CMSC 35400 / STAT 37710

Spring 2020

Homework 4

You must clearly indicate where your solutions to individual subproblems are in gradescope. If you force the graders to do this for you, points will be deducted from your total.

1. **(Empirical Risk Minimization)** In this question, we will investigate the trade-off between approximation error and estimation error given different collections of predictors. Use the following code to generate a dataset:

```
#python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

def data_generator(n_samples):
    x = np.random.uniform(-10, 10, n_samples)
    y = np.cos(0.5 + np.exp(-x)) + 1/(1 + np.exp(-x))
    noise = np.random.normal(0, 0.01, n_samples)
    y += noise
    return x, y

complete_X, complete_Y = data_generator(5000)
train_X, train_Y = complete_X[:100], complete_Y[:100]
large_X, large_Y = complete_X[100:], complete_Y[100:]

loss_func = mean_squared_error
```

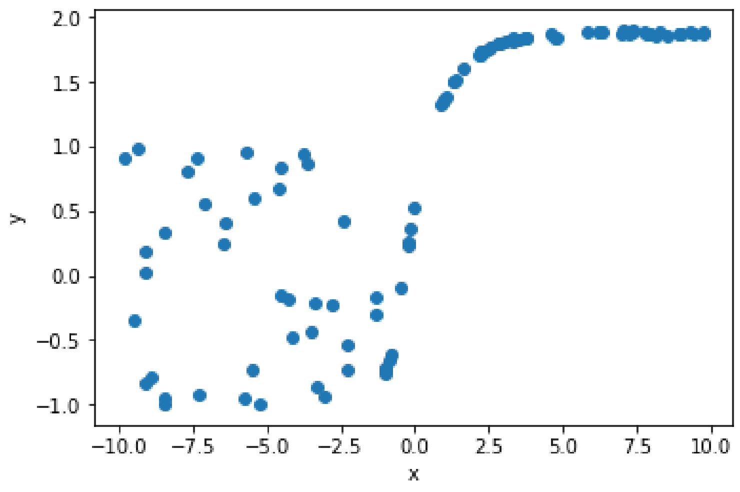
Use train_X and train_Y as training samples for ERM. large_X and large_Y are for the approximation of true data distribution of X and Y, in order to estimate true risk. A plot of a small portion of the dataset.

Use mean squared error as loss function in this problem, unless noted otherwise.

a) Let's first define $\mathcal{H}_k$ to be a collection of all possible polynomial functions of degree k. Implement the ERM process to select the predictor $\hat{h} \in \mathcal{H}_k$ with the lowest empirical risk. (Hint: polyfit function in numpy could be useful.)

SOLUTION:

```
#python
def evaluate(clf, test_x, test_y, loss_func):
    y_hat = clf(test_x)
    loss = loss_func(test_y, y_hat)
    return loss
```

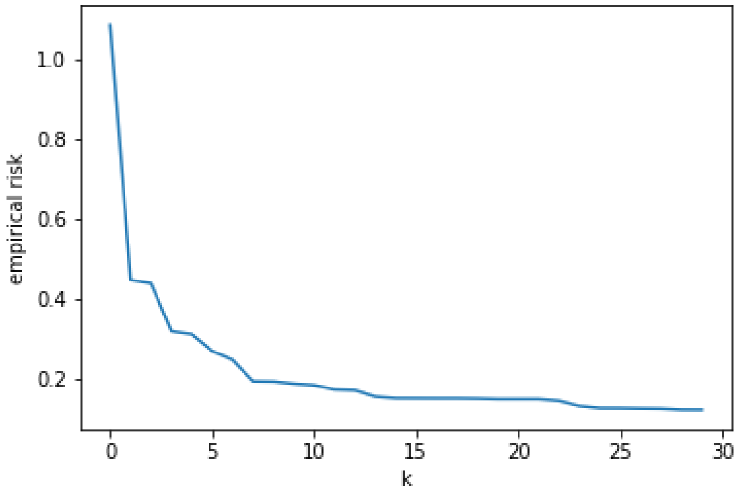


```
def erm(k, train_x, train_y, loss_func):
    clf = np.polyfit(train_x, train_y, k)
    clf = np.poly1d(clf)
    empirical_loss =
        evaluate(clf, train_x, train_y, loss_func)
    return clf, empirical_loss
```

b) Experiment with the ERM you built with k of $\mathcal{H}_k$ ranging from 0 to 30. Report empirical loss and plot a graph of empirical risk v.s. k.

