In [1]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt

   import sklearn.preprocessing as skl_pre
   import sklearn.linear_model as skl_lm
   import sklearn.discriminant_analysis as skl_da
   import sklearn.neighbors as skl_nb

1 Bias-variance trade-off, model selection and cross validation – Computer exercises

1.1 6.1 Cross validation in \( k \)-NN

In this exercise we will return to the Biopsy data set also used in Exercise 4.1 (Lesson 4). We will try to determine suitable value of \( k \) in \( k \)-NN for this data. For simplicity, we will only consider the three attributes in columns \( V_3, V_4 \) and \( V_5 \) in this problem.

1.1.1 (a)

Consider all data as training data. Investigate how the training error varies with different values of \( k \) (hint: use a for-loop). Which \( k \) gives the best result? Is it a good choice of \( k \)?

In [2]: np.random.seed(1)
   biopsy = pd.read_csv('Data/biopsy.csv',
   dtype={'ID': str}).dropna().reset_index(drop=True)
   X = biopsy[['V3', 'V4', 'V5']]
   Y = biopsy['class']

   misclassification = []

   for k in range(200): # Try \( n_{neighbors} = 1, 2, \ldots, 50 \)
       model = skl_nb.KNeighborsClassifier(n_neighbors=k+1)
       model.fit(X, Y)
       prediction = model.predict(X)
       misclassification.append(np.mean(prediction != Y))
\( K = \text{np.linspace}(1, 200, 200) \)
\( \text{plt.plot}(K, \text{misclassification}) \)
\( \text{plt.title('training error for kNN')} \)
\( \text{plt.xlabel('k')} \)
\( \text{plt.ylabel('training error')} \)
\( \text{plt.show()} \)

# \( k = 1 \) has the best performance. That is, however, expected and is not a good measure of which \( k \) should be used.

1.1.2 (b)

Split the data randomly into a training and validation set, and see how well you perform on the validation set. (Previously, we have used the terminology "training" and "test" set. If the other set (not the training set) is used to make design decisions, such as choosing \( k \), it is really not a test set, but rather a "validation" set. Hence the terminology.) Which \( k \) gives the best result?

In [3]: # sampling indices for training
   trainIndex = \text{np.random.choice(biopsy.shape[0], size=600, replace=False)}
   train = biopsy.iloc[trainIndex] # training set
   validation = biopsy.iloc[-biopsy.index.isin(trainIndex)]

   \( X_{\text{train}} = \text{train[['V3', 'V4', 'V5']]} \)
   \( Y_{\text{train}} = \text{train['class']} \)
   \( X_{\text{validation}} = \text{validation[['V3', 'V4', 'V5']]} \)
\[ Y_{\text{validation}} = \text{validation['class']} \]

\[ \text{misclassification} = [\] \]

\[
\text{for } k \text{ in range}(200): \quad \# \text{Try } n\_\text{neighbors} = 1, 2, \ldots, 50
\]
\[
\text{model} = \text{skl_nb.KNeighborsClassifier}(n\_\text{neighbors}=k+1) \# \text{loops starts from 0 in python}
\]
\[
\text{model.fit}(X\_\text{train}, Y\_\text{train})
\]
\[
\text{prediction} = \text{model.predict}(X\_\text{validation})
\]
\[
\text{misclassification.append}(\text{np.mean}((\text{prediction} \neq Y\_\text{validation})))
\]

\[ K = \text{np.linspace}(1, 200, 200) \]
\[ \text{plt.plot}(K, \text{misclassification}) \]
\[ \text{plt.title('validation set error for kNN')} \]
\[ \text{plt.xlabel('k')} \]
\[ \text{plt.ylabel('validation error')} \]
\[ \text{plt.show()} \]

\# $k$ between 5-20 or 50-125 seems good, but the result seems noisy and hard to interpret.

1.1.3 (c)

Perform (b) 10 times for different validation sets and average the result. Which $k$ gives the best result?
In [4]: misclassification = np.zeros((10, 200))

    for i in range(10):
        trainIndex = np.random.choice(biopsy.shape[0], size=600, replace=False)
        train = biopsy.iloc[trainIndex]  # training set
        validation = biopsy.iloc[-biopsy.index.isin(trainIndex)]

        X_train = train[['V3', 'V4', 'V5']]
        Y_train = train['class']
        X_validation = validation[['V3', 'V4', 'V5']]
        Y_validation = validation['class']

        for k in range(200):  # Try n_neighbors = 1, 2, ..., 50
            model = skl_nb.KNeighborsClassifier(n_neighbors=k+1)
            model.fit(X_train, Y_train)
            prediction = model.predict(X_validation)
            misclassification[i, k] = (np.mean(prediction != Y_validation))

    average_mis = np.mean(misclassification, axis=0)

    K = np.linspace(1, 200, 200)
    plt.plot(K, average_mis)
    plt.title('cross validation error for kNN')
    plt.xlabel('k')
    plt.ylabel('validation error')
    plt.show()

    # The result shows that k between 5 and 60 perform the best.
1.1.4 (d)

Perform 10-fold cross-validation by first randomly permute the data set, divide the data set into 10 equally sized parts and loop through them by taking one part as validation set and the rest as training set each time. Which \( k \) gives the best result?

