Exercise 5 – Evaluation I

Introduction to Machine Learning

Hint: useful `Python` *libraries*

```python
# Consider the following libraries for this exercise sheet:

# general
import numpy as np
import pandas as pd
import math

# plots
import matplotlib.pyplot as plt

# sklearn
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
```

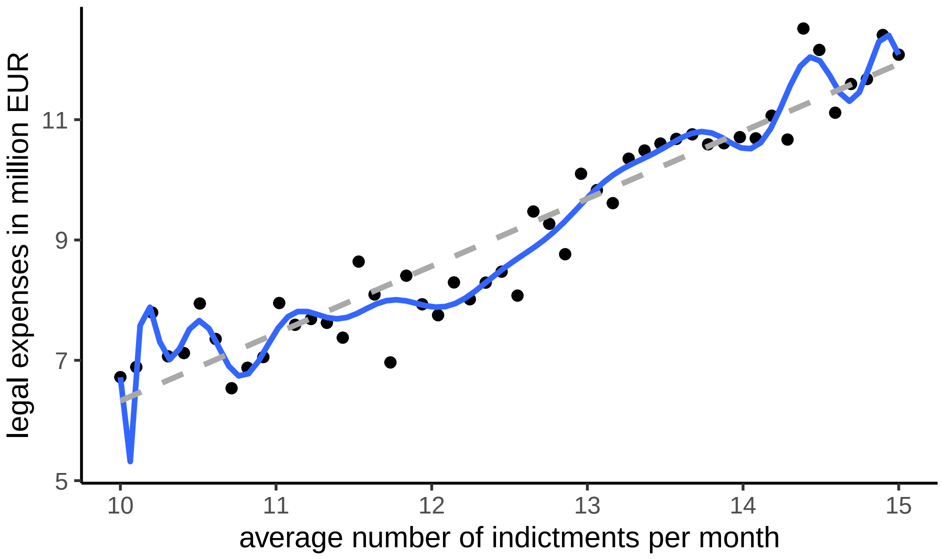
Exercise 1: Evaluating regression learners

Learning goals

1. Understand importance of using test error for evaluation
2. Discuss choice of performance metric in given situation

Imagine you work for a data science start-up and sell turn-key statistical models. Based on a set of training data, you develop a regression model to predict a customer's legal expenses from the average monthly number of indictments brought against their firm.

Due to the financial sensitivity of the situation, you opt for a very flexible learner that fits the customer's data ($n_{train} = 50$ observations) well, and end up with a degree-21 polynomial (blue, solid). Your colleague is skeptical and argues for a much simpler linear learner (gray, dashed). Which of the models will have a lower empirical risk if standard $L2$ loss is used?



Solution

Since the polynomial learner clearly achieves a better fit for the training data and some observations lie rather far from the regression line, which is strongly penalized by $L2$ loss, it will have lower empirical risk than the linear learner.

Why might evaluation based on training error not be a good idea here?

Solution

First and foremost, evaluation on the training data is almost never a good idea. Under certain conditions the training error does tell us something about generalization ability, but for flexible learners and/or small training data it is deceptive due to optimistic bias. In this particular situation, we have few training observations and quite some points that look a little extreme. A low training error might be achieved by a learner that fits every quirk in the training data but generalizes poorly to unseen points with only slightly different distribution. Evaluation on separate test data is therefore non-negotiable.

Evaluate both learners using

i. mean squared error (MSE), and

ii. mean absolute error (MAE).

State your performance assessment and explain potential differences. Use the code below to create the training and test points.

