

1. Classification on Colab using MNIST dataset

◦ Process

1. Study the course material of [this chapter](#).
2. Run all the cells described in the notebook [Chapter 3 – Classification](#) on Colab and try to understand the code along the way.
3. [Adding notebooks to your portfolio](#)

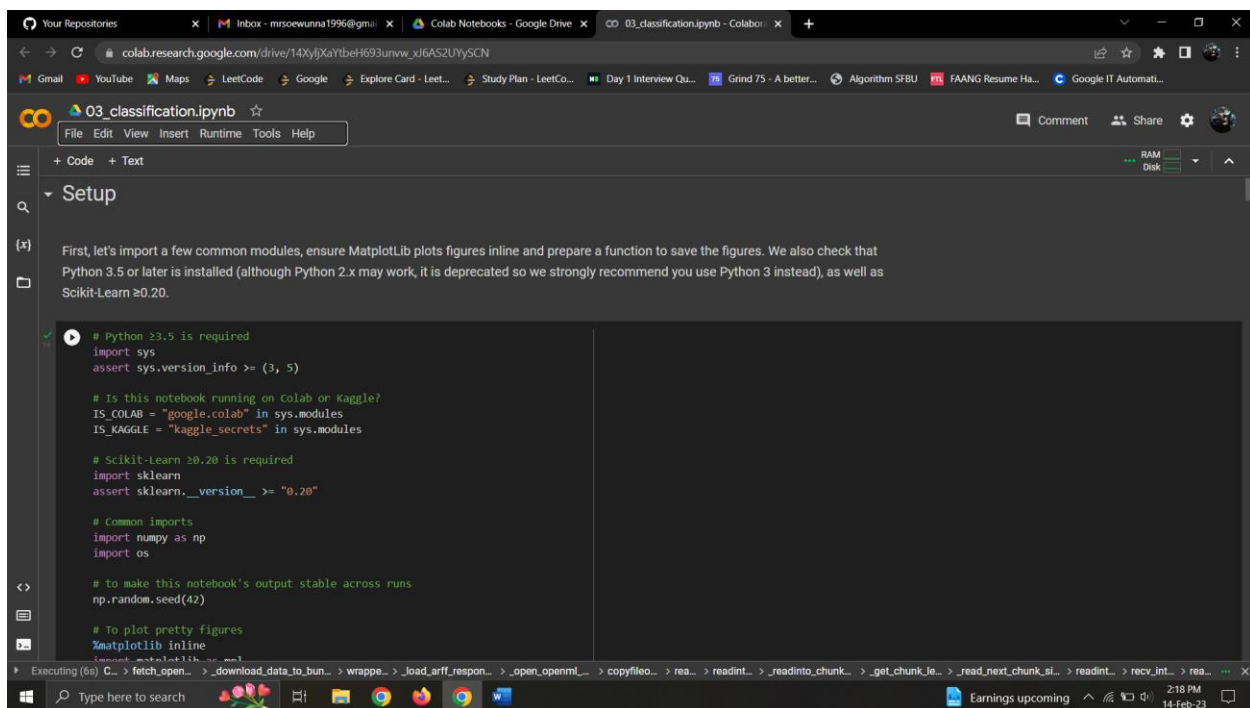
Machine Learning
Supervised Learning
Classification on Colab using MNIST dataset

◦ References

- Run the code on [Colab](#)
- References
 - [Get Start with Colab](#)

GitHub Link - <https://github.com/SoeWunna29/-Machine-Learning-Supervised-Learning-Classification-on-Colab-using-MNIST-dataset.git>

Running the notebook file



```
# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['axes', labelsizes=14]
mpl.rcParams['xtick', labelsizes=12]
mpl.rcParams['ytick', labelsizes=12]

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "classification"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

Warning: since Scikit-Learn 0.24, `fetch_opensml()` returns a Pandas DataFrame by default. To avoid this and keep the same code as in the book, we use `as_frame=False`.

```
[2] from sklearn.datasets import fetch_opensml
```

```
[2] from sklearn.datasets import fetch_opensml
mnist = fetch_opensml('mnist_784', version=1, as_frame=False)
mnist.keys()

dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

```
[3] X, y = mnist["data"], mnist["target"]
X.shape

(70000, 784)
```

```
[4] y.shape

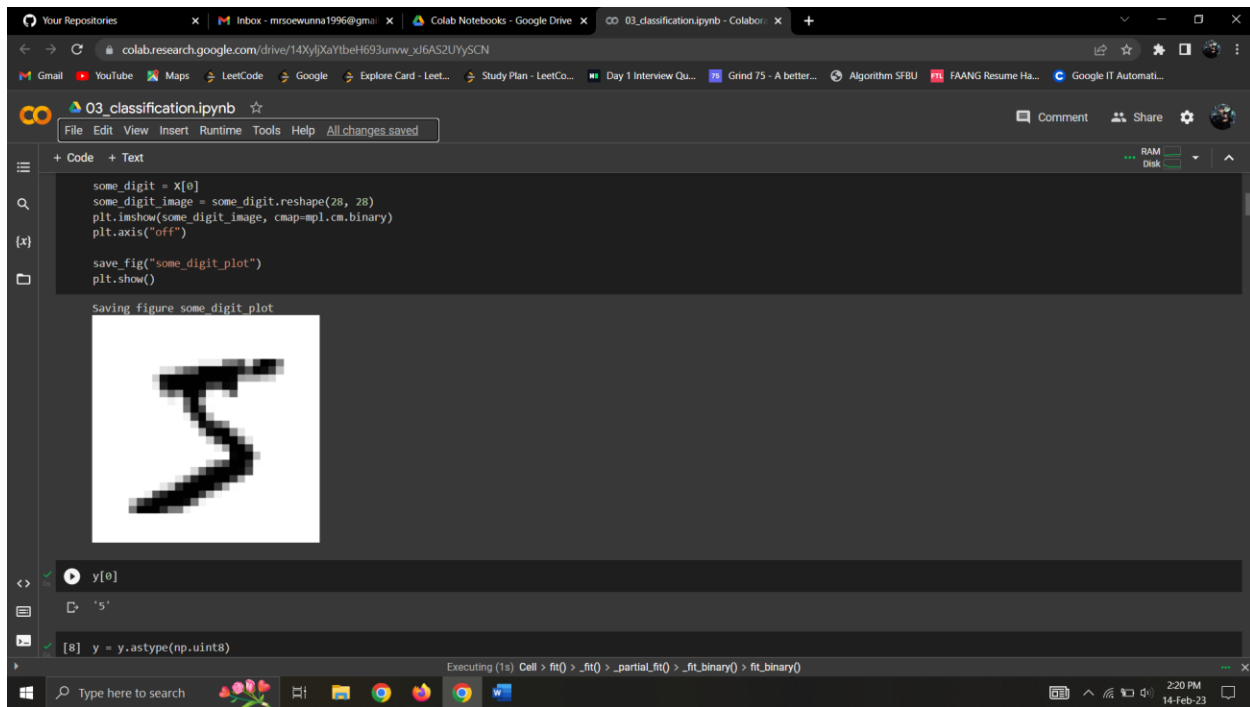
(70000,)
```

```
[5] 28 * 28

784
```

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
```

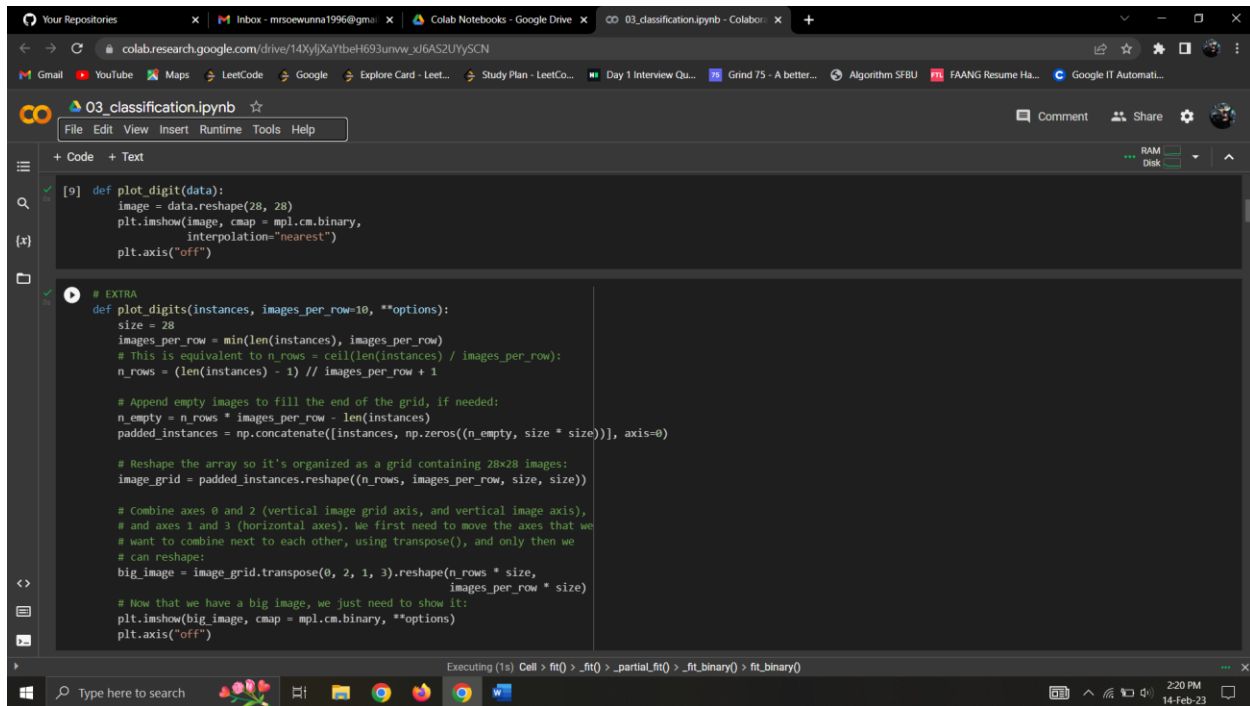


The screenshot shows a Google Colab notebook titled "O3_classification.ipynb". The code cell contains the following Python code:

```
some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=matplotlib.cm.binary)
plt.axis("off")

save_fig("some_digit_plot")
plt.show()
```

Below the code, a message says "Saving figure some digit plot" and a plot of a handwritten digit '3' is displayed. The plot is a 28x28 binary image. The console output shows the execution of the code, including the command `y = y.astype(np.uint8)`.



The screenshot shows a Google Colab notebook titled "O3_classification.ipynb". The code cell contains the following Python code:

```
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = matplotlib.cm.binary,
               interpolation="nearest")
    plt.axis("off")

# EXTRA
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    # This is equivalent to n_rows = ceil(len(instances) / images_per_row):
    n_rows = (len(instances) - 1) // images_per_row + 1

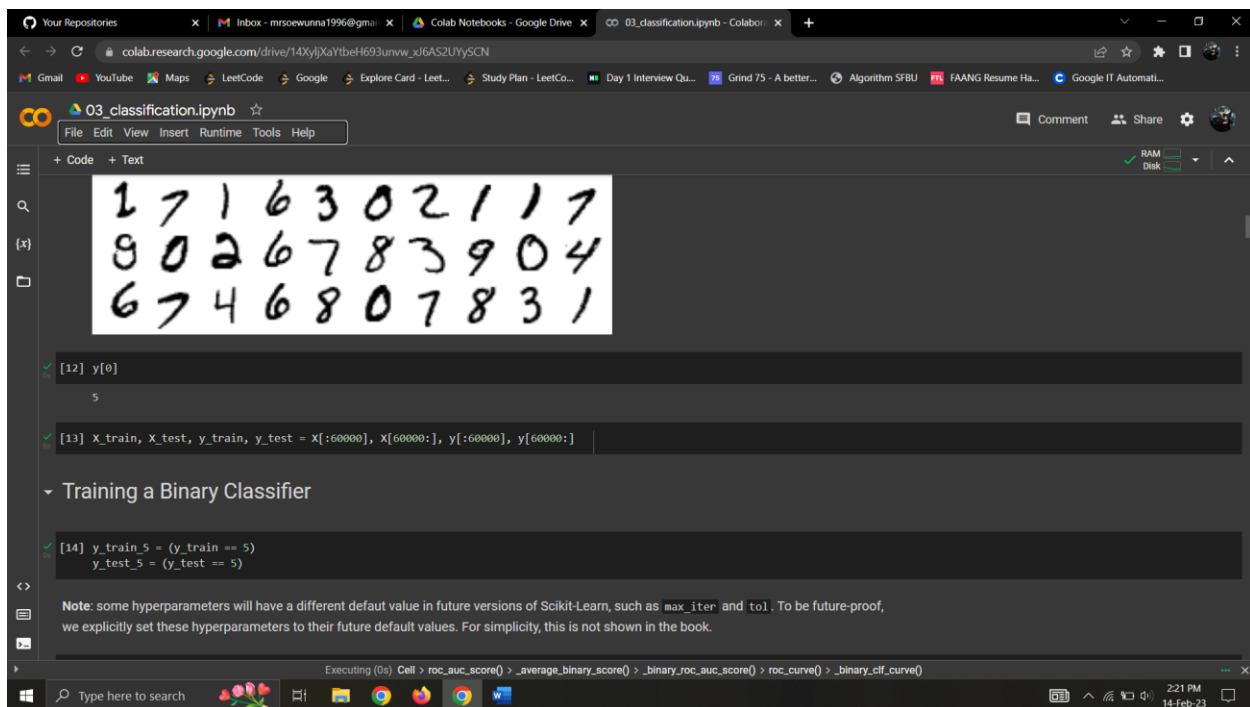
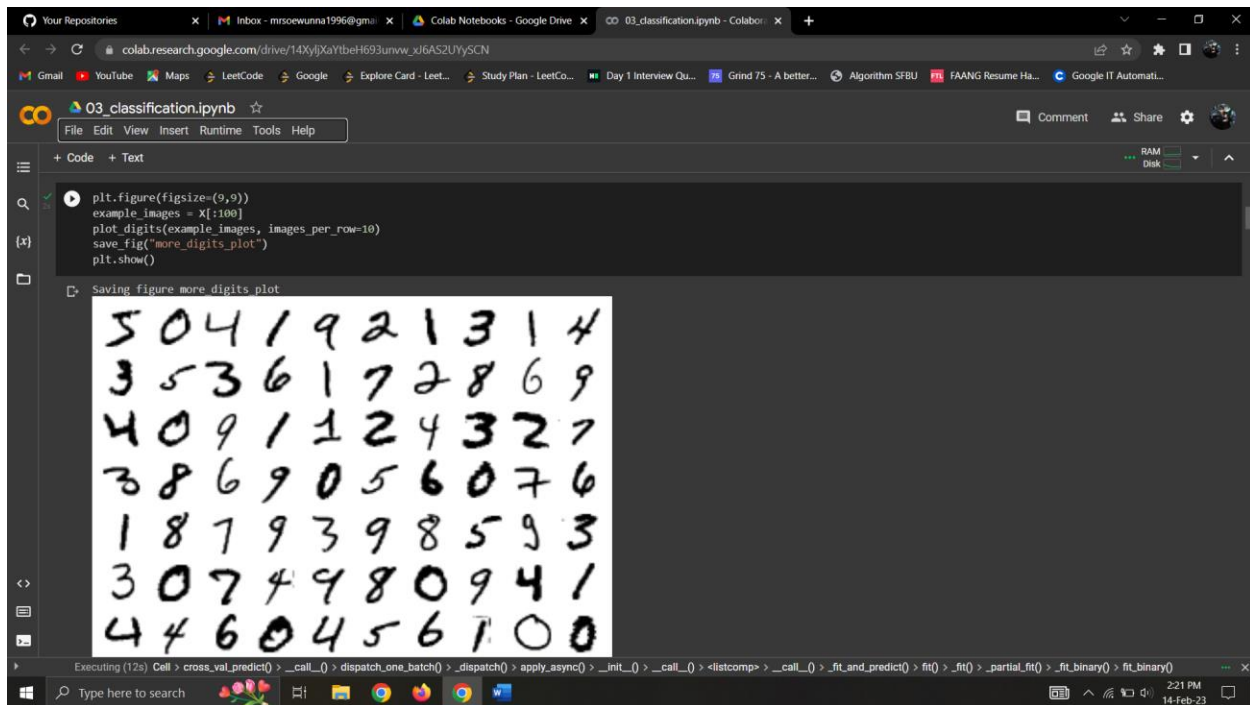
    # Append empty images to fill the end of the grid, if needed:
    n_empty = n_rows * images_per_row - len(instances)
    padded_instances = np.concatenate([instances, np.zeros((n_empty, size * size))], axis=0)