In [5]: randomize_indices = np.random.choice(biopsy.shape[0], biopsy.shape[0], replace=False)
misclassification = np.zeros((10, 200))

for i in range(10):
    n = np.ceil(biopsy.shape[0]/10)  # number of samples in each fold
    validationIndex = np.arange(i*n, min(i*n+n, biopsy.shape[0]), 1).astype('int')
    randomize_validationIndex = randomize_indices[validationIndex]
    train = biopsy.iloc[-biopsy.index.isin(randomize_validationIndex)]
    validation = biopsy.iloc[randomize_validation_index]

    X_train = train[['V3', 'V4', 'V5']]
    Y_train = train['class']
    X_validation = validation[['V3', 'V4', 'V5']]
    Y_validation = validation['class']

    for k in range(200):  # Try n_neighbors = 1, 2, ...., 50
        model = skl_nb.KNeighborsClassifier(n_neighbors=k+1)
        model.fit(X_train, Y_train)
prediction = model.predict(X_validation)
misclassification[i, k] = (np.mean(prediction != Y_validation))

average_mis = np.mean(misclassification, axis=0)

K = np.linspace(1, 200, 200)
plt.plot(K, average_mis)
plt.title('cross validation error for kNN')
plt.xlabel('k')
plt.ylabel('validation error')
plt.show()

# The result shows that k between 5 and 60 perform the best

1.2  6.2 Cross validation for model choice

In this problem we will consider the data sets Data/Pima_tr.csv and Data/Pima_te.csv. Your task is to do as good prediction as possible for the test set Pima_te, but you are only allowed to look at the true output in Pima_te once (like in the real life, where you design and implement a method, and then hand it over to the ultimate test, namely the user). Hence, you will have to use Pima_tr for both deciding which model to use and training the model.

The data set describes the prevalence of diabetes in women at least 21 years old of Pima Indian heritage, living near Phoenix, Arizona, USA. The data set describes, for each individual, whether she has diabetes or not, her age, the diabetes pedigree function (a summary of the diabetes history...
in her family), BMI, skin thickness, blood pressure, plasma glucose concentration and number of pregnancies.

The data frame contains the following columns:

- npreg: number of pregnancies.
- glu: plasma glucose concentration in an oral glucose tolerance test.
- bp: diastolic blood pressure (mm Hg).
- skin: triceps skin fold thickness (mm).
- bmi: body mass index (weight in kg/(height in m)^2).
- ped: diabetes pedigree function.
- age: age in years.
- type: Yes or No, for diabetic according to WHO criteria.

1.2.1 (a)

Load the library and familiarize yourself with Pima_tr

```python
In [6]: np.random.seed(1)
    pima_tr = pd.read_csv('Data/Pima_tr.csv')
    pima_tr.columns.values

Out[6]: array(['npreg', 'glu', 'bp', 'skin', 'bmi', 'ped', 'age', 'type'],
                     dtype=object)
```

1.2.2 (b)

See how well you can fit the Pima_tr with logistic regression, LDA, QDA and k-NN (k = 2). The output is whether an individual has diabetes or not, and the input the remaining variables. What error rate does each method have? Is it a good indicator of which method is preferable?

```python
In [7]: X = pima_tr.drop(columns=['type'])
    Y = pima_tr['type']

    # logistic regression
    model = skl_lm.LogisticRegression(solver='liblinear')
    model.fit(X, Y)
    prediction = model.predict(X)
    err = np.mean(prediction != Y)
    print('Error rate for logistic regression: ' + str(err))

    # LDA
    model = skl_da.LinearDiscriminantAnalysis()
    model.fit(X, Y)
    prediction = model.predict(X)
    err = np.mean(prediction != Y)
    print('Error rate for LDA: ' + str(err))

    # QDA
    model = skl_da.QuadraticDiscriminantAnalysis()
    model.fit(X, Y)
```
prediction = model.predict(X)
err = np.mean(prediction != Y)
print('Error rate for QDA: ' + str(err))

model = skl_nb.KNeighborsClassifier(n_neighbors=2)
model.fit(X, Y)
prediction = model.predict(X)
err = np.mean(prediction != Y)
print('Error rate for kNN: ' + str(err))

Error rate for logistic regression: 0.245
Error rate for LDA: 0.23
Error rate for QDA: 0.23
Error rate for kNN: 0.15

1.2.3 (c)

Instead of (a), perform 10-fold cross-validation by first randomly permute Pima_tr and divide it in 10 parts. Then, in a loop with one of the 10 parts held out as validation data, fit logistic regression, LDA, QDA and k-NN (k = 2) to the training data and evaluate the performance on the validation data. Plot your results in a box plot with the error rates. Feel free to play around with the choice of inputs and other settings to improve the performance. Which method does this suggest us to use?

