COMPSCI 389
Introduction to Machine Learning

Models Evaluation

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

Jupyter Notebook

- This slide presentation follows the Jupyter notebook:

`6 Model Evaluation.ipynb`

Model Evaluation

- **Model Evaluation**: Quantifying how effective a model is at making predictions.
- **Note**: Different from evaluating an ML algorithm.

Review

- Recall from last time:
  - We loaded the GPA data
  - We split it into X and y data

```
# Load the data set
df = pd.read_csv("https://people.cs.umass.edu/~pthomas/courses/COMPSCI_389_Spring2024/GPA.csv", delimiter=',')
#df = pd.read_csv("data/GPA.csv", delimiter=',')

# Display the data set
display(df)

# Split into X (inputs) and y (labels)
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
```

Review (cont.)

- We confirmed that this algorithm (SciKit-Learn model) can make predictions.
- We used a K-D tree to improve runtimes when running many queries.
- Next we will evaluate how “good” the predictions are.

```python
class NearestNeighbor(BaseEstimator):
    def fit(self, X, y):
        # Convert X and y to NumPy arrays if they are DataFrames.
        # This makes fit compatible with numpy arrays or DataFrames
        if isinstance(X, pd.DataFrame):
            X = X.values
        if isinstance(y, pd.Series):
            y = y.values

        # Store the training data and labels.
        self.X_data = X
        self.y_data = y

        # Create a KDTree for efficient nearest neighbor search
        self.tree = KDTree(X)

        return self

    def predict(self, X):
        # Convert X to a NumPy array if it's a DataFrame
        if isinstance(X, pd.DataFrame):
            X = X.values

        # Query the tree for the nearest neighbors of all points in X.
        # ind will be a 2D array where ind[i,j] is the index of the
        # j'th nearest point to the i'th row in X.
        dist, ind = self.tree.query(X, k=1)

        # Extract the nearest labels.
        # ind[:,0] are the indices of the nearest neighbors to each
        # query (each row in x))
        return self.y_data[ind[:,0]]
```

Model Evaluation

```
# Train the model on the data
model = NearestNeighbor()
model.fit(X, y)
predictions = model.predict(X)

# Compute the average error
average_error = (predictions - y).mean()

print("Average Error:", average_error)
```

- **Question**: What will this output?

```
Average Error: 0.0
```

Perfect Predictions?

- We've seemingly achieved perfect predictions with our model!
- **Question**: Are our predictions genuinely perfect?
- **Answer**: Not really. We evaluated our model using the training data.
- Evaluating a model on the training data answers the question:

How well does our model predict outcomes for data it has already seen?

- The real question we want to answer is:

How well can our model predict outcomes for new, unseen data?

- This problem arises when evaluating *any* ML algorithm, not just NN.

Train/Test Splits: Idea

- **Idea**: To accurately assess a model's performance, we need to test it on data that it hasn't seen during training.
- **Training Set**: A subset of the data used to train the model.
- **Testing Set**: A different subset of the data used to evaluate the model.
  - **Note**: The training and testing sets **form distinct, non-overlapping subsets** of the available data.

Train/Test Split: Sizes

- **Question**: If we have data_size points (rows), how many should we use for training and how many for testing?
- **Answer**: No fixed answer.
- If we use too much for training, our evaluation will have high variance (it will not be reliable).
- If we use too little for training, the models we learn will not perform well.
- Some research studies optimizing the train/test split.
- The *vast* majority of the time people pick a split based on intuition.
  - Often 50/50, 60/40, 80/20.

Data Splitting

- **Question**: If we use $X\%$ for training and $(100 - X)\%$ for testing, what's something we should watch out for in real applications?
- **Answer**: The data could be sorted, biasing our evaluation.
- **Solution**: Randomly select which points go into the training and testing sets.
- Equivalently, *shuffle* the data.
- We will use `train_test_split` from scikit-learn, which does this for us when `shuffle=True`.

```
# We already loaded X and y, but do it again as a reminder
X = df.iloc[:, :-1]
y = df.iloc[:, -1]

# Split the data into training and testing sets (60% train, 40% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, shuffle=True)

# Display the training data.
display(X_train)

# Train the model on the training data
model.fit(X_train, y_train)

# Predict on the testing data
predictions = model.predict(X_test)

# Compute the average error on the testing data
average_error = (predictions - y_test).mean()

print("Average Error:", average_error)
```

What will this output?

Is it a good evaluation of the accuracy of the predictions?

