CMSE381_Project

April 25, 2025

1 CMSE 381 Final Project

1.0.1 Group members: Tyler Lehman

1.0.2 Section_002

4/23/2025

2 *Classifying Surfaces Through Classification Modeling*

2.1 Background and Motivation

Classification and Regression models can be a very powerful tool, and are very helpful in predicting outputs. Putting together all of the methods we have learned in class, we now have an understanding what running models like this does, and how to tweak it, etc, to find the best parameters and predictions.

Working with the Default dataset, it consists of 3 types of concrete: Decks, Walls, and Pavements. There are then 2 subcategories of each of these classifications, and that's if each surface is Cracked or Non-cracked. Every JPG in the Default dataset will fall into each of these categories, and I will be running a classification model and decision tree model to determine this. There will be six categories: Cracked Decks, Non-cracked Decks, Cracked Walls, Non-cracked Walls, Cracked Pavements, and Non-cracked Pavements.

The questions I am looking to answer in this project are: - Can I build a classification model to correctly tell cracked surfaces vs non-cracked?

- What is the best method to accomplish this? I will be choosing to test an Support Vector Classifier (SVC) and a Decision Tree model.
- Can the model be improved through techniques like feature selection or cross-validation tuning?

3 Data & Imports

The dataset I am using is the Default dataset, which consists of over 56,000 images of cracked and non-cracked concrete bridge decks, walls, and pavements. The dataset also includes images with a variety of obstructions, including shadows, surface roughness, scaling, edges, holes, and background debris.

Each image is segmented into 256 × 256 pixel sub-images. Each sub-image is labelled as 'Cracked' if there was a crack in the sub-image or 'Non-cracked' if there was not a crack.

More details about the data can be found here: https://www.kaggle.com/datasets/aniruddhsharma/structural-defects-network-concrete-crack-images/data

[45]:
```
# Imports for loading in the images
import os
from skimage.io import imread
from skimage.transform import resize

# SVC Imports
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Decision Tree Imports
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
```

4 Models for Classification

(What models will you be using for classification? Why did you choose to use them? What questions would you answer with them? How would you evaluate if each model? What cross-validation method did you use?)

The models I chose to do are a **Support Vector Classifier** and **Decision Tree Model**. An SVC model is useful here because it works well with high dimension data and has an RBF kernel to clasify images. Decision Trees are very easy to read and implement, but suffer a little bit in accuracy, so we'll see how this goes.

The questions I need to answer with these models are if I build these classification models, will they accurately classify cracked surfaces vs non-cracked? Which is a better method for this? SVC or Decision Tree? And last, can the model be improved by tweaking parameters such as feature selection or cross-validation? I will be evaluating how these models perform by accuracy scores, confusion matrices, and plots, and then performing best parameter findings by doing 5-fold cross validations. I chose 5-fold because it was the number that ran in a reasonable time for my laptop.

5 Methodology

First, I had to load in all of the images. This is different than just loading in a dataset from a csv, so I had to do other methods. I found a YouTube video that helped with this, as well as use of ChatGPT, and my sources are cited in the References section.

[5]:
```
# Code to find the folder that is the Default dataset. It's called archive (5)␣
↪on my machine.
base_dir = 'C:/Users/pgleh/Downloads/archive (5)'
```

```
# Creating categories and labels that my model will use and classify images␣
 ↪into.
categories = ['Walls', 'Pavements', 'Decks']
labels = ['Cracked', 'Non-cracked']

# data will store the pixel values and target will store corresponding labels␣
 ↪for each image.
data = []
target = []

# Looping through the folders and each subfolder.
for category in categories:
    for label in labels:
        # Creates the path to each folder and subfolder so I can access them.
        folder_path = os.path.join(base_dir, category, label)

        # Creates a path to each file so I can access them.
        for file in os.listdir(folder_path):
            img_path = os.path.join(folder_path, file)

            # Skip non-image files. This is here so it doesn't try to add a␣
 ↪folder or subfolder to the dataset I'm making.
            # I want to add only images, and was getting errors about adding␣
 ↪folders to the dataset.
            if not file.lower().endswith(('.png', '.jpg', '.jpeg')):
                continue

            try:
                # Reading the images and converting them to np arrays.
                img = imread(img_path)
                # Resizing all of the images to 64x64 pixels.
                img_resized = resize(img, (64, 64), anti_aliasing=True)
                # Adding the now flattened pixels to the dataset I will use in␣
 ↪Python.
                data.append(img_resized.flatten())
                # Storing the labels for each image.
                target.append((material, label))

            # For errors; if there are any.
            except Exception as e:
                print(f"Skipping {img_path}: {e}")
```

Next I had to convert tuples that were being made to strings. When I converted the images to arrays in the last block, the labels were turned into tuples, so I turned them into strings so they had one name, which makes it easier to loop through and work with.

