Technical Report

CS5062

Machine Learning

Assignment 1: Machine Learning Implementation

Author:
Doroteya Stoyanova

October 30, 2024

Contents

1 Task 1: Titanic Survival Prediction using Logistic Regression

1.1 Introduction

In recent years, the Titanic dataset has become a popular resource for developing predictive models of passenger survival. Our task is to create a Logistic Regression (LR) model from scratch to predict passenger survival based on demographic and socio-economic factors [8].

1.2 Task 1.1: Data EDA and Preprocessing

1.2.1 Data Import and Initial Analysis

First, we import all the necessary libraries and load our dataset:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, KFold,
   GridSearchCV, cross_val_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression as
   SKLogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score,
   classification_report
from sklearn.feature_selection import RFE

# Import the data
df = pd.read_csv('dataset/titanic.csv')
```

1.2.2 Dataset Description

The Titanic dataset contains 891 entries with 11 features describing passenger characteristics. Key variables include:

Feature	Description
Pclass	Passenger class (1st, 2nd, or 3rd class) — a proxy for socio-economic status
Survived	Survival status (0 = No, 1 = Yes)
Name	Passenger's name
Sex	Passenger's gender
Age	Passenger's age in years
SibSp	Number of siblings or spouses aboard
Parch	Number of parents or children aboard
Ticket	Ticket number
Fare	Ticket fare price in British pounds
Cabin	Cabin number
Embarked	Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

Table 1: Titanic dataset features and descriptions

1.2.3 Data Analysis

Initial analysis using pandas' descriptive statistics revealed several key insights:

Metric	**Details**
Survival Rate	Approximately 38% of passengers survived
Class Distribution	Slight skew towards lower classes (mean Pclass of 2.309)
Missing Data	
	Age: 177 missing entries Cabin: 687 missing entries Embarked: 2 missing entries
Data Complexity	Mixed numeric and alphanumeric values in the Ticket column

Table 2: Summary of Titanic Dataset Characteristics

1.2.4 Data Preprocessing

Based on our analysis, we performed the following preprocessing steps:

1. Feature Removal:

```
df = df.drop(['Ticket', 'Cabin', 'Name', 'PassengerId'], axis=1)
```

We removed several features that were either problematic or less relevant, for instance, Name was irrelevant, and Cabin had too many missing values.

For handling missing values, we employed the following strategies:

```
df['Age'] = df['Age'].fillna(df['Age'].median())
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
```

- Age: Median imputation to handle skewed distribution and outliers
- Embarked: Mode imputation for categorical data preservation

We also used sklearn's LabelEncoder to convert categorical variables into a numerical format for model compatibility. The 'Sex' variable was encoded as 0 for females and 1 for males, while 'Embarked' was similarly transformed into numerical values.

A new variable was created, 'Family_Size,' by combining 'SibSp' (siblings/spouses) and 'Parch' (parents/children) and adding 1 for the passenger. We hypothesized that family size could impact survival probability.

