have to

# Calculate the number of missing values per column
missing_per_column = df.isna().sum()

# Print the number of missing values for each column
print("Missing values per column:")
for column, missing_count in missing_per_column.items():
    print(f"{column}: {missing_count}")

df_cleaned = df.dropna()
```

```
# Compute the median of the specific column (e.g., 'A')
median_A = df['A'].median()

# Impute missing values in column 'A' with its median
df['A'] = df['A'].fillna(median_A)

print(df['column_name'].unique())


 Example DataFrame
df = pd.DataFrame({
    'feature': [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5]
})

# Plot histogram for the 'feature' column
df['feature'].hist(bins=10, edgecolor='black')

# Add title and labels
plt.title('Histogram of Feature')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Plot histogram for the 'feature' column
plt.hist(df['feature'], bins=10, edgecolor='black')

# Add title and labels
plt.title('Histogram of Feature')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Show the plot
plt.show()

df['feature'].describe()



# Verify the presence of NaN values
print("\nCount of NaN values in 'feature':")
print(df['feature'].isna().sum())

# Drop rows where 'feature' column has NaN values
df_cleaned = df.dropna(subset=['feature'])
```

```
Alternatively

df_c = df.dropna(axis=0)


axis=0: This specifies that you want to drop entire rows that contain
any NaNs. (Alternatively, axis=1 would drop columns with any NaNs)
```

1.1 Machine Learning Model

```
df_c = df.dropna(axis=0)


axis=0: This specifies that you want to drop entire rows that contain
any NaNs. (Alternatively, axis=1 would drop columns with any NaNs)


Select target RV

y = melbourne_data.Price

choose feautures

features_names = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']

X = df_c_data[features_names]


Fit model

from sklearn.tree import DecisionTreeRegressor

# Define model. Specify a number for random_state to ensure same results
each run
melbourne_model = DecisionTreeRegressor(random_state=1)

# Fit model
melbourne_model.fit(X, y)

print(melbourne_model.predict(X.head()))


from sklearn.metrics import mean_absolute_error
```

```
predicted_home_prices = melbourne_model.predict(X)
MAE_training_error=mean_absolute_error(y, predicted_home_prices)

from sklearn.model_selection import train_test_split

# split data into training and validation data, for both features and
target
# The split is based on a random number generator. Supplying a numeric
value to
# the random_state argument guarantees we get the same split every
time we
# run this script.
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state
= 0)
# Define model
melbourne_model = DecisionTreeRegressor()
# Fit model
melbourne_model.fit(train_X, train_y)

# get predicted prices on validation data
val_predictions = melbourne_model.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))



#Silly cross validaation

def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)

def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)

#choose model with lowers MAE on validation set
```

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

forest_model = RandomForestRegressor(random_state=1)
forest_model.fit(train_X, train_y)
melb_preds = forest_model.predict(val_X)
print(mean_absolute_error(val_y, melb_preds))
```

