University of Macau

Faculty of Science and Technology

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

CISC3023 Machine Learning Final Project

Wound Area Location In Animal Model Images

by

Huang Yanzhen, DC126732

Group Member: Yang Zhihan, DC127992

Project Supervisor

Prof. Chen Long

December 15, 2024

Contents

1 Abstract

In this project, we focus on the application of animal wound location detection. Detecting wound in an image involves 4 regression problems. The model will predict the coordinates, namely a scalar tuple of x, y, and the size of the wound, namely a scalar tuple of w, h.

We chose Random Forest Regressor and Support Vector Regression as our two candidate models, where each model has their own customized user-definable hyper-parameters. Respectively, we proposed 4 key parameters for each model. Each hyper-parameter has 4 to 5 candidate values. Grid searches of combinations of parameters will be performed to inspect the effects of these parameters on the performance of the models.

We adopted a 4-model design by introducing ModelSet, training 4 separated models under a single training process, and embedding them into an Experiment Object. Under each training process, we use 3-fold cross-validation with the evaluation metric of mean square error (MSE) to inspect the separated training performance of the 4 models. The separated MSE of an individual ModelSet will be summarized into one for choosing the best performing ModelSet in a Grid Search. Both RFR and SVR will produce their own best model.

The training and testing performance of RFR and SVR will be compared and analyzed, and the inference results of both of the best models on the test set will be visualized. Both models performed fairly well.

2 Methodology

2.1 Basic Functions

2.1.1 Get Labels

This project only involves supervised learning, with labels pre-defined for each image. Labels are stored in a `.csv` file, with each row having the format of fileName, x, y, w, h. Each image corresponds to the 4 key labels. Varying the type of input data (Training and Test), the function `get_labels` converts the `.csv` file into a numpy array for processing.

```
def get_labels(data_type: str) -> np.ndarray:
    df = pd.read_csv(f"./Wound/{data_type}/myData.csv", delimiter=";")
    return df.to_numpy()
```

Listing 1: Function to get all the labels.

2.1.2 Get Images

Using the `pillow` package, the function `get_images` read the image file using the image names given in the label `.csv` file. Under the trade-off between a shorter training time and richer information preservation, each image is resized into the size of a $32 \times 32 \times 3$ matrix data structure. Each input is a colored, 32×32 sized image, flattened to a total of 3072 numeric scalar.

The function `get_images` also injects the augmentation function for image augmentation purposes with allowance of arbitrary keyword arguments `**kwargs` for better extendability.

```
def get_images(data_type: str,
               image_names: np.ndarray,
               augmentation: Union[Callable[[np.ndarray, Any], np.ndarray],
                   None] = None,
               flatten=True,
               resize=True,
               **kwargs) -> np.ndarray:
    images = []
    for i_name in image_names:
        img = Image.open(os.path.join(f"./Wound/{data_type}/", i_name))
        if resize:
            img = img.resize((32, 32), Image.BICUBIC)
            img = np.array(img)
        if augmentation:
            img = augmentation(img, **kwargs)
        images.append(img.flatten() if flatten else img)
    images = np.array(images)
    return images
```

Listing 2: Function to get all the images.

2.2 Data Augmentation

Data augmentation involves adding varieties to the current data to increase the abundance and variance of training data, aiming for better training performance.

Augmentation with respect to images for wound detection follows this basic idea: First, as long as the wound part settled around the center of the image is preserved, the model should detect it without problem even if the parts that's relatively close to the edge is altered. Second, if the image is stretched along the x or y axis without the center of the image moving, only the corresponding w and h values should be changed while the coordinate of the wound x, y should be preserved.

Following this basic idea, we designed two types of image augmentation functions: Adding Black Edge and Image Stretching.

2.2.1 Add Black Edge

The augmentation function `add_black_edge` adds an inner black edge with a custom width to the 32×32 colored image. The augmented version of the image will be used as the training data along with the original one with the same x, y, w, h tuple to inform the model that the edge parts of the image should not affect the prediction results.

```
def add_black_edge(img: np.array, w: int = 4) -> np.array:
    if w > min(img.shape[0:2]) // 2:
        raise ValueError(f"Width of the edge must be smaller"
                         "than half of the shorter side of an image.")
    new_img = np.zeros_like(img)
    new_img[w:-w, w:-w, :] = img[w:-w, w:-w, :]
    return new_img
```

Listing 3: Image augmentation function that adds a black edge with a variable width.

We used the default 4 pixels from the `add_black_edge` function as our preference for the black edge's width.

```
# Add black edge
Y_be = get_labels(data_type="Training")
X_be = get_images(data_type="Training",
                  image_names=Y_be[:, 0],
                  augmentation=add_black_edge,
                  w=4)
```

Listing 4: Injecting the `add_black_edge` function into `get_images` function to retrieve an augmented image set.

2.2.2 Stretching

The augmentation function `stretch` uses the `PIL` package to stretch the images' width w and height h with the factors of f_w and f_h using bi-cubic interpolation. The image will be re-cropped to 32×32 with the same image center.

