Problem 1

First we need to load the data. After extracting the zip file from UCI, we see `communities.data` and `communities.names`. The `.data` file contains the raw data (with no column headers), and the `.names` file has the headers. We put the column names into a list below.

```
column_names = [
    "state", "county", "community", "communityname", "fold", "population",
    "householdsize",
    "racepctblack", "racePctWhite", "racePctAsian", "racePctHisp", "agePct12t21",
    "agePct12t29",
    "agePct16t24", "agePct65up", "numbUrban", "pctUrban", "medIncome",
    "pctWWage", "pctWFarmSelf",
    "pctWInvInc", "pctWSocSec", "pctWPubAsst", "pctWRetire", "medFamInc",
    "perCapInc", "whitePerCap",
    "blackPerCap", "indianPerCap", "AsianPerCap", "OtherPerCap", "HispPerCap",
    "NumUnderPov",
    "PctPopUnderPov", "PctLess9thGrade", "PctNotHSGrad", "PctBSorMore",
    "PctUnemployed", "PctEmploy",
    "PctEmplManu", "PctEmplProfServ", "PctOccupManu", "PctOccupMgmtProf",
    "MalePctDivorce",
    "MalePctNevMarr", "FemalePctDiv", "TotalPctDiv", "PersPerFam", "PctFam2Par",
    "PctKids2Par",
    "PctYoungKids2Par", "PctTeen2Par", "PctWorkMomYoungKids", "PctWorkMom",
    "NumIlleg", "PctIlleg",
    "NumImmig", "PctImmigRecent", "PctImmigRec5", "PctImmigRec8",
    "PctImmigRec10", "PctRecentImmig",
    "PctRecImmig5", "PctRecImmig8", "PctRecImmig10", "PctSpeakEnglOnly",
    "PctNotSpeakEnglWell",
    "PctLargHouseFam", "PctLargHouseOccup", "PersPerOccupHous",
    "PersPerOwnOccHous", "PersPerRentOccHous",
    "PctPersOwnOccup", "PctPersDenseHous", "PctHousLess3BR", "MedNumBR",
    "HousVacant", "PctHousOccup",
    "PctHousOwnOcc", "PctVacantBoarded", "PctVacMore6Mos", "MedYrHousBuilt",
    "PctHousNoPhone",
    "PctWOFullPlumb", "OwnOccLowQuart", "OwnOccMedVal", "OwnOccHiQuart",
    "RentLowQ", "RentMedian",
    "RentHighQ", "MedRent", "MedRentPctHousInc", "MedOwnCostPctInc",
    "MedOwnCostPctIncNoMtg",
    "NumInShelters", "NumStreet", "PctForeignBorn", "PctBornSameState",
    "PctSameHouse85", "PctSameCity85",
    "PctSameState85", "LemasSwornFT", "LemasSwFTPerPop", "LemasSwFTFieldOps",
    "LemasSwFTFieldPerPop",
    "LemasTotalReq", "LemasTotReqPerPop", "PolicReqPerOffic", "PolicPerPop",
    "RacialMatchCommPol",
    "PctPolicWhite", "PctPolicBlack", "PctPolicHisp", "PctPolicAsian",
    "PctPolicMinor",
    "OfficAssgnDrugUnits", "NumKindsDrugsSeiz", "PolicAveOTWorked", "LandArea",
    "PopDens",
    "PctUsePubTrans", "PolicCars", "PolicOperBudg", "LemasPctPolicOnPatr",
    "LemasGangUnitDeploy",
    "LemasPctOfficDrugUn", "PolicBudgPerPop", "ViolentCrimesPerPop"
]
```

```
import pandas as pd
import warnings

warnings.filterwarnings('ignore')

file_path = "/content/communities.data"
df = pd.read_csv(file_path, names=column_names, na_values=["?"])
df.head(10)
```

	state	county	community	communityname	fold	population	householdsize	racepctblack	raceP
0	8	NaN	NaN	Lakewoodcity	1	0.19	0.33	0.02	0.90
1	53	NaN	NaN	Tukwilacity	1	0.00	0.16	0.12	0.74
2	24	NaN	NaN	Aberdeentown	1	0.00	0.42	0.49	0.56
3	34	5.0	81440.0	Willingborotownship	1	0.04	0.77	1.00	0.08
4	42	95.0	6096.0	Bethlehemtownship	1	0.01	0.55	0.02	0.95
5	6	NaN	NaN	SouthPasadenacity	1	0.02	0.28	0.06	0.54
6	44	7.0	41500.0	Lincolntown	1	0.01	0.39	0.00	0.98
7	6	NaN	NaN	Selmacity	1	0.01	0.74	0.03	0.46
8	21	NaN	NaN	Hendersoncity	1	0.03	0.34	0.20	0.84
9	29	NaN	NaN	Claytoncity	1	0.01	0.40	0.06	0.87

```
missing_percentage = (df.isnull().sum() / len(df)) * 100
missing_percentage = missing_percentage[missing_percentage > 0]  # Filter only
missing ones

print("Percentage of missing values per column:")
print(missing_percentage)
```

```
Percentage of missing values per column:
county                  58.876630
```

```
community                59.027081
OtherPerCap               0.050150
LemasSwornFT             84.002006
LemasSwFTPerPop          84.002006
LemasSwFTFieldOps        84.002006
LemasSwFTFieldPerPop     84.002006
LemasTotalReq            84.002006
LemasTotReqPerPop        84.002006
PolicReqPerOffic         84.002006
PolicPerPop              84.002006
RacialMatchCommPol       84.002006
PctPolicWhite            84.002006
PctPolicBlack            84.002006
PctPolicHisp             84.002006
PctPolicAsian            84.002006
PctPolicMinor            84.002006
OfficAssgnDrugUnits      84.002006
NumKindsDrugsSeiz        84.002006
PolicAveOTWorked         84.002006
PolicCars                84.002006
PolicOperBudg            84.002006
LemasPctPolicOnPatr      84.002006
LemasGangUnitDeploy      84.002006
PolicBudgPerPop          84.002006
dtype: float64
```

