Lista 2 - Tópicos em ML

Luben M. C. Cabezas Reinaldo C. Anacleto

Primeiramente, importando bibliotecas que serão utilizadas

```
# pacotes para transformacao dos dados e graficos
import pandas as pd
import numpy as np
import cdetools.cde_loss
import matplotlib.pyplot as plt
import seaborn as sns

# pacote do sklearn para data splitting

# pacotes para o normalizing flows e flexcode
import normflows as nf
import flexcode
from flexcode.regression_models import RandomForest

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# importando o modulo de modelos de mistura
from mixture_models.blocks import MixtureDensityNetwork
from mixture_models.blocks import NoiseType

# importando modulos auxiliares do torch
import torch
from torch.utils.data import TensorDataset, DataLoader

# modulo para acompanhar progresso das redes neurais
from tqdm import tqdm

# modulos do sklearn para calibração e classificação
# modelos de classificação
```

```
from sklearn.linear_model import LogisticRegressionCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier

# metricas
from sklearn.metrics import brier_score_loss, log_loss

# calibracao: platt scaling, reg isotonica e calibration plot
from sklearn.calibration import (
    CalibratedClassifierCV,
    calibration_curve,
    CalibrationDisplay,
)

# histogram binning
from calibration_module.calibrator import HistogramCalibrator
```

Exercício 1

Para o Exercício 1, consideraremos o banco de dados de propriedade físico-químicas de Proteínas que pode ser encontrado no repositório de Machine Learning UCI:

```
# importando os dados
protein_data = pd.read_csv("data/CASP.csv")
# visualizando numero de variaveis, observações e tipo de cada variavel
protein_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45730 entries, 0 to 45729
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   RMSD    45730 non-null  float64
 1   F1      45730 non-null  float64
 2   F2      45730 non-null  float64
 3   F3      45730 non-null  float64
 4   F4      45730 non-null  float64
 5   F5      45730 non-null  float64
 6   F6      45730 non-null  float64
```

```
 7   F7      45730 non-null  float64
 8   F8      45730 non-null  int64
 9   F9      45730 non-null  float64
dtypes: float64(9), int64(1)
memory usage: 3.5 MB
```

A variável resposta considerada nesse problema é a variável RMSD. Assim, temos interesse em modelar a incerteza aleatórica de RMSD em função das outras variáveis que nos dão diferentes características de cada proteina. Nota-se que todas as covariáveis consideradas são quantitativas, tendo apenas uma delas como discreta (F8). Como o conjunto de dados contém 45730 observações, separaremos o conjunto em 90% treino e validação e 10% para teste, já que assim, conseguimos destinar uma parcela considerável dos dados para ajuste dos diferentes modelos ao passo que alcançamos boas estimativas para as perdas e PIT-values.

```python
# separando os dados em treino+validação e teste
train_val_data, test_data = train_test_split(
    protein_data, test_size=0.1, random_state=42
    )
# tamanho do conjunto de treino
print(f"Tamanho do conjunto de treino e validação {train_val_data.shape[0]}")
print(f"Tamanho do conjunto de teste {test_data.shape[0]}")
```

```
Tamanho do conjunto de treino e validação 41157
Tamanho do conjunto de teste 4573
```

Já, para o treinamento e validação, consideraremos uma divisão de 70% para treino e 30% para validação:

```python
# separando os dados em treino+validação e teste
train_data, valid_data = train_test_split(
    train_val_data, test_size=0.3, random_state=125
)
# tamanho do conjunto de treino
train_data.shape[0]
print(f"Tamanho do conjunto de treino {train_data.shape[0]}")
print(f"Tamanho do conjunto de validação {valid_data.shape[0]}")
```

```
Tamanho do conjunto de treino 28809
Tamanho do conjunto de validação 12348
```

Além disso, os dados foram transformados do dataframe do pandas para arrays do numpy, e na sequência a matriz de covariáveis foi padronizada. Para serem utilizados como entradas na rede neural os arrays foram transformados em torch.tensor e separados em minilotes (batches) de tamanho 350 no teste e 150 na validação.

```
torch.manual_seed(167)
# tamanho de batch para data loader
batch_size_t = 350
batch_size_val = 150
# separando dataframe do pandas em arrays do numpy
X_train, y_train = train_data.iloc[:, 1:].values, train_data.iloc[:, 0].values
X_valid, y_valid = valid_data.iloc[:, 1:].values, valid_data.iloc[:, 0].values
X_test, y_test = test_data.iloc[:, 1:].values, test_data.iloc[:, 0].values

# adicionando padronizacao para redes neurais
# criando o scaler
scaler = StandardScaler()

# ajustando o scaler aos dados de treino
scaler.fit(X_train)

# Padronizando a matriz de covariaveis para as redes neurais
X_train_scaled = scaler.transform(train_data.iloc[:, 1:])
X_valid_scaled = scaler.transform(valid_data.iloc[:, 1:])
X_test_scaled = scaler.transform(test_data.iloc[:, 1:])

# transformando os arrays em torch.tensor e depois em data-loaders para
# os metodos baseados em redes neurais
X_train_t, X_valid_t, X_test_t = (
    torch.tensor(X_train_scaled, dtype=torch.float32),
    torch.tensor(X_valid_scaled, dtype=torch.float32),
    torch.tensor(X_test_scaled, dtype=torch.float32),
)

y_train_t, y_valid_t, y_test_t = (
    torch.tensor(y_train, dtype=torch.float32).view(-1, 1),
    torch.tensor(y_valid, dtype=torch.float32).view(-1, 1),
    torch.tensor(y_test, dtype=torch.float32).view(-1, 1),
)

# Criando datasets de torch tensors e usando data loader
train_loader, valid_loader = (
```

```
    DataLoader(
        TensorDataset(X_train_t, y_train_t),
        shuffle=True,
        batch_size=batch_size_t,
    ),
    DataLoader(
        TensorDataset(X_valid_t, y_valid_t), shuffle=True, batch_size=batch_size_val
    ),
)
```

Item 1

Implementaremos a seguir os diversos métodos, avaliando também a curva de aprendizado de cada:

- Mistura de Normais via rede neural:

Primeiro definiremos os parâmetros do modelo de mistura, a arquitetura e o otimizador da rede. Fixaremos um total de 10 componentes de mistura e consideraremos um total de 50 neurônios nas duas camadas intermediarias da rede.

```
# ajustando a rede de misturas
n_epochs = 500
d = X_train_t.shape[1]
n_componentes = 10
hidden_dim = 50
# algumas variaveis para performar early stopping
best_loss = torch.inf
counter = 0
patience = 30

# definindo arquitetura do modelo
mx_model = MixtureDensityNetwork(
    d,
    1,
    n_components=n_componentes,
    hidden_dim=hidden_dim,
    noise_type=NoiseType.DIAGONAL,
)

# definindo otimizador da rede neural
```

```
optimizer = torch.optim.Adamax(
    mx_model.parameters(),
    weight_decay=4e-5,
    lr=0.002,
)

scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, n_epochs)
```

Ajustando a rede neural de mistura de normais:

```
# salvando erros do treinamento e da validacao
avg_loss_train_list, avg_loss_val_list = [], []
best_loss_history = []

# percorrendo cada epoca
# starting neural network loop
for it in tqdm(range(n_epochs),
desc="Ajustando o rede neural de mistura de normais"):
    loss_train_array, loss_valid_array = [], []

    for X_tr, y_tr in train_loader:
        optimizer.zero_grad()
        loss = mx_model.loss(X_tr, y_tr).mean()

        # salvando o valor da perda no treino
        loss_value = loss.data
        loss_train_array.append(loss_value)

        # fazendo backpropagation
        loss.backward()
        optimizer.step()
    scheduler.step()

    # validando agora no conjunto de validacao
    for X_val, y_val in valid_loader:
        with torch.no_grad():
            mx_model.eval()
            # computando perda
            loss = mx_model.loss(X_val, y_val).mean()

            # salvando perda
```

```
            loss_value = loss.data
            loss_valid_array.append(loss_value)

    # salvando o historico das perdas
    avg_loss_train = np.mean(np.array(loss_train_array))
    avg_loss_valid = np.mean(np.array(loss_valid_array))

    avg_loss_val_list.append(avg_loss_valid)
    avg_loss_train_list.append(avg_loss_train)

    # performando early stopping pelo conjunto de validação
    if avg_loss_valid < best_loss:
        best_loss = avg_loss_valid
        best_loss_history.append(best_loss)
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print(f"Early stopping na época {it}")
            break
    mx_model.train()
```

```
Early stopping na época 363
```

Podemos visualizar a curva de aprendizado da rede pela Figura 1. Podemos observar no gráfico que a convergência ocorre rapidamente após poucas épocas e chega a um platô para as demais épocas. Além disso, a perda no treino e na validação possuem comportamento semelhante.

- Normalizing flows:

Definindo primeiro a arquitetura do modelo:

```
# definindo a arquitetura do normalizing flows
# juntamente com o numero de flows e o tipo de cada flow
# usaremos a spline quadratica como transformação bijetora
torch.manual_seed(190)
# numero de transformações a serem executadas
n_flows = 6
# numero de neuronios em cada camada
hidden_units=128
# numero de camadas
hidden_layers = 2
```

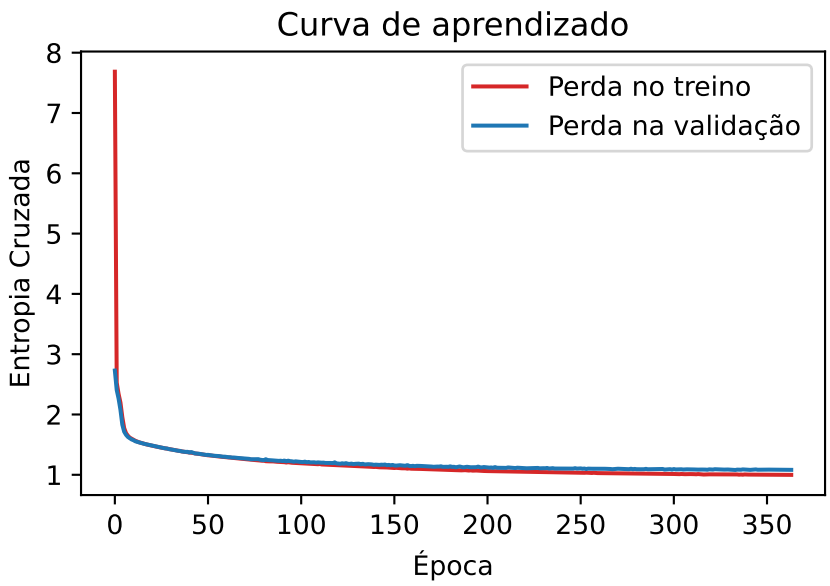

Figure 1: Curva de aprendizado do modelo de mistura de Normais

```
# numero para camada latente
latent_size = 1
# tamanho do contexto (numero de covariaveis)
context_size = X_train_t.shape[1]

flows = []
for i in range(n_flows):
    flows += [
            nf.flows.AutoregressiveRationalQuadraticSpline(
                latent_size,
                hidden_layers,
                hidden_units,
                num_context_channels=context_size,
                permute_mask=True,
                dropout_probability=True,
            )
        ]
    flows += [nf.flows.LULinearPermute(latent_size)]

# Estabelecendo a distribuição base da variável latente como uma normal(0,1)
q0 = nf.distributions.DiagGaussian(latent_size, trainable=False)
```

```
# Construindo a arquitetura do normalizing flow
nf_model = nf.ConditionalNormalizingFlow(q0, flows)

nf_optimizer = torch.optim.Adam(
            nf_model.parameters(),
            lr=3e-4,
            weight_decay=1e-5,
        )
```