In this step I also split my data into training and testing sets. This is important to do when building any model, because I first have to train and make the model learn what I want it to do. Testing is also important too, since this is were I measure how well the model does.

You will see I named my splits _full; this is for testing purposes. Since the Default dataset has over 56,000 images, I found working with only 2000 of the total worked best for my laptop, and kept reasonable runtimes.

[122]:
```
# The if statement is here because when I messed up coding in the future, or␣
↪wanted to change something, I wouldn't have to rerun the line that actually
# converts tuples to strings.
if isinstance(target[0], tuple):
    target = [f"{category}_{label}" for (category, label) in target]

# Train/test split
x_train_full, x_test_full, y_train_full, y_test_full = train_test_split(data,␣
↪target, test_size = 0.2, shuffle = True, stratify = target)

# Taking 2,000 of the 56,000 for runtime purposes.
x_train = x_train_full[:2000]
y_train = y_train_full[:2000]
```

5.0.1 Moving on, to actually creating the model and fitting it! First up, is my Support Vector Classifier model.

[19]:
```
svc = SVC()

# Creating an amount of parameters to test and go through. Again, for runtime␣
↪purposes, I found 12 different combinations was ok to run through.
params = [{'gamma': [0.01, 0.001, 0.0001], 'C': [1, 10, 100, 1000]}]
# This trains 12 image classifiers, going through gamma and C params.

# Cross-validation technique for my SVC.
grid_search = GridSearchCV(svc, params, n_jobs = -1)
# Fitting the training data to the model.
grid_search.fit(x_train, y_train)
```

[19]:
```
GridSearchCV(estimator=SVC(), n_jobs=-1,
             param_grid=[{'C': [1, 10, 100, 1000],
                          'gamma': [0.01, 0.001, 0.0001]}])
```

Once I had the model fit, I could look for the best paramters/estimators.

[29]:
```
best_estimator = grid_search.best_estimator_
best_estimator

# Best params:
# C = 10
# gamma = 0.01
```

```
[29]: SVC(C=10, gamma=0.01)
```

With the best parameters now selected, I can now test to see the accuracy of the model.

```
[75]: # Making my test sets the same size as my training sets were.
x_test = x_test_full[:2000]
y_test = y_test_full[:2000]

# Predicting the y values.
y_pred = best_estimator.predict(x_test)

score = accuracy_score(y_pred, y_test)
score
```

```
[75]: 0.68
```

5.0.2 SVC Model Improving

While my best fit model found best parameters, I wanted to check and see if they were actually the best fits. My method to figure this out was using K-Fold Cross-Validations, which will split up the data for training and testing multiple times to figure out what parameters are best for me.

```
[77]: # I found Pipeline helps make feature selection run faster when I looked for␣
 ↪ways to speed up my code. Source info in References section.
# This code snippet helps run through the code faster and will select the best␣
 ↪k from my SVC.
pipeline = Pipeline([
    ('select', SelectKBest(score_func = f_classif)),  # Don't set k yet until␣
 ↪GridSearchCV.
    ('svc', SVC())])

params = {
    'select__k': [100, 300, 500, 1000, 2000],
    'svc__C': [1, 10, 1000, 1000],
    'svc__gamma': [0.0001, 0.001, 0.01]
}

# Fitting the model now to see what parameters are best.
grid_search = GridSearchCV(pipeline, params, cv = 5, n_jobs = -1)
grid_search.fit(x_train, y_train)

print("Best params:", grid_search.best_params_)
print("Best accuracy:", grid_search.best_score_)
```

```
Best params: {'select__k': 2000, 'svc__C': 1, 'svc__gamma': 0.01}
Best accuracy: 0.54
```

5.0.3 It's usually best practice to double check you tests and results, as well as models. Maybe a Support Vector Classifier isn't the best option for this project? So I decided to do a Decision Tree model, mainly since it will be faster then an SVC. It should also be easier for me to code and interpret too.

