Computational Statistics and Data Analysis (MVComp2)

Solutions to exercise 10

Lecturer Tristan Bereau

Semester Wi23/24
Due Jan. 18, 2024, 23:59

1 Log-likelihood for multinomial logistic regression (3 points)

Recall binary logistic regression. We consider multinomial logistic regression—an extension to C classes. We write the classifier as a categorical distribution, Cat, a discrete probability distribution. For a datapoint $\boldsymbol{x}_n \in \mathbb{R}^D$, we have the discriminative classification model

$$p(y_n|\boldsymbol{x}_n, \boldsymbol{\theta}) = \mathrm{Cat}(y_n|\mathrm{softmax}(\mathbf{W}^\top \boldsymbol{x}_n)) = \prod_{c=1}^{C} \mathrm{softmax}(\boldsymbol{w}_c^\top \boldsymbol{x}_n)^{\mathbb{1}(y_n=c)},$$

where $\mathbf{W}$ is a $C \times D$ weight matrix for C classes and D features, $\mathbb{1}(\cdot)$ is the indicator function,[1] and the softmax function is defined below. We can rewrite the class label as a one-hot encoding: $y_{nc} = \mathbb{1}(y_n = c)$. This yields

$$p(y_{nc} = 1|\boldsymbol{x}_n, \boldsymbol{\theta}) = \mu_{nc} = \mathrm{softmax}(\boldsymbol{\eta}_{nc}) = \frac{\mathrm{e}^{\eta_{nc}}}{\sum_{c'=1}^{C} \mathrm{e}^{\eta_{nc'}}},$$

where $\boldsymbol{\eta}_{nc} = \boldsymbol{w}_c^\top \boldsymbol{x}_n$ is the vector of logits for the n-th datapoint and class c.

(a) Show that the Jacobian of the softmax is

$$\frac{\partial \mu_{ik}}{\partial \eta_{ij}} = \mu_{ik}(\delta_{kj} - \mu_{ij}),$$

where δ_{kj} is the Kronecker delta.

(b) Show that the gradient of the negative log-likelihood is given by

$$\nabla_{\boldsymbol{w}_c} \ell = \sum_i (\mu_{ic} - y_{ic}) \boldsymbol{x}_i.$$

(c) Show that the Hessian block between classes c and c' is given by

$$\mathbf{H}_{c,c'} = \mu_{nc}(\delta_{c'c} - \mu_{nc'}) \boldsymbol{x}_n \boldsymbol{x}_n^\top.$$

[1] $\mathbb{1}(\cdot)$ is 1 if its argument is satisfied and 0 otherwise.

1.1 Solution

(a) There are two cases, $j = k$ and $j \neq k$.

1. $j = k$:

$$\begin{aligned}\frac{\partial \mu_{ik}}{\partial \eta_{ik}} &= \frac{\partial}{\partial \eta_{ik}} \frac{\mathrm{e}^{\eta_{ik}}}{\sum_{k'=1}^{K} \mathrm{e}^{\eta_{ik'}}} \\ &= \mu_{ik} - \left(\frac{\mathrm{e}^{\eta_{ik}}}{\sum_{k'=1}^{K} \mathrm{e}^{\eta_{ik'}}} \right)^2 \\ &= \mu_{ik} - \mu_{ik}^2\end{aligned}$$

2. $j \neq k$:

$$\begin{aligned}\frac{\partial \mu_{ik}}{\partial \eta_{ij}} &= \frac{\partial}{\partial \eta_{ij}} \frac{\mathrm{e}^{\eta_{ik}}}{\sum_{k'=1}^{K} \mathrm{e}^{\eta_{ik'}}} \\ &= -\frac{\mathrm{e}^{\eta_{ik}} \mathrm{e}^{\eta_{ij}}}{(\sum_{k'=1}^{K} \mathrm{e}^{\eta_{ik'}})^2} \\ &= -\mu_{ik}\mu_{ij}\end{aligned}$$

Overall, we obtain

$$\frac{\partial \mu_{ik}}{\partial \eta_{ij}} = \mu_{ik}(\delta_{kj} - \mu_{ij}).$$

