linear_regression

January 28, 2024

1 ECE 176 Assignment 2: Linear Regression

For this part of assignment, you are tasked to implement a linear regression algorithm for multiclass classification and test it on the CIFAR10 dataset.

You sould run the whole notebook and answer the questions in the notebook.

CIFAR 10 dataset contains 32x32x3 RGB images of 10 distinct cateogaries, and our aim is to predict which class the image belongs to

TO SUBMIT: PDF of this notebook with all the required outputs and answers.

[1]:
```
!python get_datasets.py
```

```
Downloading cifar-10
Done
```

[2]:
```python
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from utils.data_processing import get_cifar10_data

# Use a subset of CIFAR10 for the assignment
dataset = get_cifar10_data(
    subset_train=5000,
    subset_val=250,
    subset_test=500,
)

print(dataset.keys())
print("Training Set Data  Shape: ", dataset["x_train"].shape)
print("Training Set Label Shape: ", dataset["y_train"].shape)
print("Validation Set Data  Shape: ", dataset["x_val"].shape)
print("Validation Set Label Shape: ", dataset["y_val"].shape)
print("Test Set Data  Shape: ", dataset["x_test"].shape)
print("Test Set Label Shape: ", dataset["y_test"].shape)
```

```
dict_keys(['x_train', 'y_train', 'x_val', 'y_val', 'x_test', 'y_test'])
Training Set Data  Shape:  (5000, 3072)
Training Set Label Shape:  (5000,)
Validation Set Data  Shape:  (250, 3072)
Validation Set Label Shape:  (250,)
Test Set Data  Shape:  (500, 3072)
Test Set Label Shape:  (500,)
```

[3]:
```python
x_train = dataset["x_train"]
y_train = dataset["y_train"]
x_val = dataset["x_val"]
y_val = dataset["y_val"]
x_test = dataset["x_test"]
y_test = dataset["y_test"]
```

[4]:
```python
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = [
    "plane",
    "car",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
    "truck",
]
samples_per_class = 7


def visualize_data(dataset, classes, samples_per_class):
    num_classes = len(classes)
    for y, cls in enumerate(classes):
        idxs = np.flatnonzero(y_train == y)
        idxs = np.random.choice(idxs, samples_per_class, replace=False)
        for i, idx in enumerate(idxs):
            plt_idx = i * num_classes + y + 1
            plt.subplot(samples_per_class, num_classes, plt_idx)
            plt.imshow(dataset[idx])
            plt.axis("off")
            if i == 0:
                plt.title(cls)
    plt.show()


visualize_data(
    x_train.reshape(5000, 3, 32, 32).transpose(0, 2, 3, 1), classes,␣
  ↪samples_per_class
)
```

2 Linear Regression for multi-class classification

A Linear Regression Algorithm has these hyperparameters:

- **Learning rate** - controls how much we change the current weights of the classifier during each update. We set it at a default value of 0.5, and later you are asked to experiment with different values. We recommend looking at the graphs and observing how the performance of the classifier changes with different learning rate.
- **Number of Epochs** - An epoch is a complete iterative pass over all of the data in the dataset. During an epoch we predict a label using the classifier and then update the weights of the classifier according the linear classifier update rule for each sample in the training set. We evaluate our models after every 10 epochs and save the accuracies, which are later used to plot the training, validation and test VS epoch curves.
- **Weight Decay** - Regularization can be used to constrain the weights of the classifier and prevent their values from blowing up. Regularization helps in combatting overfitting. You will be using the ‘weight_decay’ term to introduce regularization in the classifier.

2.0.1 Implementation (50%)

You first need to implement the Linear Regression method in `algorithms/linear_regression.py`. The formulations follow the lecture (consider binary classification for each of the 10 classes, with labels -1 / 1 for not belonging / belonging to the class). You need to fill in the training function as well as the prediction function.

[5]:
```
# Import the algorithm implementation (TODO: Complete the Linear Regression in␣
 ↪algorithms/linear_regression.py)
from algorithms import Linear
from utils.evaluation import get_classification_accuracy

num_classes = 10  # Cifar10 dataset has 10 different classes

# Initialize hyper-parameters
learning_rate = 0.0001  # You will be later asked to experiment with different␣
 ↪learning rates and report results
num_epochs_total = 200  # Total number of epochs to train the classifier
epochs_per_evaluation = 10  # Epochs per step of evaluation; We will evaluate␣
 ↪our model regularly during training
N, D = dataset[
    "x_train"
].shape  # Get training data shape, N: Number of examples, D:Dimensionality of␣
 ↪the data
weight_decay = 0.0

# Insert additional scalar term 1 in the samples to account for the bias as␣
 ↪discussed in class
x_train = np.insert(x_train, D, values=1, axis=1)
x_val = np.insert(x_val, D, values=1, axis=1)
x_test = np.insert(x_test, D, values=1, axis=1)
```

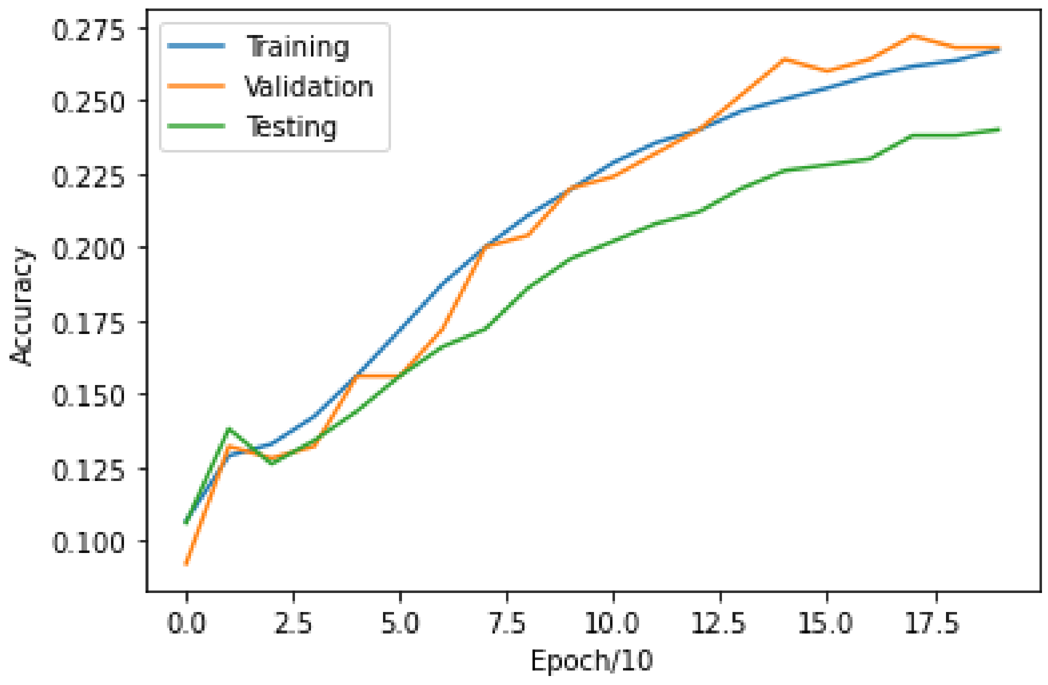
[6]:
```
# Training and evaluation function -> Outputs accuracy data
def train(learning_rate_, weight_decay_):
    # Create a linear regression object
    linear_regression = Linear(
        num_classes, learning_rate_, epochs_per_evaluation, weight_decay_
    )

    # Randomly initialize the weights and biases
    weights = np.random.randn(num_classes, D + 1) * 0.0001

    train_accuracies, val_accuracies, test_accuracies = [], [], []

    # Train the classifier
    for _ in range(int(num_epochs_total / epochs_per_evaluation)):
        # Train the classifier on the training data
        weights = linear_regression.train(x_train, y_train, weights)

        # Evaluate the trained classifier on the training dataset
        y_pred_train = linear_regression.predict(x_train)
        train_accuracies.append(get_classification_accuracy(y_pred_train,␣
↪y_train))

        # Evaluate the trained classifier on the validation dataset
        y_pred_val = linear_regression.predict(x_val)
        val_accuracies.append(get_classification_accuracy(y_pred_val, y_val))

        # Evaluate the trained classifier on the test dataset
        y_pred_test = linear_regression.predict(x_test)
        test_accuracies.append(get_classification_accuracy(y_pred_test, y_test))

    return train_accuracies, val_accuracies, test_accuracies, weights
```

2.0.2 Plot the Accuracies vs epoch graphs

[7]:
```
import matplotlib.pyplot as plt


def plot_accuracies(train_acc, val_acc, test_acc):
    # Plot Accuracies vs Epochs graph for all the three
    epochs = np.arange(0, int(num_epochs_total / epochs_per_evaluation))
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch/10")
    plt.plot(epochs, train_acc, epochs, val_acc, epochs, test_acc)
    plt.legend(["Training", "Validation", "Testing"])
    plt.show()
```