From the above, we see that most variables that have missing entries are missing them for the majority of data points. Because imputing so many missing entries is unlikely to be accurate, we drop all of these variables except `OtherPerCap`. For `OtherPerCap`, we impute with the median.

Also, because the UCI page says that `state`, `communityname`, and `fold` are not counted as predictive, we drop them as well. However, `state` could be included and treated as a 50-length one-hot encoded vector, giving 50 extra features.

```python
cols_to_drop = [
    "state", "county", "community", "fold", "communityname", "LemasSwornFT",
    "LemasSwFTPerPop",
    "LemasSwFTFieldOps", "LemasSwFTFieldPerPop", "LemasTotalReq",
    "LemasTotReqPerPop",
    "PolicReqPerOffic", "PolicPerPop", "RacialMatchCommPol", "PctPolicWhite",
    "PctPolicBlack",
    "PctPolicHisp", "PctPolicAsian", "PctPolicMinor", "OfficAssgnDrugUnits",
    "NumKindsDrugsSeiz",
    "PolicAveOTWorked", "PolicCars", "PolicOperBudg", "LemasPctPolicOnPatr",
    "LemasGangUnitDeploy", "PolicBudgPerPop"
]
df.drop(columns=cols_to_drop, inplace=True)

df["OtherPerCap"].fillna(df["OtherPerCap"].median(), inplace=True)
```

```
df.head(10)
```

	population	householdsize	racepctblack	racePctWhite	racePctAsian	racePctHisp	agePct12t21	ageP
0	0.19	0.33	0.02	0.90	0.12	0.17	0.34	0.47
1	0.00	0.16	0.12	0.74	0.45	0.07	0.26	0.59
2	0.00	0.42	0.49	0.56	0.17	0.04	0.39	0.47
3	0.04	0.77	1.00	0.08	0.12	0.10	0.51	0.50
4	0.01	0.55	0.02	0.95	0.09	0.05	0.38	0.38
5	0.02	0.28	0.06	0.54	1.00	0.25	0.31	0.48
6	0.01	0.39	0.00	0.98	0.06	0.02	0.30	0.37
7	0.01	0.74	0.03	0.46	0.20	1.00	0.52	0.55
8	0.03	0.34	0.20	0.84	0.02	0.00	0.38	0.45
9	0.01	0.40	0.06	0.87	0.30	0.03	0.90	0.82

We see that the aforementioned variables are no longer in the table. Also, we remove the target variable from the data and standardize the features, which is important for Lasso and Elastic Net.

```
X = df.drop(columns=["ViolentCrimesPerPop"])  # Exclude target variable
y = df["ViolentCrimesPerPop"]

# Standardize features
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Part (a)

Part (i)

Below we find the most important features as determined by the output p-values for ordinary least squares. We show the top 10 features. Note that smaller p-values indicate that the variable is more important in this interpretation.

```
import statsmodels.api as sm
import numpy as np
from sklearn.linear_model import LinearRegression, LassoCV, ElasticNetCV
from sklearn.feature_selection import RFE
from itertools import combinations

num_features = 10
```

```
X_const = sm.add_constant(X)
ols_model = sm.OLS(y, X_const).fit()

# Get p-values and sort features by significance
ols_pvalues = ols_model.pvalues.sort_values()
print("Top features based on Least Squares p-values:")
print(ols_pvalues.head(num_features))
```

```
Top features based on Least Squares p-values:
racepctblack            0.000077
NumStreet               0.000111
RentLowQ                0.000415
PctWorkMom              0.000596
MedOwnCostPctIncNoMtg   0.000660
MalePctNevMarr          0.000712
PersPerRentOccHous      0.001467
PctEmploy               0.001697
PctVacMore6Mos          0.001926
pctUrban                0.002924
dtype: float64
```

For best subset selection, we have too many variables to run this in a reasonable time: $\binom{101}{10} \approx 1.92 \times 10^{13}$. To reduce the number of variables, we select only those variables in the above OLS for which the p-value is less than 0.01.

Also, note that we should use `X_scaled` and not `X` for Lasso and Elastic Net since the scaling affects these models.