R

```r
set.seed(123)
x_train <- seq(10, 15, length.out = 50)
y_train <- 10 + 3 * sin(0.15 * pi * x_train) + rnorm(length(x_train), sd = 0.5)
data_train <- data.frame(x = x_train, y = y_train)
set.seed(321)
x_test <- seq(10, 15, length.out = 10)
y_test <- 10 + 3 * sin(0.15 * pi * x_test) + rnorm(length(x_test), sd = 0.5)
data_test <- data.frame(x = x_test, y = y_test)
```

Python

```python
np.random.seed(43)
x_train = np.linspace(10, 15, num=50)
y_train = 10 + 3 * np.sin(0.15 * math.pi * x_train)
y_train += np.random.normal(loc=0.0, scale=0.5, size=len(x_train))
data_train = pd.DataFrame({"y": y_train, "x": x_train})
np.random.seed(2238)
x_test = np.linspace(10, 15, num=50)
y_test = 10 + 3 * np.sin(0.15 * math.pi * x_test)
y_test += np.random.normal(loc=0.0, scale=0.5, size=len(x_test))
data_test = pd.DataFrame({"y": y_test, "x": x_test})
```

Solution

R

Data

```r
set.seed(123)
x_train <- seq(10, 15, length.out = 50)
```

```r
y_train <- 10 + 3 * sin(0.15 * pi * x_train) + rnorm(length(x_train), sd = 0.5)
data_train <- data.frame(x = x_train, y = y_train)
set.seed(321)
x_test <- seq(10, 15, length.out = 10)
y_test <- 10 + 3 * sin(0.15 * pi * x_test) + rnorm(length(x_test), sd = 0.5)
data_test <- data.frame(x = x_test, y = y_test)
head(data_train)
```

A data.frame: 6 × 2

	x <dbl>	y <dbl>
1	10.00000	6.719762
2	10.10204	6.888379
3	10.20408	7.793217
4	10.30612	7.066415
5	10.40816	7.119966
6	10.51020	7.943824

Train and evaluate

```r
# train learners
polynomial_learner <- lm(y ~ poly(x, 21), data_train)
linear_learner <- lm(y ~ x, data_train)

# predict with both learners
y_polynomial <- predict(polynomial_learner, data_test)
y_lm <- predict(linear_learner, data_test)

# compute errors
abs_differences <- lapply(
  list(y_polynomial, y_lm),
  function(i) abs(data_test$y - i)
)
errors_mse <- sapply(abs_differences, function(i) mean(i^2))
errors_mae <- sapply(abs_differences, mean)

sprintf(
    "MAE for Polynomial and Linear fit: %.4f, %.4f",
    errors_mae[1],
    errors_mae[2]
)
```

```
sprintf(
    "MSE for Polynomial and Linear fit: %.4f, %.4f",
    errors_mse[1],
    errors_mse[2]
)
```

'MAE for Polynomial and Linear fit: 0.4140, 0.3828'

'MSE for Polynomial and Linear fit: 0.2731, 0.3032'

Python

Data

```
np.random.seed(43)
x_train = np.linspace(10, 15, num=50)
y_train = 10 + 3 * np.sin(0.15 * math.pi * x_train)
y_train += np.random.normal(loc=0.0, scale=0.5, size=len(x_train))
data_train = pd.DataFrame({"y": y_train, "x": x_train})
np.random.seed(2238)
x_test = np.linspace(10, 15, num=50)
y_test = 10 + 3 * np.sin(0.15 * math.pi * x_test)
y_test += np.random.normal(loc=0.0, scale=0.5, size=len(x_test))
data_test = pd.DataFrame({"y": y_test, "x": x_test})
print(data_train.head())
```

```
          y          x
0  7.128700  10.000000
1  6.549227  10.102041
2  6.824611  10.204082
3  6.763703  10.306122
4  7.484359  10.408163
```

Train and evaluate

```
# train polynomial regression
poly = PolynomialFeatures(degree=21)
poly_model = LinearRegression()
poly_model.fit(poly.fit_transform(x_train.reshape(-1, 1)), y_train)

# train linear regression
```

```
lm_model = LinearRegression()
lm_model.fit(x_train.reshape(-1, 1), y_train) # reshaping necessary because single feature

# make predictions
y_poly = poly_model.predict(poly.fit_transform(x_test.reshape(-1, 1)))
y_lm = lm_model.predict(x_test.reshape(-1, 1))

# compute errors
abs_differences = pd.DataFrame(np.column_stack([y_poly, y_lm])).apply(lambda y: abs(data_t
errors_mse = abs_differences.apply(lambda x: x**2).mean() # ** is power symbol
errors_mae = abs_differences.mean()

print("MAE for Polynomial and Linear fit:", errors_mae, sep='\n')
print("MSE for Polynomial and Linear fit:", errors_mse, sep='\n')
```

```
MAE for Polynomial and Linear fit:
0    0.436762
1    0.421898
dtype: float64
MSE for Polynomial and Linear fit:
0    0.266721
1    0.276253
dtype: float64
```