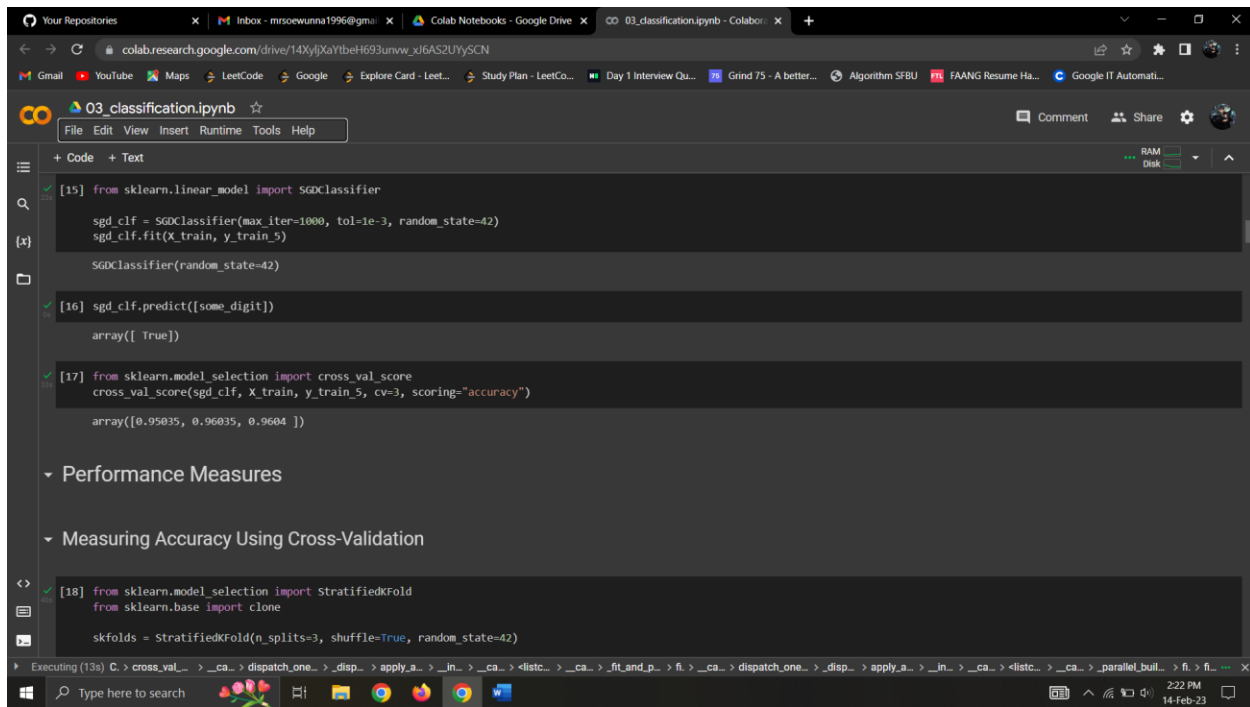
    # Reshape the array so it's organized as a grid containing 28x28 images:
    image_grid = padded_instances.reshape((n_rows, images_per_row, size, size))

    # combine axes 0 and 2 (vertical image grid axis, and vertical image axis),
    # and axes 1 and 3 (horizontal axes). We first need to move the axes that we
    # want to combine next to each other, using transpose(), and only then we
    # can reshape:
    big_image = image_grid.transpose(0, 2, 1, 3).reshape(n_rows * size,
                                                         images_per_row * size)

    # Now that we have a big image, we just need to show it:
    plt.imshow(big_image, cmap = matplotlib.cm.binary, **options)
    plt.axis("off")
```

The code defines a function `plot_digit` and an extra function `plot_digits`. The `plot_digits` function takes a list of instances and a grid size, and plots a grid of handwritten digits. The console output shows the execution of the code, including the command `plt.imshow(big_image, cmap = matplotlib.cm.binary, **options)`.





```
[15] from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)

SGDClassifier(random_state=42)

[16] sgd_clf.predict([some_digit])

array([ True])

[17] from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")

array([0.95035, 0.96035, 0.9604 ])
```

Performance Measures

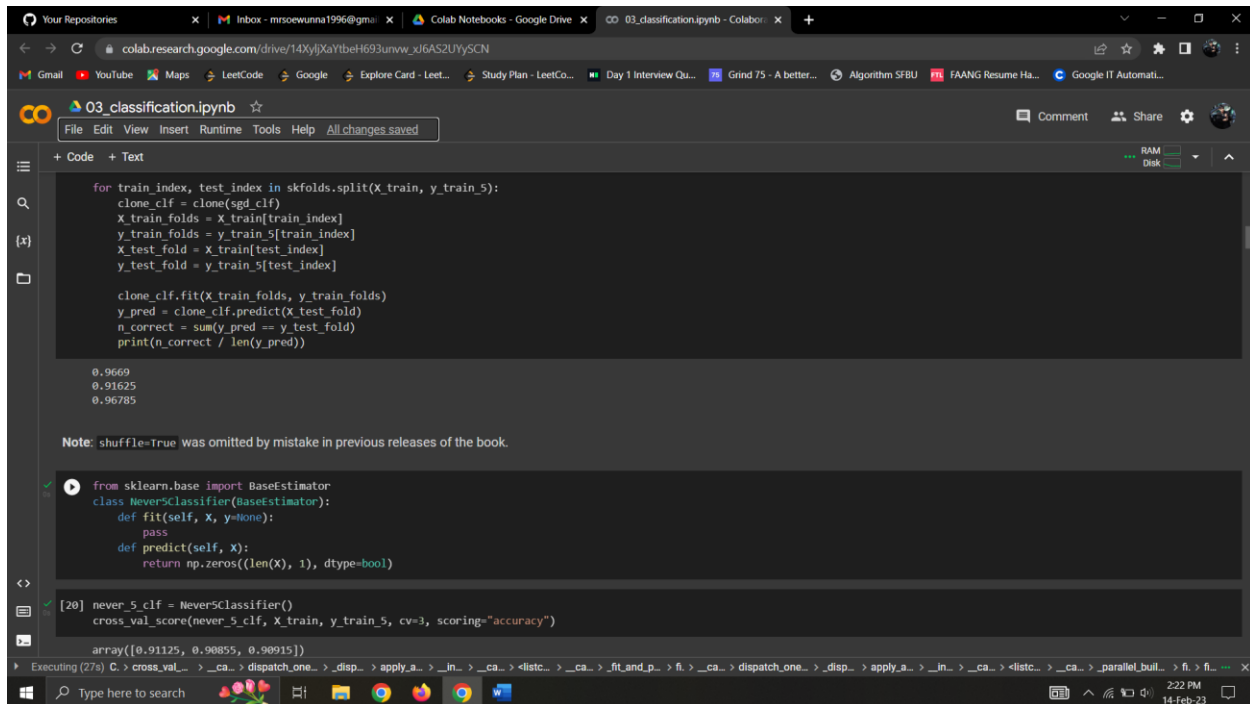
Measuring Accuracy Using Cross-Validation

```
[18] from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
```

Executing (13s) C: > cross_val_... > _ca... > dispatch_one... > _disp... > apply_a... > _in... > _ca... > <listc... > _ca... > _fit_and_p... > fi... > _ca... > dispatch_one... > _disp... > apply_a... > _in... > _ca... > <listc... > _ca... > _parallel_buil... > fi... > fi... > fi...

2:22 PM
14-Feb-23



```
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))

0.9669
0.91625
0.96785
```

Note: shuffle=True was omitted by mistake in previous releases of the book.

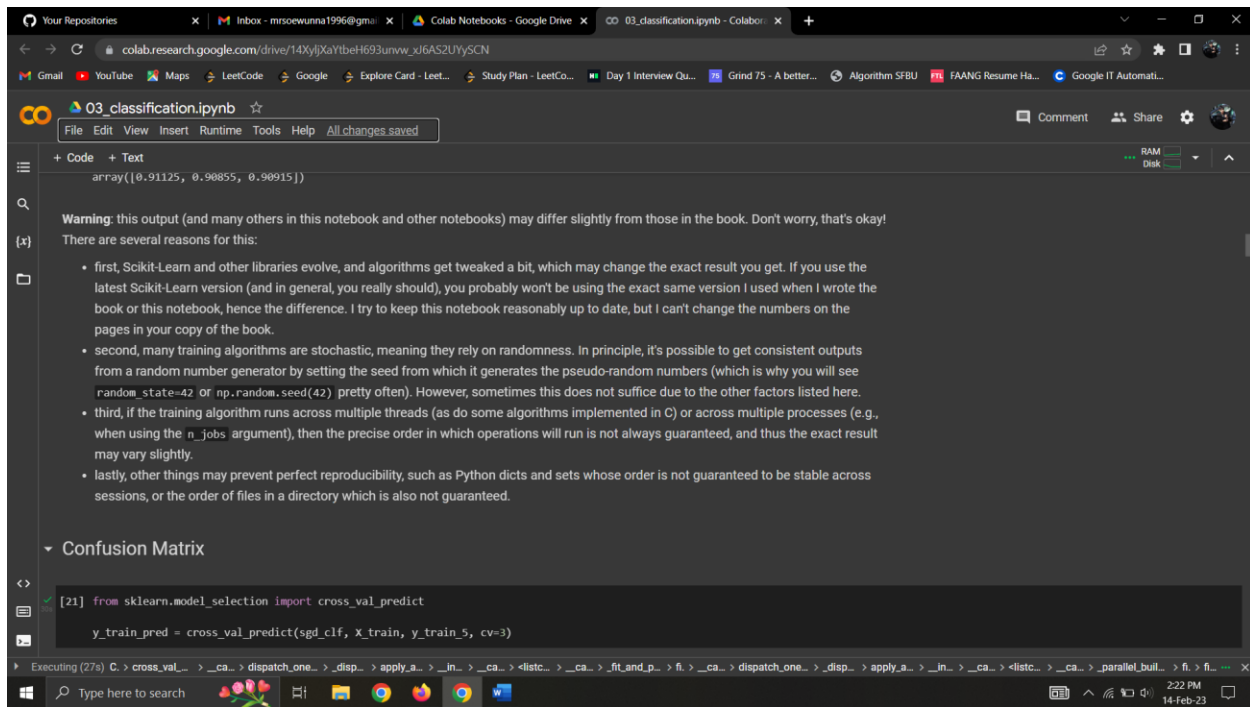
```
[19] from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)