Ajustando o modelo do normalizing flows:

```
# salvando erros do treinamento e da validacao
nf_avg_loss_train_list, nf_avg_loss_val_list = [], []
nf_best_loss_history = []
counter = 0
patience = 30

# percorrendo cada epoca
# starting neural network loop
for it in tqdm(
    range(n_epochs), desc="Ajustando o estimador de densidade via normalizing flows"
):
    loss_train_array, loss_valid_array = [], []

    for X_tr, y_tr in train_loader:
        nf_optimizer.zero_grad()
        loss = nf_model.forward_kld(y_tr, X_tr)

        # calculando valor da perda no treino
        loss_value = loss.data.numpy()
        loss_train_array.append(loss_value)

        # fazendo backpropagation e atualização dos parametros
        if ~(torch.isnan(loss) | torch.isinf(loss)):
            loss.backward()
            nf_optimizer.step()

    # validando agora no conjunto de validacao
    for X_val, y_val in valid_loader:
        with torch.no_grad():
            nf_model.eval()
```

```
            # computando perda
            loss = nf_model.forward_kld(y_val, X_val)

            # salvando perda
            loss_value = loss.data.numpy()
            loss_valid_array.append(loss_value)

    # salvando o historico das perdas
    avg_loss_train = np.mean(np.array(loss_train_array))
    avg_loss_valid = np.mean(np.array(loss_valid_array))

    nf_avg_loss_val_list.append(avg_loss_valid)
    nf_avg_loss_train_list.append(avg_loss_train)

    # performando early stopping pelo conjunto de validação
    if avg_loss_valid < best_loss:
        best_loss = avg_loss_valid
        nf_best_loss_history.append(best_loss)
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print(f"Early stopping na época {it}")
            break
    nf_model.train()
```

```
Early stopping na época 29
```

Avalia-se agora a curva de aprendizado dada pela Figura 2. Pelo gráfico, nota-se uma curva menos acentuada que a observada pela Mistura de Normais, tendo também maiores variações nos valores obserados de entropia cruzada. Também observa-se que as curvas do treino e da validação são semelhantes, tendo porém a perda no treino levemente superior a da validação principalmente nas primeiras épocas.

- FlexCode:

Para o flexcode, consideraremos a floresta aleatória como algoritmo de regressão para o ajuste de cada coeficiente da combinação linear das funções de base. Além disso, fixaremos 5 como o máximo de funções base ortonormais, utilizando a base de cossenos:

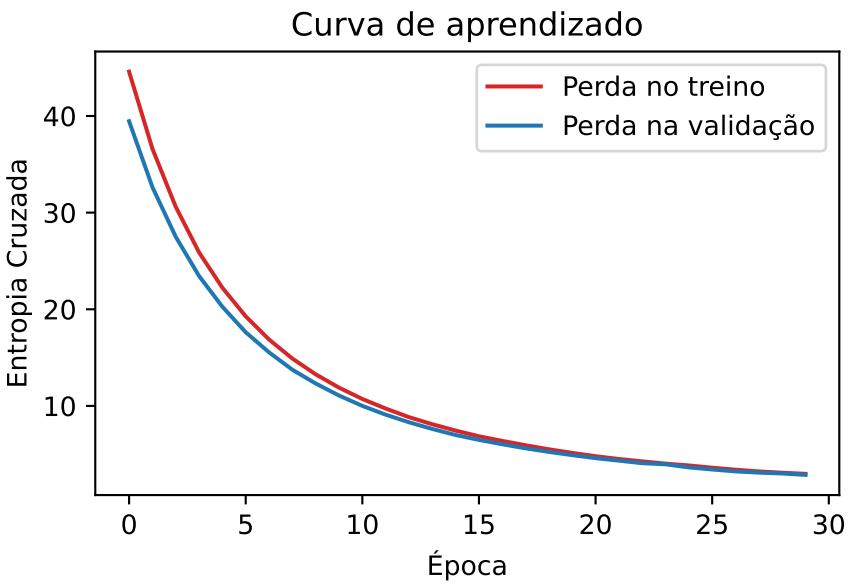

Figure 2: Curva de aprendizado do Normalizing Flows

```
fx_model = flexcode.FlexCodeModel(RandomForest, max_basis=5, basis_system="cosine")

# ajustando e tunando o modelo
fx_model.fit(X_train, y_train)
fx_model.tune(X_valid, y_valid)
```

A seguir, obteremos para todos os métodos as estimativas de densidades condicionais cada observação do conjunto de teste, juntamente com a estimativa do risco baseado na log-verossimilhança negativa (ou log-loss). Temos:

- Modelo de Mistura:

```
# estimativa de densidade via Modelo de Mistura
# obtendo estimativas do vetor de probabilidade, vetor de medias e
# matriz de covariancias de cada grupo
mx_model.eval()
log_pi, mu, sigma = mx_model.forward(X_test_t)

log_pi = log_pi.detach().numpy()
mu_mat = mu.detach().numpy().reshape(mu.size()[0], mu.size()[1])
sigma_mat = sigma.detach().numpy().reshape(mu.size()[0], mu.size()[1])
```

```
y_grid = np.linspace(0, np.max(y_train), 1000)

results = np.empty((mu.size()[0], mu.size()[1], y_grid.size))
res_log_loss = np.zeros(X_test_t.shape[0])

# obtendo densidade para cada y no grid fino
for i in range(mu.size()[0]):
    # media de cada grupo
    aux_mu = mu_mat[i, :]
    # variancias associadas a cada grupo
    aux_sigma = sigma_mat[i, :]
    # probabilidade de cada grupo
    aux_log_pi = log_pi[i, :]

    # computando densidade da normal com tais parametros em cada grupo
    z_score = (y_grid - aux_mu[:, np.newaxis]) / aux_sigma[:, np.newaxis]
    aux_log_pi = aux_log_pi[:, np.newaxis]
    loglik = (
        aux_log_pi
        - (1 / 2) * z_score**2
        - np.log(np.sqrt(np.pi * 2) * aux_sigma[:, np.newaxis])
    )

    # juntando tudo nos resultados
    results[i, :, :] = np.exp(loglik)

    # computando densidade da normal para o valor de y_test
    z_score = (y_test[i] - aux_mu) / aux_sigma
    aux_log_pi = log_pi[i, :]
    loglik_test = np.exp(
        aux_log_pi - (1 / 2) * z_score**2 - np.log(np.sqrt(np.pi * 2) * aux_sigma)
    )
    res_log_loss[i] = np.log(np.sum(loglik_test))


# densidades associada a cada Y para cada amostra de teste
densities_mx = np.sum(results, axis=1)

# estimando log-loss
mx_log_loss = [-np.mean(res_log_loss), np.std(res_log_loss) / np.sqrt(X_test.shape[0])]
```

