Homework 1:

STATS348, UChicago, Spring 2024

Daniel F. Noriega

Open in Colab

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.

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 $\mathrm{LR} > c$, 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

In [ ]:
```
import numpy as np
import numpy.random as rn
import matplotlib.pyplot as plt
import seaborn as sns


# 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
```

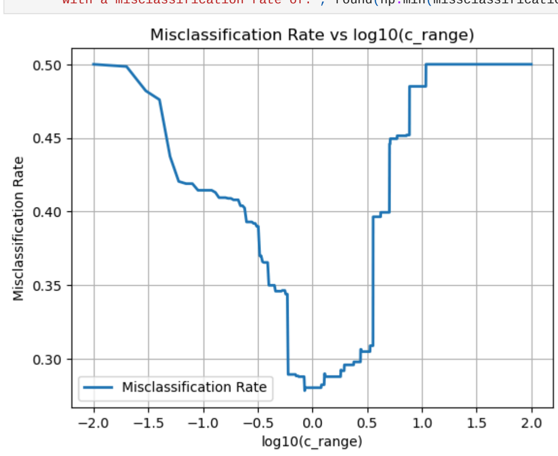
To complete the first part of this problem, you should complete the following functions.

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

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


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

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


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])
    return np.prod((p**x) * ((1-p)**(1-x)))

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])
    return np.prod((p**x) * ((1-p)**(1-x)))
```

- **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.

In [ ]:
```
n_Savannah=1000
n_Forest=1000

SimulatedData=np.concatenate((simulate_tusks_savannah(n_Savannah), simulate_tusks_forest(n_Forest)))
true_labels=np.concatenate((np.ones(n_Savannah), np.zeros(n_Forest)))
LRatios=[likelihood_savannah(i)/likelihood_forest(i) for i in SimulatedData]
c_range=np.linspace(0.01, 100, 10000)
missclassification_rate=1-([LRatios>c for c in c_range]==true_labels).mean(axis=1)

#Plotting
sns.lineplot(x=np.log10(c_range), y=missclassification_rate, linewidth=2)

# Set the labels and title
plt.xlabel('log10(c_range)')
plt.ylabel('Misclassification Rate')
plt.title('Misclassification Rate vs log10(c_range)')

# Add legend
plt.legend(['Misclassification Rate'])
plt.grid(True)

# Show the plot
plt.show()

print('The c that minimizes the misclassification rate is:', round(c_range[np.argmin(missclassification_rate)],2),
      'with a misclassification rate of:', round(np.min(missclassification_rate)*100,2), '%' )
```

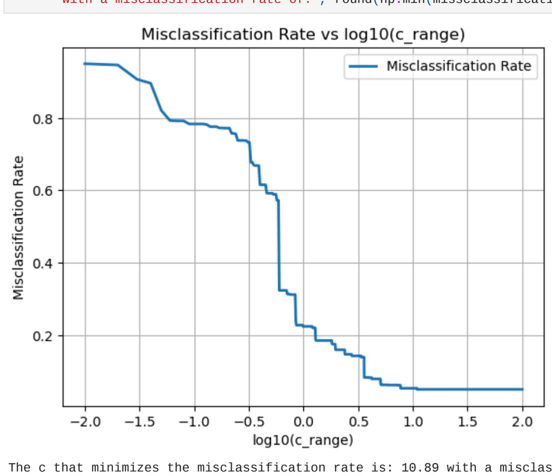



```
The c that minimizes the misclassification rate is: 0.85 with a misclassification rate of: 27.85 %
```

- **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.

In [ ]:
```
n_Savannah=100
n_Forest=1900

SimulatedData=np.concatenate((simulate_tusks_savannah(n_Savannah), simulate_tusks_forest(n_Forest)))
true_labels=np.concatenate((np.ones(n_Savannah), np.zeros(n_Forest)))
LRatios=[likelihood_savannah(i)/likelihood_forest(i) for i in SimulatedData]
c_range=np.linspace(0.01, 100, 10000)
missclassification_rate=1-([LRatios>c for c in c_range]==true_labels).mean(axis=1)

#Plotting
sns.lineplot(x=np.log10(c_range), y=missclassification_rate, linewidth=2)