```
significant_features = ols_pvalues[ols_pvalues < 0.01].index.drop("const",
errors="ignore")  # Remove intercept
X_significant = X[significant_features]
```

```
def best_subset(X, y):
    best_score = float("inf")
    best_features = None

    for subset in combinations(X.columns, num_features):
        X_sub = X[list(subset)]
        X_sub_const = sm.add_constant(X_sub)
        model = sm.OLS(y, X_sub_const).fit()
        score = model.aic  # Use AIC to evaluate model

        if score < best_score:
            best_score = score
            best_features = subset

    return best_features
```

```
best_features = best_subset(X_significant, y)
print("Best subset features:")
print("\n".join(best_features))
```

```
Best subset features:
racepctblack
NumStreet
RentLowQ
MedOwnCostPctIncNoMtg
PersPerRentOccHous
PctEmploy
pctUrban
PctPersDenseHous
MedRent
pctWInvInc
```

```
rfe = RFE(LinearRegression(), n_features_to_select=num_features)
rfe.fit(X_scaled, y)

selected_features = X.columns[rfe.support_]
print("Top features from RFE:")
print("\n".join(selected_features))
```

```
Top features from RFE:
population
racepctblack
numbUrban
MalePctDivorce
FemalePctDiv
TotalPctDiv
PctKids2Par
PctPersDenseHous
OwnOccLowQuart
OwnOccMedVal
```

```
lasso = LassoCV(cv=5).fit(X_scaled, y)
lasso_coefs = pd.Series(lasso.coef_, index=X.columns)

# Interpret most important features as those with largest absolute weight
top_lasso_features = np.abs(lasso_coefs).sort_values(ascending=False)
print("Top Lasso Features:")
print(top_lasso_features[:num_features])
```

```
Top Lasso Features:
PctKids2Par         0.055324
racepctblack        0.048707
```

```
PersPerOccupHous    0.046583
MedRent             0.045962
RentLowQ            0.043216
agePct12t29         0.034119
PctPersDenseHous    0.033699
PctIlleg            0.033338
PctPopUnderPov      0.031131
MalePctDivorce      0.027432
dtype: float64
```

```
elastic_net = ElasticNetCV(cv=5).fit(X_scaled, y)
enet_coefs = pd.Series(elastic_net.coef_, index=X.columns)

# Interpret most important features as those with largest absolute weight
top_enet_features = np.abs(enet_coefs).sort_values(ascending=False)
print("Top Elastic Net Features:")
print(top_enet_features[:num_features])
```

```
Top Elastic Net Features:
PctKids2Par         0.054727
racepctblack        0.048526
MedRent             0.045423
PersPerOccupHous    0.045108
RentLowQ            0.042862
agePct12t29         0.033858
PctIlleg            0.033567
PctPersDenseHous    0.033411
PctPopUnderPov      0.030891
MalePctDivorce      0.027249
dtype: float64
```

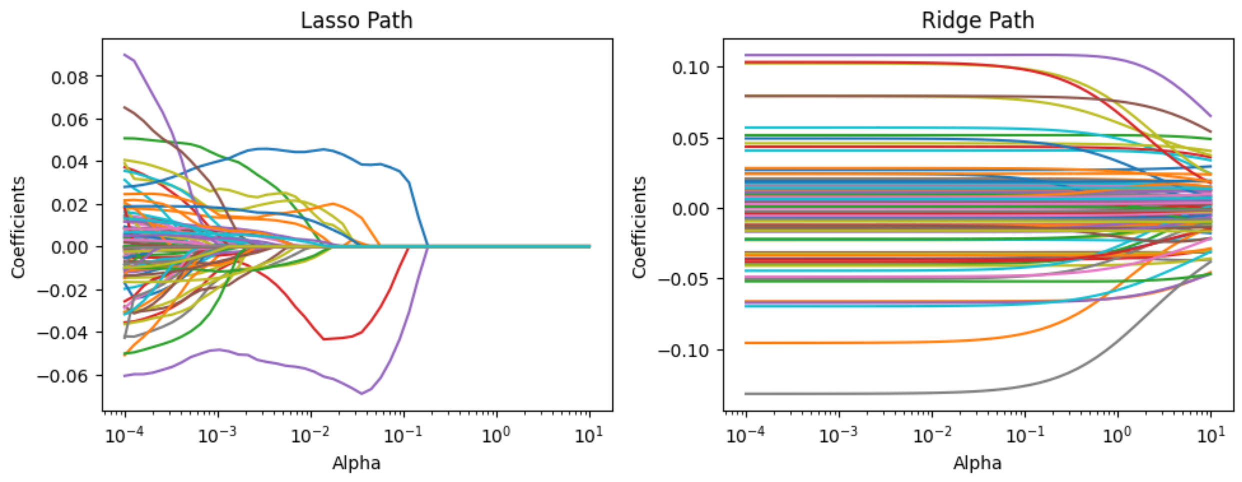
Part (ii)