- Normalizing flows

```
densities_nf = np.zeros((X_test_t.shape[0], 1000))
log_loss_nf = np.zeros(X_test.shape[0])
i = 0
for X in X_test_t:
    X_fixed = X.view(1, -1).repeat_interleave(1000, dim = 0)
    nf_model.eval()
    y_tensor = torch.tensor(y_grid, dtype = torch.float32).view(-1, 1)
    log_prob = nf_model.log_prob(y_tensor, X_fixed)
    nf_model.train()
    prob = torch.exp(log_prob)
    prob[torch.isnan(prob)] = 0
    densities_nf[i,:] = prob.detach().numpy()

    # log verossimilhança
    nf_model.eval()
    log_loss_nf[i] = nf_model.log_prob(
        y_test_t[i].view(1,-1),
        X.view(1, -1)
        ).detach().numpy()
    nf_model.train()
    i += 1

# estimando log verossimilhança e erro padrão desta
nf_log_loss = [
    -np.mean(log_loss_nf),
    np.std(log_loss_nf)/np.sqrt(X_test.shape[0])
    ]
```

- FlexCode

```
# densidades usadas para grafico e para computar perda L2
densities_fx, fx_y_grid = fx_model.predict(X_test, n_grid=1000)

# densidade em grid mais fino para computar log-loss mais precisamente
densities_fx_ll, fx_y_grid_ll = fx_model.predict(X_test, n_grid=5000)

# computando log-loss tambem para os pontos do grid mais proximos do teste
i = 0
fx_log_loss = np.zeros(X_test.shape[0])
for y in y_test:
    # encontrando valor do grid fino mais proximo do y no teste
```

```
    idx = (np.abs(fx_y_grid_ll.flatten() - y)).argmin()
    dens = densities_fx_ll[i, idx]
    # caso densidade seja nula, trocar por um epsilon
    if dens == 0.0:
        dens = 1e-8
    fx_log_loss[i] = np.log(dens)
    i += 1

# estimando log verossimilhança e erro padrão desta
fx_log_loss = [-np.mean(fx_log_loss), np.std(fx_log_loss) / np.sqrt(X_test.shape[0])]
```

Calcularemos agora para cada uma das densidade estimadas no conjunto de teste o risco baseado na perda L2:

```
###### codigo para instalar o cdetools para calcular a perda L2
# import sys
# sys.path.append('/content/cdetools/python/src')
# !pip install git+https://github.com/lee-group-cmu/cdetools.git#subdirectory=python

###### codigo para calcular perda L2 usando o pacote
import cdetools as cdetools

# computando perda L2
# Mistura de normais
gm_loss = cdetools.cde_loss(densities_mx, fx_y_grid, y_test)
# Normalizing flow
nf_loss = cdetools.cde_loss(densities_nf, fx_y_grid, y_test)
# FlexCode
fx_loss = cdetools.cde_loss(densities_fx, fx_y_grid, y_test)
```

Obtemos então os seguintes resultados:

```
results = pd.DataFrame({
    'Mistura de gaussianas': [f"{gm_loss[0]:.4f} ({gm_loss[1]:.4f})",
    f"{mx_log_loss[0]:.4f} ({mx_log_loss[1]:.4f})"],
    'FlexCode': [f"{fx_loss[0]:.4f} ({fx_loss[1]:.4f})",
    f"{fx_log_loss[0]:.4f} ({fx_log_loss[1]:.4f})"],
    'NFlow': [f"{nf_loss[0]:.4f} ({nf_loss[1]:.4f})",
    f"{nf_log_loss[0]:.4f} ({nf_log_loss[1]:.4f})"]
})
```

```
# transpondo o data frame resumindo as perdas e erros padrões
results_transposed = results.T
results_transposed.columns = ['Risco L2 (Erro padrão)',
'-Log-Verossimilhança (Erro padrão)'
]
results_transposed
```

	Risco L2 (Erro padrão)	-Log-Verossimilhança (Erro padrão)
Mistura de gaussianas	-0.3499 (0.0165)	2.0583 (0.0218)
FlexCode	-0.1544 (0.0016)	2.3891 (0.0308)
NFlow	-0.1335 (0.0072)	2.9390 (0.0325)

Percebemos que, tanto em termos da perda L2 quanto da entropria cruzada, o modelo de misturas de normais parece ter a melhor performance, indicando que esse modelo seja o mais adequado para a modelagem da densidade condicional para o banco de dados explorado.

Item 2

Fixamos a seguir 5 valores de X do conjunto de teste e visualizamos a estimativa de densidade condicional para cada um através da Figura 3. É possivel observar nesses gráficos que a densidade condicional obtida via Mistura de Normais possui comportamento mais suave em relação as demais nas 5 observações. Já a densidade condicional estimada pelo Normalizing Flows tem comportamento mais conturbado e pouco suave, enquanto a densidade estimada pelo FlexCode apesar de ser suave, possui poucos trechos de alta densidade. Ressalta-se também um comportamento unimodal da mistura de gaussianas para a maioria das observações, tendo apenas as observações 251 e 36 com certa bimodalidade.

É possivel observar através da Figura 3 que a densidade condicional obtida via Mistura de Normais possui comportamento mais suave em relação as demais nas 5 observações. A densidade condicional estimada pelo Normalizing Flows tem comportamento mais conturbado e pouco suave, enquanto a densidade estimada pelo FlexCode apesar de ser suave, possui poucos trechos de maiores densidades.