The stretch factor will be input into the function as a list-like tuple, containing the factor for width and height respectively. The augmented version of the image will be used as the training data along with the original one with the same center coordinate x, y, but the width and height of the wound will be stretched according to the stretching factor.

```
def stretch(img: np.ndarray, f: List[float]) -> np.ndarray:
    fw, fh = f
    if fw < 1 or fh < 1:
        raise ValueError(f"Width and height factors should be"
                         "greater than or equal to 1.")
    # New widths
    new_width = int(img.shape[1] * fw)
```

```
    new_height = int(img.shape[0] * fh)

    # Adjust image
    img_pil = Image.fromarray(img)
    img_resized = img_pil.resize((new_width, new_height), Image.BICUBIC)

    # Crop regions
    # Keep 32x32 size
    left = (new_width - 32) // 2
    top = (new_height - 32) // 2
    right = left + 32
    bottom = top + 32

    # Crop image
    img_cropped = img_resized.crop((left, top, right, bottom))

    # Convert to numpy array
    img_stretched = np.array(img_cropped)
    return img_stretched
```

Listing 5: Image augmentation function that stretches the image on the width and height with both factors larger than 1.

We chose the factor for both w and h of 1.05 as it provides the smallest mean squared error during the test in pre-processing. The corresponding target values are also updated according to the factors

```
# Stretch height
Y_sh = get_labels(data_type="Training")
X_sh = get_images(data_type="Training",
                  image_names=Y_sh[:, 0],
                  augmentation=stretch,
                  f=[1.0, 1.05])
Y_sh[:, 4] *= 1.05
# Stretch Width
Y_sw = get_labels(data_type="Training")
X_sw = get_images(data_type="Training",
                  image_names=Y_sw[:, 0],
                  augmentation=stretch,
                  f=[1.05, 1.0])
Y_sw[:, 3] *= 1.05
```

Listing 6: Injecting the `stretch` function into `get_images` function to retrieve an augmented image set.

2.2.3 Visualization of Augmentation

While keeping the original data, we made 3 augmentations. The first augmentation adds a black edge with a width of 4 pixels; The second and third augmentation stretches the images along their heights and widths respectively. Below is a visualization of all the image augmentation process.

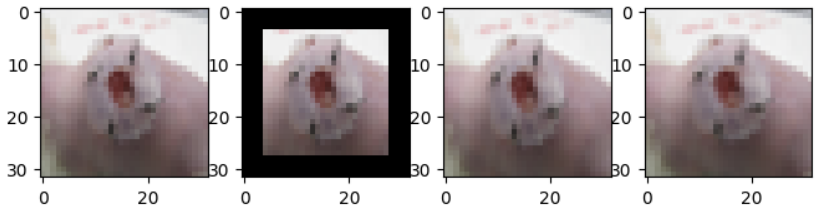

Figure 1: Image augmentation process. First image: The original image. Second Image: Add black edge. Third and forth images: Stretch on height and weight.

2.3 Train

2.3.1 ModelSet: 4-Model Design

In this project, we introduce the concept of a ModelSet. A ModelSet is a set of 4 models that produces 4 different target values when given the same input. As a whole, it could be considered as a whole "model" that takes a flattened image as an input and outputs a vector of inference results with the length of 4.

The 4 models in a single ModelSet are trained separately, each aiming to solve a single regression problem. A `train` function will produce a single ModelSet.