2 Machine Learning Module 2

2.1 Missing valuues

```
df_c = df.dropna(axis=0)
```

axis=0: This specifies that you want to drop entire rows that contain any NaNs. (Alternatively, axis=1 would drop columns with any NaNs)

```
    # 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)

from sklearn.impute import SimpleImputer

# 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))
```

```
X_train_plus[col + '_was_missing']:

Creates a new column in the DataFrame X_train_plus. The name of this
new column is derived from the original column name (col) with '_was_missing'
appended to it.
X_train_plus[col].isnull():

X_train_plus[col]: Selects the original column from the DataFrame.
.isnull(): Returns a boolean Series where each entry is True if the
corresponding entry in the original column is NaN (missing), and False
otherwise.
Summary:
For each column with missing values in cols_with_missing, this code
creates a new boolean column indicating whether each entry in the original
column was missing (True) or not (False).
```

```
Here's a breakdown of what each line does:

my_imputer = SimpleImputer():

This line creates an instance of the SimpleImputer class from the sklearn.impute
module.
By default, this imputer fills missing values with the mean of each
column, though this behavior can be changed by passing arguments like
strategy='median', strategy='most_frequent', etc.
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus)):
```

```
The fit_transform() method does two things:
Fit: It computes the imputation values (e.g., mean, median) from the
X_train_plus dataset.
Transform: It fills in the missing values in X_train_plus using the
computed imputation values.
The result is a NumPy array, which is then converted into a pandas
DataFrame (as imputation removes the original column labels).
imputed_X_train_plus is the imputed version of X_train_plus, with missing
values replaced.

my_imputer = SimpleImputer(strategy='median')
```

```
# Columns to impute
columns_to_impute = ['A', 'B']

# Create the imputer
my_imputer = SimpleImputer(strategy='mean')

# Impute the selected columns and assign them back directly
X_train_plus[columns_to_impute] = my_imputer.fit_transform(X_train_plus[columns_to_impute])
```

```
The double brackets ([['Length']]) are used to select a DataFrame with
a single column rather than a Series. In pandas, there is a difference
between selecting a column as a Series and selecting it as a DataFrame:

Single brackets (['Length']) select a column as a Series.
Double brackets ([['Length']]) select a column as a DataFrame.
Why the double brackets are required:
The SimpleImputer.fit_transform() method expects a 2D array-like structure
```

```
(such as a DataFrame) as input, even if it only has one column.

male_data['Length'] would give a 1D Series, which is not valid input
for fit_transform().
male_data[['Length']] gives a 2D DataFrame with one column, which matches
the expected input format.



# Impute missing 'Length' based on 'Gender'
data['Length'] = data.groupby('Gender')['Length'].transform(lambda
x: x.fillna(x.median() if x.name == 'Male' else x.mean()))
```

2.2 Categorical variables

```
# Print the data type of a specific column
print(X_train['column_name'].dtype)


    Ordinal encoding assigns each unique value to a different integer.

Ordinal encoding assigns each unique value to a different integer.
categorical variables that have a clear ordering in the values, but
we refer to
those that do as ordinal variables

One-hot encoding creates new columns indicating the presence (or absence)
of each possible value in the original data.
We refer to categorical variables without
an intrinsic ranking as nominal variables.

One-hot encoding generally does not perform well if the categorical
variable takes
on a large number of values (i.e., you generally won't use it for variables
taking more than 15 different values).



# Get list of categorical variables
s = (X_train.dtypes == 'object')
object_cols = list(s[s].index)

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

```
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])
```

```
We use the OneHotEncoder class from scikit-learn to get one-hot encodings.
There are a number of parameters that can be used to customize its
behavior.

We set handle_unknown='ignore' to avoid errors when the validation
data contains classes that aren't represented in the training data,
and
setting sparse=False ensures that the encoded columns are returned
as a numpy array (instead of a sparse matrix).
To use the encoder, we 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]. (object_cols in the code cell below
is a list of the column names with categorical data, and so X_train[object_cols]
contains all of the categorical data in the training set.)

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)
```

2.3 Pipelines

```
First we do standard raeding data, test split etc


# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient
but arbitrary)
categorical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].nunique()
< 10 and
                        X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dtype
in ['int64', 'float64']]

# Keep selected columns only
my_cols = categorical_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()


Step 2: Define the Model
Next, we define a random forest model with the familiar RandomForestRegressor
class.

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=0)
Step 3: Create and Evaluate the Pipeline
Finally, we use the Pipeline class to define a pipeline that bundles
the preprocessing and modeling steps. There are a few important things
to notice:

With the pipeline, we preprocess the training data and fit the model
in a single line of code. (In contrast, without a pipeline, we have
```

to do imputation, one-hot encoding, and model training in separate steps. This becomes especially messy if we have to deal with both numerical and categorical variables!)
With the pipeline, we supply the unprocessed features in X_valid to the predict() command, and the pipeline automatically preprocesses the features before generating predictions. (However, without a pipeline, we have to remember to preprocess the validation data before making predictions.)

