Module 4.3 Lecture 10 & 11. Multi-class regression

COMP 551 Applied machine learning

Yue Li
Assistant Professor
School of Computer Science
McGill University

Feb 6 & 11, 2025

Learning objectives

- Multi-class linear classifiers
  - Softmax function
  - Loss function
- Probabilistic view of multi-class classification
- Applications in toy data and Iris flower type classification

Reinforcing the concepts learned in the logistic regression lecture

The concepts of this lecture are very similar to the logistic regression lecture and serves as a way to enforce we have learned previously.

Outline

Objectives

Linear classifier for multiple classes

Learning multi-class regression by gradient descent

Toy data application

Application: Iris flower classification

Probabilistic view of multi-class regression

Summary

Midterm (Feb 19) cover up to this point

Multi-class classification

Classes are mutually exclusive (i.e., each data point belongs to one and exactly one of the K classes). An example is classifying Iris flowers using hand-crafted features.

(a)

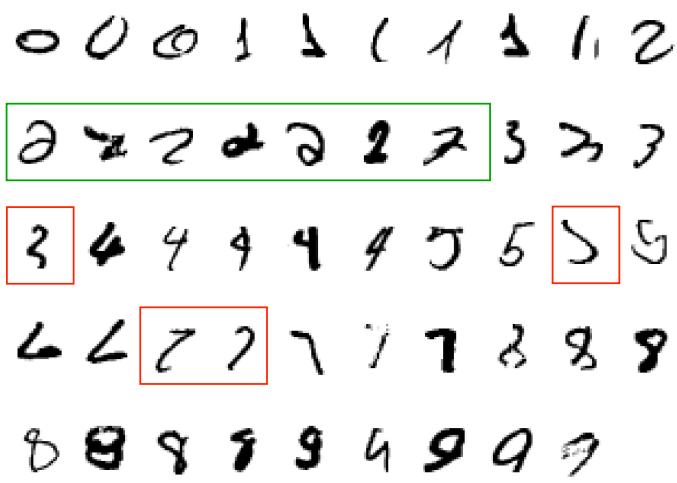
(b)

(c)

index	sepal length	sepal width	petal length	petal width	label
0	5.1	3.5	1.4	0.3	Setosa
1	4.9	3.0	1.7	0.2	Setosa
⋮	⋮	⋮	⋮	⋮	⋮
50	7.0	3.2	4.7	1.4	Versicolor
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	Virginica

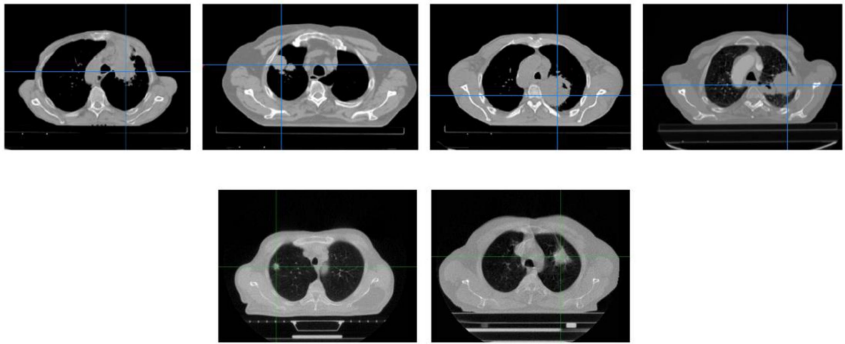
Multi-class classification example: MNIST digit prediction

Predicting digit $y \in \{0, \ldots, 9\}$ from 28×28 images of hand-written digits.



Multi-class example: predicting lung cancer stages

Predicting cancer stages of lung cancer patients using the Lung-RT dataset. Sample images are shown below. Top row: From left to right the stage I, II, IIIa and IIIb of squamous cell carcinomas. Bottom row: From left to right the stage I and II. The lung tumor position is indicated by the crossing lines.



Ubaldi, L., et al. "Strategies to develop radiomics and machine learning models for lung cancer stage and histology prediction using small data samples." Physica Medica 90 (2021): 13-22.