The first step for this is creating my Decision Tree instance and fit it.

```
[47]: tree = DecisionTreeClassifier(random_state = 42)
      tree.fit(x_train, y_train)
```

```
[47]: DecisionTreeClassifier(random_state=42)
```

In a much faster process, this is already the predicting and accuracy score step. Decision Trees are compatible with the accuracy_score function as well as classification_report, allowing for a better look at the results.

```
[49]: y_pred_tree = tree.predict(x_test)

      print(accuracy_score(y_test, y_pred_tree))
      print(classification_report(y_test, y_pred_tree))
```

```
0.56
                      precision    recall  f1-score   support

       Decks_Cracked       0.09      0.09      0.09        23
   Decks_Non-cracked       0.48      0.61      0.54        92
   Pavements_Cracked       0.14      0.15      0.14        20
Pavements_Non-cracked       0.74      0.75      0.75       183
       Walls_Cracked       0.27      0.24      0.25        38
   Walls_Non-cracked       0.60      0.51      0.55       144

            accuracy                           0.56       500
           macro avg       0.39      0.39      0.39       500
        weighted avg       0.56      0.56      0.56       500
```

Decision Tree Improving

```
[58]: # Decision Trees are compatible with cross_val_score, whcih will automatically␣
       ↪split up my data for k-folds.
      from sklearn.model_selection import cross_val_score

      cv_scores = cross_val_score(tree, x_train, y_train, cv = 5)

      print("Scores per Fold:", cv_scores)
      print("Mean CV accuracy:", np.mean(cv_scores))
```

```
Scores per Fold: [0.52   0.5675 0.515  0.535  0.5425]
Mean CV accuracy: 0.536
```

6 Results

(What did you find when you carried out your methods? Some of your code related to presenting results/figures/data may be replicated from the methods section or may only be present in this section. All of the plots that you plan on using for your presentation should be present in this section)

Now it's time to see exactly how the models performed, as well as what is the best parameters for the models I created.

The below code is from the Support Vector Classifier before I ran the K-Fold CV to find the best parameters.

```
[146]: score
```

```
[146]: 0.68
```

This means the SVC an accuracy of 68%. That's not the best, but not bad either, it's a passing grade. This makes sense because I only used 2,000 out of the 56,000 images from the Default set. That's only about 3.5% of the original dataset, but I had to do this for time constraintas. Let's see what the best parameters are exactly for the SVC and how they performed:

```
[151]: print("Best params:", grid_search.best_params_)
       print("Best accuracy:", grid_search.best_score_)
```

```
Best params: {'select__k': 2000, 'svc__C': 1, 'svc__gamma': 0.01}
Best accuracy: 0.54
```

An accuracy of 0.54 is less than 0.68, even though this used the best parameters. This could be because the first GridSearch I ran was on the full 2000 subset, while the K-Fold will split up the data multiple times to run it's models. If you have less data, then yes, you will probably be less accurate since it doesn't truly represent the population.

Lets see how the Decision Tree did too:

```
[154]: print(accuracy_score(y_test, y_pred_tree))
       print(classification_report(y_test, y_pred_tree))
```

```
0.56
                       precision    recall  f1-score   support

        Decks_Cracked       0.09      0.09      0.09        23
    Decks_Non-cracked       0.48      0.61      0.54        92
    Pavements_Cracked       0.14      0.15      0.14        20
Pavements_Non-cracked       0.74      0.75      0.75       183
        Walls_Cracked       0.27      0.24      0.25        38
    Walls_Non-cracked       0.60      0.51      0.55       144

             accuracy                           0.56       500
            macro avg       0.39      0.39      0.39       500
         weighted avg       0.56      0.56      0.56       500
```

The Decision Tree has a lower score than the SVC, which makes sense. Decision Trees sacrifice accuracy for speed and interpretability, they are not the most accurate. 0.56 accuracy is pretty bad too, as it barely gets half of the data correct. Again, maybe the score would be improved on the full 56,000 images. What's interesting too is that the model predicts Non-cracked surfaces a lot more accurately than cracked ones. Maybe because it can just see a surface with minimal changes and calls it good.

