

Applied Machine Learning

NB:

All group members have participated equally to this project, and all group members take full responsibility for the full project

Indhold

Problem 1	4
Data preparation / preprocessing (Ch).....	4
Question 1 (Ch)	4
Question 1 – SVM (Shallow Learners) (Ch)	5
Random Forest	8
Gradient Boosting	9
Combining models.....	11
The role of Image resolution.....	12
Question 2 (Th)	12
1. Briefly explain why convolutional neural networks (CNN's) are well-suited for skin lesion classification based on images? (Th).....	12
2. Train one or more CNN architectures to classify the skin lesions. (Th).....	13
3. Evaluate the performance for CNN and Shallow learners (Th & Ch).....	18
Question 3 (Th)	19
1. Visualize activation maps (e.g., Grad-CAM) for your preferred CNN model and for your selected skin lesion class.(Th)	19
2. Discuss whether these maps help identify regions of interest corresponding to your selected skin lesion classes. (Th).....	19
Problem 2	19
Question 1(C).....	19
Question 2 (C)	21
Model development	21
Data preparation.....	22
Training process	24
Performance evaluation and final RNN model	26
Insights and analysis	27
Question 3 (C)	28
Question 4 (C & Ch)	30
References.....	31
Appendix.....	33
1. Architecture for ResNet-inspired model	33
2. Architecture for EfficientNet-inspired model.....	34
3. Model summary	35
4. Model Architecture for transfer learning model.....	36

4.	Structure and scope of the data in problem 2	37
5.	Text length	37
6.	Emotion distribution in training, test and validation.....	40
7.	Word frequency – Dair AI	41
8.	Word frequency – Bluesky	43
9.	Frequency of 2500 most used words.....	43
10.	Base model accuracy and loss	44
11.	Model 2 accuracy and loss	45
12.	Model 3 accuracy and loss	47
13.	Key metrics model 1 and model 2	48

Problem 1

Data preparation / preprocessing (Ch)

The code can be found in the file 'loading_data_problem1_tf- Shallow Learners'

We started by running a simple CNN and SVM model on the full data. This gave us a starting point we then had to improve. Knowing that there was the possibility that the data contained duplicates or non-sense data/errors we used the code in file to remove duplicates. The removal of duplicates was done by creating a hash value for each picture in the entire dataset, both for training, validation and testing. When the hash values are unique the images stays and whenever 2 pictures have the same hash values one is removed.

Question 1 (Ch)

When dealing with skin lesion classification, classical machine learning methods such as support vector machines (SVMs), random forests (RFs), and gradient boosting (GB) are still viable even though deep learning models have gained more prominence. These classical shallow machine learnings techniques are viable because they don't require as much computational resources.

Support Vector machines (SVM):

SVMs are a powerful machine learning algorithm that are used for regression, outlier detection as well as linear and nonlinear classification. They are very adaptable, which makes them suitable for various applications such as text classification, image classification (*GeeksforGeeks. 2024, October 10*). When working with SVM for skin lesion classification, it requires feature extraction, because the raw pixels can become too complex for traditional classifiers.

Random forest (RF):

For skin lesion RF relies on features such as color, texture, and shape descriptors. The FR model aggregates predictions from multiple trees, making the model robust to imbalanced dataset and noise. The RF model can be effective, however its performance relies heavily on the quality of feature extraction, which makes is less scalable then a deep learning model such as CNN. In general, random forest provides very good predictions even with large datasets however it can become computational expensive with a large number of trees. (*GeeksforGeeks. (2025, January 16)*).

Gradient Boosting:

Gradient boosting is also a powerful tool for boosting the algorithm to combine several weak learners into strong learners (*GeeksforGeeks. (2023, March 31)*). When using gradient boosting for skin lesion it can effectively handle imbalanced datasets and non-linear relationships. The model is learning from the previous errors, therefore gradient boosting can provide high accuracy, but a careful hyperparameter tuning is needed.

Question 1 – SVM (Shallow Learners) (Ch)

The code can be found in the file 'loading_data_problem1_tf- Shallow Learners'

In this study, a Support Vector Machine (SVM) was used for the multiclass classification of 14 distinct skin lesion types. While SVMs are traditionally designed for binary classification problems, they can be effectively extended to handle multiclass classification tasks. A "simple" SVM model was provided as a benchmark for comparison. The task involved optimizing the model through parameter tuning, implementing functions, and addressing noise within the dataset to improve its performance. The baseline model achieved an accuracy of approximately 62.9% on the training set but demonstrated a reduced accuracy of only 57% on the test set, as seen in figure 1.

Performance Metrics:					

Training Set:					
Accuracy: 0.6295					
	precision	recall	f1-score	support	
0	0.00	0.00	0.00	382	
1	0.44	0.44	0.44	1473	
2	0.89	0.03	0.05	1161	
3	0.99	0.21	0.35	486	
4	0.95	0.45	0.61	427	
5	0.00	0.00	0.00	107	
6	0.90	0.80	0.85	1069	
7	0.96	0.72	0.82	747	
8	0.92	0.42	0.58	360	
9	0.59	0.93	0.73	5559	
10	0.74	0.33	0.45	1959	
11	0.59	0.91	0.72	1874	
12	0.00	0.00	0.00	280	
13	0.00	0.00	0.00	116	
...					
accuracy			0.57	3674	
macro avg	0.52	0.30	0.32	3674	
weighted avg	0.59	0.57	0.50	3674	

Figure 1 Basic Model

After the initial run, we applied Principal Component Analysis (PCA) to reduce the dataset's dimensionality, while preserving 99% of the variance. PCA helps convert high-dimensional data into a smaller feature set, while maintaining the essential patterns. Initially, the dataset contained 1024 features, which led to long runtimes and potential overfitting. As shown in Figure 2, we successfully reduced the feature set by 588 dimensions, resulting in a final dataset with 436 features

Original number of features: 1024
Reduced number of features: 588

Figure 2 Number of features

The reduction made the runtime significantly lower, as we can see from figure 9, this only didn't improve the runtime but also the accuracy. When running the model, after the dataset was reduced, the accuracy increased to 60%

Training Set:				
Accuracy: 0.6460				
	precision	recall	f1-score	support
0	0.71	0.01	0.01	693
1	0.43	0.44	0.44	2658
2	0.83	0.03	0.07	2098
3	0.93	0.33	0.49	900
4	0.98	0.55	0.70	792
5	0.00	0.00	0.00	191
6	0.92	0.83	0.87	1931
7	0.95	0.76	0.84	1367
8	0.91	0.54	0.68	660
9	0.60	0.94	0.73	10294
10	0.74	0.34	0.46	3617
11	0.64	0.92	0.75	3407
12	0.00	0.00	0.00	502
13	0.00	0.00	0.00	202
...				
accuracy			0.60	3674
macro avg	0.51	0.35	0.37	3674
weighted avg	0.60	0.60	0.54	3674

Figure 3 - 60% Accuracy

We now have a SVM model, which is slightly enhanced from the basic one in the beginning. The next thing that was done was a hyperparameter tuning also known as a grid search. The grid search was used to find the optimal C, optimal kernel and the optimal decision function. It is important to find the optimal C because it helps controls the trade-off between a low error on the training set and maintaining a decision boundary that generalizes well on unseen data. If C is too large the model risks overfitting on the training data, an perform poorly on validation and test set, however if C is small the model risks of underfitting. The choosing of the right kernel is also an importing feature, and the wrong kernel can have a high impact on the model performance. We have 3 different kernels, the linear which is good for linear separatable data, the polynomial captures more complex data unlike linear patterns and lastly, we have the Radial basis Function (RBF), which is very good at capturing complex nonlinear relationships. Lastly in our gridsearch, we have the decisions functions "ovo" and "ovr", which mean one-vs-one (ovo) and one-vs-rest (ovr). The ovo decision function trains one classifier per class against all other classes, whereas ovr trains one classifier for every pair of class and is fitted against all other classes.

```
kernels = ["linear", "poly", "rbf"] |
Cs = [0.1, 1, 5]
decision_functions = ["ovr", "ovo"]
```

Figure 4 - Parameter Grid

In figure 4 we can see the parameter grid we conducted our searched on, and the output was:

“*Kernel = RBF*” – “*C = 5*” – “*decision function = ovr*”.

These variables are then put into the basic SVM model, and we see a slight increase in the accuracy to 63%. However, we do see the overfits slightly more with a training accuracy on 76%, this could indicate that C is too high.

```

-----
Training Set:
Accuracy: 0.7661

```

	precision	recall	f1-score	support
0	0.78	0.19	0.31	685
1	0.54	0.59	0.56	2654
2	0.88	0.23	0.37	2089
3	0.95	0.80	0.87	832
4	1.00	0.88	0.93	744
5	1.00	0.07	0.14	191
6	1.00	0.97	0.98	1812
7	1.00	0.95	0.97	1281
8	0.97	0.87	0.92	612
9	0.68	0.97	0.80	10288
10	0.92	0.51	0.66	3612
11	0.88	0.96	0.92	3150
12	0.97	0.15	0.26	502
13	1.00	0.08	0.15	202
...				
accuracy			0.63	3663
macro avg	0.50	0.42	0.43	3663
weighted avg	0.60	0.63	0.58	3663

Figure 5 - Best parameter result

When looking at the output from the model with the optimal parameters we see that there is a significant weight imbalance which can cause a decrease in accuracy. The reason being is that if one class dominates the dataset a naïve classifier could predict the majority classes most of the time and still achieve a high accuracy, however it would then fail to classify the minority classes. In figure 5, we see that the recall predictions for some of the classes are low, among them class 0 and 13. To address this issue, we applied “*class_weight='balanced'*”, which automatically adjusts the class weights inversely proportional to their frequency in the dataset. This method ensures that the model pays more attention to minority classes (low support) while reducing the dominance of majority classes, ultimately leading to a more balanced classification performance.

```

Performance Metrics:
-----
Training Set:
Accuracy: 0.7729

```

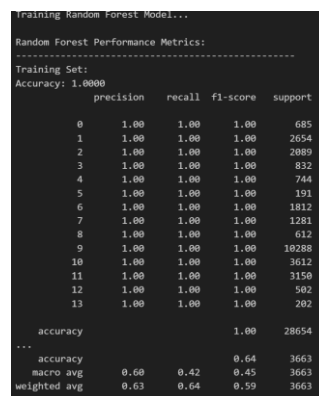
	precision	recall	f1-score	support
0	0.44	0.92	0.60	685
1	0.58	0.71	0.64	2653
2	0.51	0.65	0.57	2089
3	0.85	0.99	0.91	832
4	0.93	0.99	0.96	744
5	0.38	1.00	0.55	191
6	0.99	0.99	0.99	1813
7	0.99	0.99	0.99	1282
8	0.91	0.99	0.95	613
9	0.88	0.68	0.77	10286
10	0.80	0.61	0.69	3610
11	0.94	0.95	0.95	3151
12	0.57	0.96	0.71	502
13	0.36	0.99	0.53	202
...				
accuracy			0.57	3663
macro avg	0.47	0.50	0.48	3663
weighted avg	0.62	0.57	0.59	3663

Figure 6 - Class imbalance

As seen from figure 6 after adjusting and implementing the “*class_weight='balanced'*”, we see that the recall for each class has now enhanced compared to the base model in figure 1. However, we see that the accuracy of the training set increased but the accuracy on the test set decreases back to 57%. The *Class_weight=imbalanced*, gets a lower accuracy because it forces the model to predict the minority classes more, and that comes at a cost which is creating false positives for the majority classes. It improves overall recall and fairness but reduces the accuracy. Therefore, the best SVM model remains the one obtained after hyperparameter tuning, where a better balance between recall and accuracy was achieved.

Random Forest

Random Forest (RF) Was also one of the methods we had to try and implement, and in figure 6 we see the output from a simple RF model.



```

Training Random Forest Model...

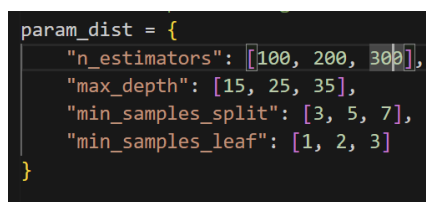
Random Forest Performance Metrics:
-----
Training Set:
Accuracy: 1.0000

```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	685
1	1.00	1.00	1.00	2654
2	1.00	1.00	1.00	2089
3	1.00	1.00	1.00	832
4	1.00	1.00	1.00	744
5	1.00	1.00	1.00	191
6	1.00	1.00	1.00	1812
7	1.00	1.00	1.00	1281
8	1.00	1.00	1.00	612
9	1.00	1.00	1.00	18288
10	1.00	1.00	1.00	3612
11	1.00	1.00	1.00	3150
12	1.00	1.00	1.00	502
13	1.00	1.00	1.00	202
accuracy			1.00	28654
...				
accuracy			0.64	3663
macro avg	0.60	0.42	0.45	3663
weighted avg	0.63	0.64	0.59	3663

Figure 7 Random forest basic model

We clearly see a tendency for overfitting when running a simple Random Forest model. Because of the overfitting and the accuracy being 64%, we will perform a grid search, like we did for the SVM.



```

param_dist = {
    "n_estimators": [100, 200, 300],
    "max_depth": [15, 25, 35],
    "min_samples_split": [3, 5, 7],
    "min_samples_leaf": [1, 2, 3]
}

```

Figure 8 - Hypertuning Random forest

As we can see from Figure 8 it illustrates the grid search for tuning the hyperparameters of our model.

n_estimators defines the number of trees in the ensemble. A higher number of trees can improve performance but increases training time and the risk of overfitting.

Max_depth controls the depth of each decision tree. Deeper trees capture more patterns but are prone to overfitting, while shallower trees are simpler and generalize better.

Min_samples_split specifies the minimum number of samples required to split a node. We allow splits when there are at least 2, 5, or 7 samples, which helps control the tree's growth.

Min_samples_leaf sets the minimum number of samples required for a leaf node. A leaf node is the final node where no further splitting occurs, representing the model's final prediction. Higher values lead to simpler trees and reduce overfitting. By fine-tuning these parameters, we aim to balance model complexity and generalization to achieve the best possible performance.

Accuracy: 0.9971				
	precision	recall	f1-score	support
0	1.00	1.00	1.00	685
1	0.98	1.00	0.99	2653
2	1.00	1.00	1.00	2888
3	1.00	1.00	1.00	832
4	1.00	1.00	1.00	742
5	1.00	1.00	1.00	191
6	1.00	1.00	1.00	1813
7	1.00	1.00	1.00	1282
8	1.00	1.00	1.00	613
9	1.00	0.99	1.00	10288
10	1.00	1.00	1.00	3611
11	0.99	1.00	1.00	3152
12	1.00	1.00	1.00	502
13	1.00	1.00	1.00	201
...				
accuracy			1.00	28653
macro avg	1.00	1.00	1.00	28653
weighted avg	1.00	1.00	1.00	28653
...				
accuracy			0.63	3663
macro avg	0.56	0.43	0.45	3663
weighted avg	0.62	0.63	0.59	3663

Figure 9 Random forest

We see from figure 9 that we have reduced the overfitting marginally, however this has also resulted in a decreased performance. The reason for this drop was that the original model was very much overfitting. Whenever we tried to perform a more larger grid search, with more parameters to put into the model, we only saw a small decline training accuracy, which still indicates overfitting. Because of this and our limited computational power it didn't managed us to perform a grid search large enough to improve the accuracy and reduce the overfitting significantly.