```
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)
```

We construct the full pipeline in three steps.

Step 1: Define Preprocessing Steps
Similar to how a pipeline bundles together preprocessing and modeling steps, we use the ColumnTransformer class to bundle together different preprocessing steps. The code below:

imputes missing values in numerical data, and
imputes missing values and applies a one-hot encoding to categorical 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)
    ])


def missing_indicator(X):
    return pd.DataFrame(X.isna(), columns=[f"{col}_missing" for col
in X.columns])

# Preprocessing for numerical data with missing indicator
numerical_transformer = Pipeline(steps=[
    ('missing_indicator', FunctionTransformer(func=missing_indicator,
validate=False)),
    ('imputer', SimpleImputer(strategy='constant'))
])

# Preprocessing for categorical data with missing indicator
categorical_transformer = Pipeline(steps=[
    ('missing_indicator', FunctionTransformer(func=missing_indicator,
validate=False)),
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

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

2.4 cross validation

```
    from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
```

```
my_pipeline = Pipeline(steps=[('preprocessor', SimpleImputer()),
                              ('model', RandomForestRegressor(n_estimators=50,
                                                              random_state=0))
                             ])
```

```
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)
```

2.5 XG Boost

```
    from xgboost import XGBRegressor

my_model = XGBRegressor()
my_model.fit(X_train, y_train)

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)
```

```
early_stopping_rounds offers a way to automatically find the ideal
value for n_estimators. Early stopping causes the model to stop iterating
when the validation score stops improving, even if we aren't at the
hard stop for n_estimators. It's smart to set a high value for n_estimators
and then use early_stopping_rounds to find the optimal time to stop
iterating.
```

Since random chance sometimes causes a single round where validation scores don't improve, you need to specify a number for how many rounds of straight deterioration to allow before stopping. Setting early_stopping_rounds=5 is a reasonable choice. In this case, we stop after 5 straight rounds of deteriorating validation scores.

When using early_stopping_rounds, you also need to set aside some data for calculating the validation scores - this is done by setting the eval_set parameter.

earning_rate
Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number (known as the learning rate) before adding them in.

This means each tree we add to the ensemble helps us less. So, we can set a higher value for n_estimators without overfitting. If we use early stopping, the appropriate number of trees will be determined automatically.

In general, a small learning rate and large number of estimators will yield more accurate XGBoost models, though it will also take the model longer to train since it does more iterations through the cycle. As default, XGBoost sets learning_rate=0.1.

n_jobs
On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter n_jobs equal to the number of cores on your machine. On smaller datasets, this won't help.

The resulting model won't be any better, so micro-optimizing for fitting time is typically nothing but a distraction. But, it's useful in large datasets where you would otherwise spend a long time waiting during the fit command.

2.6 Data leakage

3 PANDAS Module

3.1 Part 1

Creating data
There are two core objects in pandas: the DataFrame and the Series.

DataFrame
A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column.

For example, consider the following simple DataFrame:

```
pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})
Yes No
```

We are using the pd.DataFrame() constructor to generate these DataFrame objects. The syntax for declaring a new one is a dictionary whose keys are the column names (Bob and Sue in this example), and whose values are a list of entries. This is the standard way of constructing a new DataFrame, and the one you are most likely to encounter.

A Series is, in essence, a single column of a DataFrame. So you can assign row labels to the Series the same way as before, using an index parameter. However, a Series does not have a column name, it only has one overall name:

```
pd.Series([30, 35, 40], index=['2015 Sales', '2016 Sales', '2017 Sales'],
name='Product A')
```

We'll use the pd.read_csv() function to read the data into a DataFrame. This goes thusly:

```
wine_reviews = pd.read_csv("../input/wine-reviews/winemag-data-130k-v2.csv")
```
We can use the shape attribute to check how large the resulting DataFrame is:

```
wine_reviews.shape
(129971, 14)
```
So our new DataFrame has 130,000 records split across 14 different columns. That's almost 2 million entries!

We can examine the contents of the resultant DataFrame using the head() command, which grabs the first five rows:

```
wine_reviews.head()

quantities = ['4 cups', '1 cup', '2 large', '1 can']
items = ['Flour', 'Milk', 'Eggs', 'Spam']
recipe = pd.Series(quantities, index=items, name='Dinner')

reviews = pd.read_csv("../input/wine-reviews/winemag-data_first150k.csv",index_col=0)


Dont do the index double!