The picture is inconclusive: based on MSE, we should prefer the complex polynomial model, while MAE tells us to pick the linear one. It is important to understand that the choices of inner and outer loss functions encode our requirements and may have substantial impact on the result. Our learners differ strongly in their complexity: we have an extremely flexible polynomial and a very robust (though perhaps underfitting) linear one. If, for example, our test data contains an extreme point far from the remaining observations, the polynomial model might be able to fit it fairly well, while the LM incurs a large MSE because the distance to this point enters quadratically. The MAE, on the other hand, is more concerned with small residuals, and there, our LM fares better. Here, following the MAE assessment would signify preference for a robust model.

However, we must keep in mind that our performance evaluation is based on a single holdout split, which is not advisable in general and particularly deceptive with so little data. For different test data we quickly get in situations where the polynomial has both worse MSE and MAE – after all, slapping a learner with 21 + 1 learnable parameters on a 50-points data set should strike you as a rather bad idea.

Take-home message: the choice of our performance metric matters, and making decisions based on a single train-test split is risky in many data settings.

Exercise 2: Importance of train-test split

Learning goals

1) Understand how strongly performance estimates can depend on data samples
2) Explain bias-variance trade-off of GE estimator for repeated sampling across different train/test splits

We consider the `CaliforniaHousing` data for which we would like to predict the median house value (`MedHouseVal`) from the median income in the neighborhood (`MedInc`).

