Predicting Video Game Sales

At a whopping USD 138 billion per year, the video game industry has by now vastly outpaced the music and movie industry combined. For instance, one of the best generating box office opening (Avengers: Infinity Wars) generated about USD 259 million. In comparison, Grand Theft Auto V reached USD 818 million in sales in the first 24 hours.

Even so, the gaming industry suffers from the same problems as the music and movie industries: it's a hit-driven business, meaning that the great majority of the video game industry's software releases have been commercial failures.

In this scenario, we will investigate whether or not we can predict if a game will be a hit ... or not.

Submission

In order to submit on gradescope, you need to submit the following:

- the homework jupyter notebook it self `hw6.ipynb`
- the pdf generated from the notebook, you can get the pdf from `File->Print Preview`

In [ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

1. Load the data

In [ ]:
```
df = pd.read_csv("https://raw.githubusercontent.com/yx1215/Machine_Learning_Dataset/main/Video
GameSales.csv")
df.head()
```

Note that the name column is just an index column, so we will remove it as we would not use it for predictions.

In [ ]:
```
df = ...
df.head()
```

2. Data Overview

The dataframe contains 7 variables which we will use to predict the Global Sales (continuous). We are therefore dealing with a multi-variate regression problem. Before using ML to solve the problem, we want to have an overview of the data, so plotting is a good tool.

2.1 Sales each year

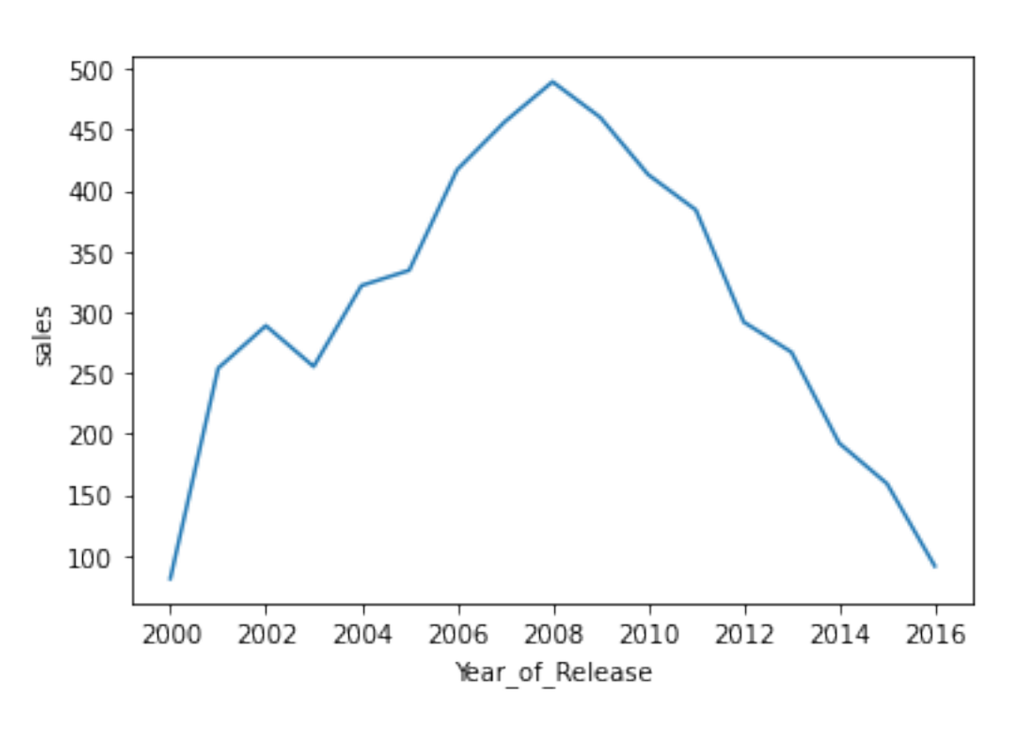
We first would like to take a look at how global sales in general change every year, so we want to create a plot whose x-axis represents the year_of_release and y-axis represents the total_sale for each year.

To create such plot, you might want to follow these steps:

1. group the records by "Year_of_Release"
2. extract the "Global_Sales" sum series
3. apply plot() to the series

You should be able to do this in one line of code using pandas.