Gradient Boosting

Gradient boosting was also one of the methods we tried using on our dataset to see how accurate it could classify the different classes. In the lectures we were presented with different boosting methods, such as XGBoost and LGBM. When running a simple boosting method, XGBoosting provided the best accuracy on the test set with 65% accuracy. We also tried doing a simple LGBM model however it only had 63% accuracy. We therefore decided to explore xgboost further.

XGBoost Model Performance:				
	precision	recall	f1-score	support
0	0.30	0.03	0.06	88
1	0.44	0.54	0.49	333
2	0.36	0.14	0.20	262
3	0.83	0.53	0.65	111
4	0.94	0.65	0.77	99
5	0.00	0.00	0.00	25
6	0.84	0.74	0.79	239
7	0.91	0.73	0.81	170
8	0.88	0.54	0.67	81
9	0.67	0.90	0.77	1288
10	0.52	0.36	0.43	453
11	0.68	0.90	0.77	424
12	1.00	0.03	0.06	64
13	0.00	0.00	0.00	26
accuracy				0.65 3663
macro avg				0.60 0.44 0.46 3663
weighted avg				0.64 0.65 0.62 3663

Figure 10 - XGBOOST Base

After the initial run we again did a hyperparameter tuning to find the best parameters for optimizing our XGBoost classifier. For this parameter search we used a library called optuna. This Library uses the Bayesian optimization method, so it efficiently searches for the best hyper parameters that maximize a given metric, which in our case is the accuracy.

```
def objective(trial):
    params = {
        "n_estimators": trial.suggest_int("n_estimators", 100, 700), |
        "max_depth": trial.suggest_int("max_depth", 3, 30),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.2),
        "subsample": trial.suggest_float("subsample", 0.6, 1.0),
        "colsample_bytree": trial.suggest_float("colsample_bytree", 0.6, 1.0),
        "gamma": trial.suggest_float("gamma", 0, 0.3),
    }
```

Figure 11 Optuna Parameters

In figure 8 we see the parameters that the model searches within. For example, in `n_estimators` it isn't hardcoded to use 100 or 700 estimators, but the model can fluctuate between this interval. The model does this for every parameter within the interval. The optuna library uses this search space, to test different hyperparameter combinations during the optimization process. In the first run the optuna library selects random numbers, however as the trials progresses the model focusses on areas in the search space that have provided the best accuracy on the validation set. We have set the model to only return 100 trials, and for each trail it estimates a new set of parameters.

```
final_model = xgb.XGBClassifier(
    n_estimators=670,
    max_depth=44,
    learning_rate=0.04303725372313019,
    subsample=0.6029115855774464,
    colsample_bytree=0.6000667872069467,
    gamma=0.020130953938987152,
    objective="multi:softmax",
    eval_metric="mlogloss",
    tree_method="gpu_hist",
    enable_categorical=False,
    random_state=42
)
```

Figure 12 – Best Parameters

In figure 9 we can see the best parameters returned in the optuna model, when inserting these new hyper tuned parameters into our base model we see that the accuracy increases from 65% to 67% accuracy on the test dataset, with these new parameters. The gradient boosting strategy has therefore provided us with the best results so far.

Combining models

We have now implemented three classical machine learning methods for the multi-class classification problem. Each model began with a base version, with SVM being provided as a starting point. After the initial runs, we performed hyperparameter tuning using grid search to optimize each model. Below we can see the different model with the best test accuracys.

SVM	Gradient booster	Random forest	Stacked Model
63%	67%	64%	66%

Among the three, XGBoost showed the highest improvement and achieved the best accuracy. Based on this, we explored a stacking approach combining SVM and XGBoost to try and further enhance performance.

```
Stacking Performance:
-----
Training Set:
Accuracy: 0.9860

```

	precision	recall	f1-score	support
0	0.90	0.99	0.95	548
1	0.93	1.00	0.96	2122
2	1.00	1.00	1.00	1670
3	1.00	1.00	1.00	666
4	1.00	1.00	1.00	594
5	0.00	0.00	0.00	153
6	1.00	1.00	1.00	1450
7	1.00	1.00	1.00	1026
8	1.00	1.00	1.00	490
9	1.00	1.00	1.00	8230
10	1.00	1.00	1.00	2889
11	0.99	1.00	1.00	2521
12	0.83	1.00	0.91	462
13	0.00	0.00	0.00	161
accuracy			0.99	22922
macro avg	0.83	0.86	0.84	22922
weighted avg	0.97	0.99	0.98	22922
...				
accuracy			0.66	2930
macro avg	0.56	0.46	0.48	2930
weighted avg	0.64	0.66	0.63	2930

Figure 13 Best Model (SVM & XGBoost)

As seen in Figure 12, stacking SVM and Gradient Boosting achieved 66% accuracy, slightly lower than the best standalone Gradient Boosting model by 1%. However, the training accuracy remains extremely high at 98.6%, indicating significant overfitting. This is likely due to using only 20% of the dataset, constrained by computational limitations. Additionally, the high `n_estimators` and `max_depth` values further contribute to overfitting. To improve generalization, the model's complexity should be reduced by further optimizing the hyperparameters and utilizing a larger dataset.

Here we can see the differences shallow learners, and the model's best accuracy. It is here possible to see that the gradient boosting method was the best overall model even compared to the stacking model.

SVM	Gradient booster	Random forest	Stacked Model
63%	67%	64%	66%

The role of Image resolution

Image resolution play a crucial role, when trying to classify images and when it comes to multi class classification. For example, in our case with skin lesions because it can impact the quality of features that a machine learning model can extract. For higher resolution images will contain more details such as color gradients, patterns and texture variations however the training time and memory requirements also increase significantly due to the high resolution. Deep learning models like the CNN we used can handle higher resolution, however more classical machine learning method like SVM is not as great.

For the classical machine learning tools like the ones, we have used, if we increased the images, one of the risk could be increased training time and the risk of overfitting therefor in our assignment we choose to keep the resolution of 224x224, both for the CNN and the SVM, RF and XGBoost.

Question 2 (Th)

The code for this part can be found in the file "CNN_DS807_1.ipynb"

1. Briefly explain why convolutional neural networks (CNN's) are well-suited for skin lesion classification based on images? (Th)

CNN are well suited for image classification due to its architectural design and the way they process information. It is comprised of 3 main types of layers used in this order, convolutional layers, pooling layers and a fully connected layer (IBM, n.d.). The convolutional layer will learn smaller local patterns and the next convolutional layer will learn larger patterns based on the features of the first layer. This enables the model to learn spatial hierarchies in the data. The purpose of the pooling layers is to reduce the number of parameters in the input. This does result in a loss of information, but by reducing the complexity it makes up for it by improving efficiency and limit the risk of overfitting. By using convolutional layers and pooling layers the model achieves translation invariance, meaning that it can recognize patterns anywhere after it has learned it. This gives it a big advantage compared to more densely connected models like FNN, which would have to learn the same pattern again if it

appeared in a different place. At the end the fully connected layer performs the classification often using a SoftMax activation function, which produces a probability from 0 to 1 (Chollet, 2017).

2. Train one or more CNN architectures to classify the skin lesions. (Th)

a. Discuss and implement alternative CNN architectures Get inspired, for example, by the ResNet and EfficientNet architectures.

The simple CNN provided in “*loading_data_problem1*” was used as a baseline model. We added early stopping to this model and ran it for 10 epochs. It provided an accuracy of 35% when performed on the dataset with duplicates removed. Since this model is very simple, it would need to be modified to perform better.

We decided to expand this model by increasing the number of convolutional layers and adding MaxPooling after each convolution. When running the model with 10 epochs and adding *Early stopping*, it yielded an accuracy of 60% on the test set.

The ResNet (Residual Network) is a CNN Architecture that introduces residual connections, also called skip-connections to solve the issue of vanishing gradients often encountered in deep networks. The skip connections allow gradients to flow directly through shortcut connections, which enables the training of deeper networks. When implementing a basic architecture inspired by ResNet onto the baseline model, we achieved a validation accuracy of around 52% keeping the same learning rate and optimizer as the other model. The architecture of this model can be seen in Appendix 1.

Another Architecture introduced during the course is the EfficientNet, which is an architecture that scales depth, width, and input resolution uniformly. This method balances the trade-offs between network depth, width, and resolution. When we implemented a model that was inspired by the EfficientNet architecture (see appendix 2) we obtain an accuracy of around 48%.

We chose to continue with the extension of the provided model, since this yielded the highest accuracy while being the simplest to adjust. The performance of the model is shown in figure 14.

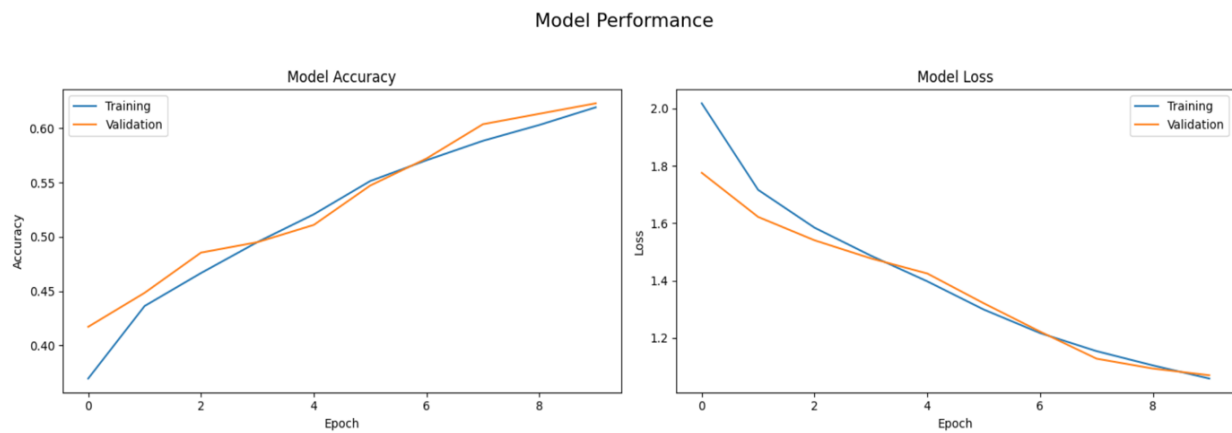


Figure 14: Model Performance of the extended base model after 10 epoch.

b. Experiment with different optimizers (e.g., SGD, Adam) and visualize their performance.

To evaluate the impact of different optimization algorithms, we tested Stochastic Gradient Descent (SGD), Adam, and RMSprop on our model. We first considered keeping the same learning rate for all three optimizers, but this seemed unfair as SGD applies the same learning rate to all parameters without adaptation, which makes it sensitive to the choice of learning rate. Therefore, we decided to compare the optimizers using different learning rates:

- SDG had Learning rate = 0.01.
- Adam had Learning rate = 0.0001.
- RMSprop had Learning rate = 0.001.

The performance of each is visualized in figure 15 showing that SDG performed best having an accuracy of 65%.

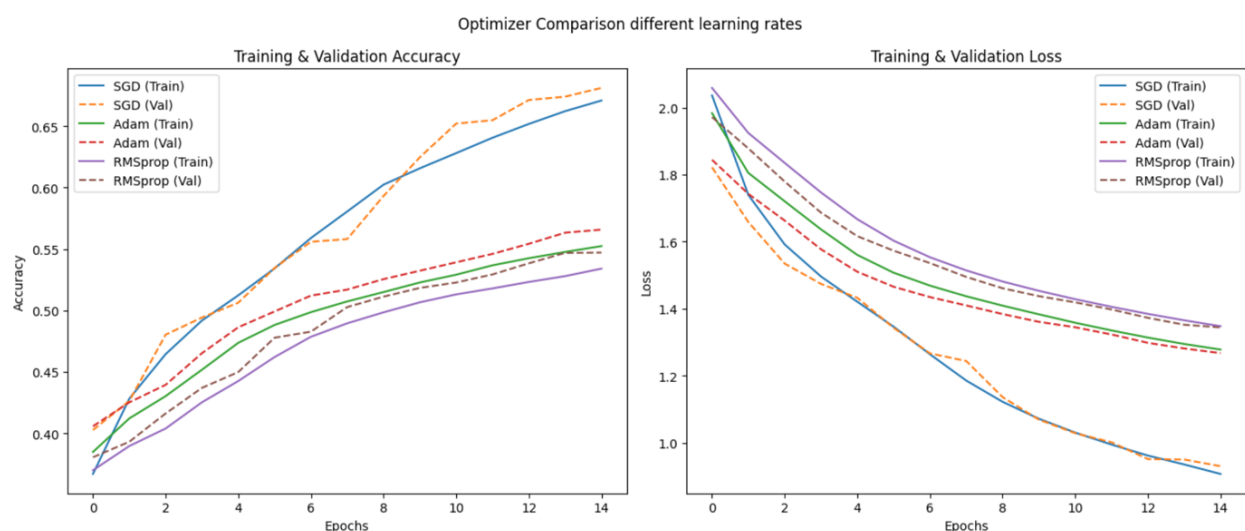


Figure 15: Optimizer comparison with different learning rates.

c. Compare the impact of regularization methods like dropout, weight decay, and early stopping on training and validation losses. Discuss regularization and its relation to overfitting and provide visualizations.

When training models over many epochs, there is a risk of the model starting to memorize the training data instead of learning how to generalize. The issue is called overfitting, and it can be observed when the training accuracy continues to rise while the validation accuracy either stagnates or declines, indicating that the model is no longer learning meaningful patterns but instead memorizing the training data.

Regularization methods can be applied as a tool to stop the model from overfitting by making it harder to train, ensuring that the model generalizes well to unseen data.

Our model already had Early stopping before, as it interrupts the training when overfitting starts to occur, saving the model before it starts to memorize. The model also had dropout in the last convolutional layer.

To compare the impact of the different regularization methods, we trained three models, each using one of the methods: Dropout, Weight Decay (L2 Regularization), and a combination of both Dropout and Weight Decay. Additionally, all models incorporated early stopping to stop training when validation performance stagnated or decreased.

The results, visualized in figure 16, indicates that the model using Weight Decay alone outperformed the others, achieving the best balance between training and validation accuracy.