SOLUTION:

```
#python
empirical_loss_list = []
loss_func = mean_squared_error
for k in range(30):
    _, emp_loss = erm(k, train_X, train_Y, loss_func)
    empirical_loss_list += emp_loss,

plt.clf()
x = range(30)
plt.plot(x, empirical_loss_list)
plt.show()
```



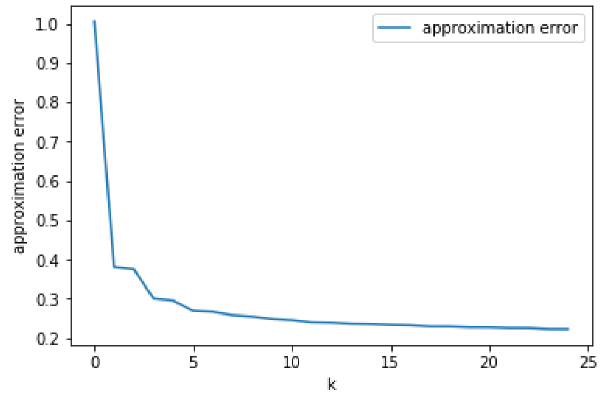
c) We will further explore the approximation vsėstimation trade-off. First, use the noise-free distribution in data_generator to estimate R^* with the **complete dataset**. i.e., the complete dataset has noisy (x, y) pairs and we know that without noise

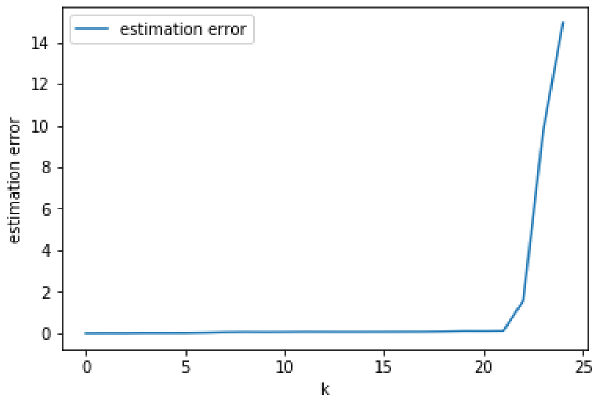
$$y^* := \mathbb{E}[Y|X = x] = \cos(0.5 + e^{-x}) + \frac{1}{1 + e^{-x}}. \tag{1}$$

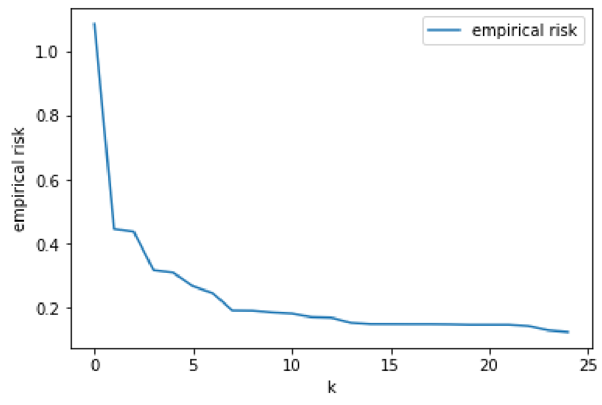
(Note that in real applications, you normally do not have access to the true distribution of X and Y.) Now use large_X, large_Y to estimate the Bayes risk R^*, the risk of the ERM for each k, the estimation error, and the approximation error. Plot graphs of these errors v.s. k. Experiment with k range from 0 to 25. (Note: these different errors might not be in the same scale. You can plot one graph for each error v.s. k)

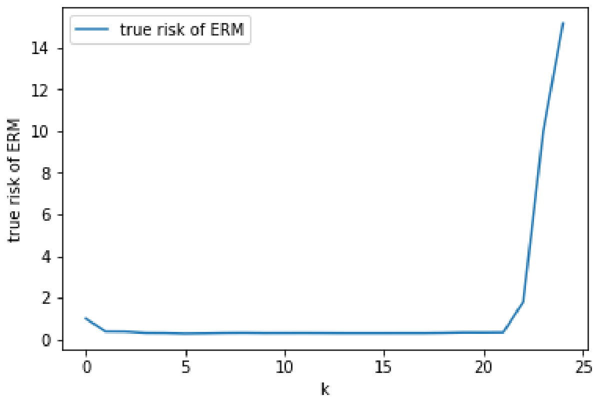
SOLUTION:

```
#python
def compute_r_star(x, y, loss_func):
    y_hat = np.cos(0.5 + np.exp(-x)) + 1/(1 + np.exp(-x))
    risk = loss_func(y, y_hat)
    return risk

r_star = compute_r_star(complete_X, complete_Y, loss_func)

emp_risk_list = []
estimation_list = []
approximation_list = []
real_risk_list = []
erm_risk_list = []
max_k = 25

for k in range(max_k):
    clf, empirical_risk = erm(k, train_X, train_Y, loss_func)
    emp_risk_list += empirical_risk,

    erm_risk = evaluate(clf, large_X, large_Y, loss_func)
    erm_risk_list += erm_risk,

    clf, real_risk = erm(k, large_X, large_Y, loss_func)

    estimation_error = erm_risk - real_risk
    approximation_error = real_risk - r_star

    estimation_list += estimation_error,
    approximation_list += approximation_error,
    real_risk_list += real_risk,
```