The following is how your plot should look like.



In [ ]:
```
...
```

2.2 Sales each genre

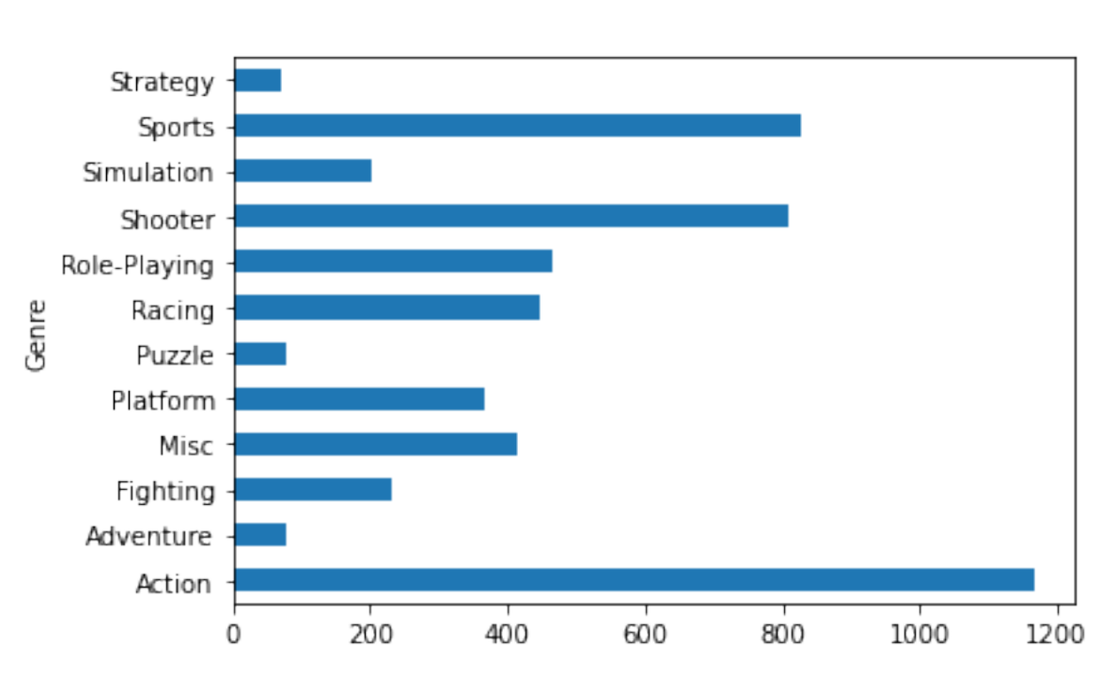
Then we want to look at sales for each genre. We want to create a bar chart whose x-axis represents the sales for each genrn while the y-axis represents different genre.

To create such plot, you might want to follow these steps:

1. group the dataframe by "Genre"
2. extract the "Global_Sales" sum series
3. to get barplots, use .plot.barh() on this series

You should be able to do this in one line of code using pandas.

The following is how your plot should look like.



In [ ]:
```
...
```

3. Train-Test split

Before training the model, we would like to split the data set as the proper way of testing the validity of an algorithm is to have a test set and set it aside for future testing. We do not touch this test-set while training the model.

In [ ]:
```
from sklearn.model_selection import train_test_split

y = df[["Global_Sales"]].values.ravel()

# Generate dummies for all catagrical features
X = pd.get_dummies(df.drop(["Global_Sales"], axis=1)).values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=0, shuf
fle=False)


print("Original:", X.shape)
print("Train:   ", X_train.shape,y_train.shape)
print("Test:    ", X_test.shape,y_test.shape)
```

Question: Note that we set the parameter shuffle=False when doing train-test split. Why is that?

(Write your solutions here)

4. Base Model

When we train some models, we always want a target to beat. Normally, we will use the constant model whose prediction of any given data is the mean of the train data, because our model should at least be better than this naive prediction, otherwise the model would be meaningless.

So we would like to know the MSE for the simple constant model, write down the code here.

In [ ]:
```
from sklearn.metrics import mean_squared_error

constant_mse = ...
print("Constant MSE: %.2f" % constant_mse)
```

5. K Nearest Neighbours

The first model we will look at is KNN Regression. sklearn has an implementation in `sklearn.neighors`.