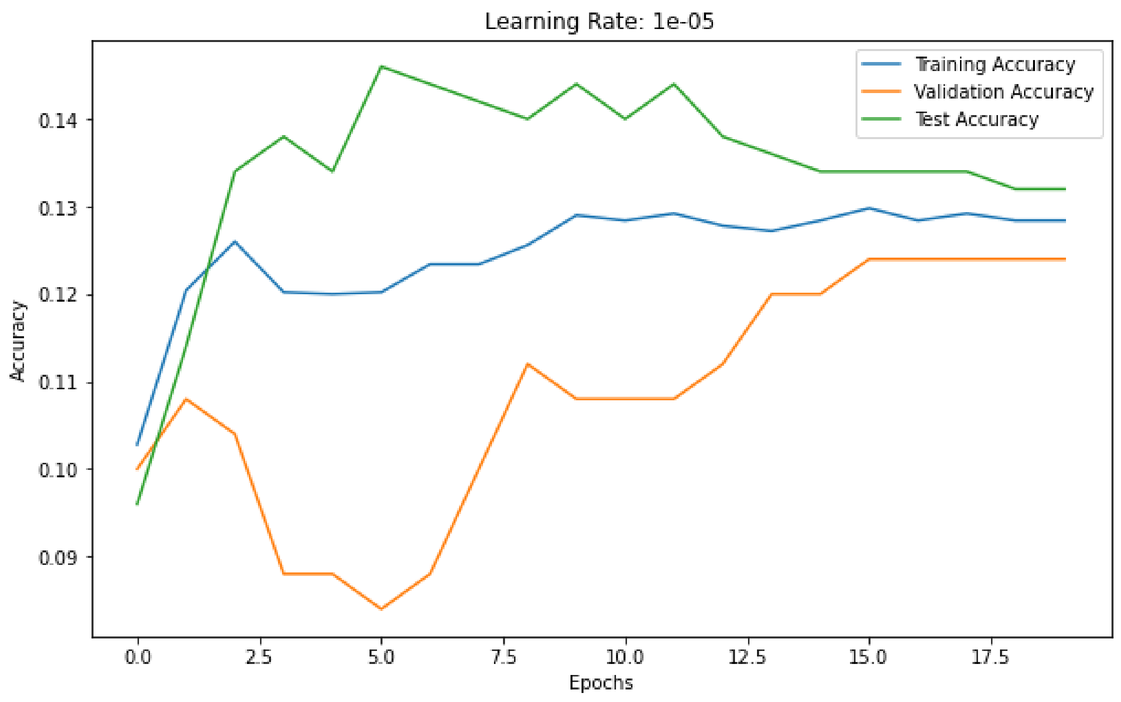
[8]:
```
# Run training and plotting for default parameter values as mentioned above
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
```

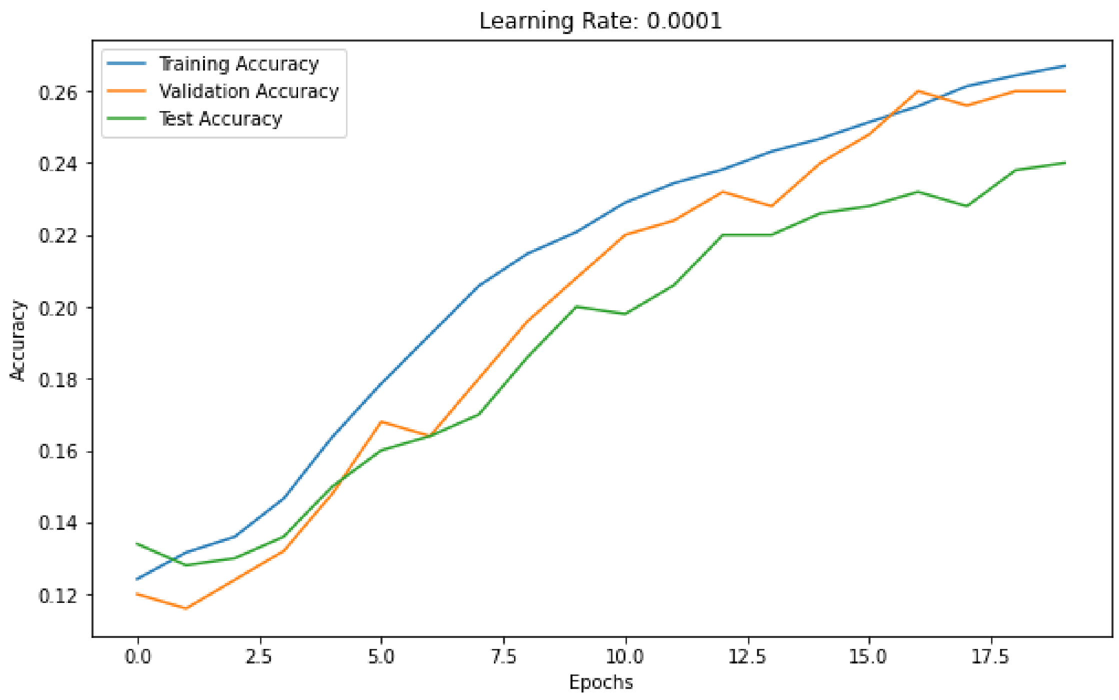
[9]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```