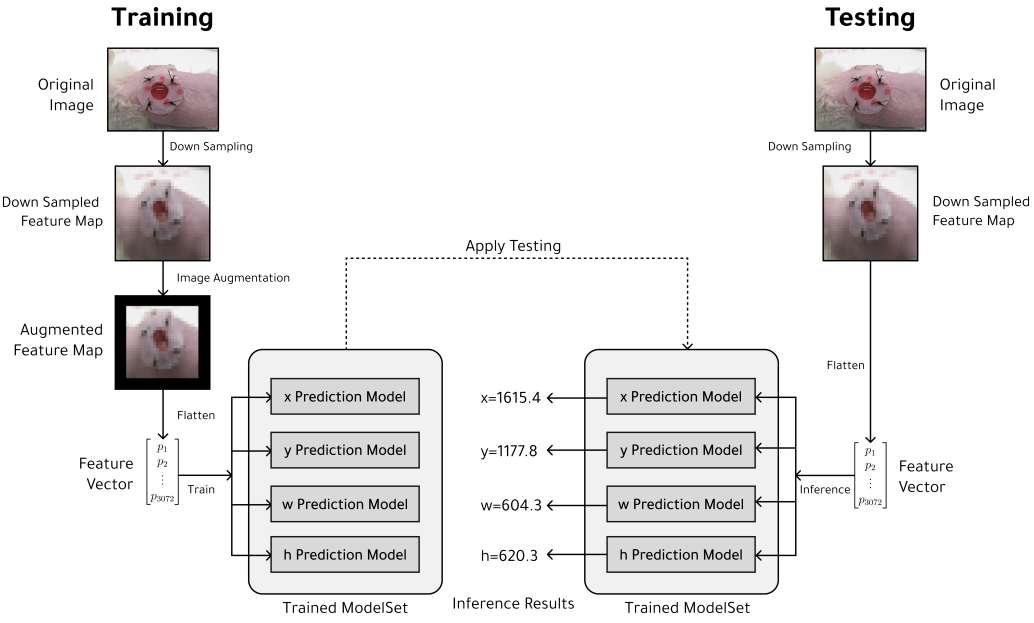

Figure 2: The algorithm flow chart of training and testing process of a ModelSet. The image augmentation is only conducted as data pre-processing during training. This figure is for reference only, and the values inside may not be the actual model output.

2.3.2 3-Fold Cross-Validation

N-Fold cross-validation is a technique to improve the model's performance during training. It requires that the training data be divided into *N* parts, and the model should be trained *N* times, where during the *N*-th test, the *N*-th fold will be used as the validation data while the others be used as the training data.

The original training dataset contains 150 images. As we performed augmentation for 3 times respectively, we obtained a total of 600 images. Therefore, we used 3-Fold cross-validation, where each fold has 200 images. We train 4 models separately for 3 times and get its best fold. After training, the best models for predicting *x*, *y*, *w* and *h* are assembled into a single Experiment Object.

2.3.3 Experiment Objects

The `train` function will output an Experiment Object. A single Experiment Object has the structure given below.

```
{
    "x": {
        "Best␣MSE": smallest_mse,
        "Best␣Fold": best_fold_idx,
        "Avg␣MSE": avg_mse,
        "model": ModelInstance,
    },
    "y": {
        "Best␣MSE": smallest_mse,
        "Best␣Fold": best_fold_idx,
        "Avg␣MSE": avg_mse,
        "model": ModelInstance,
    },
    "w": {
        "Best␣MSE": smallest_mse,
        "Best␣Fold": best_fold_idx,
        "Avg␣MSE": avg_mse,
        "model": ModelInstance,
    },
    "h": {
        "Best␣MSE": smallest_mse,
        "Best␣Fold": best_fold_idx,
        "Avg␣MSE": avg_mse,
        "model": ModelInstance,
    },
}
```

Listing 7: The data structure of an Experiment Object, the output of the `train` function.

Each Experiment Object has 4 sub-objects with the key of *x*, *y*, *w* and *h* respectively, corresponding to the 4 regression targets. To record the training performance, a single sub-object records the model instance, along with the best mean squared error, the index of the fold that produces the best MSE, and the average MSE of the three folds.

```
def train(ModelInstance, X, Y,
          desc: str = "DESC", n_fold: int = 3, save: bool = False):
    ...
    exp = {}    # Initialize the output experiment object.
    for i in range(1, Y.shape[1]):
        # Totally 4 labels to predict. Select one of them.
        y = Y[:, i]

        # Split original data into 3 parts to perform cross-validation
        kf = KFold(n_splits=n_fold, shuffle=True, random_state=1919810)
        splits = kf.split(X)

        # Record MSE of each fold. Keep the model with the smallest MSE
        mse_scores = []
        cur_best_model = None
        cur_smallest_MSE = np.inf
        for train_index, val_index in splits:
            X_train, X_val = X[train_index], X[val_index]
            y_train, y_val = y[train_index], y[val_index]

            model = copy.deepcopy(ModelInstance)
            model.fit(X_train, y_train)

            y_pred = model.predict(X_val)
            mse = mean_squared_error(y_val, y_pred)
            mse_scores.append(mse)

            if cur_smallest_MSE > mse_scores[-1]:
                cur_best_model = copy.deepcopy(model)

        exp[semantic_y[i]] = {
            "Best MSE": cur_smallest_MSE,
            "Best Fold": np.argmin(mse_scores),
            "Avg MSE": np.mean(mse_scores),
            "model": copy.deepcopy(cur_best_model)
        }
        del cur_best_model
    ...
```

Listing 8: The main training iteration in the `train` function.

The abstract ModelSet is embedded in the Experiment Object, as it contains the four separated regression models. Since we train the four models separately, the four separate regression models may not come from the same fold.

The `train` function left us a high extendability in the type of the models that we could choose different types of model classes and instantiate them outside the `train` function, and inject the model instance into the function to train the model.

Inputted a model instance instantiated with the configured hyper-parameters, the `train` function outputs the best ModelSet, embedded in the form of an Experiment Object. By altering the hyper-parameters during the instantiation of a model class before injecting the instance into the `train` function, we could adjust the best model performance concerning the hyper-parameters.

