AML - Spam Detection and Face Alignment System

Introduction

This report presents my implementation and evaluation of two applied machine learning tasks: a spam detection system and a face alignment system. Both were designed, built, tested and analysed. They have been implemented using Python version 3.9.6 locally on my computer and my submission contains all the python scripts and CSV files required for reproducibility.

1. Spam Detection System

1.1 Introduction

Spam detection is a key application of Natural Language Processing (NLP). The goal of this task was to determine if an email was spam or not. This is a binary classification problem, where each document belongs to one of two classes: spam (1) or not spam (0).

The datasets provided consisted of two CSV files: *spam_detection_training_data.csv*, which included labelled examples, and *spam_detection_test_data.csv,* which contained unlabelled emails for final predictions. To solve this task I used the Natural Language Toolkit (NLTK), alongside supporting libraries such as NumPy and pandas.

1.2 - Text Preprocessing

Text preprocessing involves cleaning and preparing raw text data for model training and further analysis. Proper text preprocessing can significantly impact the performance and accuracy of NLP models. The general goal was to reduce noise and vocabulary size whilst keeping semantic meaning. When the vocabulary size is too small, documents may lose useful tokens, so it is important to aim for a balanced vocabulary.

The first key step was word tokenization, which spits up text into individual words. My system used NLTKs word tokenizer for this task because it's quick and easy to implement, removing a lot of unnecessary overhead if chosen to do manually.

The second preprocessing step I included was lowercasing. This step is important because two words with different capitalization (e.g. "Hello" and "hello") would count as separate features, making the vocabulary unnecessarily large. In the given datasets, almost all words were already lowercase, so this didn't impact the vocabulary size, but I still included this step for flexibility.

The third preprocessing step was stopword removal. This removes common words (e.g. "a", "to") to focus on meaningful words. NLTKs stopword list was used to remove them. In total NLTKs stopword list contains 198 words, so the unique vocabulary size could have only reduced by 198 at most, so a small difference was expected.

Two popular preprocessing techniques are stemming and lemmatization. Stemming removes common suffixes from the end of word tokens whereas lemmatization ensures the output word is an existing normalized form of the word that can be found in the dictionary. I chose to use lemmatization in my system because it will always produce real words and be more context-aware.

The final preprocessing steps I included were punctuation and number removal. Since every number counts as a separate feature, this should help reduce the vocabulary size greatly.

I used a vocabulary_size() function to determine the size of the vocabulary in all documents, then the reduction in vocabulary could be calculated at each stage to gauge the improvement. Overall there was a 15% reduction in vocabulary size. To better visualise this I graphed the size of the vocabulary in figure 1 for the different preprocessing steps mentioned above.

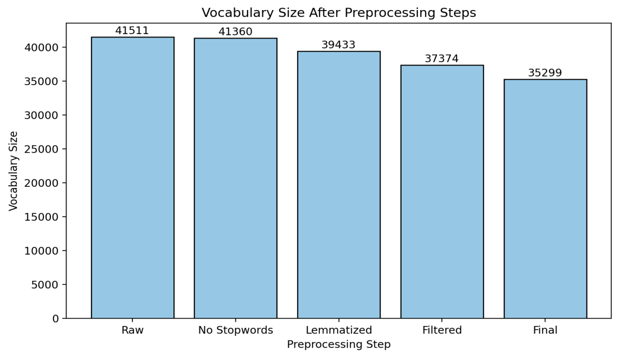


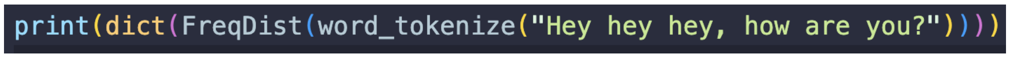
Figure 1: Vocabulary size for different preprocessing steps

It appeared punctuation and number removal (filtered) made a big impact in the overall reduction in vocabulary. Lemmatization also appeared to have a positive impact whilst not being too harsh.