[20] never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")

array([0.91125, 0.90855, 0.90915])
```

Executing (27s) C: > cross_val_... > _ca... > dispatch_one... > _disp... > apply_a... > _in... > _ca... > <listc... > _ca... > _parallel_buil... > fi... > fi... > fi...

2:22 PM
14-Feb-23

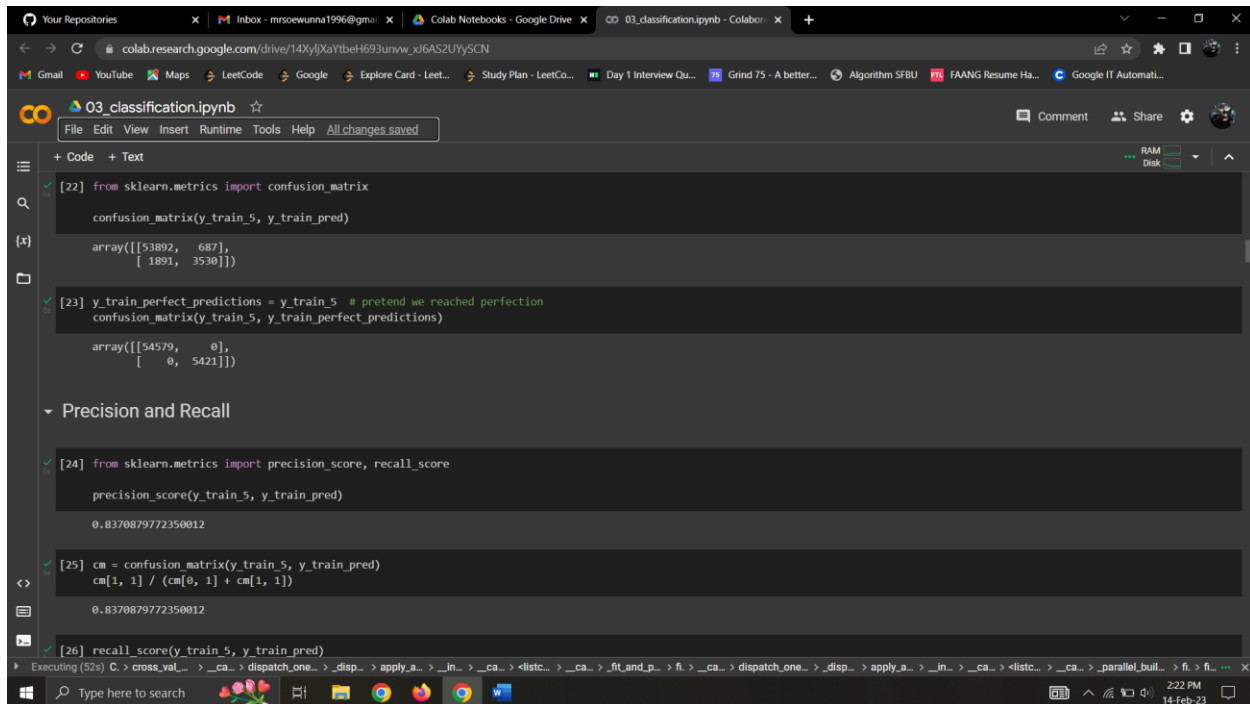


Warning: this output (and many others in this notebook and other notebooks) may differ slightly from those in the book. Don't worry, that's okay! There are several reasons for this:

- first, Scikit-Learn and other libraries evolve, and algorithms get tweaked a bit, which may change the exact result you get. If you use the latest Scikit-Learn version (and in general, you really should), you probably won't be using the exact same version I used when I wrote the book or this notebook, hence the difference. I try to keep this notebook reasonably up to date, but I can't change the numbers on the pages in your copy of the book.
- second, many training algorithms are stochastic, meaning they rely on randomness. In principle, it's possible to get consistent outputs from a random number generator by setting the seed from which it generates the pseudo-random numbers (which is why you will see `random_state=42` or `np.random.seed(42)` pretty often). However, sometimes this does not suffice due to the other factors listed here.
- third, if the training algorithm runs across multiple threads (as do some algorithms implemented in C) or across multiple processes (e.g., when using the `n_jobs` argument), then the precise order in which operations will run is not always guaranteed, and thus the exact result may vary slightly.
- lastly, other things may prevent perfect reproducibility, such as Python dicts and sets whose order is not guaranteed to be stable across sessions, or the order of files in a directory which is also not guaranteed.

Confusion Matrix

```
[21] from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```



```
[22] from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
array([[53892, 687],
       [1891, 3530]])

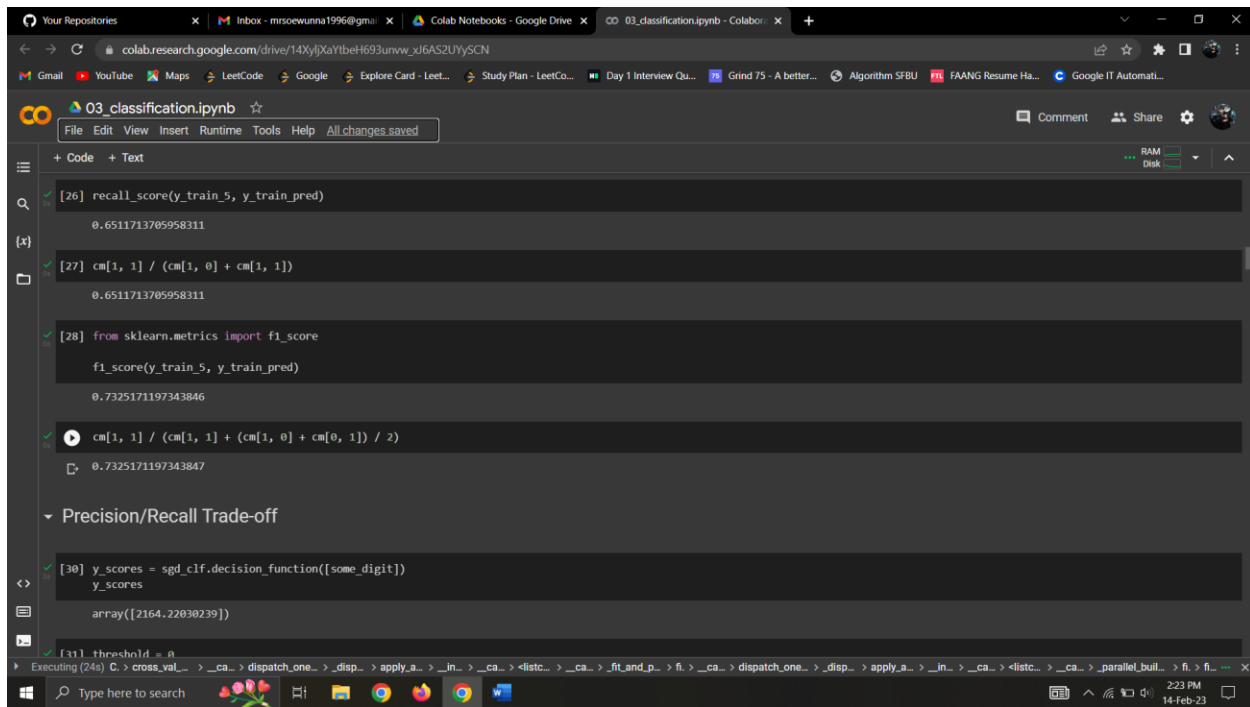
[23] y_train_perfect_predictions = y_train_5 # pretend we reached perfection
confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579, 0],
       [0, 5421]])
```

Precision and Recall

```
[24] from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred)
0.8370879772350012

[25] cm = confusion_matrix(y_train_5, y_train_pred)
cm[1, 1] / (cm[0, 1] + cm[1, 1])
0.8370879772350012

[26] recall_score(y_train_5, y_train_pred)
```