Item 3

Nesse item iremos computar os PIT values de cada metodologia e checar se cada uma delas está calibrada, com os valores PIT seguindo uma distribuição uniforme. Essa checagem será feita tanto visualmente quanto a partir de um teste de KS. Primeiramente, calculamos os pit values:

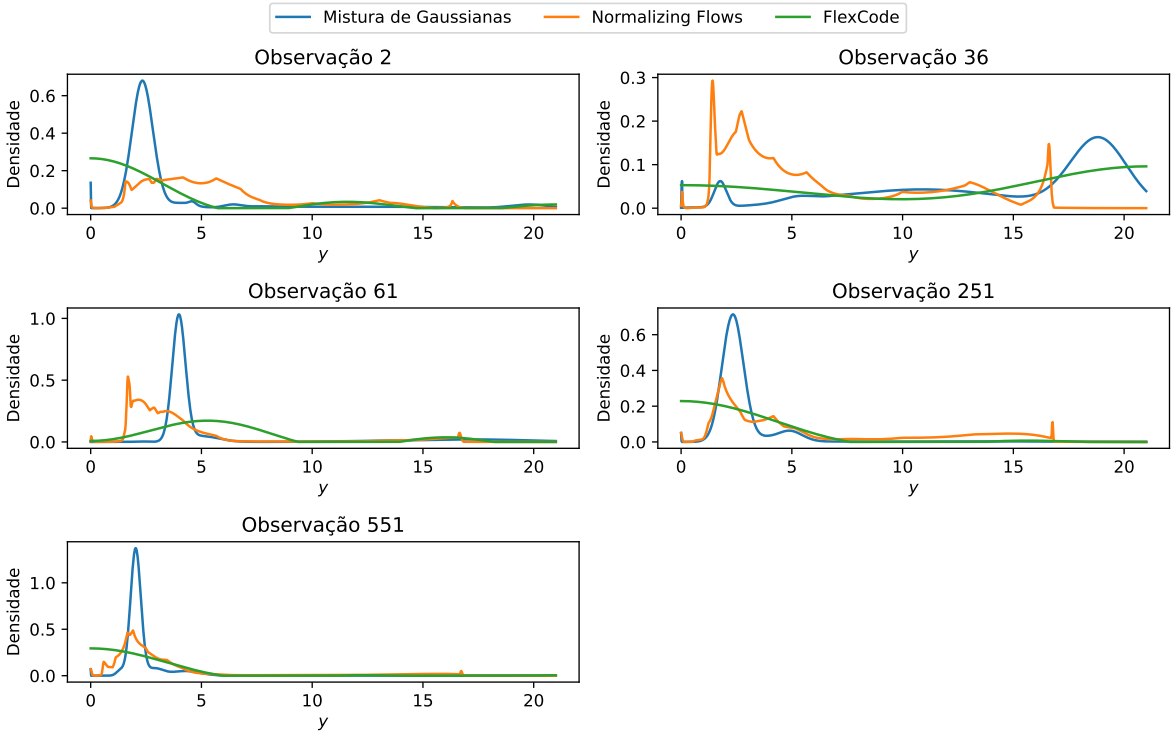

Figure 3: Densidades estimadas de cada método para 5 observações do conjunto de teste

```
# Mistura de normais
pit_mx = cdetools.cdf_coverage(densities_mx, fx_y_grid, y_test)
# FlexCode
pit_fx = cdetools.cdf_coverage(densities_fx, fx_y_grid, y_test)
# NFlow
pit_nf = cdetools.cdf_coverage(densities_nf, fx_y_grid, y_test)
```

Em seguida, podemos visualizar o histograma dos pit-values para cada metodologia através da Figura 4.

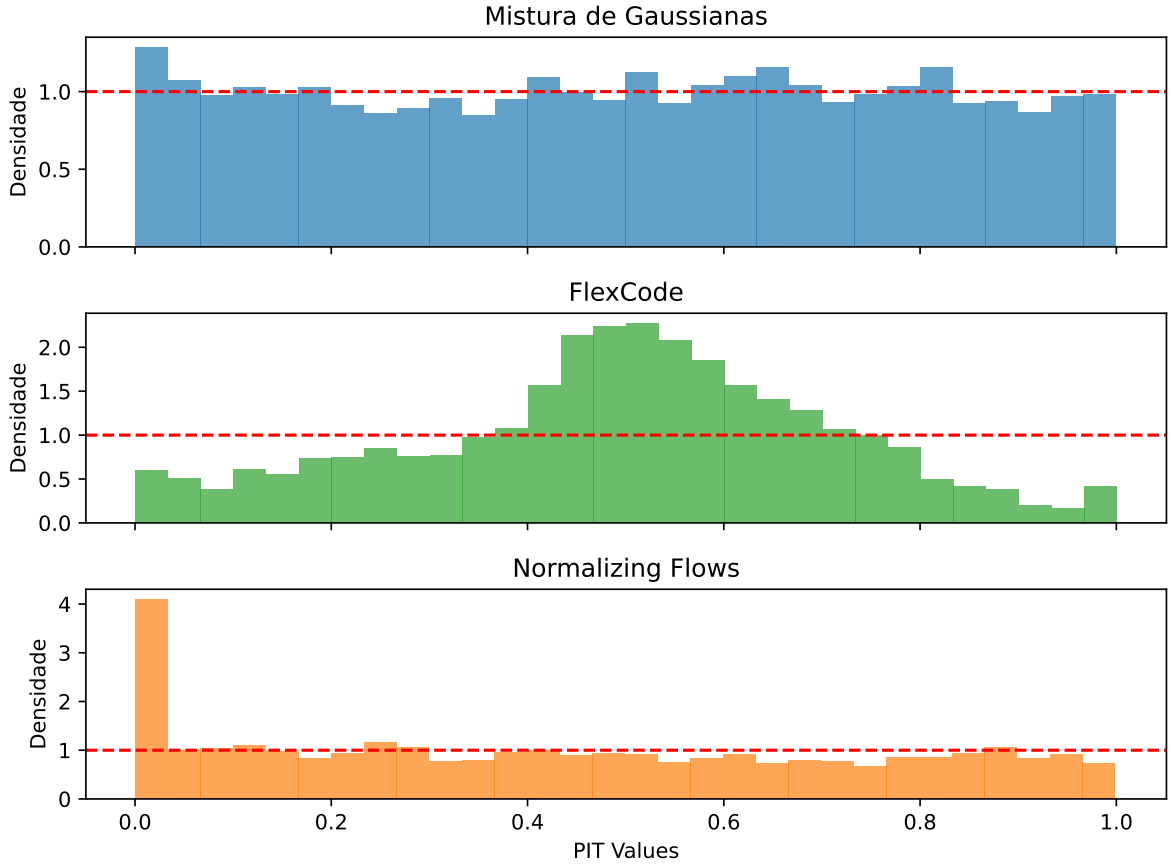

Figure 4: Histograma dos pit-values de cada método.