R

Get data

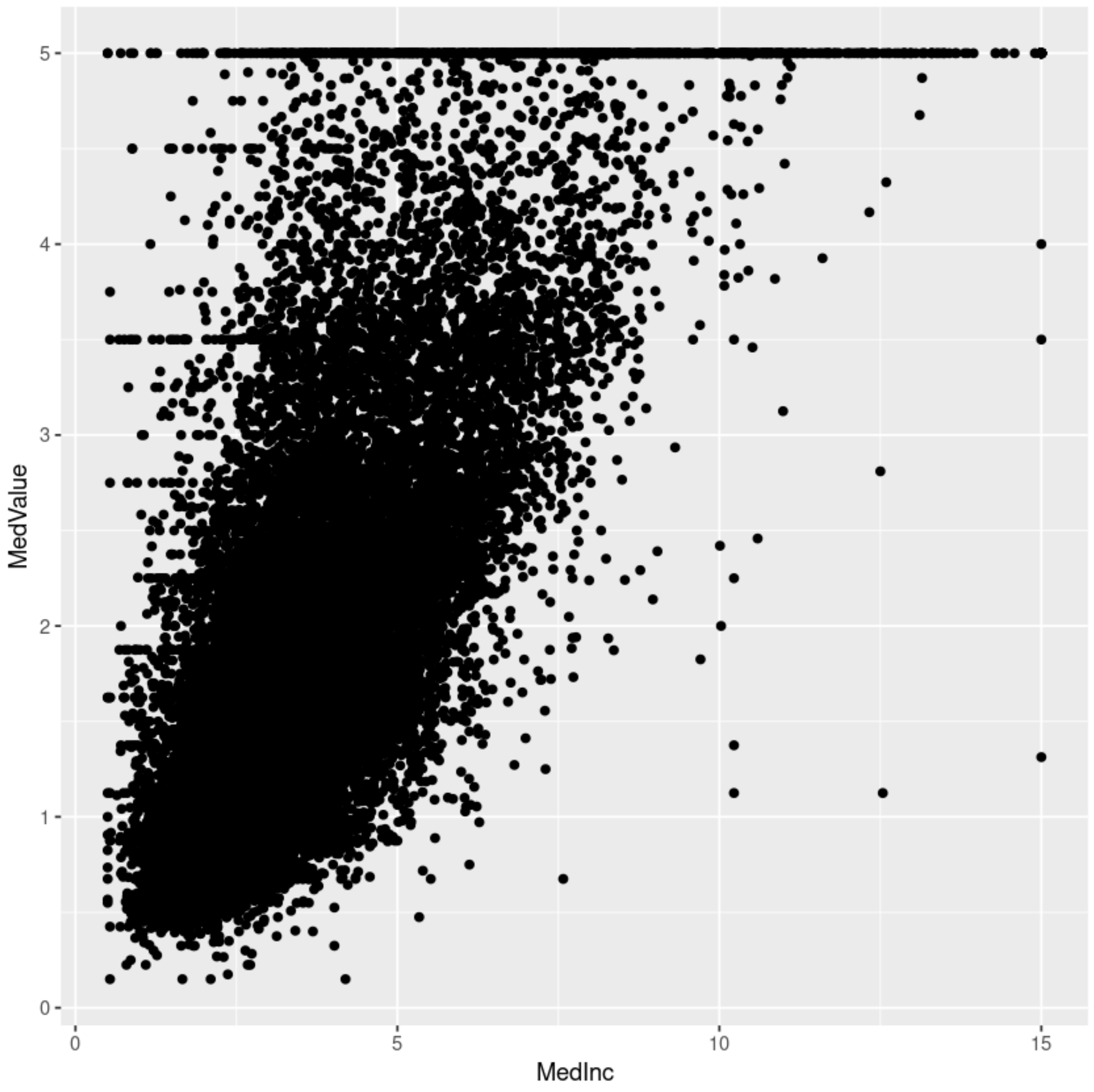
```r
# Adapt to version available on sklearn following description from
# https://gist.github.com/bgreenwell/b1330460eec5acf1c81fae71902e331c

setwd(tempdir())
url <- "https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.tgz"
download.file(url, destfile = "cal.tar.gz")
untar("cal.tar.gz")

# Read the data into R and provide column names
df_california <- read.csv("CaliforniaHousing//cal_housing.data", header = FALSE)
columns_index <- c(9, 8, 3, 4, 5, 6, 7, 2, 1)
df_california <- df_california[, columns_index]
names(df_california) <- c(
    "MedValue",
    "MedInc",
    "HouseAge",
    "AveRooms",
    "AveBedrms",
    "Population",
    "AveOccup",
    "Latitude",
    "Longitude"
)
df_california$MedValue <- df_california$MedValue / 100000
df_california <- df_california[, c("MedInc", "MedValue")]
head(df_california)
```

A data.frame: 6 × 2

	MedInc <dbl>	MedValue <dbl>
1	8.3252	4.526
2	8.3014	3.585
3	7.2574	3.521
4	5.6431	3.413
5	3.8462	3.422
6	4.0368	2.697

Inspect

```
ggplot(dataset_california, aes(x = MedInc, y = MedValue)) +
    geom_point()
```