# Set the labels and title
plt.xlabel('log10(c_range)')
plt.ylabel('Misclassification Rate')
plt.title('Misclassification Rate vs log10(c_range)')

# Add legend
plt.legend(['Misclassification Rate'])
plt.grid(True)

# Show the plot
plt.show()

print('The c that minimizes the misclassification rate is:', round(c_range[np.argmin(missclassification_rate)],2),
      'with a misclassification rate of:', round(np.min(missclassification_rate)*100,2), '%' )
```




```
The c that minimizes the misclassification rate is: 10.89 with a misclassification rate of: 5.0 %
```

In scenario 1a, a balanced distribution of tusks (1000 from savannah (M_S) and 1000 from forest (M_F)) led to identifying a critical/optimal 'c' value of ~1, achieving a misclassification rate of ~27%. This scenario illustrates the balanced capability of the LR test in differentiating between M_S and M_F under symmetrical conditions.

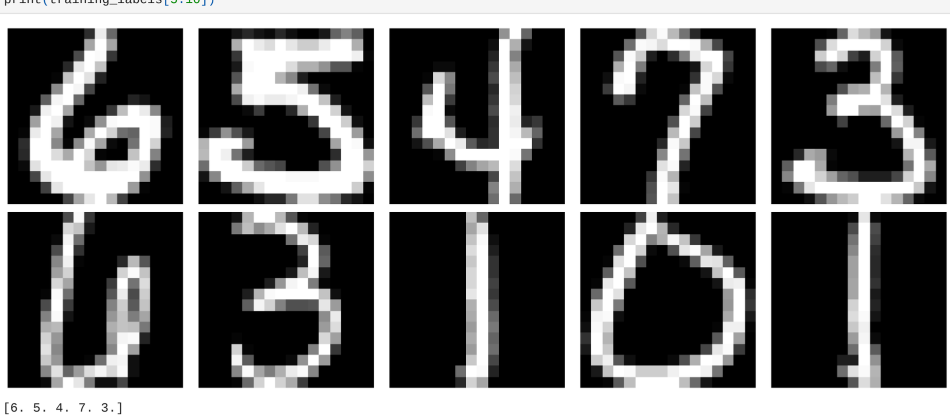
Contrastingly, scenario 1b presented a pronounced imbalance (100 from M_S and 1900 from M_F), resulting in a higher optimal 'c' value of ~10 favoring assignments to the forest class. This strategy yielded a significantly lower misclassification rate of ~5% (essentially optimizing to assigning all data to the majority class). This reflects that the lower misclassification rate predominantly stems from the model's inclination towards the majority class (M_F), suggesting that under such imbalanced conditions, the lower misclassification rate might misleadingly suggest a higher discrimination capability. Instead, in this case, it raises concerns about the model's effectiveness in distinguishing between M_S and M_F when one class overwhelmingly dominates, indicating a potential overfitting from a classification perspective towards the prevalent class rather than classification accuracy.

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.]

In [ ]:
```
import gzip

training_raw = np.loadtxt(gzip.open('../data/hw1/zip.train.gz'))
testing_raw = np.loadtxt(gzip.open('../data/hw1/zip.test.gz'))
training_labels, training_data = training_raw[:,0], training_raw[:,1:]
testing_labels, testing_data = testing_raw[:,0], testing_raw[:,1:]

fig, axes = plt.subplots(2, 5, figsize=(10, 4))
axes = axes.ravel()

for i in range(10):
    image = training_data[i].reshape(16, 16)
    axes[i].imshow(image, cmap='gray')
    axes[i].axis('off')

plt.tight_layout()
plt.show()