2.4 Grid Search Implementation

A grid search process is a list of training processes. In a grid search process, two lists of candidate values that belong to two hyper-parameters to be tested are cartesian-produced, and generate a list of candidate value tuples.

$$\text{param}_a = \{a_1, a_2, a_3\} \tag{1}$$

$$\text{param}_b = \{b_1, b_2, b_3\} \tag{2}$$

$$\text{GridSearch} = \text{cart}(\text{param}_a, \text{param}_b) \tag{3}$$

$$= \{(a_1, b_1), (a_1, b_2), ..., (a_3, b_2), (a_3, b_3)\} \tag{4}$$

Using the list of hyper-parameter tuples as configurations, a list of model instances will be generated accordingly and trained individually into a list of Experiment Objects.

```
def grid_search(ModelClass,
                hyper_params, hyper_param_names,
                kwarg_names, **kwargs):
    param_exps = []
    n1, n2 = hyper_param_names
    kw1, kw2 = kwarg_names
    model_name = ModelClass.__name__

    # ... logic for retrieving X Y data and image augmentation

    for param1, param2 in hyper_params:
        # Dynamically add the hyperparameters to kwargs
        model_kwargs = {
            kw1: param1,
            kw2: param2
        }

        model_kwargs.update(kwargs)

        # Instantiate Model
        model_instance = ModelClass(**model_kwargs)

        # Experiment Object
        exp = train(model_instance, X, Y,
                    desc=f"{n1}-{param1}--{n2}-{param2}", n_fold=3, save=False)

        # Grid Search Object
        param_exps.append({
            n1: param1,
            n2: param2,
            "exp": exp
        })

    time_str = str(time.time()).replace(".", "")
    pickle.dump(param_exps, open(os.path.join(model_path,
        f"{model_name}_{n1}-{n2}_{time_str}.sav"), "wb"))
    return param_exps
```

Listing 9: Function to perform a grid search on a hyper parameter list.

In each iteration over the list of hyper-parameter tuples, the model will be instantiated with the model's class. To construct the model instance, we need to put the two hyper-parameters' official names and corresponding values into the signature dictionary as the "dynamic part". Moreover, some settled hyper-parameters could also be injected into the `grid_search` function through `**kwargs` as the "static part". Lastly, the two parts will be merged together to form the model's signature dictionary for this iteration and passed as the signature of the constructor function of the model's class. This technique ensures that the model instances are created with the desired configuration of the combination of hyper-paramers.

2.4.1 Grid Search Objects

A single `grid_search` function performs a single combination of two lists of hyper-parameters, and outputs a list of Grid Search Objects. A Grid Search Object has the following form.

```
{
    "hyper_param_name1": hyper_param_value1,
    "hyper_param_name2": hyper_param_value2,
    "exp": {
        "x": {
            "Best MSE": smallest_mse,
            "Best Fold": best_fold_idx,
            "Avg MSE": avg_mse,
            "model": ModelInstance,
        },
        "y": {
            "Best MSE": smallest_mse,
            "Best Fold": best_fold_idx,
            "Avg MSE": avg_mse,
            "model": ModelInstance,
        },
        "w": {
            "Best MSE": smallest_mse,
            "Best Fold": best_fold_idx,
            "Avg MSE": avg_mse,
            "model": ModelInstance,
        },
        "h": {
            "Best MSE": smallest_mse,
            "Best Fold": best_fold_idx,
            "Avg MSE": avg_mse,
            "model": ModelInstance,
        },
    }
}
```

Listing 10: Data structure of a single Grid Search Object.

The Grid Search Object contains the two hyper-parameters, along with the corresponding trained experiment object, embedding the ModelSet. At the end of each grid search, the `grid_search` function will store the Grid Search Object List in `.sav` file format.

2.5 Tests

Tests are performed on the loaded Grid Search Object List. From each Grid Search Object in the list, the `test` function retrieves the Experiment Object and evaluates the ModelSet using the Test dataset. The `test` function yields a Grid Search Result Object List, where individual elements is a Grid Search Result Object.

```
def test(exp_list, Y_test, X_test):
    results = []
    for i, exp in enumerate(exp_list):
        param_name1, param_name2, _ = exp.keys()
        param1, param2, models = list(exp.values())

        model_x, model_y, model_w, model_h = (models["x"]["model"],
                                              models["y"]["model"],
                                              models["w"]["model"],
                                              models["h"]["model"])

        y_x, y_y, y_w, y_h = (Y_test[:, 1], Y_test[:, 2],
                              Y_test[:, 3], Y_test[:, 4])

        y_pred_x, y_pred_y, y_pred_w, y_pred_h = (model_x.predict(X_test),
                                                  model_y.predict(X_test),
                                                  model_w.predict(X_test),
                                                  model_h.predict(X_test))

        mse_x, mse_y, mse_w, mse_h = (mean_squared_error(y_x, y_pred_x),
                                      mean_squared_error(y_y, y_pred_y),
                                      mean_squared_error(y_w, y_pred_w),
                                      mean_squared_error(y_h, y_pred_h))

        weighted_avg_mse = (mse_x + mse_y) * 0.3 + (mse_w + mse_h) * 0.2

        results.append({
            param_name1: param1,
            param_name2: param2,
            "weighted_avg_mse": weighted_avg_mse
        })
    return results
```

Listing 11: Function to perform tests on a Grid Search Object list.