In [ ]:
```
from sklearn.neighbors import KNeighborsRegressor
```

Let's train some knn models with different ks and see their performace on test set. Complete the following:

- Train knn models for k from 2 to 50.
- Draw a curve representing the relationship between k and test mse.
- Draw the test mse for the constant model on the same graph(this should be a horizontal line).

In [ ]:
```
test_mse_list = []
k_range = range(2, 51)
# train multiple knn models and record test mse
for k in k_range:
    ...
```

In [ ]:
```
# make the plots
plt.plot(..., label="test")
plt.plot(..., label="constant")
plt.legend()
plt.show()
```

Question: Compare the test mse for knn models with constant model, what do you find? How would you explain it?

(Write your solutions here)

6. Regularized Regression Model

Then we will look at regularized model. We've seen L1/L2 regularized linear models in lecture. Sklearn has an efficient SGD(Stochastic Gradient Descent) implementation called SGDRegressor where allow us to solve regularized linear regression.

Compared to direct solver, SGDRegressor has the following advantages:

- Efficiency
- Ease of implementation (lots of opportunities for code tuning).

SGDRegressor has the following disadvantages:

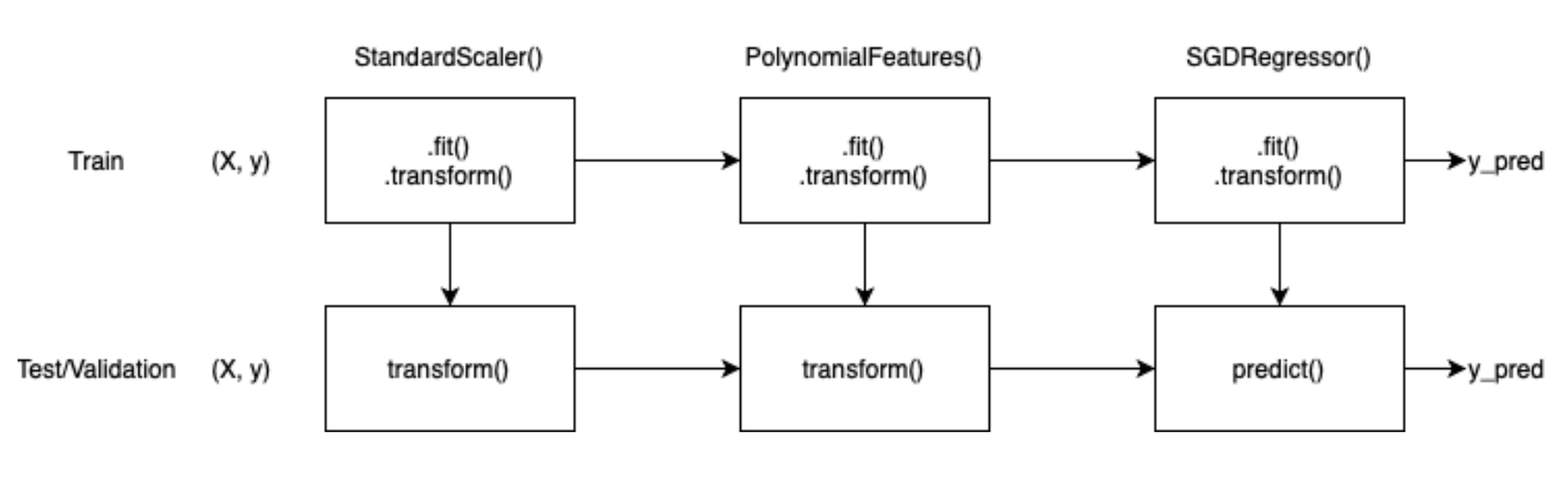
- SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

The following is an example of using it and it shows some parameters that you can tune with:

In [ ]:
```
from sklearn.linear_model import SGDRegressor


#limit the max_iter and set the random seed to fix the out put.
sgd_model = SGDRegressor(
    penalty="l1",              # the type of regularization component to use,
                               # l1 indicates Lasso and l2 indicats Ridge
    max_iter=1000,             # maximal number of epochs
    tol=1e-3,                  # tolerance for the stopping condition (stops if it
                               # can't improve the result more than tol), this speeds
                               # up model building and is great for prototyping
    alpha = 0.01,              # regularization strength, low = free model, high = controlled mo
del
    random_state=0             # random seed, fix the output, keep it 0 all the time in this hw
)
```

