Naïve Bayes Classifier for UCI Census-Income (KDD) Data Set:

Python Implementation:

```
# Imports
import pandas as pd
import numpy as np
import warnings

from math import log
from sklearn.metrics import accuracy_score

# Disable warnings from being printed
warnings.filterwarnings('ignore')
```

```
# Read the data and get the categorical attrbiutes
data = pd.read_csv("census/census-income.data")
categorical_attributes = ['ACLSWKR','ADTIND','ADTOCC','AHGA','AHSCOL','AMARITL','AMJIND','AMJOCC','ARACE',
                          'AREORGN','ASEX','AUNMEM','AUNTYPE','AWKSTAT','FILESTAT','GRINREG','GRINST','HHDFMX',
                          'HHDREL','MIGMTR1','MIGMTR3','MIGMTR4','MIGSAME','MIGSUN','PARENT','PEFNTVTY',
                          'PEMNTVTY','PENATVTY','PRCITSHP','SEOTR','VETQVA','VETYN','YEAR', 'INCOME']
data_categorical = data.loc[:,categorical_attributes]
```

```
# Preprocessing
# Fix missing values to mode as all are categorical variables
# and have one particular value which is very dominatingly occurring.

data_categorical.loc[data_categorical.GRINST == " ?", "GRINST"] = \
data_categorical.loc[data_categorical.GRINST != " ?", "GRINST"].mode().iloc[0]

data_categorical.loc[data_categorical.MIGMTR3 == " ?", "MIGMTR3"] = \
data_categorical.loc[data_categorical.MIGMTR3 != " ?", "MIGMTR3"].mode().iloc[0]

data_categorical.loc[data_categorical.MIGMTR4 == " ?", "MIGMTR4"] = \
data_categorical.loc[data_categorical.MIGMTR4 != " ?", "MIGMTR4"].mode().iloc[0]

data_categorical.loc[data_categorical.MIGSAME == " ?", "MIGSAME"] = \
data_categorical.loc[data_categorical.MIGSAME != " ?", "MIGSAME"].mode().iloc[0]

data_categorical.loc[data_categorical.PEFNTVTY == " ?", "PEFNTVTY"] = \
data_categorical.loc[data_categorical.PEFNTVTY != " ?", "PEFNTVTY"].mode().iloc[0]

data_categorical.loc[data_categorical.PEMNTVTY == " ?", "PEMNTVTY"] = \
data_categorical.loc[data_categorical.PEMNTVTY != " ?", "PEMNTVTY"].mode().iloc[0]

data_categorical.loc[data_categorical.PENATVTY == " ?", "PENATVTY"] = \
data_categorical.loc[data_categorical.PENATVTY != " ?", "PENATVTY"].mode().iloc[0]

# No other attributes have missing values

# As code is a categorical thing. Keeping it float poses problem in using it as dict key.
data_categorical["MIGMTR1"] = str(data_categorical["MIGMTR1"])

# Add relevant continuous attributes with binning. No of bins decided based on distinct values present.
data_categorical["wage_bins"] = pd.cut(data.AHRSPAY, bins=1000, labels=False)
categorical_attributes.insert(0, "wage_bins")

data_categorical["capgain_bins"] = pd.cut(data.CAPGAIN, bins=132, labels=False)
categorical_attributes.insert(0, "capgain_bins")

data_categorical["caploss_bins"] = pd.cut(data.CAPLOSS, bins=113, labels=False)
categorical_attributes.insert(0, "caploss_bins")
```

```
def iterate(accuracies):

    # Take a random samples of all data
    # and divide it equally into train and test of size 2000 each.
    data_randomised = data_categorical.iloc[np.random.permutation(data_categorical.shape[0])]
    train = data_randomised.iloc[:10000]
    test = data_randomised.iloc[10000:20000]

    # Separate the train data classwise.
    class_less = train.loc[data_categorical.INCOME == " - 50000.", :]
    class_more = train.loc[data_categorical.INCOME == " 50000+.", :]

    # Compute number on instances classwise and total.
    num_less = class_less.shape[0]
    num_more = class_more.shape[0]
    num_total = train.shape[0]

    # Compute priors for each class
    prob_less = num_less/num_total
    prob_more = num_more/num_total

    # Use log probabilities to avoid numerical errors
    log_prob_less = log(prob_less)
    log_prob_more = log(prob_more)
    # Compute likelihoods and take log of them
    probabilities = {}

    for categorical_attribute in categorical_attributes[:-1]:
        probabilities[categorical_attribute] = {
            " - 50000." : dict(class_less[categorical_attribute].value_counts()/num_less),
            " 50000+." : dict(class_more[categorical_attribute].value_counts()/num_more)
        }

    for categorical_attribute in probabilities.keys():
        for sal_class in probabilities[categorical_attribute].keys():
            for attribute_val in probabilities[categorical_attribute][sal_class].keys():
                probabilities[categorical_attribute][sal_class][attribute_val] = \
                log(probabilities[categorical_attribute][sal_class][attribute_val])

    # Predict
    results = []

    for i in range(test.shape[0]):
        record = test.iloc[i:i+1]

        posterior_less = 0
        posterior_more = 0

        for categorical_attribute in categorical_attributes[:-1]:

            if record[categorical_attribute].iloc[0] in probabilities[categorical_attribute][" - 50000."].keys():
                posterior_less = posterior_less + \
                probabilities[categorical_attribute][" - 50000."][record[categorical_attribute].iloc[0]]

            if record[categorical_attribute].iloc[0] in probabilities[categorical_attribute][" 50000+."].keys():
                posterior_more = posterior_more + \
                probabilities[categorical_attribute][" 50000+."][record[categorical_attribute].iloc[0]]

        posterior_less = posterior_less + log_prob_less
        posterior_more = posterior_more + log_prob_more

        if posterior_less >= posterior_more:
            cur_class = " - 50000."
        else:
            cur_class = " 50000+."

        results.append(cur_class)

    test['PREDICTION'] = results

    # Compute accuracy
    accuracies.append(accuracy_score(test.INCOME, test.PREDICTION))
```

```
# Run 10 times to compute mean and standard deviation of accuracy
accuracies = []
for i in range(10):
    iterate(accuracies)

print("Mean accuracy: ", np.mean(accuracies))
print("Standard deviation of accuracies: ", np.std(accuracies))
```

