DATA 221 Homework 4

Chris Low

Problem 1

```python
from sklearn.datasets import load_breast_cancer
bc = load_breast_cancer()
X = bc.data
y = bc.target
```

1(a)

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

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)
```

1(b)

Standardization is unnecessary for decision trees because splits depend only on the ordering of feature values, such as whether a feature exceeds a threshold. Scaling a feature is a monotonic transformation and does not change this ordering, so the resulting tree structure is unchanged.

In contrast, **logistic regression is trained using gradient-based optimization.** When features are on very different scales, the loss surface becomes poorly conditioned, which leads to unstable updates and slow convergence. Standardizing the predictors puts them on comparable scales, so we get more stable gradients and faster convergence.

1(c)

The standardization parameters, such as the mean and standard deviation, should be computed using only the training data to **avoid data leakage.** The test set is meant to represent unseen future data. If its information is used when scaling the predictors, the model indirectly gains access to the test distribution, which can lead to overly optimistic performance estimates. Doing this keeps the test set truly out of sample and matches how the model would be applied to new data in practice.

1(d)

Let

$$p_i \;=\; P(y_i = 1 \mid X_i) \;=\; \sigma(\eta_i), \qquad \eta_i = w^\top X_i, \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

Since $y_i \in \{0,1\}$, the Bernoulli likelihood for observation i is

$$P(y_i \mid X_i) = p_i^{y_i}(1-p_i)^{1-y_i}.$$

Assuming independent observations, the log-likelihood for $\{(X_i, y_i)\}_{i=1}^n$ is

$$\ell(w) = \sum_{i=1}^n \log P(y_i \mid X_i) = \boxed{\sum_{i=1}^n \Big[y_i \log(p_i) + (1-y_i)\log(1-p_i)\Big]}.$$

where $p_i = \sigma(w^\top X_i)$.

1(e)

The binary cross-entropy loss (average form) is

$$J(w) = -\frac{1}{n}\sum_{i=1}^n \Big[y_i \log(p_i) + (1-y_i)\log(1-p_i)\Big].$$

Comparing with the log-likelihood in part (d),

$$J(w) = -\frac{1}{n}\,\ell(w).$$

Multiplying by $-1/n$ only rescales and flips the optimization direction, so the optimizer is the same.

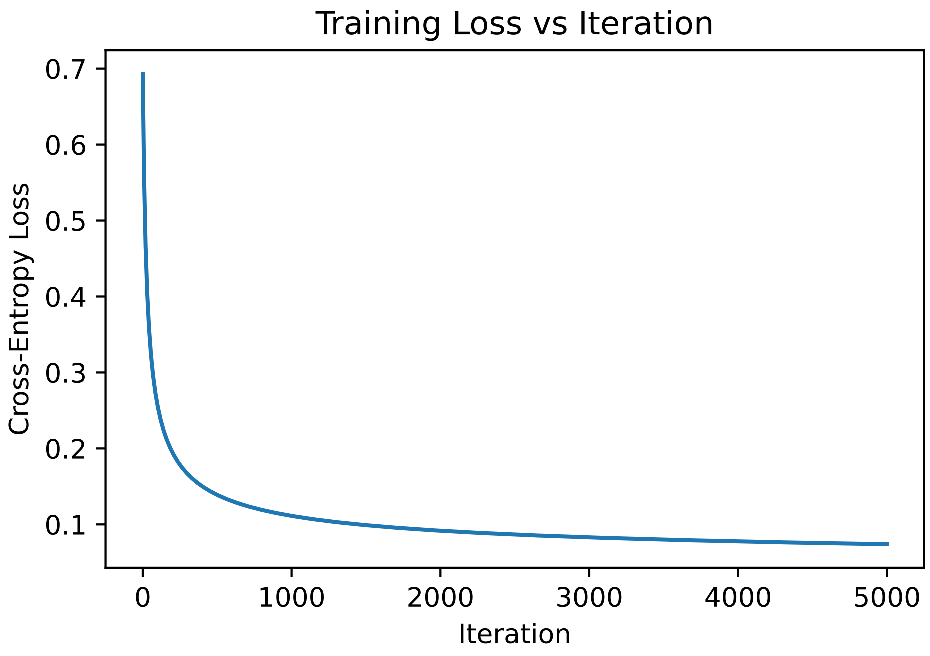
So, we can conclude that minimizing cross-entropy is equivalent to maximizing the log-likelihood.

1(f)