One-hot-encoding for categorical label and softmax function for prediction

Suppose we have C classes: $t^{(n)} \in \{1, \ldots, C\}$. It is computationally convenient to convert the categorical label to a **one-hot-encoding** vector where one of the K values is 1 and the rest of the values are zeros:

$$\mathbf{y}^{(n)} = \left[\mathbb{I}[t^{(n)} = 1], \quad \mathbb{I}[t^{(n)} = 2], \quad \ldots \mathbb{I}[t^{(n)} = C]\right] \quad \text{or} \quad y_c^{(n)} = \begin{cases} 1 & \text{if } t^{(n)} = c \\ 0 & \text{otherwise} \end{cases}$$

The linear function that predicts the score of the c^{th} class from a D-dimensional input feature vector $\mathbf{x}^{(n)}$ is:

$$a_c^{(n)} = \sum_{d=1}^{D} x_d^{(n)} w_{d,c} = \mathbf{x}^{(n)} \mathbf{w}_c$$

To convert $a_c^{(n)} \in \mathbb{R}$ into probability so that $\sum_c \hat{y}_c^{(n)} = 1$, we use **softmax** function:

$$\hat{y}_c^{(n)} = \frac{\exp(a_c^{(n)})}{\sum_{c'=1}^{C} \exp(a_{c'}^{(n)})}$$

For C classes, we will have C sets of linear coefficients $\mathbf{w}_c$ or $D \times C$ matrix $\mathbf{W}$.

Sigmoid function is a special case of softmax funtion when $C = 2$

The sigmoid function we learned in binary classification is a just special case of softmax when $C = 2$:

$$\hat{y}_1 = \frac{\exp(a_1)}{\exp(a_1) + \exp(a_2)} = \frac{1}{1 + \exp(a_2 - a_1)} = \frac{1}{1 + \exp(-(a_1 - a_2))}$$

where

$$a_1 - a_2 = \mathbf{x}\mathbf{w}_1 - \mathbf{x}\mathbf{w}_2 = \mathbf{x}(\mathbf{w}_1 - \mathbf{w}_2) \equiv \mathbf{x}\mathbf{w} = a$$

Therefore, for binary classification, we only need to learn *one* set of coefficients $\mathbf{w}$ that corresponds to the class $y = 1$.

The probability of $y = 0$ is simply $1 - \hat{y}$.

For $C > 2$ classes, we only need to learn $C - 1$ sets of coefficients

More generally, the C class probabilities need to sum to 1: $\sum_{c=1}^{C} \hat{y}_c = 1$. We only need to learn $C - 1$ sets of coefficients and derive the probability for the last class as follows:

$$\hat{y}_C = \frac{\exp(\mathbf{x}\mathbf{w}_C)}{\sum_{c=1}^{C} \exp(\mathbf{x}\mathbf{w}_c)} = \frac{1}{1 + \sum_{c=1}^{C-1} \exp(\mathbf{x}\mathbf{w}_c - \mathbf{x}\mathbf{w}_C)} = \frac{1}{1 + \sum_{c=1}^{C-1} \exp(\mathbf{x}\underbrace{(\mathbf{w}_c - \mathbf{w}_C)}_{\mathbf{w}_c^*})}$$

$$= 1 - \frac{\sum_{c=1}^{C-1} \exp(\mathbf{x}\mathbf{w}_c^*)}{1 + \sum_{c=1}^{C-1} \exp(\mathbf{x}\mathbf{w}_c^*)} = 1 - \sum_{c'=1}^{C-1} \hat{y}_{c'}$$

where $\mathbf{w}_c^*$ is the redefined coefficients for $c \neq C$. Therefore, for prediction, we only need to fit $D \times (C - 1)$ coefficients $\mathbf{W}$. This is equivalent to setting $\mathbf{w}_C^* = \mathbf{0}$ (because $\exp(\mathbf{x}\mathbf{w}_C^*) = 1$).

