hw1

March 30, 2024

1 Homework 1:

STATS348, UChicago, Spring 2024

Your name here:
Daniel Li

1.1 Instructions

The purpose of this homework is to apply the concepts raised in week 1 on supervised learning and decision problems:

- overfitting / underfitting
- CV and model selection
- logistic regression
- KNNs

This homework will also get you familiar with Python and the scikit-learn package.

For reference, this homework is a close adaption of homework 1 from the 2021 version of STAT348.

Assignment is due **Saturday March 30, 11:59pm** on GradeScope.

1.2 Problem 1: Elephants

Read through Matthew Stephens' vignette on classifying savannah versus forest elephants and then do exercises 2a and 2b from the vignette, which are copied here (and relabeled 1a and 1b).

> 1a) Perform the following simulation study. Simulate 1000 tusks (values of x) from each of the models M_S and M_F. For each simulated tusk compute the LR for M_S vs M_F, so you have computed 2000 LRs. Now consider using the LR to classify each tusk as being from a savanna or a forest elephant. Recall that large values for LR indicate support for M_S, so a natural classification rule is "classify as savanna if LR $> c$, otherwise classify as forest" for some threshold c. Plot the misclassification rate (= number of tusks wrongly classified/2000) for this rule, as c ranges from 0.01 to 100. What value of c minimizes the misclassification rate? [Hint: the plot will look best if you do things on the log scale, so you could let $\log_{10}(c)$ vary from -2 to 2 using an equally spaced grid, and plot the misclassification rate on the y-axis against $\log_{10}(c)$ on the x-axis.]

1b) Repeat the above simulation study using 100 tusks from MS and 1900 tusks from MF. What value of c minimizes the misclassification rate? Comment.

To complete this problem in Python, here are some useful tools: - To plot you can use matplotlib and seaborn. You can see the week 1 notebook for examples.

- To sample random variables you can use numpy.random

[192]:
```python
import numpy as np
import numpy.random as rn

# sample a Bernoulli with probability p
p = 0.5
x = rn.binomial(1, p)

# sample n Bernoullis iid with probability p
n = 100
x = rn.binomial(1, p, size=n)

# sample n Bernoullis independently with different probabilities
p = np.array([0.1, 0.4, 0.5, 0.1, 0.9])
x = rn.binomial(1, p)

# this last example uses broadcasting.
# See here: https://numpy.org/doc/stable/user/basics.broadcasting.html
```

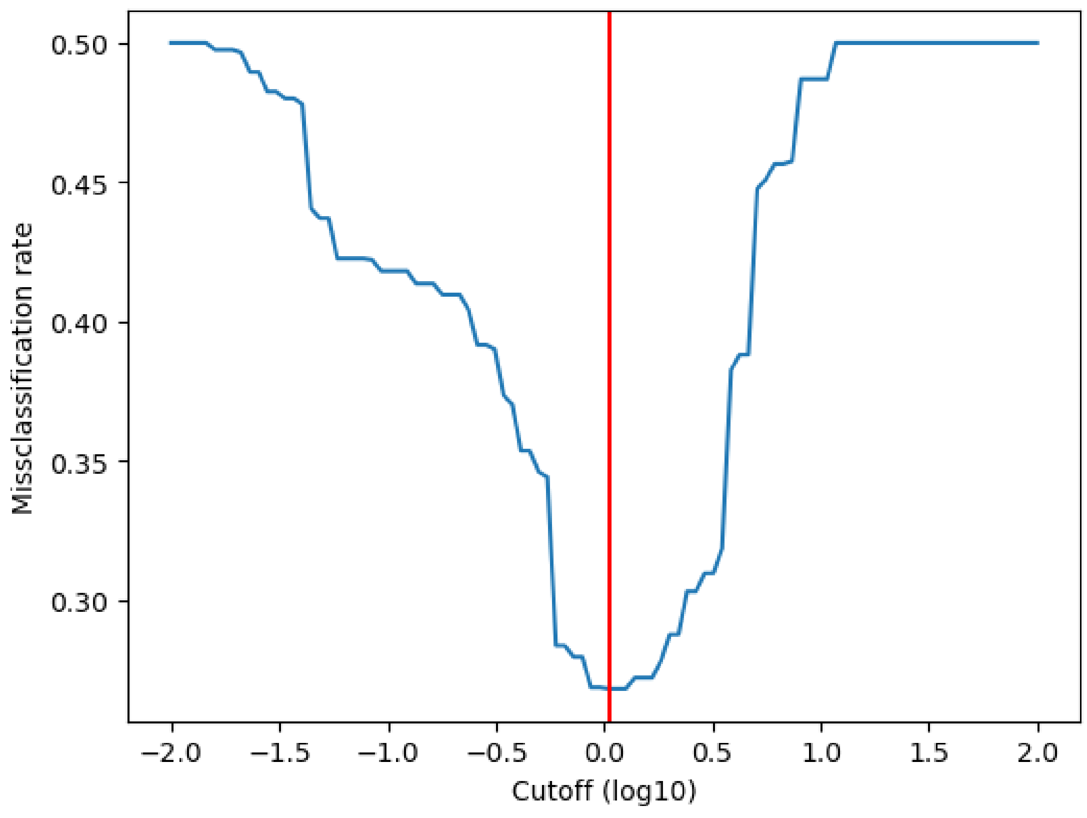
To complete the first part of this problem, you should complete the following functions.