```python
import numpy as np

def sigmoid(z):
    z = np.clip(z, -500, 500)
    return 1.0 / (1.0 + np.exp(-z))

def cross_entropy(y, p):
    eps = 1e-12
    p = np.clip(p, eps, 1 - eps)
    return -np.mean(y * np.log(p) + (1 - y) * np.log(1 - p))

def fit_logistic_regression(
    X, y,
    lr=0.01,
    max_iter=1000,
    tol=1e-4
):
    # Initialize coefficients
    n, d = X.shape
    w = np.zeros(d)
    b = 0.0

    losses = []

    for t in range(max_iter):
        scores = X @ w + b
        probs = sigmoid(scores)

        # Compute cross-entropy loss
        loss = cross_entropy(y, probs)
        losses.append(loss)

        # Compute gradients
        grad_w = (1 / n) * X.T @ (probs - y)
        grad_b = (1 / n) * np.sum(probs - y)

        # Update coefficients
        w -= lr * grad_w
        b -= lr * grad_b

        # Reasonable stopping criterion
        if t > 0 and abs(losses[-2] - losses[-1]) < tol:
            break

    return w, b, losses
```

1(g)

```
import matplotlib.pyplot as plt

w_hat, b_hat, losses = fit_logistic_regression(
    X_train_std, y_train,
    lr=0.01,
    max_iter=5000,
    tol=1e-6
)

plt.figure()
plt.plot(losses)
plt.xlabel("Iteration")
plt.ylabel("Cross-Entropy Loss")
plt.title("Training Loss vs Iteration")
plt.show()
```



The loss drops quickly at the beginning and then flattens out, which suggests that gradient descent is converging. After a certain point, additional iterations do not noticeably improve the loss.

1(h)

Due to using L2 regularization by default, we need to set a large value of C to use it without regularization.

```python
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(
    C=1e6,
    solver="lbfgs",
    max_iter=5000
)

clf.fit(X_train_std, y_train)


w_sklearn = clf.coef_.flatten()
b_sklearn = clf.intercept_[0]
```

```
probs_sklearn = clf.predict_proba(X_train_std)[:, 1]
sklearn_loss = cross_entropy(y_train, probs_sklearn)

print("Final GD loss:", losses[-1])
print("Final sklearn loss:", sklearn_loss)

print("First 5 GD coefficients:", w_hat[:5])
print("First 5 sklearn coefficients:", w_sklearn[:5])
print("GD intercept:", b_hat)
print("Sklearn intercept:", b_sklearn)
```

```
Final GD loss: 0.0739236175496964
Final sklearn loss: 0.0002954851332951214
First 5 GD coefficients: [-0.48241728 -0.55337154 -0.46661102 -0.50266762 -
0.23216018]
First 5 sklearn coefficients: [ 22.91087147  -6.13887295  45.08367648   7.97471529 -
23.28746162]
GD intercept: 0.5012271610638136
Sklearn intercept: -36.98822516136943
```

Run some predictions to compare:

```
pred_gd = (sigmoid(X_train_std @ w_hat + b_hat) >= 0.5).astype(int)
pred_sklearn = clf.predict(X_train_std)

print("Train agreement:", np.mean(pred_gd == pred_sklearn))
print("Train acc GD:", np.mean(pred_gd == y_train))
print("Train acc sklearn:", np.mean(pred_sklearn == y_train))
```