However, learning the full $D \times C$ matrix allows us to associate D features with every class including the C^{th} class via $\mathbf{w}_C$. In the remaining lecture, we will assume that we are fitting the full $D \times C$ matrix $\mathbf{W}$.

Loss function is still cross entropy but for multiple classes

For multi-class loss, we can write the cross entropy as follows:

$$J(\mathbf{W}) = -\sum_{c=1}^{C} y_c \log \hat{y}_c$$

Similar to the binary cross-entropy, the multi-class cross entropy also has probabilistic interpretation (Section 6).

When $C = 2$, we see this loss becomes the same as the binary cross entropy loss:

$$\begin{aligned} J(\mathbf{W}) &= -y_1 \log \hat{y}_1 - y_2 \log \hat{y}_2 \\ &= -y_1 \log \hat{y}_1 - (1 - y_1) \log(1 - \hat{y}_1) \\ &= -y^* \log \hat{y}^* - (1 - y^*) \log(1 - \hat{y}^*) \end{aligned}$$

The last equality assumes $y^* \in \{0, 1\}$.

Outline

Objectives

Linear classifier for multiple classes

Learning multi-class regression by gradient descent

Toy data application

Application: Iris flower classification

Probabilistic view of multi-class regression

Summary

Midterm (Feb 19) cover up to this point

Gradient derivation of multi-class linear regression (updated in Lec 11)

For simplicity, we will work with one example and omit the superscript n for now. Our goal is to minimize CE w.r.t. $w_{d,j} \in \mathbf{W} \in \mathbb{R}^{D \times C}$, where d indexes the d^{th} feature and c indexes the c^{th} class:

$$J(\mathbf{W}) = -\sum_{c=1}^{C} y_c \log \hat{y}_c, \quad \text{where} \quad \hat{y}_c = \frac{\exp(a_c)}{\sum_c \exp(a_c)} \quad \text{and} \quad a_c = \sum_{d=1}^{D} x_d w_{d,c}$$

Using chain rule, we can express the partial derivative w.r.t. $w_{d,j}$ for the j^{th} class and d^{th} feature:

$$\frac{\partial J}{\partial w_{d,j}} = \frac{\partial}{\partial w_{d,j}} - \sum_{c=1}^{C} y_c \log \hat{y}_c = -\sum_{c=1}^{C} \frac{\partial}{\partial w_{d,j}} y_c \log \hat{y}_c = \sum_{c=1}^{C} \frac{\partial -y_c \log \hat{y}_c}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial a_j} \frac{\partial a_j}{\partial w_{d,j}} \tag{1}$$

Note that here we need to sum over all classes to compute the gradient for $w_{d,j}$ because the gradients when $j = c$ and $j \neq c$ are different (more details follow).
Same as in the binary logistic regression, we solve each partial derivative one by one.

$$\frac{\partial}{\partial \hat{y}_c} - y_c \log \hat{y}_c = -\frac{y_c}{\hat{y}_c} \qquad (2) \qquad\qquad \frac{\partial a_j}{\partial w_{d,j}} = \frac{\partial}{\partial w_{d,j}} \sum_d x_d w_{d,j} = x_d \qquad (3)$$

$$\frac{\partial \hat{y}_c}{\partial a_j} = \frac{\partial}{\partial a_j} \frac{\exp(a_c)}{\sum_c \exp(a_c)} = \frac{\partial}{\partial a_j} \exp(a_c) \left[\sum_c \exp(a_c)\right]^{-1} + \exp(a_c) \frac{\partial}{\partial a_j} \left[\sum_c \exp(a_c)\right]^{-1}$$

If $j = c$:

$$\frac{\partial \hat{y}_c}{\partial a_j} = \frac{\exp(a_c)}{\sum_c \exp(a_c)} - [\exp(a_c)]^2 \left[\sum_c \exp(a_c)\right]^{-2}$$

$$= \hat{y}_c - \left[\frac{\exp(a_c)}{\sum_c \exp(a_c)}\right]^2 = \hat{y}_c - \hat{y}_c^2$$

