Automated Parking Space Detection: A Real-World Need

This project is finding a parking spot in crowded cities is a daily struggle for millions of drivers. It is limited space, growing vehicle ownership, and inefficient parking management wastes time, fuel, and energy while adding to congestion and stress. An automated parking detection system offers a smart, practical solution.

By using cameras placed around parking lots, the system captures images of each space. A machine learning model such as the Support Vector Machine (SVM) classifier from the provided code then analyzes these images to detect whether a space is empty or occupied.

This real-time data can be shown on digital boards, integrated into navigation apps, or shared through parking management platforms. The impact is significant: drivers find parking faster, traffic within lots is reduced, and fuel consumption drops, lowering emissions. For operators, these systems improve space utilization, generate valuable usage insights, and reduce reliance on constant human monitoring.

In short, automated parking detection goes beyond convenience and it is a vital step toward smarter cities, smoother traffic, and a more sustainable urban future.

Code:

This code trains a machine learning model to recognize whether a parking space is empty or occupied by analyzing images. It works in several steps:

- First, it reads images from two labeled folders—one for empty spaces and one for occupied spaces.
- The images are resized and converted into numerical data that the model can understand.
- The dataset is then split into training and testing sets.
- A Support Vector Machine (SVM) classifier is trained, with Grid Search used to find the best parameters.
- The model’s performance is tested, and its accuracy is displayed.
- Finally, the trained model is saved so it can be reused later without retraining.

Step-by-step explanation

1. Importing libraries

```
import os
import pickle

from skimage.io import imread
from skimage.transform import resize
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
```

os lets us move around folders and find image files.

pickle lets us save the trained model, so I don't have to train it again later.

imread reads images from the computer into Python.

resize shrinks or stretches images to the same size so they can be compared.

NumPy (np) stores image data in fast numerical arrays.

train_test_split splits our data into training and testing sets.

GridSearchCV tests different model settings to find the best ones.

SVC is the machine learning model (Support Vector Machine).

accuracy_score tells us how many predictions the model got correct

These libraries are the building blocks of the pipeline. Without them, you couldn't load data, train models, or measure performance

---

2. Setting up the dataset

```
input_dir = 'C:/Users/RyanK/.../clf-data'
categories = ['empty', 'not_empty']
```

- **`input_dir`** is the folder containing two subfolders: one for empty spots, one for occupied spots.

- **`categories`** stores the two possible labels.

I tell the program where the images are stored on the computer. I also define two categories: empty and not_empty, which represent parking spaces.

---

3. Reading and processing images

```
data = []
labels = []
for category_idx, category in enumerate(categories):
    for file in os.listdir(os.path.join(input_dir, category)):
        img_path = os.path.join(input_dir, category, file)
        img = imread(img_path)                    # Read image
        img = resize(img, (15, 15))                # Resize to 15×15
pixels
        data.append(img.flatten())                 # Flatten into 1D
vector
        labels.append(category_idx)                # 0 for empty, 1
for not_empty
```

I make two empty lists: one for the images, one for their labels.
Then I go through each category's folder:

- Each image is opened, resized to **15×15 pixels**, and flattened into a long row of numbers.

- The row of numbers is added to our **data list**.

- At the same time, I give the image a label: **0 for empty** and **1 for not_empty**, and store that in the **labels list**.

This step turns folders of pictures into numbers that the computer can understand.

---

4. Converting to NumPy arrays

```
data = np.asarray(data)
labels = np.asarray(labels)
```

I change the lists into NumPy arrays, which are faster and work well with scikit-learn.

---

5. Splitting into training and test sets

```
x_train, x_test, y_train, y_test = train_test_split(
    data, labels, test_size=0.2, shuffle=True, stratify=labels
)
```

- `test_size=0.2` → 20% of images go into the test set, 80% into training.
- `shuffle=True` → Randomizes order.
- `stratify=labels` → Ensures the train/test split keeps the same ratio of empty vs not_empty.

I divide the dataset:

- **80% of the images** are used for training the model.
- **20% of the images** are saved aside to test the model later.
  I also shuffle the data to make it random and keep the ratio of empty vs not_empty balanced.

This way, I can check if the model works well on new data it hasn't seen before.

---

6. Training the classifier

```
classifier = SVC()
parameters = [{'gamma': [0.01, 0.001, 0.0001], 'C': [1, 10, 100,
1000]}]
grid_search = GridSearchCV(classifier, parameters)
grid_search.fit(x_train, y_train)
```

- Creates an SVM model (`SVC()`).
- Defines **parameter ranges** for: `gamma`: how far the influence of a single training example reaches. `C`: How strict the model is about avoiding misclassification.
- `GridSearchCV` → tests every combination of these parameters to find the best one.
- `.fit()` trains the models and selects the best.

I create an SVM model. I then tell the computer to try different settings: **Gamma** controls how far the influence of one image spreads. **C** controls how strict the model is about avoiding mistakes. Grid Search tests all the possible combinations of these settings and picks the best one

---

7. Evaluating the model

```
best_estimator = grid_search.best_estimator_
y_prediction = best_estimator.predict(x_test)
score = accuracy_score(y_prediction, y_test)
print('{}% of samples were correctly classified'.format(str(score *
100)))
```

- `best_estimator` → gets the best model from the grid search.
- Predicts labels for the **test set**.
- Calculates **accuracy**.
- Prints the percentage of correctly classified images.

Once trained, the best model is chosen. It predicts whether the test images are empty or not_empty. I calculate the accuracy by comparing predictions with the real answers. Finally, I print out the percentage of correct classifications.

---

8. Saving the model

```
pickle.dump(best_estimator, open('./model.p', 'wb'))
```

The best model is saved into a file called **model.p**.This means next time I can just load it and use it, and no need to retrain from scratch.