```
import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso, ElasticNet, Ridge

alphas = np.logspace(-4, 1, 50)

lasso_coefs = []
ridge_coefs = []

for alpha in alphas:
    lasso = Lasso(alpha=alpha, max_iter=5000)
    lasso.fit(X_scaled, y)
    lasso_coefs.append(lasso.coef_)
```

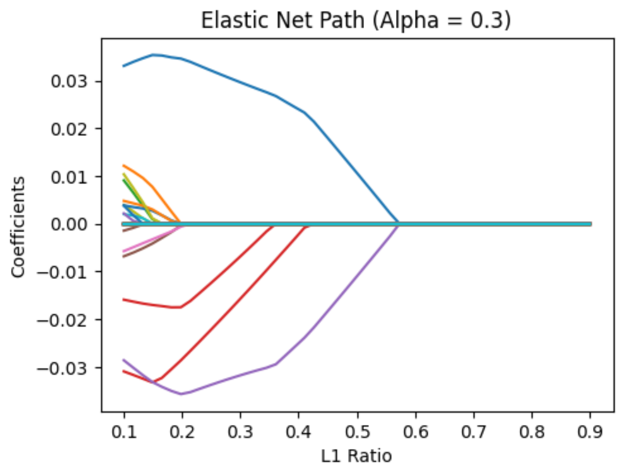
```python
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_scaled, y)
    ridge_coefs.append(ridge.coef_)

lasso_coefs = np.array(lasso_coefs)
ridge_coefs = np.array(ridge_coefs)
```

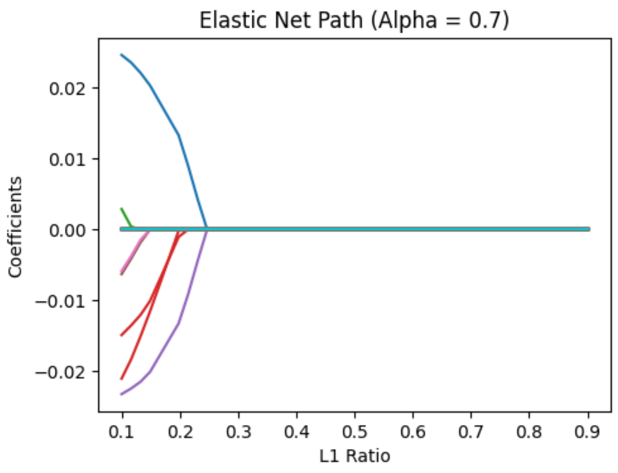
```python
alpha_1 = 0.3
alpha_2 = 0.7

l1_ratios = np.linspace(0.1, 0.9, 50)  # From mostly Ridge to mostly Lasso

elastic_net_coefs_1 = []
elastic_net_coefs_2 = []

for l1_ratio in l1_ratios:
    # Elastic Net with first alpha
    elastic_net_1 = ElasticNet(alpha=alpha_1, l1_ratio=l1_ratio, max_iter=5000)
    elastic_net_1.fit(X_scaled, y)
    elastic_net_coefs_1.append(elastic_net_1.coef_)

    # Elastic Net with second alpha
    elastic_net_2 = ElasticNet(alpha=alpha_2, l1_ratio=l1_ratio, max_iter=5000)
    elastic_net_2.fit(X_scaled, y)
    elastic_net_coefs_2.append(elastic_net_2.coef_)

elastic_net_coefs_1 = np.array(elastic_net_coefs_1)
elastic_net_coefs_2 = np.array(elastic_net_coefs_2)
```

```python
plt.figure(figsize=(10, 7))

# Lasso
plt.subplot(2, 2, 1)
plt.plot(alphas, lasso_coefs)
plt.xscale("log")
plt.xlabel("Alpha")
plt.ylabel("Coefficients")
plt.title("Lasso Path")

# Ridge
plt.subplot(2, 2, 2)
plt.plot(alphas, ridge_coefs)
plt.xscale("log")
plt.xlabel("Alpha")
plt.ylabel("Coefficients")
plt.title("Ridge Path")
plt.tight_layout()
plt.show()
```



```
# Plot Elastic Net regularization paths
plt.figure(figsize=(10, 7))

# Elastic Net (Alpha 1)
plt.subplot(2, 2, 1)
plt.plot(l1_ratios, elastic_net_coefs_1)
plt.xlabel("L1 Ratio")
plt.ylabel("Coefficients")
plt.title(f"Elastic Net Path (Alpha = {alpha_1})")

# Elastic Net (Alpha 2)
plt.subplot(2, 2, 2)
plt.plot(l1_ratios, elastic_net_coefs_2)
plt.xlabel("L1 Ratio")
plt.ylabel("Coefficients")
plt.title(f"Elastic Net Path (Alpha = {alpha_2})")

plt.tight_layout()
plt.show()
```





Part (iii)

We see that the top features are indeed different for each method. This is expected since each method fits different models or judges them according to different algorithms (e.g. best subsets and step-wise approaches examine variables in different collections and different orders). Also, different tuning parameters indeed yield different important features, as we in the Lasso regularization path (some coefficients are smaller and larger than others depending on the value of `alpha`). We chose tuning parameters using cross-validation.

The variable `racepctblack` appears across all selection methods. These variables appear in at least three methods:

```
racepctblack
MalePctDivorce
MedRent
PctKids2Par
PctPersDenseHous
RentLowQ
```

Hence it is reasonable to choose these as the most important features.

Part (b)

Part (i)

We create the `results` dictionary to keep track of the mean squared error (MSE) for each method. Notably, for each iteration, we split the data as instructed and scale `X_train`. We use the same scaling for `X_train` on `X_val` and `X_test`.

For any method that has tunable hyperparameters, on each iteration we use grid search to find a value of the hyperparameter that minimizes the MSE on the validation set, and then we use this value to fit the model and evaluate it on the test set. For RFE, the hyperparameter is the proportion of features we use.