Output (Accuracy on scale 0-1):

```
Mean accuracy:  0.71125

Standard deviation of accuracies:  0.00923246987539
```

Dealing with numerical attributes:

If we want to include numerical attributes then we can can use binning as follows: Using the ones below slightly improved the accuracy.

```
# Add relevant continuous attributes with binning. No of bins decided based on distinct values present.
data_categorical["wage_bins"] = pd.cut(data.AHRSPAY, bins=1000, labels=False)
categorical_attributes.insert(0, "wage_bins")

data_categorical["capgain_bins"] = pd.cut(data.CAPGAIN, bins=132, labels=False)
categorical_attributes.insert(0, "capgain_bins")

data_categorical["caploss_bins"] = pd.cut(data.CAPLOSS, bins=113, labels=False)
categorical_attributes.insert(0, "caploss_bins")
```

Procedure:

10 iterations performed on random 20000 samples of total samples, out of which random 10000 were taken to train the classifier and other 10000 were used to validate and find mean and standard deviation of accuracies reported in each iteration. Increasing the number of samples used for training improves accuracy, but also takes much longer to run.

Dealing with ties:

In case of tie between posteriors of each class, I assigned the class of "<50,000" (93.80%) income, as prior of "<50,000" is much more than prior of ">50,000" (6.20%). I understand that this is already modelled in the posterior calculation. But still the error of misclassification in tie cases will be least ~ 6.20%.

Handling missing entries:

As we are dealing with missing values only in categorical attributes and each such attribute with missing values has one particular value which is very dominatingly occurring for that attribute. So we can use mode treatment, i.e. assign missing values to the mode of values for that attribute. This is done according to the entire dataset. Missing entries in numerical attributes can be treated with filling 0 or mean/median of the values, as appropriate.