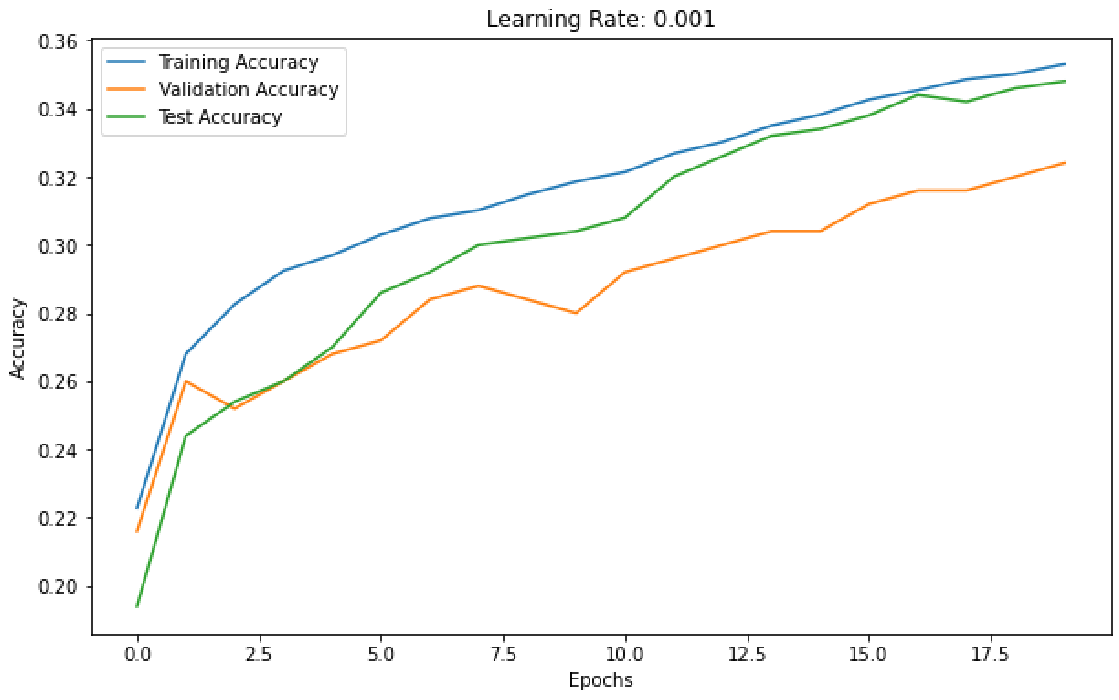


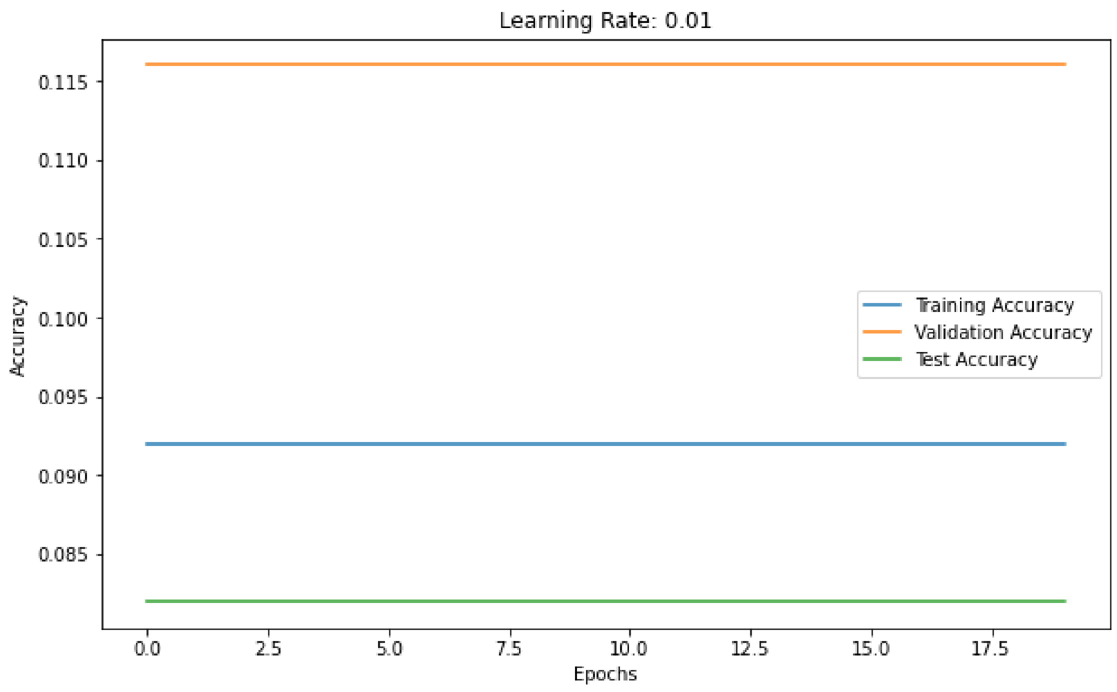
2.0.3 Try different learning rates and plot graphs for all (20%)

[10]:
```
# Initialize the best values
best_weights = weights
best_learning_rate = learning_rate
best_weight_decay = weight_decay
best_val_accuracy = 0.0

# TODO
# Repeat the above training and evaluation steps for the following learning␣
↪rates and plot graphs
# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your learning rate experiments
learning_rates = [0.00001, 0.0001, 0.001, 0.01]
weight_decay = 0.0  # No regularization for now

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
↪ACHIEVE A BETTER PERFORMANCE

# for lr in learning_rates: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)

for learning_rate in learning_rates:
    # TODO: Train the classifier with different learning rates and plot
    train_accu, val_accu, test_accu, weights = train(learning_rate,␣
↪weight_decay)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(train_accu, label='Training Accuracy')
    plt.plot(val_accu, label='Validation Accuracy')
    plt.plot(test_accu, label='Test Accuracy')
    plt.title(f'Learning Rate: {learning_rate}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

    # Update best values if needed
    if val_accu[-1] > best_val_accuracy:
        best_weights = weights
        best_learning_rate = learning_rate
        best_weight_decay = weight_decay
```




Learning Rate: 0.0001
Training Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Epochs


Learning Rate: 0.001
Training Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Epochs



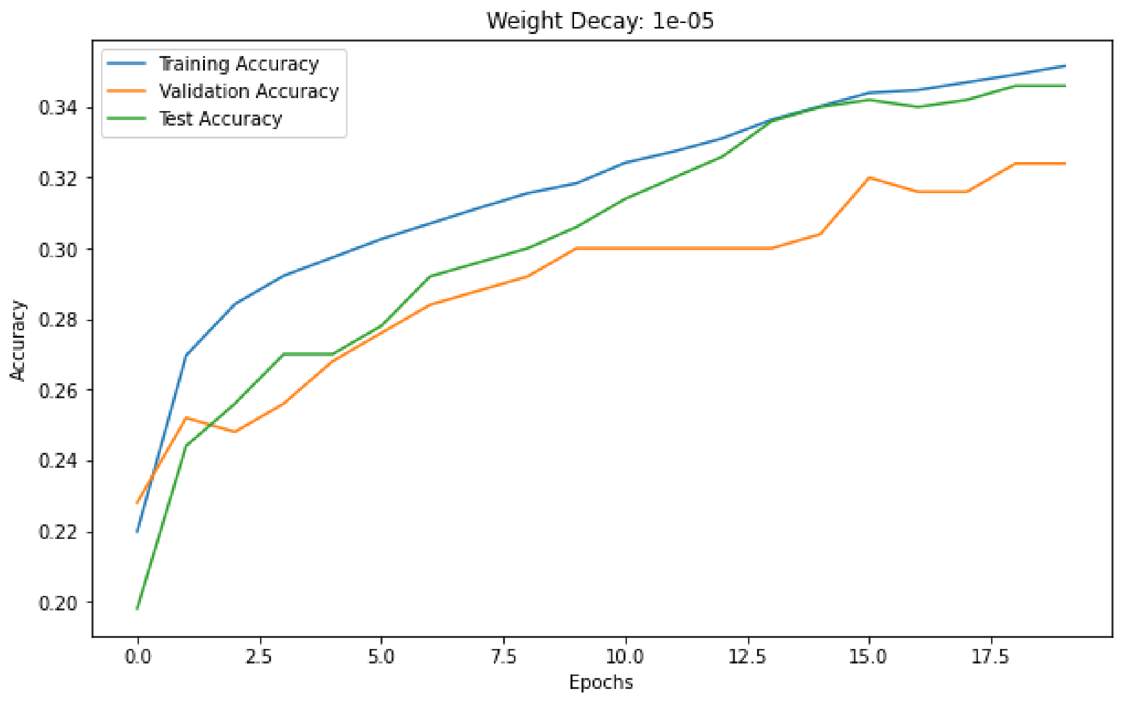
Inline Question 1. Which one of these learning rates (best_lr) would you pick to train your model? Please Explain why.

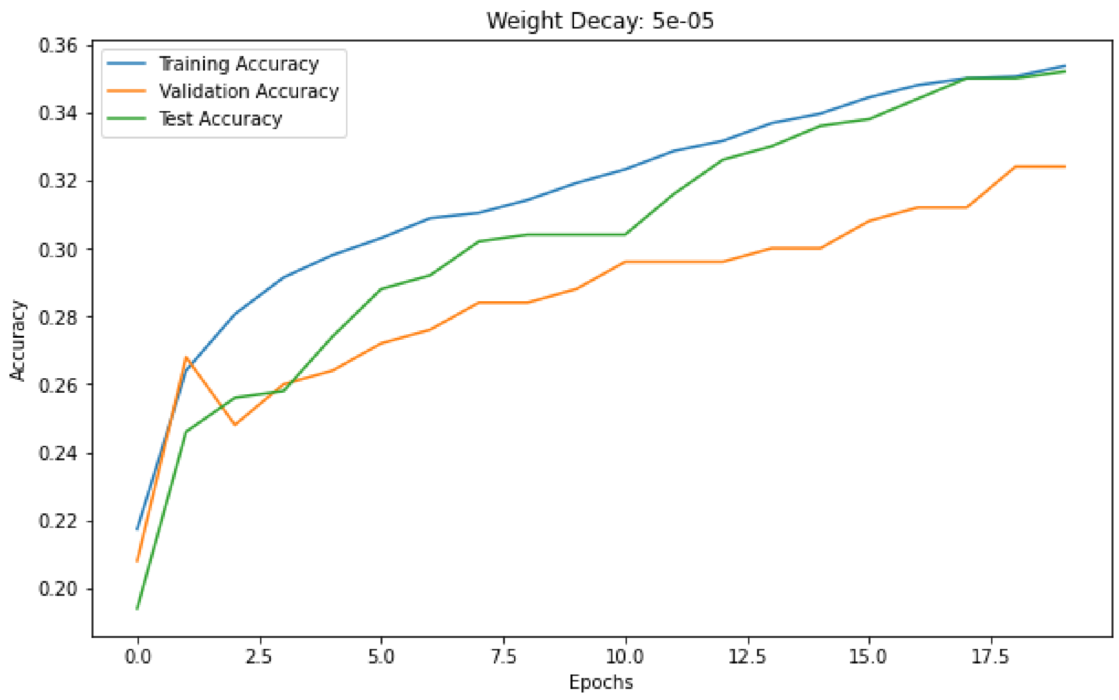
Your Answer: I would choose the first learning rate of 0.001, because it leads to the highest accuracy and most stable/useful results. We can see that 0.01 already flatlines which makes it difficult to analyze the data, and that 0.1 is too large that it leads to numerical instability, with there being an error from the multiplication. This is likely from an exploding gradient due to the large learning rate. Just for reference, I went back and added a few more learning rates. Even after adding 0.00001 and 0.0001, I would still choose 0.001 as it has the highest peak accuracy. Note: I removed 0.1 for the submission to exclude the error from the gradient.