```
df['Family_Size'] = df['SibSp'] + df['Parch'] + 1
```

We also created the correlation matrix between the variables, so that we observe the trends(See Fig.1).

The most significant observation from the correlation matrix is that survival was strongly influenced by sex, with females having a notably higher chance of survival (correlation of -0.543351).

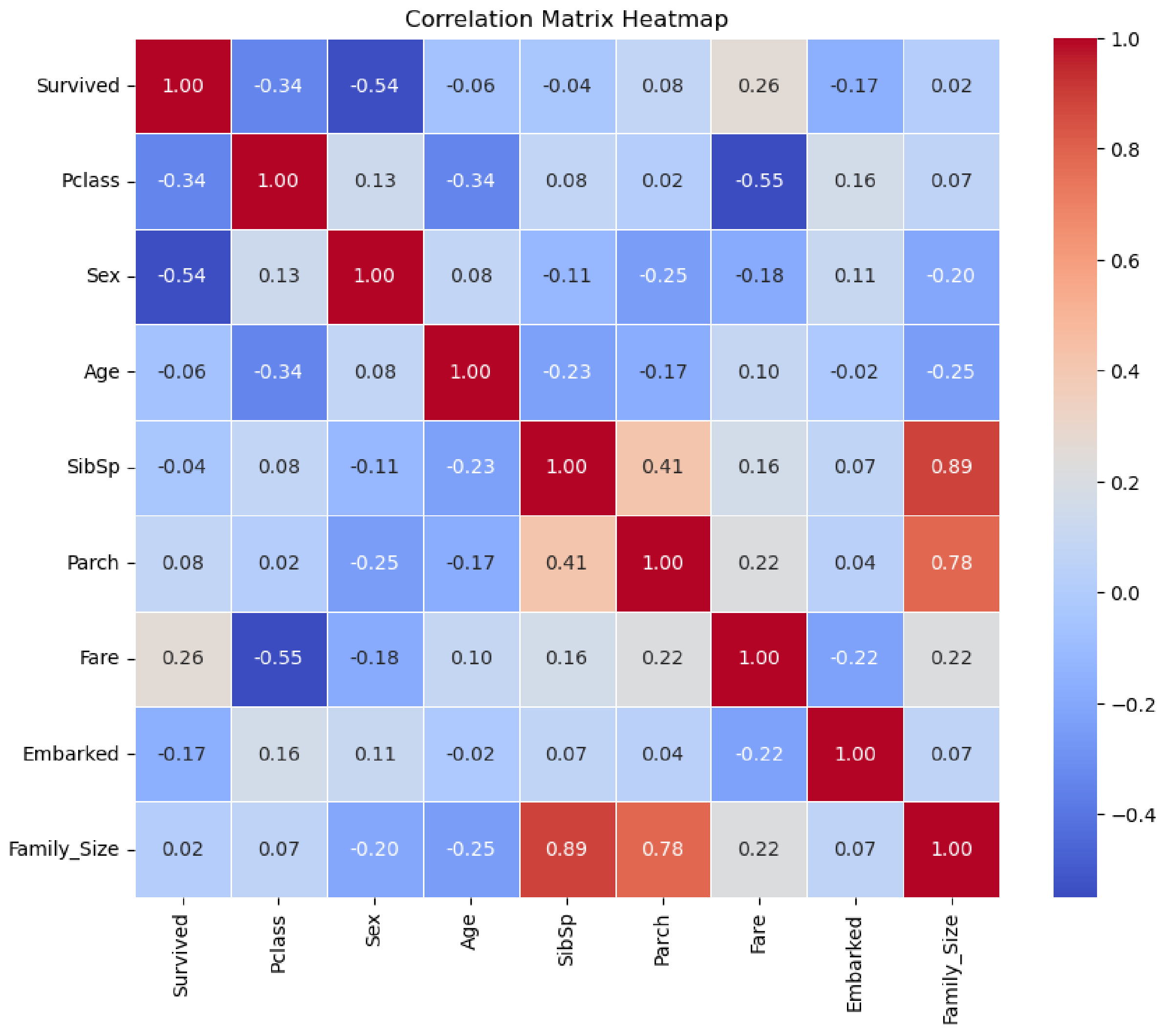


Figure 1: Correlation Matrix of Features

1.3 Task 2: Model Scenario

When selecting a model to predict Titanic survival, it is critical to look beyond simple test accuracy. For instance, a model that predicts every passenger did not survive could achieve around 90% accuracy, yet it would fail to identify any survivors. Such high accuracy may signal overfitting, where the model performs well on training data but generalizes poorly to new data.

Logistic Regression (LR) is a valuable choice for this binary classification task, as it is well-suited to handle categorical predictions effectively. To gain a more comprehensive understanding of model performance, using evaluation metrics like the confusion matrix, precision, recall, and F1-score is essential. The confusion matrix, for example, provides an insightful breakdown of predictions versus actual outcomes, showcasing both correct predictions and areas where the model falls short. Precision, measures the accuracy of positive predictions, indicating the model's ability to correctly identify survivors without falsely classifying non-survivors as survivors. Recall, on the other hand, reflects the model's sensitivity in identifying all actual survivors, helping to assess how well it captures positive cases without missing any. The F1-score combines both precision and recall into a single metric, giving a balanced perspective on model performance, especially useful when classes (survived vs. not survived) are imbalanced [10].

1.4 Task 3 Logistic Regression Creation

1.4.1 Data Splitting

We split the data into three sets [4]:

- Training: 70%
- Validation: 20%
- Test: 10%

```
X = df.drop('Survived', axis=1)
y = df['Survived']

X_train_val, X_test, y_train_val, y_test = train_test_split(X,
   y, test_size=0.1, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
   y_train_val, test_size=0.2, random_state=42)
```

1.4.2 Feature Selection

We used Recursive Feature Elimination (RFE) to select the top 5 most important features [7]:

```
rfe = RFE(estimator=SKLogisticRegression(random_state=42),
   n_features_to_select=5)
X_train_selected = rfe.fit_transform(X_train_scaled, y_train)
```

We then implemented a custom classification model, initialized with configurable hyperparameters such as learning rate, iteration count, regularization type (L1 or L2), and regularization strength. The `fit` method optimizes weights and biases via gradient descent, iteratively refining them to minimize the error. Using the sigmoid function, we transform the weighted feature combinations into probability scores, and the `predict` method classifies outcomes based on a 0.5 probability threshold. This structure ensures flexibility and control over training dynamics, allowing for fine-tuning based on model requirements.

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, num_iterations=1000,
   penalty='l2', penalty_param=0.0):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.penalty = penalty
        self.penalty_param = penalty_param
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))
    def fit(self, X, y):
        # Implementation details as shown in your original code
        pass

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(linear_model)
        return [1 if i > 0.5 else 0 for i in y_predicted]
```

1.5 Task 4: Model Optimization

We performed hyperparameter tuning using a grid search with the following parameters. For our grid search, we used StraffieldKFold, given our data is not balanced.:

```
    def grid_search(X, y, param_grid, k=5):
    best_score = 0
    best_params = {}

    for lr in param_grid['learning_rate']:
        for num_iter in param_grid['num_iterations']:
            for penalty in param_grid['penalty']:
                for penalty_param in param_grid['penalty_param']:
                    cv_scores = []
                    kf = StratifiedKFold(n_splits=k,
  shuffle=True, random_state=42)

                    for train_index, val_index in kf.split(X, y):
                        X_train_fold, X_val_fold =
  X[train_index], X[val_index]
                        y_train_fold, y_val_fold =
  y.iloc[train_index], y.iloc[val_index]

                        model =
  LogisticRegression(learning_rate=lr, num_iterations=num_iter,

  penalty=penalty, penalty_param=penalty_param)
                        model.fit(X_train_fold, y_train_fold)

                        y_val_pred = model.predict(X_val_fold)
                        fold_accuracy = accuracy(y_val_fold,
  y_val_pred)
                        cv_scores.append(fold_accuracy)

                    mean_cv_score = np.mean(cv_scores)
                    if mean_cv_score > best_score:
                        best_score = mean_cv_score
                        best_params = {'learning_rate': lr,
  'num_iterations': num_iter,
                                    'penalty': penalty,
  'penalty_param': penalty_param}
```

```
                    print(f"lr: {lr}, num_iterations:
   {num_iter}, penalty: {penalty}, "
                          f"penalty_param: {penalty_param}, Mean
   CV Score: {mean_cv_score:.4f}")

    return best_params, best_score
```

```
param_grid = {
    'learning_rate': [0.0001, 0.001, 0.01],
    'num_iterations': [500, 1000, 2000],
    'penalty': ['l1', 'l2'],
    'penalty_param': [0.0, 0.01, 0.1,1.0]
```

The best parameters we found were:

- Learning rate: 0.01
- Number of iterations: 2000
- Penalty: L2
- Penalty parameter: 0.10
- Best Cross-Validation Score: 0.7906

1.6 Task 5: Results and Analysis

1.6.1 Model Performance

Validation Results:

Table 3: Validation Performance Metrics

Class	Precision	Recall	F1-Score	Support
0	0.84	0.84	0.84	103
1	0.72	0.72	0.72	58
Accuracy	0.80			

Test Results:

Table 4: Test Performance Metrics

Class	Precision	Recall	F1-Score	Support
0	0.86	0.81	0.84	54
1	0.74	0.81	0.77	36
Accuracy	0.81			

1.6.2 Feature Importance

```
Feature Importance:
    feature  importance
1       Sex    0.971953
```

```
0    Pclass    0.651570
2       Age    0.185353
3     SibSp    0.178909
4  Embarked    0.161603
```

The model demonstrates consistent performance across validation and test sets, with an overall accuracy of 80% on the validation set and a slight improvement to 81% on the test set, indicating a decent generalization. However, it performs better at predicting non-survivors (Class 0) than survivors (Class 1), likely due to class imbalance within the dataset. Specifically, for survivors, the model has higher recall (0.81) than precision (0.74) in the test set, suggesting it successfully identifies survivors but may also generate some false positives. The balanced F1 scores across classes highlight a fair trade-off between precision and recall, affirming its effectiveness in handling both.