If $j \neq c$:

$$\frac{\partial \hat{y}_c}{\partial a_j} = -\exp(a_c)\exp(a_j) \left[\sum_c \exp(a_c)\right]^{-2}$$

$$= -\frac{\exp(a_c)}{\sum_c \exp(a_c)} \frac{\exp(a_j)}{\sum_c \exp(a_c)} = -\hat{y}_c \hat{y}_j$$

Therefore,

$$\frac{\partial \hat{y}_c}{\partial a_j} = \begin{cases} \hat{y}_c(1 - \hat{y}_c) & \text{if } j = c \\ -\hat{y}_c \hat{y}_j & \text{if } j \neq c \end{cases} = \hat{y}_c(\mathbb{I}[j = c] - \hat{y}_j) \qquad (4)$$

Gradient of multi-class linear regression

Plugging (2), (4), (3) into (1), we have

$$\begin{aligned}\frac{\partial J}{\partial w_{d,j}} &= \sum_c \frac{\partial -y_c \log \hat{y}_c}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial a_j} \frac{\partial a_j}{w_{d,j}} \\ &= \sum_c -\frac{y_c}{\hat{y}_c} \hat{y}_c (\mathbb{I}[j=c] - \hat{y}_j) x_d \\ &= \sum_c y_c (\hat{y}_j - \mathbb{I}[j=c]) x_d \\ &= \left[\underbrace{(\sum_c y_c)}_{1} \hat{y}_j - \underbrace{\sum_c y_c \mathbb{I}[j=c]}_{y_j} \right] x_d \\ &= (\hat{y}_j - y_j) x_d\end{aligned}$$

Recall the gradient for logistic: $\frac{\partial J}{\partial w_d} = (\hat{y} - y) x_d$

Now we recover the subscript for N data points $\{\mathbf{x}^{(n)}, \mathbf{y}^{(n)}\}^N$. We can represent the gradients over *all D features* for *all C classes* as a *Jacobian matrix*:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{W}} &= \sum_{n=1}^{N} (\mathbf{x}^{(n)})^\top (\hat{\mathbf{y}}^{(n)} - \mathbf{y}^{(n)}) \\ &= \mathbf{X}^\top (\hat{\mathbf{Y}} - \mathbf{Y}) \qquad (5)\end{aligned}$$

where $(\mathbf{x}^{(n)})^\top$ is a $D \times 1$ vector, $\hat{\mathbf{y}}^{(n)}$ and $\mathbf{y}^{(n)}$ are $1 \times C$ vectors, $\mathbf{X} \in \mathbb{R}^{N \times D}$, and $\mathbf{Y}$ is a $N \times C$ binary matrix, where each row is a one-hot-encoded vector.

Pop quiz: is Eq (5) equivalent to updating each $\mathbf{w}_c$ separately as C separate logistic regression models?

Multi-class regression training algorithm by gradient descent

Similar to the logistic regression (and unlike linear regression), there is no closed form solution to the coefficients $\mathbf{W}$.
To update the multi-class regression model, we perform **gradient descent** by *subtracting* the gradients from the existing weight *iteratively*:

$$\mathbf{W}^{\{t\}} \leftarrow \mathbf{W}^{\{t-1\}} - \alpha \frac{\partial J(\mathbf{W}^{\{t-1\}})}{\partial \mathbf{W}^{\{t-1\}}}$$

- We subtract the gradients from the weights because we want to minimize the error function.
- We multiply the gradients by a learning rate $\alpha \in [0, 1]$ to avoid overshooting the optimal values.