II. Bayesian Parameter Estimation:

Background:

We say $p(x)$ is unknown but has a known parametric form i.e. $p(x \mid \theta)$ is known, where θ is an unknown parameter. We might have some information about prior density $p(\theta)$ and posterior $p(\theta \mid D)$ where D is the set of samples. So,

$$p(x \mid D) = \int p(x, \theta \mid D) d\theta$$
$$\Rightarrow p(x \mid D) = \int p(x \mid \theta, D) p(\theta \mid D) d\theta$$

Since the selection of x and that of D is done independently, $p(x \mid \theta, D) = p(x \mid \theta)$.

$$\Rightarrow p(x \mid D) = \int p(x \mid \theta) p(\theta \mid D) d\theta \tag{1}$$

If $p(\theta \mid D)$ peaks very sharply about some value $\hat{\theta}$, we obtain,

$$p(x \mid D) \simeq p(x \mid \hat{\theta})$$

Bayesian Parameter estimation for Gaussian Distribution:

We want to calculate the a posteriori density $p(\theta \mid D)$ and the desired probability density $p(x \mid D)$ for the case where $p(x \mid \mu) \sim N(\mu, \Sigma)$.

The Univariate Case:

$$p(x \mid \mu) \sim N(\mu, \sigma^2) \tag{2}$$

where μ is unknown, σ^2 is known.

Assume that,

$$p(\mu) \sim N(\mu_0, \sigma_0^2) \tag{3}$$

where μ_0, σ_0^2 are known, μ_0 represents our best a priori guess for μ, and σ_0^2 measures our uncertainty about this guess.

Using Bayes Formula,

$$p(\mu \mid D) = \frac{p(D \mid \mu) p(\mu)}{\int p(D \mid \mu) p(\mu) d\mu}$$
$$\Rightarrow p(\mu \mid D) = \alpha \cdot p(D \mid \mu) p(\mu)$$

where α is simply a normalization factor, independent of μ.

$$\Rightarrow p(\mu \mid D) = \alpha \cdot \prod_{k=1}^{n} p(x_k \mid \mu) p(\mu) \tag{4}$$

From (2), (3) and (4)

$$p(\mu \mid D) = \alpha \cdot \prod_{k=1}^{n} [\frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{\frac{-1}{2}(\frac{x_k - \mu}{\sigma})^2}] \cdot [\frac{1}{\sqrt{2\pi}\sigma_0} \mathrm{e}^{\frac{-1}{2}(\frac{\mu - \mu_0}{\sigma_0})^2}]$$

$$\Rightarrow p(\mu \mid D) = \dot{\alpha} \cdot \mathrm{e}^{\frac{-1}{2}[\sum_{k=1}^{n} (\frac{\mu - x_k}{\sigma})^2 + (\frac{\mu-\mu_0}{\sigma_0})^2]}$$

$$\Rightarrow p(\mu \mid D) = \ddot{\alpha} \cdot \mathrm{e}^{\frac{-1}{2}[(\frac{n}{\sigma^2}+\frac{1}{\sigma_0^2})\mu^2 - 2(\frac{1}{\sigma^2}\sum_{k=1}^{n} x_k + \frac{\mu_0}{\sigma_0^2})\mu]} \quad (5)$$

Comparing this to standard form,

$$p(\mu \mid D) = \frac{1}{\sqrt{2\pi}\sigma_n} \mathrm{e}^{\frac{-1}{2}(\frac{\mu-\mu_n}{\sigma_n})^2} \quad (6)$$

we have,

$$\frac{1}{\sigma_n^2} = \frac{n}{\sigma^2} + \frac{1}{\sigma_0^2} \text{and} \frac{\mu_n}{\sigma_n^2} = \frac{n}{\sigma^2}\overline{x_n} + \frac{\mu_0}{\sigma_0^2} \quad (7)$$

where $\overline{x_n}$ is the sample mean. $\overline{x_n} = \frac{1}{n}\sum_{k=1}^{n} x_k$.