Percebe-se graficamente que a mistura de Gaussianas parece ser o método com pit-values mais próximos da distribuição uniforme, enquanto os pit-values do normalizing flows e do flexcode parecem ter certos desvios quanto a distribuição Uniforme postulada sob H_0, o que sugere que ambos os modelos podem não estar adequadamente calibrados. Para confirmar tais indícios, realizamos o teste KS dos pit-values, testando se a distribuição dos pit-values de fato segue

uma $U(0,1)$:

```
from scipy.stats import kstest

# Perform KS test for uniformity on each PIT dataset
ks_mx = kstest(pit_mx, 'uniform', args=(0, 1))
ks_fx = kstest(pit_fx, 'uniform', args=(0, 1))
ks_nf = kstest(pit_nf, 'uniform', args=(0, 1))

ks_df = pd.DataFrame({
    "Modelo": ["Mistura de normais", "FlexCode", "NFlow"],
    "Estatística KS": [ks_mx.statistic, ks_fx.statistic, ks_nf.statistic],
    "p-valor": [ks_mx.pvalue, ks_fx.pvalue, ks_nf.pvalue]
})
ks_df
```

	Modelo	Estatística KS	p-valor
0	Mistura de normais	0.014635	2.785128e-01
1	FlexCode	0.136883	3.365221e-75
2	NFlow	0.108051	5.986581e-47

Pelos p-valores e estatísticas do teste KS apresentados na tabela acima, temos evidências de que o modelo de densidade condicional baseado em Mistura de Normais esteja bem calibrado, com seus respectivos pit-values seguindo de forma muito próxima uma distribuição uniforme. Já modelos obtidos via FlexCode e Normalizing Flows estão descalibrados, com ambas metodologias apresentando p-valores muito pequenos juntamente com uma estatística KS razoavelmente grande.

Exercício 2

Primeiramente, importaremos o banco de dados utilizado:

```
# importando os dados
credit_card_data = pd.read_excel("data/default_of_credit_card_clients.xls",
header = 1).drop("ID", axis = 1)
# visualizando numero de variaveis, observações e tipo de cada variavel
credit_card_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 24 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   LIMIT_BAL                   30000 non-null  int64
 1   SEX                         30000 non-null  int64
 2   EDUCATION                   30000 non-null  int64
 3   MARRIAGE                    30000 non-null  int64
 4   AGE                         30000 non-null  int64
 5   PAY_0                       30000 non-null  int64
 6   PAY_2                       30000 non-null  int64
 7   PAY_3                       30000 non-null  int64
 8   PAY_4                       30000 non-null  int64
 9   PAY_5                       30000 non-null  int64
 10  PAY_6                       30000 non-null  int64
 11  BILL_AMT1                   30000 non-null  int64
 12  BILL_AMT2                   30000 non-null  int64
 13  BILL_AMT3                   30000 non-null  int64
 14  BILL_AMT4                   30000 non-null  int64
 15  BILL_AMT5                   30000 non-null  int64
 16  BILL_AMT6                   30000 non-null  int64
 17  PAY_AMT1                    30000 non-null  int64
 18  PAY_AMT2                    30000 non-null  int64
 19  PAY_AMT3                    30000 non-null  int64
 20  PAY_AMT4                    30000 non-null  int64
 21  PAY_AMT5                    30000 non-null  int64
 22  PAY_AMT6                    30000 non-null  int64
 23  default payment next month  30000 non-null  int64
dtypes: int64(24)
memory usage: 5.5 MB
```

A seguir, transformaremos as variáveis categóricas em dummies e separaremos a variavel resposta Y da matriz de covariáveis X:

```
# obtendo variavel resposta y
y = credit_card_data.loc[:, "default payment next month"].values

# covariaveis categoricas
# PAY strings
pay_str = [col for col in credit_card_data if (
    (col.startswith('PAY'))& ("AMT" not in col))]
```

```
cat_cols = ["SEX", "EDUCATION", "MARRIAGE"] + pay_str

# retirando ID e utilizando one-hot encoding em covariaveis categoricas
credit_card_filtered = pd.get_dummies(
    credit_card_data.drop(
    ["default payment next month"],
    axis = 1),
    columns = cat_cols,
    drop_first = True,
    dtype=float
    )
# obtendo matriz de covariaveis
X = credit_card_filtered.values
```

Após importar o conjunto de dados, dividiremos o conjunto de dados em treino/calibragem e teste, utilizando-se 90% para treino e calibragem e 10% para teste:

```
# separando os dados em treino+calibragem e teste
X_train_val, X_test, y_train_val, y_test = (
    train_test_split(
    X,
    y,
    test_size=0.1,
    random_state=145)
    )
```

A seguir, diviremos o conjunto de treino e calibragem em 70% treino e 30% calibragem,

```
# dividindo o conjunto em treino e calibração
X_tr, X_cal, y_tr, y_cal = (
    train_test_split(
    X_train_val,
    y_train_val,
    test_size=0.3,
    random_state=75)
    )

# padronizando os dados para o KNN
# criando o scaler
scaler = StandardScaler()
```

```
# ajustando o scaler aos dados de treino
scaler.fit(X_tr)

# Padronizando todas as matrizes de covariaveis
X_tr_scaled = scaler.transform(X_tr)
X_cal_scaled = scaler.transform(X_cal)
X_test_scaled = scaler.transform(X_test)
```

Com os dados divididos em conjuntos de treino, calibragem e teste, podemos primeiramente ajustar os diferentes modelos de classificação considerados e compara-los em termos do score de brier e log-loss.