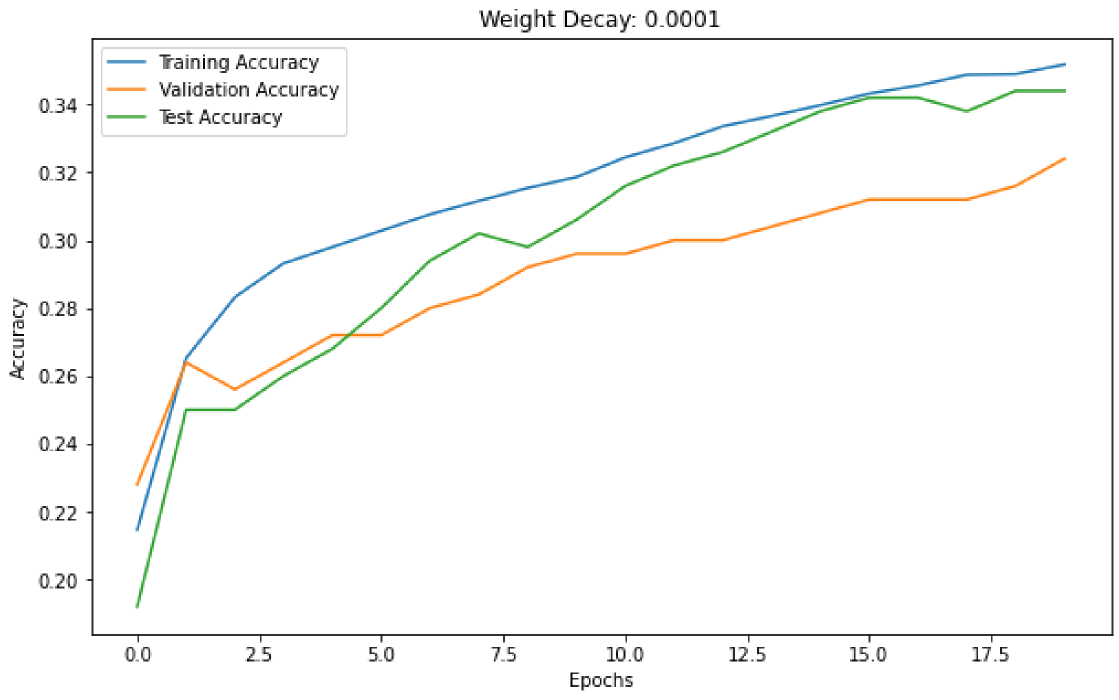
2.0.4 Regularization: Try different weight decay and plot graphs for all (20%)

[11]:
```
# Initialize a non-zero weight_decay (Regularization constant) term and repeat␣
↪the training and evaluation
# Use the best learning rate as obtained from the above exercise, best_lr

# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your weight decay experiments
best_lr = 0.001
weight_decays = [0.00001, 0.00005, 0.0001]

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
↪ACHIEVE A BETTER PERFORMANCE
```

```
# for weight_decay in weight_decays: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(best_lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)

for weight_decay in weight_decays:
    # TODO: Train the classifier with different weighty decay and plot
    train_accu, val_accu, test_accu, weights = train(best_lr, weight_decay)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(train_accu, label='Training Accuracy')
    plt.plot(val_accu, label='Validation Accuracy')
    plt.plot(test_accu, label='Test Accuracy')
    plt.title(f'Weight Decay: {weight_decay}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```







Inline Question 2. Discuss underfitting and overfitting as observed in the 3 graphs obtained by changing the regularization. Which weight_decay term gave you the best classifier performance?

HINT: Do not just think in terms of best training set performance, keep in mind that the real utility of a machine learning model is when it performs well on data it has never seen before

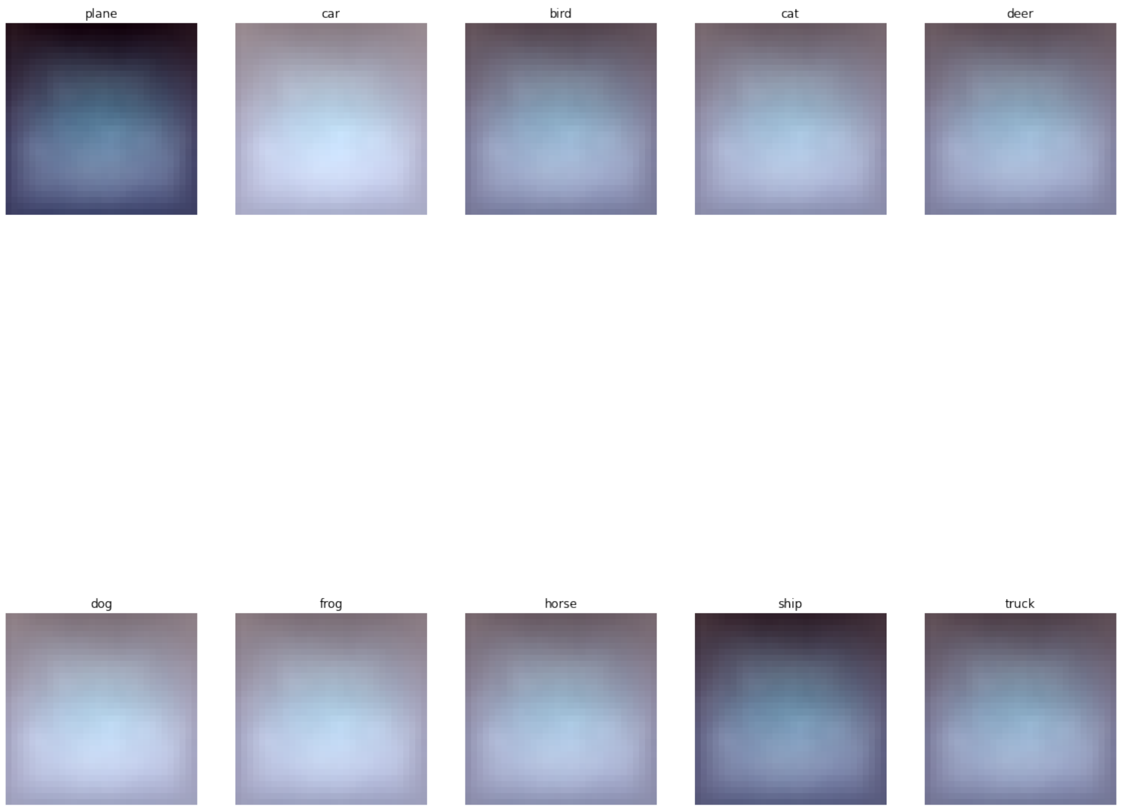
Your Answer: When changing the regularization term, low regularization may lead to overfitting, struggling to generalize to the validation set. On the other hand, high regularization may lead to underfitting, where the model is too constrained to capture underlying patterns. The ideal is a moderate regularization that strikes a balance between the two, which allows us to have a good generalization to both the training and validation sets. The best classifier performance is often associated with moderate regularization, as it prevents overfitting without sacrificing the model's ability to generalize to new, unseen data. The choice of the weight_decay term should prioritize how well the model performs on a separate test set, emphasizing the importance of evaluating generalization performance beyond the training and validation sets. Adjustments to weight_decay can be fine-tuned based on observed behavior during training and validation.

2.0.5 Visualize the filters (10%)

[12]:
```
# These visualizations will only somewhat make sense if your learning rate and␣
↪weight_decay parameters were
# properly chosen in the model. Do your best.

# TODO: Run this cell and Show filter visualizations for the best set of␣
↪weights you obtain.
# Report the 2 hyperparameters you used to obtain the best model.