(b) The negative log-likelihood is given by

$$\mathrm{NLL}(\theta) = -\sum_{i=1}^{N} \sum_{c=1}^{K} \log \mu_{nc}^{y_{nc}} = -\sum_{i=1}^{N} \sum_{c=1}^{K} y_{nc} \log \mu_{nc}.$$

We then work out the gradient for datapoint n using the chain rule

$$\begin{aligned}\nabla_{\boldsymbol{w}_j} \mathrm{NLL}_n &= \sum_c \frac{\partial \mathrm{NLL}}{\partial \mu_{nc}} \frac{\partial \mu_{nc}}{\partial \eta_{nj}} \frac{\partial \eta_{nj}}{\partial \boldsymbol{w}_j} \\ &= -\sum_c \frac{y_{nc}}{\mu_{nc}} \mu_{nc}(\delta_{jc} - \mu_{nj})\boldsymbol{x}_n \\ &= \left(\sum_c y_{nc} \right) \mu_{nj}\boldsymbol{x}_n - \sum_c \delta_{jc} y_{nj} \boldsymbol{x}_n \\ &= (\mu_{nj} - y_{nj})\boldsymbol{x}_n\end{aligned}$$

and simply sum over all datapoints to get the desired result.

(c) Recall the definition of the Hessian and use the result of part (b)

$$\begin{aligned}\mathbf{H}_{c,c'} &= \nabla_{\boldsymbol{w}_{c'}} \nabla_{\boldsymbol{w}_c}^{\top} \mathrm{NLL} \\ &= \nabla_{\boldsymbol{w}_c'} (\mu_{nc} - y_{nc}) \boldsymbol{x}_n^{\top} \\ &= \left(\frac{\partial \mu_{nc}}{\partial \eta_{nc'}} \frac{\partial \eta_{nc'}}{\partial \boldsymbol{w}_{c'}} - \underbrace{\frac{\partial y_{nc}}{\partial \boldsymbol{w}_{c'}}}_{0} \right) \boldsymbol{x}_n^{\top} \\ &= \mu_{nc}(\delta_{c'c} - \mu_{nc'}) \boldsymbol{x}_n \boldsymbol{x}_n^{\top}.\end{aligned}$$

2 Classification of penguins (4 points)

Download the following dataset about penguins: penguins.csv. We will focus on the following features:

Variable	Description
`species`	Penguin species
`bill_length_mm`	bill (or beak) length in mm
`bill_depth_mm`	bill depth in mm
`flipper_length_mm`	flipper (or wing) length in mm

The objective of the exercise is to construct a classifier for the response variable `species` from the other features. The dataset contains three penguin species: Adelie, Chinstrap, and Gentoo.

(a) For each species, plot the one-dimensional cumulative distribution functions of the different features. Argue whether one-dimensional classifiers are likely to perform well to seperate each species.

(b) Let's classify penguins. To this end, assign a (uninformative) uniform prior distribution for each species. Model the likelihood using univariate Gaussian distributions. Write a function that can compute the posterior probability of each species given a feature and its value. Make sure to normalize your probabilities. Test your model for a flipper length of 213 mm, as well as 197 mm. Comment on the results.

(c) Use the one-dimensional posterior distributions from (b) applied to the dataset to evaluate their performance as classifier. Classify according to the highest probability encountered. Evaluate the resulting proportion of correctly predicted labels for each feature across your dataset.

(d) Use the prior and likelihood distributions of part (b) to build a Naïve Bayes classifier across the three features. Evaluate the performance of the classification by measuring the proportion of correctly predicted labels. Compare to (c) and comment.