In [8]: randomize_indices = np.random.choice(X.shape[0], X.shape[0], replace=False)
models = []
models.append(skl_lm.LogisticRegression(solver='liblinear'))
models.append(skl_da.LinearDiscriminantAnalysis())
models.append(skl_da.QuadraticDiscriminantAnalysis())
models.append(skl_nb.KNeighborsClassifier(n_neighbors=2))

misclassification = np.zeros((10, np.shape(models)[0]))

for i in range(10):
    n = np.ceil(X.shape[0]/10)  # number of samples in each fold
    validationIndex = np.arange(i*n, min(i*n+n, X.shape[0]), 1).astype('int')
    randomize_validationIndex = randomize_indices[validationIndex]
    X_train = X.iloc[~X.index.isin(randomize_validationIndex)]
    Y_train = Y.iloc[~Y.index.isin(randomize_validationIndex)]
    X_validation = X.iloc[randomize_validationIndex]
    Y_validation = Y.iloc[randomize_validationIndex]

    for m in range(np.shape(models)[0]):  # try different models
        model = models[m]
        model.fit(X_train, Y_train)
        prediction = model.predict(X_validation)
        misclassification[i, m] = (np.mean(prediction != Y_validation))
plt.boxplot(misclassification)
plt.title('cross validation error for different methods')
plt.xticks(np.arange(4)+1, ('logReg', 'LDA', 'QDA', 'kNN'))
plt.ylabel('validation error')
plt.show()

1.2.4 d)
Now, decide which method to choose and train it on the entire data set Pima_tr and predict Pima_te. How well do you perform?

1.2.5 (e)
Now, since we are in a simulated environment, we can cheat and break the rule that we were only allowed to look at the true output in Pima_te once. That is, explore how well the other methods do when you train them on Pima_tr and predict Pima_te. Did you make the "right" choice in (d)?

In [9]: # d)-e)
pima_te = pd.read_csv('Data/Pima_te.csv')
X_test = pima_te.drop(columns=['type'])
Y_test = pima_te['type']

model = skl_lm.LogisticRegression(solver='liblinear')
model.fit(X, Y)
prediction = model.predict(X_test)
err = np.mean(prediction != Y_test)
print('Error rate for logistic regression: ' + str(err))

# LDA
model = skl_da.LinearDiscriminantAnalysis()
model.fit(X, Y)
prediction = model.predict(X_test)
err = np.mean(prediction != Y_test)
print('Error rate for LDA: ' + str(err))

# QDA
model = skl_da.QuadraticDiscriminantAnalysis()
model.fit(X, Y)
prediction = model.predict(X_test)
err = np.mean(prediction != Y_test)
print('Error rate for QDA: ' + str(err))

model = skl_nb.KNeighborsClassifier(n_neighbors=1)
model.fit(X, Y)
prediction = model.predict(X_test)
err = np.mean(prediction != Y_test)
print('Error rate for kNN: ' + str(err))

Error rate for logistic regression: 0.25
Error rate for LDA: 0.20180722891566266
Error rate for QDA: 0.2289156626506024
Error rate for kNN: 0.31626506024096385

### 1.3 Implementing problem 5.3

Verify your theoretical findings from problem 5.3 by repeating the experiment \( N \) times and approximating all expected values with sums. Let \( \sigma^2 = 1 \).

#### 1.3.1 a)

Generate training data \((n = 1)\), estimate \(\beta_0\) and compute \(\hat{y}(x, T)\). Repeat \(N\) times and store the results in a vector. Choose the regularization parameter yourself.

```python
In [10]: N = 1000 # number of Monte Carlo samples to replace an integral/expected value
sigma2 = 1 # noise variance
r = 0.1 # regularization parameter

yhat = np.zeros(N)
for i in range(N):
    y = 1+np.sqrt(sigma2)*np.random.normal()
```
\[
\beta_0 = \frac{y}{1+r} \\
yhat[i] = \beta_0
\]

1.3.2 b)

Estimate \(g(x_*) = \mathbb{E}_T[y(x_*, T)]\) from your vector of \(\hat{y}(x_*, T)\). Compare your result to your theoretical findings in 5.3b.