Multi-class regression training algorithm by gradient descent

Algorithm 1 MulticlassRegression.fit($\mathbf{X}$, $\mathbf{Y}$, $\alpha = 0.005$, $\epsilon = 10^{-5}$, max_iters=10^5)

Randomly initialize regression coefficients $w_{d,c} \sim \mathcal{N}(0,1) \forall d \forall c$
for niter $= 1 \ldots$ max_iters **do**
    $\mathbf{W}^{\{t\}} \leftarrow \mathbf{W}^{\{t-1\}} - \alpha \mathbf{X}^\top (\hat{\mathbf{Y}} - \mathbf{Y})$
    for n $= 1, \ldots,$ N **do**
        for c $= 1, \ldots,$ C **do**
            $\hat{y}_c^{(n)} = \exp(\mathbf{x}^{(n)} \mathbf{w}_c^{\{t\}}) / \sum_c \exp(\mathbf{x}^{(n)} \mathbf{w}_c^{\{t\}})$
        end for
    end for
    $J(\mathbf{W}^{\{t\}}) = -\sum_n \sum_c y_c^{(n)} \log(\hat{y}_c^{(n)})$
    if $|J(\mathbf{W}^{\{t\}}) - J(\mathbf{W}^{\{t-1\}}|) < \epsilon$ **then**
        break // *Converged so we quite before completing all iterations*
    end if
end for

Key implementation of the Multi-class prediction class

```
class Multinomial_logistic:
    def __init__(self, nFeatures, nClasses):
        self.W = np.random.rand(nFeatures, nClasses)

    def predict(self, X):
        y_pred = np.exp(np.matmul(X, self.W))
        return y_pred / y_pred.sum(axis=1).reshape(X.shape[0], 1)

    def grad(self, X, y):
        return np.matmul(X.transpose(), self.predict(X) - y)

    def ce(self, X, y):
        return -np.sum(y * np.log(self.predict(X))) // element-wise

    def fit(self, X, y, lr=0.005, niter=100):
        for i in range(niter):
            self.W = self.W - lr * self.grad(X, y)
```

Outline

Objectives

Linear classifier for multiple classes

Learning multi-class regression by gradient descent

Toy data application

Application: Iris flower classification

Probabilistic view of multi-class regression

Summary

Midterm (Feb 19) cover up to this point

Toy data simulation

- Simulate a 3-class response $\mathbf{y}$ using a 4-dimensional binary input $\mathbf{X} \in \{0,1\}^{N \times D}$, which are sampled from Bernoulli with 0.5 rate: $x_d^{(n)} \sim \pi^{x_d^{(n)}}(1-\pi)^{1-x_d^{(n)}}$
- Features 2, 0, and (1,3) are the causal features of class 0, 1, and 2, respectively.
- The 4×3 true causal effects $\mathbf{W}$ are also binary with $w_{d,c} = 1$ if feature d is causal on class c, otherwise $w_{d,c} = 0$.
- For each example, its class label is set to be the class with the highest linear combination $y^{(n)} \leftarrow \arg\max_c \mathbf{x}^{(n)}\mathbf{w}_c$. Note that $\mathbf{y}^{(n)}$ is one-hot encoded.

```
N = 150
X = np.column_stack((np.random.binomial(1, 0.5, N),
                     np.random.binomial(1, 0.5, N),
                     np.random.binomial(1, 0.5, N),
                     np.random.binomial(1, 0.5, N)))
W_true = np.array([[0,1,0],
                   [0,0,1],
                   [1,0,0],
                   [0,0,1]])
a = np.matmul(X, W_true)
y = np.zeros_like(a)
y[np.arange(len(a)), a.argmax(1)] = 1
```

Splitting the data into 1/3 training, 1/3 validation, 1/3 testing

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size = 0.33, random_state=1, shuffle=True)

X_train, X_valid, y_train, y_valid =
↪  model_selection.train_test_split(
    X_train, y_train, test_size = 0.5, random_state=1, shuffle=True)
```

Here the validation is being used for monitoring whether the model starts to overfit.

Recall: we can verify gradient calculation via small perturbation

Gradient calculation by hand as shown above can be error-prone. We can verify the gradient as follow:

1. $\epsilon \sim Uniform([0, 1])$
2. $w_{d,c}^{(+)} = w_{d,c} + \epsilon$
3. $w_{d,c}^{(-)} = w_{d,c} - \epsilon$
4. $\nabla w_{d,c} = \frac{J(w_{d,c}^{(+)}) - J(w_{d,c}^{(-)})}{2\epsilon}$ (numerically estimated gradient)
5. $\frac{(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} - \nabla w_{d,c})^2}{(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} + \nabla w_{d,c})^2}$ must be small (e.g., 10^{-8}) otherwise your gradient calculation and/or your loss function is incorrect

