Part 1: The James-Stein Estimator (20 P)

Let $x_1, \dots, x_N \in \mathbb{R}^d$ be independent draws from a multivariate Gaussian distribution with mean vector μ and covariance matrix $\Sigma = \sigma^2 I$. It can be shown that the maximum-likelihood estimator of the mean parameter μ is the empirical mean given by:

$$\hat{\mu}_{\text{ML}} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

Maximum-likelihood appears to be a strong estimator. However, it was demonstrated that the following estimator

$$\hat{\mu}_{JS} = \Big(1 - \frac{(d-2) \cdot \frac{\sigma^2}{N}}{\|\hat{\mu}_{\text{ML}}\|^2}\Big) \hat{\mu}_{\text{ML}}$$

(a shrinked version of the maximum-likelihood estimator towards the origin) has actually a smaller distance from the true mean when $d \geq 3$. This however assumes knowledge of the variance of the distribution for which the mean is estimated. This estimator is called the James-Stein estimator. While the proof is a bit involved, this fact can be easily demonstrated empirically through simulation. This is the object of this exercise.

The code below draws ten 50-dimensional points from a normal distribution with mean vector $\mu = (1, \dots, 1)$ and covariance $\Sigma = I$.

In [25]:
```
import numpy

def getdata(seed):

    n = 10                 # data points
    d = 50                 # dimensionality of data
    m = numpy.ones([d])    # true mean
    s = 1.0                # true standard deviation

    rstate = numpy.random.mtrand.RandomState(seed)
    X = rstate.normal(0,1,[n,d])*s+m

    return X,m,s
```

The following function computes the maximum likelihood estimator from a sample of the data assumed to be generated by a Gaussian distribution:

In [26]:
```
def ML(X):
    return X.mean(axis=0)
```

Implementing the James-Stein Estimator (10 P)

- **Based on the ML estimator function, write a function that receives as input the data $(X_i)_{i=1}^n$ and the (known) variance σ^2 of the generating distribution, and computes the James-Stein estimator**

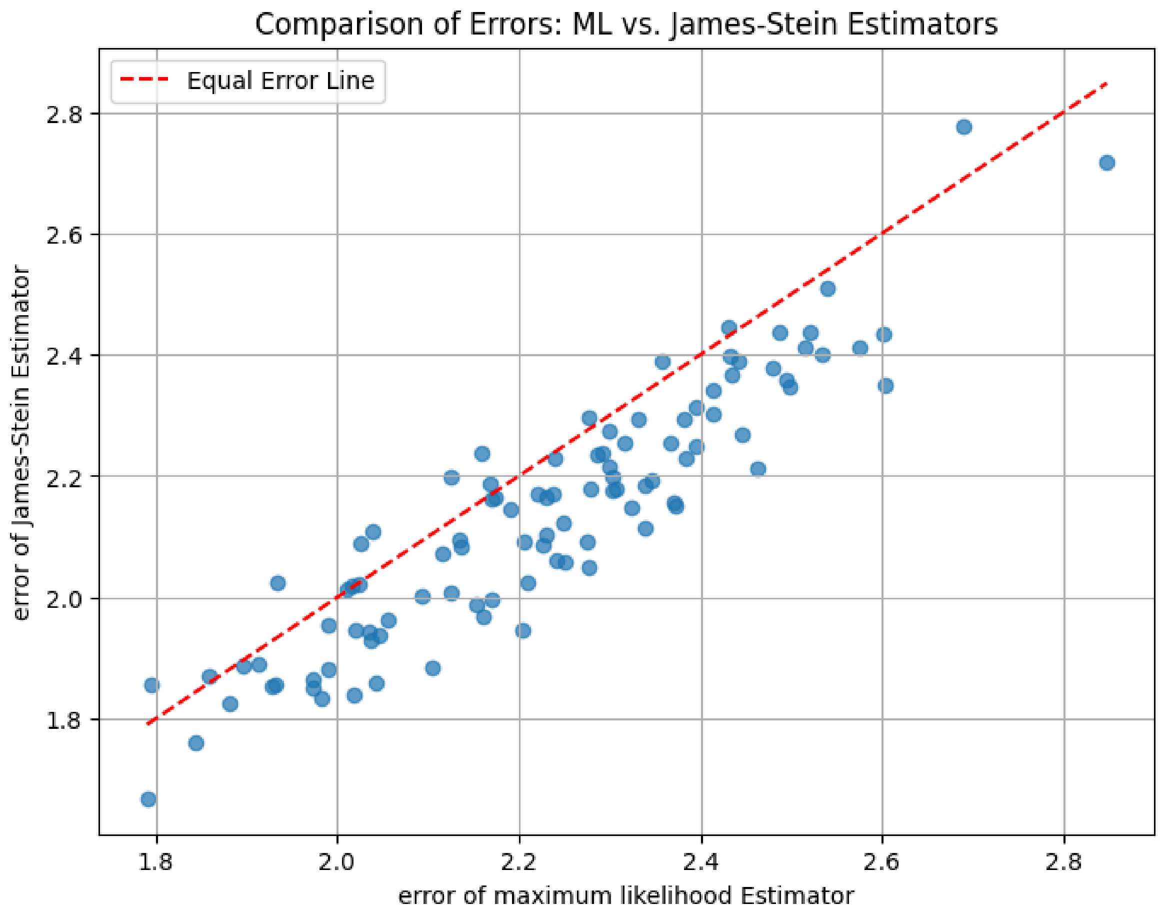
In [27]:
```python
def JS(X,s):
    # Compute the ML estimator
    mu_ML = X.mean(axis=0)  # shape (d,)
    # Compute the squared norm of mu_ML
    mu_ML_norm2 = numpy.sum(mu_ML ** 2)  # scalar
    # Get n and d from the shape of X
    n, d = X.shape
    # Compute sigma squared
    sigma2 = s ** 2
    # Compute the shrinkage factor
    factor = 1 - ((d - 2) * sigma2 / n) / mu_ML_norm2
    # Apply the shrinkage factor to mu_ML
    mu_JS = factor * mu_ML
    return mu_JS
```