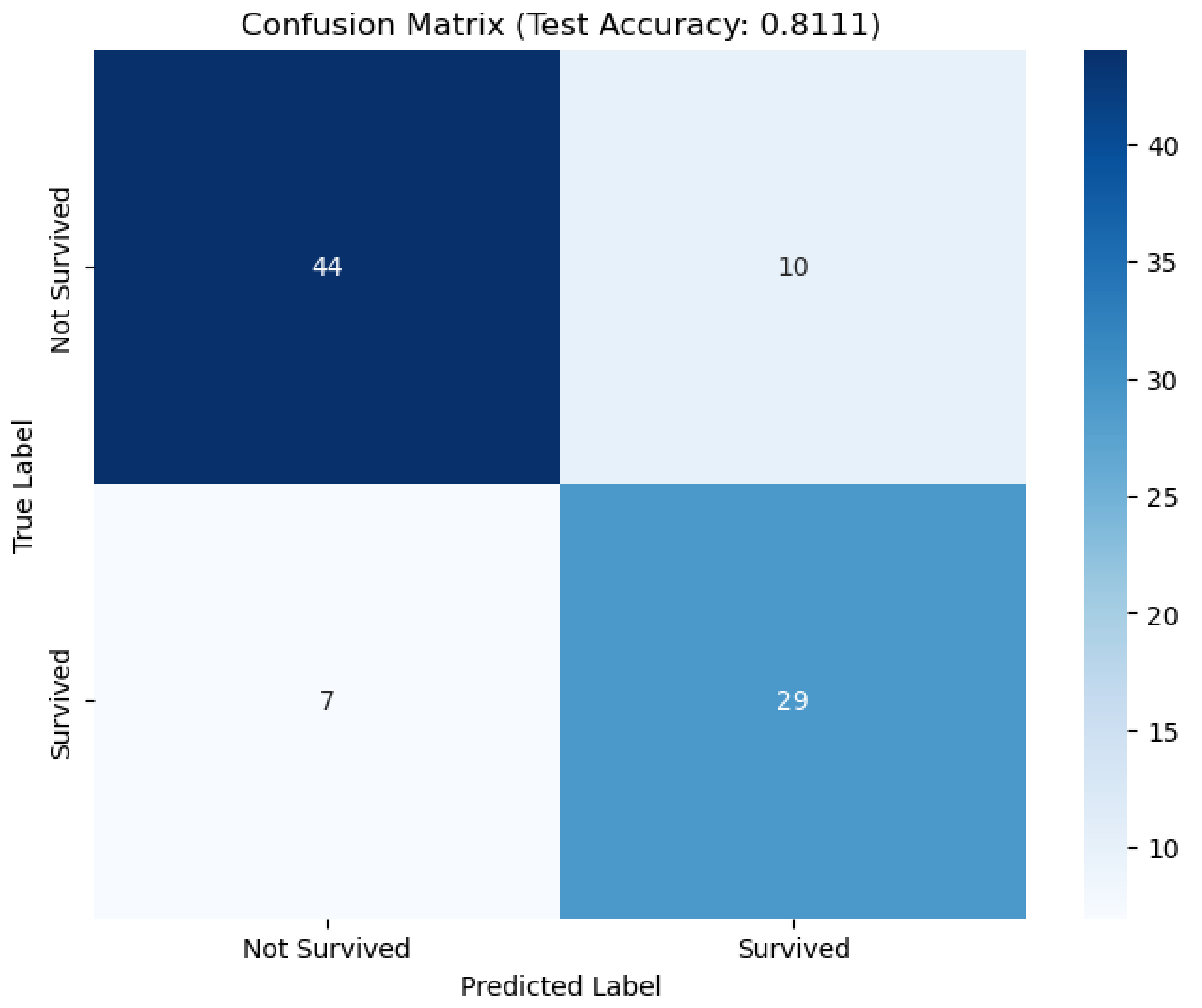


Figure 2: Confusion Matrix of Model Predictions

1.6.3 Task 5: Luck Element

Feature importance analysis reveals 'Sex' as the most critical predictor (0.971953), reflecting the "women and children first" policy observed during the Titanic disaster. The second most significant feature, 'Pclass' (0.651570), emphasizes how socioeconomic status shaped survival odds, possibly through access to lifeboats and other resources. 'Age'