# NOTE: You need to set `best_learning_rate` and `best_weight_decay` to the␣
↪values that gave the highest accuracy
best_learning_rate = 0.001
best_weight_decay = 0.00005
print("Best LR:", best_learning_rate)
print("Best Weight Decay:", best_weight_decay)

# NOTE: You need to set `best_weights` to the weights with the highest accuracy
w = best_weights[:, :-1]
w = w.reshape(10, 3, 32, 32).transpose(0, 2, 3, 1)

w_min, w_max = np.min(w), np.max(w)

fig = plt.figure(figsize=(20, 20))
classes = [
    "plane",
    "car",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
```

```python
    "ship",
    "truck",
]
for i in range(10):
    fig.add_subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[i, :, :, :].squeeze() - w_min) / (w_max - w_min)
    # plt.imshow(wimg.astype('uint8'))
    plt.imshow(wimg.astype(int))
    plt.axis("off")
    plt.title(classes[i])
plt.show()
```

```
Best LR: 0.001
Best Weight Decay: 5e-05
```



logistic_regression

January 28, 2024

1 ECE 176 Assignment 2: Logistic Regression

For this part of assignment, you are tasked to implement a logistic regression algorithm for multi-class classification and test it on the CIFAR10 dataset.

You sould run the whole notebook and answer the questions in the notebook.

TO SUBMIT: PDF of this notebook with all the required outputs and answers.

[1]:
```
!python get_datasets.py
```

```
Downloading cifar-10
Done
```

[2]:
```
# Prepare Packages
import numpy as np
import matplotlib.pyplot as plt

from utils.data_processing import get_cifar10_data

# Use a subset of CIFAR10 for KNN assignments
dataset = get_cifar10_data(
    subset_train=5000,
    subset_val=250,
    subset_test=500,
)

print(dataset.keys())
print("Training Set Data  Shape: ", dataset["x_train"].shape)
print("Training Set Label Shape: ", dataset["y_train"].shape)
print("Validation Set Data  Shape: ", dataset["x_val"].shape)
print("Validation Set Label Shape: ", dataset["y_val"].shape)
print("Test Set Data  Shape: ", dataset["x_test"].shape)
print("Test Set Label Shape: ", dataset["y_test"].shape)
```

```
dict_keys(['x_train', 'y_train', 'x_val', 'y_val', 'x_test', 'y_test'])
Training Set Data  Shape:  (5000, 3072)
Training Set Label Shape:  (5000,)
Validation Set Data  Shape:  (250, 3072)
Validation Set Label Shape:  (250,)
Test Set Data  Shape:  (500, 3072)
Test Set Label Shape:  (500,)
```

2 Logistic Regression for multi-class classification

A Logistic Regression Algorithm has these hyperparameters:

- **Learning rate** - controls how much we change the current weights of the classifier during each update. We set it at a default value of 0.5, and later you are asked to experiment with different values. We recommend looking at the graphs and observing how the performance of the classifier changes with different learning rate.
- **Number of Epochs** - An epoch is a complete iterative pass over all of the data in the dataset. During an epoch we predict a label using the classifier and then update the weights of the classifier according the linear classifier update rule for each sample in the training set. We evaluate our models after every 10 epochs and save the accuracies, which are later used to plot the training, validation and test VS epoch curves.
- **Weight Decay** - Regularization can be used to constrain the weights of the classifier and prevent their values from blowing up. Regularization helps in combatting overfitting. You will be using the 'weight_decay' term to introduce regularization in the classifier.

The only way how a Logistic Regression based classification algorithm is different from a Linear Regression algorithm is that in the former we additionally pass the classifier outputs into a sigmoid function which squashes the output in the (0,1) range. Essentially these values then represent the probabilities of that sample belonging to class particular classes

2.0.1 Implementation (40%)

You need to implement the Linear Regression method in `algorithms/logistic_regression.py`. The formulations follow the lecture (consider binary classification for each of the 10 classes, with labels -1 / 1 for not belonging / belonging to the class). You need to fill in the training function as well as the prediction function. You need to fill in the sigmoid function, training function as well as the prediction function.

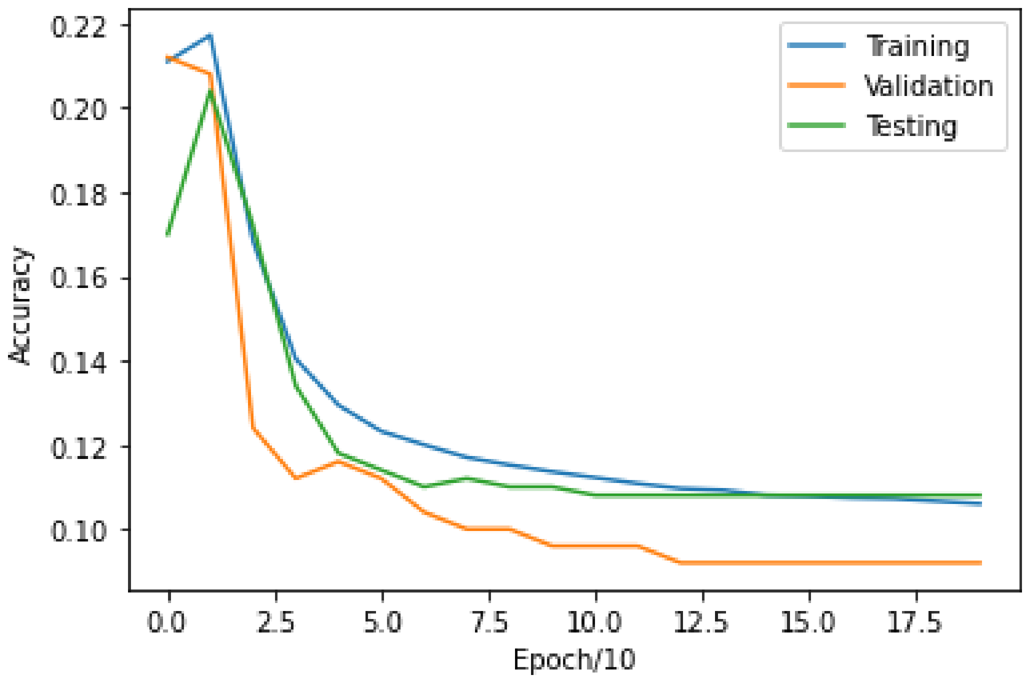
[3]:
```python
# Import the algorithm implementation (TODO: Complete the Logistic Regression␣
↪in algorithms/logistic_regression.py)
from algorithms import Logistic
from utils.evaluation import get_classification_accuracy

num_classes = 10  # Cifar10 dataset has 10 different classes

# Initialize hyper-parameters
learning_rate = 0.01  # You will be later asked to experiment with different␣
↪learning rates and report results
num_epochs_total = 200  # Total number of epochs to train the classifier
epochs_per_evaluation = 10  # Epochs per step of evaluation; We will evaluate␣
↪our model regularly during training
N, D = dataset[
    "x_train"
].shape  # Get training data shape, N: Number of examples, D:Dimensionality of␣
 ↪the data
weight_decay = 0.00002

x_train = dataset["x_train"].copy()
y_train = dataset["y_train"].copy()
x_val = dataset["x_val"].copy()
y_val = dataset["y_val"].copy()
x_test = dataset["x_test"].copy()
y_test = dataset["y_test"].copy()

# Insert additional scalar term 1 in the samples to account for the bias as␣
 ↪discussed in class
x_train = np.insert(x_train, D, values=1, axis=1)
x_val = np.insert(x_val, D, values=1, axis=1)
x_test = np.insert(x_test, D, values=1, axis=1)
```

[4]:
```
# Training and evaluation function -> Outputs accuracy data
def train(learning_rate_, weight_decay_):
    # Create a linear regression object
    logistic_regression = Logistic(
        num_classes, learning_rate_, epochs_per_evaluation, weight_decay_
    )

    # Randomly initialize the weights and biases
    weights = np.random.randn(num_classes, D + 1) * 0.0001

    train_accuracies, val_accuracies, test_accuracies = [], [], []

    # Train the classifier
    for _ in range(int(num_epochs_total / epochs_per_evaluation)):
        # Train the classifier on the training data
        weights = logistic_regression.train(x_train, y_train, weights)

        # Evaluate the trained classifier on the training dataset
        y_pred_train = logistic_regression.predict(x_train)
        train_accuracies.append(get_classification_accuracy(y_pred_train,␣
 ↪y_train))

        # Evaluate the trained classifier on the validation dataset
        y_pred_val = logistic_regression.predict(x_val)
        val_accuracies.append(get_classification_accuracy(y_pred_val, y_val))

        # Evaluate the trained classifier on the test dataset
        y_pred_test = logistic_regression.predict(x_test)
        test_accuracies.append(get_classification_accuracy(y_pred_test, y_test))
    return train_accuracies, val_accuracies, test_accuracies, weights
```

[5]:
```
import matplotlib.pyplot as plt