Comparing the ML and James-Stein Estimators (10 P)

We would like to compute the error of the maximum likelihood estimator and the James-Stein estimator for 100 different samples (where each sample consists of 10 draws generated by the function `getdata` with a different random seed). Here, for reproducibility, we use seeds from 0 to 99. The error should be measured as the Euclidean distance between the true mean vector and the estimated mean vector.

- **Compute the maximum-likelihood and James-Stein estimations.**
- **Measure the error of these estimations.**
- **Build a scatter plot comparing these errors for different samples.**

In [28]:
```python
def compare_ML_JS():
    import matplotlib.pyplot as plt

    errors_ML = []
    errors_JS = []

    for seed in range(100):
        # Generate data with the given seed
        X, m, s = getdata(seed)
        # Compute the ML estimator
        mu_ML = ML(X)
        # Compute the James-Stein estimator
        mu_JS = JS(X, s)
        # Compute the Euclidean distance (error) between the estimators a
        error_ML = numpy.linalg.norm(mu_ML - m)
        error_JS = numpy.linalg.norm(mu_JS - m)
        # Store the errors
        errors_ML.append(error_ML)
        errors_JS.append(error_JS)

    # Create a scatter plot comparing the errors
    plt.figure(figsize=(8, 6))
    plt.scatter(errors_ML, errors_JS, alpha=0.7)
    plt.plot([min(errors_ML), max(errors_ML)], [min(errors_ML), max(error
    plt.xlabel('error of maximum likelihood Estimator')
    plt.ylabel('error of James-Stein Estimator')
    plt.title('Comparison of Errors: ML vs. James-Stein Estimators')
    plt.legend()
    plt.grid(True)
    plt.show()
compare_ML_JS()
###
```



Part 2: Bias/Variance Decomposition (30 P)

In this part, we would like to implement a procedure to find the bias and variance of different predictors. We consider one for regression and one for classification. These predictors are available in the module utils.

- **`utils.ParzenRegressor`** : A regression method based on Parzen window. The hyperparameter corresponds to the scale of the Parzen window. A large scale creates a more rigid model. A small scale creates a more flexible one.
- **`utils.ParzenClassifier`** : A classification method based on Parzen window. The hyperparameter corresponds to the scale of the Parzen window. A large scale creates a more rigid model. A small scale creates a more flexible one. Note that instead of returning a single class for a given data point, it outputs a probability distribution over the set of possible classes.