The screenshot shows a Jupyter Notebook titled "O3_classification.ipynb" in a web browser. The notebook contains several code cells. The first cell calculates the recall score for a specific threshold. The second cell calculates the F1 score. The third cell calculates the precision-recall trade-off. The fourth cell calculates the y_scores for a specific digit. The fifth cell calculates the threshold for a specific digit. The sixth cell calculates the cross-validated y_scores. The seventh cell imports the precision-recall curve function. The eighth cell defines a function to plot the precision-recall curve. The ninth cell calculates the recall at a specific precision.

```
[26] recall_score(y_train_5, y_train_pred)
0.6511713705958311

[27] cm[1, 1] / (cm[1, 0] + cm[1, 1])
0.6511713705958311

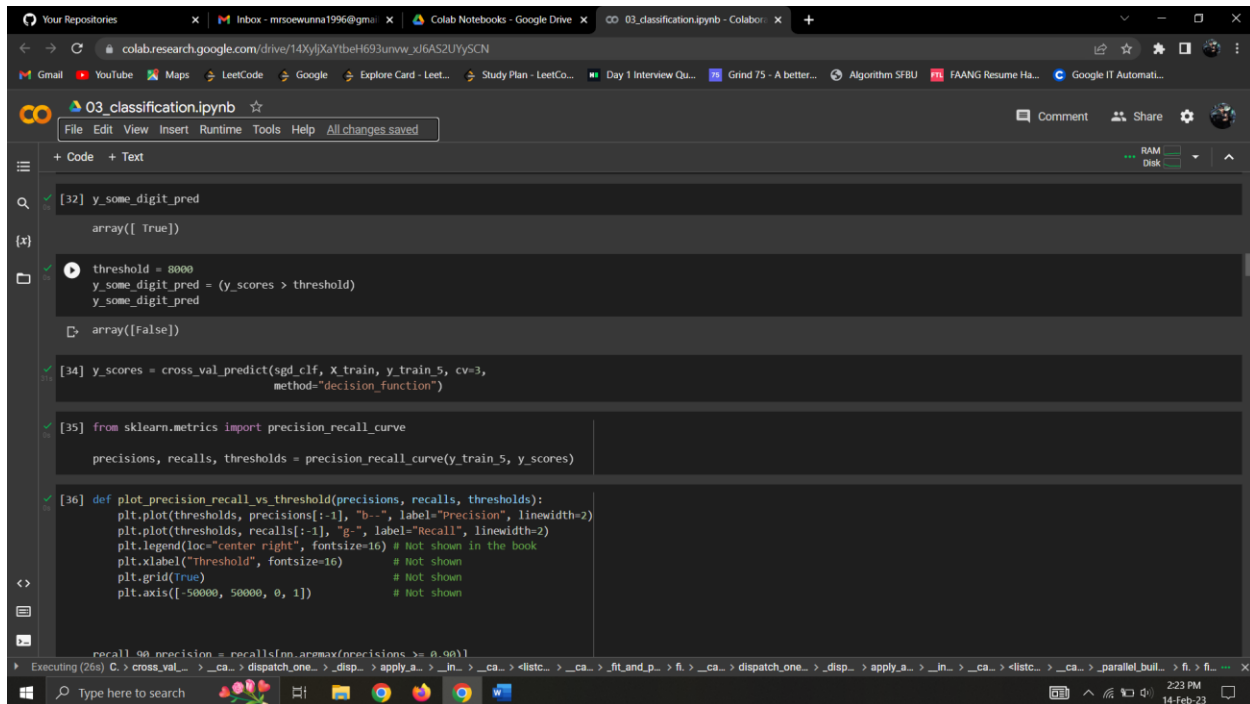
[28] from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
0.7325171197343846

[29] cm[1, 1] / (cm[1, 1] + (cm[1, 0] + cm[0, 1]) / 2)
0.7325171197343847

Precision/Recall Trade-off

[30] y_scores = sgf_clf.decision_function([some_digit])
y_scores
array([2164.22030239])

[31] threshold = 0
C> cross_val_...> _ca...> dispatch_one...> _disp...> apply_a...> _in...> _ca...> <list...> _ca...> _fit_and_p...> fi...> _ca...> dispatch_one...> _disp...> apply_a...> _in...> _ca...> <list...> _ca...> _parallel_bui...> fi...> fi...
```