```
Train agreement: 0.9849246231155779
Train acc GD: 0.9849246231155779
Train acc sklearn: 1.0
```

Both models are minimizing the same cross-entropy loss, but they end up behaving differently during optimization. The breast cancer data are close to linearly separable, so when using sklearn's logistic regression with very weak regularization, the optimizer can keep increasing the size of the coefficients to push the training loss closer and closer to zero.

In my gradient descent implementation, I stop updating once the improvement in loss becomes very small, using the stopping rule `abs(losses[-2] - losses[-1]) < tol`. Because of this, the gradient descent solution settles at a higher loss with smaller, finite coefficients, while the sklearn model continues toward much larger coefficient values. Even so, the two models make very similar predictions and agree on about 98.5% of the training data.

1(i)

Assume the coefficients are independent and satisfy the prior

$$w_j \sim \mathcal{N}(0, \tau^2).$$

The joint prior over $w = (w_1, \dots, w_l)$ is

$$p(w) = \prod_{j=1}^{l} \frac{1}{\sqrt{2\pi\tau^2}} \exp\left(-\frac{w_j^2}{2\tau^2}\right).$$

Taking logs gives the log-prior:

$$\log p(w) = \sum_{j=1}^{l} \left[-\frac{1}{2}\log(2\pi\tau^2) - \frac{w_j^2}{2\tau^2}\right] = -\frac{l}{2}\log(2\pi\tau^2) - \frac{1}{2\tau^2}\sum_{j=1}^{l} w_j^2.$$

Up to an additive constant:

$$\boxed{\log p(w) = -\frac{1}{2\tau^2}\|w\|_2^2 + C}$$

1(j)

The posterior distribution satisfies

$$p(w \mid X, y) \propto p(y \mid X, w)\, p(w).$$

Taking logs,

$$\log p(w \mid X, y) = \log p(y \mid X, w) + \log p(w) + C.$$

Maximizing the posterior equals minimizing the negative log-posterior:

$$-\log p(w \mid X, y) = -\log p(y \mid X, w) - \log p(w) + C.$$

For logistic regression, $-\log p(y \mid X, w)$ is the cross-entropy loss (up to averaging). Using the result from part (i),

$$-\log p(w) = \frac{1}{2\tau^2}\|w\|_2^2 + C.$$

Therefore, the MAP estimation is:

$$\min_{w} \; \text{CrossEntropy}(w) + \frac{1}{2\tau^2}\|w\|_2^2.$$

The second term is exactly an **L2 regularization (ridge) penalty**, with regularization strength that is inversely proportional to τ^2.

1(k)

The prior controls how large the model coefficients are allowed to be. A strong prior (small τ^2) implies a strong penalty, shrinking the coefficients toward zero and making the model simpler and less sensitive to noise. This **reduces variance but can increase bias** if the model becomes too constrained. A weak prior (large τ^2) allows larger coefficients and a more flexible model, which can **reduce bias but increase variance and the risk of overfitting**. Thus, the prior variance determines the bias–variance tradeoff.

Problem 2

2(a)

```python
import numpy as np

def entropy(y):
    """
    Computes entropy for a binary response vector y.
    """
    n = len(y)
    if n == 0:
        return 0.0

    p1 = np.mean(y)
    p0 = 1 - p1

    eps = 1e-12
    ent = 0.0
    if p1 > 0:
        ent -= p1 * np.log2(p1 + eps)
    if p0 > 0:
        ent -= p0 * np.log2(p0 + eps)

    return ent

def gini(y):
    """
    Computes Gini impurity for a binary response vector y.
    """
    p1 = np.mean(y)
    p0 = 1 - p1
    return 1 - (p1**2 + p0**2)

def information_gain(y_parent, y_left, y_right, criterion="entropy"):
    """
    Computes information gain from splitting y_parent
    into y_left and y_right.