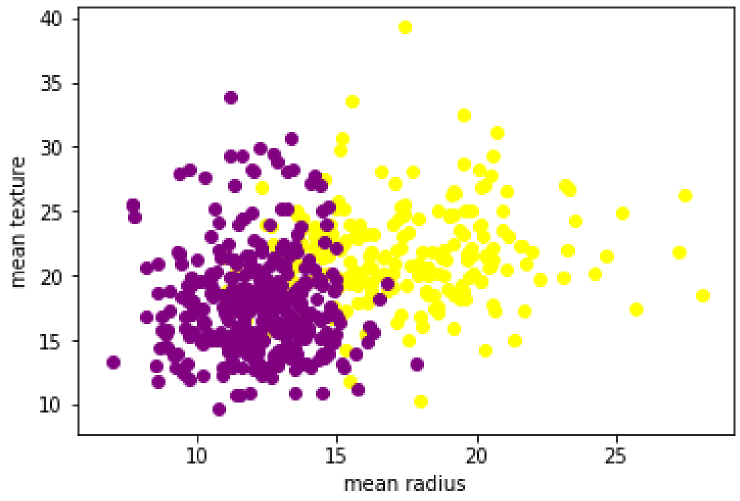






2. **(Decision Tree and Pruning)** In this question, we will solve a classification task with the breast cancer dataset. Use the following code to load dataset. We will use only first two features in this question.

```
#python
from sklearn.datasets import load_breast_cancer
import  matplotlib.pyplot  as plt

X, y = load_breast_cancer(return_X_y=True)
X = X[:, :2]
plt.clf()
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```



a) Reserve 25% of the samples as test data, and train a decision tree using the rest without any regularization. Report your training accuracy and test accuracy. (Hint: you can use DecisionTreeClassifier from sklearn package, and train_test_split to split data.)

SOLUTION:

```
#python
from sklearn.tree import DecisionTreeClassifier

X_train, X_test, y_train, y_test = train_test_split(X, y)
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

train_acc = clf.score(X_train, y_train)
test_acc = clf.score(X_test, y_test)
```

training acc: 100%, test acc: 87.4%

b) Now implement pruning process as described in the lecture slides. Recall that we want to minimize

$$C(\mathcal{T}) = -\sum_{v\in\mathcal{T}} L(S_v) + \lambda|\mathcal{T}| \tag{2}$$

where $|\mathcal{T}|$ is the number of leaves in tree $\mathcal{T}$. Experiment with different regularization weight λ, and report the best accuracy and its corresponding λ. (Hint: You can use ccp_alpha in DecisionTreeClassifier to do the pruning. sklearn version ≥ 0.22 required.)

SOLUTION:

```
#python
lam = np.arange(0, 0.5, 0.001)
rst = []
train_accs = []
accs = []

for alpha in lam:
    clf = DecisionTreeClassifier(ccp_alpha=alpha)
    clf.fit(X_train, y_train)
    rst.append(clf)
    acc = clf.score(X_test, y_test)
    accs.append(acc)
    train_acc = clf.score(X_train, y_train)
    train_accs.append(train_acc)