def plot_accuracies(train_acc, val_acc, test_acc):
    # Plot Accuracies vs Epochs graph for all the three
    epochs = np.arange(0, int(num_epochs_total / epochs_per_evaluation))
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch/10")
    plt.plot(epochs, train_acc, epochs, val_acc, epochs, test_acc)
    plt.legend(["Training", "Validation", "Testing"])
    plt.show()
```

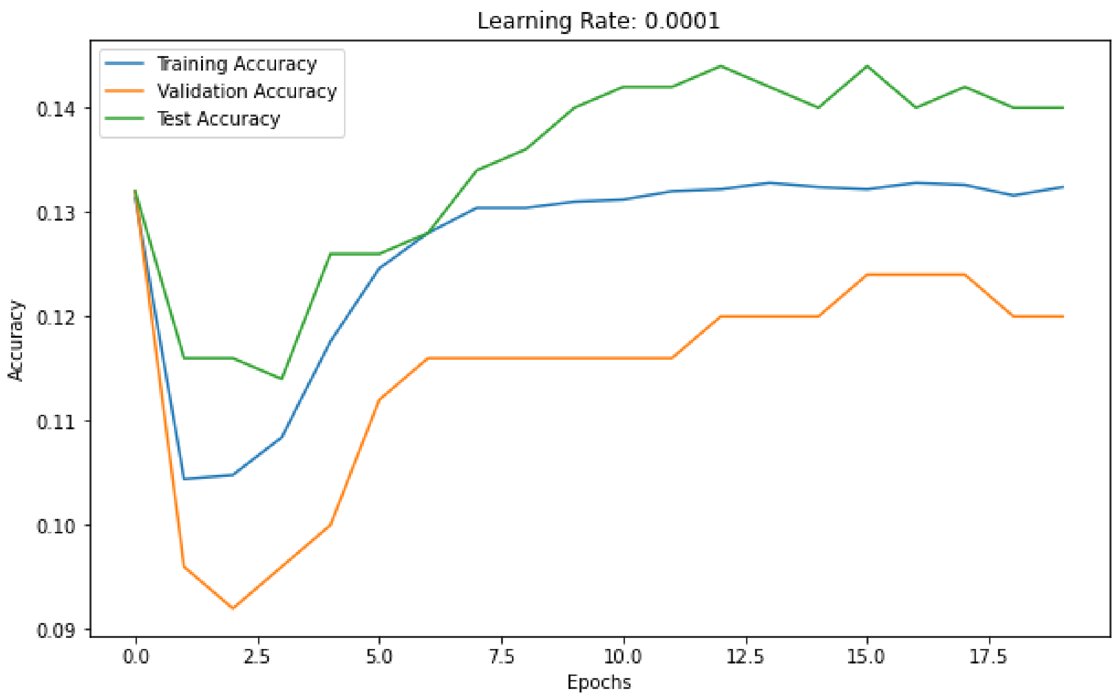
[6]:
```
# Run training and plotting for default parameter values as mentioned above
t_ac, v_ac, te_ac, weights = train(learning_rate, weight_decay)
```

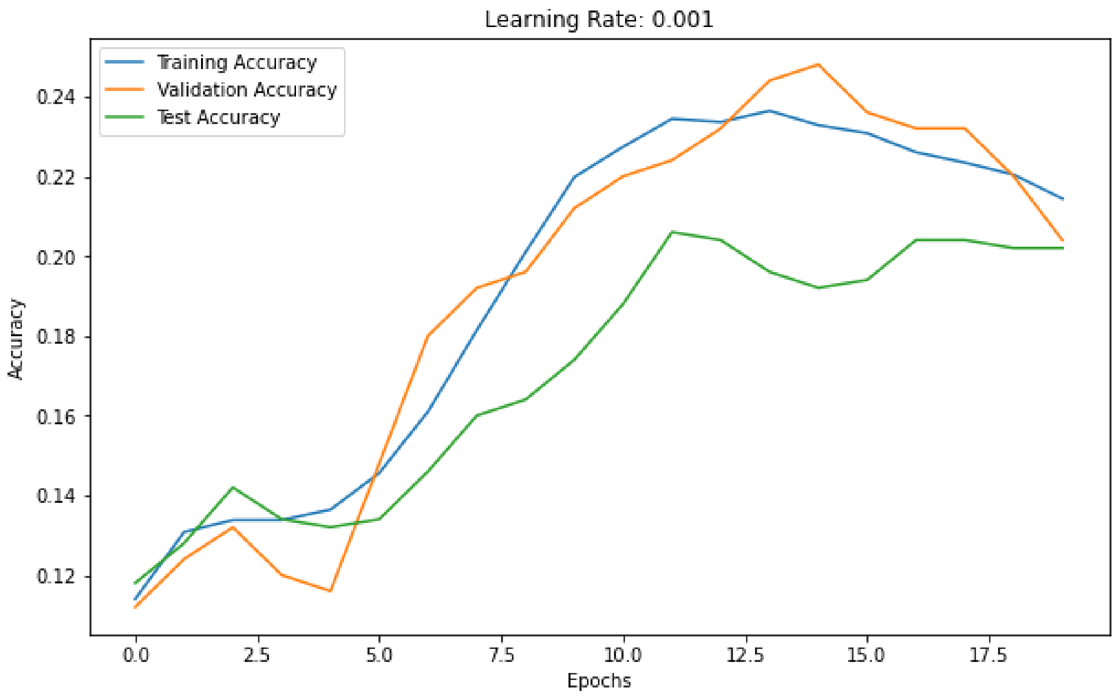
[7]:
```
plot_accuracies(t_ac, v_ac, te_ac)
```

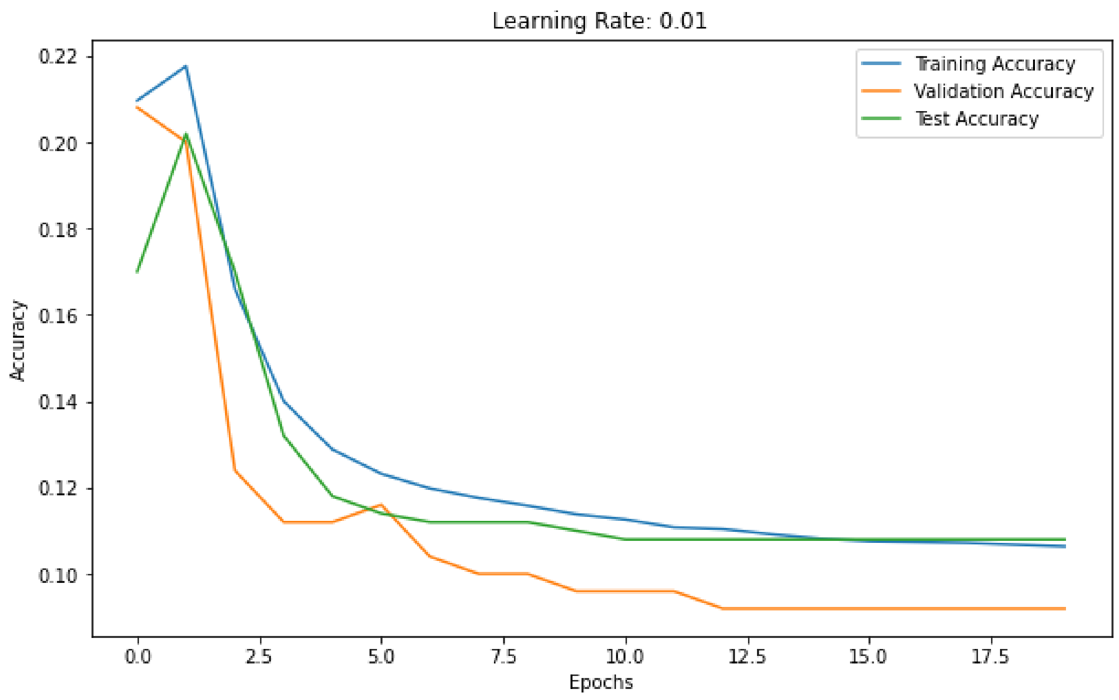


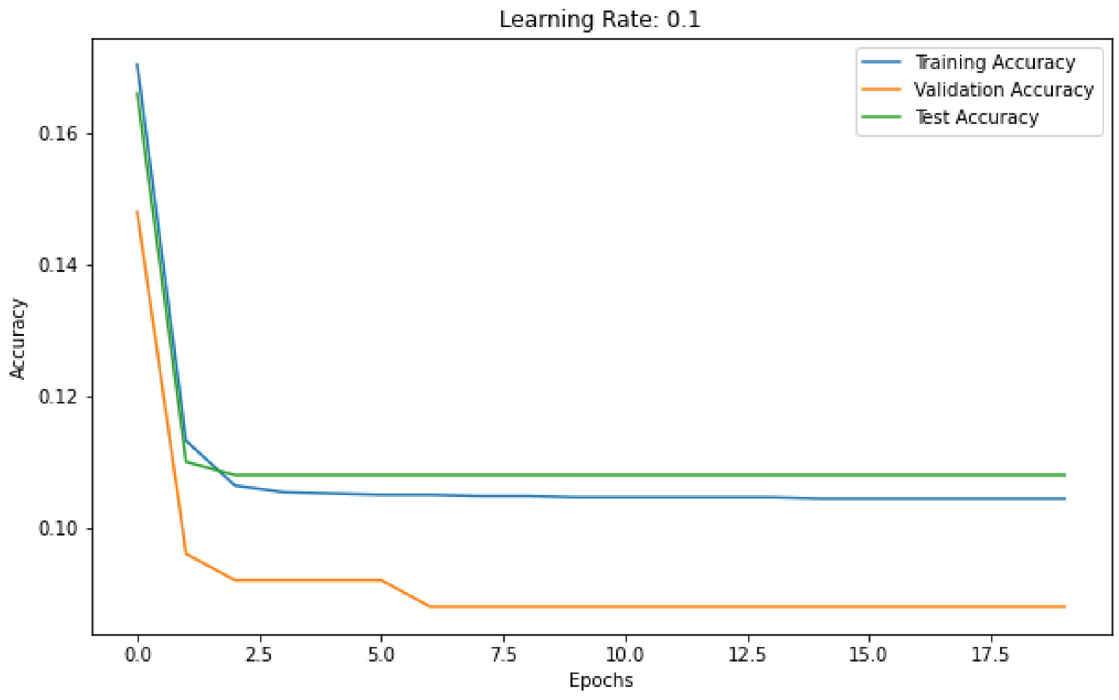
2.0.2 Try different learning rates and plot graphs for all (20%)

[8]:
```
# Initialize the best values
best_weights = weights
best_learning_rate = learning_rate
best_weight_decay = weight_decay
best_val_accuracy = 0.0