Python

Get data

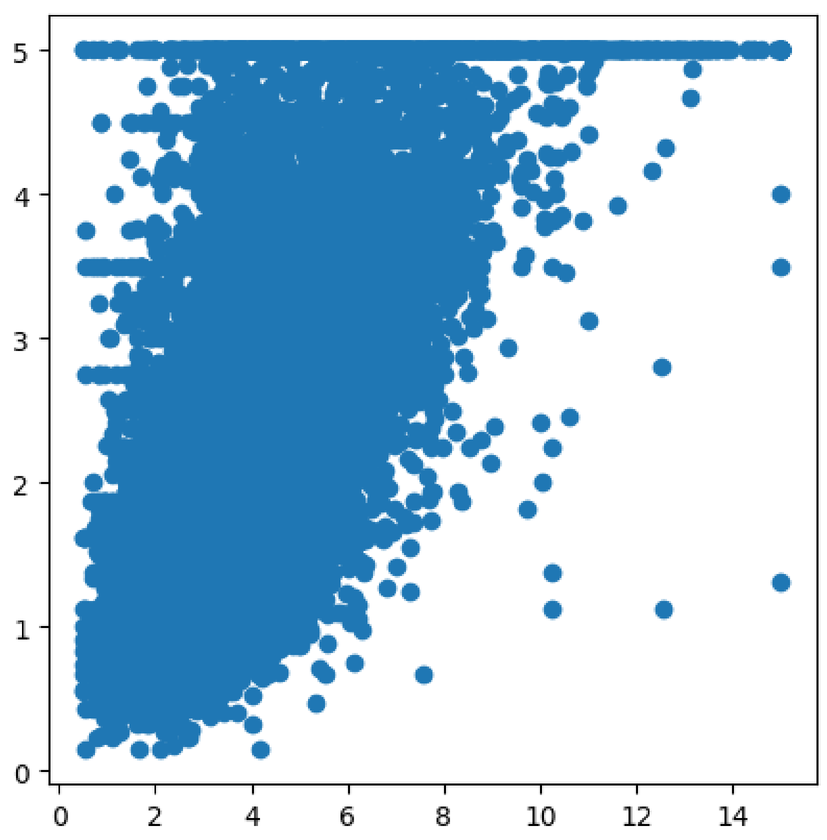
```
from sklearn.datasets import fetch_california_housing
dataset_california = fetch_california_housing(as_frame=True)
df_california = dataset_california.frame.loc[:, ['MedInc', 'MedHouseVal']]
```

```
df_california.head()
```

	MedInc	MedHouseVal
0	8.3252	4.526
1	8.3014	3.585
2	7.2574	3.521
3	5.6431	3.413
4	3.8462	3.422

Inspect

```
# instantiate plot
fig = plt.figure(figsize=(4, 4))
# Creating axes instance
ax = fig.add_axes([0, 0, 1, 1])
# Creating plot
sp = ax.scatter(
    x=df_california.loc[:, ['MedInc']],
    y=df_california.loc[:, ['MedHouseVal']]
)
plt.show()
```



Use the first 100 observations as training data to compute a linear model and evaluate the performance of your learner on the remaining data using MSE.

Solution

Get the data, define a task and corresponding train-test split, and predict with trained model:

R

Get data

```
# Adapt to version available on sklearn following description from
# https://gist.github.com/bgreenwell/b1330460eec5acf1c81fae71902e331c

setwd(tempdir())
```

```r
url <- "https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.tgz"
download.file(url, destfile = "cal.tar.gz")
untar("cal.tar.gz")

# Read the data into R and provide column names
df_california <- read.csv("CaliforniaHousing//cal_housing.data", header = FALSE)
columns_index <- c(9, 8, 3, 4, 5, 6, 7, 2, 1)
df_california <- df_california[, columns_index]
names(df_california) <- c(
    "MedValue",
    "MedInc",
    "HouseAge",
    "AveRooms",
    "AveBedrms",
    "Population",
    "AveOccup",
    "Latitude",
    "Longitude"
)
df_california$MedValue <- df_california$MedValue / 100000
df_california <- df_california[, c("MedInc", "MedValue")]
head(df_california)
```

A data.frame: 6 × 2

	MedInc <dbl>	MedValue <dbl>
1	8.3252	4.526
2	8.3014	3.585
3	7.2574	3.521
4	5.6431	3.413
5	3.8462	3.422
6	4.0368	2.697

Perform split

```r
df_california_sorted <- df_california[order(df_california$MedInc), ]
task <- TaskRegr$new(
    "housing", backend = df_california_sorted, target = "MedValue"
)
train_set = 1:100
test_set = setdiff(seq_len(task$nrow), train_set)
```

Train and evaluate

```r
# train linear learner
learner <- lrn("regr.lm")
learner$train(task, row_ids = train_set)

# predict on test data
predictions <- learner$predict(task, row_ids = test_set)
sprintf("MSE of test data %.4f:", predictions$score())
```

'MSE of test data 4.1522:'