```
mlr = Multinomial_logistic(D, C)
print(mlr.check_grad(X_train, y_train)) # expected very small value
```

In Python

```python
def check_grad(self, X, y):
    N, C = y.shape
    D = X.shape[1]
    diff = np.zeros((D, C))
    W = self.W.copy()

    for i in range(D):
        for j in range(C):
            epsilon = np.zeros((D, C))
            epsilon[i, j] = np.random.rand() * 1e-4

            self.W = self.W + epsilon
            J1 = self.ce(X, y)
            self.W = W # restore original W

            self.W = self.W - epsilon
            J2 = self.ce(X, y)
            self.W = W # restore original W

            numeric_grad = (J1 - J2) / (2 * epsilon[i, j])
            derived_grad = self.grad(X, y)[i, j]

            diff[i, j] = np.square(derived_grad - numeric_grad).sum() /
            np.square(derived_grad + numeric_grad).sum()
    return diff.sum()
```

Modifying the `fit` to monitor training and validation errors

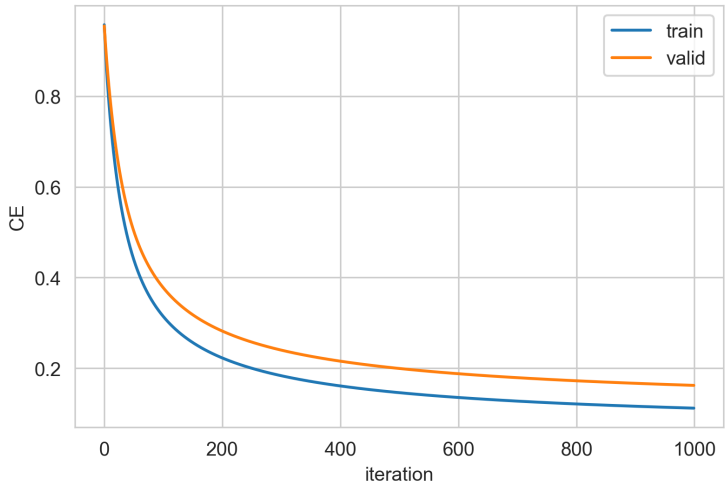
```python
class Multinomial_logistic:
    # modify it to add stopping criteria (what can you think of?)
    def fit(self, X, y, X_valid=None, y_valid=None, lr=0.005, niter=100):
        losses_train = np.zeros(niter)
        losses_valid = np.zeros(niter)
        for i in range(niter):
            self.W = self.W - lr * self.grad(X, y)
            loss_train = self.ce(X, y)
            losses_train[i] = loss_train
            if X_valid is not None and y_valid is not None:
                loss_valid = self.ce(X_valid, y_valid)
                losses_valid[i] = loss_valid
                print(f"iter {i}: {loss_train:.3f}; {loss_valid:.3f}")
            else:
                print(f"iter {i}: {loss_train:.3f}")
        return losses_train, losses_valid
```

Fitting multiclass logistic regression for 1000 iterations with lr set to 0.005

```
ce_train, ce_valid = mlr.fit(X_train, y_train, X_valid, y_valid,
↪  niter=1000)