(0.185353) and 'SibSp' (0.178909) follow, with age suggesting vulnerability and 'SibSp' pointing to family dynamics that likely affected survival. Although 'Embarked' (0.161603) was the least influential, it still added to prediction accuracy. Overall, survival probabilities were influenced by class, gender, age, family size, and fare, with higher-class passengers and women having better odds, and younger individuals and smaller families also advantaged. Despite the model's high accuracy, these factors' complex interactions underscore the unpredictable role of chance in survival outcomes.

```
Feature Importance:
    feature  importance
1       Sex    0.971953
0    Pclass    0.651570
2       Age    0.185353
3     SibSp    0.178909
4  Embarked    0.161603
```

1.6.4 Conclusion

- Gender was the strongest predictor of survival
- Passenger class significantly influenced survival chances
- The model showed good generalization between validation and test sets
- L1 regularization provided the best performance among tested configurations

2 Task 2: Cat and Dog Image Classification

2.1 Introduction

In this report, we investigate the development of several machine learning models aimed at classifying images as either containing a cat or a dog, while also assessing their accuracy [3].

2.2 Task 1:Data Preprocessing and EDA

We worked with two compact and evenly distributed datasets for our image classification project: 2,000 images for training and 1,000 for testing (See Fig. 3). Both sets were organized into "cat" and "dog" subdirectories. This even distribution is crucial for preventing bias in our model [3].

(a) Cats

(b) Dogs

Figure 3: Cats and Dogs Samples

```
    # Import the data we were given
train_directory = pathlib.Path("dataset/train_data_small")
test_directory = pathlib.Path("dataset/test_data_small")
import os
import shutil
# move the data per class
def organize_dataset(base_dir):
    # Paths
    cat_dir = os.path.join(base_dir, 'cat')
    dog_dir = os.path.join(base_dir, 'dog')


    os.makedirs(cat_dir, exist_ok=True)
    os.makedirs(dog_dir, exist_ok=True)


    for filename in os.listdir(base_dir):
        file_path = os.path.join(base_dir, filename)
        #skip directories
        if os.path.isdir(file_path):
            continue
        #we access the images which start with cat or dog
        if filename.lower().startswith('cat'):
            shutil.move(file_path, os.path.join(cat_dir,
   filename))
        elif filename.lower().startswith('dog'):
            shutil.move(file_path, os.path.join(dog_dir,
   filename))

    print(f"Data successfully organized in '{base_dir}'.")


organize_dataset('dataset/train_data_small')
And the same applied for the train dataset
```

After organizing the data, image pixel values were normalized to [0, 1] to improve training stability. We employed data augmentation including flipping, shearing, and

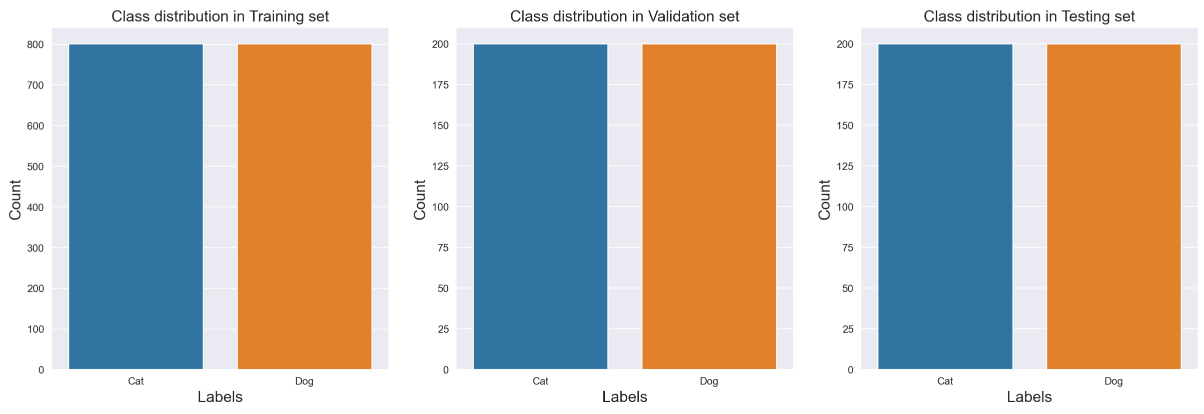


Figure 4: Distribution of Cats and Dogs after split

zooming with 'reflect' fill mode in the training dataset to aid the model's training. However, excessive augmentation negatively impacted accuracy, so we limited the augmentations to a carefully selected set for optimal results [6].