Python

Get data

```python
from sklearn.datasets import fetch_california_housing
dataset_california = fetch_california_housing(as_frame=True)
df_california = dataset_california.frame.loc[:, ['MedInc', 'MedHouseVal']]
df_california.head()
```

	MedInc	MedHouseVal
0	8.3252	4.526
1	8.3014	3.585
2	7.2574	3.521
3	5.6431	3.413
4	3.8462	3.422

Perform split

```python
df_california_sorted = df_california.sort_values('MedInc')
df_california_sorted.reset_index(drop=True, inplace=True)
x_train = df_california_sorted.loc[:99, ['MedInc']]
y_train = df_california_sorted.loc[:99, ['MedHouseVal']]
x_test = df_california_sorted.loc[100:, ['MedInc']]
y_test = df_california_sorted.loc[100:, ['MedHouseVal']]
```

Train and evaluate

```
# train linear learner
lm_california = LinearRegression()
lm_california.fit(X = x_train, y = y_train)

pred = lm_california.predict(x_test)
mse = mean_squared_error(y_test, pred)
print(f"MSE of test data: {mse:.4f}")
```

```
MSE of test data: 4.1522
```

What might be disadvantageous about the previous train-test split?

Solution

We have chosen the first few observations from a data set that is ordered by feature value, which is not a good idea because this is not a random sample and covers only a particular area of the feature space. Consequently, we obtain a pretty high test MSE (relatively speaking – we will see in the next exercise which error values we can usually expect for this task). Looking at the data, Looking at the data, this gives us a regression line that does not reflect the overall data situation. Also, the training set is pretty small and will likely lead to poor generalization.

Now, sample your training observations from the data set at random. Use a share of 0.1 through 0.9, in 0.1 steps, of observations for training and repeat this procedure ten times. Afterwards, plot the resulting test errors (in terms of MSE) in a suitable manner.

Hint

R

`rsmp` is a convenient function for splitting data – you will want to choose the “holdout” strategy. Afterwards, `resample` can be used to repeatedly fit the learner.

Python

`from sklearn.model_selection import train_test_split` is a convenient function for splitting data. It has an optional parameter `random_state`, which can be used to split the data randomly in each iteration.

Solution

We repeat the above procedure for different train-test splits like so:

R

Train and predict for different splits

```r
# define train-test splits
repetitions <- 1:10
split_ratios <- seq(0.1, 0.9, by = 0.1)

# create resampling objects with holdout strategy, using lapply for efficient computation
split_strategies <- lapply(split_ratios, function(i) rsmp("holdout", ratio = i))

# train linear learners and predict in one step (remember to set a seed)
set.seed(123)
results <- list()
lgr::get_logger("mlr3")$set_threshold("warn")
for (i in repetitions) {
    results[[i]] <- lapply(
        split_strategies, function(i) mlr3::resample(task, learner, i)
    )
}
```