print(training_labels[:5])
print(training_labels[5:10])
```



```
[6. 5. 4. 7. 3.]
[6. 3. 1. 0. 1.]
```

- **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.

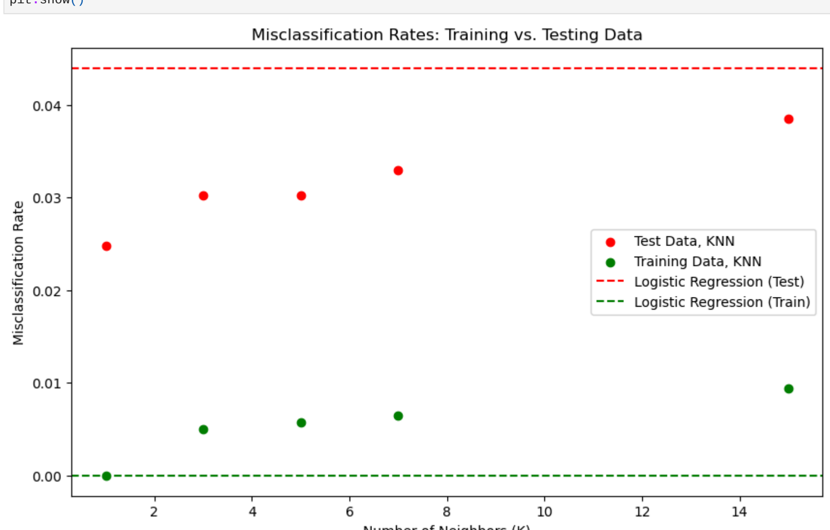
  To complete this in Python you will want to use scikit-learn, and refer to the week 1 notebook for examples.

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

# Filter data for labels 2 and 3
training_data_2_3 = training_data[np.where((training_labels == 2) | (training_labels == 3))]
training_labels_2_3 = training_labels[np.where((training_labels == 2) | (training_labels == 3))]
testing_data_2_3 = testing_data[np.where((testing_labels == 2) | (testing_labels == 3))]
testing_labels_2_3 = testing_labels[np.where((testing_labels == 2) | (testing_labels == 3))]

# Convert labels: 2 to 1 (positive class), and 3 to 0 (negative class)
training_labels_2_3 = np.where(training_labels_2_3 == 2, 1, 0)
testing_labels_2_3 = np.where(testing_labels_2_3 == 2, 1, 0)

# Dictionary to store models
models = {}

# Add Logistic Regression Model
models['logreg'] = LogisticRegression(fit_intercept=True, max_iter=10000, penalty=None)
models['logreg'].fit(training_data_2_3, training_labels_2_3)

# KNN models with varying numbers of neighbors
neighbor_settings = [1, 3, 5, 7, 15]
for n_neighbors in neighbor_settings:
    key = f'knn_{n_neighbors}'
    models[key] = KNeighborsClassifier(n_neighbors=n_neighbors)
    models[key].fit(training_data_2_3, training_labels_2_3)
```

- **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.

In [ ]:
```
# Initialize dictionaries for storing predictions and misclassification rates
predictions_train, predictions_test = {}, {}
misclassification_rates_train, misclassification_rates_test = {}, {}

# Iterate through the models to generate predictions and calculate misclassification rates
for key, model in models.items():
    # Generate predictions for training and testing data
    predictions_train[key] = model.predict(training_data_2_3)
    predictions_test[key] = model.predict(testing_data_2_3)

    # Calculate misclassification rates
    misclassification_rates_train[key] = 1 - np.mean(predictions_train[key] == training_labels_2_3)
    misclassification_rates_test[key] = 1 - np.mean(predictions_test[key] == testing_labels_2_3)

# Extract KNN keys for plotting
knn_keys = [f'knn_{n}' for n in neighbor_settings]

# Plotting
plt.figure(figsize=(10, 6))
plt.scatter(neighbor_settings, [misclassification_rates_test[k] for k in knn_keys], label='Test Data, KNN', color ='r')
plt.scatter(neighbor_settings, [misclassification_rates_train[k] for k in knn_keys], label='Training Data, KNN', color ='g')
plt.axhline(y=misclassification_rates_test['logreg'], color='r', linestyle='--', label='Logistic Regression (Test)')
plt.axhline(y=misclassification_rates_train['logreg'], color='g', linestyle='--', label='Logistic Regression (Train)')

# Labels and Legend
plt.xlabel('Number of Neighbors (K)')
plt.ylabel('Misclassification Rate')
plt.legend()
plt.title('Misclassification Rates: Training vs. Testing Data')
plt.show()
```



- **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.