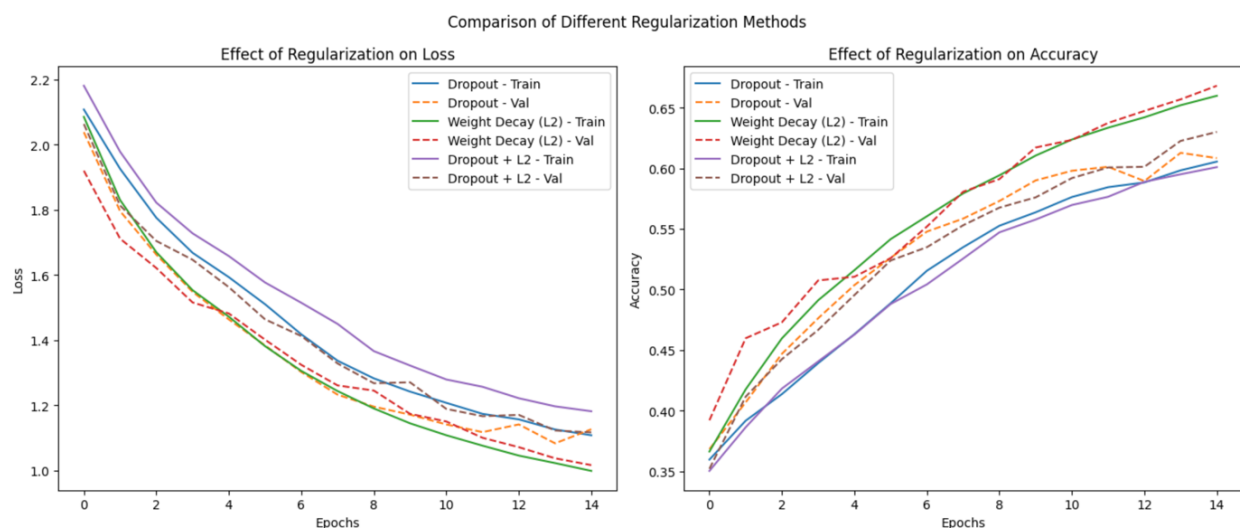


Figure 16: Comparison of Regularization methods.

d. Explore data augmentation techniques and their effect on overfitting. Visualize the training process.

For Data Augmentation we decided to test our model using random zoom, rotate and flip for our images, since these are the typical augmentation methods used. Furthermore, we decided to try and implement a small amount of augmentation for brightness and contrast as these potentially could have a bigger impact on the training since a lot of the images are close ups of the skin. The overall goal of the data augmentation was to create a diverse set of training images while maintaining the original semantic information.

When testing our model on both augmented and non-augmented training data, we saw an increase of training loss when using augmented data and a small decrease in validation loss. This indicated that the augmentation had a small impact on the model during training. Figure 17 illustrates the performance of both model during the training process. This suggest that our data augmentation acts as a form of regularization in making the model more robust to unseen data. The trade-off here is a much slower learning curve but the benefits in validation accuracy and stability make it a worthwhile technique.

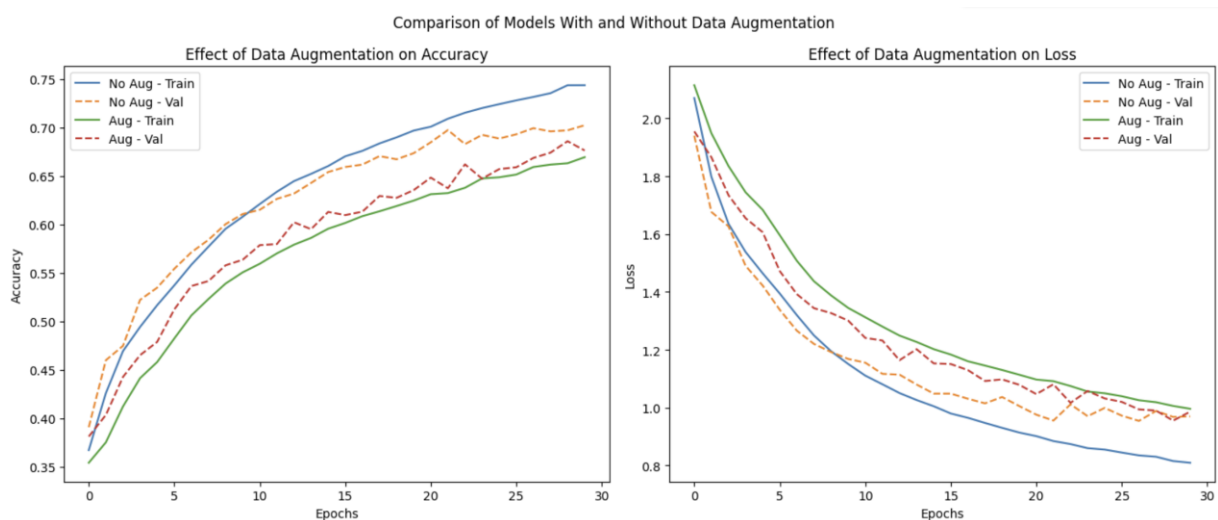


Figure 17 Augmented vs non-augmented data

e. Apply transfer learning, detailing your approach and any required adjustments to the data. Here, be sure to visualize plots of train and validation losses and accuracies.

For transfer learning we decided to try both Resnet50V2 and EfficientNet pretrained models. With weights from imagenet we compared the 2 model on performance accuracy. Restnet50v2 performed significantly better than EfficientNet. Restnet50v2 consists of 50 convolutional layers, whereas efficient net only has 24 layers. The Layer depth allows the Restnet50v2 to have a larger receptive field, which mean it can capture subtle variations in lesion morphology, shape, and texture.

Our approach to this was to make sure the dataset where robust to generalization all the images was resized to 224x224 pixels. The Data augmentation techniques was used to apply variability in lesion appearance. We here used horizontal flipping, slight rotations, zoom variation, brightness and contrast adjustments. When adding these augmentations, it will help the model ability to learn from diverse skin lesion images, while it will prevent overfitting to specific patterns in the dataset.

From figure 16 the accuracy curve demonstrates progressive learning, with both training and validation accuracy improving steadily. The validation accuracy stabilizes around 72–74%, indicating that the model generalizes well to unseen data without severe overfitting

The loss curve reveals that training loss decreases significantly in the early epochs, which is expected as the model learns key patterns. However, the validation loss flattens after epoch 4 and slightly fluctuates, indicating that the model maintains relatively stable performance, however overfitting could become an issue if training continues after this point.

As we can see from the figure 18 below the transfer learning method from restnet50v2 shows string classification performance for the isic skin lesion dataset. The model gives a test accuracy of 73,3%.

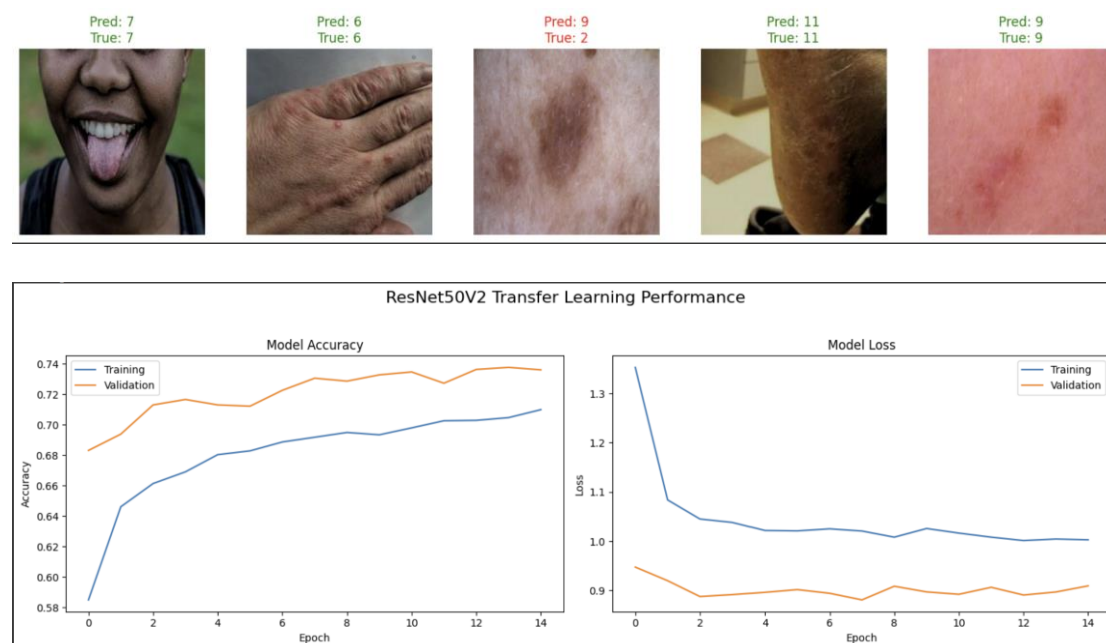


Figure 18 Transfer learning plots

f. Evaluate the importance of image resolution relative to model depth and width, referencing EfficientNet principles.

Scaling up convolutional neural networks is a strategy to increase model accuracy. Models like ResNet which we used in *task e*, have been scaled by increasing their depth. Other approaches have focused on width scaling or image resolution scaling. A higher image resolution could provide the model with visual details, that can improve feature extraction and classification performance. However, according to Tan and Le (2019) an increase in resolution needs adjustments in depth and width to fully use the added spatial information. EfficientNet introduces a compound scaling method, which addresses the limitations of single-dimension scaling by uniformly adjusting depth, width, and resolution with a set of fixed scaling coefficients (Tan & Le, 2019).

3. Evaluate the performance for CNN and Shallow learners (Th & Ch)

When comparing the CNN to a shallow learner it is important to say that deep neural networks have demonstrated a superior performance compared to the shallow learners. Especially in image classification, because they are extremely efficient in extraction features. We can also see from the models we have created, that it is the CNN model that has shown to be best. Below is a table that showcases the best model from our transfer learning model, where we used `resnet50v2` and the best model from our shallow learners which was the gradient boosting method, `XGBoost`.

Model	Type	Performance
CNN Transfer learning model	Deep Neural Network	73,5%
Gradient Boosting	Shallow Learner	67%

When looking at the table we can clearly see that our transfer learning model achieves by far the best accuracy. This showcases the CNN is better at capturing deep spatial features whereas the shallow learners struggle. In general the CNN will outperform the shallow learners due to the fact that it can automatically extract hierarchical features from images, whereas shallow models rely on handcrafted features. CNNs leverage spatial convolutions, enabling them to detect fine-grained variations in shape, color, and texture, whereas shallow learners cannot fully utilize pixel dependencies.

Question 3 (Th)

1. Visualize activation maps (e.g., Grad-CAM) for your preferred CNN model and for your selected skin lesion class.(Theis)

It was not possible for us to extract the last layer in the Transfer model, and we were therefore unable to create the activation maps for this model. To address the question, we decided to use our best performing model without transfer learning, when creating activation maps for class 9.

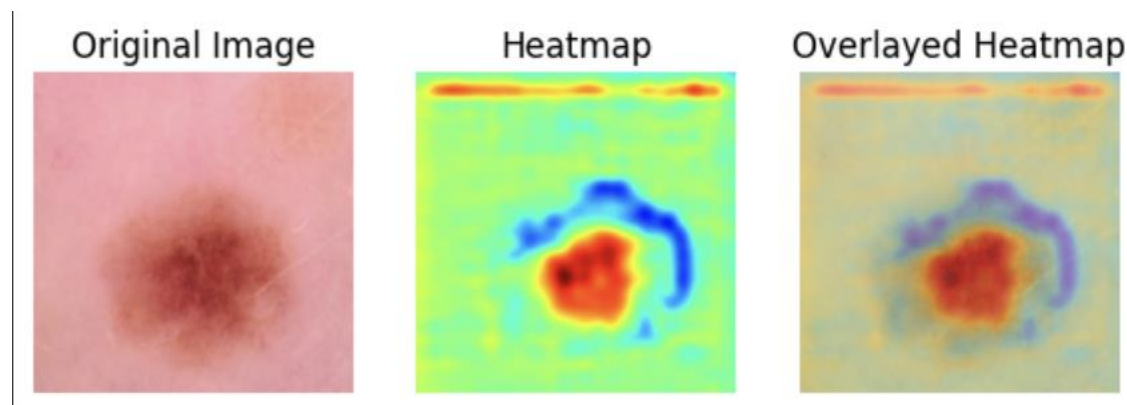


Figure 19: Activation map

2. Discuss whether these maps help identify regions of interest corresponding to your selected skin lesion classes. (Theis)

The Grad-CAM heatmap emphasizes clinically relevant features, such as the darkened, irregularly shaped spot. The high activation over the area, rather than the background suggests that the model is focusing on pathologically significant features. However, there is also some activation in areas with no pathology, such as at the top of the image in figure 19, which suggests that the model might be sensitive to noise.

The integration of convolutional neural networks (CNNs) in healthcare could have potential for assisting dermatologists in the identification and classification of skin lesions. However, given the importance of medical diagnosis, where errors could lead to life-threatening consequences, it is crucial that the model would perform perfectly.

Problem 2

Question 1(C)

The code for the EDA can be found in file “Anvendt maskin EDA.ipynb”.

RNN's are a class of neural networks, that are specifically good at handling sequential data. This is because they use information from prior input and, thanks to their looped architecture, combine it with information from the current input to generate the current output. There are different types of RNN's which include, standard RNN, BRNNs, LSTM, GRUs, and encoder-decoder RNNs. Some of these will be explored in this project (Chollet, 2017).

First, we conducted an EDA to see how the DAIR AI data was structured and to gain some insights. The EDA was performed in the file "Anvendt maskin ETA". The data has 6 categories and contains the columns "text", "label", and "label_name" and each data set have the following numbers of entries:

Data set:	Number of entries:
Training data	300102
Test data	41681
Validation data	75026
Total entries	416809

Table 1 Number of entries in the data sets

Looking at appendix 6 we can see that the distribution of the emotion categories is skewed, with "joy" representing around 34 % percent of the data while "surprise" is only represented around 3.6%. This imbalance could result in the model biasing towards the majority classes. We can however see that the class distribution is almost the same across our training, test and validation sets. This is good as it ensures consistency and makes the metrics in the trained and evaluated data comparable.

For analyzing the content of the textual entries, we have used wordclouds, which visualize the most frequently used words, as well as giving insight into main topics and potential stopwords that needs to be removed during the preprocessing of the data (appendix 7). From them we can see that the word "feel" and "feeling" are some of the most frequent words. We also notice that there are words or letters that we wish to remove because they only introduce noise. These include "don", "u", "ve", "t", "co", "href".

The frequency of texts and their lengths are shown in appendix 5. From the charts we can see that length of the texts vary, with most of the entries ranging from 20 to 170 characters where the frequency seems to peak at 50. The frequency distribution also looks similar across the data sets which is positive, because it means that the model will be evaluated on similar data to what it was trained on. Because there is a great range of the number of characters the texts contain, we will be looking into applying a maximum sequence length (truncation) and use padding for shorter sequences.

By looking at the number of entries, the split ratios are circa 70% for the training set, 17.5% for the validation set and 10% for the test set. This aligns with a typical split ratio, and therefore we won't change the split ratio.