We are getting to a point where applying all the required steps for fitting a model is becoming cumbersome. Sklearn has a great feature called pipelines which allows you to apply all the necessary steps at once. For instance, consider a similar set-up to what we ended up with last time. We want to apply the following three steps:

1. Scale the data using a StandardScaler
2. Add polynomial features using PolynomialFeatures (in this section let's make degree=1)
3. Train a linear model using SGDRegressor (in this section let's make penalty=l1 and alpha=0.01)



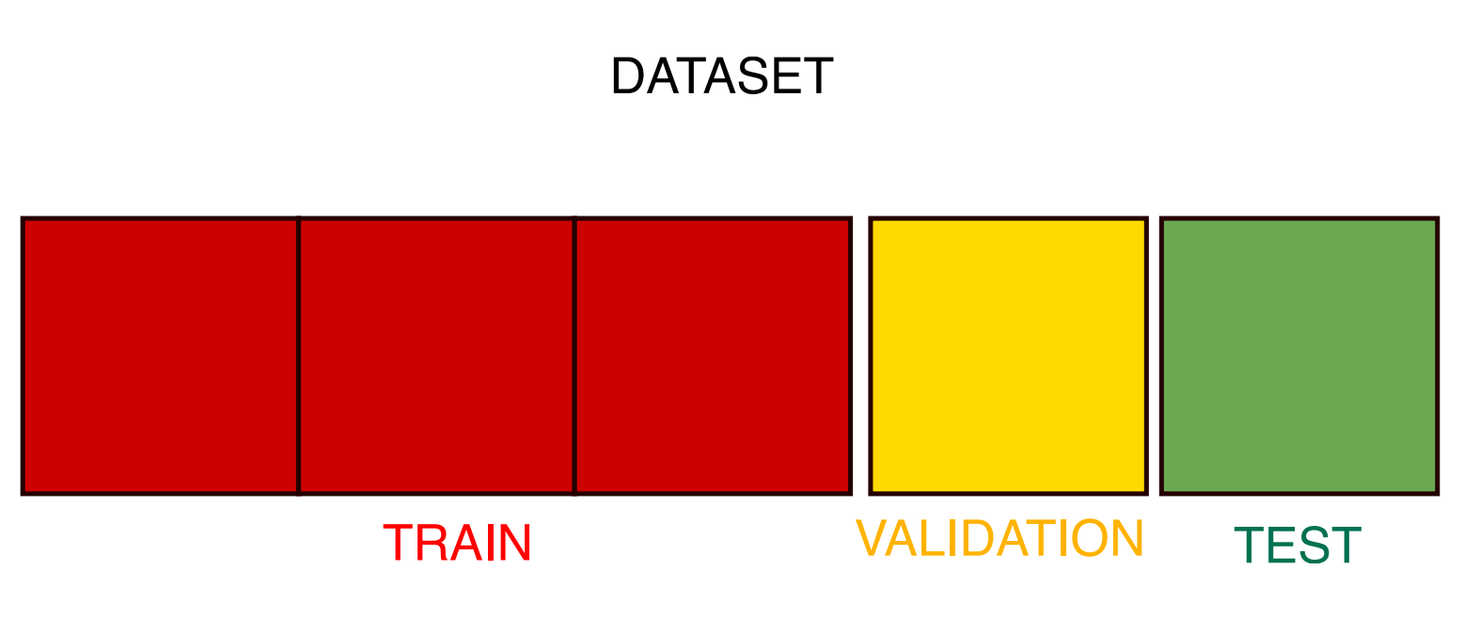
The following code implements this pipeline for our current dataset. Complete the pipeline and report the test mse.

Note: for all the sections below, whenever you train a SGDRegressor, please set the random seed to 0 to fix the output.