Save a DataFrame as a csv file!
animals_df.to_csv("cows_and_goats.csv")
```

3.2 Part Indexing, Assigning

```
    reviews['country']
    reviews['country'][0]
```

Pandas indexing works in one of two paradigms. The first is index-based selection: selecting data based on its numerical position in the data. iloc follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

```
reviews.iloc[0]
```

This means that it's marginally easier to retrieve rows, and marginally harder to get retrieve columns. To get a column with iloc, we can do the following:

```
reviews.iloc[:, 0]
```

To select just the second and third entries, we would do:

```
reviews.iloc[1:3, 0]
```

It's also possible to pass a list:

```
reviews.iloc[[0, 1, 2], 0]
```

The second paradigm for attribute selection is the one followed by the loc operator: label-based selection. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in reviews, we would now do the following:

```
reviews.loc[0, 'country']
'Italy'
```

iloc is conceptually simpler than loc because it ignores the dataset's indices. When we use iloc we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. loc, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using loc instead. For example, here's one operation that's much easier using loc:

```
reviews.loc[:, ['taster_name', 'taster_twitter_handle', 'points']]
```

iloc uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So 0:10 will select entries 0,...,9. loc, meanwhile, indexes inclusively. So 0:10 will select entries 0,...,10.

This result can then be used inside of loc to select the relevant data:

```
reviews.loc[reviews.country == 'Italy']
reviews.loc[(reviews.country == 'Italy') & (reviews.points >= 90)]
```

Suppose we'll buy any wine that's made in Italy or which is rated above average. For this we use a pipe (|):

```
reviews.loc[(reviews.country == 'Italy') | (reviews.points >= 90)]

reviews.loc[reviews.country.isin(['Italy', 'France'])]
```

The second is isnull (and its companion notnull). These methods let you highlight values which are (or are not) empty (NaN). For example, to filter out wines lacking a price tag in the dataset, here's what we would do:

```
reviews.loc[reviews.price.notnull()]
```

```
These things select column description
reviews.loc[:,'description']
reviews['description']
desc = reviews.description


: means everything!!!


Select the first value from the description column of reviews, assigning
it to variable first_description

first_description =   reviews.description[0]
first_description = reviews.description.iloc[0]


Select the first row using: reviews.iloc[0]

Select records with index labels using sample_reviews = reviews.iloc[[1,2,3,5,8]]

Select sub dataframe df = reviews.loc[[0,1,10,100],['country','province','region_1','region_

The first 100 entries, since loc uses INCLUSIVE indexing.

reviews.loc[0:99,['country','variety']]

The full columns
reviews.loc[0:100,['country','variety']]
```

3.3 Part 3 Summary functions and maps

```
median_points = reviews.points.median()

reviews['ratio']=reviews.points/reviews.price

reviews['ratio']=reviews.points/reviews.price



bargain_wine = reviews.iloc[reviews.ratio.argmax()].title
```

```
bargain_idx = (reviews.points / reviews.price).idxmax()
bargain_wine = reviews.loc[bargain_idx, 'title']


list of unique countries
reviews.country.unique()


    reviews.taster_name.unique()

    reviews.taster_name.value_counts()
```

map() is the first, and slightly simpler one. For example, suppose that we wanted to remean the scores the wines received to 0. We can do this as follows:

```
review_points_mean = reviews.points.mean()
reviews.points.map(lambda p: p - review_points_mean)
```

The function you pass to map() should expect a single value from the Series (a point value, in the above example), and return a transformed version of that value. map() returns a new Series where all the values have been transformed by your function.

apply() is the equivalent method if we want to transform a whole DataFrame by calling a custom method on each row.