Question 2 (C)

The code for this part can be found in the file "rnn preprocess and model.ipynb"

Model development

We used 3 models for this problem. Our basic model had the structure shown in figure 20 which is heavily inspired by a model from Christians exercises. To help find the optimal values for the hyperparameters embedding dimension, LSTM units and learning rate, we used a grid search with nested loops. Based on this learning rate was specified as 0.0001 and embedding dimension and LSTM units were 128 for all our models. As mentioned, we decided to put to vocabulary at 1500 and the max sequence length to 150.

```
#base model
embedding_dimension = 128
model3 = tf.keras.models.Sequential([
    tf.keras.layers.Embedding(input_dim=1501,
                              output_dim=embedding_dimension,
                              input_length=100,
                              name="embedding"),
    tf.keras.layers.LSTM(128),
    tf.keras.layers.Dense(6, activation='softmax')
])
```

Figure 20 Base RNN model

The base model is a sequential model using keras, and it is consisting of 3 layers. The first layer is the embedding layer. This layer converts the input words into vector representations. It has a vocabulary of 1501, where the +1 is to accommodate out-of-vocabulary words, which allows the model to assign an index to words not in the top 1500 most used words. For the development of the base model, we tried switching between different optimizers, Adam and RMSProp, as well as implementing a LSTM and GRU layer to see the difference in performance. This was to see which configuration performed best. The purpose of implementing LSTM and/or GRU was also to make the model better at capturing long-range dependencies.

After going through multiple versions of the base model we decided to make our model more complex by adding more layers by implementing a second LSTM layer and add dropout and L2

regularization. This was done to make the model more accurate and to prevent it from overfitting. We also added a bidirectional wrapper for the first LSTM layer. This was to try and capture more contextual information as the sequences will be processed both forwards and backwards. This model has 6 layers and can be seen in figure 21.

```
#Model 2 with added bidirectional(LSTM) and dropout
embedding_dimension = 128
model2 = tf.keras.models.Sequential([
    tf.keras.layers.Embedding(input_dim=1501,
                              output_dim=embedding_dimension,
                              input_length=150,
                              name="embedding"),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(128, return_sequences=True)), # Bidirectional LSTM
    tf.keras.layers.Dropout(0.2), # Dropout
    tf.keras.layers.LSTM(64), # Second LSTM layer
    tf.keras.layers.Dropout(0.2), # Dropout
    tf.keras.layers.Dense(6, activation='softmax', kernel_regularizer='l2') # L2 regularization
])
```

Figure 21 Model 2 and its architecture

The third model contains 7 layers in total. We added a GRU layer to process the output of the bidirectional layer in the hopes that it would allow the model to capture different temporal dependencies and thereby increase accuracy. We furthermore increased the vocabulary from 1500 to 2000 and max sequence length from 150 to 200. This model can be seen in figure 22.

```
embedding_dimension = 128
model3 = tf.keras.models.Sequential([
    tf.keras.layers.Embedding(input_dim=2001,
                              output_dim=embedding_dimension,
                              input_length=200,
                              name="embedding"),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(128, return_sequences=True)), # Bidirectional LSTM
    tf.keras.layers.Dropout(0.2), # Dropout
    tf.keras.layers.GRU(64, return_sequences=True), # Added GRU layer
    tf.keras.layers.LSTM(64), # Second LSTM layer
    tf.keras.layers.Dropout(0.2), # Dropout
    tf.keras.layers.Dense(6, activation='softmax', kernel_regularizer='l2') # L2 regularization
])
```

Figure 22 Model 3 with the added GRU layer and larger vocabulary and input length

Data preparation

To ensure that our RNN model had the best prerequisites for classifying our sequential data, we performed a series of preprocessing steps. This included cleaning, organizing and standardizing the data to improve performance and accuracy. We applied the following:

1. Duplicate removal

First, we searched the training data for duplicates. These were identified by going through each row, and if a row had identical values to a previous row, it was considered a duplicate. This method

identified 353 duplicates which were removed. We also searched the data sets for missing values, and we did not find any.

2. Data cleaning

We then cleaned the training, validation and test set using the nltk stopword library where we also added custom stopwords based on what could be seen in the wordclouds. We then converted all characters to lower case and used tokenization to make the text into individual tokens. Then we removed digits, punctuation and special characters, and single characters with space around if they were a part of the stopwords library. This is done to clean and normalize the data as well as reducing noise and make it easier for the model to focus on the important words.

3. Tokenization and Vocabulary

Tokenization with keras tokenizer was then used to convert the cleaned text into numerical sequences and create a vocabulary of the most frequent words used in the text. To decide the size of this vocabulary we have created a chart showing the frequency of the 2500 most used words in the texts (appendix 9). From this we can see that there is a quick decline in the frequency of words and the decline in frequency starts to plateau at around 1500 to 2000. Based on this we decided to start our model with a vocabulary of 1500 words and later ended up changing it to 2000 words.

4. Sequence length and padding/truncation

As mentioned, we used truncation to standardize the sequence lengths. This ensures that the model is only fed the sentences of the same length which motivates the training and computation time/efficiency. Based on the distribution of text lengths from the EDA we set the max sequence length to 150 at first. This could result in some loss of meaning in text longer than 150 but should be outweighed by the gain in computational efficiency. We later changed the max length to 200. Padding was also added to make sure that all the sequences had the same length.

5. Embedding dimension

Embeddings assign each word in the vocabulary a unique vector in the embedding space. The length of the embedding dimension decides the length of the vector and we use it to better capture the connections between words based on their meaning. It is important to choose the right embedding dimension because it can affect model performance and computational load. We used a grid search with nested loops to tune this parameter.

Training process

The training process we followed were to start with a base model and see how well it could perform. Then we would add more training parameters to, hopefully improve the performance of accuracy of the model. This resulted in 3 main models, with different versions. In all of the versions, we would keep the same number of layers and only the number of epochs, optimizer and layer type. For model 3 however we did end up changing the size of the vocabulary and max sequence length, to see if model would perform better. The batch size of 64 remained constant through all our models and versions.

Looking at figure 23, 25, and 26 we have listed the different versions of models, represented by an ID, of each model and specified their training parameters. The first version seems to have performed well considering its complexity and ran for the full 15 epoch. From figure 24 we can see the model accuracy and model loss.

Base model:

Version ID	Batch size	Number of Epochs	Optimizer	Regularization techniques	Number of layers
1	64	15	Adam	Early stopping	3
2	64	15	RMSProp	Early stopping	3
3	64	24	RMSProp	Early stopping	3
4	64	16	Adam	Early stopping	3

Figure 23 Training parameters for base model

The training accuracy starts higher than the validation accuracy, as expected, but they both increases steadily and starts to converge fast only after a few epochs. The training and validation accuracy seem to stabilize at around 92%. Both training and validation loss decreases sharply and stabilized after a few epochs. The only regularization technique used for the base model was early stopping which was specified to track the validation loss and activate after 5 epochs with no improvement and the restore the model with the best weights. In version 2 we decided to try out a different optimizer and this version also ran the full 15 epochs, but it was a bit more stable in its validation accuracy and loss. We therefore tried to run it for 25 epochs instead which did not yield any improvement in accuracy (appendix 10). Lastly, we tried to switch out the LSTM layer with a GRU layer. This yielded our best model regarding accuracy and can be seen in figure 23 marked with green.

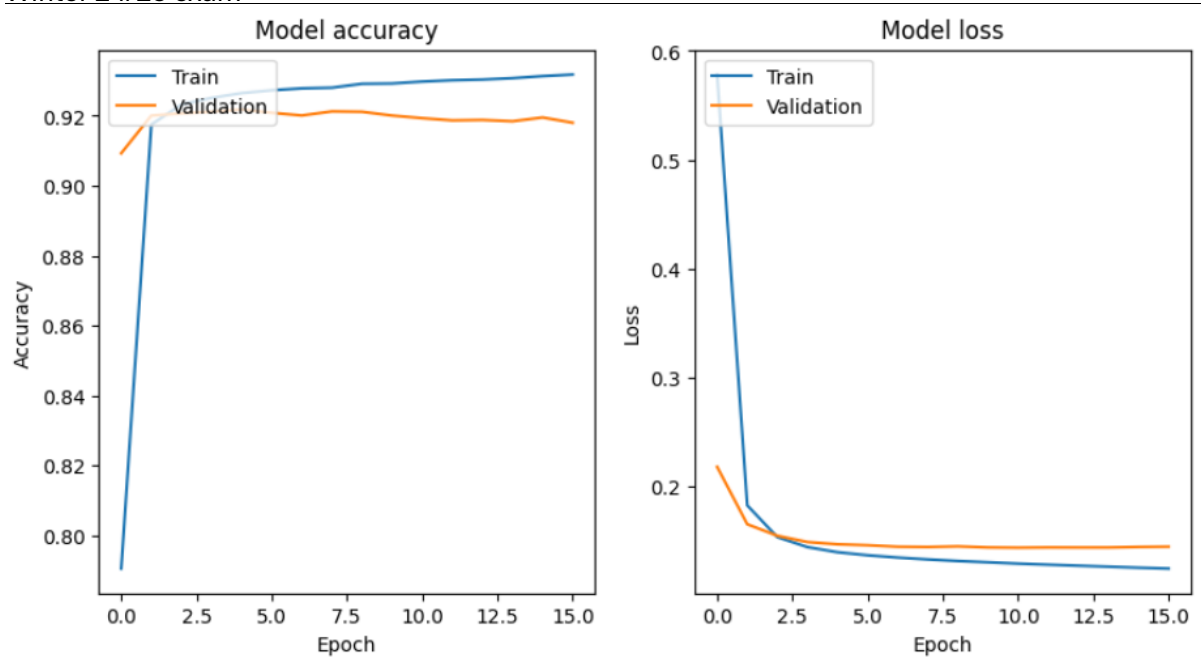


Figure 24 Base model version 4 with GRU

We then decided to change the architecture of the base model, and this resulted in model 2, with its specification shown in figure 25. As shown in appendix 11 this model was slower at converging in model loss and accuracy than the base model. This is due to it having a more complex architecture and more parameters to compute on. We also added dropout which helps prevent overfitting but also introduces noise to the model and slows down the learning process. To accommodate the slower learning and converging, we decided to run the rest of the models and versions on 25 epochs. For version 3 of model 2 we tried RMSProp again and the accuracy was a bit worse but the convergence sped up compared to using Adam (appendix 11).

Model 2:

Version ID	Batch size	Number of Epochs	Optimizer	Regularization techniques	Number of layers
1	64	15	Adam	Early stopping, dropout, L2	6
2	64	21	Adam	Early stopping, dropout, L2	6
3	64	25	RMSProp	Early stopping, dropout, L2	6

Figure 25 Model 2 and versions with training parameters for model 2

For model 3 we decided to add a GRU layer, and this increased the accuracy by a small amount. In model 3 version 2 we also decided to change the vocabulary from 1500 to 2000 and max sequence length from 150 to 200. The purpose of this was to make our model capture more information and process longer sequence lengths, thereby losing less information/meaning in longer sequences. This

yielded our best model, but it did trigger early stopping after 17 epochs. This could have been a sign of overfitting and therefore in model 3 version 3 we added batch normalization to try and counteract this. This sped up the convergence but yielded a worse accuracy (appendix 12).

Model 3 with GRU:

Version ID	Batch size	Number of Epochs	Optimizer	Regularization techniques	Number of layers
1	64	22	Adam	Early stopping, dropout, L2	7
2	64	17	Adam	Early stopping, dropout, L2	7
3	64	13	Adam	Early stopping, dropout, L2, batch normalization	10

Figure 26 Model 3 versions and training parameters

Performance evaluation and final RNN model

Performance metrics for model 1 and 2 can be found in appendix 13. Across our 3 main models we started out well, with our first model achieving 92.1% test accuracy in version 4. From there we tried to make our model more complex to improve its performance. From this we saw slight increases in the overall performance. Overall, we saw that our models performed slightly better on the training set which is to be expected. There was not a big difference in the performance on the training data compared to the validation data which meant that overall, our models have performed well at generalizing to unseen data and not overfitting. We also saw that the test performance in our models was close to the validation performance which again indicate that the models perform well on unseen data.

The best RNN performing RNN was model 3 version 2 which can be seen in figure 27 and 28 marked with green. It achieved the highest test accuracy, test recall, and test precision.

Model 3 – accuracy

Version ID	Training accuracy	Validation accuracy	Test accuracy
1	0.931	0.924	0.923
2	0.936	0.930	0.931
3	0.940	0.926	0.926

Figure 27 Model 3 accuracy metrics

Model 3 – precision and recall

Version ID	Test recall	Test precision	Validation recall	Validation precision
1	0.923	0.925	0.924	0.925
2	0.931	0.934	0.930	0.932
3	0.926	0.928	0.926	0.928

Figure 28 Model 3 key metrics

Insights and analysis

During the development and training of our models, we did hit some obstacles in regard to making any significant improvements to the performance from the base model to the final model. Looking at the data set there is an imbalance in the categories, and we could have tried to address this by using class weights where we would apply higher weights for lower represented classes. Taking a closer look at the confusion matrix for our best model where the columns are represented by the following emotions: {0:'sadness', 1:'joy', 2:'love', 3:'anger', 4:'fear', 5:'surprise'}. Here we can see that 921 post were wrongly labelled as 'love' instead of 'joy'. These false positives for love indicate that our best model has are harder time distinguishing between these emotions.

```
array([[11740,    42,     0,   191,    89,     0],
       [   50, 13055,   921,    81,    17,     3],
       [    3,   167, 3304,     4,     4,     3],
       [  190,    45,     2, 5296,   147,     0],
       [  175,    15,     2,   168, 4400,    11],
       [   13,   156,     4,     2,   405,   976]])
```

Figure 29 Confusion matrix for the best performing model

To address this, we could have improved to model complexity potentially by adding more layers or reviewing our preprocessing and explore features specific to these two emotions. We also thought about using recurrent dropout to examine how it would affect our models. However, due to computational restraints we decided not to use recurrent dropout as it was too computationally expensive.

Overall, the good performance of our models showed us that RNN's are very suitable for the emotion classification task we had here. Because the simpler models also performed good on the data, the text might have been in a format that was distinctive in what emotions were present and therefore easier to classify compared to other datasets with social media posts, where there tend to be a lot of domain specific jargon and noise in the data.