Simplifying (7) we get, $\mu_n = \frac{n\sigma_0^2}{n\sigma_0^2+\sigma^2}\overline{x_n} + \frac{\sigma^2}{n\sigma_0^2+\sigma^2}\mu_0$ and $\sigma_n^2 = \frac{\sigma_0^2\sigma^2}{n\sigma_0^2+\sigma^2}$.

This final step is to determine $p(x \mid D)$ using (1), (6) and (8). (8)

$$p(x \mid D) = \int [\frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{\frac{-1}{2}(\frac{x-\mu}{\sigma})^2}][\frac{1}{\sqrt{2\pi}\sigma_n} \mathrm{e}^{\frac{-1}{2}(\frac{\mu-\mu_n}{\sigma_n})^2}]d\mu$$

$$\Rightarrow p(x \mid D) = \int [\frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{\frac{-1}{2}(\frac{x-\mu}{\sigma})^2}][\frac{1}{\sqrt{2\pi}\sigma_n} \mathrm{e}^{\frac{-1}{2}(\frac{\mu-\mu_n}{\sigma_n})^2}]d\mu$$

$$\Rightarrow p(x \mid D) = \frac{1}{2\pi\sigma\sigma_n} \mathrm{e}^{\frac{-1}{2}\frac{(x-\mu_n)^2}{\sigma^2+\sigma_n^2}} \int \mathrm{e}^{\frac{-1}{2}\frac{\sigma^2+\sigma_n^2}{\sigma^2\sigma_n^2}(\mu - \frac{\sigma_n^2 x + \sigma^2\mu_n}{\sigma^2+\sigma_n^2})^2} d\mu$$

Since the integral only depends on σ, σ_n, we have,

$$p(x \mid D) \sim N(\mu_n, \sigma^2 + \sigma_n^2).$$

<u>The Multivariate Case:</u>

$$p(x \mid \mu) \sim N(\mu, \Sigma) \text{and} p(\mu) \sim N(\mu_0, \Sigma_0)$$

where μ is unknown, Σ, μ_0 and Σ_0 are known.

As before, using Bayes Formula, we have,

$$\Rightarrow p(\mu \mid D) = \alpha \cdot \prod_{k=1}^{n} p(x_k \mid \mu)p(\mu) \quad (4)$$

Analogous to (5), we have

$$p(\mu \mid D) = \ddot{\alpha} \cdot \mathrm{e}^{\frac{-1}{2}[\mu^t(n\Sigma^{-1}+\Sigma_0^{-1})\mu - 2\mu^t(\Sigma^{-1}\sum_{k=1}^{n} x_k + \Sigma_0^{-1}\mu_0)]}$$

Comparing this to standard form,

$$p(\mu \mid D) = \ddot{\alpha}\mathrm{e}^{\frac{-1}{2}(\mu-\mu_n)^t\Sigma_n^{-1}(\mu-\mu_n)} \quad (9)$$

we have analogous to (7),

$$\Sigma_n^{-1} = n\Sigma^{-1} + \Sigma_0^{-1} \text{and } \Sigma_n^{-1}\mu_n = n\Sigma^{-1}\widehat{\mu_n} + \Sigma_0^{-1}\mu_0$$

where $\widehat{\mu_n}$ is the sample mean, $\widehat{\mu_n} = \frac{1}{n}\sum_{k=1}^{n} x_k$.

Using matrix identity $(A^{-1} + B^{-1})^{-1} = A(A+B)^{-1}B = B(A+B)^{-1}A$, we have analogous to (8),

$$\mu_n = \Sigma_0(\Sigma_0 + \frac{1}{n}\Sigma)^{-1}\widehat{\mu_n} + \frac{1}{n}\Sigma(\Sigma_0 + \frac{1}{n}\Sigma)^{-1}\mu_0 \text{and} \Sigma_n = \Sigma_0(\Sigma_0 + \frac{1}{n}\Sigma)^{-1}\frac{1}{n}\Sigma. \quad (10)$$

Finally we can solve $p(x \mid D)$ using (1), (9) and (10) as in the univariate case.
Thus, as with univariate case, we have,

$$p(x \mid D) \sim N(\mu_n, \Sigma + \Sigma_n).$$

III. PCA, LDA on Dorothea dataset, and Gaussian Naive Bayes Classifier:

Used all 100000 features, 50000 of which are probes.