The screenshot shows a Jupyter Notebook titled "O3_classification.ipynb" in a web browser. The notebook contains several code cells. The first cell calculates the y_scores for a specific digit. The second cell calculates the threshold for a specific digit. The third cell calculates the cross-validated y_scores. The fourth cell imports the precision-recall curve function. The fifth cell defines a function to plot the precision-recall curve. The sixth cell calculates the recall at a specific precision.

```
[32] y_some_digit_pred
array([ True])

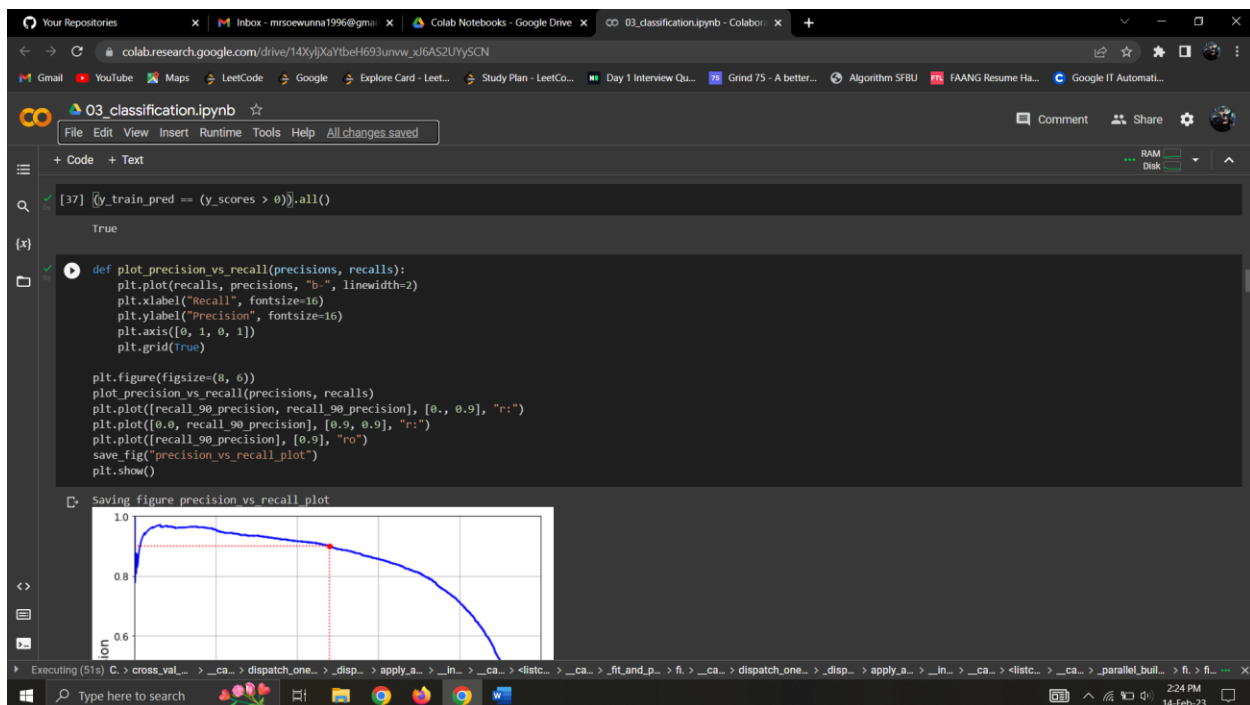
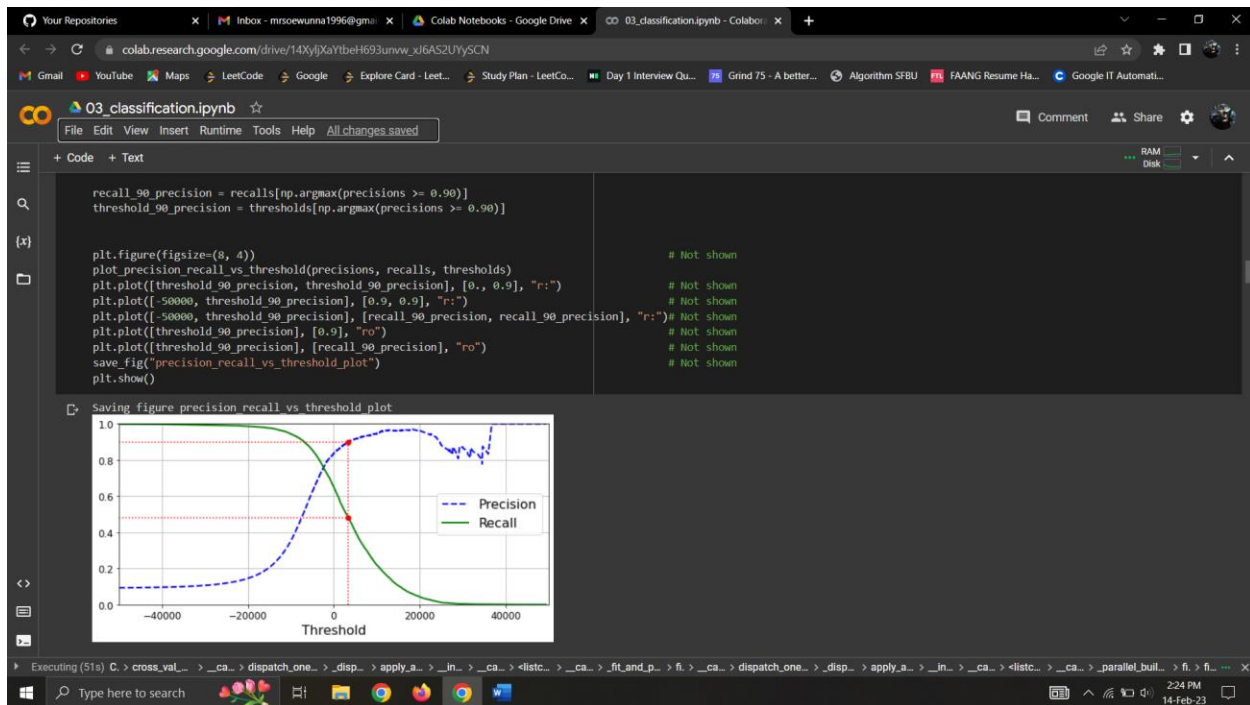
[33] threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
array([False])

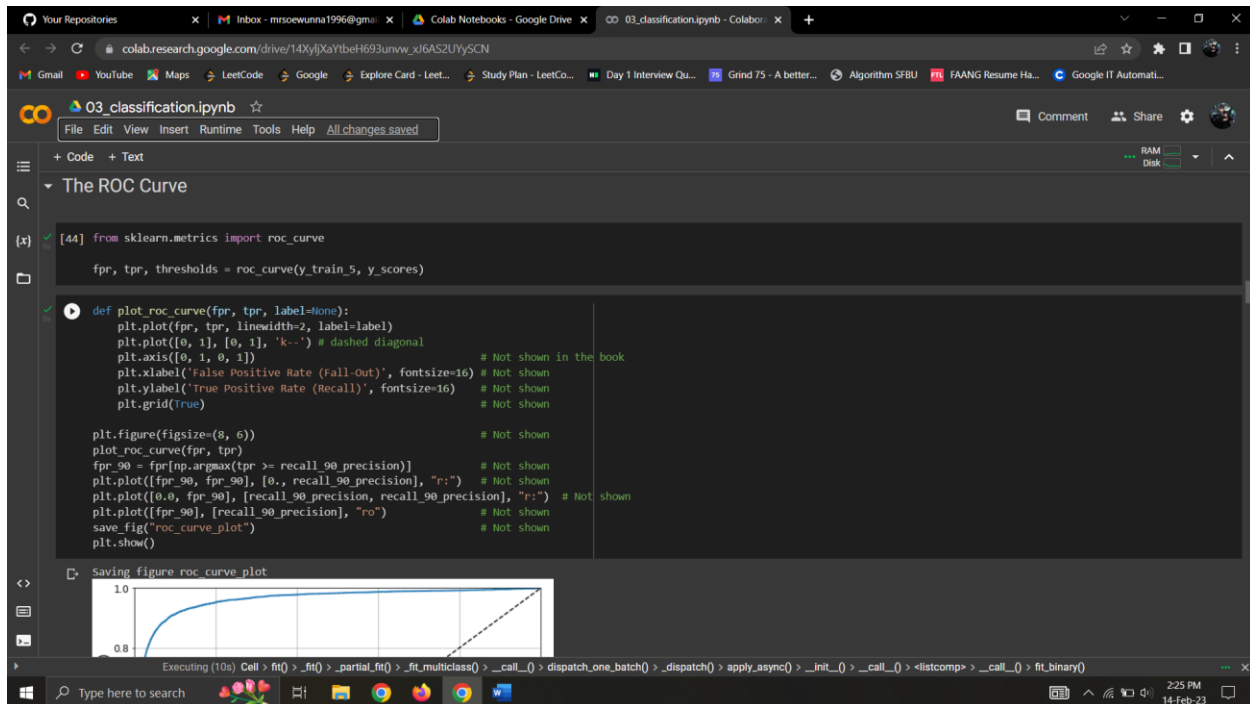