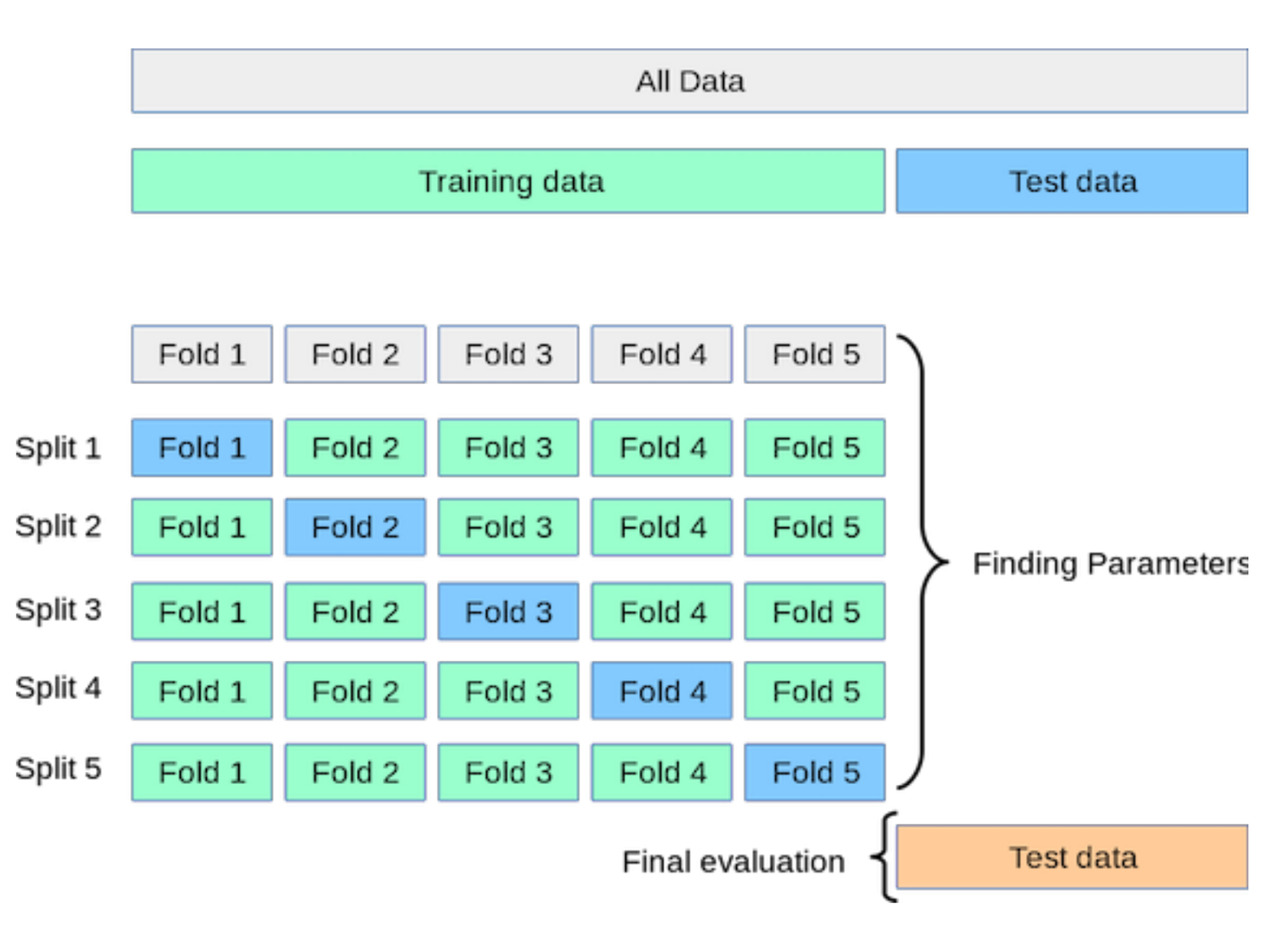
In [ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.metrics import mean_squared_error

# Design the pipeline as a sequence of steps each step
# is specified as a tuple ('name',model). We will refer
# to this name later.
pipeline = ...