By employing cross-validation to select the best K for K-NN (Best K: 5, with accuracy: 99.14%), we aim to balance model complexity with the ability to generalize. This approach helps in mitigating the risk of overfitting, which is possible particularly when models are calibrated based solely on a single training-test split. The cross-validated optimal K, which is higher than the one suggested by the lowest misclassification rate in the initial single training-test evaluation ((K=1), see plot above), suggests a model that may sacrifice some training fit in favor of generalization/predictive power.

Overall, this analysis emphasizes the critical role of parameter tuning for maximizing model accuracy and ensuring model robustness against overfitting. The cross-validation procedure proves valuable in identifying models with better generalization potential.

In [ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

# Define the range of K values to test
k_range = list(range(1, 16))

# Set up the parameter grid to search
param_grid = dict(n_neighbors=k_range)

# Configure the cross-validation procedure
cv_knn = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10, scoring='accuracy')

# Fit the model on the training data
cv_knn.fit(training_data_2_3, training_labels_2_3)

# The best K value from cross-validation
best_k = cv_knn.best_params_['n_neighbors']
best_score = cv_knn.best_score_

print(f"Best K: {best_k} with cross-validation score (accuracy): {round(best_score*100,2)}%")
```

```
Best K: 5 with cross-validation score (accuracy): 99.14%
```

- **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.

I am calculating an optimal threshold based on new cost figures, making predictions based on the already calculated probabilities, and calculating and printing loss values.

By using the theoretically-derived threshold to minimize expected loss we saw in class, we can consider the asymmetric cost structure (5 points for misclassifying a 2 as a 3 (i.e. false negative), and 1 point for misclassifying a 3 as a 2 (i.e. false positive)), the overall test set loss was reduced relative to the original threshold. This adjustment, from a default threshold of 0.5 to a calculated lower threshold (favoring positive, meaning predictions of 2s), resulted in decreasing the total loss from 52 to 43. This strategy showcases the importance of customizing the decision threshold based on specific misclassification costs.

In [ ]:
```
from sklearn.metrics import confusion_matrix

# hard-code losses and calculate threshold; recall 2 is positive.
# 5 points for false negative, 1 point for false positive
loss_TN = 0; loss_FP = 1
loss_FN = 5; loss_TP = 0

thresh = (loss_FP - loss_TN) / (loss_FP - loss_TN + loss_FN - loss_TP)

y_proba = models['logreg'].predict_proba(testing_data_2_3)[:,1]

y_pred_mod_thresh = np.where(y_proba > thresh, 1, 0)
y_pred_original = np.where(y_proba > 0.5, 1, 0)

cm_mod_thresh = confusion_matrix(testing_labels_2_3, y_pred_mod_thresh)
cm_original = confusion_matrix(testing_labels_2_3, y_pred_original)

test_pred_loss_mod_thresh = (cm_mod_thresh * np.array([[loss_TN, loss_FP],[loss_FN, loss_TP]])).sum()
test_pred_loss_original = (cm_original * np.array([[loss_TN, loss_FP],[loss_FN, loss_TP]])).sum()

print(f"Loss for modified threshold ({round(thresh,2)}): {test_pred_loss_mod_thresh}")
print(f"Loss for original threshold (0.5): {test_pred_loss_original}")
```

```
Loss for modified threshold (0.17): 43
Loss for original threshold (0.5): 52
```

- **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.

K-NN relies on majority voting instead of producing probability estimates like logistic regression. To address this, I used 'probabilities' approximated by considering the proportion of neighbors belonging to each class within a sample's neighborhood. Lowering the threshold biased the classifier towards predicting the positive class more frequently, reflecting the asymmetric misclassification costs where misclassifying a 2 as a 3 incurred a higher penalty than the reverse (similar as before).

This methodology does not work for all values of K. For example, if K=1, our kNN 'probabilities' can only take {0,1} values since they come from voting.

This underscores the importance of thoughtful consideration when adjusting models to accommodate varying loss functions.

In [ ]:
```
def calculate_loss_with_threshold(model, X, y_true, threshold, loss_matrix):
    # Use predict_proba to get the probabilities for the positive class
    y_proba = model.predict_proba(X)[:, 1]
    # Apply the custom threshold to determine class predictions
    y_pred = np.where(y_proba > threshold, 1, 0)
    # Calculate confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Calculate and return loss
    return (cm * loss_matrix).sum()