```
###from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

num_iterations = 10
results = {
    "Least Squares": [],
    "Ridge": [],
    "Best Subsets": [],
    "RFE": [],
    "Lasso": [],
    "Elastic Net": []
}
```

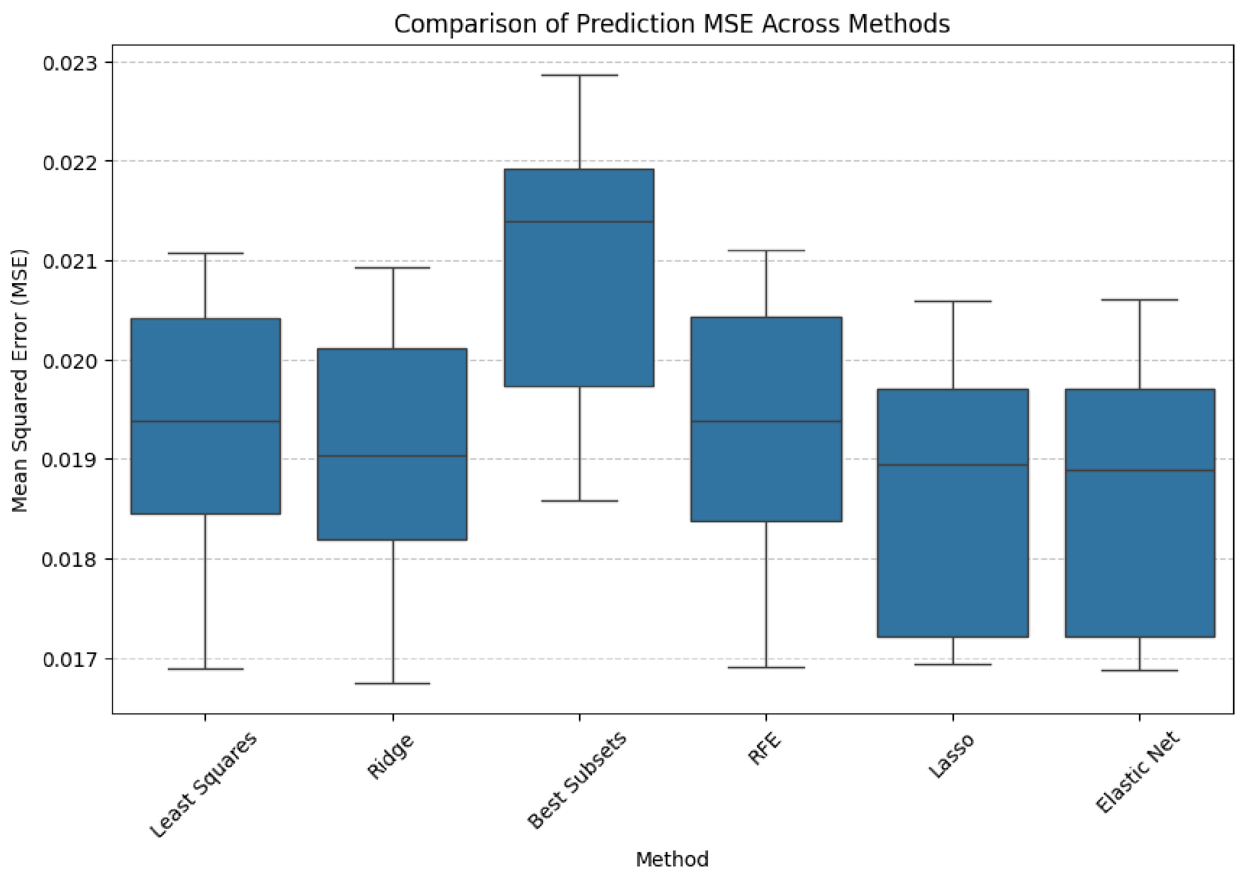
```
for _ in range(num_iterations):
    X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4)
    scaler_X_train = StandardScaler()
    X_train = pd.DataFrame(scaler_X_train.fit_transform(X_train),
columns=X.columns)

    X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5)
    X_val = pd.DataFrame(scaler_X_train.transform(X_val), columns=X_val.columns)
    X_test = pd.DataFrame(scaler_X_train.transform(X_test),
columns=X_test.columns)

    # Least Squares
    model_ls = LinearRegression().fit(X_train, y_train)
    y_pred_ls = model_ls.predict(X_test)
    results["Least Squares"].append(mean_squared_error(y_test, y_pred_ls))