Item 1

Ajustaremos a seguir os seguintes modelos de classificação: Regressão logística com penalização lasso, KNN e Random Forest.

```
# logistica com penalização lasso
# tunando parametero de regularização lambda
logistic_clf = LogisticRegressionCV(
    cv=5,
    random_state=19,
    solver="saga",
    penalty="l1",
    scoring="neg_log_loss",
    max_iter=250,
).fit(X_tr_scaled, y_tr)

# KNN com K = 30
knn_clf = KNeighborsClassifier(n_neighbors=30).fit(X_tr_scaled, y_tr)

# Random Forest com 300 arvores, criterio dado pela log-loss e
# profundidade maxima de 10
rf_clf = RandomForestClassifier(
    n_estimators=300, criterion="log_loss", max_depth=10, random_state=15
).fit(X_tr, y_tr)
```

Para esse caso, o KNN foi ajustado com $K = 30$ e o Random Forest com 300 árvores e profundidade total de 10. Agora, computaremos o score de brier e a log-loss para cada método no conjunto de teste, tendo a tabela:

```
# calculando score de brier
logis_brier, knn_brier, rf_brier = (
    brier_score_loss(
    y_test,
    logistic_clf.predict_proba(X_test_scaled)[:, 1]
    ),
    brier_score_loss(
    y_test,
    knn_clf.predict_proba(X_test_scaled)[:, 1]
    ),
    brier_score_loss(
    y_test,
    rf_clf.predict_proba(X_test_scaled)[:, 1]
    )
)

# computando log-loss
logis_ll, knn_ll, rf_ll = (
    log_loss(
    y_test,
    logistic_clf.predict_proba(X_test_scaled)
    ),
    log_loss(
    y_test,
    knn_clf.predict_proba(X_test_scaled)
    ),
    log_loss(
    y_test,
    rf_clf.predict_proba(X_test_scaled)
    )
)

# tabela com cada risco
results = pd.DataFrame({
    'Logistica': [f"{logis_brier:.4f}",
    f"{logis_ll:.4f}"],
    'KNN':[f"{knn_brier:.4f}",
    f"{knn_ll:.4f}"],
    'RF':[f"{rf_brier:.4f}",
    f"{rf_ll:.4f}"]
})
```

```
# Transpose the DataFrame
results_transposed = results.T
results_transposed.columns = ['Score de brier', 'Log-Verossimilhança']
results_transposed
```

	Score de brier	Log-Verossimilhança
Logistica	0.1324	0.4285
KNN	0.1350	0.4921
RF	0.1592	0.5016

É possível notar que a Regressão Logística tem melhor desempenho, pois possui os menores valores de Score de Brier e log-loss, seguido de KNN e Random Forest, respectivamente.

Item 2

A seguir, podemos analisar o calibration plot de cada metodologia através da Figura 5:

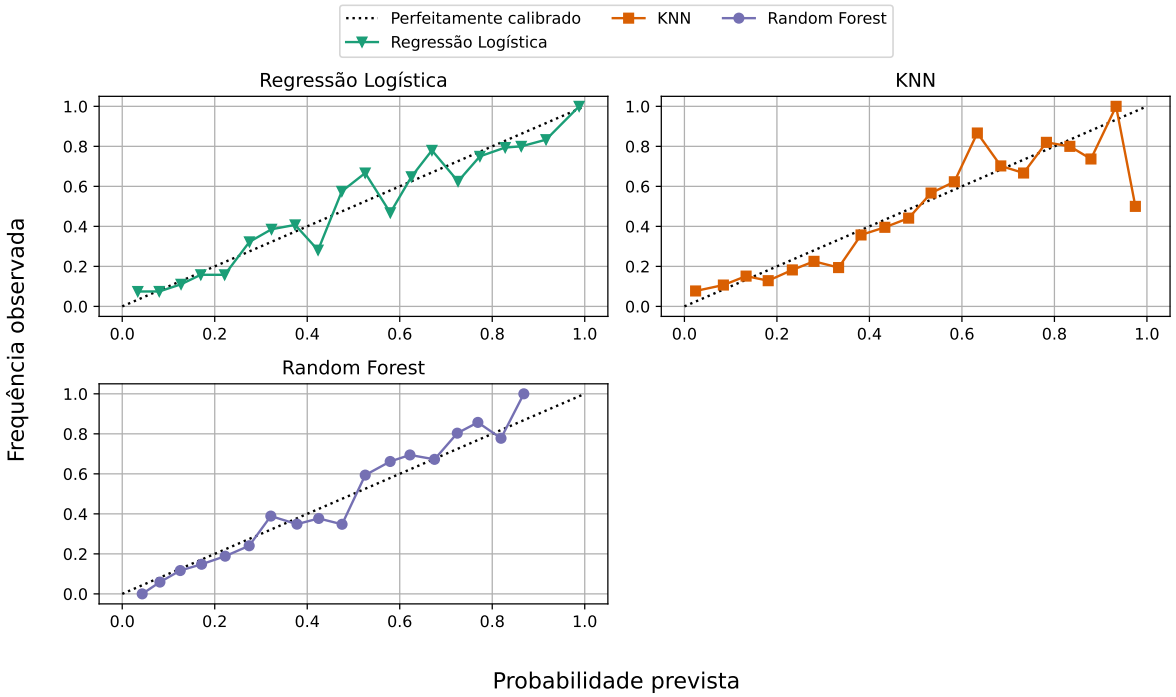

Figure 5: Calibration plot dos métodos de classificação ajustados

Nota-se na Figura 5 que o Random Forest aparenta estar mais calibrado em relação aos outros métodos. A Regressão Logística apesar de maior variação, contem valores mais ajustados nos limites do intervalo favorecendo a valor do Score de Brier em relação aos outros métodos. O KNN parece não estar muito bem calibrado principalmente para valores próximos a 1.

Item 3

Vamos a seguir recalibrar cada método via histogram binning. Para tal, ajustaremos cada método ao histogram binning no conjunto de calibragem separado inicialmente, tendo:

```
hist_logis, hist_knn, hist_rf = (
    HistogramCalibrator(n_bins=20),
    HistogramCalibrator(n_bins=20),
    HistogramCalibrator(n_bins=20),
)

# predicoes no conjunto de calibracao
logis_cal, knn_cal, rf_cal = (
    logistic_clf.predict_proba(X_cal_scaled)[:, 1],
    knn_clf.predict_proba(X_cal_scaled)[:, 1],
    rf_clf.predict_proba(X_cal)[:, 1],
)

# ajustando via histogram binning e avaliando os calibration plots
hist_logis.fit(logis_cal, y_cal)
hist_knn.fit(knn_cal, y_cal)
hist_rf.fit(rf_cal, y_cal)
```

```
HistogramCalibrator(n_bins=20)
```

Podemos então visualizar o calibration plot das probabilidades recalibradas no conjunto de teste a partir da Figura 6.