plt.clf()
plt.plot(ce_train/X_train.shape[0], label='train') // avg train CE
plt.plot(ce_valid/X_valid.shape[0], label='valid') // avg valid CE
plt.xlabel("iteration")
plt.ylabel("CE")
plt.legend()
# plt.show()
plt.savefig("training_ce.png", bbox_inches="tight", dpi=300)
```

Fitting multiclass logistic regression for 1000 iterations with lr set to 0.005

From the plot, we can see that the training and validation error curves both continue to decrease. Therefore, there is no sign of overfitting.



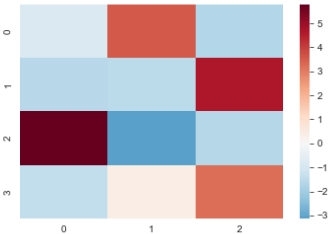
Evaluating prediction accuracy

Prediction accuracies are 100% for training, validation, and test datasets, which is not surprising since our data are generated by a linear function.

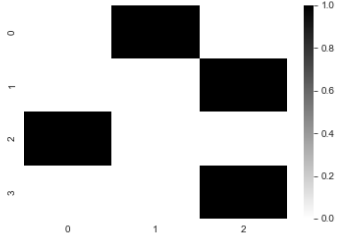
```
def evaluate(y, y_pred):
    accuracy = sum(y_pred.argmax(axis=1) == y.argmax(axis=1))
    accuracy = accuracy / y.shape[0]
    return accuracy

train_accuracy = evaluate(mlr.predict(X_train), y_train)
valid_accuracy = evaluate(mlr.predict(X_valid), y_valid)
test_accuracy = evaluate(mlr.predict(X_test), y_test)

print(train_accuracy) # 1
print(valid_accuracy) # 1
print(test_accuracy) # 1
```

Visualizing the linear coefficients using `seaborn.heatmap`

Fitted coefficient $\hat{\mathbf{W}}$:



True coefficient $\mathbf{W}$:

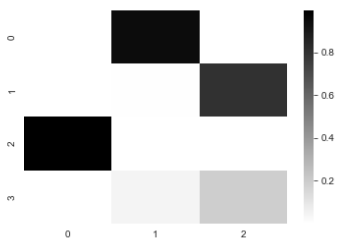


Transforming the fitted coefficient by softmax:

$$\hat{w}^*_{d,c} = \frac{\exp(\hat{w}_{d,c})}{\sum_d \exp(\hat{w}_{d,c})} \quad \Longrightarrow$$