    # Ridge Regression
    best_alpha = min(
        np.arange(0.1, 1.1, 0.1),
        key=lambda a: mean_squared_error(y_val, Ridge(alpha=a).fit(X_train,
y_train).predict(X_val))
    )
    model_ridge = Ridge(alpha=best_alpha).fit(X_train, y_train)
    y_pred_ridge = model_ridge.predict(X_test)
    results["Ridge"].append(mean_squared_error(y_test, y_pred_ridge))

    # Best Subsets
    best_features = best_subset(X_significant, y)
    model_best_subset = LinearRegression().fit(X_train[list(best_features)],
y_train)
    y_pred_best_subset = model_best_subset.predict(X_test[list(best_features)])
    results["Best Subsets"].append(mean_squared_error(y_test,
y_pred_best_subset))
```

```
    # Recursive Feature Elimination
    def evaluate_prop(prop):
        selector = RFE(LinearRegression(), n_features_to_select=prop)
        selector.fit(X_val, y_val)
        selected_features = X_val.columns[selector.support_]

        model_val = LinearRegression().fit(X_val[selected_features], y_val)
        y_pred_val = model_val.predict(X_val[selected_features])

        return mean_squared_error(y_val, y_pred_val)

    best_prop = min(np.arange(0.1, 1, 0.1), key=evaluate_prop)
    rfe = RFE(LinearRegression(), n_features_to_select=best_prop)
    selector = rfe.fit(X_train, y_train)
    selected_features = X_train.columns[selector.support_]

    model_rfe = LinearRegression().fit(X_train[selected_features], y_train)
    y_pred_rfe = model_rfe.predict(X_test[selected_features])
    results["RFE"].append(mean_squared_error(y_test, y_pred_rfe))

    # Lasso
    best_alpha = min(
        np.logspace(-4, 1, 10),
        key=lambda a: mean_squared_error(y_val, Lasso(alpha=a,
max_iter=10000).fit(X_train, y_train).predict(X_val))
    )
    model_lasso = Lasso(alpha=best_alpha, max_iter=10000).fit(X_train, y_train)
    y_pred_lasso = model_lasso.predict(X_test)
    results["Lasso"].append(mean_squared_error(y_test, y_pred_lasso))

    # Elastic Net
    alpha_values = np.logspace(-4, 1, 10)
    l1_ratio_values = np.arange(0.1, 1.1, 0.1)
    best_alpha, best_l1_ratio = min(
        [(a, l1) for a in alpha_values for l1 in l1_ratio_values],
        key=lambda params: mean_squared_error(
            y_val, ElasticNet(alpha=params[0], l1_ratio=params[1],
max_iter=10000).fit(X_train, y_train).predict(X_val)
        )
    )
    model_elastic = ElasticNet(alpha=best_alpha, l1_ratio=best_l1_ratio,
max_iter=10000).fit(X_train, y_train)
    y_pred_elastic = model_elastic.predict(X_test)
    results["Elastic Net"].append(mean_squared_error(y_test, y_pred_elastic))
```

Part (ii)

The results are visualized in the boxplots below.

```python
import seaborn as sns

mse_data = []
for method, mse_values in results.items():
    for mse in mse_values:
        mse_data.append({"Method": method, "MSE": mse})

df_mse = pd.DataFrame(mse_data)

plt.figure(figsize=(10, 6))
sns.boxplot(x="Method", y="MSE", data=df_mse)
plt.xticks(rotation=45)
plt.title("Comparison of Prediction MSE Across Methods")
plt.ylabel("Mean Squared Error (MSE)")
plt.xlabel("Method")
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```



Part (iii)

From the plot above, we see that Lasso and Elastic Net performed the best, with most other methods slightly behind. Best Subsets performed significantly worse, suggesting it may be overfitting. This may be the result of having to cut down on the number of variates used to fit the model, as otherwise the model would be computationally intractable to fit. Hence this result is not very surprising. Note that these results are sensitive to the choice of hyperparameters, so yours may look quite different. When initially running this, Lasso performed the worst because I was only searching through $\alpha \in [0.1, 1]$.

Note that Lasso and Elastic Net performed almost identically and also chose the same set of variables (see part (a), part (i)). Interestingly, least squares and RFE performed similarly but chose different variables.

Based on the test set MSE, it appears Lasso and Elastic Net are the overall best methods for prediction on this dataset.

Problem 2

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
np.set_printoptions(precision=4)
```

(a)

Empirical Demonstration

```
np.random.seed(1234)
n = 500
p = 10

X = np.random.normal(size = (n, p))
beta = np.random.normal(size=p)
beta_0 = np.random.normal()
y = X @ beta + beta_0 + np.random.normal(size=n)

reg_int = LinearRegression().fit(X, y)
print(reg_int.coef_)
print(f'{reg_int.intercept_:.4f}')
```

```
[ 1.3536  0.0102 -0.8053  1.246   1.3494  0.1093  0.7609 -0.1995  0.8226
 -0.6918]
0.2307
```

(i)

```
yc = y - np.mean(y)
Xc = X - np.mean(X, axis=0)
reg_no_int = LinearRegression(fit_intercept=False).fit(Xc, yc)
print(reg_int.coef_)
print(f'{reg_int.intercept_:.4f}')
```