Hint: You may find the following functions useful:

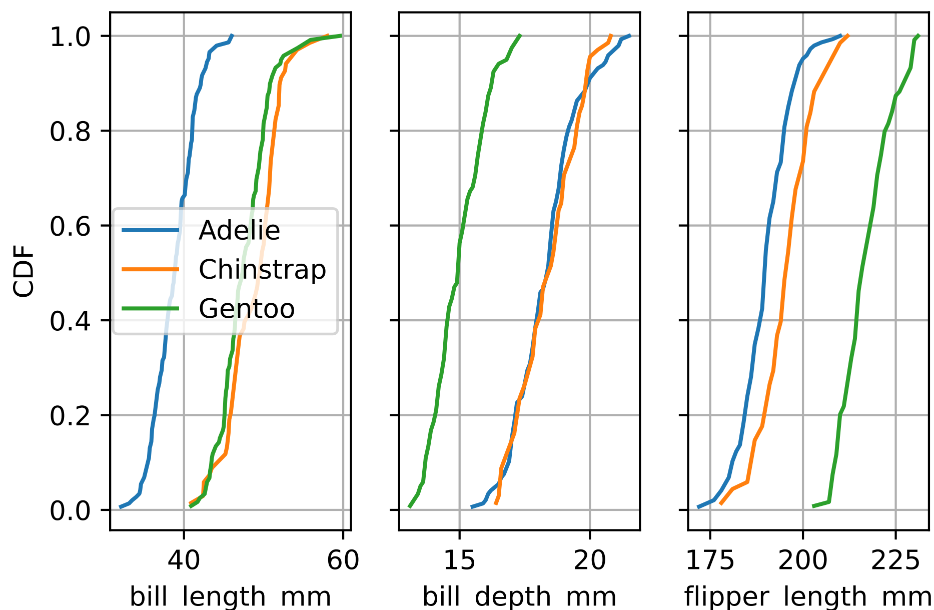
- `empiricaldist.Cdf`
- `empiricaldist.Pmf`
- `scipy.stats.norm`

2.1 Solution

(a) We plot the one-dimensional CDFs, splitting in different features and species

```
import pandas as pd
import matplotlib.pyplot as plt
from empiricaldist import Cdf, Pmf
from scipy.stats import norm

df = pd.read_csv("data_10_penguins.csv").dropna()
X = df[['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']]
y = df['species']

def col_name_to_cdf(df_: pd.DataFrame, col_name: str) -> dict:
    cdf_map = {}
```

```python
    for species, group in df_.groupby(by="species")[col_name]:
        cdf_map[species] = Cdf.from_seq(group, name=species)
    return cdf_map

fig, ax = plt.subplots(1, 3, sharey=True)
for i, feature in enumerate(["bill_length_mm", "bill_depth_mm", "flipper_length_mm"]):
    cdf_map = col_name_to_cdf(df, feature)
    for species, cdf in cdf_map.items():
        ax[i].plot(cdf, label=f"{species}")
    ax[i].grid()
    ax[i].set_xlabel(f"{feature}")
ax[0].set_ylabel("CDF")
ax[0].legend()
plt.show()
```



we observe that Adelie can likely easily be classified using `bill_length_mm`, Gentoo using `bill_depth_mm`. On the other hand, Chinstrap is likely not separable using one-dimensional classifiers.

(b) Build the prior and likelihood to compute the posterior for flipper length 213 mm:

```python
def col_to_normals(df_: pd.DataFrame, col_name: str) -> pd.DataFrame:
    norm_map = {}
    for species, group in df_.groupby(by="species")[col_name]:
        norm_map[species] = norm(group.mean(), group.std())
    return norm_map

def compute_posterior(prior_, data, normals_map):
    species = prior_.qs
    likelihood = [normals_map[specie].pdf(data) for specie in species]
```

```python
    posterior = prior_ * likelihood
    posterior.normalize()
    return posterior

species = ["Adelie", "Chinstrap", "Gentoo"]
prior = Pmf(1/3, species)
flipper_normals = col_to_normals(df, "flipper_length_mm")
compute_posterior(prior, 213, flipper_normals)
```

	probs
Adelie	0.002455
Chinstrap	0.058650
Gentoo	0.938895

where it's clear that Gentoo is the likely species. Second case: 193 mm, we get

```python
compute_posterior(prior, 197, flipper_normals)
```

	probs
Adelie	0.385581
Chinstrap	0.608469
Gentoo	0.005950

where we can only really rule out Gentoo, but cannot idenfity the correct species.