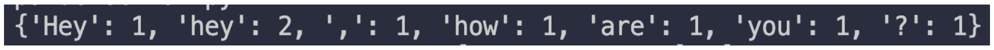
1.2.1 Text Features

Although the data had now been normalized, it still needed further formatting to be inputted into a model. This is where I chose to use a Bag of Word (BoW) representation, which treats each document as a collection of words, and stores the frequency of each word.

Each document is represented as a mapping: {word: count}



```
print(dict(FreqDist(word_tokenize("Hey hey hey, how are you?"))))
```



```
{'Hey': 1, 'hey': 2, ',': 1, 'how': 1, 'are': 1, 'you': 1, '?': 1}
```

The output shows the token “hey” has a frequency of 2, which matches the example string.

A drawback of the BoW model is it treats all words independently and ignores context or relationships between them. It can also lead to high dimensionality due to the potentially large vocabulary size. However, I think it's a suitable choice for this task because it’s simple and effective for smaller datasets like the one used here.

1.3 - Evaluation Metrics

1.3.1 Train/Test Split

Since the testing data had no labels, I had to create validation data to test the model. This involved splitting the training data (with the labels) into two lists, a train split and a test split. With actual ground truths it could be run through a confusion matrix to calculate the performance of the model. I chose an 80/20 split because I thought it gave the model adequate data to test with. I created a function get_train_test_split() to correctly split the data. I decided to create a flowchart in figure 2 to better understand the process.

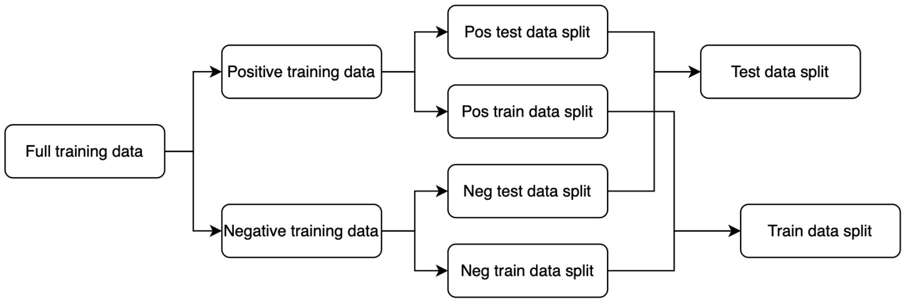


Figure 2: Test/train split pipeline

1.3.2 Confusion Matrix

I used a custom confusion matrix class to determine the true positives, false positives, true negatives and false negatives. With these 4 values the precision, recall and f1 can be calculated. Precision: Out of all the positives predicted, what percentage is truly positive. Recall: Out of the total positive, what percentage are predicted positive. The F1 Score It is the harmonic mean of precision and recall (Jayaswal 2020). This means it is a good overall indicator of performance.

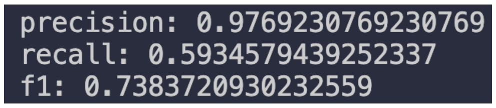
1.4 - Classification Method

1.4.1 - Word List Based Classifier

My first approach was to use a simple word list based classifier. It works by creating two word lists, one for words that appear most often in spam emails, and another for words that appear most often in non-spam emails. When a new email is given, the classifier checks how many words in the message match each list. If more words match the spam list, it classifies the message as spam; if more match the non-spam list, it classifies it as not spam. This approach is simple and fast, but unlike Naive Bayes (See 1.4.2) it doesn’t use probabilities, it just counts word matches, which can make it less flexible and less accurate on complex data.

Results

I ran this classifier on the test data split and used the confusion matrix to determine the precision, recall and f1 values shown below:

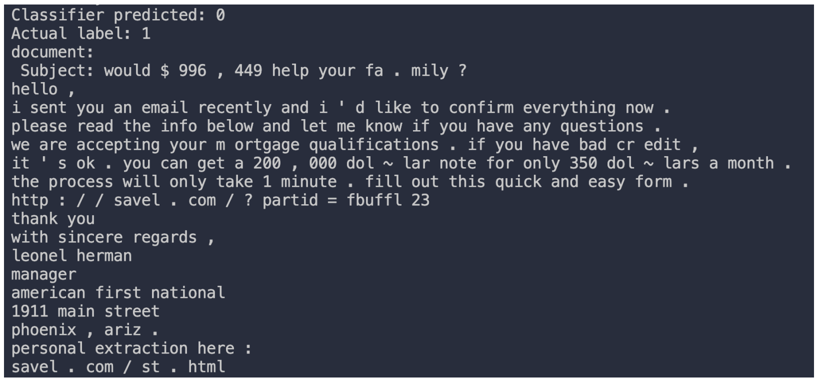


```
precision: 0.9769230769230769
recall: 0.5934579439252337
f1: 0.7383720930232559
```

The model achieved very high precision, meaning that when it predicted an email as spam, it was usually correct. However, its recall was relatively low, which indicates that it failed to identify a significant portion of actual spam messages. This is a concern in a spam detection system, where I think the ability to catch as much spam as possible is crucial.

Failure Case

I randomly selected an email that had been misclassified for further analysis. Figure 3 shows the email that was selected along with the prediction and the actual label.



```
Classifier predicted: 0
Actual label: 1
document:
 Subject: would $ 996 , 449 help your fa . mily ?
hello ,
i sent you an email recently and i ' d like to confirm everything now .
please read the info below and let me know if you have any questions .
we are accepting your m ortgage qualifications . if you have bad cr edit ,
it ' s ok . you can get a 200 , 000 dol ~ lar note for only 350 dol ~ lars a month .
the process will only take 1 minute . fill out this quick and easy form .
http : / / savel . com / ? partid = fbuffl 23
thank you
with sincere regards ,
leonel herman
manager
american first national
1911 main street
phoenix , ariz .
personal extraction here :
savel . com / st . html
```

Figure 3: An email the word list based classifier misclassified

The classifier predicted this spam email as not spam (false negative). This may be due to the presence of many neutral words balancing the score. Also, unusual spacing or characters in words like “m ortgage” and “dol ~ lar” may have affected preprocessing or reduced word frequency, making them less recognizable as spam.

1.4.2 - Naive Bayes Classifier

I then decided to implement the Naive Bayes classification technique with NLTK, hoping it would perform better. Naive Bayes is a classifier that uses Bayes’ Theorem to predict the class of a document based on word probabilities. It assumes all words are independent and calculates how likely a document is to be spam or not by combining the probabilities of its words. I could simply use NLTKs train method to return a classifier object, then use the classify_many() function on the documents.

```
naive_bayes_classifier = NaiveBayesClassifier.train(train_data_split_freq_dist)
cm = ConfusionMatrix(naive_bayes_classifier.classify_many(docs), goldstandard)
```

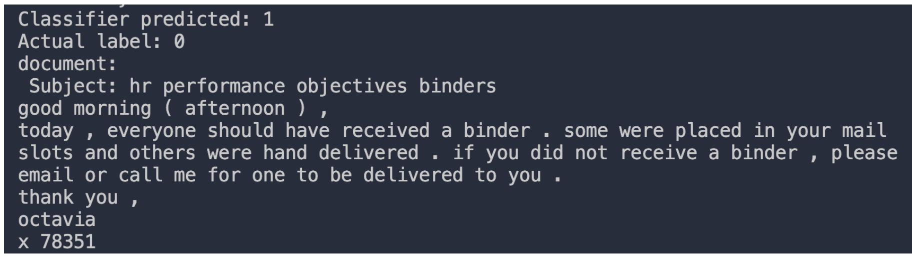
Results

I again ran this classifier on the test data split and used the confusion matrix to determine the metrics:

```
naive bayes precision: 0.8299595141700404
naive bayes recall: 0.9579439252336449
naives bayes f1: 0.8893709327548807
```

Its recall was very high, it correctly identified about 96% of all actual spam emails, so it is very effective at catching spam. Its precision was also solid at 83%. The F1 score therefore came out to be around 89% which I think is a strong well rounded score.