```
[ 1.3536  0.0102 -0.8053  1.246   1.3494  0.1093  0.7609 -0.1995  0.8226
 -0.6918]
0.2307
```

(ii)

```
X0 = np.hstack((np.ones((n, 1)), X))
reg_0 = LinearRegression(fit_intercept=False).fit(X0, y)
print(reg_int.coef_)
print(f'{reg_int.intercept_:.4f}')
```

```
[ 1.3536  0.0102 -0.8053  1.246   1.3494  0.1093  0.7609 -0.1995  0.8226
 -0.6918]
0.2307
```

Mathematical Demonstration

(i)

Let $\mathbf{X} \in \mathbb{R}^{n\times p}$ be the original covariate matrix, and $\mathbf{X}^0 \in \mathbb{R}^{n\times p}$ be the covariate matrix after centering each column, i.e. $\mathbf{X}^0_{\cdot,i} = \mathbf{X}_{\cdot,i} - \bar{\mathbf{X}}_{\cdot,i}$. Let $y \in \mathbb{R}^n$ be the original response vector, and $y^0 \in \mathbb{R}^n$ be the response vector after centering, i.e. $y^0 = y - \bar{y}$.

By taking the derivative of the least square of the original model, we have the normal equation,

$$\frac{\partial \mathrm{RSS}}{\partial \beta_0} = -2\sum_{i=1}^{n}(y_i - \beta_0 - \mathbf{x}_i^\top \beta) = 0,$$

from which we can obtain,

$$\hat{\beta}_0 = \bar{y} - \bar{\mathbf{X}}\hat{\beta}.$$

By substituting this formula into the original least squares equation, we can obtain,

$$RSS = \left\|y - \bar{y} - \left(\mathbf{X} - \bar{\mathbf{X}}\right)\beta\right\|_2 = \left\|y^0 - \mathbf{X}^0\beta\right\|_2.$$

Therefore, the centered RSS equals to the original RSS for β. Hence, $\beta^0 = \beta$, i.e. they have the same linear regression coefficients. $$

(ii)

Let $\mathbf{X} \in \mathbb{R}^{n\times p}$ be the original covariate matrix, and $\tilde{\mathbf{X}} \in \mathbb{R}^{n\times(p+1)}$ be the covariate matrix with an additional column of 1s. Let $y \in \mathbb{R}^n$ be the original response vector.

The model for fitting linear regression with an intercept is,

$$\begin{aligned} y &= \mathbf{X}\beta + \beta_0 1 + \epsilon \\ &= \begin{bmatrix} 1 & \mathbf{X} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta \end{bmatrix} + \epsilon \\ &= \tilde{\mathbf{X}}\tilde{\beta} + \epsilon, \end{aligned}$$

where we define$\tilde{\beta} = [\beta_0, \beta^\top]^\top$, the second equality is due to block matrix multiplication. Since the model is the same, and we both have the same algorithm OLS that can give unique optimal to solve the least squares problem, the solution is the same.

(b)

Empirical Demonstration

```python
np.random.seed(1234)
n = 100
p = 200

X = np.random.normal(size = (n, p))
beta = np.random.normal(size=p)
beta_0 = np.random.normal()
y = X @ beta + beta_0 + np.random.normal(size=n)
reg_b = LinearRegression().fit(X, y)
print(reg_b.score(X, y))
```

```
1.0
```

Mathematical Demonstration

Let $y = \mathbf{X}\beta$, there is in total n equations and p unknowns. Since $p > n$, the system is underdetermined, i.e. there exists β such that the above equality holds.

Choose $\hat{\beta}$ to satisfy the system of linear equations. Then, one can see that RSS $= \sum_{i=1}^{n} \left\| y - \mathbf{X}\hat{\beta} \right\|_2^2 = 0$. Since this definitely minimizes the RSS, $\hat{\beta}$ is the OLS solution. Therefore, when $p > n$, there is 0 training error.

(c)

Empirical Demonstration

(i)

First, for independent features, under the same generation procedure, we can see that the average mse over 100 iterations would be

```python
np.random.seed(1234)
n = 100
p = 50
m = 100

MSE_ind = np.zeros(m)
for i in range(m):
    X = np.random.normal(size = (n, p))
    np.random.seed(i)
    beta = np.random.normal(size=p)
    beta_0 = np.random.normal()
    y = X @ beta + beta_0 + np.random.normal(size=n)
    reg = LinearRegression().fit(X, y)
    MSE_ind[i] = np.mean((y - reg.predict(X))**2)
```

For correlated features,

```python
np.random.seed(1234)

def gen_cov_mat(p, lam = 2):
    A = np.random.normal(size = (p, p))
    Q, R = np.linalg.qr(A)
    Lambda = np.diag(np.random.poisson(lam, p))
    return Q @ Lambda @ Q.T

MSE_cor = np.zeros(m)
for i in range(m):
    X = np.random.normal(size = (n, p))
    np.random.seed(i)
    beta = np.random.normal(size=p)
    beta_0 = np.random.normal()
    y = X @ beta + beta_0 + np.random.normal(size=n)
    cov_mat = gen_cov_mat(p)
    X = X @ cov_mat
    reg = LinearRegression().fit(X, y)
    MSE_cor[i] = np.mean((y - reg.predict(X))**2)
```