[164]:
```python
# k values were: [100, 300, 500, 1000, 2000]
print("Scores per Fold:", cv_scores)
print("Mean CV accuracy:", np.mean(cv_scores))
```

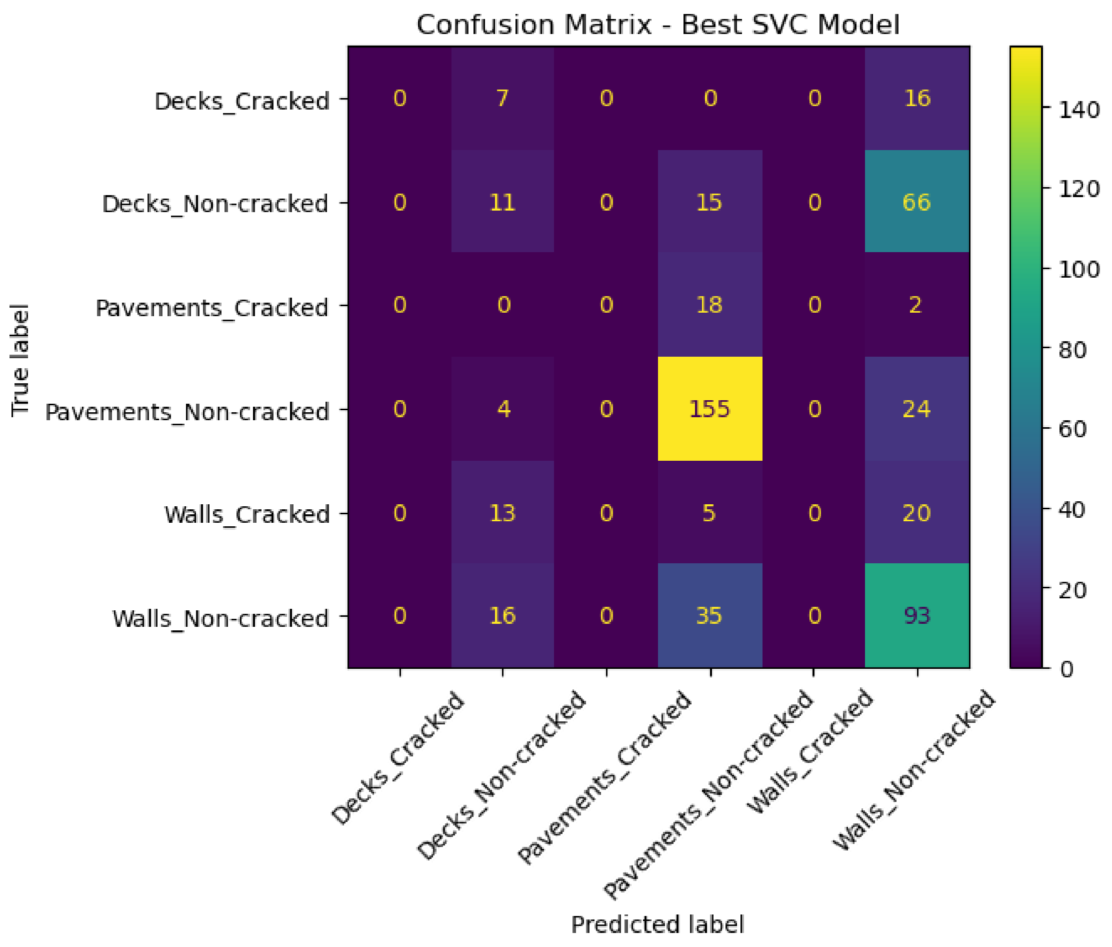
```
Scores per Fold: [0.52   0.5675 0.515  0.535  0.5425]
Mean CV accuracy: 0.536
```

Again, the accuracy drops when doing K-Fold CV, interesting. Using the 2,000 subset, the score was 0.5425, still less than the total, but a k=300 produced the highest score here. Maybe this fold had more Non-cracked surfaces and was therefore able to predict better. These accuracies are still not very good and definitely need more improvements.