Failure Case

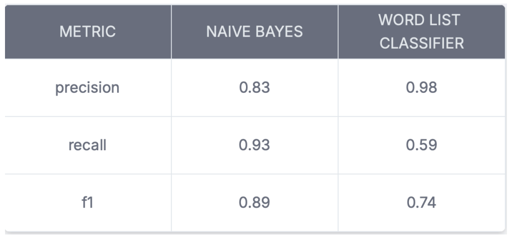


```
Classifier predicted: 1
Actual label: 0
document:
 Subject: hr performance objectives binders
good morning ( afternoon ) ,
today , everyone should have received a binder . some were placed in your mail
slots and others were hand delivered . if you did not receive a binder , please
email or call me for one to be delivered to you .
thank you ,
octavia
x 78351
```

Figure 4: An email the Naive Bayes classifier misclassified

Figure 4 shows another misclassified email. In this example, the classifier predicted this document to be spam (1) when the actual label was not spam (0). So this was a false positive. From a human perspective it is hard to see why this email was classified as spam, as it looked legitimate. My best guess was this email contains a lack of words associated with non-spam.

1.4.3 - Comparing Classifiers



METRIC	NAIVE BAYES	WORD LIST CLASSIFIER
precision	0.83	0.98
recall	0.93	0.59
f1	0.89	0.74

Figure 5: Table comparison between naive bayes classifier and word list classifier on the validation data

For my final predictions I went with the Naive Bayes Classifier. It performs noticeably better overall and would be a good option for real world use. The word list classifier's low recall meant it would fail to detect real spam emails, and I would prefer to have a system that marks messages as spam which aren't instead of the other way around.

1.5 - Final Results

I ran the Naive Bayes model on the full testing data and stored the prediction labels in a CSV file named *results_task1.csv*. I made use of the show_most_informative_features() function from NLTK, which shows the most important words/words with the biggest influence in the decision making process.

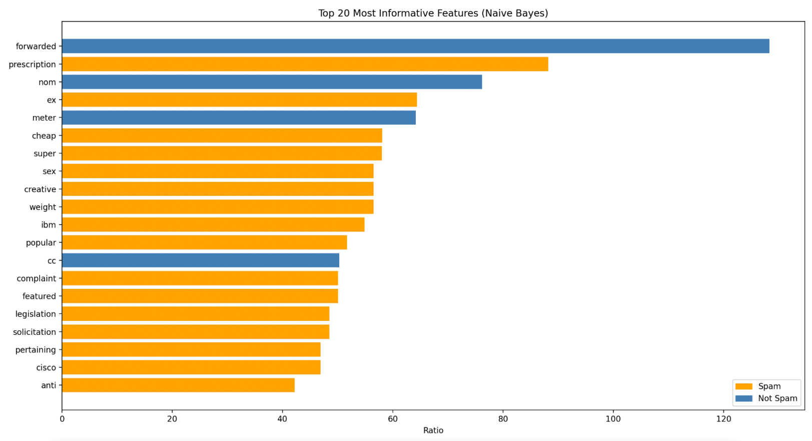


Figure 6: Top 20 most informative features with naive bayes classifier

The table in figure 6 shows common spam words in orange and common non-spam words in blue. 16 out of the top 20 are spam words, this suggests spam emails generally contain more stand out words compared to non-spam.

2. Face Alignment System

2.1 - Introduction

Face alignment or facial landmark detection is a core computer vision task.The goal of this task was to locate the positions of 5 key facial features: left eye, right eye, left mouth corner, right mouth corner and nose tip. There are many approaches to solving this problem. I decided to treat it as a regression problem, where given an image it predicts the set of continuous landmark coordinate locations.

I was provided with two large files: *face_alignment_training_images.npz*, which included images and their ground truth points, and *face_alignment_test_images.npz*, which included images for final predictions. I made use of scikitlearn for its regression model, along with libraries such as NumPy and cv2.

2.2 - Preprocessing

Preprocessing is an important step for having a well trained model. Whilst I could pass the flattened image directly into the model, a better approach would be to use feature descriptors at useful locations, this is where I have utilised Scale-Invariant Feature Transform (SIFT). It generates descriptors at key points to be used as input for the model. My goal was to ensure that the input images were in an optimal condition for extracting high quality SIFT descriptors.