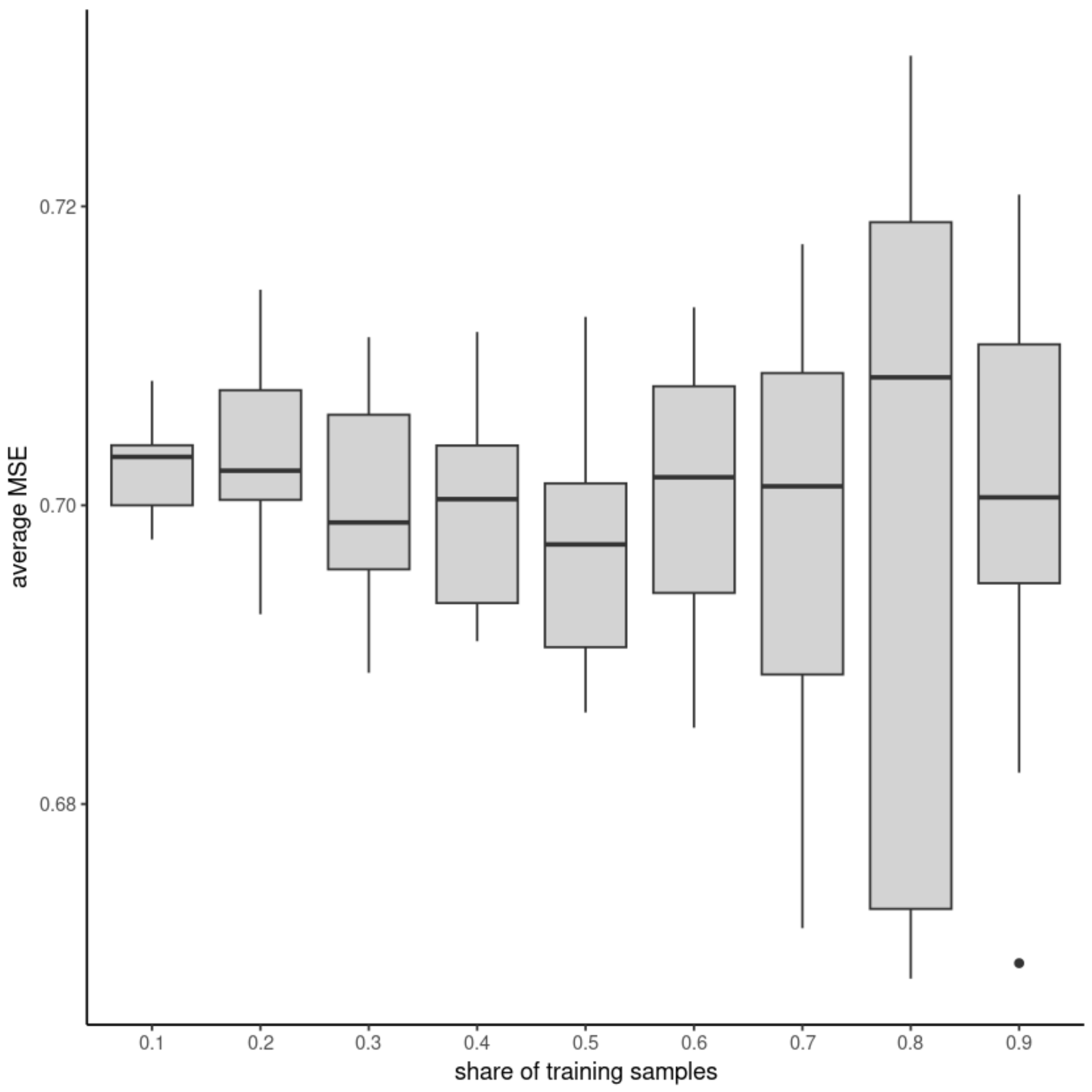
Compute errors

```r
errors <- lapply(
  repetitions,
  function(i) sapply(results[[i]], function(j) j$score()$regr.mse)
)

# assemble everything in data.frame and convert to long format for plotting
errors_df <- as.data.frame(do.call(cbind, errors))
errors_df$split_ratios <- split_ratios
errors_df_long <- reshape2::melt(errors_df, id.vars = "split_ratios")
```

```r
names(errors_df_long)[2:3] <- c("repetition", "mse")
```

Visualize

```r
# plot errors vs split ratio
ggplot(errors_df_long, aes(x = as.factor(split_ratios), y = mse)) +
  geom_boxplot(fill = "lightgray") +
  theme_classic() +
  labs(x = "share of training samples", y = "average MSE")
```