```
def remean_points(row):
    row.points = row.points - review_points_mean
    return row

reviews.apply(remean_points, axis='columns')
```

If we had called reviews.apply() with axis='index', then instead of passing a function to transform each row, we would need to give a function to transform each column.

Note that map() and apply() return new, transformed Series and DataFrames, respectively. They don't modify the original data they're called on. If we look at the first row of reviews, we can see that it still has its original points value.

```
review_points_mean = reviews.points.mean()
reviews.points - review_points_mean
```

COUNT HOW MANY TIMES FRUITY IS IN DESCRIPTIONS OVER THE ENTIRE DATAFRAMA

```
n_fruity = reviews.description.map(lambda desc: "fruity" in desc).sum()
```

To create a feature that is 1 if the description mentions the word "fruity" and 0 otherwise, you can use a method similar to the one in your example. Here's how you can do it:

```
reviews['is_fruity'] = reviews.description.map(lambda desc: 1 if "fruity"
in desc else 0)
```

We'd like to host these wine reviews on our website, but a rating system ranging from 80 to 100 points is too hard to understand - we'd like to translate them into simple star ratings. A score of 95 or higher counts as 3 stars, a score of at least 85 but less than 95 is 2 stars. Any other score is 1 star.

Also, the Canadian Vintners Association bought a lot of ads on the site, so any wines from Canada should automatically get 3 stars, regardless of points.

Create a series star_ratings with the number of stars corresponding to each review in the dataset.

```
def stars(row):
    if row.country == 'Canada':
        return 3
    elif row.points >= 95:
        return 3
    elif row.points >= 85:
        return 2
    else:
        return 1

star_ratings = reviews.apply(stars, axis='columns')
reviews['star_ratings'] = reviews.apply(stars, axis='columns')
```

3.4 Group and Sort

```
reviews.groupby('points').points.count()
```

groupby() created a group of reviews which allotted the same point values to the given wines. Then, for each of these groups, we grabbed the points() column and counted how many times it appeared. value_counts() is just a shortcut to this groupby() operation.

For example, to get the cheapest wine in each point value category, we can do the following:

```
reviews.groupby('points').price.min()
```

For example, here's one way of selecting the name of the first wine reviewed from each winery in the dataset:

```
reviews.groupby('winery').apply(lambda df: df.title.iloc[0])
```

For an example, here's how we would pick out the best wine by country and province:

```
reviews.groupby(['country', 'province']).apply(lambda df: df.loc[df.points.idxmax()])
```

For example, we can generate a simple statistical summary of the dataset as follows:

```
reviews.groupby(['country']).price.agg([len, min, max])
```

converting back to a regular index, the reset_index() method:

```
countries_reviewed = reviews.groupby(['country', 'province']).description.agg([len])
countries_reviewed.reset_index()
```

To get data in the order want it in we can sort it ourselves. The sort_values() method is handy for this.

```
countries_reviewed = countries_reviewed.reset_index()
countries_reviewed.sort_values(by='len')
countries_reviewed.sort_values(by='len', ascending=False)
```

Finally, know that you can sort by more than one column at a time:

```
countries_reviewed.sort_values(by=['country', 'len'])
```

Data types

```
 reviews.price.dtype
```

Alternatively, the dtypes property returns the dtype of every column in the DataFrame:

```
reviews.dtypes
```

For example, we may transform the points column from its existing int64 data type into a float64 data type:

```
reviews.points.astype('float64')
```

```
reviews[pd.isnull(reviews.country)]
```

```
reviews.region_2.fillna("Unknown")
```

```
reviews.taster_twitter_handle.replace("@kerinokeefe", "@kerino")
```

```
reviews.rename(columns={'points': 'score'})
```

```
canadian_youtube = pd.read_csv("../input/youtube-new/CAvideos.csv")
british_youtube = pd.read_csv("../input/youtube-new/GBvideos.csv")
```