[193]:
```python
def simulate_tusks_forest(size=1000):
    """Samples from the likelihood of P(x | forest).

    Parameters
    ----------
    size : int
        The number of samples to draw.
    """
    p = [0.8, 0.2, 0.11, 0.17, 0.23, 0.25]
    return rn.binomial(1, p, size=(size, len(p)))


def simulate_tusks_savannah(size=1000):
    """Samples from the likelihood of P(x | savannah).

    Parameters
    ----------
    size : int
        The number of samples to draw.
    """
    p = [0.4, 0.12, 0.21, 0.12, 0.02, 0.32]
    return rn.binomial(1, p, size=(size, len(p)))


def likelihood_forest(x):
    """Computes the likelihood of the data under the M_F model (i.e., given␣
↪that the elephant is forest elephant)."""
    p = np.array([0.8, 0.2, 0.11, 0.17, 0.23, 0.25])
    powers = p**x * (1 - p) ** (1 - x)
    return np.prod(powers, axis=1)


def likelihood_savannah(x):
    """Computes the likelihood of the data under the M_S model (i.e., given␣
↪that the elephant is a savannah elephant)."""
    p = np.array([0.4, 0.12, 0.21, 0.12, 0.02, 0.32])
    powers = p**x * (1 - p) ** (1 - x)
    return np.prod(powers, axis=1)
```

- **1a)**: Use the functions above to perform the simulations and generate the plots in 1a.

  Use code block below for this. The output should display the plot(s), and show which c minimizes the misclassification rate.

[194]:
```
# Your code here for 1a.

import matplotlib.pyplot as plt


def classify(x, cutoff):
    likelihood_ratio = likelihood_savannah(x) / likelihood_forest(x)
    return likelihood_ratio > cutoff


def compute_misclassification_rate(data, truth, cutoff):
    predictions = classify(data, cutoff)
    return 1 - np.mean(predictions == truth)


def generate_data(n_forest, n_savannah):
    forest_data = simulate_tusks_forest(n_forest)
    savannah_data = simulate_tusks_savannah(n_savannah)
    data = np.vstack([forest_data, savannah_data])
    truth = np.hstack([np.zeros(n_forest), np.ones(n_savannah)])
    return data, truth


def plot_accuracy(data, truth, log_cutoffs):
    cutoffs = 10**log_cutoffs
    mr = [compute_misclassification_rate(data, truth, cutoff) for cutoff in␣
 ↪cutoffs]
    plt.plot(log_cutoffs, mr)
    plt.xlabel("Cutoff (log10)")
    plt.ylabel("Missclassification rate")
    plt.axvline(x=log_cutoffs[np.argmin(mr)], color="red")
    plt.show()


def simulation_experiment(n_forest, n_savannah, log_cutoffs):
    data, truth = generate_data(n_forest, n_savannah)
    plot_accuracy(data, truth, log_cutoffs)


n_forest = 1000
n_savannah = 1000
log_cutoffs = np.linspace(-2, 2, 100)
simulation_experiment(n_forest, n_savannah, log_cutoffs)
```