Percebemos que todos os métodos estão razoavelmente mais calibrados que as versões originais, havendo principal destaque para a regressão logística e random forest, que tem suas curvas de calibração muito próximas da reta do modelo perfeitamente calibrado.

Item 4

Recalcularemos com base nas recalibrações dos métodos as perdas do score de brier e log-loss, tendo:

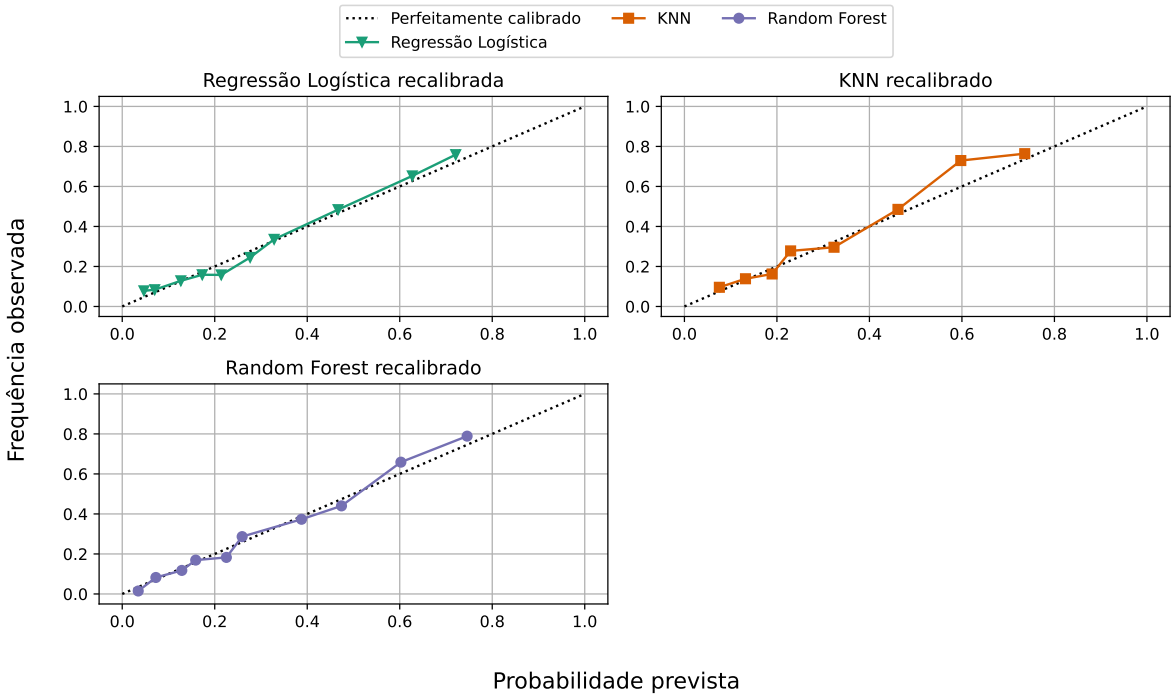

Figure 6: Calibration plot dos métodos de classificação recalibrados

```
# calculando score de brier para probabilidades recalibradas
logis_pred = logistic_clf.predict_proba(X_test_scaled)[:, 1]
cal_prob_logis = hist_logis.predict(logis_pred)

knn_pred = knn_clf.predict_proba(X_test_scaled)[:, 1]
cal_prob_knn = hist_knn.predict(knn_pred)

rf_pred = rf_clf.predict_proba(X_test)[:, 1]
cal_prob_rf = hist_rf.predict(rf_pred)

logis_brier_cal, knn_brier_cal, rf_brier_cal = (
    brier_score_loss(y_test, cal_prob_logis),
    brier_score_loss(y_test, cal_prob_knn),
    brier_score_loss(y_test, cal_prob_rf),
)

# elaborando uma matriz de predicoes
neg_cal_prob_logis = 1 - cal_prob_logis
neg_cal_prob_knn = 1 - cal_prob_knn
```

```
neg_cal_prob_rf = 1 - cal_prob_rf

# concatenando ambas probs
logis_prob_cal, knn_prob_cal, rf_prob_cal = (
    np.stack((neg_cal_prob_logis, cal_prob_logis), axis=1),
    np.stack((neg_cal_prob_knn, cal_prob_knn), axis=1),
    np.stack((neg_cal_prob_rf, cal_prob_rf), axis=1),
)

# computando log-loss
logis_ll_cal, knn_ll_cal, rf_ll_cal = (
    log_loss(y_test, logis_prob_cal),
    log_loss(y_test, knn_prob_cal),
    log_loss(y_test, rf_prob_cal),
)

# tabela com cada risco
results = pd.DataFrame(
    {
        "Logistica": [f"{logis_brier_cal:.4f}", f"{logis_ll_cal:.4f}"],
        "KNN": [f"{knn_brier_cal:.4f}", f"{knn_ll_cal:.4f}"],
        "RF": [f"{rf_brier_cal:.4f}", f"{rf_ll_cal:.4f}"],
    }
)

# Transpose the DataFrame
results_transposed = results.T
results_transposed.columns = ["Score de brier", "Log-Verossimilhança"]
results_transposed
```

	Score de brier	Log-Verossimilhança
Logistica	0.1338	0.4310
KNN	0.1348	0.4347
RF	0.1319	0.4241

Em geral, nota-se que o Random Forest e KNN tem melhora de performace, tendo menores valores para o Score de Brier e log-loss com relação aos desempenhos inciais dos métodos pré-calibração. Em particular, ressalta-se a consideravel melhora do Random Forest e ligeira melhora do KNN. Já a Regressão Logística teve uma leve piorada no seu desempenho em relação a pré-calibração, possivelmente devido ao reagrupamento de probabilidades estimadas acima de 0.8 em menores valores, como pode ser observado na Figura 6, tendo probabilidades

estimadas mais distante do valor 1 após a calibragem. Enfim, nota-se que após a recalibragem, o Random Forest passa a possuir melhor performance com relação ao KNN e Regressão Logística tanto pré quando pós calibração.