```
pd.concat([canadian_youtube, british_youtube])
```

The middlemost combiner in terms of complexity is join(). join() lets you combine different DataFrame objects which have an index in common. For example, to pull down videos that happened to be trending on the same day in both Canada and the UK, we could do the following:

```
left = canadian_youtube.set_index(['title', 'trending_date'])
right = british_youtube.set_index(['title', 'trending_date'])
```

```
left.join(right, lsuffix='_CAN', rsuffix='_UK')
```

4 Feature Engineering Module

```
    X = df.copy()
y = X.pop("CompressiveStrength")

# Create synthetic features
X["FCRatio"] = X["FineAggregate"] / X["CoarseAggregate"]
X["AggCmtRatio"] = (X["CoarseAggregate"] + X["FineAggregate"]) / X["Cement"]
X["WtrCmtRatio"] = X["Water"] / X["Cement"]

# Train and score model on dataset with additional ratio features
model = RandomForestRegressor(criterion="absolute_error", random_state=0)
score = cross_val_score(
    model, X, y, cv=5, scoring="neg_mean_absolute_error"
)
score = -1 * score.mean()

print(f"MAE Score with Ratio Features: {score:.4}")
```

4.1 Mutual information

The entropy of a variable means roughly: "how many yes-or-no questions you would need to describe an occurance of that variable, on average." The more questions you have to ask, the more uncertain you must be about the variable. Mutual information is how many questions you expect the feature to answer about the target.)

MI can help you to understand the relative potential of a feature as a predictor of the target, considered by itself.
It's possible for a feature to be very informative when interacting with other features, but not so informative all alone. MI can't detect interactions between features. It is a univariate metric.
The actual usefulness of a feature depends on the model you use it with. A feature is only useful to the extent that its relationship with the target is one your model can learn. Just because a feature has a high MI score doesn't mean your model will be able to do anything with that information. You may need to transform the feature first to expose the association.

```
X = df.copy()
y = X.pop("price")

# Label encoding for categoricals
for colname in X.select_dtypes("object"):
    X[colname], _ = X[colname].factorize()
```

```
# All discrete features should now have integer dtypes (double-check
this before using MI!)
discrete_features = X.dtypes == int
Scikit-learn has two mutual information metrics in its feature_selection
module: one for real-valued targets (mutual_info_regression) and one
for categorical targets (mutual_info_classif). Our target, price, is
real-valued. The next cell computes the MI scores for our features
and wraps them up in a nice dataframe.

from sklearn.feature_selection import mutual_info_regression

def make_mi_scores(X, y, discrete_features):
    mi_scores = mutual_info_regression(X, y, discrete_features=discrete_features)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
    return mi_scores

mi_scores = make_mi_scores(X, y, discrete_features)

And now a bar plot to make comparisions easier:

def plot_mi_scores(scores):
    scores = scores.sort_values(ascending=True)
    width = np.arange(len(scores))
    ticks = list(scores.index)
    plt.barh(width, scores)
    plt.yticks(width, ticks)
    plt.title("Mutual Information Scores")


plt.figure(dpi=100, figsize=(8, 5))
plot_mi_scores(mi_scores)



import seaborn as sns

plt.style.use("seaborn-whitegrid")


sns.relplot(x="curb_weight", y="price", data=df);

The fuel_type feature has a fairly low MI score, but as we can see
from the figure, it clearly separates two price populations with different
trends within the horsepower feature. This indicates that fuel_type
```

contributes an interaction effect and might not be unimportant after all. Before deciding a feature is unimportant from its MI score, it's good to investigate any possible interaction effects -- domain knowledge can offer a lot of guidance here.