Images were resized to 128×128 pixels with 3 color channels (RGB) and processed in shuffled batches of 64. This batch size optimized memory usage and training speed, while shuffling preserved randomness across epochs to enhance model robustness..

Ultimately, our dataset was divided into 70% 20% and 10% for train, validation and test. (see Figure 4).

```
img_height, img_width = 128, 128
batch_size = 64
seed = 42

train_datagen = ImageDataGenerator(
    rescale=1.0/255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='reflect',
    validation_split=0.2
)


test_datagen = ImageDataGenerator(rescale=1.0/255)


train_generator = train_datagen.flow_from_directory(
    directory='dataset/train_data_small',
    target_size=(img_height, img_width),
    class_mode='binary',
    subset='training',
    shuffle=True,
    batch_size = batch_size,
    seed=seed
)

val_generator = train_datagen.flow_from_directory(
    directory='dataset/train_data_small',
    target_size=(img_height, img_width),
    class_mode='binary',
    subset='validation',
    shuffle=False,
    batch_size = batch_size,
    seed=seed
)

test_generator = test_datagen.flow_from_directory(
    directory='dataset/test_data_small',
    target_size=(img_height, img_width),
    class_mode='binary',
    batch_size = batch_size,
    shuffle=False,
    seed=seed
)

# Print the number of samples in each generator
print(f"Number of training samples: {train_generator.samples}")
print(f"Number of validation samples: {val_generator.samples}")
print(f"Number of testing samples: {test_generator.samples}")
```

2.3 Task 2: Is Convolutional Neural Network(CNN) the best answer?

CNNs excel at image classification through their layered architecture, where each layer automatically learns increasingly complex visual patterns. Their convolutional layers share parameters, reducing memory requirements and training time compared to fully connected networks. Pooling layers provide translation invariance, making CNNs robust to object position changes. The end-to-end learning capability eliminates the need for manual feature engineering. These properties make CNNs particularly effective for accurate binary image classification [1].

2.4 Task 3: Training Process

After normalizing and standardizing our dataset, we began training the said models, which consist of two different custom CNNs, a fine-tuned pre-trained CNN [5] MobileNet(Version2), a Support Vector Machine (SVM) [2], and a zero-shot classifier - Contrastive Language-Image Pre-Training(CLIP) [9].

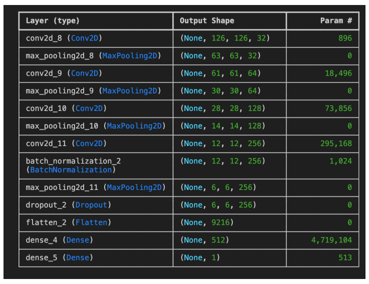
2.4.1 CNN Models

For all our CNN models, we used the ImageGenerators with our split data and applied the same overall training strategy which can be observed in Table 5.ImageGenerators are primarily used to generate batches of image data on the fly during model training. This allows for efficient memory usage and real-time data augmentation.

Table 5: Overview of Training Strategies

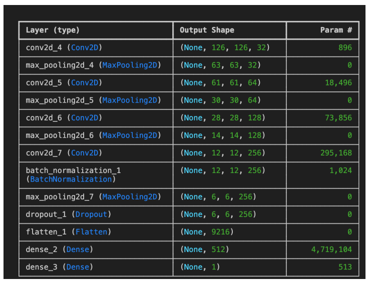
Technique	**Description**
Early Stopping	Halts training if validation loss does not improve for 5 epochs.
Learning Rate Reduction	Decreases learning rate by 0.2 after 3 epochs without validation loss improvement, with a minimum of 0.0001.
Model Checkpointing	Saves model weights to ‘customsimple.keras‘ on validation loss improvement.
Training Process	Trains for up to 100 epochs using training and validation generators with verbose output.

- Simple CNN The Simple CNN consists of 6 layers: 3 convolutional (with 32, 64, and 128 filters) and 3 pooling layers. With no batch normalization.
- Complex CNN The Complex CNN features 8 layers: 4 convolutional (with 32, 64, 128, and 256 filters), 4 pooling layers, 1 batch normalization, and 1 dropout layer.
- Pre-trained MobileNet V2(2018) The MobileNet V2 architecture consists of 5 layers designed for image classification on mobile devices, including a Functional Layer, an Average Pooling Layer, a Dense Layer, a Dropout Layer, and an Output Layer for binary classification.Notably the architecture condenses values well.