Question 3 (C)

The code for this part can be found in the file 'Preprocessing and apply Pretrained model for classification on Bluesky.ipynb'

a. The best model was model 3 version 2. This model will be used to classify the Bluesky posts. The model as well as the tokenizer was saved. We then applied the same preprocessing steps to the Bluesky dataset as we did with the Dair AI data set. Afterwards we handpicked 30 entries from the preprocessed data. We choose these 30 entries because we think that they exhibit the emotion “anger” clearly. The pretrained was then used to classify these entries which gave the result shown in figure 30. Based on this a total of 25 out of 30 was classified correctly which is an accuracy of 83.33%. This seems to be good considering that the model was trained on data where the emotion where very clear on the text. But this subset was also picked specifically because it showed clear signs of emotion. Our pretrained model might not have performed as well on more general/random posts.

	label	clean_text	predicted_label
0	anger	oh gobshite news plethora lunatic daily	3
1	anger	fuck alexia	3
2	anger	fuck sane people wont drink lets sit back let ...	3
3	anger	would never pay	3
4	anger	hate mfs anything else bro go back felons app ...	3
5	anger	good god edit post function platform	3
6	anger	yeah first image fuck trump pretty angry decid...	3
7	anger	every time see idiot behind wheel car say dont...	3
8	anger	makes angry avindmanbyskysocial honorably serve...	3
9	anger	lol dont get upset often	3
10	anger	words joyful andor relatively calm mood make c...	3
11	anger	hate ohio one fucking failure hope michigan ki...	3
12	anger	make friends ur insane never really learned so...	3
13	anger	typical old angry single democrat	3
14	anger	pew pew idiot	3
15	anger	one heard fentanyl idiot	3
16	anger	im going argue idiot accuses bullshit nobodys ...	3
17	anger	anyone believes idiots post agreed close south...	3
18	anger	idk men take dick pics without modicum shame l...	0
19	anger	proved point fucking idiot	3
20	anger	black friday im couple friends high school bac...	4
21	anger	id definitely call fucking idiot terrorist apo...	3
22	anger	watching fox news lately like seeing dog caught...	3
23	anger	people get nervous hear asshole dont want come...	0
24	anger	cant even get proper supervillains rich asshol...	1
25	anger	fuck everyone listens joe rogan	3
26	anger	creepy scumbags	3
27	anger	fucking hate man much asshole	3
28	anger	breaking local dipshit wants dumb shit	0
29	anger	one biggest ways shooting foot indie artist ga...	3

Figure 30 Handpicked and labeled data (30 entries)

b. The pretrained model was then used to classify the first 10000 entries in the preprocessed Bluesky set. It gave the emotion distribution showed in figure 31. Comparing the distribution from the Bluesky data with the emotion distribution in the DAIR AI data shown in appendix 6 we see a clear difference. In the Bluesky set ‘anger’ account for a lot more of the distribution and emotions like ‘joy’ and ‘sadness’ represent considerably less. We can also see that emotions like ‘love’ and ‘surprise’ are very

underrepresented. This could either indicate a data bias, where there simply were fewer instances of these emotion or a model bias where our pretrained model is not sensitive enough to capture the emotions in the Bluesky data set. Based on the distribution this indicates a domain shift, meaning that the data our model was trained on has a different distribution than the Bluesky data. This can have a negative impact on the model's performance, resulting in less accurate classifications.

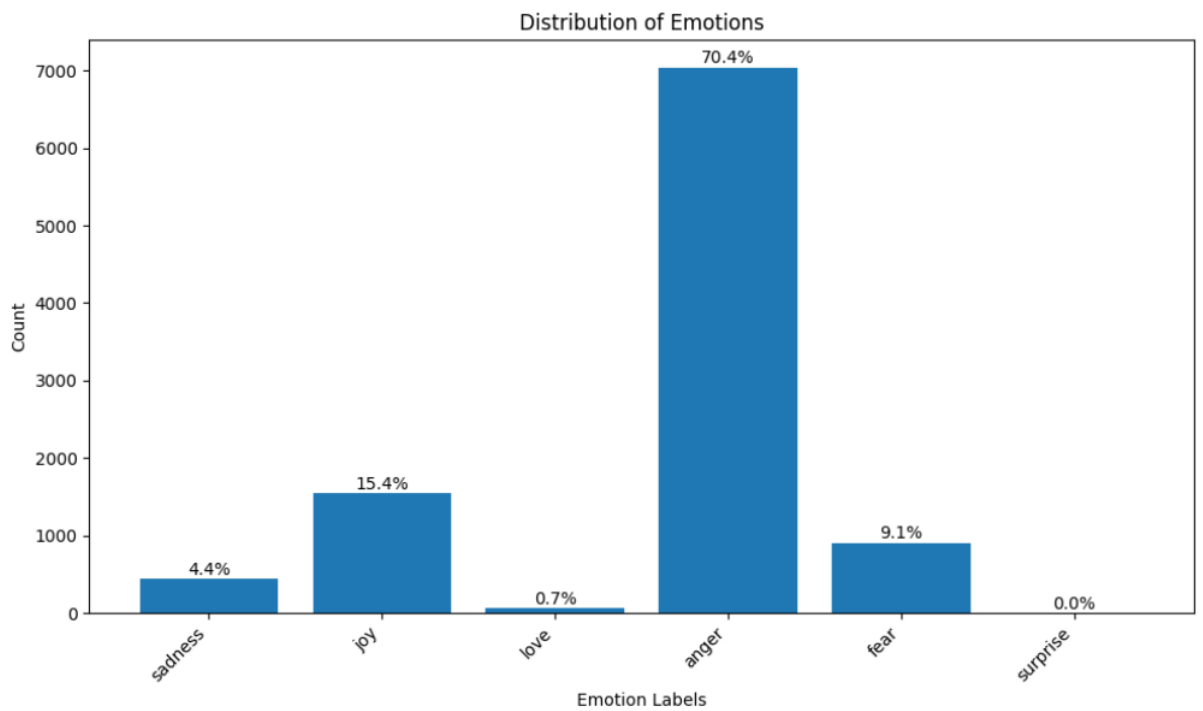


Figure 31 Distribution of emotions on Bluesky of the first 10000 entries

c. When looking at the dataset from DAIR AI emotion dataset, we can see it consists of well-structured and clearly labels of text. This makes it the perfect example for our model to run and train on. However, when the model is being trained on such a clean dataset it might struggle to learn another dataset like the one from BlueSky, due to the linguistic variability of the language. The linguistic variability refers to the social or contextual differences in the ways that people use a particular language. When looking on the dataset from BlueSky, we see that the texts are often shorter and rich in slangs, abbreviation and emojis, which is not represented in our training data for our pretrained model. Because our RNN model isn't trained on the same type of data that is in BlueSky it leads to a domain shift. Domian shift is when a model is trained on data from one domain but deployed and used for data from another domain. Because the data in these domains have different distributions, the model struggles to perform well in the target domain (Salem, H. B., 2024, November 14). To complement and try to increase the performance, one of the strategies here is fine-tuning on

the platform where the target domain is. When training on the target domain the linguistic variability and domain shifting problem disappears and we could see the accuracy rise.

Question 4 (C & Ch)

- a. When training the model on DAIR AI dataset we achieve a promising result of 93% testing accuracy, however when the model is then applied to the BlueSky dataset it has some limitation, due to the linguistic variability. Our RNN model is prone to overfitting because it is trained on such clean and structured data. When the model is applied to social media text, such as the one in BlueSkye the performance decreased on the 30 handpicked entries with only 83.33% accuracy. This decrease in accuracy would probably be worse on the 10000 classified entries in Bluesky. Another challenge our model faces is the context specific nuances. The BlueSky dataset consist of many rows, where many, if not all of the rows contain language with sarcasm and irony and this can be hard for our model to capture the meaning. This also leads us to the domain shift problem.
- b. If we were to enhance our model, so it could capture more underlying patterns and it could be used for real world applications, we could have to fine tune our model to the BlueSky dataset or data in a similar domain. By doing so, we could capture the linguistic patterns, such as the slang and abbreviations that are used. For this we could use a pretrained transformer-based model named Bidirectional Encoder Representation from Transformer (BERT). This model can enhance contextual understanding and generalization capabilities. Another model that that might also be worth looking at is the Valence Aware Dictionary and Sentiment Reasoner (VADER), as this model is pretrained to interpret informal expressions and sentiment intensity. When combining these 2 methods we then create a hybrid approach where VADER provides sentiment scores, and BERT captures the deep emotions in the dataset. With these two models we could eventually increase the accuracy of predictions the dataset from BlueSky.

Another strategy we also could have used was data augmentation and a more expanded dataset from BlueSky. We could have labeled more of the data from BlueSky to improve the model's generalization, and the reduce the impact of the linguistic variability.

By incorporating all these methods into or model in the future, it would better adapt to all the linguistic variation and improve in emotion recognition.

References

- Chollet, F. (2017). Deep Learning with Python. <http://cds.cern.ch/record/2301910>
- GeeksforGeeks. (2024, October 10). Support Vector Machine (SVM) algorithm. GeeksforGeeks. <https://www.geeksforgeeks.org/support-vector-machine-algorithm/>
- GeeksforGeeks. (2025, January 16). Random Forest algorithm in machine learning. GeeksforGeeks. <https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/>
- GeeksforGeeks. (2023, March 31). Gradient boosting in ML. GeeksforGeeks. <https://www.geeksforgeeks.org/ml-gradient-boosting/>
- IBM. (n.d.). What are convolutional neural networks? <https://www.ibm.com/think/topics/convolutional-neural-networks>
- Salem, H. B. (2024, November 14). Tackling Domain Shift in AI: A Deep Dive into Domain Adaptation. *Medium*. <https://medium.com/@bensalemh300/tackling-domain-shift-in-ai-a-deep-dive-into-domain-adaptation-c2debd758edd>
- Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. Retrieved from <https://arxiv.org/pdf/1905.11946>.

Appendix

1. Architecture for ResNet-inspired model

```
#Resnet inspired model
def bottleneck_block(x, filters):
    """Simple Bottleneck Residual Block"""

    shortcut = x #input for the skip connection

    # 1x1 Conv
    x = Conv2D(filters // 4, (1, 1), padding='same', activation='relu')(x)

    # 3x3 Conv
    x = Conv2D(filters // 4, (3, 3), padding='same', activation='relu')(x)

    # 1x1 Conv
    x = Conv2D(filters, (1, 1), padding='same', activation=None)(x)

    # If input and output channel dimensions doesnt match, then we adjust shortcut, might need sadjustment
    if shortcut.shape[-1] != filters:
        shortcut = Conv2D(filters, (1, 1), padding='same', activation=None)(shortcut)

    # Add shortcut (skip connection)
    x = Add()([x, shortcut])
    x = Activation('relu')(x)

    return x

# Our ResNet-inspired model
def resnet_model():
    input_img = Input(shape=(224, 224, 3))

    x = Conv2D(32, (3, 3), padding='same', activation='relu')(input_img)
    x = bottleneck_block(x, 32)
    x = bottleneck_block(x, 64)
    x = bottleneck_block(x, 128)
    x = GlobalAveragePooling2D()(x)

    x = Dense(128, activation='relu')(x)
    x = Dropout(rate=0.2)(x)
    y = Dense(14, activation='softmax')(x)

    model = Model(inputs=input_img, outputs=y)
    return model

# Compile the model
sgd_opt = SGD(learning_rate=0.01, momentum=0.9, decay=0.0, nesterov=True)
model = resnet_model()
model.compile(optimizer=sgd_opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

2. Architecture for EfficientNet-inspired model

```
# MBConv Block (inspired from EfficientNet)
def mbconv_block(x, filters, expansion_factor=4, kernel_size=(3, 3)):
    input_filters = x.shape[-1]

    expanded_filters = input_filters * expansion_factor

    x = Conv2D(expanded_filters, (1, 1), activation='relu')(x)

    x = DepthwiseConv2D(kernel_size=kernel_size, padding='same', activation='relu')(x)

    x = Conv2D(filters, (1, 1), activation=None)(x)

    if input_filters == filters:
        x = Add()([x, x])
    return x

# EfficientNet-inspired CNN
def efficientnet_model():
    input_img = Input(shape=(224, 224, 3))

    # Initial Conv Layer (Downsampling)
    x = Conv2D(32, (3, 3), strides=2, padding='same', activation=None)(input_img)

    # MBConv Blocks
    x = mbconv_block(x, 32)
    x = mbconv_block(x, 64)

    # Classification Head
    x = GlobalAveragePooling2D()(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.2)(x)
    y = Dense(14, activation='softmax')(x)

    model = Model(inputs=input_img, outputs=y)
    return model

# Train EfficientNet-inspired model
sgd_opt = SGD(learning_rate=0.01, momentum=0.9, decay=0.0, nesterov=True)
model = efficientnet_model()
model.compile(optimizer=sgd_opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

hist = model.fit(train_dataset_224,
                 validation_data=validation_dataset_224,
                 epochs=10)
```

3. Model summary

Model Summary: Model: "sequential_6"		
Layer (type)	Output Shape	Param #
resnet50v2 (Functional)	(None, 7, 7, 2048)	23,564,800
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 2048)	0
batch_normalization_3 (BatchNormalization)	(None, 2048)	8,192
dense_6 (Dense)	(None, 256)	524,544
dropout_3 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 14)	3,598
Total params: 24,101,134 (91.94 MB)		
Trainable params: 532,238 (2.03 MB)		
Non-trainable params: 23,568,896 (89.91 MB)		

4. Model Architecture for transfer learning model

```
# Function to create the model
def create_transfer_learning_model(num_classes):

    # Loading ResNet50V2 with pre-trained ImageNet weights
    base_model = ResNet50V2(
        weights='imagenet',
        include_top=False,
        input_shape=(224, 224, 3)
    )

    # Freeze base model layers
    base_model.trainable = False

    # Define the new model
    model = Sequential([
        # Input layer
        Input(shape=(224, 224, 3)),

        # Pretrained ResNet50V2
        base_model,

        # Global Average Pooling instead of Flatten
        GlobalAveragePooling2D(),

        # Batch Normalization for stability
        BatchNormalization(),

        # Fully Connected Layer with L2 regularization
        Dense(256, activation='relu', kernel_regularizer=l2(0.0001)),
        Dropout(0.5), # Dropout for regularization