best = max(accs)
best_ind = accs.index(best)
best_lam = lam[best_ind]
```

Best accuracy: 91.61%, best λ: 0.004 (optimal lambda might not match exactly. Pruning algorithm in sklearn is slightly different from what lecture slides describe.)

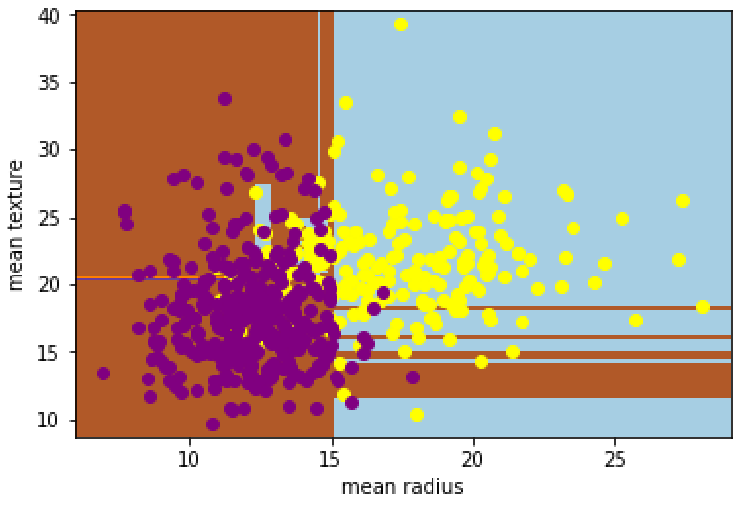
c) Following code plots decision regions of a trained decision tree on breast cancer data.

```
# Python
def plot_regions(tree):
    plot_colors = ['yellow', 'purple']
    plot_step = 0.02

    breast_cancer = load_breast_cancer()
    X = breast_cancer.data
    y = breast_cancer.target

    plt.clf()

    idx = np.arange(X.shape[0])

    np.random.shuffle(idx)
    X = X[idx]
    y = y[idx]

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                         np.arange(y_min, y_max, plot_step))

    Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)

    plt.xlabel(breast_cancer.feature_names[0])
    plt.ylabel(breast_cancer.feature_names[1])
    plt.axis("tight")

    for i, color in zip(range(2), plot_colors):
            idx = np.where(y == i)
            plt.scatter(X[idx, 0], X[idx, 1], c=color,
                label=breast_cancer.target_names[i],
                cmap=plt.cm.Paired)
```

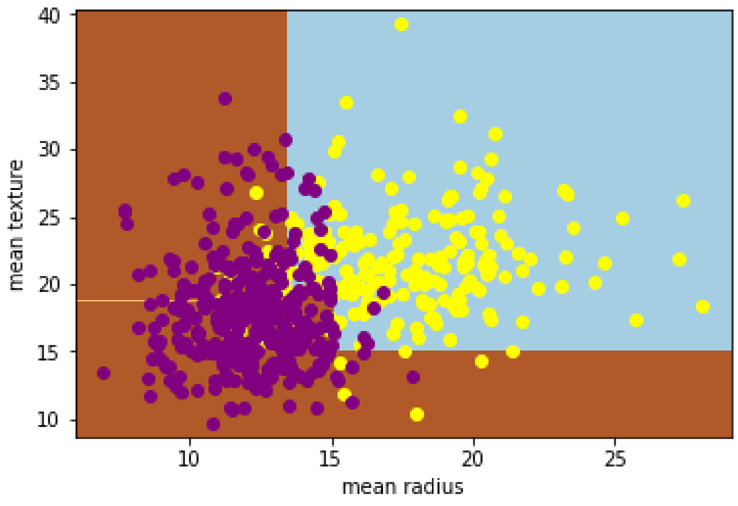
Use the code to plot decision regions of your decision tree from a) and the best regularized tree from b).

SOLUTION:

Decision tree without regularization:



Decision tree with $\lambda = 0.004$:

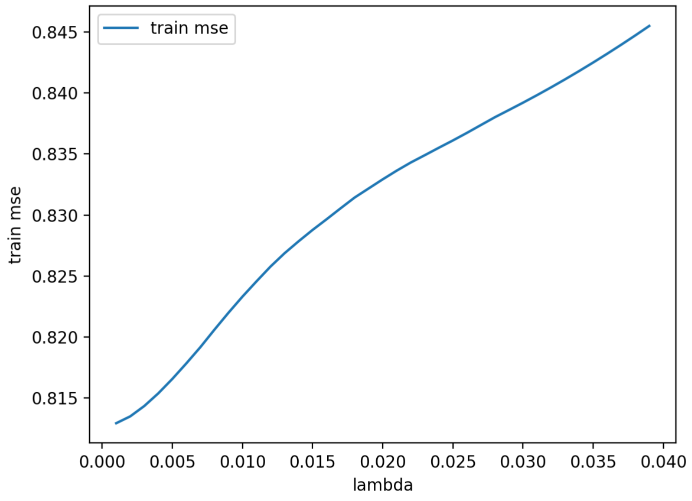


3. **(Lasso estimate)**In this question, you are required to fit data with a Lasso regression. Recall that the Lasso objective is to minimize RSS(β) + $\lambda \sum_i \beta_i$. Following the script below here, you will be able to generate the training and testing data.