```
sns.lmplot(x="horsepower", y="price", hue="fuel_type", data=df);
```

4.2 Create Features

```
    autos["stroke_ratio"] = autos.stroke / autos.bore

autos[["stroke", "bore", "stroke_ratio"]].head()
```

```
roadway_features = ["Amenity", "Bump", "Crossing", "GiveWay",
    "Junction", "NoExit", "Railway", "Roundabout", "Station", "Stop",
    "TrafficCalming", "TrafficSignal"]
accidents["RoadwayFeatures"] = accidents[roadway_features].sum(axis=1)
```

You could also use a dataframe's built-in methods to create boolean values. In the Concrete dataset are the amounts of components in a concrete formulation. Many formulations lack one or more components (that is, the component has a value of 0). This will count how many components are in a formulation with the dataframe's built-in greater-than gt method:

```
components = [ "Cement", "BlastFurnaceSlag", "FlyAsh", "Water",
               "Superplasticizer", "CoarseAggregate", "FineAggregate"]
concrete["Components"] = concrete[components].gt(0).sum(axis=1)

concrete[components + ["Components"]].head(10)
```

The gt(0) in this line stands for "greater than 0." It creates a boolean DataFrame where each element is True if the value is greater than 0, and False otherwise.

If you run df.sum(axis=0), it will sum the values down each column:

If you run df.sum(axis=0), it will sum the values down each column:

Neural nets especially need features scaled to values not too far from 0. Tree-based models (like random forests and XGBoost) can sometimes benefit from normalization, but usually much less so.
Tree models can learn to approximate almost any combination of features, but when a combination is especially important they can still benefit from having it explicitly created, especially when data is limited.
Counts are especially helpful for tree models, since these models don't have a natural way of aggregating information across many features at once.

4.2.1 Group Transforms

. For an "average income by state", you would choose State for the grouping feature, mean for the aggregation function, and Income for the aggregated feature. To compute this in Pandas, we use the groupby and transform methods:

```
customer["AverageIncome"] = (
    customer.groupby("State")  # for each state
    ["Income"]                 # select the income
    .transform("mean")         # and compute its mean
)

customer[["State", "Income", "AverageIncome"]].head(10)
```

```
customer["StateFreq"] = (
    customer.groupby("State")
    ["State"]
    .transform("count")
    / customer.State.count()
)

customer[["State", "StateFreq"]].head(10)
```

You could use a transform like this to create a "frequency encoding" for a categorical feature.

If you're using training and validation splits, to preserve their independence,

it's best to create a grouped feature using only the training set and then join it to the validation set. We can use the validation set's merge method after creating a unique set of values with drop_duplicates on the training set:

```
# Create splits
df_train = customer.sample(frac=0.5)
df_valid = customer.drop(df_train.index)

# Create the average claim amount by coverage type, on the training
set
df_train["AverageClaim"] = df_train.groupby("Coverage")["ClaimAmount"].transform("mean")

# Merge the values into the validation set
df_valid = df_valid.merge(
    df_train[["Coverage", "AverageClaim"]].drop_duplicates(),
    on="Coverage",
    how="left",
)

df_valid[["Coverage", "AverageClaim"]].head(10)
```

4.3 Klustering

When used for feature engineering, we could attempt to discover groups of customers representing a market segment, for instance, or geographic areas that share similar weather patterns. Adding a feature of cluster labels can help machine learning models untangle complicated relationships of space or proximity.

Cluster Labels as a Feature
Applied to a single real-valued feature, clustering acts like a traditional "binning" or "discretization" transform. On multiple features, it's like "multi-dimensional binning" (sometimes called vector quantization).

The motivating idea for adding cluster labels is that the clusters will break up complicated relationships across features into simpler chunks. Our model can then just learn the simpler chunks one-by-one instead having to learn the complicated whole all at once. It's a "divide and conquer" strategy.

Since k-means clustering is sensitive to scale, it can be a good idea rescale or normalize data with extreme values. Our features are already

roughly on the same scale, so we'll leave them as-is.

As spatial features, California Housing's 'Latitude' and 'Longitude' make natural candidates for k-means clustering. In this example we'll cluster these with 'MedInc' (median income) to create economic segments in different regions of California.