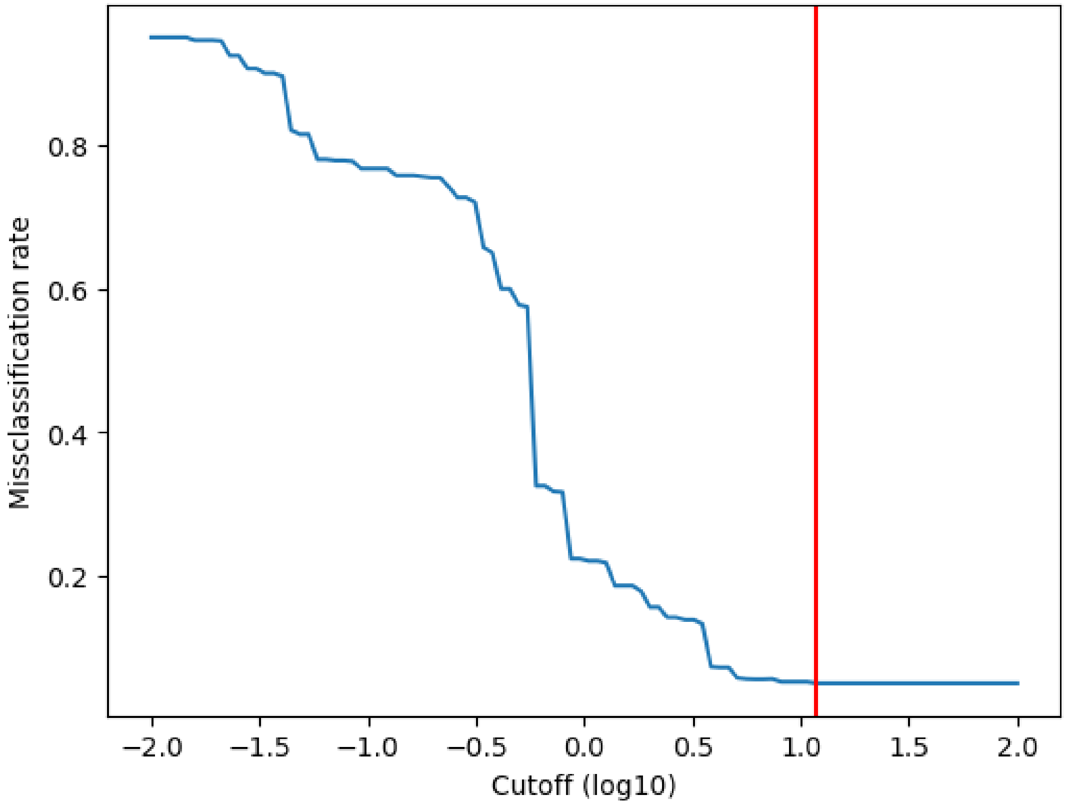


- **1b)** Now do the same for 1b.

  Use code block below for this. The output should display the plot(s), and show which c

minimizes the misclassification rate.

[195]:
```
n_forest = 1900
n_savannah = 100
log_cutoffs = np.linspace(-2, 2, 100)
simulation_experiment(n_forest, n_savannah, log_cutoffs)
```



1.3 Problem 2: Digits

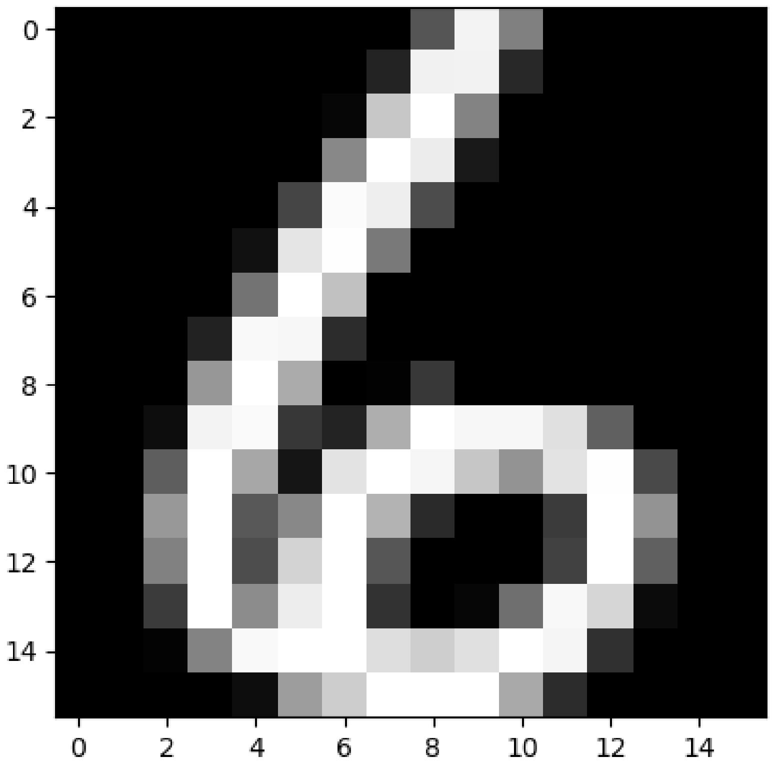
Consider the zipcode data from *Elements of Statistical Learning* (ESL). Note there is both a train and test set.