2.5.1 Grid Search Result Objects

A Grid Search Result Object has the same structure as the Grid Search Object, except that the weighted average MSE of the ModelSet of the corresponding Grid Search Object replaces the Experiment Object.

```
{
    "hyper_param_name1": hyper_param_value1,
    "hyper_param_name2": hyper_param_value2,
    "weighted_avg_mse": w_avg_mse_value
}
```

Listing 12: Data structure of a single Grid Search Result Object.

2.5.2 MSE Summarization

A test on a single Grid Search Object will yield 4 MSE values, respectively from the four models in the embedded Experiment Object. We used the weighted average MSE to summarize the 4 values into a single unified evaluation metric. The weighted average is computed as follows.

$$\text{Given: } \{E_x, E_y, E_w, E_h\} \tag{5}$$

$$\text{Compute: } \text{avg}_E = \frac{3}{10}E_x + \frac{3}{10}E_y + \frac{2}{10}E_w + \frac{2}{10}E_h \tag{6}$$

This weighting arrangement puts a higher importance on the ellipse's center coordinate and a lower importance on the ellipse's size for two major reasons.

The first reason is that the size detection has a native lower accuracy due to its higher variance, creating a higher MSE than the coordinate values that may potentially dominate the summarized MSE.

The second reason is that the importance of accurately detecting the center coordinate outweighs that of accurately detecting the correct size.

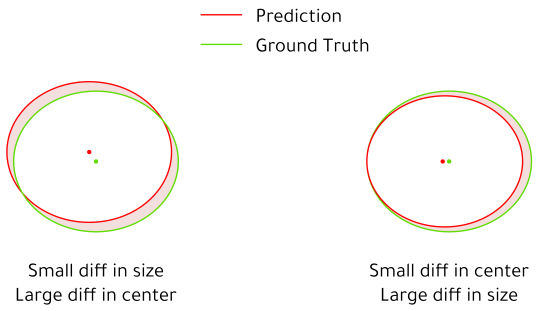

Figure 3: Effects on the overall detection performance of the ellipse's coordinate and size.

This difference is due to the fact that a significant bias in the center coordinate can lead to substantial damage in detection accuracy even if the size is accurate. In contrast, variations in size cause less overall damage in detection accuracy as long as the center coordinate remains accurate.

3 Grid Search Experiments

3.1 Model and Parameter Selection

3.1.1 Random Forest Regressor

RFR is hard to over-fit since the data is bootstrapped and the output comes from the models trained from the bootstrapped data, which could cover more patterns using more shallow trees. Moreover, we could cut the trees and allow some errors to make the model more robust facing new data.

Abbreviation	sklearn Parameter Name	Description
nest	n_estimators	Number of estimators in the random forest.
maxd	max_depth	Maximum depth of the trees in the random forest.
mins	min_samples_split	The Minimum number of samples number in the leaf that allows a leaf to be split again.
minl	min_samples_leaf	The minimum number of samples that a leaf requires to not be split again.

Table 1: Experimental hyper-parameters of the Random Forest Regressor.

We will be inspecting the effect of the hyper-parameters above on the training performance, measured by the mean squared error.

3.1.2 Support Vector Regression

Support Vector Regression (SVR) is based on the support vector machine. It is trained to determine a separating function that best predicts the numeric output based on the given input vector. We will be inspecting the effect of the hyper-parameters below on the training performance.

Abbreviation	sklearn Parameter Name	Description
krnl	kernel	Kernel of the non-linear SVM.
C	C	Trade-off factor between margin and wrong samples
epsl	epsilon	Margin of tolerance around predicted values.
gamm	gamma	Defines how far the influence of a single training example reaches.

Table 2: Experimental hyper-parameters of the Support Vector Regression.

3.2 Grid Search Results: Random Forest Regressor

3.2.1 Grid Search 0: Number of Estimators and Max Depth

maxd \ nest	10	20	30	40	50
11	1037.8667	1041.9625	1002.6387	979.8591	1003.4410
13	1117.2512	1131.6653	978.7854	1008.4079	1026.1018
15	1277.2783	1081.8425	1027.4105	995.2617	990.8895
17	1105.7404	1022.9589	997.9180	1008.4304	979.7051
19	1210.9086	1022.3115	977.6828	1009.1555	1065.0098

Table 3: Grid Search results of combination of `nest` and `maxd` in weighted average MSE.

We ranged the number of estimators from 10 to 50 with a step size of 10. The max depth ranges from 11 to 19, with a step size of 2.

The best configuration at this grid search is nest = 30 $\wedge$ maxd = 19, with the lowest overall MSE of 977.6828, with a difference of -66.3340 compared to the mean weighted average MSE of this grid search.