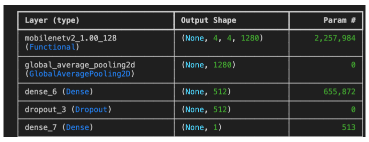
Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d_8 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_9 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_9 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_10 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_10 (MaxPooling2D)	(None, 14, 14, 128)	0
conv2d_11 (Conv2D)	(None, 12, 12, 256)	295,168
batch_normalization_2 (BatchNormalization)	(None, 12, 12, 256)	1,024
max_pooling2d_11 (MaxPooling2D)	(None, 6, 6, 256)	0
dropout_2 (Dropout)	(None, 6, 6, 256)	0
flatten_2 (Flatten)	(None, 9216)	0
dense_4 (Dense)	(None, 512)	4,719,104
dense_5 (Dense)	(None, 1)	513

(a) Simple Architecture



Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_5 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_6 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_6 (MaxPooling2D)	(None, 14, 14, 128)	0
conv2d_7 (Conv2D)	(None, 12, 12, 256)	295,168
batch_normalization_1 (BatchNormalization)	(None, 12, 12, 256)	1,024
max_pooling2d_7 (MaxPooling2D)	(None, 6, 6, 256)	0
dropout_1 (Dropout)	(None, 6, 6, 256)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_2 (Dense)	(None, 512)	4,719,104
dense_3 (Dense)	(None, 1)	513

(b) Complex Architecture



Layer (type)	Output Shape	Param #
mobilenetv2_1.00_128 (Functional)	(None, 4, 4, 1280)	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense_6 (Dense)	(None, 512)	655,872
dropout_3 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 1)	513

(c) MobileNet Architecture

Figure 5: Architectures of our CNNs

2.4.2 Other Models

SVMs excel at binary classification tasks by finding optimal decision boundaries, while CLIP learns joint representations of images and text, enabling zero-shot classification.

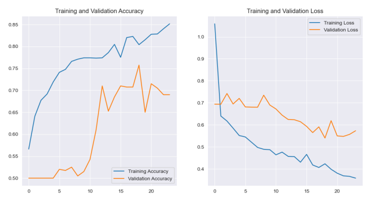
- SVM We have a train_image_svm function that processes image data from Keras generators into a flattened feature set, scales the features, trains a linear SVM classifier on the training set, and calibrates it using validation data to produce reliable probability scores for binary classification.

```
def train_image_svm(train_generator, val_generator,
   test_generator, max_iter=1000):
    """
    Train SVM classifier using image data from Keras
  generators.
    Optimized for 128x128 images with binary classification.
    """

    n_features = 128 * 128 * 3  # height * width * channels

    print("Processing training data...")

    n_train = train_generator.samples
    X_train = np.zeros((n_train, n_features))
    y_train = np.zeros(n_train)


    train_generator.reset()
    for i in tqdm(range(len(train_generator))):
        batch_x, batch_y = next(train_generator)
        start_idx = i * train_generator.batch_size
        end_idx = min((i + 1) * train_generator.batch_size,
  n_train)
        X_train[start_idx:end_idx] = batch_x.reshape(-1,
  n_features)
        y_train[start_idx:end_idx] = batch_y

    # Scale the features
    print("Scaling features...")
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)

    # Train SVM
    print("Training SVM...")
    svm = LinearSVC(
        max_iter=max_iter,
        dual='auto',
        random_state=42,

    )
    svm.fit(X_train, y_train)

    # Process validation data
    print("Processing validation data...")
    n_val = val_generator.samples
    X_val = np.zeros((n_val, n_features))
    y_val = np.zeros(n_val)

    val_generator.reset()
    for i in tqdm(range(len(val_generator))):
        batch_x, batch_y = next(val_generator)
        start_idx = i * val_generator.batch_size
        end_idx = min((i + 1) * val_generator.batch_size,
   n_val)
        X_val[start_idx:end_idx] = batch_x.reshape(-1,
   n_features)
        y_val[start_idx:end_idx] = batch_y

    X_val = scaler.transform(X_val)
```