# TODO
# Repeat the above training and evaluation steps for the following learning␣
↪rates and plot graphs
# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your learning rate experiments
learning_rates = [0.0001, 0.001, 0.01, 0.1]
weight_decay = 0.0  # No regularization for now

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
↪ACHIEVE A BETTER PERFORMANCE

# for lr in learning_rates: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)

for learning_rate in learning_rates:
    # TODO: Train the classifier with different learning rates and plot
    train_accu, val_accu, test_accu, weights = train(learning_rate,␣
↪weight_decay)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(train_accu, label='Training Accuracy')
    plt.plot(val_accu, label='Validation Accuracy')
    plt.plot(test_accu, label='Test Accuracy')
    plt.title(f'Learning Rate: {learning_rate}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

    # Update best values if needed
    if val_accu[-1] > best_val_accuracy:
        best_weights = weights
        best_learning_rate = learning_rate
        best_weight_decay = weight_decay
```


Learning Rate: 0.0001
Training Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Epochs
0.09
0.10
0.11
0.12
0.13
0.14
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Learning Rate: 0.001
Training Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Epochs
0.12
0.14
0.16
0.18
0.20
0.22
0.24
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5



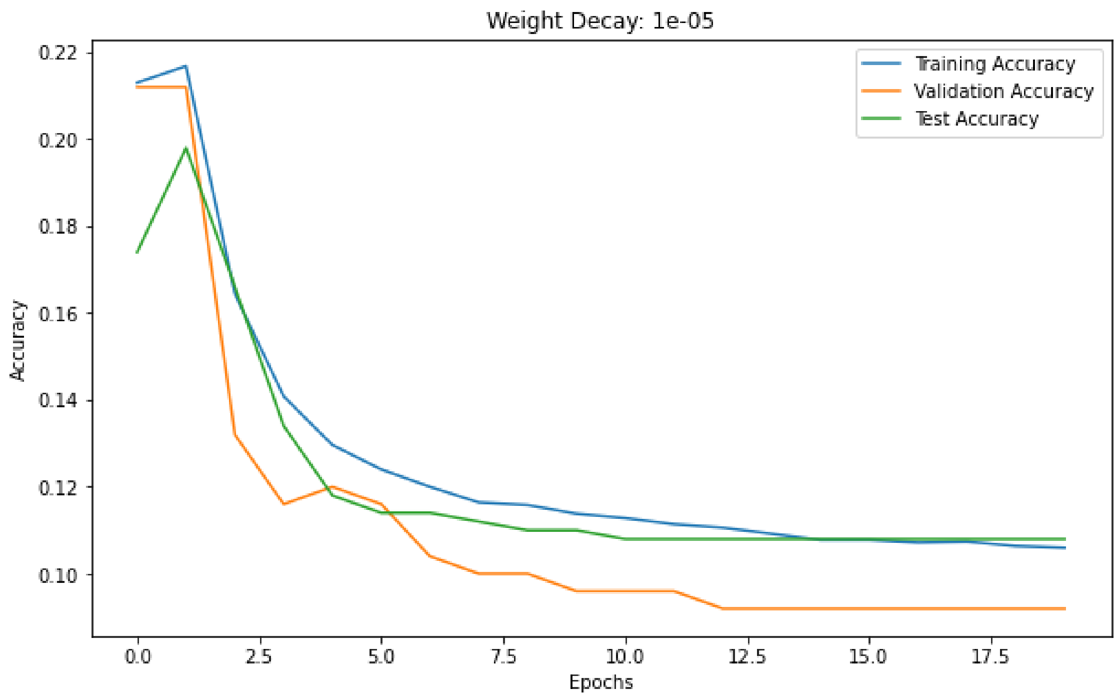


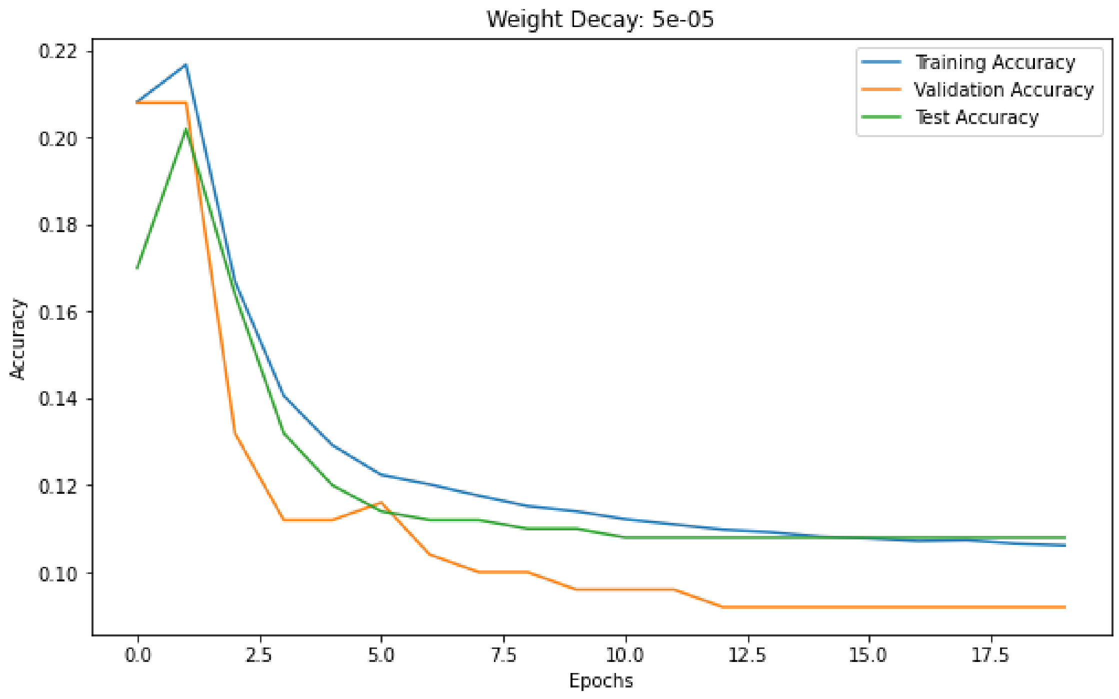
Inline Question 1. Which one of these learning rates (best_lr) would you pick to train your model? Please Explain why.

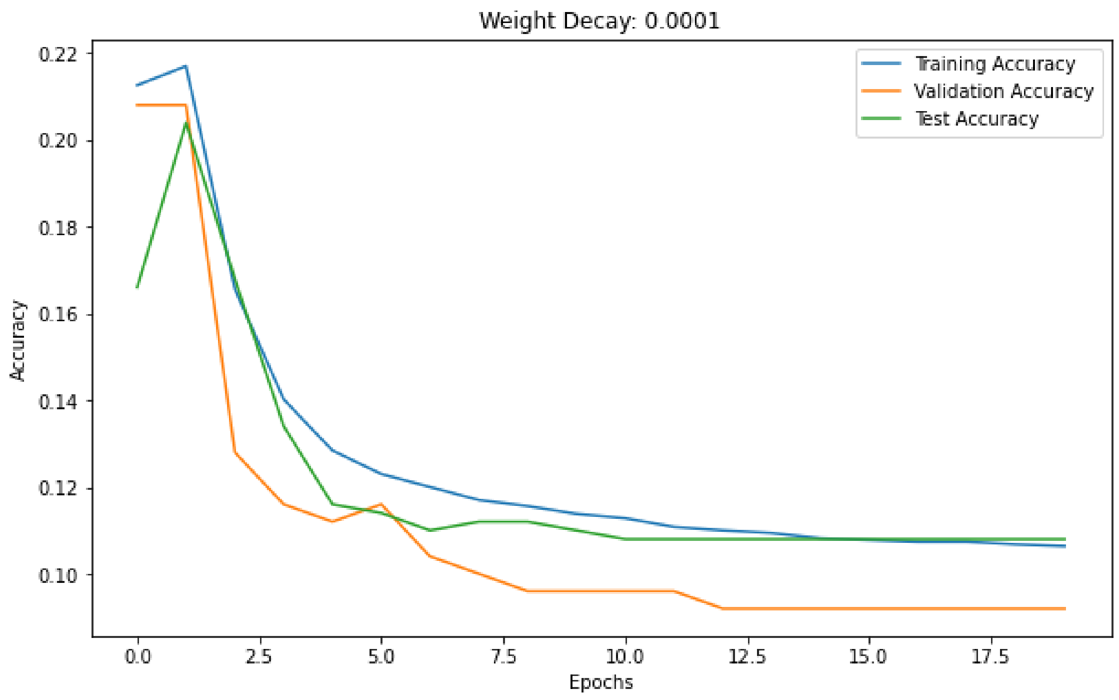
Your Answer: I would choose the learning rate of 0.001, because it leads to the highest accuracy and most stable/useful results. We can see that at the extremes of 0.0001 and 0.1, the lines are either not fitted enough or too tightly fitted. 0.001 is clearly the best learning rate among these choices as it reached around 20% accuracy when the other learning rates fail to do the same.

2.0.3 Regularization: Try different weight decay and plots graphs for all (20%)

[9]:
```python
# Initialize a non-zero weight_decay (Regulzarization constant) term and repeat␣
↪the training and evaluation
# Use the best learning rate as obtained from the above excercise, best_lr