# Define the loss matrix
loss_matrix = np.array([[loss_TN, loss_FP], [loss_FN, loss_TP]])

# Calculate loss for each K-NN model using the custom threshold
losses_modified_threshold = {}
losses_original = {}
for key, model in models.items():
    if 'knn' in key:  # Ensure we're only working with K-NN models
        losses_modified_threshold[key] = calculate_loss_with_threshold(model, testing_data_2_3, testing_labels_2_3, thresh, loss_matrix)
        # Calculate loss for original classifier (i.e., using the default K=0.5 equivalent for K-NN)
        y_pred_original = model.predict(testing_data_2_3)
        cm_original = confusion_matrix(testing_labels_2_3, y_pred_original)
        losses_original[key] = (cm_original * loss_matrix).sum()

# Output losses for comparison
for key in losses_modified_threshold.keys():
    print(f"{key} - Loss with modified threshold: {losses_modified_threshold[key]}, Loss with original classifier: {losses_original[key]}")
```

```
knn_1 - Loss with modified threshold: 33, Loss with original classifier: 33
knn_3 - Loss with modified threshold: 26, Loss with original classifier: 39
knn_5 - Loss with modified threshold: 22, Loss with original classifier: 39
knn_7 - Loss with modified threshold: 30, Loss with original classifier: 48
knn_15 - Loss with modified threshold: 26, Loss with original classifier: 58
```

Problem 3: Multiclass digits

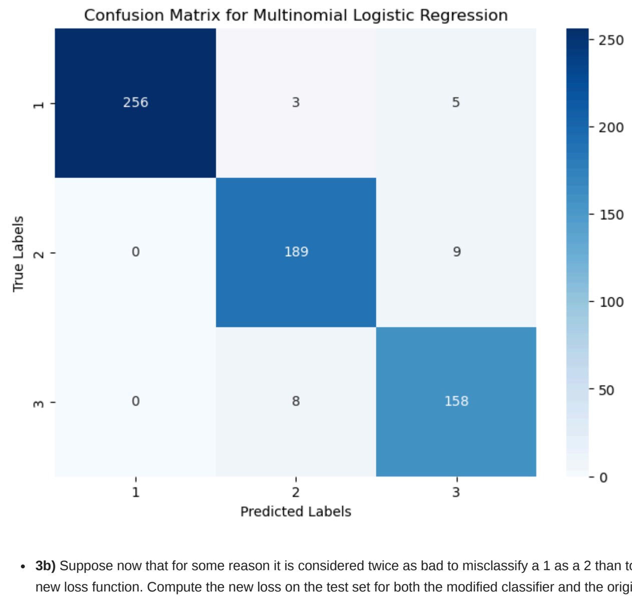
Continuing with the zipcode data, now consider distinguishing the digits 1, 2, and 3.

- 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:

In [ ]:
```
from sklearn.linear_model import LogisticRegression
# an (unregularized) multinomial logistic regression model
logreg = LogisticRegression(penalty=None, solver='liblinear', 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**.