(c)

```python
def classification_1d_test(feature: str) -> float:
    df['Classification'] = "None"
    for i, row in df.iterrows():
        data_seq = row[feature]
        posterior = compute_posterior(
            prior, data_seq, col_to_normals(df, feature)
        )
        df.loc[i, 'Classification'] = posterior.max_prob()
    valid = df["Classification"].notna()
    same = df["species"] == df["Classification"]
    return same.sum() / valid.sum()

for feature in ["bill_length_mm", "bill_depth_mm", "flipper_length_mm"]:
    print(
        f"{feature}: {classification_1d_test(feature) * 100:3.1f}%"
)
```

```
bill_length_mm: 74.5%
```

```
bill_depth_mm: 61.9%
flipper_length_mm: 78.1%
```

(d)

```
def naive_bayes(prior, data_seq, norm_maps):
    posterior = prior.copy()
    for data, norm_map in zip(data_seq, norm_maps):
        posterior = compute_posterior(
            posterior, data, norm_map
        )
    return posterior

df['Classification'] = "None"
norm_maps = [
    col_to_normals(df, "bill_length_mm"),
    col_to_normals(df, "bill_depth_mm"),
    col_to_normals(df, "flipper_length_mm")
]

for i, row in df.iterrows():
    data_seq = row[["bill_length_mm", "bill_depth_mm", "flipper_length_mm"]]
    posterior = naive_bayes(prior, data_seq, norm_maps)
    df.loc[i, 'Classification'] = posterior.max_prob()
valid = df["Classification"].notna()
same = df["species"] == df["Classification"]
print(
    f"Classification performance: {same.sum() / valid.sum()*100:3.1f}%"
)
```

```
Classification performance: 96.4%
```

The classification performance is significantly higher than with the one-dimensional Normal models.

3 Prostate cancer, kernelized (3 points)

Let's revisit the prostate-cancer dataset from Homework Set 8, which you can download here: prostate.csv. This time we will use kernel ridge regression to build a supervised learning model for `lpsa`.

(a) Implement yourself a kernel ridge regression. Do not use existing statistics / machine learning libraries (though feel free to use linear-algebra libraries). To simplify your task, consider extending the following template that inherits from `OrdinaryLeastSquares` from Homework Set 8 (and provided below):

```
import numpy as np
from dataclasses import dataclass, field

@dataclass
class OrdinaryLeastSquares:
```

```python
    model_params: np.ndarray = field(init=False)
    training_set: np.array = field(init=False)

    def __post_init__(self):
        self.model_params = None

    def fit(self, X_train_: np.ndarray, y_train_: np.ndarray) -> None:
        self.training_set = X_train_
        self.model_params = (
            np.linalg.inv(X_train_.T @ X_train_) @ X_train_.T @ y_train_
        )

    def predict(self, X_test_: np.ndarray) -> np.ndarray:
        return X_test_ @ self.model_params

    def rmse(self, X_test_: np.ndarray, y_test_: np.ndarray) -> float:
        y_pred = self.predict(X_test_)
        return np.sqrt(mean_squared_error(y_test_, y_pred))

@dataclass
class GaussianKernel(OrdinaryLeastSquares):
    sigma: float
    regularization: float

    def kernel_matrix(self, x_1: np.array, x_2: np.array) -> np.array:
        pass

    def fit(self, x: np.array, y: np.array) -> None:
        pass

    def predict(self, x: np.array) -> np.array:
        pass
```

where `sigma` is the length scale of a Gaussian (i.e., "radial basis function") kernel and `regularization` is the coefficient in front of the ℓ_2 parameter. The function `kernel_matrix` computes the kernel matrix between any two datasets $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$. Use the same train/test split as in Homework set 8. Generate a contour plot of the *test* root-mean-squared error (RMSE) as a function of σ and λ with suggested ranges $10^{-1} \leq \sigma \leq 10^4$ and $10^{-6} \leq \lambda \leq 10^4$. Interpret the results: What happens at low and high σ and λ, and why?