3.2.2 Grid Search 1: Min Samples Split and Min Samples Leaf

minl \ mins	4	6	8	10	12
6	1030.0320	1059.7984	923.2099	1013.3827	987.2580
8	1077.3436	1042.7518	1017.9481	1025.8023	995.0606
10	965.7390	968.3559	1001.0090	1052.0914	1096.0164
12	1001.5366	1010.3199	994.3514	1030.5991	969.2810
14	1104.5555	1069.8815	975.0766	1025.6006	981.2242

Table 4: Grid Search results of combination of `mins` and `minl` in weighted average MSE.

This Grid Search is performed based on the previous settled hyper-parameter values of nest = 30 $\wedge$ maxd = 19. The best model at this search is the best RFR.

The best configuration at this grid search is mins = 8 $\wedge$ minl = 6, with the lowest overall MSE of 923.2099, with a difference of -93.5191 compared to the mean weighted average MSE of this grid search.

3.3 Grid Search Results: Support Vector Regression

3.3.1 Grid Search 2: Kernel and C

krnl / C	linear	poly	rbf	sigmoid
1e-2	1581.1841	1880.1085	1939.4852	1940.3282
1e-1	1581.1841	1614.3857	1930.8324	1940.3444
1	1581.1841	923.7676	1845.9187	1940.4732
10	1581.1841	892.2570	1398.5448	1942.6019
100	1581.1841	1227.5168	788.4857	2446.5445

Table 5: Grid Search results of combination of `krnl` and `C` in weighted average MSE.

`scikit-learn` uses L_2 norm as default for soft margin where C is in the denominator in the kernel computation. Therefore, to ensure a more evenly distributed influence, an exponential distribution of parameter C is given.

The best configuration at this grid search is krnl = rbf $\wedge$ C = 100, with the lowest overall MSE of 788.4857, with a difference of -839.3900 compared to the mean weighted average MSE of this grid search.

3.3.2 Grid Search 3: Epsilon and Gamma

epsl / gamm	1e-2	5e-2	1e-1	5e-1	1
scale	788.6831	788.5911	788.4857	787.7619	786.8860
auto	1856.6190	1856.5331	1856.4609	1855.8475	1855.1163
1e-2	1856.6190	1856.5331	1856.4609	1855.8475	1855.1163
1e-1	1856.6190	1856.5331	1856.4609	1855.8475	1855.1163
1	1856.6190	1856.5331	1856.4609	1855.8475	1855.1163

Table 6: Grid Search results of combination of `epsl` and `gamm` in weighted average MSE.

This Grid Search is performed based on the previously settled hyper-parameter values of krnl = rbf $\wedge$ C = 100. The best model at this search is the best SVR.

The best configuration at this grid search is epsl = 1 $\wedge$ gamm = scale, with the lowest overall MSE of 786.8860, with a difference of -855.6274 compared to the mean weighted average MSE of this grid search.

4 Conclusion and Comments

4.1 Tests and Visualization

We conducted 2 grid searches on both RFR and SVR. The second grid search of each model is based on the optimized hyper-parameter configuration of the first grid search. Therefore, the best-performing model of the model type's second grid search is elected as the best-performing model of that specific model type.