- **2a)** Download the data and try plotting a few examples of the training data as 16 x 16 images to see if you can see the digits visually as expected. [Hint: Use matplotlib's imshow function.]

[196]:
```
import pandas as pd

train = pd.read_csv("zip.train.gz", sep=" ", header=None)
train.drop(columns=[257], inplace=True)
test = pd.read_csv("zip.test.gz", sep=" ", header=None)
```

```
def plot_random_digit():
    digit = train.sample(1)
    plt.imshow(digit.values[0][1:].reshape(16, 16), cmap="gray")
    plt.show()
```

[337]:
```
plot_random_digit()
```



- **2b)** Consider the problem of trying to distinguish the digit 2 from the digit 3. Use the training data to learn classifiers, using:
  - logistic regression (un-regularized)
  - K nearest neighbors (K-NNs), with $K = 1, 3, 5, 7, 15$.

  This gives 6 classifiers in total.

  To complete this in Python you will want to use scikit-learn, and refer to the week 1 notebook for examples.

[198]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

train_two_three = train[(train[0] == 2) | (train[0] == 3)]
y = train_two_three[0]
X = train_two_three.drop(columns=[0])
```

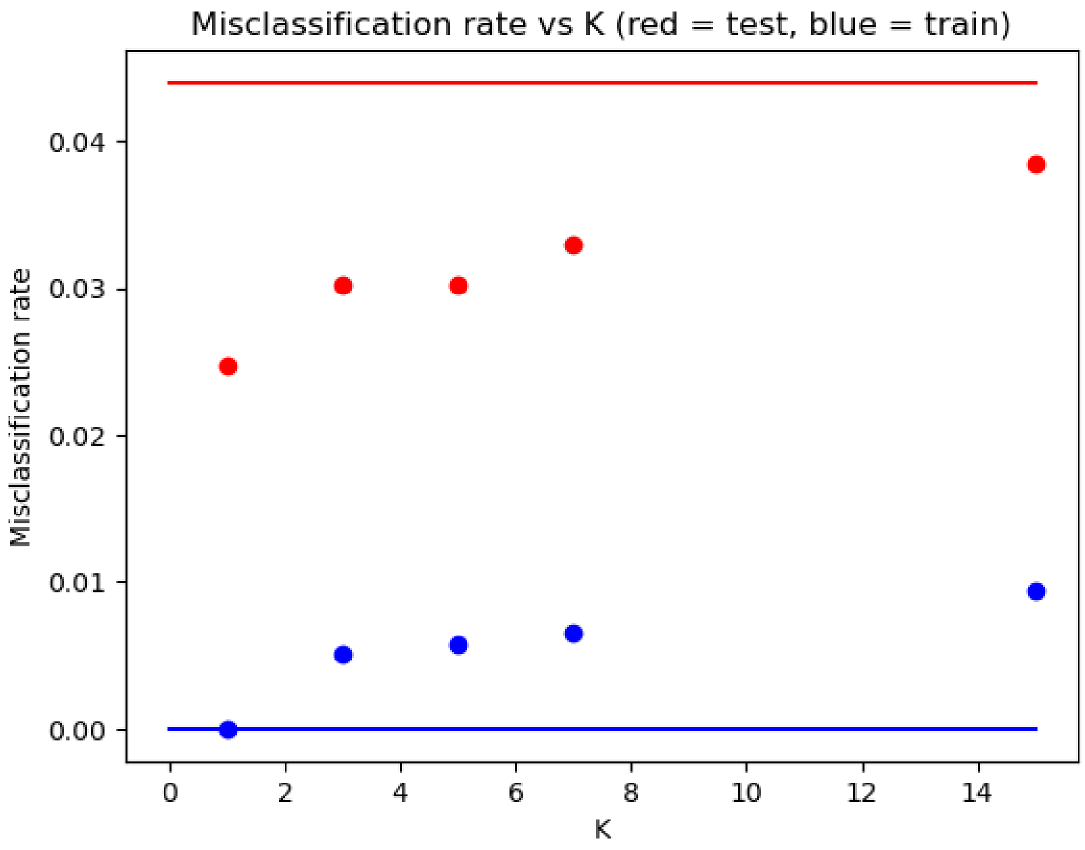
```
test_two_three = test[(test[0] == 2) | (test[0] == 3)]
y_test = test_two_three[0]
X_test = test_two_three.drop(columns=[0])

logistic = LogisticRegression(penalty="none")
logistic.fit(X, y)