Hint: You may find the following functions useful:

- `numpy.logspace`
- `numpy.meshgrid`
- `matplotlib.pyplot.contourf`

(b) Use part (a) to identify optimal hyperparameters for σ and λ. Generate a parity plot of predicted against reference labels for the test dataset. Compare the value of the RMSE to your results from ordinary least squares and linear ridge regression in Homework Set 8.

3.1 Solution

(a) First let's load the data and perform a train/test split

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

df = pd.read_csv("data_08_prostate.csv")

def extract_X_y(df_: pd.DataFrame):
    X = df_.to_numpy()[:,:-1]
    X = np.hstack((np.ones((X.shape[0], 1)), X))
    y = df_.to_numpy()[:,-1]
    return X, y

X, y = extract_X_y(df)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)
y_train.shape, y_test.shape
```

```
((67,), (30,))
```

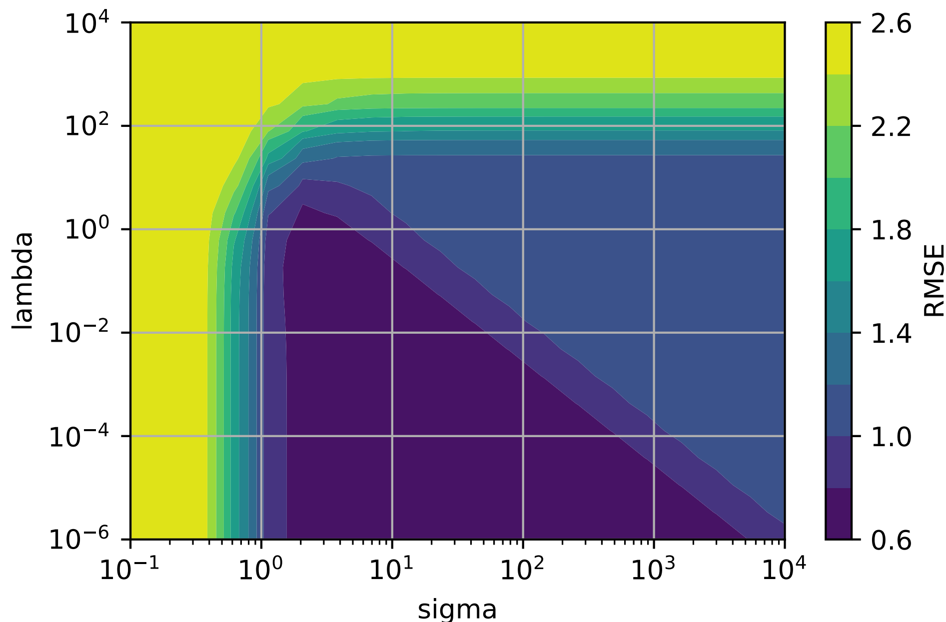
Implement kernel ridge regression

```
from scipy.spatial.distance import cdist
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

@dataclass
class GaussianKernel(OrdinaryLeastSquares):
    sigma: float
    regularization: float

    def __post_init__(self):
        self.training_set = None

    def kernel_matrix(self, x_1: np.array, x_2: np.array) -> np.array:
        return np.exp(- cdist(x_1, x_2, "euclidean") / (2. * self.sigma**2))

    def fit(self, x: np.array, y: np.array) -> None:
        self.training_set = x
        regularized_kernel = (
                self.kernel_matrix(x, x)
                + self.regularization * np.identity(len(x))
        )
        self.model_params = np.dot(y, np.linalg.inv(regularized_kernel))

    def predict(self, x: np.array) -> np.array:
        assert self.model_params is not None, "Model has not been trained yet!"
```

```python
        return np.dot(self.model_params, self.kernel_matrix(self.training_set, x))
```

We then perform a grid search for the two hyperparameters and generate the contour plot