<u>Python Implementation:</u>

```
# Imports
```

```python
import pandas as pd
import numpy as np

from warnings import filterwarnings
from sklearn.metrics import accuracy_score
from math import log

# Disable warnings from being printed
filterwarnings('ignore')
```

```python
fileN = 800
fileM = 100000

def read_data(filename):
    data = pd.DataFrame(columns=range(fileM))
    with open(filename, 'r') as datafile:
        lines = datafile.readlines()
        for i in range(len(lines)):
            record = np.fromstring(lines[i], dtype=int, sep=' ')
            record_bool = [0 for i in range(fileM)]
            for col in record:
                record_bool[col-1] = 1
            data.loc[i] = record_bool
    return data

def read_labels(filename):
    labels = []
    with open(filename, 'r') as datafile:
        lines = datafile.readlines()
        for line in lines:
            labels.append(np.fromstring(line, dtype=int, sep=' ')[0])
    return labels

# Read the data into dataframe
train_data = read_data("dorothea/dorothea_train.data")
valid_data = read_data("dorothea/dorothea_valid.data")

# Get the labels of the train data
train_data_labels = read_labels("dorothea/dorothea_train.labels")
valid_data_labels = read_labels("dorothea/dorothea_valid.labels")
```

```python
# Compute data which is constant in different runs of pca, i.e. eigenvectors

def compute_eigenvectors(data):

    # Center the data around mean
    data_centered = data - np.mean(data, axis=0)

    # Compute the covariance matrix (xx' i.e nXn), and find eigenvalues and eigenvectors
    cov_matrix = data_centered.transpose().cov()
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Now eigenvectors of x'x matrix can be obtained from these by multiplying by x', eigenvalues remain same
    eigenvectors = np.dot(np.transpose(data_centered), eigenvectors)

    # Sort the eigenvectors in decreasing order of eigenvalues
    sort_order = np.argsort(eigenvalues)[::-1]
    new_eigenvectors = np.zeros(eigenvectors.shape)
    for i in range(eigenvalues.shape[0]):
        new_eigenvectors[:, i] = eigenvectors[:, sort_order[i]]

    return new_eigenvectors
```

```python
# Get data in the new feature space of reduced dimensionality.
def pca_(data, new_eigenvectors, k):

    # Get first K eigenvectors
    eigenvectors_firstK = new_eigenvectors[:, :k]

    # Get data in reduced dimension space
    projected_data = np.dot(data, eigenvectors_firstK)

    return pd.DataFrame(projected_data)
```

```python
def GNBC(train, valid):

    # Separate the classes
    class_m = train[train["labels"] == -1]
    class_p = train[train["labels"] == 1]

    # Calculate prior probabilities for both classes
    prior_m = class_m.shape[0]/train.shape[0]
    prior_p = class_p.shape[0]/train.shape[0]

    # Calculate variances for all features
    var_m = np.var(class_m, axis=0)
    var_p = np.var(class_p, axis=0)

    # Calculate mean for all features
    mean_m = np.mean(class_m, axis=0)
    mean_p = np.mean(class_p, axis=0)
    # Predict
    results = []

    for i in range(valid.shape[0]):

        posterior_m = log(prior_m)
        posterior_p = log(prior_p)

        for j in range(valid.shape[1]-1):
            cur_x = valid.loc[i, j]
            posterior_m = posterior_m + (-0.5 * (((cur_x - mean_m[j])**2) / var_m[j])) - 0.5*log(var_m[j])
            posterior_p = posterior_p + (-0.5 * (((cur_x - mean_p[j])**2) / var_p[j])) - 0.5*log(var_p[j])

        if posterior_m >= posterior_p:
            cur_class = -1
        else:
            cur_class = 1

        results.append(cur_class)

    # Calculate accuracy
    return accuracy_score(valid["labels"], results)
```

```
def iterate_pca(train_data, valid_data, train_data_labels, valid_data_labels):