knn = {}
K = [1, 3, 5, 7, 15]
for k in K:
    knn[k] = KNeighborsClassifier(n_neighbors=k)
    knn[k].fit(X, y)
```

- **2c)** Apply these classifiers to the test data, and plot the misclassification rates for both training data and test data. (Plot the results for K-NN with K on x-axis, and misclassification rate on y-axis, with two different colors for test and training sets. Then put appropriately colored horizontal lines on the same plot—one for test and one for train—indicating the results for logistic regression.)

  Your code in the cell below should output this plot.

[199]:
```
from sklearn.metrics import accuracy_score


def misclassification_rate(model, X, y):
    return 1 - accuracy_score(y, model.predict(X))


def create_plot(logistic, K, knnmodels, train_X, train_y, test_X, test_y):
    test_mr = [misclassification_rate(knnmodels[k], test_X, test_y) for k in K]
    train_mr = [misclassification_rate(knnmodels[k], train_X, train_y) for k in␣
 ↪K]
    plt.scatter(K, test_mr, color="red")
    plt.scatter(K, train_mr, color="blue")
    x = np.linspace(0, 15, 100)
    logi_test_mr = misclassification_rate(logistic, test_X, test_y)
    logi_train_mr = misclassification_rate(logistic, train_X, train_y)
    plt.plot(x, [logi_test_mr] * len(x), color="red")
    plt.plot(x, [logi_train_mr] * len(x), color="blue")
    plt.xlabel("K")
    plt.ylabel("Misclassification rate")
    plt.title("Misclassification rate vs K (red = test, blue = train)")
    # add legend
    plt.show()
```

[200]:
```
create_plot(logistic, K, knn, X, y, X_test, y_test)
```



- **2d)** Repeat the K-NN training as above, but using cross validation (CV) *on the training set* to tune K. That is, act like you do not have access to the test data and have to decide what K to use. How does it do?

  Again, for this problem you will want to use scikit-learn's methods for cross validation.

  Please add code in the cell below, and comment on the results in the space below.

Your text answer here: It finds 5 as the best n_neighbors

---

[201]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {"n_neighbors": K}
knn_grid = KNeighborsClassifier()
grid = GridSearchCV(knn_grid, param_grid, cv=5)
grid.fit(X, y)
print(grid.best_params_)
```

```
{'n_neighbors': 5}
```

- **2e)** Suppose now that for some reason it is considered worse to misclassify a 2 as a 3 than vice versa. Specifically, suppose you lose 5 points every time you misclassify a 2 as a 3, but 1 point every time you misclassify a 3 as a 2. Modify your logistic regression classifier to take account of this new loss function. Compute the new loss on the test set for both the modified classifier and the original logistic classifier.

Please add code in the cell below, and provide a brief description / justification of your code in the space below.

Your text answer here: I use three fold CV to tune the decision threshold so that it optimizes the modified loss on the validation. I then use the average of the best thresholds across the five fitted models on a model fit on the entire test data. This should average out values of the loss and give us the most generalizable value of the threshold.

---

[202]:
```
from sklearn.metrics import make_scorer, confusion_matrix
from sklearn.model_selection import KFold


def custom_score(y_true, y_pred, w=5):
    cm = confusion_matrix(y_true, y_pred)
    score = w * cm[0][1] + cm[1][0]
    return score


def pred_threshold(model, X, t):
    return np.where(model.predict_proba(X)[:, 1] > t, 3, 2)


def modified_logistic(X_train, y_train, X_test, mod="logistic", w=5, **kwargs):
    kf = KFold(n_splits=3)
    kf.get_n_splits(X_train)
    ts = []
    for _, (train_index, test_index) in enumerate(kf.split(X_train)):
        X_curr_train, X_curr_test = X_train.iloc[train_index], X_train.
 ↪iloc[test_index]
        y_curr_train, y_curr_test = y_train.iloc[train_index], y_train.
 ↪iloc[test_index]
        if mod == "logistic":
            model = LogisticRegression(penalty="none")
        if mod == "knn":
            model = KNeighborsClassifier(n_neighbors=kwargs["n_neighbors"])
        model.fit(X_curr_train, y_curr_train)
        t = np.linspace(0, 1, 100)
        scores = []
        for i in t:
            y_pred = pred_threshold(model, X_curr_test, i)
            scores.append(custom_score(y_curr_test, y_pred, w))
        ts.append(t[np.argmin(scores)])
    model = LogisticRegression(penalty="none")
    model.fit(X_train, y_train)
    t = np.mean(ts)
    return pred_threshold(model, X_test, t), model, t


print(
    "original logistic regression loss: ",
    custom_score(y_test, logistic.predict(X_test)),
)
y_pred, model, t = modified_logistic(X, y, X_test)
print("modified logistic regression loss: ", custom_score(y_test, y_pred))
```

```
original logistic regression loss:  52
modified logistic regression loss:  38
```

- **2f)** As far as you can, repeat this for the K-NN classifiers (i.e. modify them for the new loss function and compare the loss for modified vs original classifiers). Discuss any challenges you face here.