```python
sigma_range = np.logspace(-1, 4, base=10, num=20)  # 0.1 to 100
regularization_range = np.logspace(-6, 4, base=10, num=20)  # 1e-8 to 1e-2
sigma_mesh, regularization_mesh = np.meshgrid(sigma_range, regularization_range)
rmse_values = np.zeros((len(sigma_range), len(regularization_range)))
for i, sigma in enumerate(sigma_range):
    for j, regularization in enumerate(regularization_range):
        gpr = GaussianKernel(sigma=sigma, regularization=regularization)
        gpr.fit(X_train, y_train)
        rmse_values[i, j] = gpr.rmse(X_test, y_test)

plt.contourf(sigma_mesh, regularization_mesh, rmse_values.T, levels=10)
plt.xscale('log')
plt.yscale('log')
plt.xlabel("sigma")
plt.ylabel("lambda")
plt.colorbar(label="RMSE")
plt.grid();
```



We observe the following:

- Small values of σ lead to large RMSE. This has to do with an inadequate length scale for the Gaussian kernel.
- Large values of λ lead to large RMSE. The penalty term is too large and overwhelms the fit.

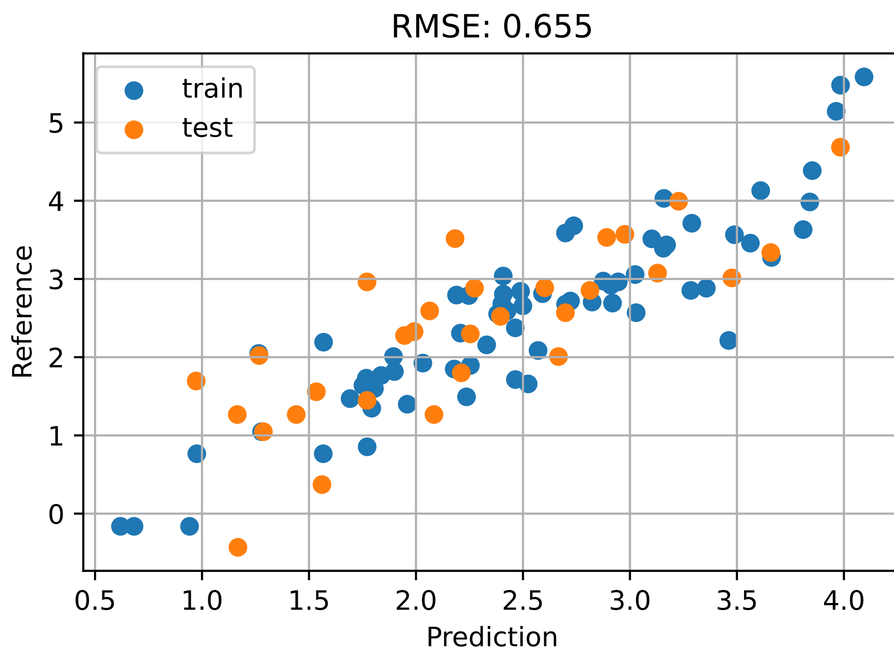
(b) Find the optimal hyperparameters

```
min_mae_index = np.unravel_index(
    np.argmin(rmse_values), rmse_values.shape
)
optimal_sigma = sigma_range[min_mae_index[0]]
optimal_regularization = regularization_range[min_mae_index[1]]
optimal_sigma, optimal_regularization
```

```
(3.79269019073225, 0.18329807108324336)
```

and generate a parity plot with said hyperparameters

```
gpr = GaussianKernel(
    sigma=optimal_sigma, regularization=optimal_regularization
)
gpr.fit(X_train, y_train)
y_predict = gpr.predict(X_test)
gpr.rmse(X_test, y_test)
fig = plt.figure()
plt.scatter(gpr.predict(X_train), y_train, label="train")
plt.scatter(y_predict, y_test, label="test")
plt.xlabel("Prediction")
plt.ylabel("Reference")
plt.title(f"RMSE: {gpr.rmse(X_test, y_test):.3f}")
plt.legend()
plt.grid();
```



We find that the RMSE is slightly less than with the ordinary least squares and linear ridge regression (which were around 0.69).