4.1.1 Visualization: Inference Results of RFR

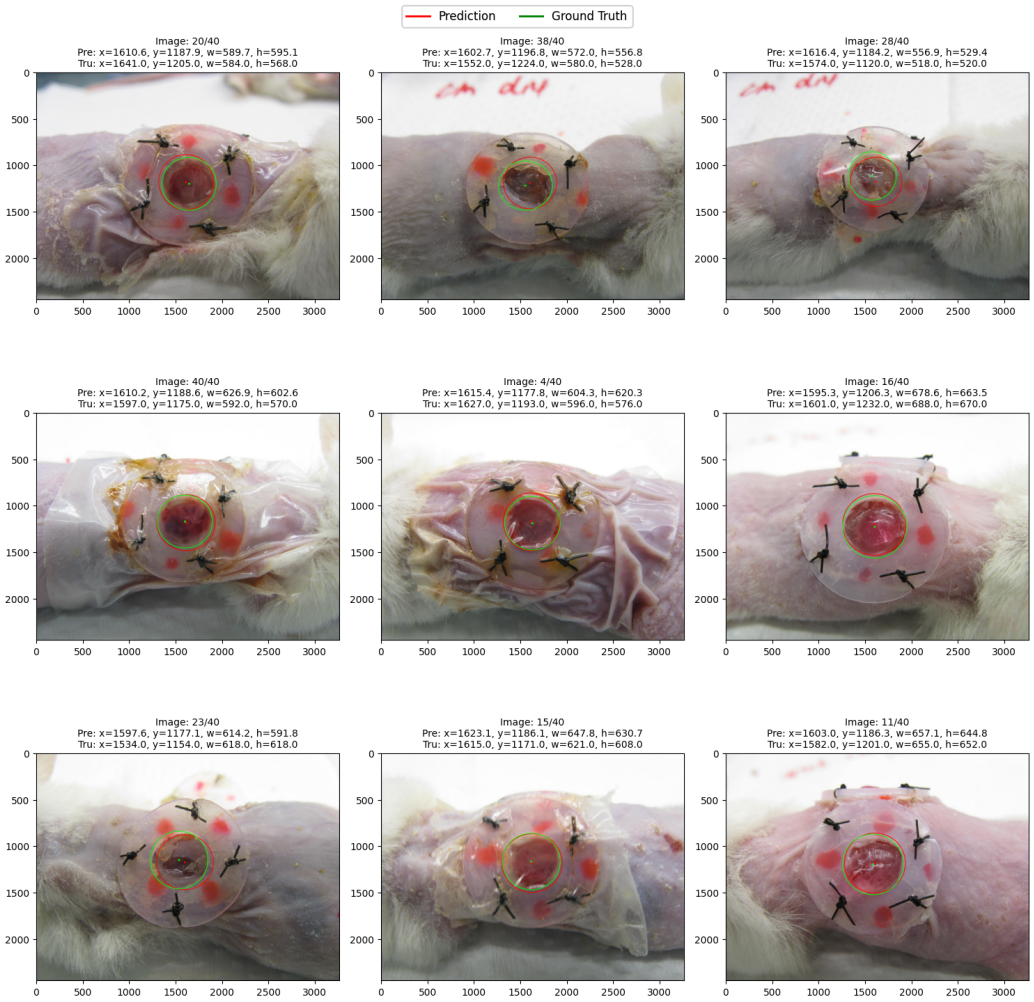

Figure 4: Visualization of the inference results of the random forest regressor.

The overall MSE of this best RFR is 923.2009.

4.1.2 Visualization: Inference Results of SVR

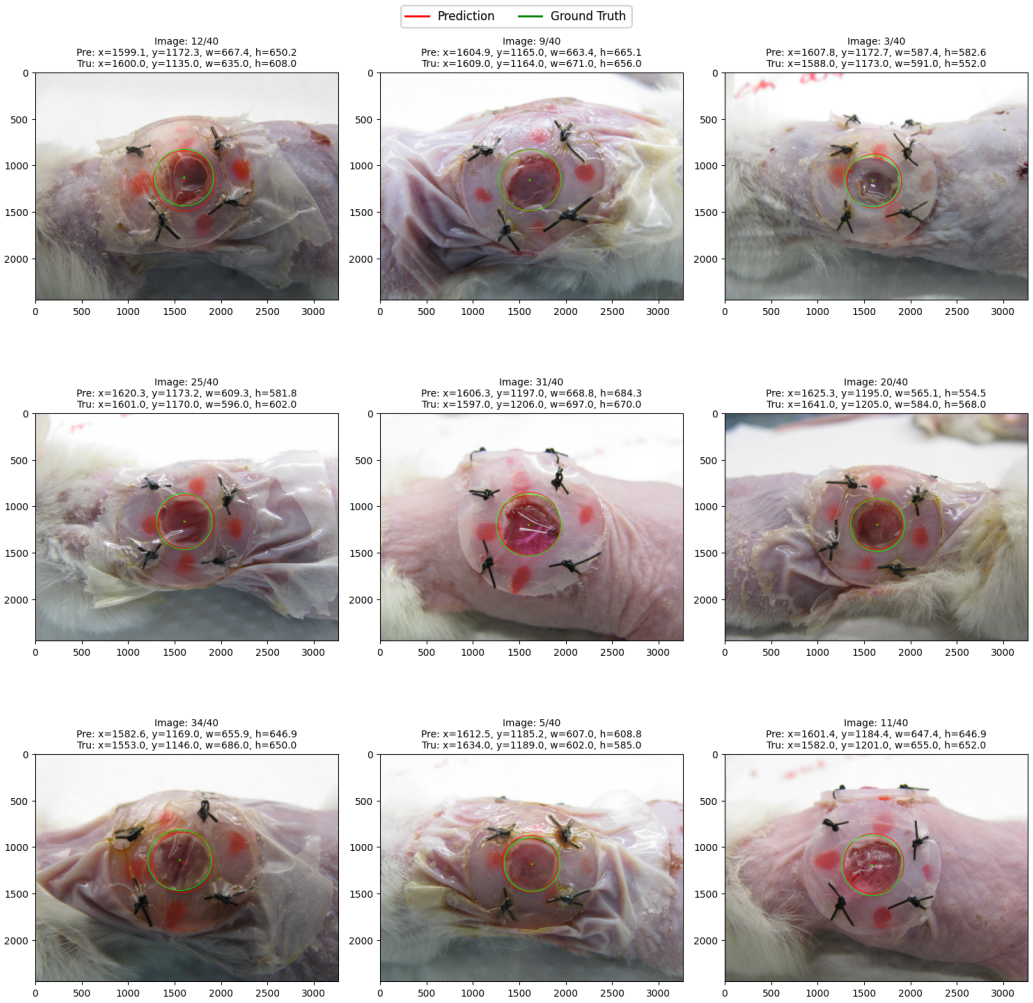

Figure 5: Visualization of the inference results of the SVR.