# Train using the whole pipeline using just 1 call!
...

# Report the MSE on test data
...
```

7. Customized Cross Vlidation

Whenever we would like to train models and select the best parameters for them, we need to set-up a validation set. This validation set will be used for tuning parameters exclusively. This is a step that is often done wrong by novices in ML: you should never validate your parameters on the test-set. Doing so would cherry-pick the best solution and suffer from overfitting/variance problems.

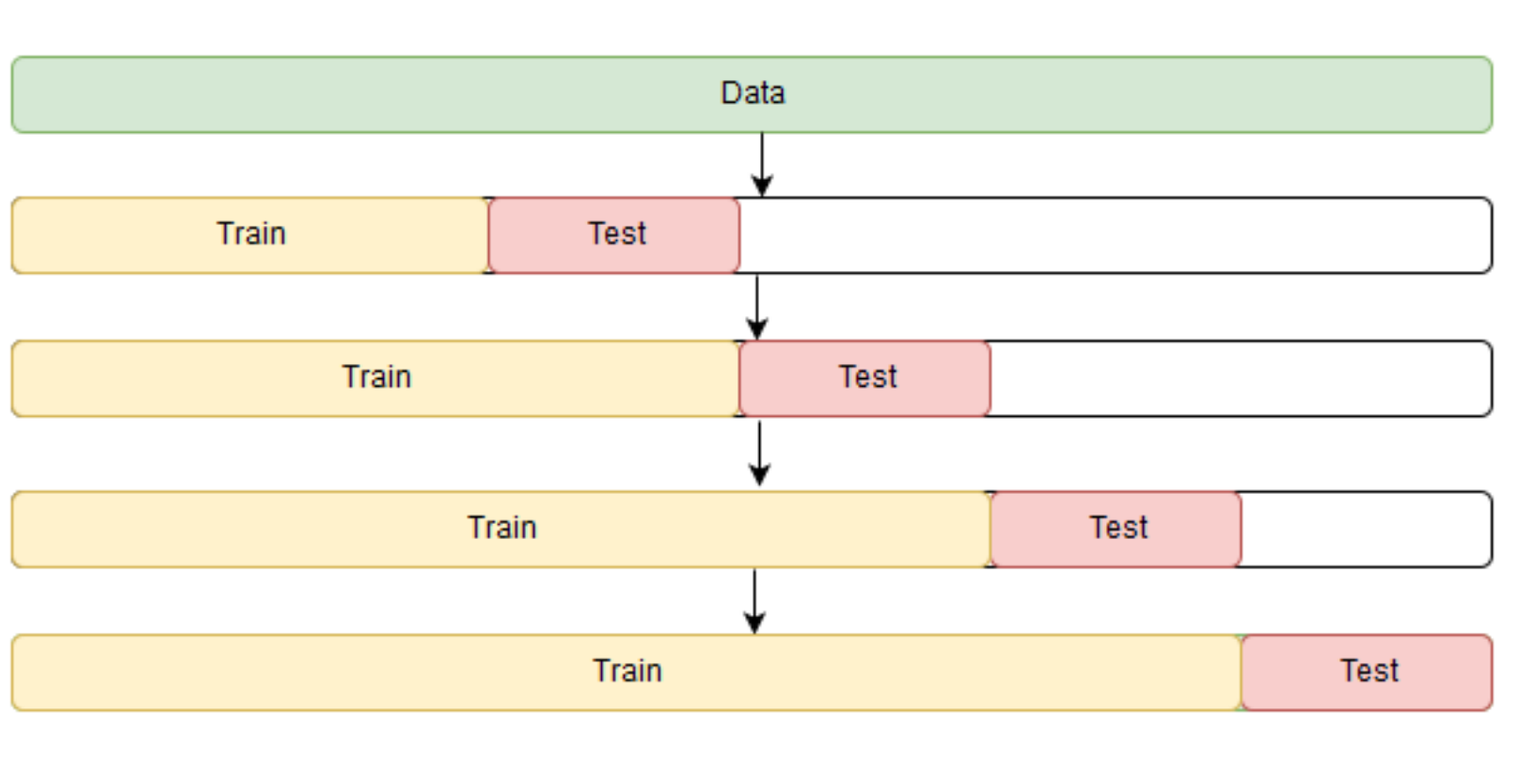


This is how we've seen it until now. In many cases, however, we will want to test it repeatedly on different parts of the data to get a more reliable out-of-sample estimate.



The final validation error is, therefore, a summary (mean) of 5 individual tests. In every test, we take a piece of the data out for the sake of validation (blue piece) and train on the remainder (green). This is how you should validate datasets without a time-component.

However, using this original way of kfold cv is also shuffling the data in some way and does not meet our requirement that we don't want to shuffle the data. In this case, we would want to split the data in the follwoing way instead:



The default `KFold()` class in sklearn.model_selection does not handle this case, so we want to implement it ourselves.

In [ ]:
```
# Note that for k=5 we only do validation for 4(k-1) times
# Input: data, an array inlucding all the train data you want to use for cv
# Output: splits, a list of tuples in the form of (train_index, val_index), just as the kfold(
) in sklearn
def customized_kfold(data, k):
    n = data.shape[0] # we first get the number of data points we have for future sue
    ...
    return splits
```

In [ ]:
```
# The function for full cross validation
# Input: X_train: full training X
#        y_train: full training y
#        k:       num of folds you want
#        model:   the sklearn model that you would like to do cv on
# Output: a tuple, (mean_train_mse, mean_val_mse)
def kfold_cross_validation(X_train, y_train, k, model):
    train_mse_list, val_mse_list = [], []
    for train_index, val_index in customized_kfold(X_train, k):
        ...
    return (mean_train_mse, mean_val_mse)
```

Then do 10-fold cross validation with the functions you just write to find the best parameters for the following models and report the test mse for the best parameters you found for each model.

- SGDRegressor with penalty=l1