        # Output layer with softmax activation
        Dense(num_classes, activation='softmax')
    ])

    # Compile the model
    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model

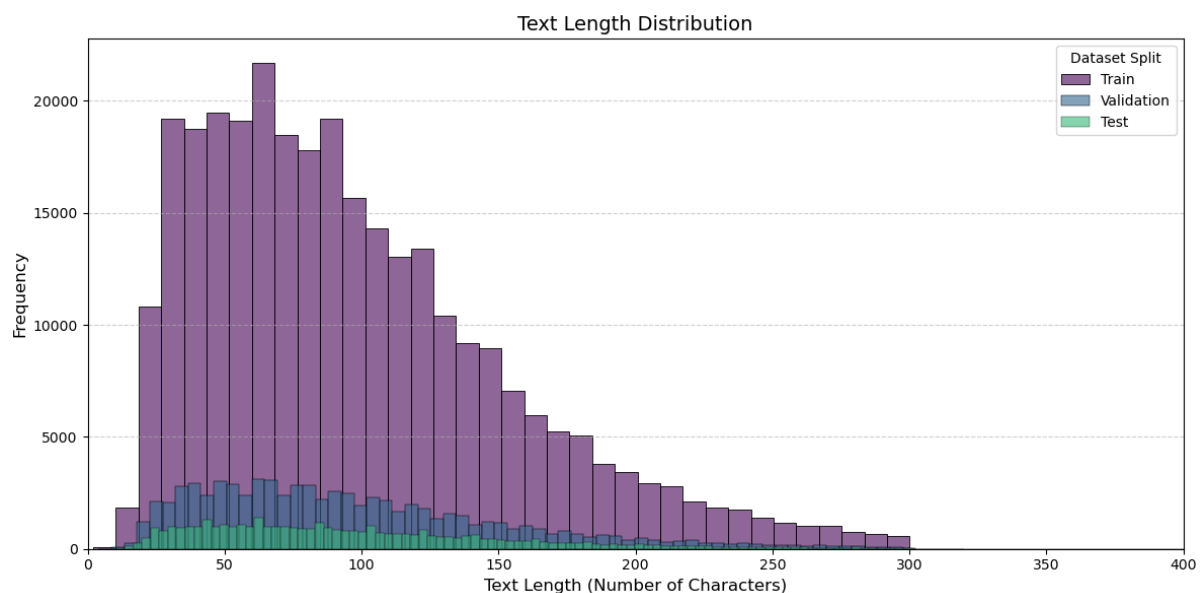
# Function to train the model
def train_model(model, train_ds, val_ds, epochs=50):
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=50,
        restore_best_weights=True
    )
```

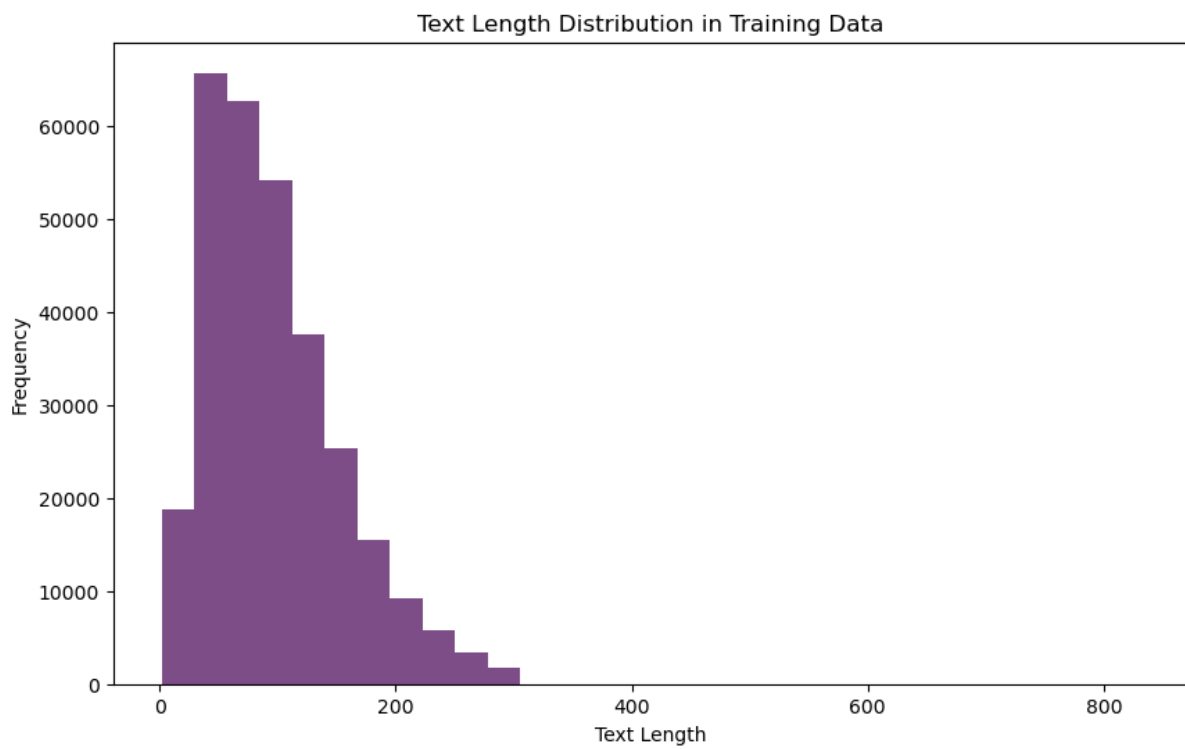
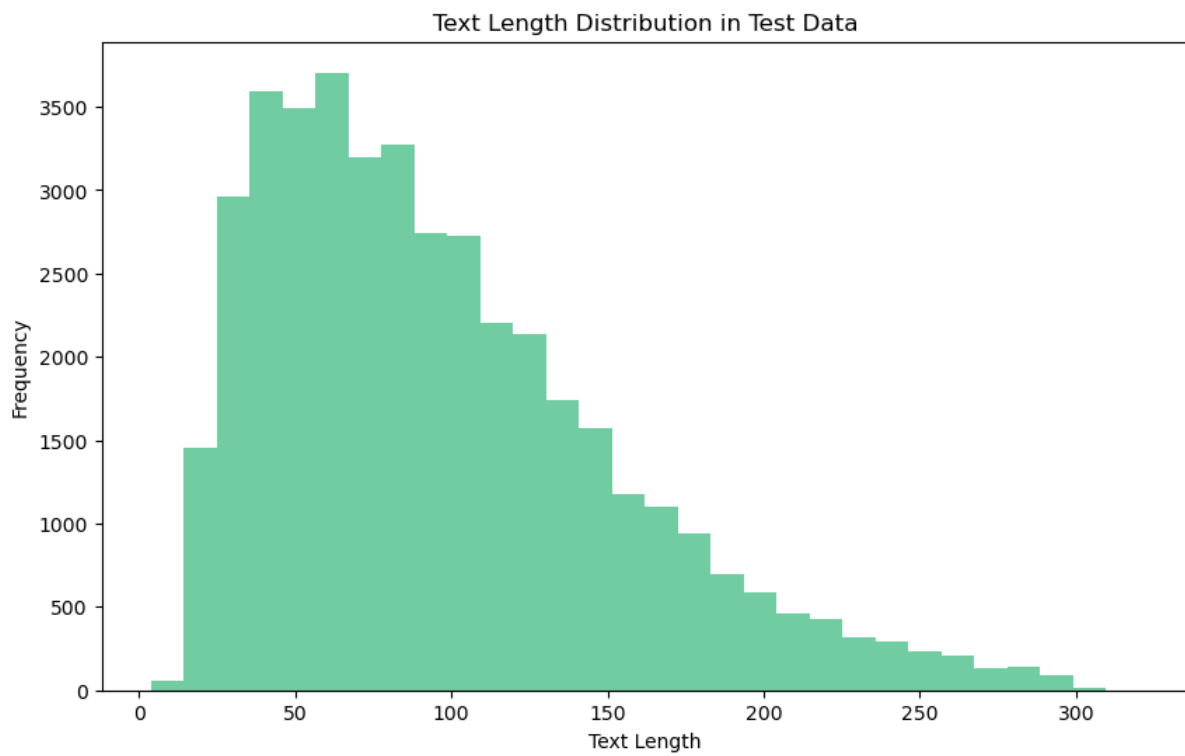
4. Structure and scope of the data in problem 2

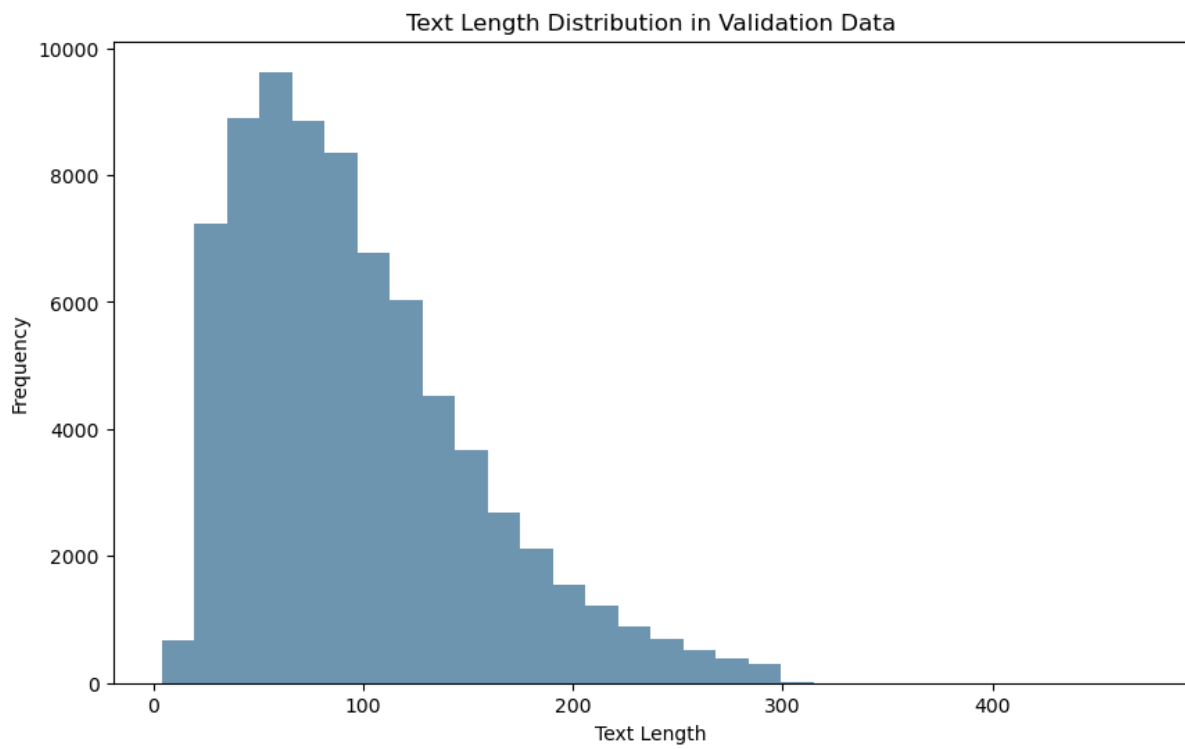
Train Dataset:

	text	label	label_name
0	i feel loyal to poor janie who has been helpin...	2	love
1	i am feeling discontent i am wasting time wish...	0	sadness
2	im not feeling that sentimental	0	sadness
3	i really feel that if one is strong about what...	1	joy
4	i have managed to get a few things done from t...	1	joy

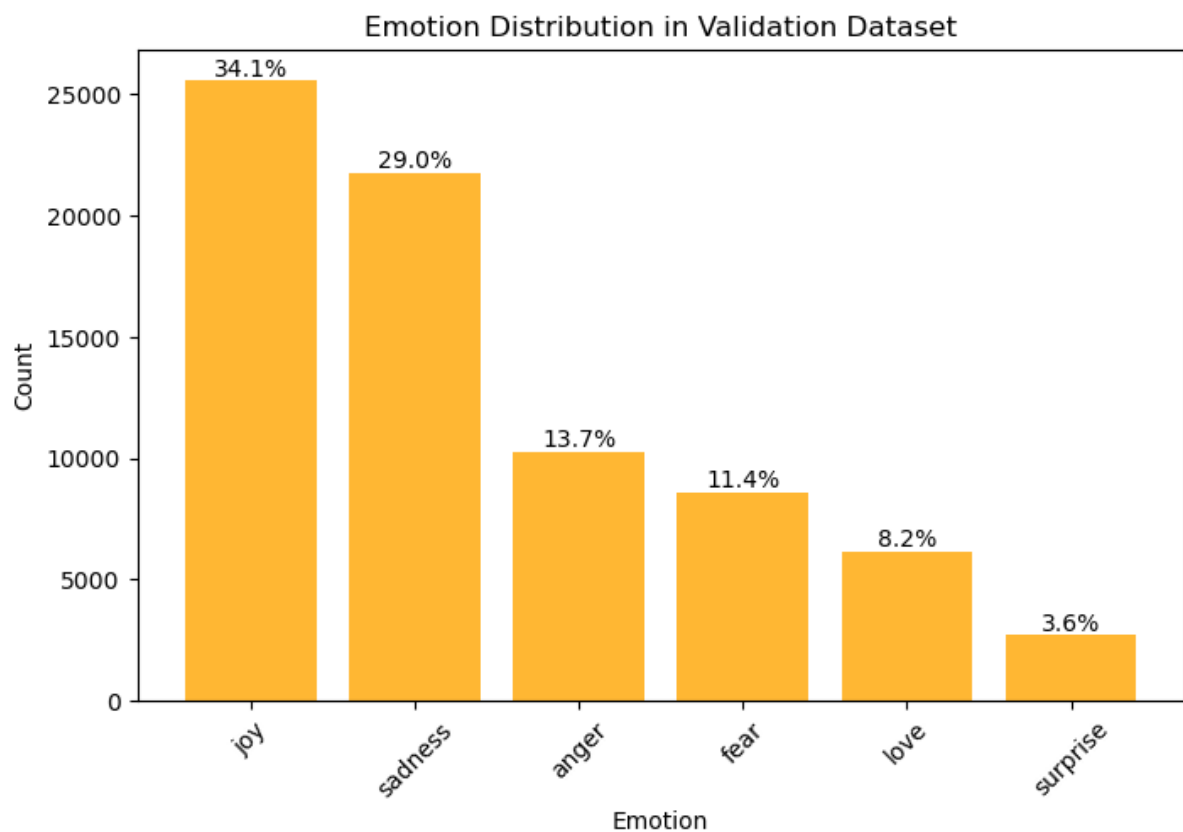