The first preprocessing step I included was resizing the image to a lower resolution. This helps improve the computational efficiency and reduce the margin of error. I reduced the original size by a factor of 4 which gave 64x64 images.

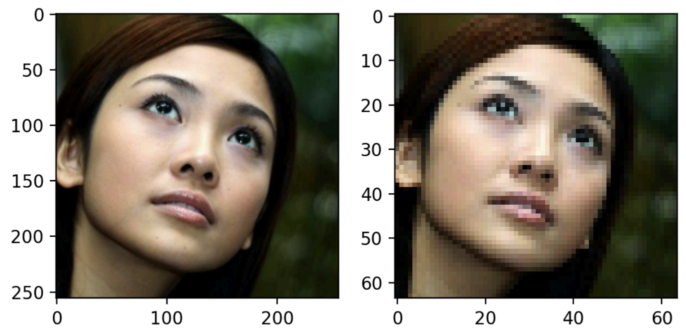


Figure 7: Side by side comparison of an image from the dataset (256x256) and the resized image (64x64)

The second key step was converting the image to grayscale. Grayscale conversion reduces dimensionality and focuses the SIFT descriptor on intensity gradients, which are more informative.

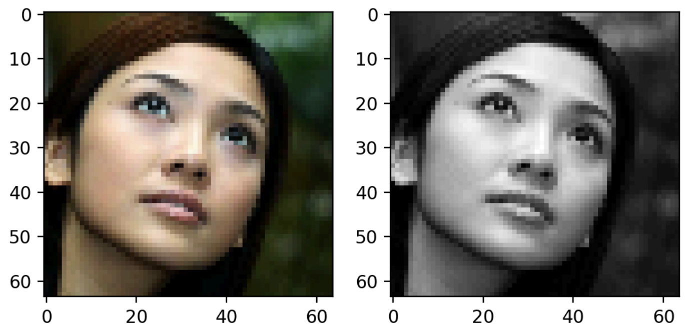


Figure 8: Side by side comparison of the resized image and the grayscale image

2.3 - Training

I chose to implement a cascaded regression model, which refines predictions over several steps instead of in a single pass. At each stage image features are extracted based on the current estimate, and a regression model predicts an update that moves the estimate closer to the correct solution. Early stages correct large errors, while later stages fine tune the details, leading to more accurate results. For the first iteration the model requires an initial estimate of the facial landmark positions, I used NumPy to calculate the mean of all the ground truth points in the training data. This basically gives us a set of average points on a face. I used the same function on all the training images to produce an “average face” image and plotted for visualisation.

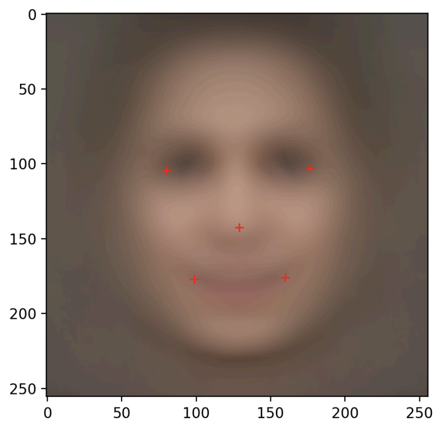


Figure 9: The average face and average points from the training data

I created pseudocode for the cascaded regression function to help with implementation:

```
for K regressors:
    for N images:
        if no previous prediction:
            use averagePoints as initial prediction

        compute SIFT features A using current prediction
        compute target delta = groundTruthPoints - current prediction

    train a Linear Regression model using A as input and delta as target

    for N images:
        update prediction using model output and damping factor
```

2.4 - Evaluation and Testing

An 80/20 train/test split was used on the training set. To evaluate the model's performance, I calculated both euclidean distance and mean squared error (MSE) between the predicted and ground truth points. Euclidean distance measures the straight-line distance between two points, while MSE takes all these distances, squares them, and averages them to give an overall measure of prediction error.

2.4.1 - Baseline setup

The baseline used 64×64 grayscale images, 4 regressors, and damping factors [0.8, 0.6, 0.4, 0.2]. I ran the test split and got the following results.