    criterion: "entropy" or "gini"
    """
    n = len(y_parent)
    n_left = len(y_left)
    n_right = len(y_right)

    if criterion == "entropy":
        impurity = entropy
    elif criterion == "gini":
        impurity = gini
    else:
        raise ValueError("criterion must be 'entropy' or 'gini'")

    parent_impurity = impurity(y_parent)
    left_impurity = impurity(y_left)
    right_impurity = impurity(y_right)

    weighted_child_impurity = (
        (n_left / n) * left_impurity +
        (n_right / n) * right_impurity
    )

    return parent_impurity - weighted_child_impurity
```

2(b)

Choose two candidate split features with corresponding split values:

```
# I will reload data just in case

import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

bc = load_breast_cancer()
X = bc.data
y = bc.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

def split_labels(X, y, feature, threshold):
    left_mask = X[:, feature] <= threshold
    right_mask = X[:, feature] > threshold
    return y[left_mask], y[right_mask]

feature_1 = 0
threshold_1 = np.median(X_train[:, feature_1])

feature_2 = 7
threshold_2 = np.median(X_train[:, feature_2])

y_left_1, y_right_1 = split_labels(X_train, y_train, feature_1, threshold_1)
y_left_2, y_right_2 = split_labels(X_train, y_train, feature_2, threshold_2)

ig_entropy_1 = information_gain(y_train, y_left_1, y_right_1, "entropy")
ig_gini_1 = information_gain(y_train, y_left_1, y_right_1, "gini")

ig_entropy_2 = information_gain(y_train, y_left_2, y_right_2, "entropy")
ig_gini_2 = information_gain(y_train, y_left_2, y_right_2, "gini")

print("Root split results:")
print("Feature 0 | Entropy IG:", ig_entropy_1, "Gini IG:", ig_gini_1)
print("Feature 7 | Entropy IG:", ig_entropy_2, "Gini IG:", ig_gini_2)
```

```
Root split results:
Feature 0 | Entropy IG: 0.3136826657695946 Gini IG: 0.18230648995048981
Feature 7 | Entropy IG: 0.38221080091149107 Gini IG: 0.21554818449522173
```

```
best_feature = 7
best_threshold = threshold_2

left_mask = X_train[:, best_feature] <= best_threshold
X_left = X_train[left_mask]
y_left = y_train[left_mask]
```

```
# Candidate splits at depth 2
feature_3 = 2
threshold_3 = np.median(X_left[:, feature_3])

feature_4 = 12
threshold_4 = np.median(X_left[:, feature_4])
```

```python
y_ll_3, y_lr_3 = split_labels(X_left, y_left, feature_3, threshold_3)
y_ll_4, y_lr_4 = split_labels(X_left, y_left, feature_4, threshold_4)

ig_entropy_3 = information_gain(y_left, y_ll_3, y_lr_3, "entropy")
ig_gini_3 = information_gain(y_left, y_ll_3, y_lr_3, "gini")

ig_entropy_4 = information_gain(y_left, y_ll_4, y_lr_4, "entropy")
ig_gini_4 = information_gain(y_left, y_ll_4, y_lr_4, "gini")

print("\nDepth-2 split results (left child):")
print("Feature 2 | Entropy IG:", ig_entropy_3, "Gini IG:", ig_gini_3)
print("Feature 12 | Entropy IG:", ig_entropy_4, "Gini IG:", ig_gini_4)
```