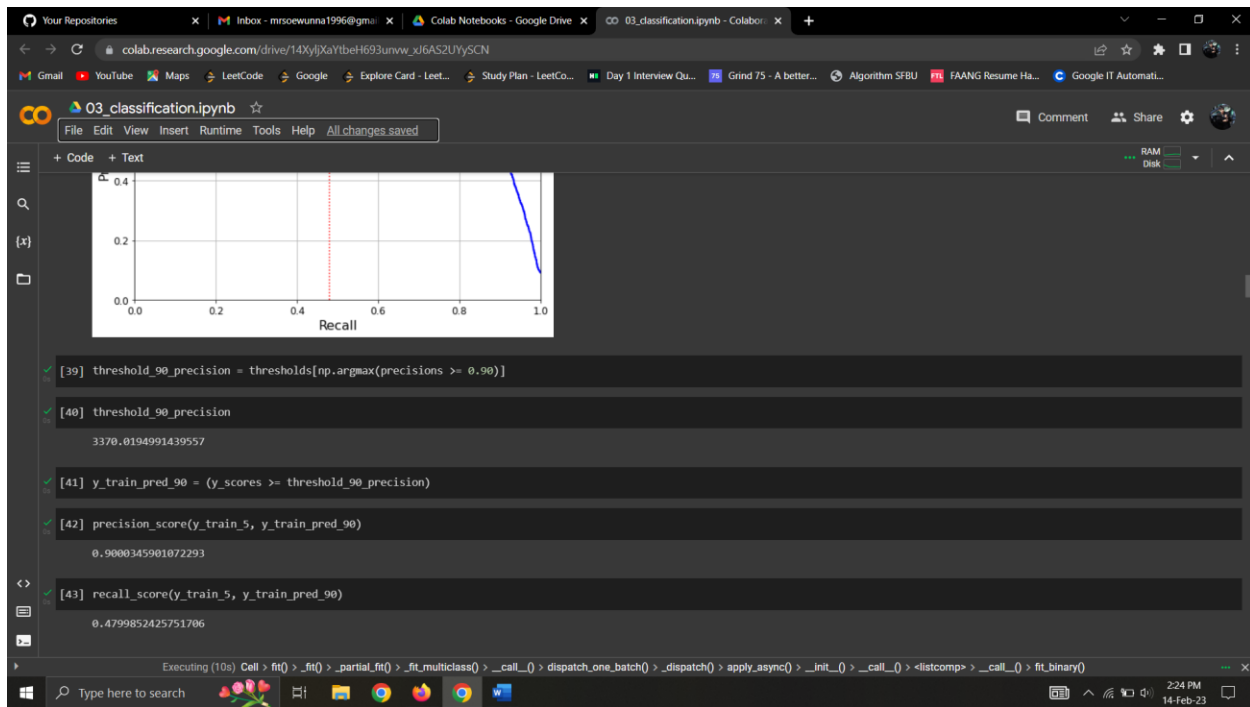
[34] y_scores = cross_val_predict(sgf_clf, X_train, y_train_5, cv=3,
method="decision_function")

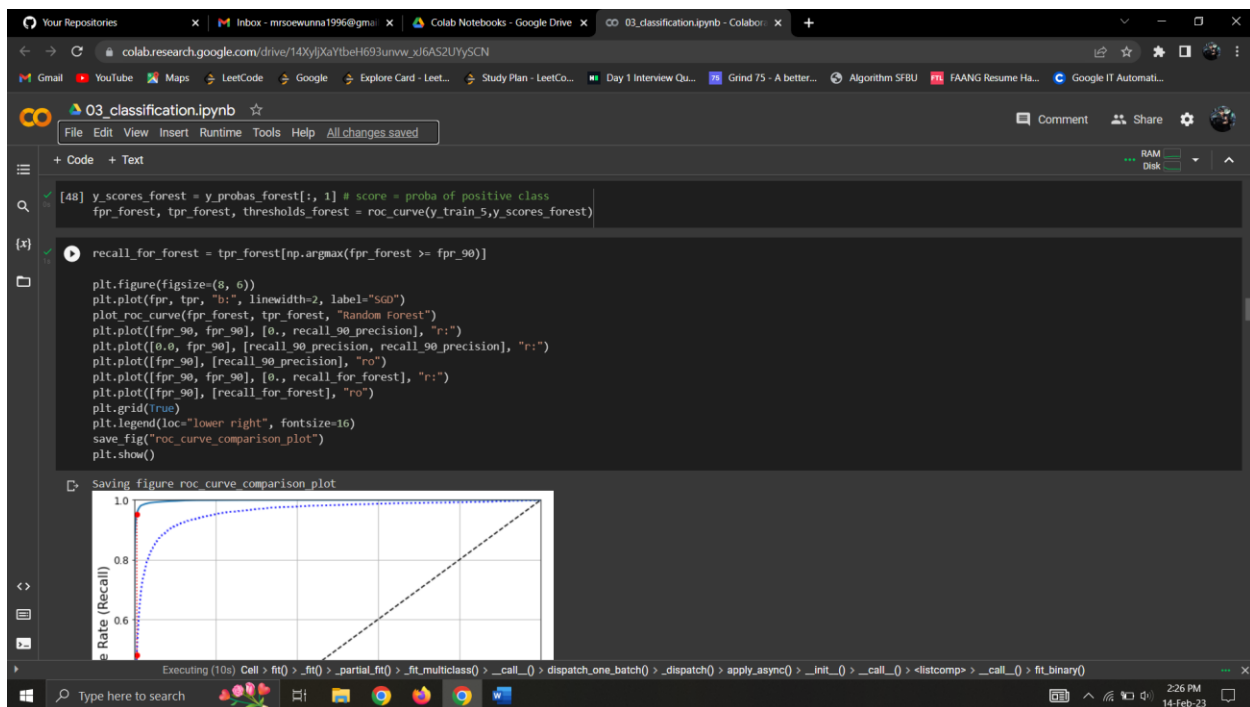
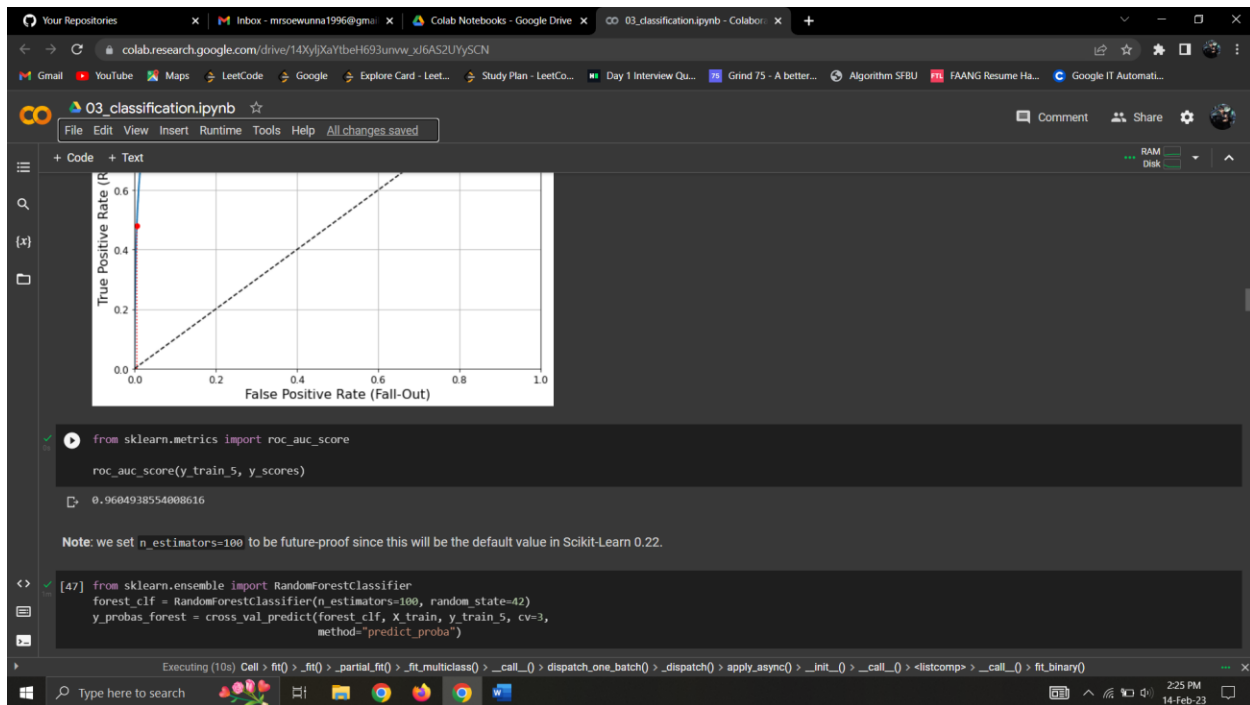
[35] from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)

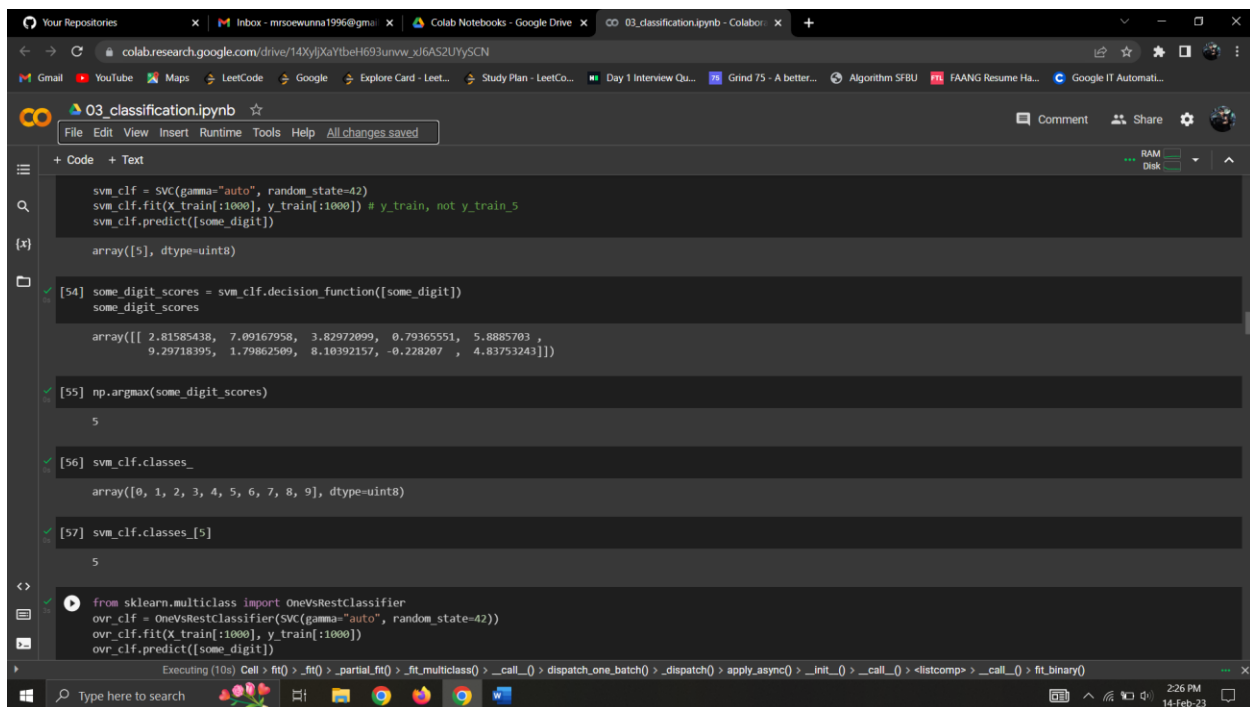
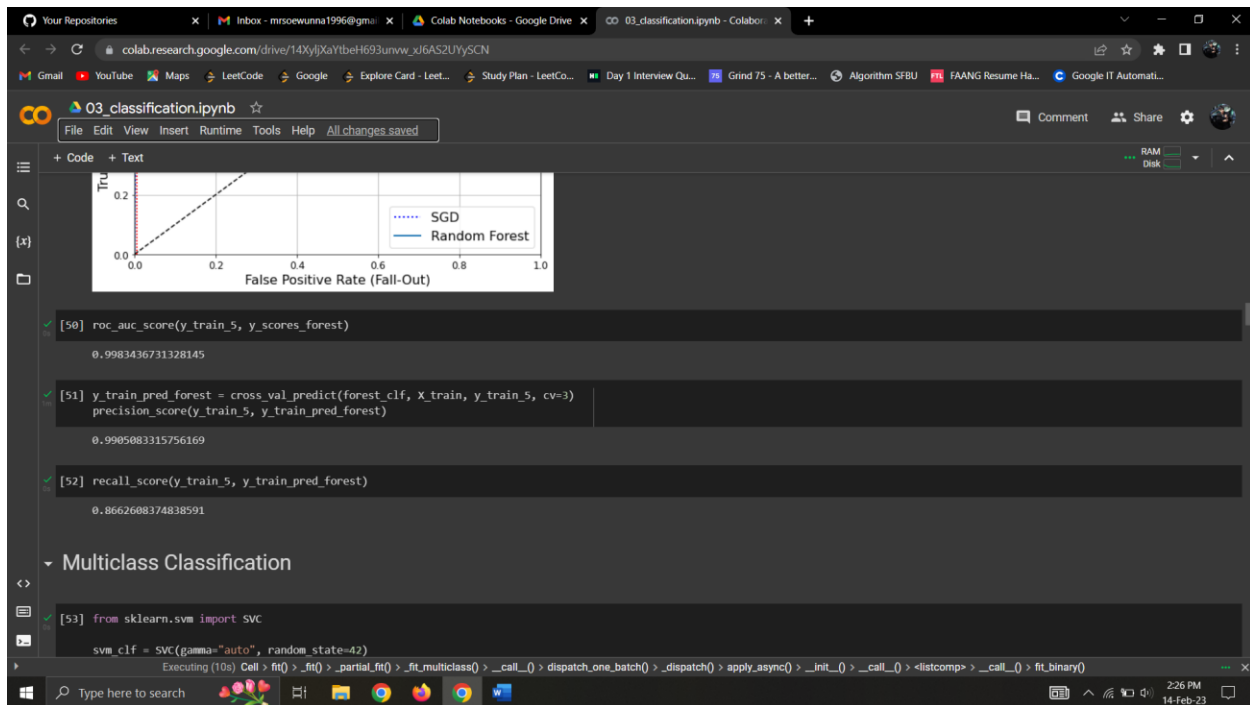
[36] def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.legend(loc="center right", fontsize=16) # Not shown in the book
plt.xlabel("Threshold", fontsize=16) # Not shown
plt.grid(True) # Not shown
plt.axis([-50000, 50000, 0, 1]) # Not shown

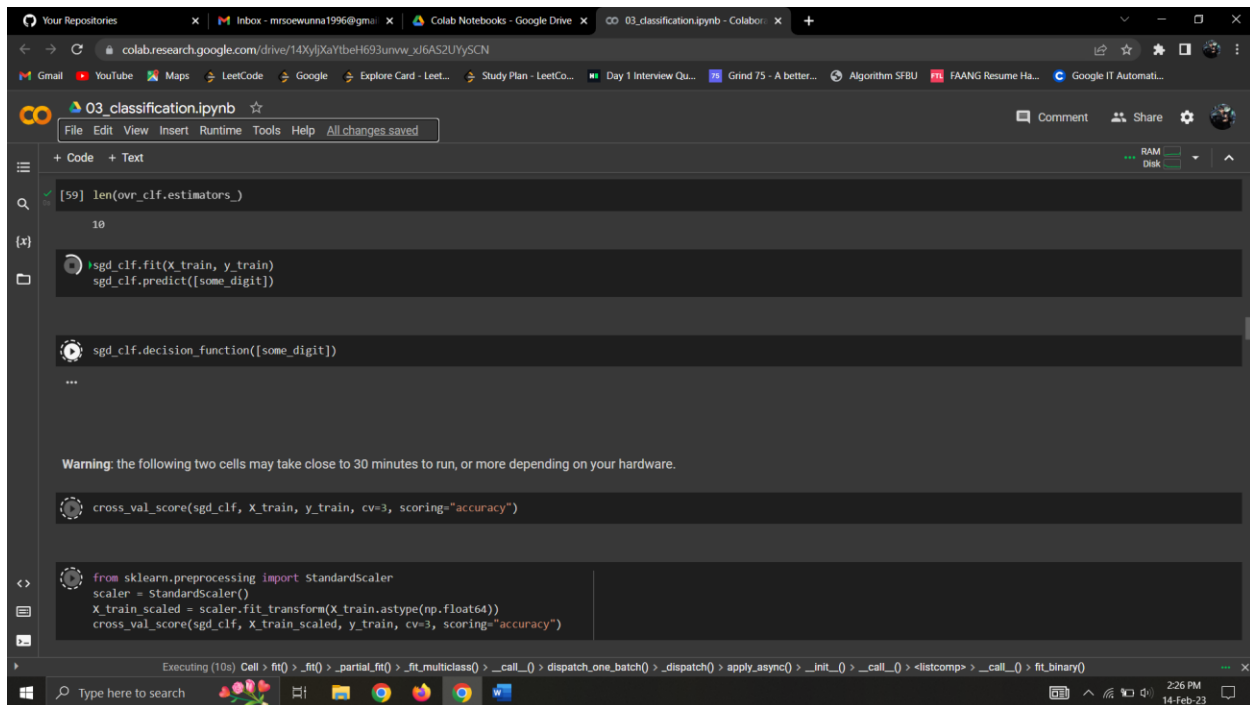
recall_at_precision = recalls[np.argmax(precisions >= 0.90)]
```