```
1 W_hat = np.exp(W_hat)
2 W_hat = W_hat /
  ↪  W_hat.sum(axis=1)[:,None]
```

Softmaxed fitted coefficient $\hat{\mathbf{W}}^*$:



Outline

Objectives

Linear classifier for multiple classes

Learning multi-class regression by gradient descent

Toy data application

Application: Iris flower classification

Probabilistic view of multi-class regression

Summary

Midterm (Feb 19) cover up to this point

Iris flower classification

$C = 3$, $D = 4$, $N = 150$

(a)

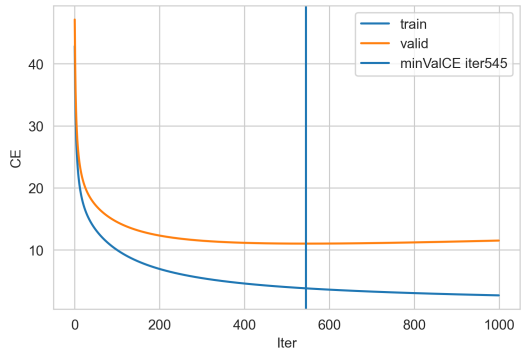
(b)

(c)

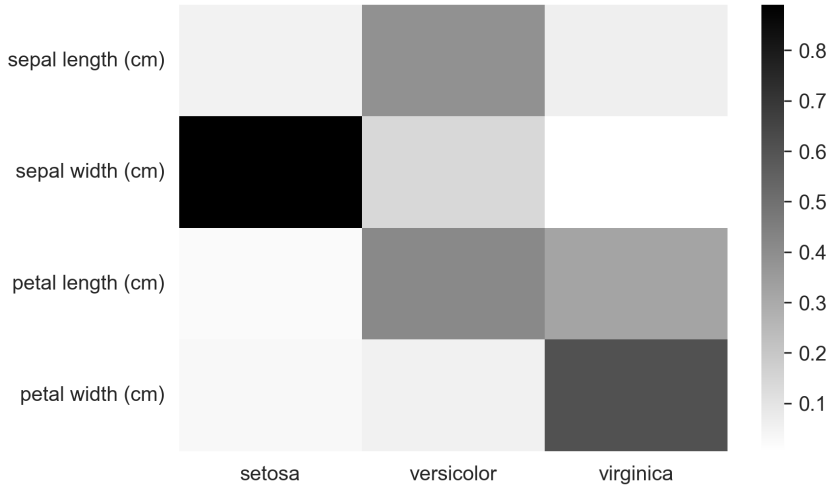
index	sepal length	sepal width	petal length	petal width	label
0	5.1	3.5	1.4	0.3	Setosa
1	4.9	3.0	1.7	0.2	Setosa
⋮	⋮	⋮	⋮	⋮	⋮
50	7.0	3.2	4.7	1.4	Versicolor
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	Virginica

Iris flower classification 1/3 training, 1/3 validation, 1/3 testing

- After 1000 iterations, we observe a slight overfitting as the training error continues to decrease but the validation error starts to increase at around 550 iterations.
- The best model with the lowest validation error is at iteration `optimal_niter = ce_valid.argmin()` (i.e., at iteration 538).
- Retraining the model for `optimal_niter`, we obtained 98% accuracy on training; 90% on validation, and 96% on the testing sets.



Feature-flower types association via $softmax(\hat{\mathbf{W}})$



Outline

Objectives

Linear classifier for multiple classes

Learning multi-class regression by gradient descent

Toy data application

Application: Iris flower classification

Probabilistic view of multi-class regression

Summary

Midterm (Feb 19) cover up to this point

Maximizing log multinomial likelihood is equivalent to minimizing CE

Categorical distribution has the following probability mass function (PMF):

$$p(y|\boldsymbol{\pi}) = \prod_c \pi_c^{[y=c]} \tag{6}$$

Representing y as the one-hot encoded binary vector, replacing π_c by $\hat{y}_c$, and taking the logarithm of the Categorical likelihood give:

$$\log p(\mathbf{y}|\hat{\mathbf{y}}) = \sum_c y_c \log \hat{y}_c \tag{7}$$

Multinomial distribution models the outcome of multiple categorical trials (i.e., N training examples):

$$\log p(\mathbf{Y}|\hat{\mathbf{Y}}) = \sum_{n=1}^{N} \sum_{c=1}^{C} y_c^{(n)} \log \hat{y}_c^{(n)} = -J(\mathbf{W}) \tag{8}$$

For this reason, the multi-class logistic is often known as *multinomial logistic regression* or *multinomial regression*. Pop quiz: Which line(s) do we need to change in Slide 16 if we are to maximize the Categorical likelihood?

Summary

- Multiclass prediction model:
  - Multiclass function: softmax
  - Cross-entropy (CE) loss
  - Gradient descent
  - The $D \times C$ $\mathbf{W}$ linear coefficient matrix offers interpretability for associations between features and classes
- Probabilistic interpretation
  - Categorical or multinomial distribution
  - Maximum likelihood estimate of Multinomial is equivalent to minimizing CE loss
  - Recall:
    - Logistic regression: MLE of log Bernoulli likelihood is equivalent to minimizing CE loss
    - Linear regression: MLE of log Gaussian likelihood is equivalent to minimizing SSE loss
- Applications and interpretations of logistic regression linear coefficients on the toy and Iris flower datasets

Outline

Objectives

Linear classifier for multiple classes

Learning multi-class regression by gradient descent

Toy data application

Application: Iris flower classification

Probabilistic view of multi-class regression

Summary

Midterm (Feb 19) cover up to this point

Midterm exam on Feb 19 in class

Format:

- 1 hour in-class;
- Open book, calculator allowed but not digital one, no tablet, cell phone or laptop
- Format is the same as the practice midterm;
- Hand-written;
- 20% MC (6 questions); 30% MS (4 questions); 50% SA (6 questions).

Understand the basics. For example,

- Basic ML experiments: training, validation, testing
- K-fold cross validation
- Model evaluation: accuracy, confusion table, TRP, FPR, Precision, Recall, ROC
- Overfitting
- Algorithms and costs functions for KNN, Decision tree, linear regression, basis functions, logistic regression, and multiclass regression
- Partial derivative