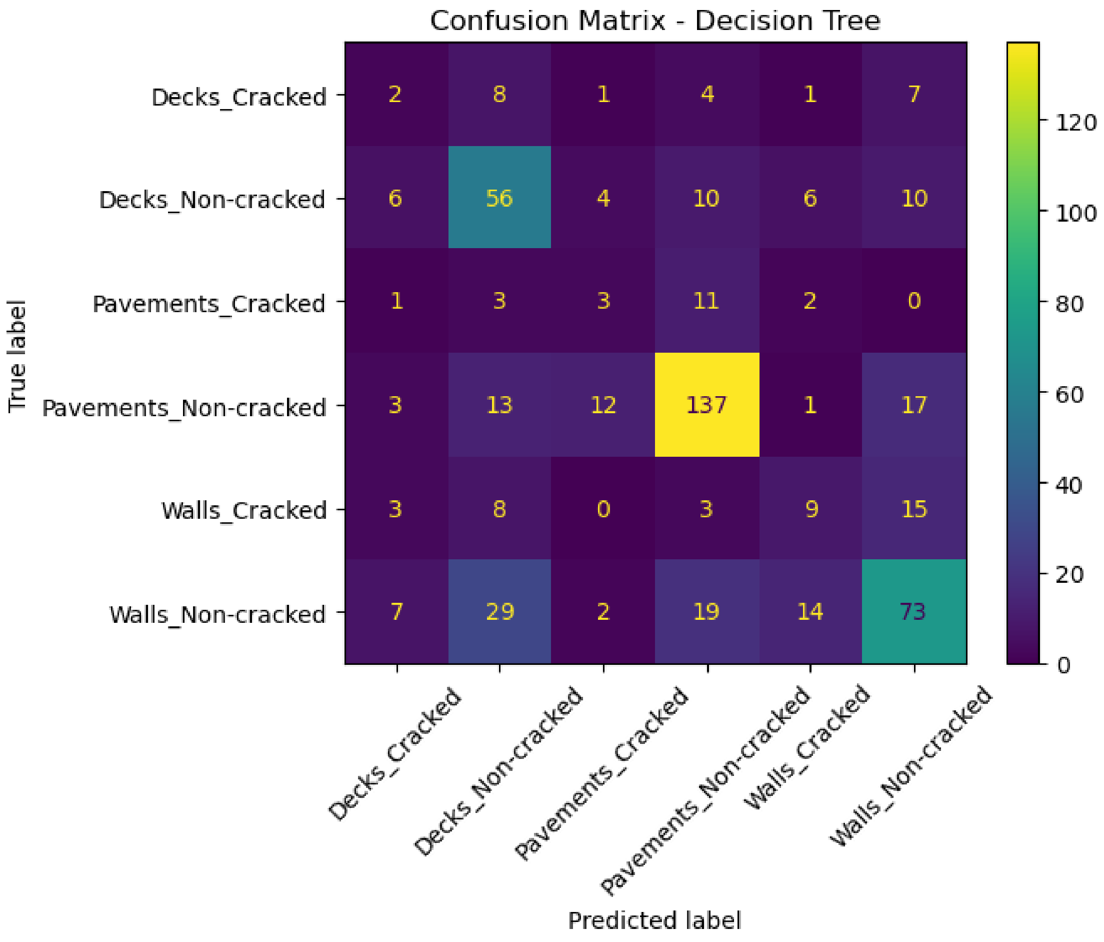
6.0.1 Everybody likes visualizations, lets get some in here to help understand more what went right and wrong here.

Confusion Matrices are very helpful and easy to interpret for seeing how variables interact with each other. In this case, I used one to see exactly how a sample of the model did predicting the labels of these surfaces. First up is the SVC, and you can see that it is not the most accurate, which lines up with the result numbers above. Same thing with the Decision Tree. Again, the Non-cracked surfaces are more often predicted correctly, except for 1 instance in the Decision Tree for Pavements, this is most likely just an outlier.

[62]:
```python
# Imports for making a Confusion Matrix.
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# SVC Confusion Matrix

# Will be comparing predicted labels vs actual labels, and making sure each␣
 ↪label is unique to unique points.
cm = confusion_matrix(y_test, y_pred, labels = np.unique(y_test))
# Creating the actual display.
disp = ConfusionMatrixDisplay(confusion_matrix = cm, display_labels = np.
 ↪unique(y_test))
# Tilts the axis labels so they aren't sideways.
disp.plot(xticks_rotation = 45)
plt.title("Confusion Matrix - SVC Model")
plt.show()

# Decision Tree Confusion Matrix

# Same process as above.
cm_tree = confusion_matrix(y_test, y_pred_tree, labels = np.unique(y_test))
disp_tree = ConfusionMatrixDisplay(confusion_matrix = cm_tree, display_labels =␣
  ↪np.unique(y_test))
disp_tree.plot(xticks_rotation = 45)
plt.title("Confusion Matrix - Decision Tree")
plt.show()
```