Each class of predictor implements the following three methods:

- `__init__(self,parameter):` Create an instance of the predictor with a certain scale parameter.
- `fit(self,X,T):` Fit the predictor to the data (a set of data points `X` and targets `T`).
- `predict(self,X):` Compute the output values arbitrary inputs `X`.

To compute the bias and variance estimates, we require *multiple samples* from the training set for a single set of observation data. To acomplish this, we utilize the `Sampler` class provided. The sampler is initialized with the training data and passed to the method for estimating bias and variance, where its function `sampler.sample()` is called repeatedly in order to fit multiple models and create an ensemble of prediction for each test data point.

Regression Case (15 P)

For the regression case, Bias, Variance and Error are given by:

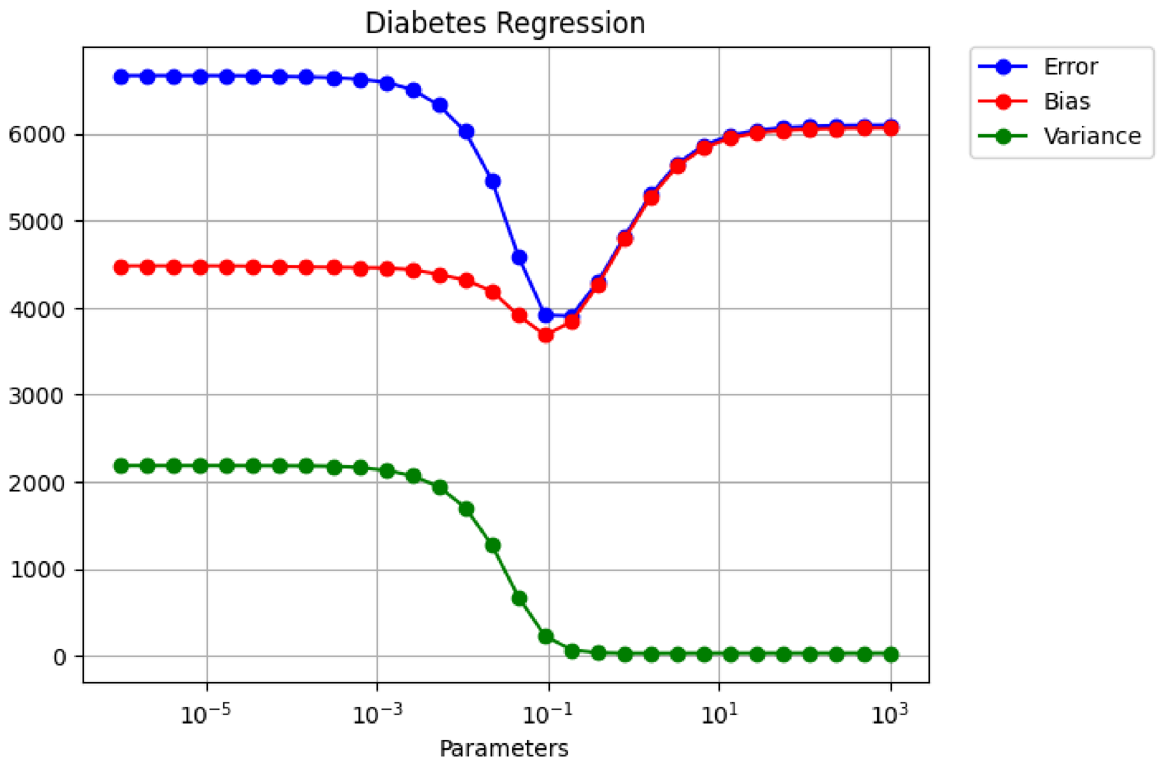
- $\text{Bias}(Y)^2 = (\mathbb{E}_Y[Y - T])^2$
- $\text{Var}(Y) = \mathbb{E}_Y[(Y - \mathbb{E}_Y[Y])^2]$
- $\text{Error}(Y) = \mathbb{E}_Y[(Y - T)^2]$

Task: Implement the KL-based Bias-Variance Decomposition defined above. The function should repeatedly sample training sets from the sampler (as many times as specified by the argument nbsamples), learn the predictor on them, and evaluate the variance on the out-of-sample distribution given by X and T.