In [ ]:
```
# Filter data for digits 1, 2, and 3
train_filter = np.isin(training_labels, [1, 2, 3])
test_filter = np.isin(testing_labels, [1, 2, 3])

training_data_123 = training_data[train_filter]
training_labels_123 = training_labels[train_filter]
testing_data_123 = testing_data[test_filter]
testing_labels_123 = testing_labels[test_filter]

# Fit a multinomial logistic regression model
model = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_iter=10000, penalty=None)
model.fit(training_data_123, training_labels_123)

# Apply the model to the test set
predictions = model.predict(testing_data_123)

# Calculate the confusion matrix
cm = confusion_matrix(testing_labels_123, predictions)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=[1, 2, 3], yticklabels=[1, 2, 3])
plt.title('Confusion Matrix for Multinomial Logistic Regression')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```



- **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.

We will be minimizing the integrated risk of a decision given by:

$$r\left(\widehat{Y}(\cdot)\right) = \mathbb{E}_{X,Y}\left[\ell\left(Y, \widehat{Y}(X)\right)\right]$$

Through Bayes risk:

$$\widehat{Y}^*(x) = \underset{a}{\operatorname{argmin}}\, \mathbb{E}_{Y|X=x}[\ell(a, Y)]$$

We can define the risk in this for each decision as the following. Recall that we want to make misclassifying a 1 as a 2 twice as bad as any other misclassification.

$$\begin{aligned} R_1(X) = \mathbb{E}_{X,Y}[\ell(Y, \hat{Y}(X))] &= \Pr(Y=1 \mid X=x) \cdot \ell_{T,1} + \Pr(Y=2 \mid X=x) \cdot \ell_{F,2} + \Pr(Y=3 \mid X=x) \cdot \ell_{F,3} \\ &= \Pr(Y=1 \mid X=x) \cdot 0 + \Pr(Y=2 \mid X=x) \cdot 2 + \Pr(Y=3 \mid X=x) \cdot 1 \\ &= \Pr(Y=2 \mid X=x) \cdot 2 + \Pr(Y=3 \mid X=x) \cdot 1 \end{aligned}$$

And

$$R_2(X) = \Pr(Y=1 \mid X=x) \cdot 1 + \Pr(Y=3 \mid X=x) \cdot 1$$

$$R_3(X) = \Pr(Y=1 \mid X=x) \cdot 1 + \Pr(Y=2 \mid X=x) \cdot 1$$

We will select the action that minimizes the conditional expected loss for any value of X = x.

The code is creating a loss matrix with adjusted cost values, calculating the expected risks and selecting the minimum for each datapoint, and calculating and printing confusion matrices and loss figures.

In [ ]:
```
#Loss factors and matrix
loss_=1
loss_12=2
loss_matrix_mod_123 = np.array([[0, loss_12, loss_], [loss_, 0, loss_], [loss_, loss_, 0]])


# Predict class probabilities for the test set
y_proba = model.predict_proba(testing_data_123)

# Initialize an array to hold the predictions based on minimizing expected loss
predictions_min_loss = np.zeros(testing_data_123.shape[0])

# Calculate expected losses for deciding each class and choose the action with the minimum expected loss
for i in range(y_proba.shape[0]):
    # Expected loss for deciding class 1
    R_1 = y_proba[i, 1] * loss_12 + y_proba[i, 2] * loss_
    # Expected loss for deciding class 2
    R_2 = y_proba[i, 0] * loss_ + y_proba[i, 2] * loss_
    # Expected loss for deciding class 3
    R_3 = y_proba[i, 0] * loss_ + y_proba[i, 1] * loss_

    # Minimize the expected loss
    R_min = np.argmin([R_1, R_2, R_3])

    # Adjust prediction based on the action that minimizes the expected loss
    predictions_min_loss[i] = R_min + 1  # Adding 1 because classes are 1, 2, 3 not 0, 1, 2

# Compare the predictions based on minimizing expected loss with the original predictions
original_predictions = model.predict(testing_data_123)

# Calculate the confusion matrix for the original model and the adjusted model
cm_original = confusion_matrix(testing_labels_123, original_predictions)
cm_adjusted = confusion_matrix(testing_labels_123, predictions_min_loss)

# Output confusion matrices
print("Confusion Matrix - Original Model:")
print(cm_original)
print("\nConfusion Matrix - Adjusted Model:")
print(cm_adjusted)

# Calculate the loss for the original and adjusted models
loss_original = (cm_original * loss_matrix_mod_123).sum()
loss_adjusted = (cm_adjusted * loss_matrix_mod_123).sum()
print(f"\nLoss - Original Model: {loss_original}")
print(f"Loss - Adjusted Model: {loss_adjusted}")
```

```
Confusion Matrix - Original Model:
[[256   3   5]
 [  0 189   9]
 [  0   8 158]]

Confusion Matrix - Adjusted Model:
[[256   3   5]
 [  0 189   9]
 [  0   8 158]]