```
X = df.loc[:, ["MedInc", "Latitude", "Longitude"]]
# Create cluster feature
kmeans = KMeans(n_clusters=6)
X["Cluster"] = kmeans.fit_predict(X)
X["Cluster"] = X["Cluster"].astype("category")

X.head()
```

Now let's look at a couple plots to see how effective this was. First, a scatter plot that shows the geographic distribution of the clusters. It seems like the algorithm has created separate segments for higher-income areas on the coasts.

```
sns.relplot(
    x="Longitude", y="Latitude", hue="Cluster", data=X, height=6,
);
```

```
X["MedHouseVal"] = df["MedHouseVal"]
sns.catplot(x="MedHouseVal", y="Cluster", data=X, kind="boxen", height=6);
```

4.4 PCA

PCA Best Practices
There are a few things to keep in mind when applying PCA:
PCA only works with numeric features, like continuous quantities or counts.
PCA is sensitive to scale. It's good practice to standardize your data before applying PCA, unless you know you have good reason not to.
Consider removing or constraining outliers, since they can have an undue influence on the results.

```
# Standardize
X_scaled = (X - X.mean(axis=0)) / X.std(axis=0)
```

Now we can fit scikit-learn's PCA estimator and create the principal components. You can see here the first few rows of the transformed dataset.

```
from sklearn.decomposition import PCA

# Create principal components
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Convert to dataframe
component_names = [f"PC{i+1}" for i in range(X_pca.shape[1])]
X_pca = pd.DataFrame(X_pca, columns=component_names)

X_pca.head()
```

4.5 Target Encoding

4.6 BOLUS

5 Time Series

```
    import numpy as np

df['Time'] = np.arange(len(df.index))

df.head()

import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use("seaborn-whitegrid")
plt.rc(
    "figure",
    autolayout=True,
    figsize=(11, 4),
    titlesize=18,
    titleweight='bold',
)
plt.rc(
    "axes",
    labelweight="bold",
    labelsize="large",
    titleweight="bold",
    titlesize=16,
    titlepad=10,
)
%config InlineBackend.figure_format = 'retina'

fig, ax = plt.subplots()
ax.plot('Time', 'Hardcover', data=df, color='0.75')
ax = sns.regplot(x='Time', y='Hardcover', data=df, ci=None, scatter_kws=dict(color='0.25'))
ax.set_title('Time Plot of Hardcover Sales');
```

Lag features

```
    To make a lag feature we shift the observations of the target series
so that they appear to have occured later in time. Here we've created
a 1-step lag feature, though shifting by multiple steps is possible
too.

df['Harcover_Lag_1'] = df['Hardcover'].shift(1)


df.head()
```

5.1 Trends

Notice how the Mauna Loa series above has a repeating up and down movement year after year – a short-term, seasonal change. For a change to be a part of the trend, it should occur over a longer period than any seasonal changes. To visualize a trend, therefore, we take an average over a period longer than any seasonal period in the series. For the Mauna Loa series, we chose a window of size 12 to smooth over the season within each year.

```
Engineering Trend
Once we've identified the shape of the trend, we can attempt to model
it using a time-step feature. We've already seen how using the time
dummy itself will model a linear trend:

target = a * time + b
We can fit many other kinds of trend through transformations of the
time dummy. If the trend appears to be quadratic (a parabola), we just
need to add the square of the time dummy to the feature set, giving
us:

target = a * time ** 2 + b * time + c
Linear regression will learn the coefficients a, b, and c


from statsmodels.tsa.deterministic import DeterministicProcess

dp = DeterministicProcess(
    index=tunnel.index,  # dates from the training data
    constant=True,       # dummy feature for the bias (y_intercept)
    order=1,             # the time dummy (trend)
    drop=True,           # drop terms if necessary to avoid collinearity
)
# `in_sample` creates features for the dates given in the `index` argument
X = dp.in_sample()

X.head()
```