In [29]:
```
def biasVarianceRegression(sampler, predictor, X, T, nbsamples=25):
    n = X.shape[0]  # Number of data points
    predictions = numpy.zeros((nbsamples, n))  # To store predictions fro

    # Loop over the number of samples
    for i in range(nbsamples):
        # Sample training data
        Xi, Ti = sampler.sample()
        # Create a new instance of the predictor with the same parameter
        model = predictor.__class__(predictor.parameter)
        # Fit the model on the sampled data
        model.fit(Xi, Ti)
        # Predict on the provided dataset X
        Yi = model.predict(X)
        # Store the predictions
        predictions[i, :] = Yi

    # Compute the expected prediction E_Y[Y] at each data point
    EY = numpy.mean(predictions, axis=0)  # Shape: (n,)

    # Compute the Bias^2 at each data point
    Bias2 = (EY - T) ** 2  # Shape: (n,)

    # Compute the Variance at each data point
    Variance = numpy.mean((predictions - EY[numpy.newaxis, :]) ** 2, axis
    # Compute the average Bias and Variance over all data points
    bias = numpy.mean(Bias2)
    variance = numpy.mean(Variance)

    return bias, variance
```

Your implementation can be tested with the following code:

In [30]:
```
import utils,numpy
%matplotlib inline
utils.plotBVE(utils.Diabetes,numpy.logspace(-6,3,num=30),utils.ParzenRegr
```



Classification Case (15 P)

We consider here the Kullback-Leibler divergence as a measure of classification error, as derived in the exercise, the Bias, Variance decomposition for such error is:

- $\text{Bias}(Y) = D_{\text{KL}}(T||R)$
- $\text{Var}(Y) = \mathbb{E}_Y[D_{\text{KL}}(R||Y)]$
- $\text{Error}(Y) = \mathbb{E}_Y[D_{\text{KL}}(T||Y)]$

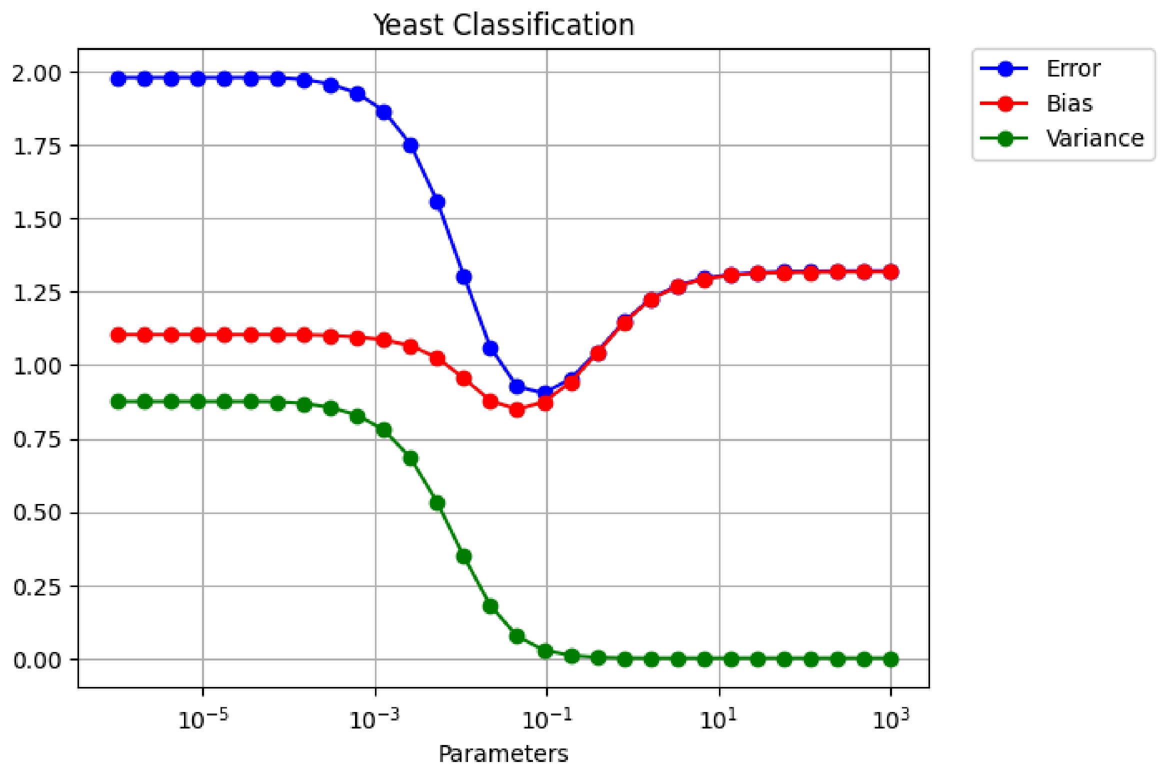
where R is the distribution that minimizes its expected KL divergence from the estimator of probability distribution Y (see the theoretical exercise for how it is computed exactly), and where T is the target class distribution.