Loss - Original Model: 28
Loss - Adjusted Model: 28
```

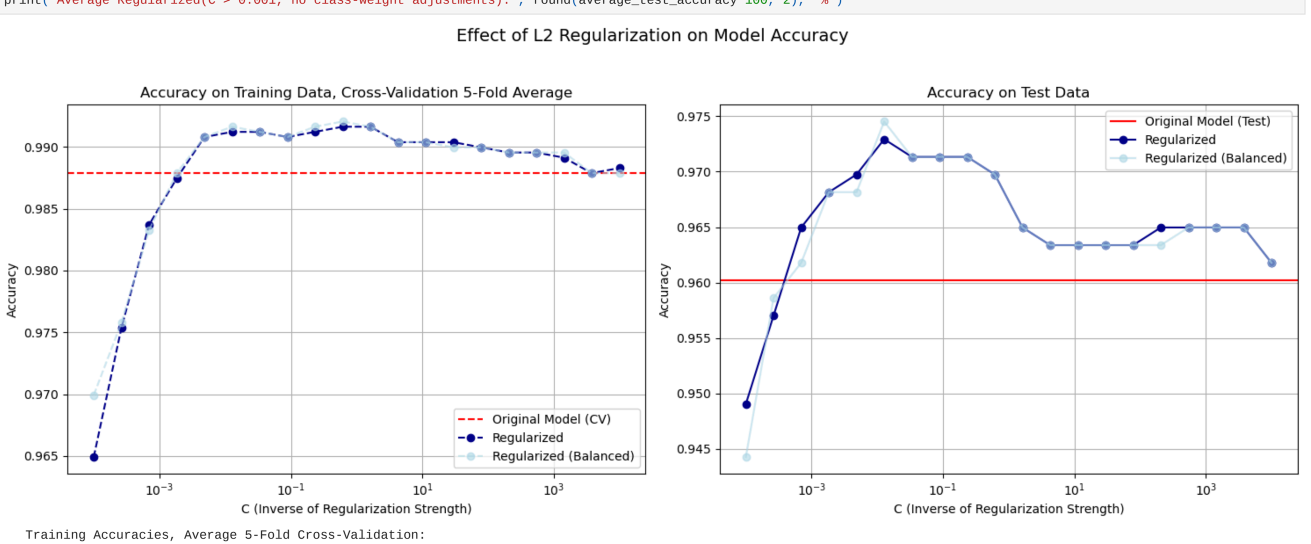
- **3c)** Now consider adding ℓ_2 regularization to your multinomial logistic regression classifier and answer the question: does it improve performance? Devise some experiments to answer this question convincingly. Use the code block below to implement and run those experiments, and to generate plot(s) 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.

I used the accuracy of the original model (no regularization, same misclassification cost) on both the training and testing set as a baseline. I then assessed multiple levels of ℓ_2 regularization strength by employing cross-validation on the training data and test data to evaluate model performance. Furthermore, I examined models with' balanced' class weights alongside their unweighted counterparts to investigate the impact of class imbalance, particularly in scenarios where certain classes might be underrepresented (as studied in exercise 2). This approach allowed for a thorough evaluation of the effectiveness of regularization in improving model performance across different datasets and conditions.

As seen below, regularized models generally perform better than no-regularization models (lines representing regularized models generally lie above those of the original model, reflecting better performance). This is the case for most regularization strengths as captured by the C parameter. Higher C values indicate weaker regularization. However, when regularization is too high (i.e., C values too low), the original model outperforms the regularized models. This phenomenon underscores the delicate balance required in selecting the appropriate level of regularization, as overly constrained models may hinder models' flexibility.

I tested a wide range of regularization strengths (0.0001 to 10000), and regularized models outperformed the original model for C values above 0.001 (virtually all of them). This means that regularized models are likely preferred for this dataset for sensible regularization strengths (not excessively constrained). Over cross-validated training data for C > 0.001 (and no class-weight adjustments), regularized accuracies averaged 99.01%, better than the original models' 98.79% performance. Over test data and similar conditions, regularized accuracies averaged 96.67%, better than the original 96.02%. The regularized approach may outperform non-regularized alternatives in this setting because regularization helps to prevent overfitting by constraining model complexity, thus improving the model's ability to generalize to unseen data. In a 256-dimensional space, regularization may enhance overfitting mitigation, thus substantiating its adoption in our multinomial logistic regression classifier for improved performance.

In [ ]:
```
from sklearn.model_selection import cross_val_score

# Calculate the accuracy of the original model with cross-validation
cv_scores_original = cross_val_score(model, training_data_123, training_labels_123, cv=5, scoring='accuracy')
accuracy_original_train = np.mean(cv_scores_original)