Figure 10 shows the average points in blue, the ground truth points in green, and the predicted points in red for a randomly selected test image. The regression model successfully adjusts for the rotation and position of the face relative to the average points, indicating it is functioning as intended. However, it struggles with accurately predicting finer details such as the corners of the mouth.

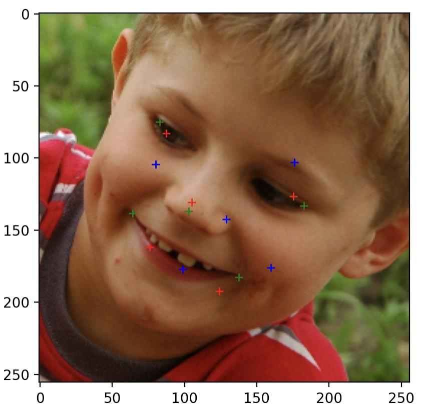


Figure 10: An image from the testing split, with the prediction points in red, and ground truth points in green

Using the provided euclid_dist() function I plotted a histogram of the distances between the predictions and the ground truth points. The results in figure 11 show good performance overall, with most points being just over 1 pixel distance away from the ground truth points. For a 64x64 image these metrics look good on paper, but for the human eye these small differences are noticeable and can definitely be improved upon.

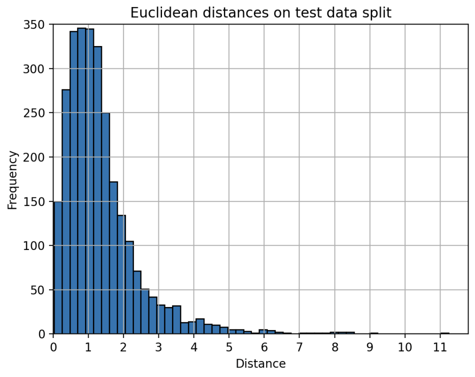
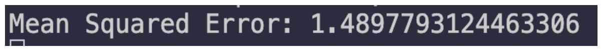


Figure 11: Histogram of distances between predictions and ground truth points on the test data

Finally the MSE value returned ≈ 1.49. This is a fairly small error and for the first run it is good.



```
Mean Squared Error: 1.4897793124463306
```

2.4.2 - Effect of Damping Values and Number of Regressors

I first experimented with varying the number of regressors and dampening values. Increasing the number of regressors should improve accuracy up to a point, because early regressors fix large misalignments and later ones refine details, but too many may lead to diminishing returns or overfitting. I tested the number of regressors from 5 to 30 with a uniform damping value of 0.1 and 1.0, then with scaling damping values from 0.1 to 1.