```
print(np.mean(MSE_ind))
print(np.mean(MSE_cor))
```

```
0.5036956212191067
4.492030655352779
```

Hence, linear regression has high variance for correlated features.

(ii)

```
## Generate p = 16 features with 4 sets of 4 highly dependent features.

np.random.seed(1234)
n = 500
p = 16
p0 = 4
Sigma0 = np.full((p0, p0), 0.9) + np.diag([0.1]*p0)
Sigma = np.zeros((p, p))

for i in range(4):
    start = i * p0
    end = (i + 1) * p0
    Sigma[start:end, start:end] = Sigma0

X = np.random.multivariate_normal(
    mean=np.zeros(p),
    cov=Sigma,
    size=n
)
beta = np.arange(p) + 1
beta_0 = np.random.normal()
y = X @ beta + beta_0 + np.random.normal(size=n)

reg_ridge = Ridge(alpha=70).fit(X, y)
print(reg_ridge.coef_)
```

```
[ 1.9401   2.107    2.4772   2.8642   5.615    6.0723   6.4873   6.8849   9.2762
  9.8143 10.6545 10.7195 13.528   13.8172 14.1102 14.4599]
```

From the estimated coefficients, one can see that highly correlated features tend to have the same coefficient, even though on the oracle level, they are different.

(iii)

```
## Generate p = 16 features with 4 sets of 4 highly dependent features.

np.random.seed(1236)
n = 500
p = 16
p0 = 4
Sigma0 = np.full((p0, p0), 0.9) + np.diag([0.1]*p0)
Sigma = np.zeros((p, p))

for i in range(4):
    start = i * p0
    end = (i + 1) * p0
    Sigma[start:end, start:end] = Sigma0

X = np.random.multivariate_normal(
    mean=np.zeros(p),
    cov=Sigma,
    size=n
)
beta = np.tile((np.arange(1, p0+1) * 3), p0)
beta_0 = np.random.normal()
y = X @ beta + beta_0 + np.random.normal(size=n)

reg_lasso = Lasso(alpha=27).fit(X, y)
print(reg_lasso.coef_)
```

```
[0.      0.      0.      4.1418 0.      0.      1.0051 0.      0.      0.
 0.5667 0.      0.      0.      0.      3.6414]
```

One can see that for each set of highly correlated features, only one of them is selected by LASSO.

Mathematical Demonstration

(i)

Without loss of generality, suppose each column of $\mathbf{X}$ is centered at 0, and are standardized so the variance of each column is 1. For simplicity, consider $p = 3$, and let the correlation between X_1 and X_2 be ρ, and they are independent of X_3. The covariance matrix for $\mathbf{X}$ is,

$$\mathbf{X}^\top \mathbf{X} = \begin{bmatrix} 1 & \rho & 0 \\ \rho & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Therefore, the covariance for estimated $\hat{\beta}$ will be,

$$\text{Cov}(\hat{\beta}) = \sigma^2 \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} = \frac{\sigma^2}{1-\rho^2} \begin{bmatrix} 1 & -\rho & 0 \\ -\rho & 1 & 0 \\ 0 & 0 & 1-\rho^2 \end{bmatrix}$$

Therefore, for marginal distribution of $\hat{\beta}_1$, the variance is,

$$\operatorname{Var}\left(\hat{\beta}_1\right) = \frac{\sigma^2}{1-\rho^2},$$

while X_3 maintains a variance of 1.

With the increase of ρ, the variance increases. Hence, for correlated features, linear regression has higher variance.

(ii)

The estimated coefficients for ridge regression with regularization λ is,

$$\hat{\beta} = \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}\right)^{-1} \mathbf{X}^\top y.$$

When λ is sufficiently large, $\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}$ is dominated by the diagonal term, and the expectation of the estimated coefficients is,

$$\mathbb{E}\hat{\beta} \approx \frac{1}{\lambda} \mathbf{X}^\top \mathbf{X} \beta,$$

where β is the oracle parameter. Then, for each entry of $\mathbb{E}\hat{\beta}$, it is

$$\left(\mathbb{E}\hat{\beta}\right)_i = \frac{1}{\lambda} \sum_j \mathbf{X}_{:,i}^\top \mathbf{X}_{:,j} \beta_j.$$

Without loss of generality, consider every feature is highly correlated with the same correlation ρ. Then, we have

$$\left(\mathbb{E}\hat{\beta}\right)_i = \frac{\rho}{\lambda} \|\mathbf{X}_{:,i}\| \sum_j \|\mathbf{X}_{:,j}\| \beta_j.$$

One can then see that if each feature is the same in terms of L_2 norm, then each estimated coefficient is the same. When there are multiple mutually independent sets of highly dependent features, each set has approximately the same estimated coefficient returned by ridge regression following the same reasoning.

(iii)

It could be realized by analyzing the projected gradient or subgradient. If someone prove it correctly, they can earn 5 extra bonus points.