- SGDRegressor with penalty=l2
- KNNRegressor

For SGDRegressor you only need to train alpha in `[0, 0.001, 0.01, 0.1, 1, 5, 10]` and degree in `[1, 2]`, and for KNNRegressor you only need to train for k in `range(10, 200, 10)`. Feel free to try larger search space but you might be careful with the run time.

7.1 SGDRegressor with penalty=l1(Lasso)

In [ ]:
```
...
best_params, best_score
```

In [ ]:
```
best_lasso = pipeline = Pipeline([
    ('ss', StandardScaler()),
    ('poly', PolynomialFeatures(...)),
    ('sgd',  SGDRegressor(penalty="l1",alpha=...,random_state=0)),
])
best_lasso.fit(X_train, y_train)
mean_squared_error(best_lasso.predict(X_test), y_test)
```

7.2 SGDRegressor with penalty=l2(Ridge)

In [ ]:
```
...
best_params, best_score
```

In [ ]:
```
best_ridge = pipeline = Pipeline([
    ('ss', StandardScaler()),
    ('poly', PolynomialFeatures(...)),
    ('sgd',  SGDRegressor(penalty="l2",alpha=...,random_state=0)),
])
best_ridge.fit(X_train, y_train)
mean_squared_error(best_ridge.predict(X_test), y_test)
```

7.3 KNN

In [ ]:
```
...
best_params, best_score
```

In [ ]:
```
best_knn = KNeighborsRegressor(...)
best_knn.fit(X_train, y_train)
mean_squared_error(best_knn.predict(X_test), y_test)
```

8. KNN Revisited

8.1 Feature Selection

We should have noticed that the performance for knn models are not that good. We would like to do something to improve its performance. We learned in class that l1 regularized regression can perform feature selection. Let's try it on our dataset.

Complete the following:

- Run SGDRegressor again with l1 penalty and the best parameters you select from cv.
- Find those features selected by the l1 regularized model and modify the dataset.
- Run knn models for k in `range(50, 2000, 100)`.
- Make a plot of 1/k versus test mse.
- Plot the test mse of the constant baseline model on the same graph.

In [ ]:
```
# Since numpy array does not have a count method, we might want to implement a
# hepler function to find the index with zero and nonzero numbers.
# Your implement does not need to take only one input, or you even don't need to implement
# this to finish the hw, this function is only used to make it easier for you to solve the
# remaining problems.
def find_zero_and_nonezero_index(lst, ...):
    ...
```

In [ ]:
```
# plug in the best params you just find.
pipeline = Pipeline([
    ('ss', StandardScaler()),
    ('poly', PolynomialFeatures(...)),
    ('sgd',  SGDRegressor(penalty="l1", alpha=...,random_state=0)),
])