The overall MSE of this best SVR is 786.8860.

4.1.3 Analysis of Visualization Results

Overall, the two models both perform well in detecting wounds with clear edges, especially for the ones with central deep red areas that are radically different from their surroundings (9^{th} prediction of RFR and 2^{nd} prediction of SVR). However, for wounds with unclear edges (2^{nd}, 3^{rd} and 7^{th} prediction of RFR), the center of the ellipse is more biased. To detect such patterns more accurately, convolution is potentially needed to extract more abstract features of an image.

4.2 Model Comparison and Result Analysis

4.2.1 Random Forest Regressor

A single decision tree in the Random Forest Regressor is better at dealing with linear inseparable datasets compared to SVR due to its recursive nature, as it divides spaces into subspaces, creating regions with multiple decision boundaries. However, this nature may also cause the over-fitting problem. The most significant hyper-parameters regarding the training performance of the Random Forest Regressor model are the number of estimators and the max depth, as they control the degree of fitting.

Under max depth of 11 and 13, the training performance is stable as the number of estimators varies, considering a tree with an over-regulated depth may be under-fitted. Under max depth of 15 and 17, the performance improves from the overall perspective as the number of estimators increases, meaning that the bottleneck of the model's training performance is the number of estimators. For max depths of 19, an optimum is discovered at 30 estimators, giving a local best-fitted model under the given candidate values.

4.2.2 SVR

SVR, on the contrary, is more difficult to over-fit compared to RFR, but it is not as good as RFR at separating linear inseparable datasets, and it requires more effort to map data into linear separable spaces. The training performance of SVR is most influenced by the parameter of `krnl`, which stands for "Kernel", a technique used for non-linearly separable datasets.

Using a non-linear separation boundary for classification in a p-dimensional space is equivalent to using a linear separation boundary in a higher q-dimensional space ($q > p$), with a generalized objective function of the dual problem as follows (take hard-margin as an example).

$$\mathcal{G}(\alpha) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i \cdot \alpha_j) \cdot (y_i \cdot y_j) \cdot f(\mathbf{x}_i)^\top f(\mathbf{x}_j) + \sum_{i=1}^{N}\alpha_i$$

f is a mapping function that maps the data points $\mathbf{x}_i$ from lower dimensions into higher dimensions.

$$f : \mathbb{R}^p \mapsto \mathbb{R}^q,\ p < q \wedge p, q \in \mathbb{N}^+$$

A Kernel is a non-linear function that takes an input of $\mathbf{x}_i$ and $\mathbf{x}_j$ that calculates the value of $f(\mathbf{x}_i)^\top f(\mathbf{x}_j)$ using only $\mathbf{x}_i$ and $\mathbf{x}_j$, without calculating $f(\mathbf{x}_i)$ and $f(\mathbf{x}_j)$ explicitly, in order to save performance.

$$K(\mathbf{x}_i, \mathbf{x}_j) = f(\mathbf{x}_i)^\top f(\mathbf{x}_j)$$

SVR made use of these kernels, and it suggested four possible kernels as our choice, which are "linear", "poly", "rbf", and "sigmoid". A detailed listing of the target dimensions of these kernels is given below.

Kernel Type	**Kernel Formula**	**Mapping**
Linear	$K(\mathbf{x}_i\mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j$	$f : \mathbb{R}^d \mapsto \mathbb{R}^d$
Polynomial	$K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^\top \mathbf{x}_j)^p$	$f : \mathbb{R}^d \mapsto \mathbb{R}^{C_p^{d+p}}$
RBF	$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\sigma^2}}$	$f : \mathbb{R}^d \mapsto \mathbb{R}^\infty$
Sigmoid	$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\beta_0 \mathbf{x}_i^\top \mathbf{x}_j + \beta_1)$	$f : \mathbb{R}^d \mapsto \mathbb{R}^\infty$

Table 7: Detailed listing of candidate values of hyper-parameter `kernel` of the SVR.

The linear kernel yields the original space of input data. Considering the model performance being constant under the Linear kernel for all the values of parameter C, the original input data is highly non linearly separable. This is very common when the inputs are images. Polynomial kernels that could map the original data into a higher yet finite dimension solve some linear inseparability, having a minimum MSE of 892.2570 at the parameter C of 10. The RBF kernel under $C = 100$ performs the best, with the lowest MSE of 788.4857 under the parameter C of 100, mapping original data into infinite dimensions.

RBF stands for "Radial basis function", producing a radial decision boundary in the original space. Under the kernel of RBF, the performance improves as the parameter C increases exponentially. As parameter C increases from 10 to 100, the training performance is significantly increased. Therefore, we could predict that RBF maps the original data into the most proper space and that the mapped data has a high linear separability. Moreover, the model could also be still underfitted since the improving trend seems to continue, while our current optimum parameter C is the largest among all the candidate values and it already regulates a small amount of error.

(END OF REPORT)