In [11]:
```python
    g = np.mean(yhat)
    print(g)
```

0.9289943393686503

1.3.3 c)

Estimate the square bias \(\mathbb{E}_*[ (g(x_*) - f(x_*) )^2 ]\) using your result from b) and your knowledge about the true \(f(x)\). Compare your result to your theoretical findings in 5.3c.

In [12]:
```python
    bias2 = (g-1)**2 #since neither f nor g depends on x_star,
# the expectation over x_star is not needed
    bias2
```

Out[12]: 0.005041803841694408

1.3.4 d)

Estimate the variance \(\mathbb{E}_*[\mathbb{E}_T[ (y(x_*, T - y_*) )^2 ]]\) using your vector of \(\hat{y}(x_*, T)\) from a) and your result from b). Compare your result to your theoretical findings in 5.3d.

In [13]:
```python
    var = np.mean((yhat-g)**2) #since neither yhat nor g depends on x_star,
# the expectation over x_star is not needed
    var
```

Out[13]: 0.7849877716231417

1.3.5 e)

Estimate the expected new data error \(\bar{E}_{new} = \mathbb{E}_T[ E_{new} ] = \mathbb{E}_T[\mathbb{E}_*[ (y(x_*, T - g(x_*) )^2 ])]\) by, for each \(\hat{y}(x_*, T)\) in your vector from a), simulate \(N\) copies of \(y_*\). Compare your result to your theoretical findings in 5.3f.

In [14]:
```python
    enew = np.zeros(N)
    for i in range(N):
        # estimate the new data error for training data i
        y_star = 1+np.sqrt(sigma2)*np.random.normal(size=N)
        enew[i] = np.mean((yhat[i]-y_star)**2)
# estimate the expected new data error (averaging over training data)
    enew_bar = np.mean(enew)
    enew_bar
```

Out[14]: 1.7877608629367197
1.3.6 f)

Make a loop over different values for the regularization parameter $\gamma$ and plot bias, variance and $E_{new}$ as a function of $\gamma$. Also plot your theoretical findings from 5.3 in the same plot.

In [15]: # different regularization parameters to try
    m = 100
    ri = np.logspace(-2,1,m)

    # pre-allocate vectors
    bias2 = np.zeros(m)
    var = np.zeros(m)
    enew_bar = np.zeros(m)

    # select variance
    sigma2 = 1

    # number of Monte Carlo samples (when replacing integrals with sums)
    N = 1000

    # loop over different regularization values
    for j in range(m):
        r = ri[j]

        # generate training data and estimate beta0
        yhat = np.zeros(N)
        for i in range(N):
            y = 1+np.sqrt(sigma2)*np.random.normal()
            beta0 = y/(1+r)
            yhat[i] = beta0

        g = np.mean(yhat)

        # estimate the squared bias
        bias2[j] = (g-1)**2

        # estimate the variance
        var[j] = np.mean((yhat-g)**2)

    enew = np.zeros(N)
    for i in range(N):
        # estimate the new data error (with training data i)
        y_star = 1+np.sqrt(sigma2)*np.random.normal(size=N)
        enew[i] = np.mean((yhat[i]-y_star)**2)

    # estimate the expected new data error (averaging over training data)
    enew_bar[j] = np.mean(enew)

In [16]: plt.semilogx(ri,bias2,'r',label='estimated bias^2')
    plt.semilogx(ri,var+bias2,'k',label='estimated bias^2+variance')
1.4 6.4 Implementing problem 5.5

Design an experiment (similarly to 6.3) where you numerically confirm the results from problem 5.5.

In [17]: # number of Monte Carlo samples (when replacing integrals with sums)
N = 1000

    # define true function
def f(x):
        return x**2

    # generate training data and estimate beta0
beta = np.zeros((2,N))
for i in range(N):
    x = np.random.uniform(-1,1,size=2)
    y = f(x)
    X = np.column_stack((np.ones(2),x))
    beta[:,i] = np.linalg.solve(X,y)

# define g
def g(x_star):
    return np.mean(beta[0,:]) + x_star*np.mean(beta[1,:])

# estimate the squared bias
x_star = np.random.uniform(-1,1,size=N)
bias2 = np.mean((g(x_star)-f(x_star))**2)

# estimate the variance
vari = np.zeros(N)
for i in range(N):
    # average over training data (test data fix at point i)
    yhat = beta[0,:i] + beta[1,:i]*x_star[i]
    vari[i] = np.mean((yhat-g(x_star[i]))**2)

# average over test data
var = np.mean(vari)

# estimate the expected new data error
enew = np.zeros(N)
for i in range(N):
    # average over test data (training data fix)
    y_star = f(x_star)
    yhat = beta[0,i] + beta[1,i]*x_star
    enew[i] = np.mean((yhat-y_star)**2)

enew_bar = np.mean(enew)

print(bias2)
print(var)
print(enew_bar)

0.1917926332719188
0.32100956094985034
0.5128021942217692