# Train using the whole pipeline using just 1 call!
pipeline.fit(X_train,y_train)

# Find the coeficients for the SGDRegressor
# Hint: You can retreive the model for each step in the pipeline use pipeline.steps
```

In [ ]:
```
# Find the zero and nonzero index.
# This step is optional, if you think you don't need it, you can skip it.
# Hint: Does the number of features you retrieve from the pipeline matches the number of
#       features in the original dataframe?
nonzero_index, zero_index = find_zero_and_nonezero_index(...)
```

In [ ]:
```
test_mse_list = []
train_mse_list = []
k_range = range(50, 2000, 100)
for k in k_range:
    ...
```

In [ ]:
```
k_list = [1/k for k in k_range]
plt.plot(..., label="test")
plt.plot(..., label="constant")
plt.legend()
plt.show()
```

Question: Compare this plot with what you have in section5, is there any difference between them? If so what caused such differences?

(Write your solutions here)

Now we would like to compare how train and test mse changes. Complete the following:

- Make two subplots in the same row, the left one should be 1/k versus train mse, and the right one should be 1/k versus test mse.

In [ ]:
```
fig = plt.figure(1, figsize=[6, 3], dpi=200)
k_list = [1/k for k in k_range]
subplot = plt.subplot(...)
subplot.plot(..., label="train")
plt.legend()
subplot = plt.subplot(...)
subplot.plot(..., label="test")
plt.legend()
plt.show()
```

Question: How do the trends differ for train mse from test mse? Explain the reason.

(Write your solutions here)

8.2 More on features deleted

In last section we train knn models on those features selected by Lasso, now we will take a look at what will happen if we train on those features that are deleted. Complete the following:

- Find the features deleted by lasso
- Run knn models for k in `range(50, 2000, 100)`
- Make two subplots in the same row, the left one should be 1/k versus train mse, and the right one should be 1/k versus test mse.

In [ ]:
```
test_mse_list = []
train_mse_list = []
k_range = range(50, 2000, 100)

for k in k_range:
    ...
```

In [ ]:
```
fig = plt.figure(1, figsize=[6, 3], dpi=200)
k_list = [1/k for k in k_range]
subplot = plt.subplot(...)
subplot.plot(..., label="train")
plt.legend()
subplot = plt.subplot(...)
subplot.plot(..., label="test")
plt.legend()
plt.show()
```

Question: Are the trends for train mse and test mse the same with what you have in section 8.1? If not what are the differences? Explain.

9. Cross Validation with Modified Dataset

Now you have modified the dataset according to lasso. Please retrain the following model on the modified dataset you have and find the best parameters for them.

- SGDRegressor with penalty=l1
- SGDRegressor with penalty=l2
- KNNRegressor

For SGDRegressor you only need to train alpha in `[0, 0.001, 0.01, 0.1, 1, 5, 10]` and degree in `[1, 2]`, and for KNNRegressor you only need to train for k in `range(10, 200, 10)`. Feel free to try larger search space but you might be careful with the run time.

9.1 SGDRegressor with penalty=l1(Lasso)

In [ ]:
```
...
best_params, best_score
```

In [ ]:
```
best_lasso = pipeline = Pipeline([
    ('ss', StandardScaler()),
    ('poly', PolynomialFeatures(...)),
    ('sgd',  SGDRegressor(penalty="l1",alpha=...,random_state=0)),
])
best_lasso.fit(...)
mean_squared_error(...)
```

9.2 SGDRegressor with penalty=l2(Ridge)

In [ ]:
```
...
best_params, best_score
```

In [ ]:
```
best_ridge = pipeline = Pipeline([
    ('ss', StandardScaler()),
    ('poly', PolynomialFeatures(...)),
    ('sgd',  SGDRegressor(penalty="l2",alpha=...,random_state=0)),
])
best_ridge.fit(...)
mean_squared_error(...)
```

9.3 KNN

In [ ]:
```
...
best_params, best_score
```

In [ ]:
```
best_knn = KNeighborsRegressor(...)
best_knn.fit(...)
mean_squared_error(...)
```

Question: Compare the test mse with what you have in section 7, is there any improvements? How much does each model improve? How would you explain the difference in the improvements?

(Write your solutions here)

Question: What is your final choice for the model type and its parameter(s)? Explain your reason.

(Write your solutions here)

In [ ]:
```
```