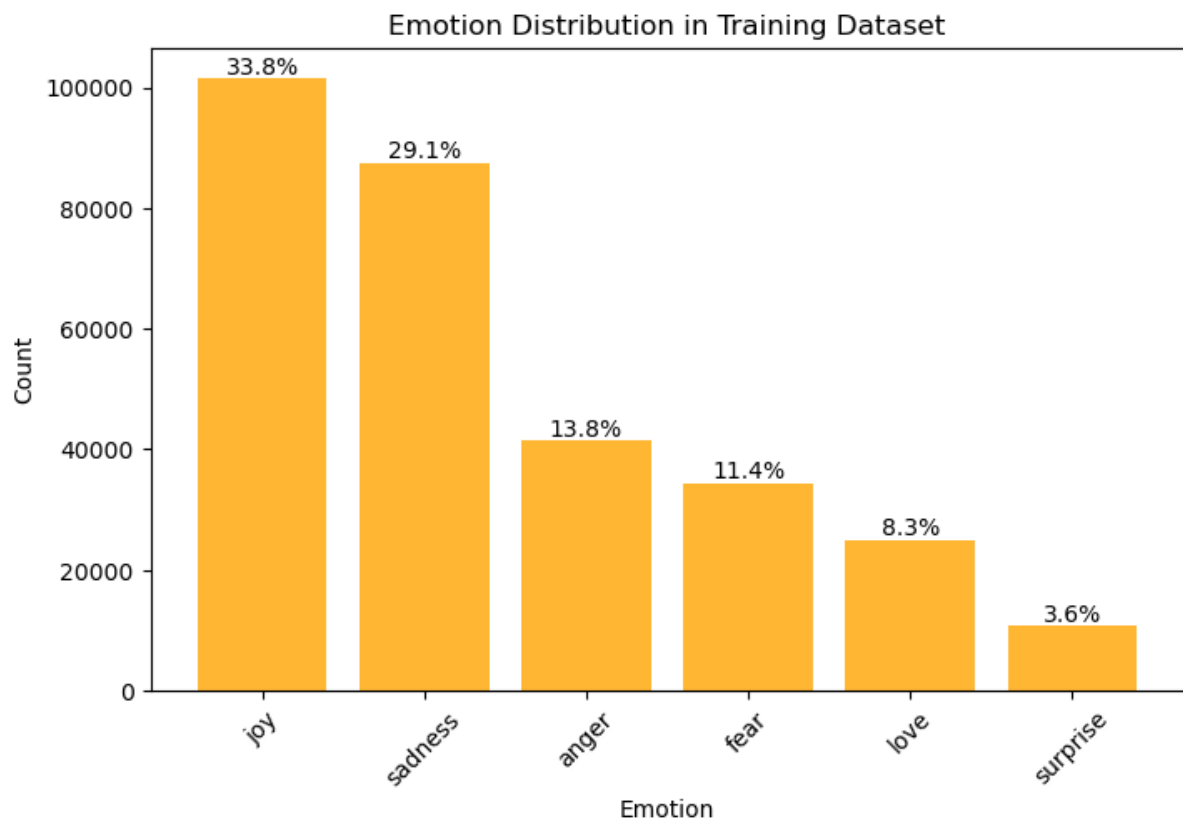
5. Text length

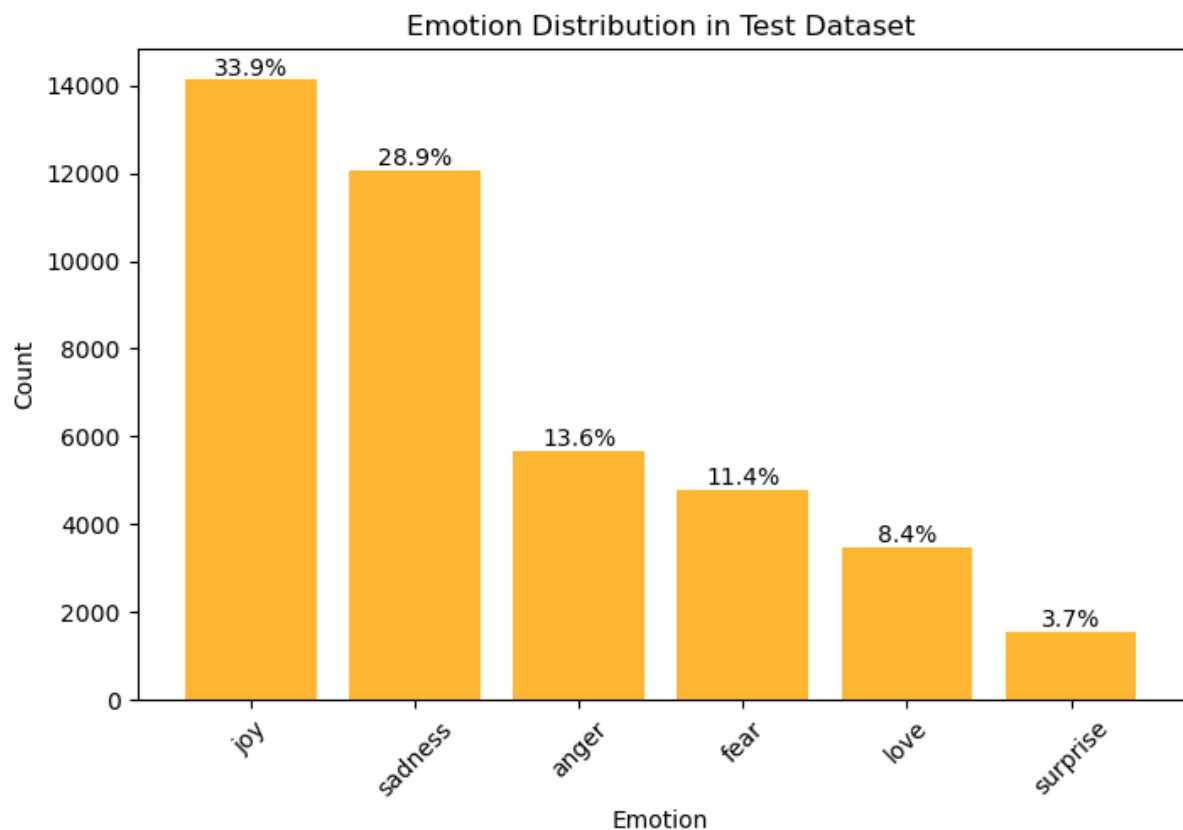






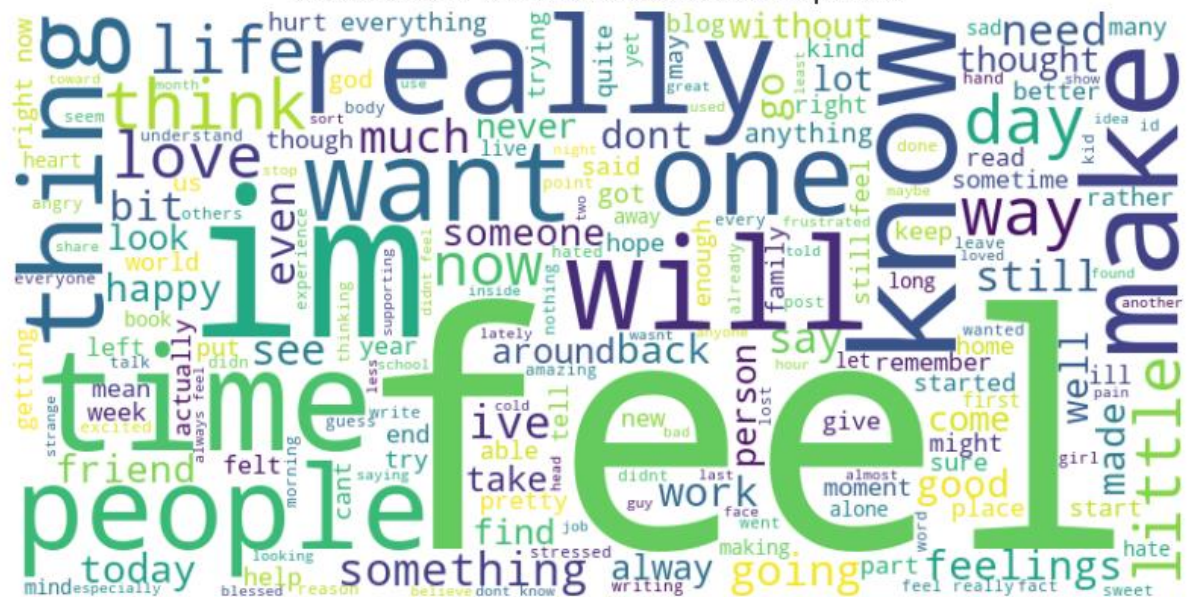
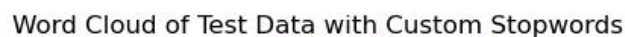
6. Emotion distribution in training, test and validation





7. Word frequency – Dair AI





10. Base model accuracy and loss

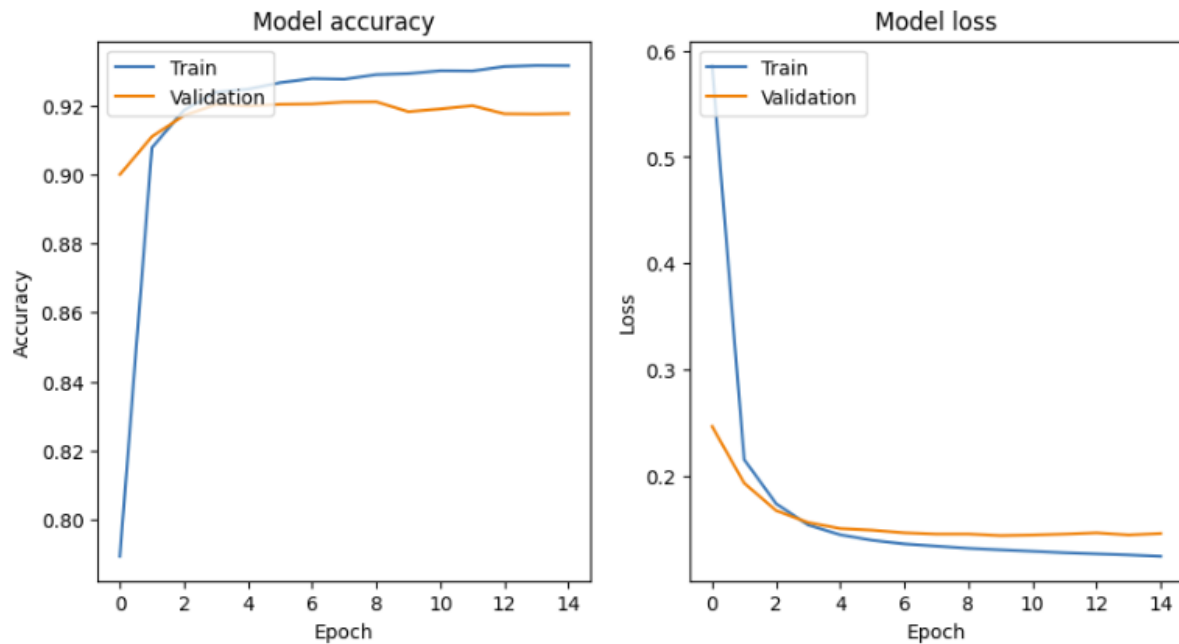


Figure 32 Base model version 1 - accuracy and loss for base model with Adam

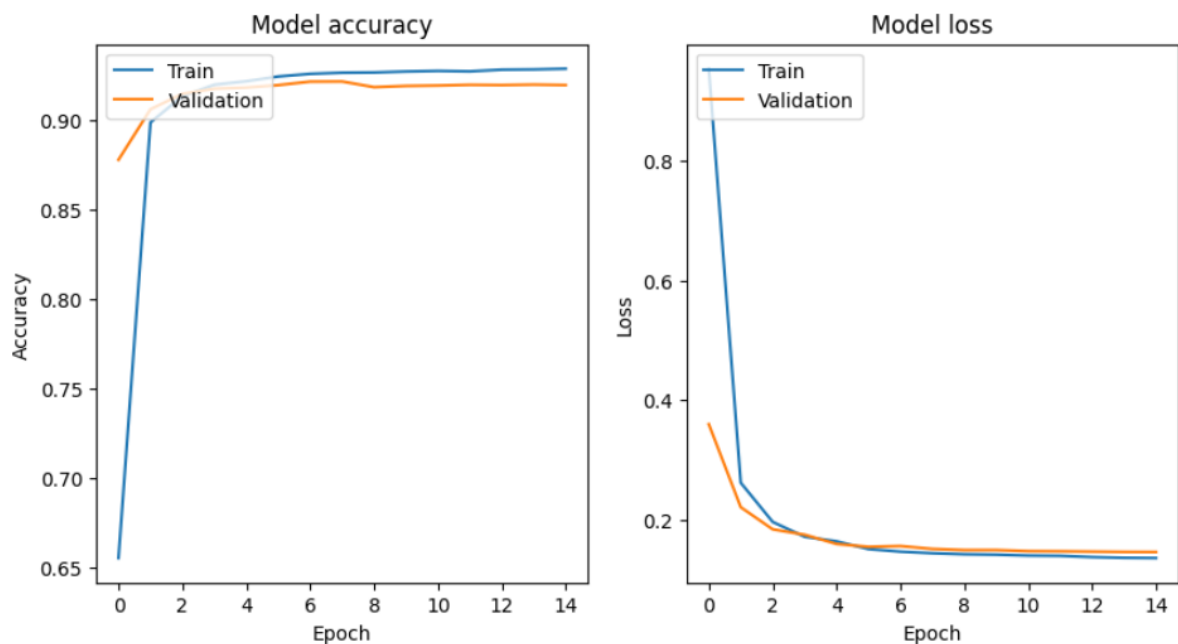


Figure 33 Base model version 2 - accuracy and loss

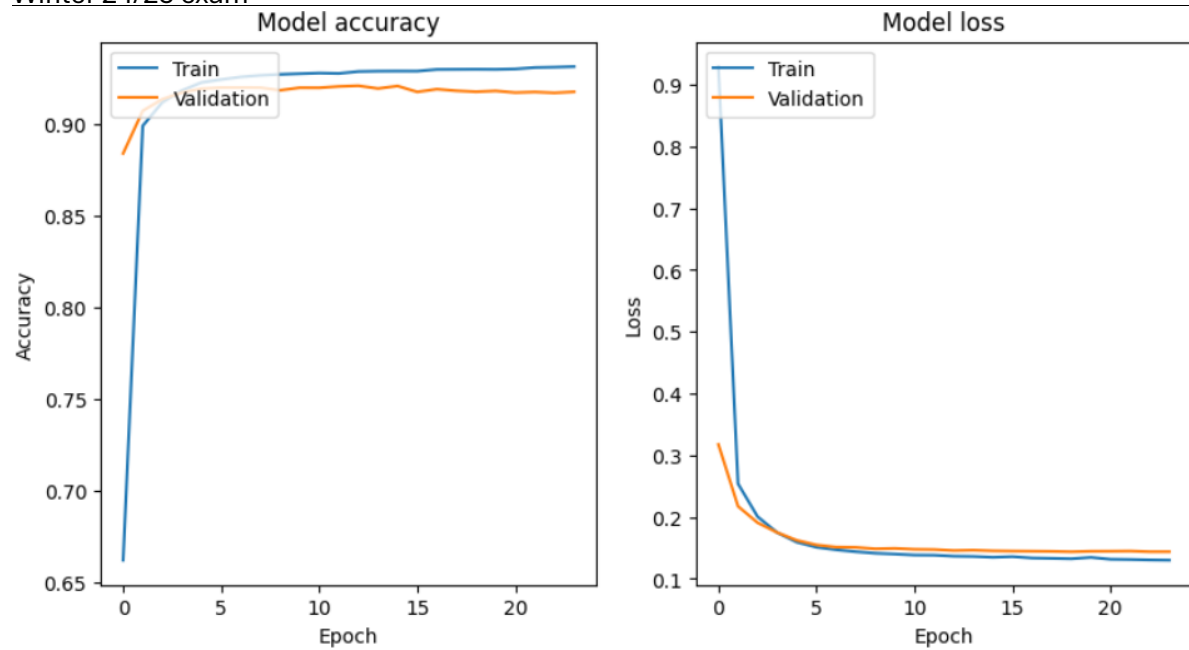


Figure 34 Base model version 3 - accuracy and loss

11. Model 2 accuracy and loss

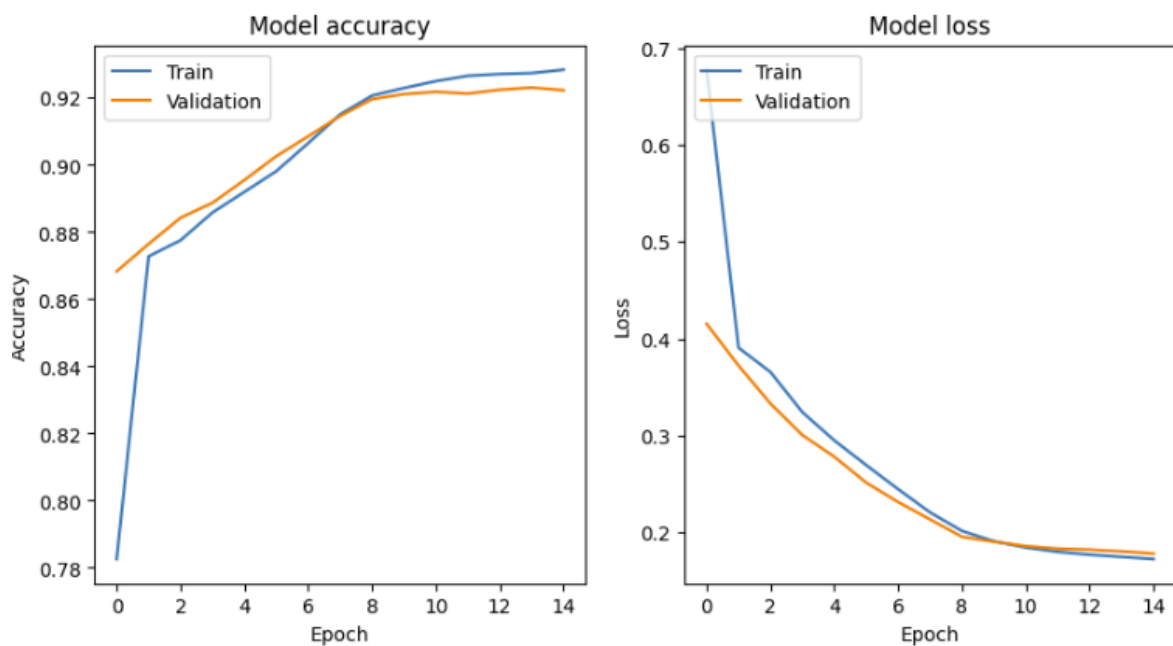


Figure 35 Model 2 version 1

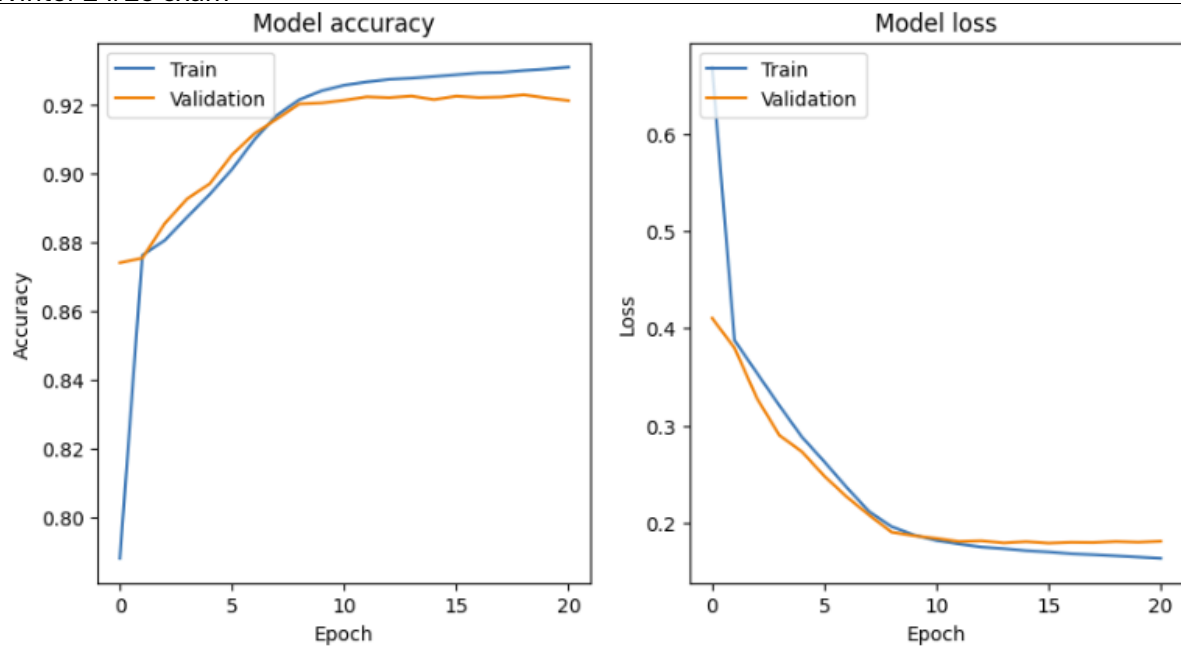


Figure 36 Model 2 version 2

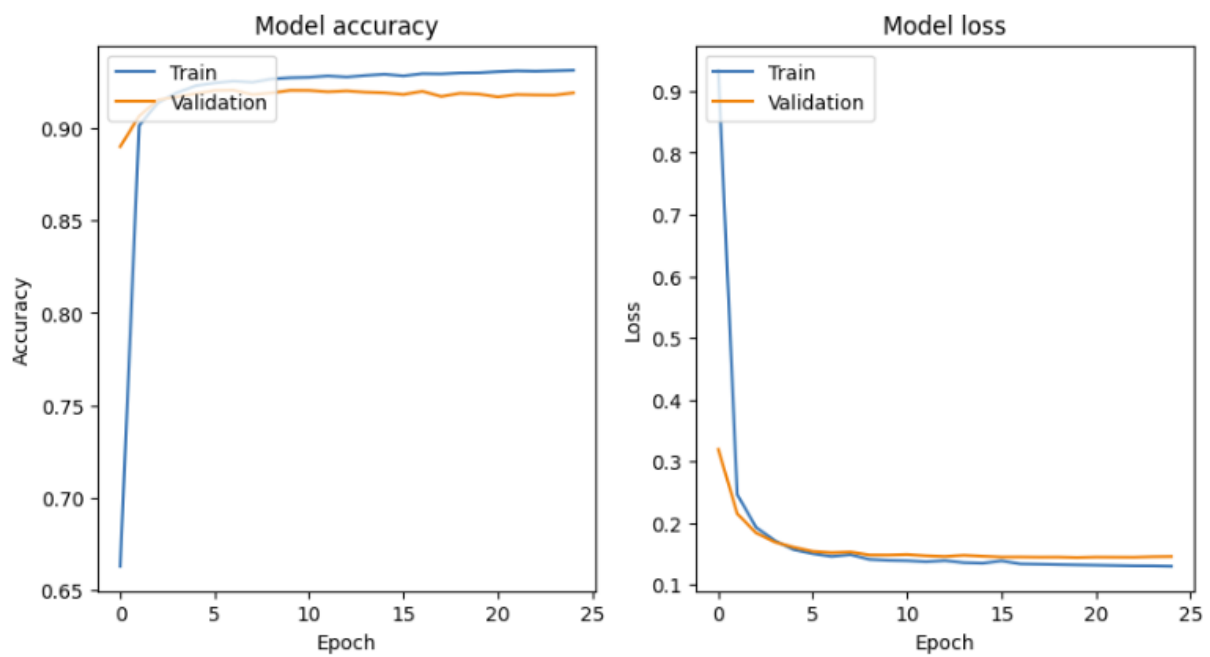


Figure 37 Model 2 version 3 (RMSProp)

12. Model 3 accuracy and loss

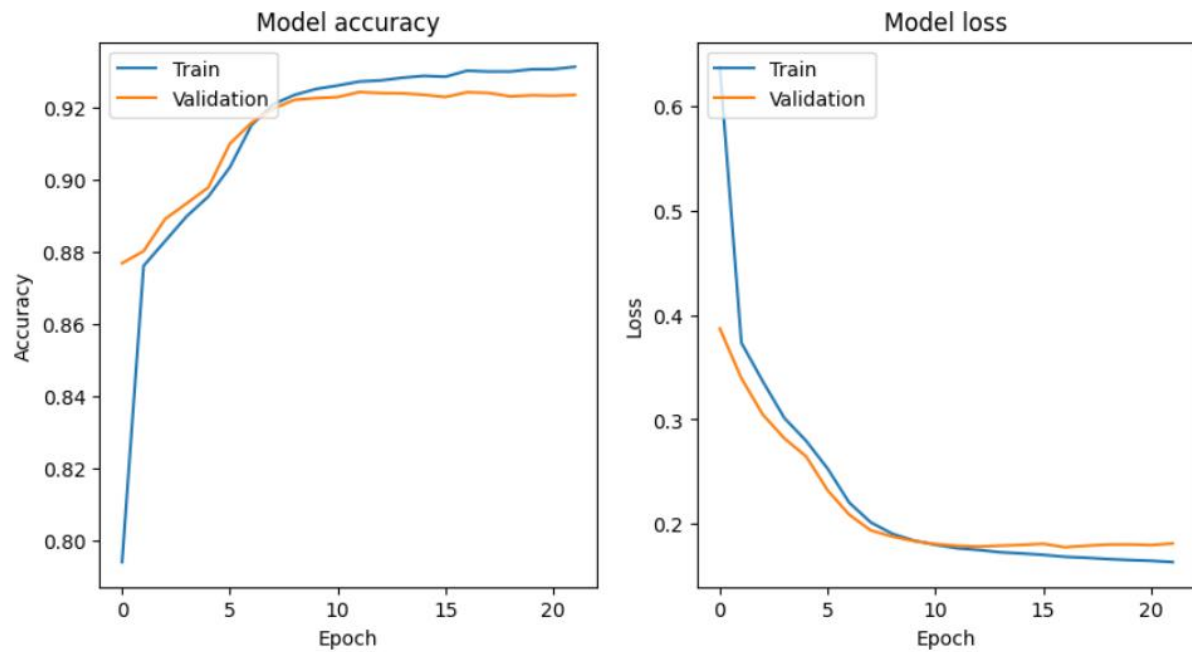