```
Depth-2 split results (left child):
Feature 2 | Entropy IG: 0.045311445175086645 Gini IG: 0.003847337642351456
Feature 12 | Entropy IG: 0.0013321490151339543 Gini IG: 0.00015389350569394722
```

Two candidate splits were evaluated at the root using **features 0 and 7**, with thresholds set to the **median** of each feature. **Feature 7 produced higher information gain under both entropy and Gini impurity, so it was selected as the root split.**

The data were then restricted to the left child node, and two additional candidate splits using features 2 and 12 were evaluated. At this level, feature 2 resulted in the larger information gain and was therefore chosen as the second split. This process shows a greedy splitting strategy, where the best local split is selected at each depth.

2(c)

The greedy splitting procedure used in decision trees makes a sequence of **locally optimal decisions**. At each node, the algorithm chooses the split that maximizes information gain at that step, without considering how this choice might affect future splits. Once a split is made, it is never revisited.

In contrast, gradient-based methods such as logistic regression **optimize a single global objective function**. The model parameters are updated gradually using gradient information, and earlier decisions can be adjusted as the optimization continues.

This allows gradient-based methods to trade off different parts of the model jointly, whereas greedy tree construction focuses only on immediate improvements. In other words, trees choose questions one at a time, while gradient methods slowly tune all parameters together.

Problem 3

3(a)

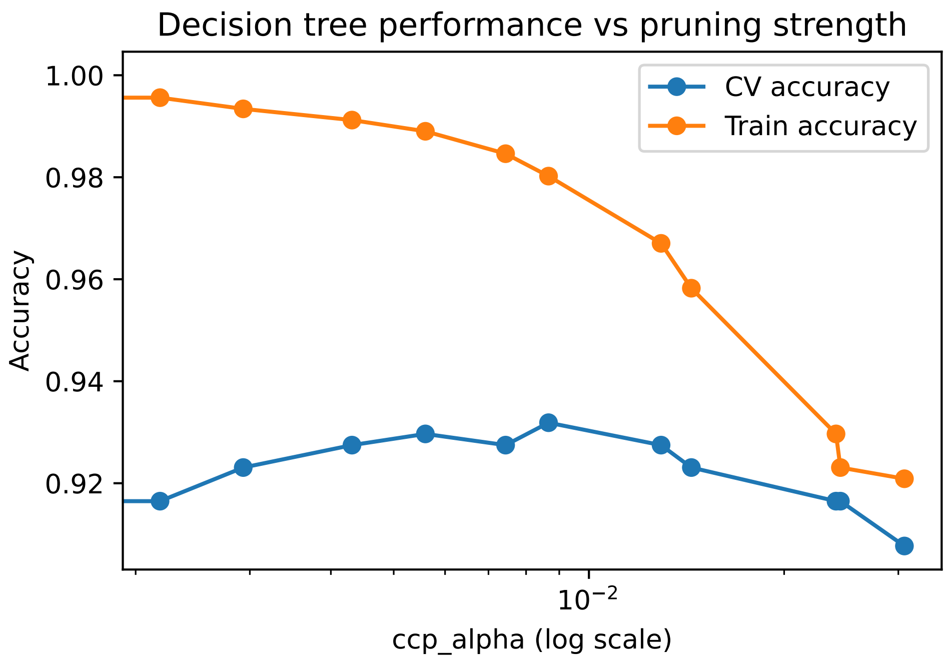
```python
import time
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

dt = DecisionTreeClassifier(random_state=42)

param_grid = {
    "max_depth": [3, 5, 7, None],          # 4 values
    "min_samples_split": [2, 5, 10, 20],   # 4 values
    "min_samples_leaf": [1, 2, 5, 10]      # 4 values
}

# Total parameter combinations = 4 * 4 * 4 = 64

# GridSearchCV
start_time = time.time()

grid_search = GridSearchCV(
    estimator=dt,
    param_grid=param_grid,
    cv=5,
    scoring="accuracy",
    n_jobs=-1
)

grid_search.fit(X_train, y_train)
grid_time = time.time() - start_time

# RandomizedSearchCV
# n_iter = 64 so total fits match GridSearchCV
start_time = time.time()

random_search = RandomizedSearchCV(
    estimator=dt,
    param_distributions=param_grid,
    n_iter=64,
    cv=5,
    scoring="accuracy",
    random_state=42,
    n_jobs=-1
)

random_search.fit(X_train, y_train)
random_time = time.time() - start_time

print("Grid Search Results")
print("-------------------")
print(f"Best CV Accuracy: {grid_search.best_score_:.4f}")
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Runtime: {grid_time:.4f} seconds\n")

print("Randomized Search Results")
print("-------------------------")
print(f"Best CV Accuracy: {random_search.best_score_:.4f}")
print(f"Best Parameters: {random_search.best_params_}")
print(f"Runtime: {random_time:.4f} seconds")
```