# Calculate the accuracy of the original model on the test set
accuracy_original_test = model.score(testing_data_123, testing_labels_123)

# Define a range of C values to explore
C_values = np.logspace(-4, 4, 20)

# Prepare lists to store results
train_scores = []
train_scores_balanced = []
test_scores = []
test_scores_balanced = []

# Loop over the values of C to train models and record performance
for C in C_values:
    t_model = LogisticRegression(multi_class='multinomial', solver='lbfgs', C=C, max_iter=10000, penalty='l2')
    scores = cross_val_score(t_model, training_data_123, training_labels_123, cv=5, scoring='accuracy')
    train_scores.append(np.mean(scores))
    t_model_balanced = LogisticRegression(multi_class='multinomial', solver='lbfgs', C=C, max_iter=10000, penalty='l2', class_weight='balanced')
    scores_balanced = cross_val_score(t_model_balanced, training_data_123, training_labels_123, cv=5, scoring='accuracy')
    train_scores_balanced.append(np.mean(scores_balanced))

    # Train models on the entire training set and evaluate on the test set
    t_model.fit(training_data_123, training_labels_123)
    test_score = t_model.score(testing_data_123, testing_labels_123)
    test_scores.append(test_score)

    t_model_balanced.fit(training_data_123, training_labels_123)
    test_score_balanced = t_model_balanced.score(testing_data_123, testing_labels_123)
    test_scores_balanced.append(test_score_balanced)

# Plotting the results
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Plot for training accuracy
axes[0].axhline(y=accuracy_original_train, color='r', linestyle='--', label='Original Model (CV)')
axes[0].plot(C_values, train_scores, label='Regularized', marker='o', color='darkblue', linestyle='--')
axes[0].plot(C_values, train_scores_balanced, label='Regularized (Balanced)', marker='o', color='lightblue', alpha=0.6, linestyle='--')
axes[0].set_xscale('log')
axes[0].set_xlabel('C (Inverse of Regularization Strength)')
axes[0].set_ylabel('Accuracy')
axes[0].set_title('Accuracy on Training Data, Cross-Validation 5-Fold Average')
axes[0].legend()
axes[0].grid(True)

# Plot for testing accuracy
axes[1].axhline(y=accuracy_original_test, color='r', linestyle='-', label='Original Model (Test)')
axes[1].plot(C_values, test_scores, label='Regularized', marker='o', color='darkblue', linestyle='-')
axes[1].plot(C_values, test_scores_balanced, label='Regularized (Balanced)', marker='o', color='lightblue', alpha=0.6, linestyle='-')
axes[1].set_xscale('log')
axes[1].set_xlabel('C (Inverse of Regularization Strength)')
axes[1].set_ylabel('Accuracy')
axes[1].set_title('Accuracy on Test Data')
axes[1].legend()
axes[1].grid(True)

# Overall title
plt.suptitle('Effect of L2 Regularization on Model Accuracy', fontsize=14)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust subplot spacing
plt.show()


# Filter C values greater than 0.001
filtered_indices = [i for i, C in enumerate(C_values) if C > 0.001]

# Calculate average accuracies for training and test data
average_train_accuracy = np.mean(np.array(train_scores)[filtered_indices])
average_test_accuracy = np.mean(np.array(test_scores)[filtered_indices])

# Output results
print('   Training Accuracies, Average 5-Fold Cross-Validation:')
print("Original:", round(accuracy_original_train*100, 2), '%')
print("Average Regularized(C > 0.001, no class-weight adjustments):", round(average_train_accuracy*100, 2), '%')
print('')
print('   Test Accuracies, Average 5-Fold Cross-Validation:')
print("Original Test Accuracy:", round(accuracy_original_test*100, 2), '%')
print("Average Regularized(C > 0.001, no class-weight adjustments):", round(average_test_accuracy*100, 2), '%')
```




```
   Training Accuracies, Average 5-Fold Cross-Validation:
Original: 98.79 %
Average Regularized(C > 0.001, no class-weight adjustments): 99.01 %

   Test Accuracies, Average 5-Fold Cross-Validation:
Original Test Accuracy: 96.02 %
Average Regularized(C > 0.001, no class-weight adjustments): 96.67 %
```