Python

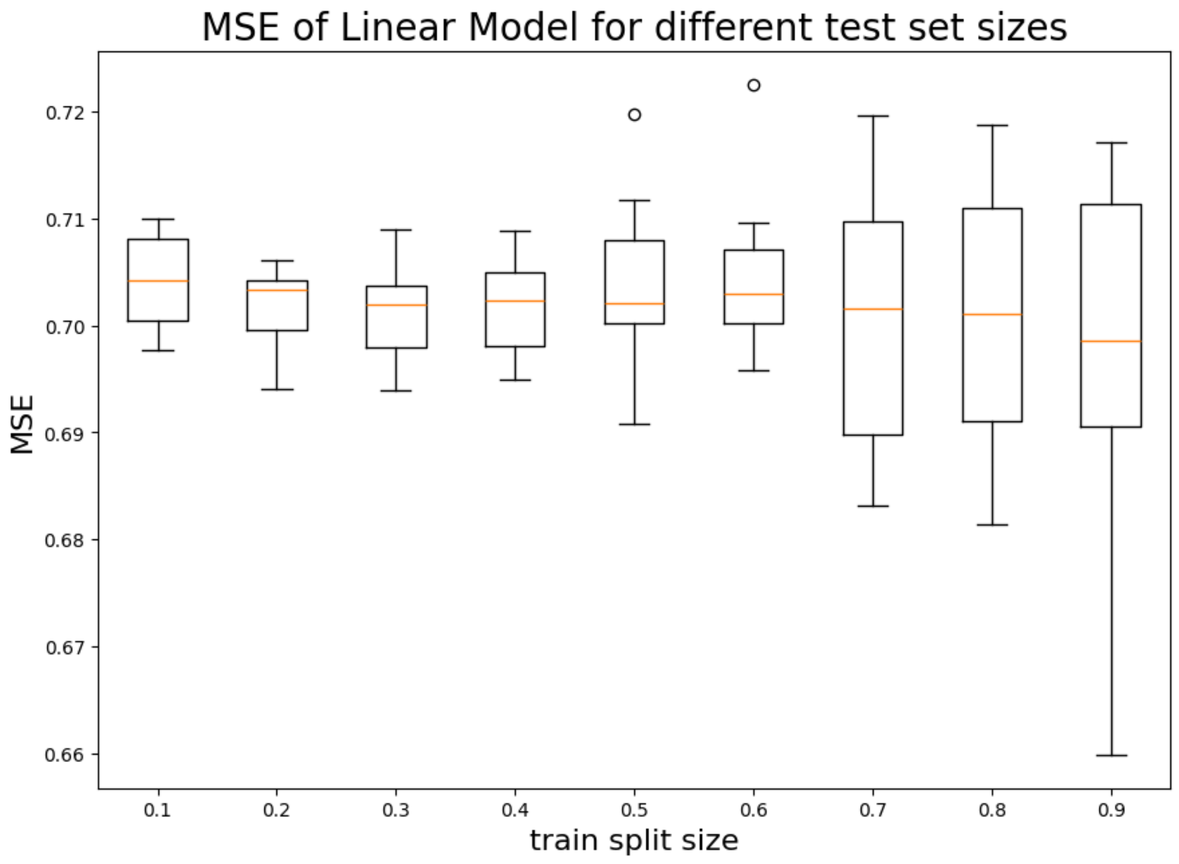
Train, predict and compute errors for different splits

```
x, y = df_california.loc[:, ['MedInc']], df_california.loc[:, ['MedHouseVal']]
split_ratio = np.linspace(0.1, 0.9, num=9)
```

```
result = pd.DataFrame()
for i in range(10):
    err = []
    for split in split_ratio:
        x_train, x_test, y_train, y_test = train_test_split(
            x, y, train_size = split, random_state=i + 10
        )
        lm_california.fit(X=x_train, y=y_train)
        pred = lm_california.predict(x_test)
        err.append(mean_squared_error(y_test, pred))
    result[i] = err

errors = result.transpose()
errors.columns = [f'0.{i}' for i in range(1, 10)]
```

Visualize

```
# plot your errors
# instantiate plot
fig = plt.figure(figsize=(8, 5.5))
# Creating axes instance
ax = fig.add_axes([0, 0, 1, 1])
# Creating plot
bp = ax.boxplot(errors)
# x-axis labels
ax.set_xticklabels([f'0.{i}' for i in range(1, 10)])
# Adding title
plt.title("MSE of Linear Model for different test set sizes", size=20)
plt.xlabel('train split size', size=16)
plt.ylabel('MSE', size=16)
# show plot
plt.show()
```



Interpret the findings from the previous question.

Solution

We can derive two conclusions:

i. A smaller training set tends to produce higher estimated generalization errors.

ii. A larger training set, at the expense of test set size, will cause high variance in the individual generalization error estimates.