```
Grid Search Results
-------------------
Best CV Accuracy: 0.9341
Best Parameters: {'max_depth': 3, 'min_samples_leaf': 5, 'min_samples_split': 2}
Runtime: 1.7728 seconds

Randomized Search Results
-------------------------
Best CV Accuracy: 0.9341
Best Parameters: {'min_samples_split': 2, 'min_samples_leaf': 5, 'max_depth': 3}
Runtime: 0.1659 seconds
```

Both methods found the exact same model and achieved the same CV accuracy (0.9341), which means randomized search did not miss anything important in this case. The big difference is runtime. Grid search took much longer because it exhaustively evaluated every combination in the grid, while randomized search happened to sample the optimal combination early and did not need to check everything else. With a small search space, both methods often end up with the same answer, but randomized search can still be faster because it avoids unnecessary evaluations.

3(b)

```python
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix


base_tree = DecisionTreeClassifier(random_state=42)
base_tree.fit(X_train, y_train)

path = base_tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path.ccp_alphas
impurities = path.impurities


# Drop last alpha (root has highest alpha))
ccp_alphas = ccp_alphas[:-1]
impurities = impurities[:-1]

cv_means = []
train_accs = []

for a in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=42, ccp_alpha=a)
    scores = cross_val_score(clf, X_train, y_train, cv=5, scoring="accuracy")
    cv_means.append(scores.mean())

    clf.fit(X_train, y_train)
    train_pred = clf.predict(X_train)
    train_accs.append(accuracy_score(y_train, train_pred))

cv_means = np.array(cv_means)
train_accs = np.array(train_accs)

plt.figure()
plt.plot(ccp_alphas, cv_means, marker="o", label="CV accuracy")
plt.plot(ccp_alphas, train_accs, marker="o", label="Train accuracy")
plt.xscale("log")
plt.xlabel("ccp_alpha (log scale)")
plt.ylabel("Accuracy")
plt.title("Decision tree performance vs pruning strength")
plt.legend()
plt.show()
```



3(c)

We select the value of α that maximizes cross-validated accuracy because it provides the best estimate of of how well that level of pruning will perform on new, unseen data, without using the test set.

```
best_idx = int(np.argmax(cv_means))
best_alpha = float(ccp_alphas[best_idx])

pruned_tree = DecisionTreeClassifier(random_state=42, ccp_alpha=best_alpha)
pruned_tree.fit(X_train, y_train)

test_pred = pruned_tree.predict(X_test)
test_acc = accuracy_score(y_test, test_pred)

print("Chosen alpha:", best_alpha)
print("CV accuracy at chosen alpha:", cv_means[best_idx])
print("Test accuracy:", test_acc)
print("\nConfusion matrix:\n", confusion_matrix(y_test, test_pred))
print("\nClassification report:\n", classification_report(y_test, test_pred))
```

```
Chosen alpha: 0.008663799968147794
CV accuracy at chosen alpha: 0.9318681318681319
Test accuracy: 0.956140350877193

Confusion matrix:
 [[40  3]
 [ 2 69]]

Classification report:
               precision    recall  f1-score   support

           0       0.95      0.93      0.94        43
           1       0.96      0.97      0.97        71

    accuracy                           0.96       114
   macro avg       0.96      0.95      0.95       114
weighted avg       0.96      0.96      0.96       114
```

Using cost-complexity pruning, I selected $\alpha = 0.0087$ based on cross-validated accuracy. The resulting pruned tree achieved a test accuracy of 95.6%, with high precision and recall for both classes.

3(d)

Cost-complexity pruning and regularization both control overfitting by **adding an explicit penalty for model complexity.** In decision trees, pruning penalizes the number of terminal nodes, so increasing α removes branches and produces a smaller, simpler tree.

In logistic regression, regularization penalizes the size of the coefficients (such as using L1 or L2 regularization), shrinking them toward zero and limiting how strongly the model can respond to individual features. In both cases, a stronger penalty trades some training accuracy for reduced variance, leading to better generalization.