Figure 38 Model 3 version 1

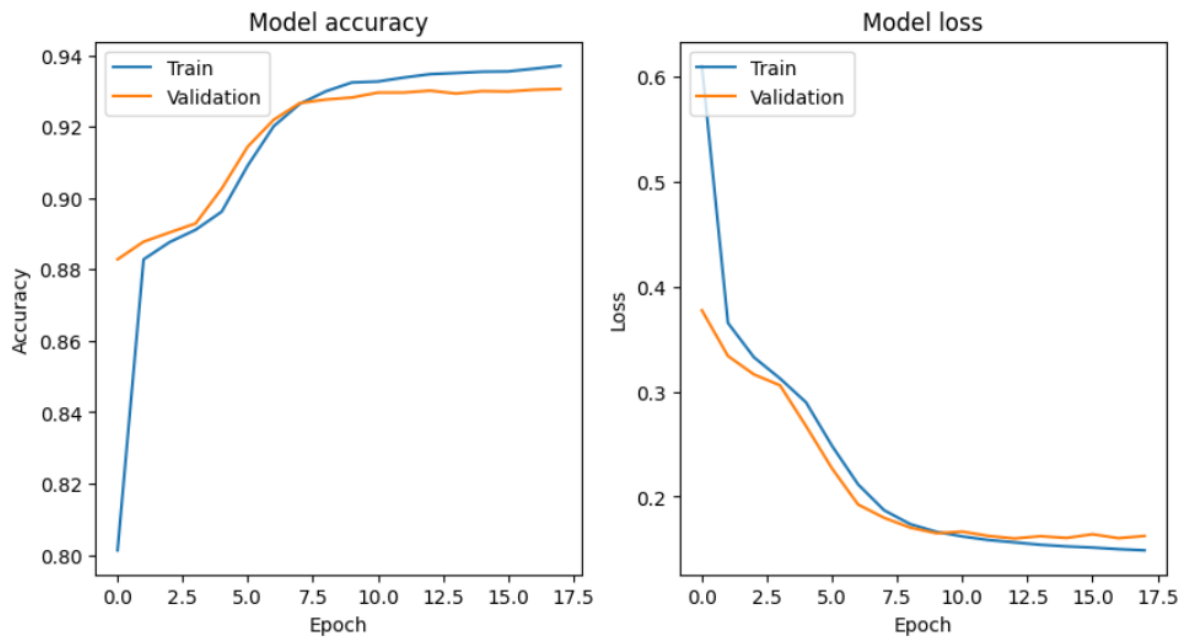


Figure 39 Model 3 version 2 (best model)

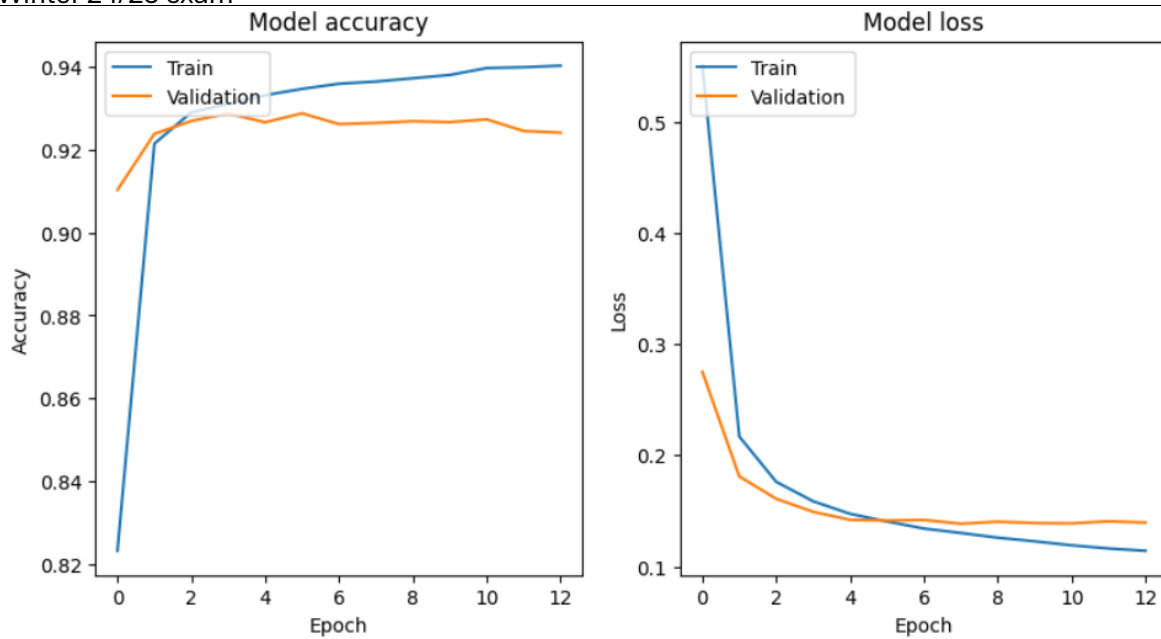


Figure 40 Model 3 version 3 - Batch normalization

13. Key metrics model 1 and model 2

Model 1 – accuracy:

Version ID	Training accuracy	Validation accuracy	Test accuracy
1	0.937	0.918	0.919
2	0.930	0.919	0.918
3	0.932	0.917	0.917
4	0.931	0.919	0.921

Model 1 – precision and recall

Version ID	Test recall	Test precision	Validation recall	Validation precision
1	0.919	0.920	0.919	0.918
2	0.918	0.918	0.919	0.919
3	0.917	0.916	0.917	0.916
4	0.921	0.922	0.919	0.920

Model 2 – accuracy

Version ID	Training accuracy	Validation accuracy	Test accuracy
1	0.931	0.922	0.922
2	0.930	0.922	0.923
3	0.931	0.918	0.918

Model 2 – precision and recall

Version ID	Test recall	Test precision	Validation recall	Validation precision
1	0.922	0.922	0.922	0.922
2	0.923	0.926	0.922	0.925
3	0.918	0.917	0.918	0.918