# You need to try 3 learning rates and submit all 3 graphs along with this␣
↪notebook pdf to show your weight decay experiments
best_lr = 0.01
weight_decays = [0.00001, 0.00005, 0.0001]

# FEEL FREE TO EXPERIMENT WITH OTHER VALUES. REPORT OTHER VALUES IF THEY␣
↪ACHIEVE A BETTER PERFORMANCE

# for weight_decay in weight_decays: Train the classifier and plot data
# Step 1. train_accu, val_accu, test_accu = train(best_lr, weight_decay)
# Step 2. plot_accuracies(train_accu, val_accu, test_accu)

for weight_decay in weight_decays:
    # TODO: Train the classifier with different weighty decay and plot
    train_accu, val_accu, test_accu, weights = train(best_lr, weight_decay)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(train_accu, label='Training Accuracy')
    plt.plot(val_accu, label='Validation Accuracy')
    plt.plot(test_accu, label='Test Accuracy')
    plt.title(f'Weight Decay: {weight_decay}')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```


Weight Decay: 1e-05
Training Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Epochs
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Weight Decay: 5e-05
Training Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Epochs
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5



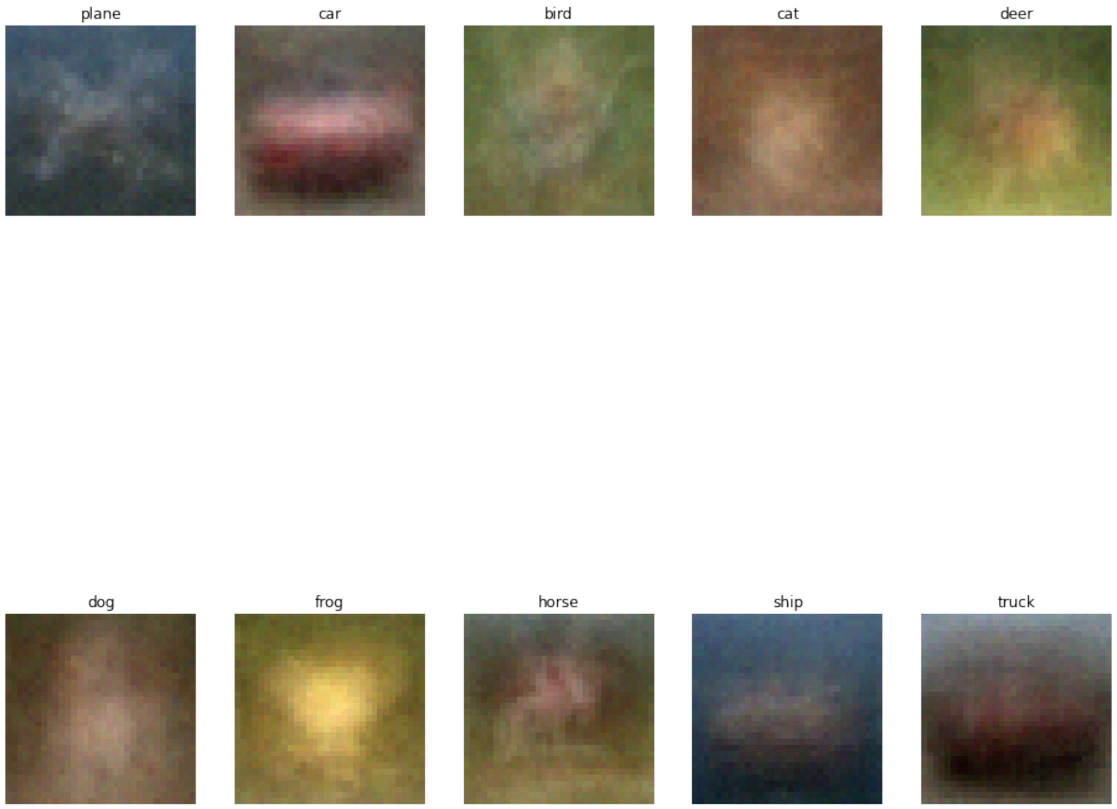
Inline Question 2. Discuss underfitting and overfitting as observed in the 3 graphs obtained by changing the regularization. Which weight_decay term gave you the best classifier performance? HINT: Do not just think in terms of best training set performance, keep in mind that the real utility of a machine learning model is when it performs well on data it has never seen before

Your Answer: Underfitting occurs when the model is too simple and regularization is excessively high, leading to insufficient complexity to capture underlying patterns. On the other hand, overfitting arises when the model is overly complex, often due to low regularization, causing it to memorize noise in the training data without generalizing well to new examples. We want a balance that minimizes overfitting while maintaining good generalization on a separate validation or a new test set. With that being said, the weight decay that gave me the best classifier performance was 0.00005. This highlights the importance of hyperparameter tuning that considers the model's ability to perform well on unseen data, in order to ensure the real-world utility of the machine learning model.

2.0.4 Visualize the filters (10%)

[12]:
```
# These visualizations will only somewhat make sense if your learning rate and␣
↪weight_decay parameters were
# properly chosen in the model. Do your best.

# TODO: Run this cell and Show filter visualizations for the best set of␣
↪weights you obtain.
# Report the 2 hyperparameters you used to obtain the best model.
# NOTE: You need to set `best_learning_rate` and `best_weight_decay` to the␣
↪values that gave the highest accuracy
best_learning_rate = 0.001
best_weight_decay = 0.00005
print("Best LR:", best_learning_rate)
print("Best Weight Decay:", best_weight_decay)

# NOTE: You need to set `best_weights` to the weights with the highest accuracy
w = best_weights[:, :-1]
w = w.reshape(10, 3, 32, 32).transpose(0, 2, 3, 1)

w_min, w_max = np.min(w), np.max(w)

fig = plt.figure(figsize=(16, 16))
classes = [
    "plane",
    "car",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
    "truck",
]
for i in range(10):
    fig.add_subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[i, :, :, :].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype(int))
    plt.axis("off")
    plt.title(classes[i])
plt.show()
```

```
Best LR: 0.001
Best Weight Decay: 5e-05
```



Inline Question 3. (9%)

a. Compare and contrast the performance of the 2 classifiers i.e. Linear Regression and Logistic Regression.
b. Which classifier would you deploy for your multiclass classification project and why?

a) Linear regression predicts continuous outcomes by fitting a linear equation, making it unsuitable for classification tasks, where as logistic regression is specifically designed for binary classification, providing probability scores between 0 and 1. While Linear Regression is not ideal for classification due to its continuous output, Logistic Regression is well-suited for binary classification scenarios.

b) For a multiclass classification project, logistic regression is the more practical choice. Logistic regression can be extended to handle multiple classes using methods like one-vs-rest or multinomial approaches. Its computational efficiency, interpretability, and ability to provide probabilities make it advantageous for multiclass problems. However, the decision to use logistic regression should consider the linearity of relationships between features and classes. If the problem exhibits complex, non-linear patterns, other models such as Decision Trees or Neural Networks may be achieve better performance.

Your Answer:

2.1 Survey (1%)

2.1.1 Question:

How many hours did you spend on assignment 2?

2.1.2 Your Answer:

Roughly 9 hours.