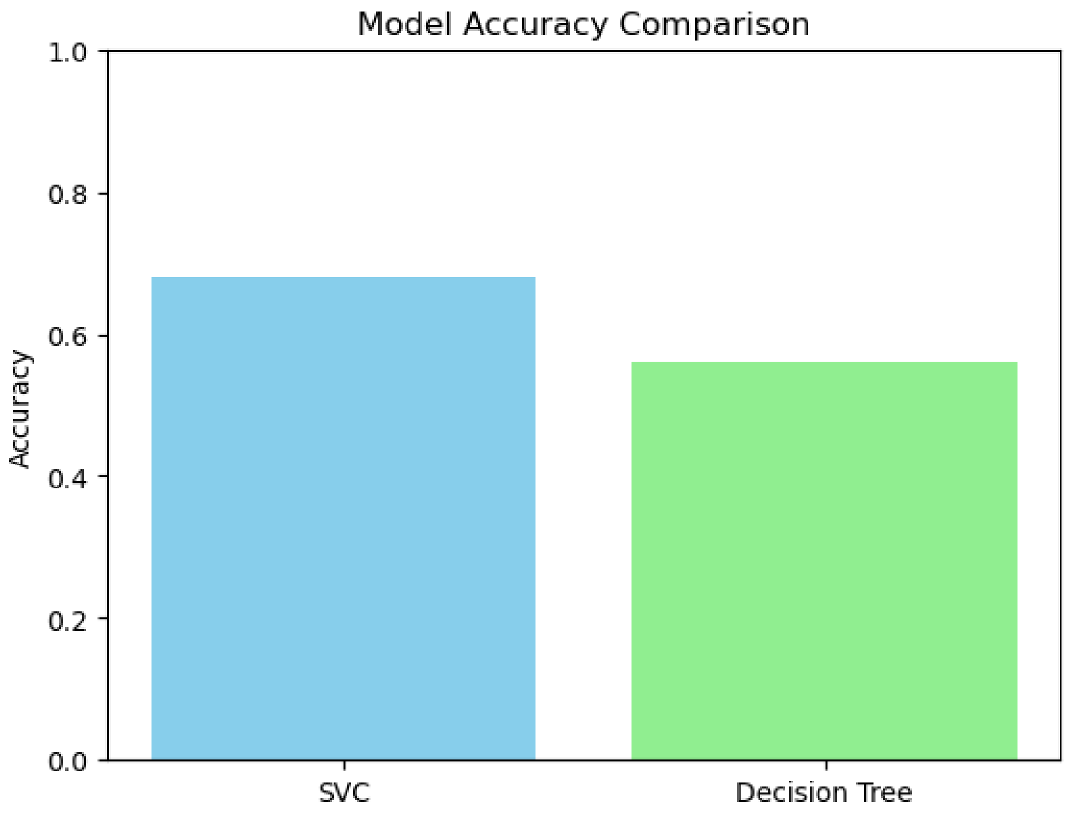
	Decks_Cracked	Decks_Non-cracked	Pavements_Cracked	Pavements_Non-cracked	Walls_Cracked	Walls_Non-cracked
Decks_Cracked	0	7	0	0	0	16
Decks_Non-cracked	0	11	0	15	0	66
Pavements_Cracked	0	0	0	18	0	2
Pavements_Non-cracked	0	4	0	155	0	24
Walls_Cracked	0	13	0	5	0	20
Walls_Non-cracked	0	16	0	35	0	93



To better help visualize what the accuracy of the two models:

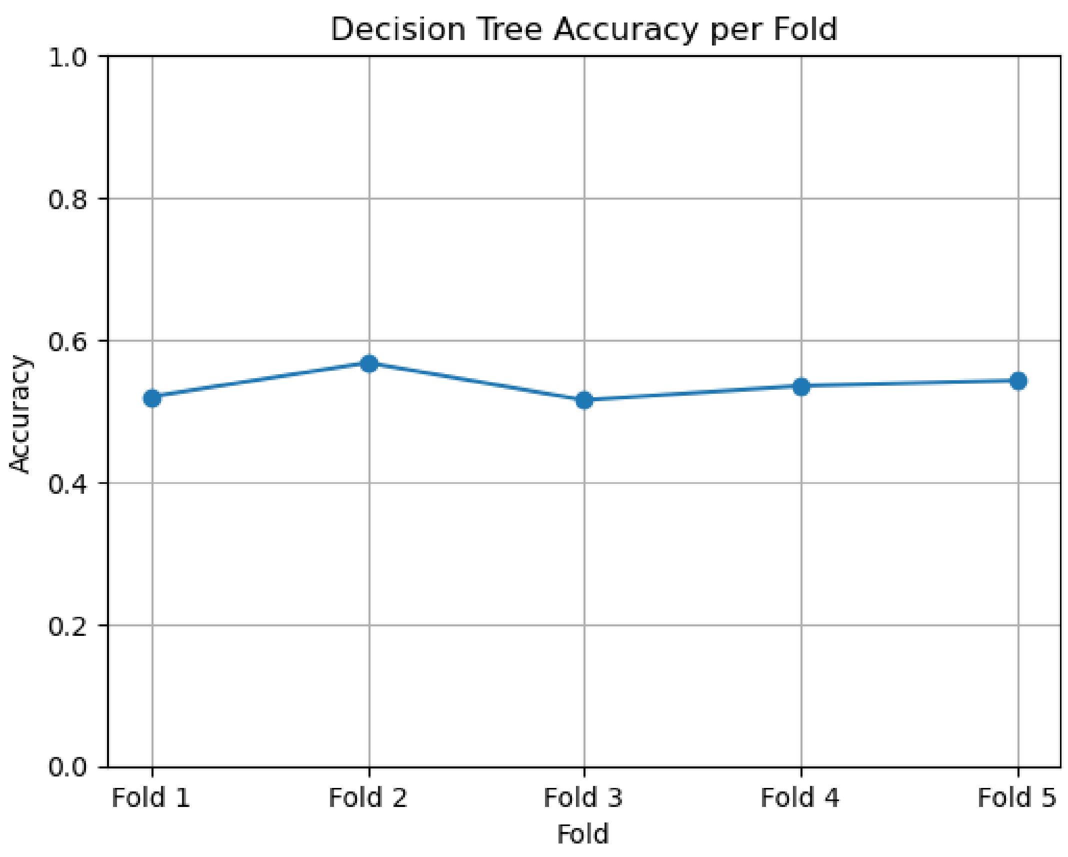
Not very good, as they both are under classifying correctly at a 75% rate. But, as you can see the SVC does have a reasonable gap in accuracy on the Decision Tree, which is to be expected.

[170]:
```
models = ['SVC', 'Decision Tree']
accuracies = [score, accuracy_score(y_test, y_pred_tree)]

plt.bar(models, accuracies, color = ['skyblue', 'lightgreen'])
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.ylim(0, 1)
plt.show()
```