    accuracies = []

    kl = [100, 500, 800]

    new_eigenvectors_train = compute_eigenvectors(train_data)
    new_eigenvectors_valid = compute_eigenvectors(valid_data)

    for k in kl:
        projected_train = pca_(train_data, new_eigenvectors_train, k)
        projected_valid = pca_(valid_data, new_eigenvectors_valid, k)

        projected_train["labels"] = train_data_labels
        projected_valid["labels"] = valid_data_labels

        cur_accuracy = GNBC_pca(projected_train, projected_valid)
        accuracies.append(cur_accuracy)

    print("Statistics")
    print(100, accuracies[0])
    print(500, accuracies[1])
    print(800, accuracies[2])

iterate_pca(train_data, valid_data, train_data_labels, valid_data_labels)
```

```
def lda_(data):
    # Separate the train data classwise.
    class_m = data[data["labels"] == -1]
    class_p = data[data["labels"] == 1]

    # Drop the last labels column for matrix calculations
    class_m = class_m.drop("labels", axis=1)
    class_p = class_p.drop("labels", axis=1)

    # Get scatter matrices for each class separately
    scatter_m = np.cov(np.transpose(class_m))
    scatter_p = np.cov(np.transpose(class_p))

    # Compute means for each feature.
    mean_m = np.mean(class_m, axis=0)
    mean_p = np.mean(class_p, axis=0)
    mean_t = np.mean(data, axis=0)
    mean_t = mean_t.drop("labels")

    # Compute with class and between class scatter matrices
    sw = scatter_m + scatter_p
    swin = np.linalg.inv(sw)
    wstar = np.dot(swin, (mean_m - mean_p))

    # Find new projected data
    new_projected_data = data.drop("labels", axis=1)
    new_projected_data = np.dot(np.transpose(wstar), new_projected_data)
    return pd.DataFrame(new_projected_data)
```

```
def iterate_lda(train_data, valid_data, train_data_labels, valid_data_labels):

    # Get projected data as input for LDA
    new_eigenvectors_train = compute_eigenvectors(train_data)
    new_eigenvectors_valid = compute_eigenvectors(valid_data)
    projected_train = pca_(train_data, new_eigenvectors_train, 800)
    projected_valid = pca_(valid_data, new_eigenvectors_valid, 800)
    projected_train["labels"] = train_data_labels
    projected_valid["labels"] = valid_data_labels

    # Get LDA applied projected data
    new_projected_train = lda_(projected_train)
    new_projected_valid = lda_(projected_valid)
    new_projected_train["labels"] = train_data_labels
    new_projected_valid["labels"] = valid_data_labels

    accuracy = GNBC(new_projected_train, new_projected_valid)
    print("Accuracy: ", accuracy)

iterate_lda(train_data, valid_data, train_data_labels, valid_data_labels)
```

Output for Gaussian naive bayes after PCA (Accuracy on scale 0-1):

Statistics

k	Accuracy
100	0.834285714286
500	0.84
800	0.84

Output for Gaussian naive bayes after LDA (Accuracy on scale 0-1):

```
Accuracy:  0.902857142857
```

Assumptions:

For the given dataset, dimensionality(100000) >> number of samples(800). Also we cannot compute covariance matrix directly as we can't store 100000X100000 matrix. So we instead compute covariance matrix xx' (800X800), its eigenvectors and multiply them with x' to get eigenvectors(800 only) of the desired x'x matrix. Hence Gaussian naive bayes has been applied for K=100, 500 and 800.

For LDA since we didn't have this workaround, I performed LDA on the already reduced dimensionality data obtained via PCA, of dimensionality 800.

Observations:

Gaussian naive bayes on PCA improves accuracy as we increase K, but converges after some time. So k=800 doesn't improve much over k=500.
Accuracy is low because of extra probe (50000) features and also because the probability distribution of features may not always be gaussian.