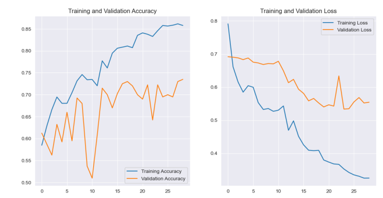
- CLIP The model was used for zero-shot image classification by processing images and text class labels, obtaining the model's output logits, and applying the softmax function to derive probabilities for each class; it iteratively classifies images from a test generator, storing predictions and true labels for evaluation.

```
def zero_shot_classify(image, class_names):
    inputs = processor(
        text=class_names,
        images=image,
        return_tensors="pt",
        padding=True
    )

    with torch.no_grad():
        outputs = model(**inputs)

    logits_per_image = outputs.logits_per_image
    probs = logits_per_image.softmax(dim=1)

    return probs[0].numpy()

predictions = []
true_labels = []
```

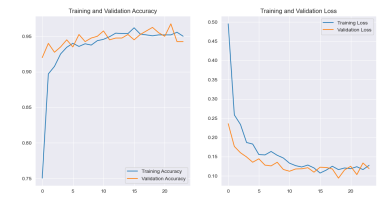
2.5 Task 4:Results of our methodologies



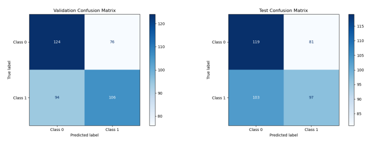
(a) Simple Architecture



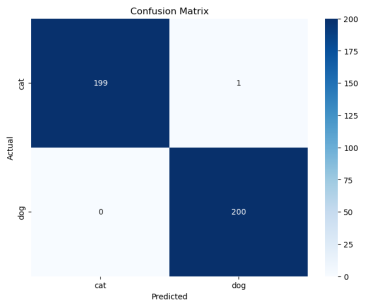
(b) Complex Architecture



(c) Pretrained Model



(d) SVM Model



(e) Clip Zero Shot

Figure 6: Results of our different methods

Method	**Test Acc.**	**Test Loss**	**Final Val Acc.**	**Final Val Loss**
Simple CNN	0.76	0.52	0.69	0.57
Complex CNN	0.72	0.53	0.74	0.55
Pretrained Model	0.97	0.10	0.94	0.12
SVM	0.54	-	0.57	-
CLIP(OPEN AI)	0.99	-	-	-

Table 6: Summary of results for the Classification models

CLIP demonstrated strong zero-shot learning with a 99% test accuracy, while the pre-trained model achieved 97% test accuracy and low test loss (0.10), indicating good generalization. The complex CNN, however, showed overfitting, with 86% training accuracy but only 72% test accuracy. Notably, the simple CNN outperformed the complex CNN on test accuracy (76% vs. 72%), showing that more complexity does not always improve results. The SVM had the lowest performance, with 54% test and 57% validation accuracy.The groundbreaking results achieved by CLIP have catapulted zero-shot learning into a new era of possibilities, however, CNNs are still used in both research and industry, and that models like MobileNetV2 could provide a balance between efficiency and accuracy for deployment in resource-constrained environments.

References

[1] Leiyu Chen et al. "Review of image classification algorithms based on convolutional neural networks". In: *Remote Sensing* 13.22 (2021), p. 4712.

[2] Vikramaditya Jakkula. "Tutorial on support vector machine (svm)". In: *School of EECS, Washington State University* 37.2.5 (2006), p. 3.

[3] Xiaofan Lin, Cong Zhao, and Wei Pan. “Towards accurate binary convolutional neural network”. In: *Advances in neural information processing systems* 30 (2017).

[4] Zuzana Reitermanova et al. “Data splitting”. In: *WDS*. Vol. 10. Matfyzpress Prague. 2010, pp. 31–36.

[5] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[6] Connor Shorten and Taghi M Khoshgoftaar. “A survey on image data augmentation for deep learning”. In: *Journal of big data* 6.1 (2019), pp. 1–48.

[7] Jason Smith. *Recursive Feature Elimination (RFE) for Feature Selection in Python*. 2023. URL: https://machinelearningmastery.com/rfe-feature-selection-in-python/ (visited on 10/30/2024).

[8] Giovanni Tripepi et al. “Linear and logistic regression analysis”. In: *Kidney international* 73.7 (2008), pp. 806–810.

[9] Weijie Tu, Weijian Deng, and Tom Gedeon. “A closer look at the robustness of contrastive language-image pre-training (clip)”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[10] Analytics Vidhya. *11 Important Model Evaluation Error Metrics*. Analytics Vidhya. 2019. URL: https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/ (visited on 10/30/2024).