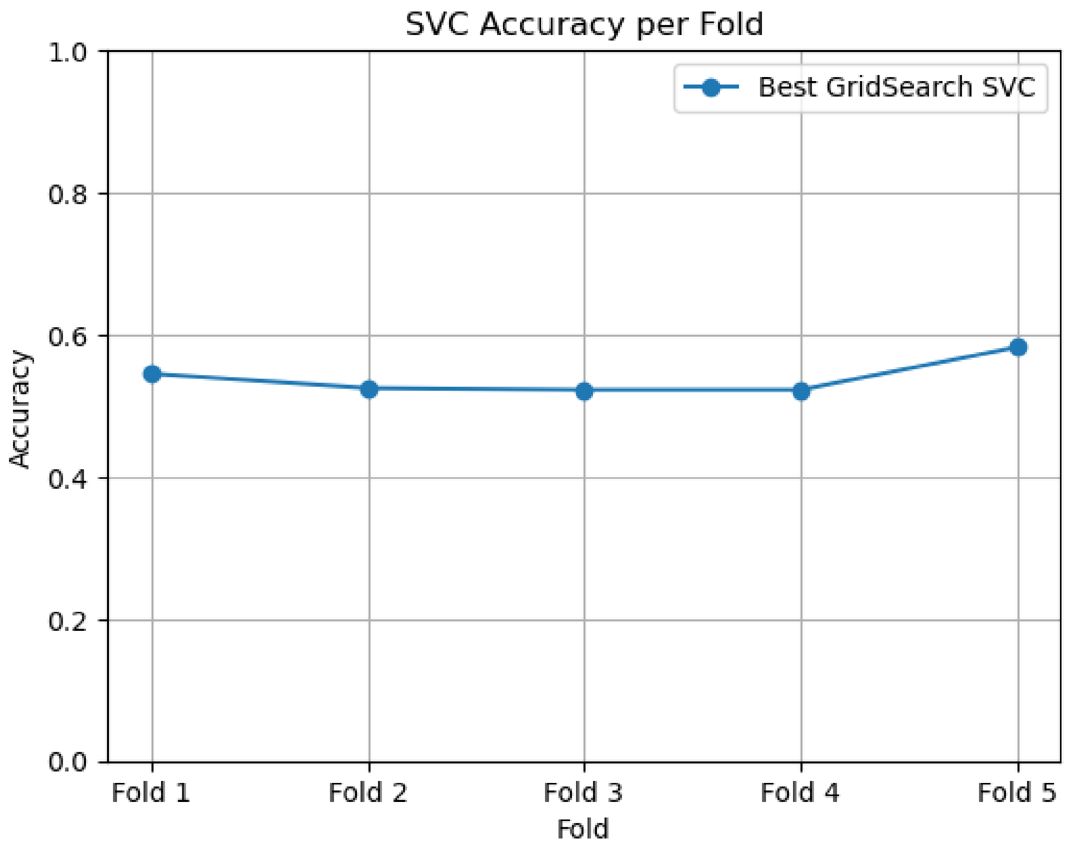


And like I said previously, using the K-Folds didn't improve the accuracy either, as you can see below, the accuracy barely changed per fold.

[68]:
```
folds = [f"Fold {i+1}" for i in range(len(cv_scores))]

plt.plot(folds, cv_scores, marker = 'o')
plt.title("Decision Tree Accuracy per Fold")
plt.ylabel("Accuracy")
plt.xlabel("Fold")
plt.ylim(0, 1)
plt.grid(True)
plt.show()
```



[178]:
```
pipeline = Pipeline([
    ('select', SelectKBest(score_func = f_classif, k = 2000)),
    ('svc', SVC(C = 1, gamma = 0.01))
])

cv_scores = cross_val_score(pipeline, x_train, y_train, cv = 5)

folds = [f"Fold {i+1}" for i in range(len(cv_scores))]

plt.plot(folds, cv_scores, marker = 'o', label = 'Best GridSearch SVC')
plt.title("SVC Accuracy per Fold")
plt.ylabel("Accuracy")
plt.xlabel("Fold")
plt.ylim(0, 1)
plt.grid(True)
plt.legend()
plt.show()
```



7 Conclusion

The models I created here do work, but not accurately enough. I learned there is still a lot of work that could be done to create a more efficient and accurate classification, SVC or Decision Tree wise. I also learned using K-Folds won't always give you more accuracy; that's what I thought before I started this, but that's not true. You can see in the two plots above, the accuracy was kind of random. Obviously you want a model with 100% accuracy, but that is most likely unrealistic.

Next time I will definitely use a higher proportion of the dataset, as 3.5% is not much at all, but that's what I had to do to get my code to run in a reasonable amount of time. Using as much of the data as possible would be best, as it's more representative of the population, and I would guess more accurate in classifying.

I would say the SVC model was successfully built to classify cracked vs non-cracked surfaces, I just didn't use enough data. If my SVC can be at 68% accuracy on only 3.5% of the whole dataset, I think the accuracy could very well improve to above 75%, which I would consider a good model. The SVC was also clearly a better choice, as comparing their best accuracies of 0.68 vs 0.56 at their highest, the SVC has a

12% better classification than the tree. It turns out, K-Fold CV isn't always going to improve the model, as it almost always decreased the accuracy score for my tests. Overall, next time I might try a regression model, to see if feature selection there can result in the best model for predicting, hopefully at 75% or better.

7.1 Author Contribution

I worked solo on this project, so everything you see here was done by Tyler Lehman.

7.2 References

"8.3. Parallelism, Resource Management, and Configuration." Scikit, scikit-learn.org/stable/computing/parallelism.html. Accessed 25 Apr. 2025.

Chatgpt | Openai, openai.com/index/chatgpt/. Accessed 25 Apr. 2025.

Computer vision engineer. "Image Classification with Python and Scikit Learn | Computer Vision Tutorial." YouTube, YouTube, www.youtube.com/watch?v=il8dMDlXrIE. Accessed 25 Apr. 2025.

"Pipeline." Scikit, scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html. Accessed 25 Apr. 2025.

Whoami-As. "Structural Defects Network (SDNET) 2018." Kaggle, 28 July 2020, www.kaggle.com/datasets/aniruddhsharma/structural-defects-network-concrete-crack-images/data.

I also used multiple of the inclass assignments for reference, including K-Fold CV and Classification lessons, Decision Trees, and SVC lessons.

```
[ ]:
```