Please add code in the cell below, and provide a discussion of any challenges in the space below.

Your text answer here: The challenge is that probabilities returned by knn change drastically based on the train data because it use the neighbors. So sometimes you get prob of 0,1 which means tuning the threshold isn't helpful. So the method for CV doesn't work

---

[203]:
```
for k in K:
    orig_score = custom_score(y_test, knn[k].predict(X_test))
    y_pred, model, t = modified_logistic(X, y, X_test, mod="knn", n_neighbors=k)
    mod_score = custom_score(y_test, y_pred)
    print(f"original knn, k = {k}, loss: ", orig_score)
    print(f"modified knn k = {k} loss: ", mod_score)
```

```
original knn, k = 1, loss:  33
modified knn k = 1 loss:  990
original knn, k = 3, loss:  39
modified knn k = 3 loss:  53
original knn, k = 5, loss:  39
modified knn k = 5 loss:  53
original knn, k = 7, loss:  48
modified knn k = 7 loss:  53
original knn, k = 15, loss:  58
modified knn k = 15 loss:  48
```

1.4 Problem 3: Multiclass digits

Continuing with the zipcode data, now consider distinguishing the digits 1, 2, and 3.

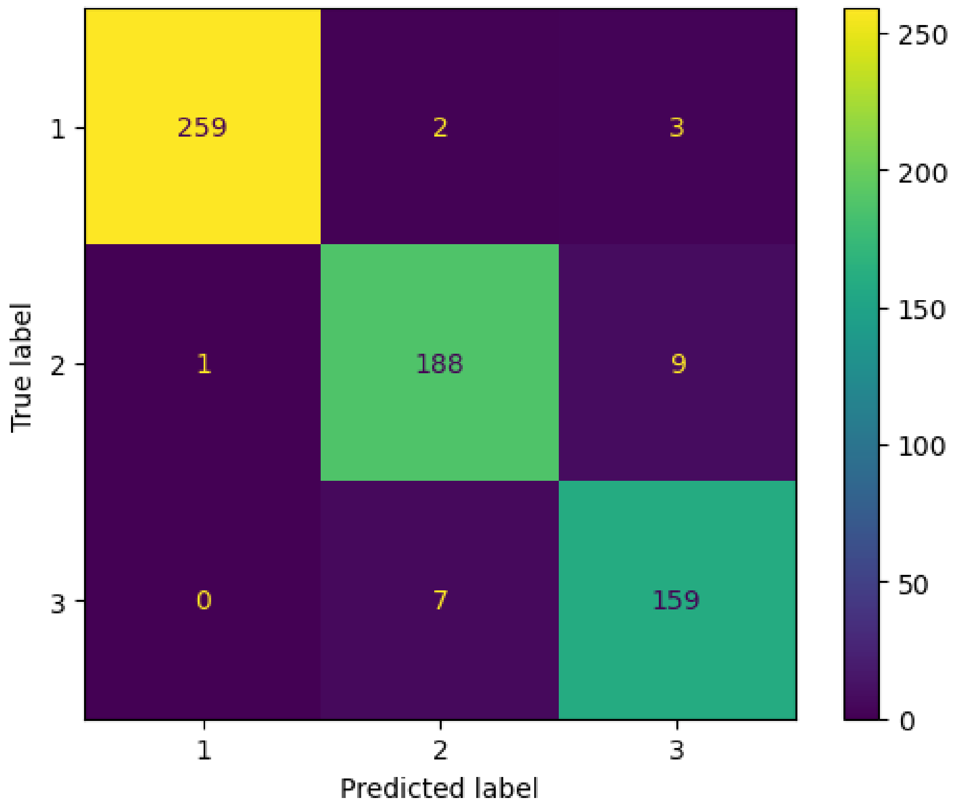
- For this problem, you will be generalizing the things we discussed in class about binary classification to **multiclass classification**, using multinomial logistic regression.
- Read Section 4.3.5 of *An Introduction to Statistical Learning with Applications in Python* on multinomial logistic regression for background.
- You can create a multinomial logistic regression model using scikit-learn as follows:

[212]:
```
from sklearn.linear_model import LogisticRegression

# an (unregularized) multinomial logistic regression model
logreg = LogisticRegression(
    penalty="none", solver="newton-cg", multi_class="multinomial"
)
```

- **3a)** Fit a multinomial logistic regression model to the training data of 1s, 2s, and 3s. Then apply it to the test set, and then calculate and plot the **confusion matrix**.