colab.research.google.com/drive/14XyIjKaYtbeH693unvw_xd6AS2UYySCN

O3_classification.ipynb

```
[59] len(ovr_clf.estimators_)

10

sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])

sgd_clf.decision_function([some_digit])

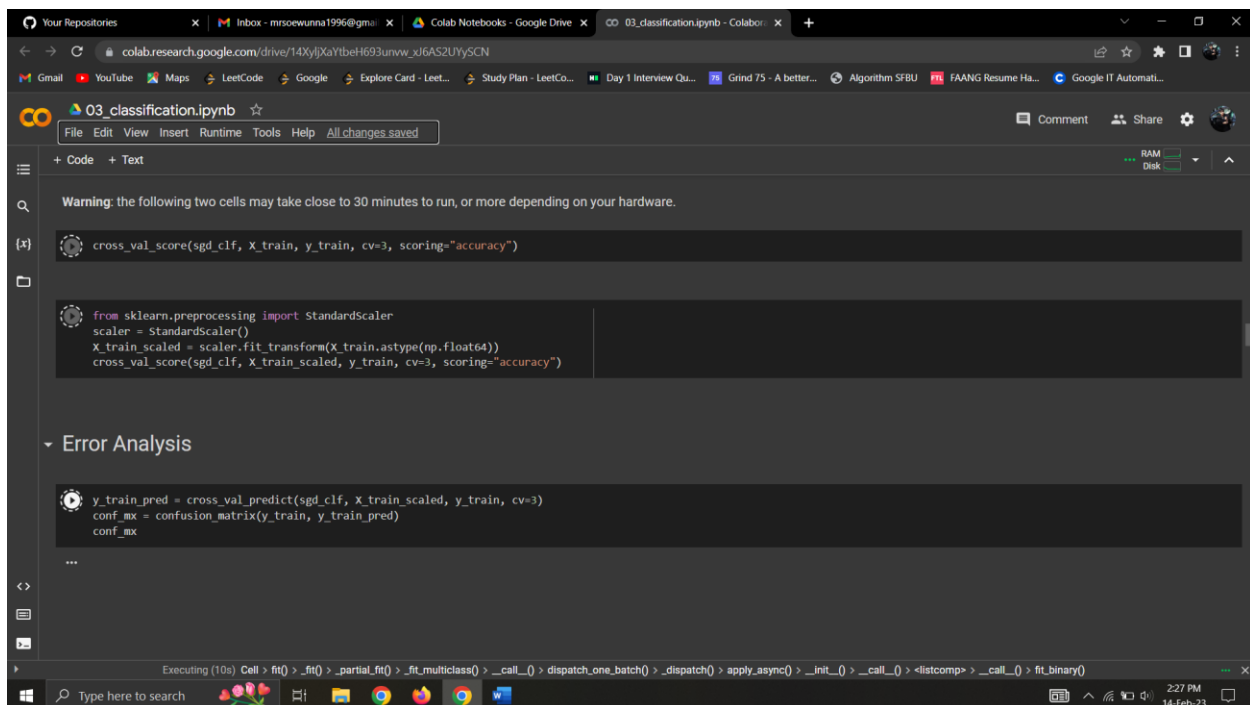
...

Warning: the following two cells may take close to 30 minutes to run, or more depending on your hardware.

cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
```

Executing (10s) Cell > fit() > _fit() > _partial_fit() > _fit_multiclass() > _call__() > dispatch_one_batch() > _dispatch() > apply_async() > _init__() > _call__() > <listcomp> > _call__() > fit_binary()



colab.research.google.com/drive/14XyIjKaYtbeH693unvw_xd6AS2UYySCN

O3_classification.ipynb

```
Warning: the following two cells may take close to 30 minutes to run, or more depending on your hardware.

cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")

Error Analysis

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx

...
```

Executing (10s) Cell > fit() > _fit() > _partial_fit() > _fit_multiclass() > _call__() > dispatch_one_batch() > _dispatch() > apply_async() > _init__() > _call__() > <listcomp> > _call__() > fit_binary()