```
#python
import numpy as np
np.random.seed(0)
N_fold = 10
N_test = 500
N_train = 1000
N = N_test + N_train
# Specify feature dimensions of X and Y
X_dim = 20
Y_dim = 10
X = np.random.randn(N,X_dim)

# Only have 10 non-zero entries in beta,
nnz = 10
beta = np.zeros((X_dim * Y_dim))
nnz_idx = np.random.choice(X_dim * Y_dim, nnz, replace = False)
beta[nnz_idx] = np.random.randn(nnz) * 2

beta = beta.reshape(X_dim, Y_dim)
Y =  X @ beta + np.random.rand(N, Y_dim)

# Split training and testing set
X_test = X[:N_test]
Y_test = Y[:N_test]
```

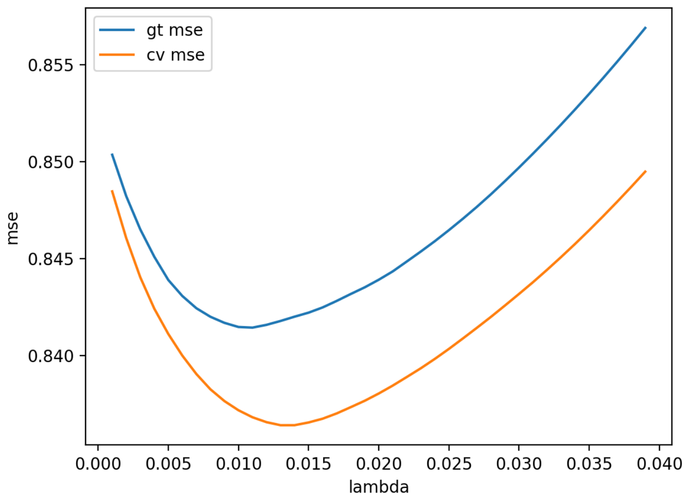
```
X_train = X[N_test:]
Y_train = Y[N_test:]
```

a) Write a function to fit the Lasso regression on the training data and calculate the MSE on the training set. Choosing λ from 0 to 0.04 (with a step of 0.001), compute the estimate $\hat{y}$ for different values λ, and plot the MSE as a function of λ.
SOLUTION:



```
#python
def train_and_eval(X_train, Y_train, X_eval, Y_eval,
lambda_):
    clf = linear_model.Lasso(alpha=lambda_)
    clf.fit(X_train, Y_train)
    Y_eval_pred = clf.predict(X_eval)
    mse = ((Y_eval - Y_eval_pred) ** 2).sum(axis = -1).mean(
    axis = 0)
    return mse

weight_list = np.arange(40)[1:] * 0.001
result_list = []
for weight in weight_list:
    test_mse = train_and_eval(X_train,
    Y_train, X_train, Y_train, weight)
    result_list.append([test_mse, weight])
result_array = np.array(result_list)
plt.figure()
plt.plot(result_array[:,-1], result_array[:,0], label =
'train␣mse')
plt.xlabel('lambda')
plt.ylabel('train␣mse')
plt.legend()
plt.show()
```

b) Implement 10-fold cross validation on the training set to select λ. Plot and compare the MSE on the hold-out set with the true MSE which is computed on the test set. And see how we get to finding the "best" λ.
SOLUTION:



```
#python
weight_list = np.arange(40)[1:] * 0.001
result_list = []
for weight in weight_list:
    cv_eval_mse_list = []
    for i in range(N_fold):
        sidx = i * int(N_train / N_fold)
        eidx = (i+1)* int(N_train / N_fold)
        x_cv_eval = X_train[sidx:eidx]
        y_cv_eval = Y_train[sidx:eidx]
        x_cv_train = np.concatenate([X_train[:sidx],
        X_train[eidx:]],axis = 0)
        y_cv_train = np.concatenate([Y_train[:sidx],
        Y_train[eidx:]],axis = 0)
        eval_mse = train_and_eval(x_cv_train, y_cv_train,
        x_cv_eval, y_cv_eval, weight)
        cv_eval_mse_list.append(eval_mse)
    test_mse = train_and_eval(X_train, Y_train, X_test,
    Y_test, weight)
    result_list.append([np.array(cv_eval_mse_list).mean(),
    test_mse, weight])
result_array = np.array(result_list)
plt.figure()
plt.plot(result_array[:,-1], result_array[:,1], label =
'gt␣mse')
plt.plot(result_array[:,-1], result_array[:,0], label =
'cv␣mse')
plt.xlabel('lambda')
plt.ylabel('mse')
plt.legend()
plt.show()
```