[317]:
```
# Your code here for 3a
train_one_two_three = train[(train[0] == 1) | (train[0] == 2) | (train[0] == 3)]
test_one_two_three = test[(test[0] == 1) | (test[0] == 2) | (test[0] == 3)]
X = train_one_two_three.drop(columns=[0])
y = train_one_two_three[0]
X_test = test_one_two_three.drop(columns=[0])
y_test = test_one_two_three[0]
from sklearn.metrics import ConfusionMatrixDisplay

logreg.fit(X, y)
y_pred = logreg.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[1, 2, 3])
disp.plot()
```

[317]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x17eb5ead0>
```



- **3b)** Suppose now that for some reason it is considered twice as bad to misclassify a 1 as a 2 than to make any other misclassification. Modify your multinomiallogistic regression classifier to take account of this new loss function. Compute the new loss on the test set for both the modified classifier and the original logistic classifier.

Please add code in the cell below, and provide justification of your code in the space below, including any derivations you had to do.

Your text answer here: For this, we will make our decision on picking $\hat{y}$ to minimize the expected loss, let C be the cost matrix, then

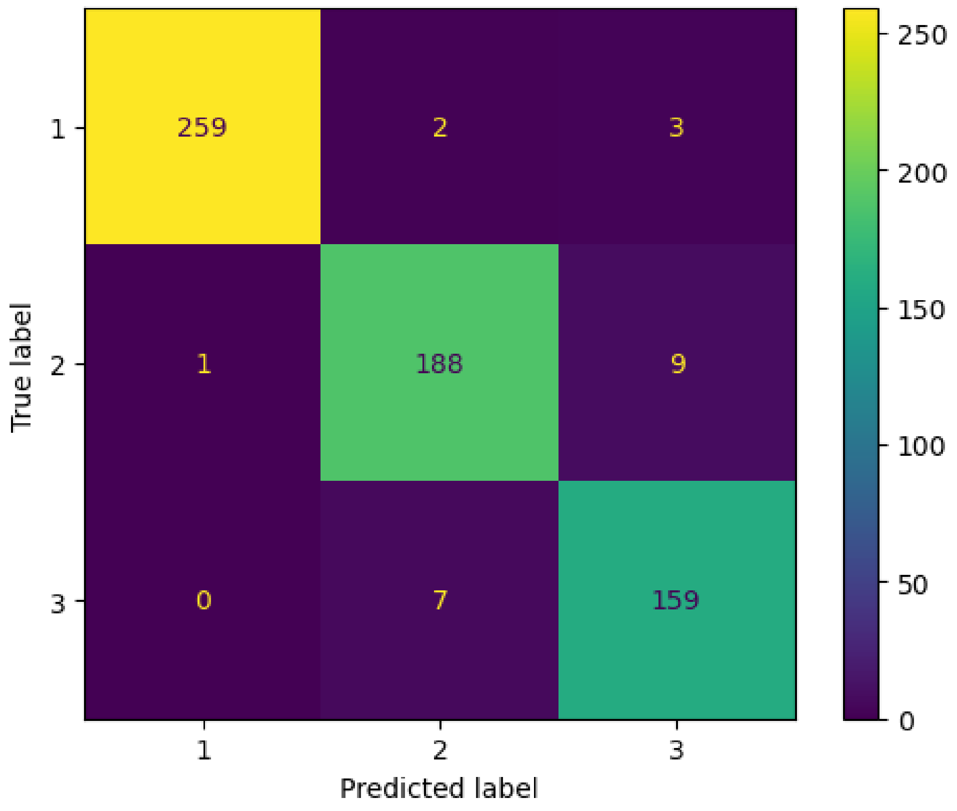
$$\hat{y} = \text{argmin}_{j \in (1,2,3)} \sum_{i=1}^{3} P(Y = i | X = x) C_{i,j}$$

---

[321]:
```
# Your code here for 3b

C = np.array([[0, 2, 1], [1, 0, 1], [1, 1, 0]])


def modified_multlogreg(model, X):
    prob = model.predict_proba(X)
    return np.argmin(np.dot(prob, C), axis=1) + 1


def new_loss(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    return np.sum(cm * C)


orig_loss = new_loss(y_test, y_pred)
y_pred_mod = modified_multlogreg(logreg, X_test)
mod_loss = new_loss(y_test, y_pred_mod)
print("original loss: ", orig_loss)
print("modified loss: ", mod_loss)
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[1, 2, 3])
disp.plot()
```