Task: Implement the KL-based Bias-Variance Decomposition defined above. The function should repeatedly sample training sets from the sampler (as many times as specified by the argument nbsamples), learn the predictor on them, and evaluate the variance on the out-of-sample distribution given by X and T.

In [31]:
```
def biasVarianceClassification(sampler, predictor, X, T, nbsamples=25):
    n_samples = X.shape[0]

    # Determine the number of classes
    # If T is provided as class labels (integers), convert it to one-hot
    if T.ndim == 1 or T.shape[1] == 1:
        T = T.flatten()
        n_classes = numpy.max(T) + 1  # Assuming class labels start from
        T_one_hot = numpy.zeros((n_samples, n_classes))
        T_one_hot[numpy.arange(n_samples), T.astype(int)] = 1
    else:
        n_classes = T.shape[1]
        T_one_hot = T

    # Initialize an array to store predictions from each sample
    predictions = numpy.zeros((nbsamples, n_samples, n_classes))

    # Loop over the number of samples
    for i in range(nbsamples):
        # Sample training data
        Xi, Ti = sampler.sample()
        # Create a new instance of the predictor with the same parameter
        model = predictor.__class__(predictor.parameter)
        # Fit the model on the sampled data
        model.fit(Xi, Ti)
        # Predict on the provided dataset X
        Yi = model.predict(X)  # Expected shape: (n_samples, n_classes)
        # Ensure Yi sums to 1 across classes
        Yi /= Yi.sum(axis=1, keepdims=True)
        # Store the predictions
        predictions[i, :, :] = Yi

    # Compute R: the mean prediction over all samples for each data point
    R = numpy.mean(predictions, axis=0)  # Shape: (n_samples, n_classes)

    # Define a function to compute D_KL(P || Q) for arrays
    def compute_DKL_PQ(P, Q):
        epsilon = 1e-12  # Small value to avoid log(0)
        P = numpy.clip(P, epsilon, 1)
        Q = numpy.clip(Q, epsilon, 1)
        P /= P.sum(axis=1, keepdims=True)
        Q /= Q.sum(axis=1, keepdims=True)
        DKL = numpy.sum(P * numpy.log(P / Q), axis=1)  # Shape: (n_sample
        return DKL

    # Compute Bias: D_KL(T || R) for each data point
    Bias_array = compute_DKL_PQ(T_one_hot, R)  # Shape: (n_samples,)
    bias = numpy.mean(Bias_array)

    # Compute Variance: mean over samples of D_KL(R || Y_i) for each data
    DKLs = numpy.zeros((nbsamples, n_samples))
    for i in range(nbsamples):
        Y_i = predictions[i, :, :]  # Shape: (n_samples, n_classes)
        DKLs[i, :] = compute_DKL_PQ(R, Y_i)

    Variance_array = numpy.mean(DKLs, axis=0)  # Shape: (n_samples,)
    variance = numpy.mean(Variance_array)

    return bias, variance
```

Your implementation can be tested with the following code:

In [32]:
```
import utils,numpy
%matplotlib inline
utils.plotBVE(utils.Yeast,numpy.logspace(-6,3,num=30),utils.ParzenClassif
```


Yeast Classification
10^{-5}
10^{-3}
10^{-1}
10^{1}
10^{3}
Parameters
Error
Bias
Variance