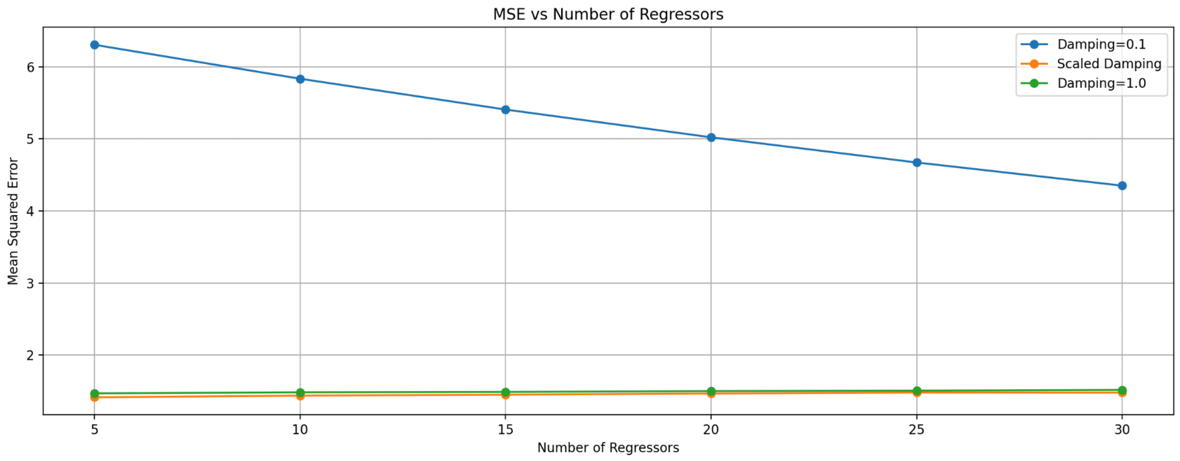


Figure 12: MSE vs number of regressors at different dampening values

The results in figure 12 show that a low constant damping value of 0.1 is ineffective, likely because it undercorrects at each step. Increasing the number of regressors helped reduce the MSE as predicted, but not significantly enough to make this configuration viable. A constant damping value of 1.0 performed much better, likely because each step had a stronger influence. However, as the number of regressors increased performance slightly declined, suggesting that larger updates in later stages can lead to overcorrections. The best performance came from scaled damping values, where the optimal number of regressors appeared to be 5. This suggests that using fewer regressors with appropriately scaled steps is more effective than relying on many regressors with either weak or overly strong corrections. With this setup I got a MSE value of ≈ 1.41.

2.4.3 - Effect of Different Preprocessing Stages

To evaluate the impact of different preprocessing stages, I conducted three tests. The first used my original preprocessing pipeline: image resizing followed by grayscale conversion. The second test added a Gaussian blur as an additional step. The third test used the raw, unprocessed images. In all cases, I used the optimal configuration identified earlier: 5 regressors with damping values ranging from 0.1 to 1.0. Gaussian blur is a common step in computer vision and helps reduce noise in an image.

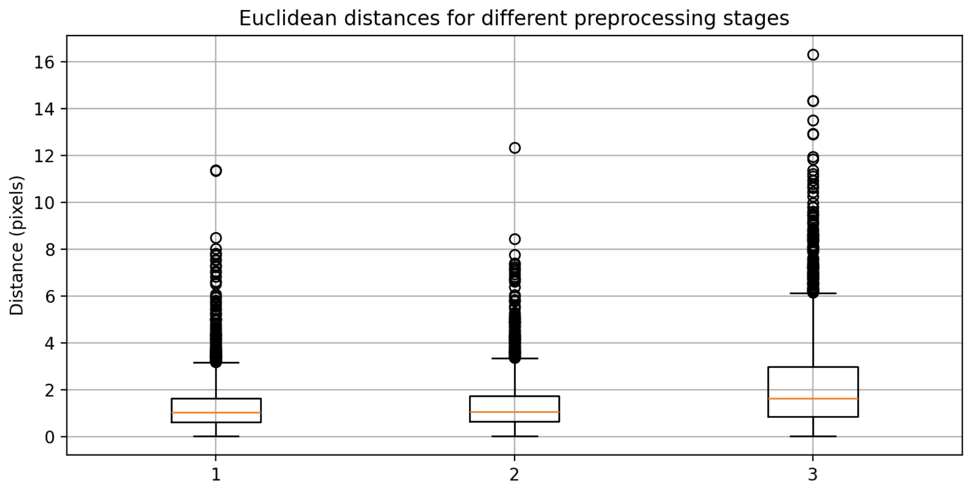


Figure 13: Boxplot comparing the distances between the predictions and ground truth points at different levels of preprocessing

The results in figure 13 show that my original preprocessing steps (1) and the additional gaussian blur step (2) performed very similarly to each other. Adding gaussian blur actually caused the furthest outlier between the two, and overall the median value was marginally lower on my original preprocessing pipeline. No preprocessing (3), as expected, performed poorly in almost every metric and confirmed the importance of the preprocessing stage.

For my final predictions I went with my original preprocessing pipeline. I ran the model on all the testing images and stored the points in a CSV file named *results_task2.csv*.

References

Jayaswal, Vaibhav. 2020. “Performance Metrics: Confusion matrix, Precision, Recall, and F1 Score.” Towards Data Science. https://towardsdatascience.com/performance-metrics-confusion-matrix-precision-recall-and-f1-score-a8fe076a2262/.