```
original loss:  24
modified loss:  24
```

```
[321]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x17f0285e0>
```

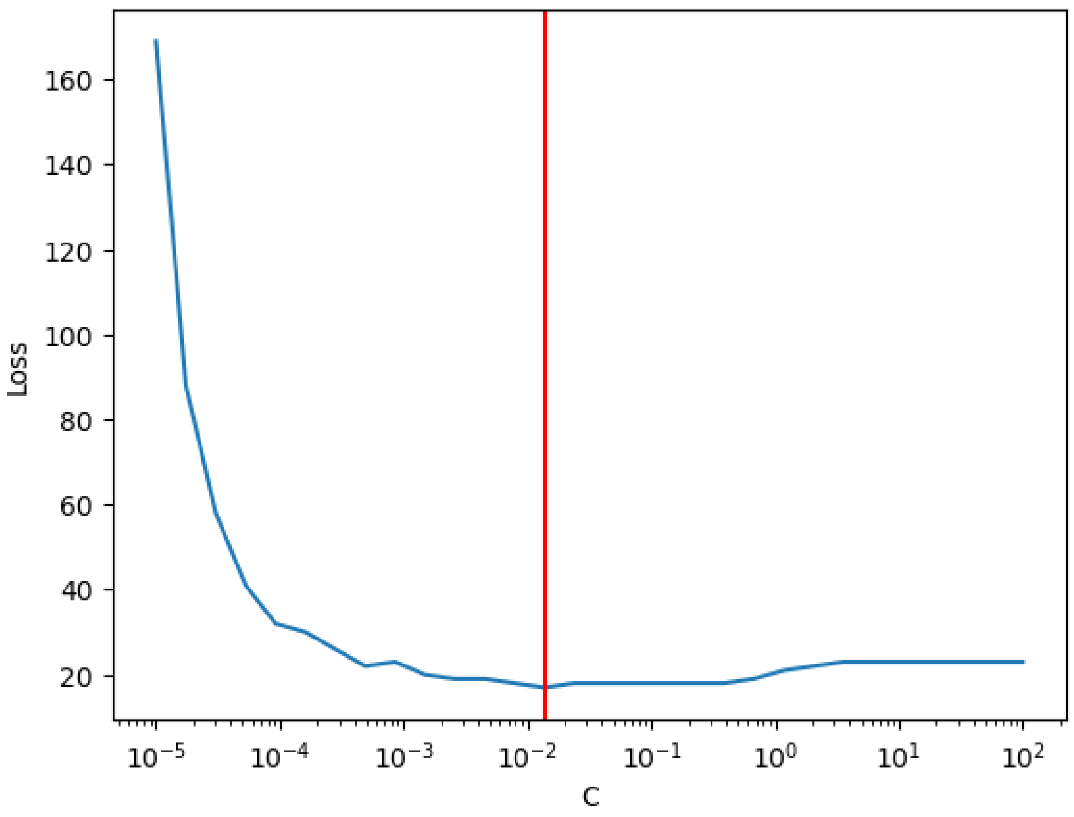


- **3c)** Now consider adding ℓ_2 regularization to your multinomial logistic regression classifier and answer the question: does it improve performance? Devise some experiments to answer this question convincingly. Use the code block below to implement and run those experiments, and to generate plot(s) that convey the results. Use the space below to briefly describe and justify your experiments, to summarize the results, and to speculate on why regularization does or does not help in this setting.

Your text answer here: Stronger regularization (lower C) does help with loss up to a certain point. My experiment fits the logistic regression with stronger and stronger penalty. One can see that the loss does first decrease before increasing again. The red line denotes the value of C with the best loss. Regularization should help because usually the edges or corners are black or very dark, so those pixels don't contribute information towards the number.

---

[330]:
```python
# Your code here for 3c
C = np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])


def loss(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    return np.sum(cm * C)


logregreg = LogisticRegression(
    penalty="l2", solver="newton-cg", multi_class="multinomial"
)
cs = 10 ** np.linspace(-5, 2, 30)
losses = []
for c in cs:
    logregreg.C = c
    logregreg.fit(X, y)
    y_pred = logregreg.predict(X_test)
    losses.append(loss(y_test, y_pred))
plt.plot(cs, losses)
plt.xlabel("C")
plt.xscale("log")
plt.ylabel("Loss")
plt.axvline(x=cs[np.argmin(losses)], color="red")
plt.show()
```



Loss
160
140
120
100
80
60
40
20
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
C