Shuffling preserves the original row indices so you can look up the corresponding labels.

```
# Display the training data.
display(X_train)
```

Note: When we convert the X data to a numpy array, it strips these indices.

```
if isinstance(X, pd.DataFrame):
    X = X.values
```

	physics	biology	history	English	geography	literature	Portuguese	math	chemistry
34466	363.61	384.30	444.75	481.66	527.76	424.35	364.44	500.46	364.59
31701	663.20	559.99	626.39	605.85	672.04	595.91	642.66	621.96	617.05
10206	752.42	595.05	676.76	708.86	677.09	621.36	722.86	727.01	667.81
15650	558.19	545.46	355.04	547.26	699.05	525.18	430.45	685.34	522.53
25664	688.19	637.15	537.93	595.85	646.29	644.87	539.35	603.14	657.16
...	...	...	...	...	...	...	...	...	...
22251	432.60	415.41	490.79	466.19	394.45	453.95	443.17	503.82	398.43
36512	465.14	637.15	630.61	541.54	579.64	573.57	481.10	511.14	460.17
5266	620.82	692.89	543.48	484.45	532.28	660.77	493.48	650.64	590.50
4233	469.73	488.78	521.77	542.20	498.94	475.30	475.85	429.04	498.83
6139	516.22	545.46	607.26	469.30	483.23	525.18	510.13	614.54	588.98

```
print("Average Error:", average_error)
```

```
Average Error: -0.0015026503463803262
```

- The predictions are *really* good!
- We can predict new applicant GPAs to within a couple *thousandths* of a GPA point!

Let’s look at some of these super-accurate predictions:

```
# The predictions are a numpy array. Convert them to a Series
predictions_series = pd.Series(predictions, name='prediction')
y_test_series = pd.Series(y_test, name='label').reset_index(drop=True) # We reset the indices in y_test.

# Calculate the difference
difference = predictions_series - y_test_series

# Create a new DataFrame
temp = pd.DataFrame({
    'label': y_test_series,
    'prediction': predictions_series,
    'difference': difference
})

print(temp)
```

Discard the old indices

What went wrong?

- The predictions aren't within a couple *thousandths* of a GPA point!
- **Question**: What went wrong?
- **Answer**: The average error lets positive and negative errors cancel out!
- The average error tells us that on average we are *under-predicting* by a small amount.

	label	prediction	difference
0	3.77333	3.87000	0.09667
1	1.81667	1.92667	0.11000
2	2.16667	2.16667	0.00000
3	3.02667	0.00000	-3.02667
4	3.50333	3.92333	0.42000
...	...	...	...
17317	2.55667	2.98000	0.42333
17318	3.83333	3.83333	0.00000
17319	3.77667	3.50000	-0.27667
17320	2.15667	3.73333	1.57666
17321	2.50000	1.89000	-0.61000

[17322 rows x 3 columns]

```
Average Error: -0.0015026503463803262
```

Evaluation Metrics (Regression)

Actual label Predicted label

- Mean (Average) Error: $\frac{1}{n}\sum_{i=1}^{n} y_i - \hat{y}_i$
  - Rarely what you want.
  - Allows positive and negative errors to cancel each other out.
- Mean Squared Error (MSE): $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
  - Very common choice.
  - Gives a higher weight to larger errors, making it sensitive to outliers. It's useful when large errors are particularly undesirable.
- Root Mean Squared Error (RMSE): $\sqrt{\mathrm{MSE}}$
  - Has the same units as the target variable (unlike MSE).

Evaluation Metrics (Regression, cont.)

- Mean Absolute Error (MAE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
  - Like MSE, but with less emphasis on outliers.
- R-squared (R^2): $1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$, where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.
  - Also called the *coefficient of determination*.
  - Indicates the proportion of the variance of the dependent variable (labels) that is predictable from the independent variables (predictions).
  - Larger is better (maximum possible is one).
    - Can be negative if predictions are particularly poor.

Evaluation Metrics (Implementation)

```python
def mean_squared_error(predictions, labels):
    return np.mean((predictions - labels) ** 2)

def root_mean_squared_error(predictions, labels):
    return np.sqrt(mean_squared_error(predictions, labels))

def mean_absolute_error(predictions, labels):
    return np.mean(np.abs(predictions - labels))

def r_squared(predictions, labels):
    ss_res = np.sum((labels - predictions) ** 2)        # ss_res is the "Sum of Squares of Residuals"
    ss_tot = np.sum((labels - np.mean(labels)) ** 2)    # ss_tot is the "Total Sum of Squares"
    return 1 - (ss_res / ss_tot)
```

Nearest Neighbor Re-Evaluation (Code)

```
# Compute the average error and other metrics on the testing data
average_error = (predictions - y_test).mean()
mse = mean_squared_error(predictions, y_test)
rmse = root_mean_squared_error(predictions, y_test)
mae = mean_absolute_error(predictions, y_test)
r2 = r_squared(predictions, y_test)

# Print the metrics
print("Average Error:", average_error)
print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)
print("R-squared:", r2)
```

Nearest Neighbor Re-Evaluation (Results)

```
Average Error: -0.005299791542547050 6
Mean Squared Error: 1.1338053245452968
Root Mean Squared Error: 1.0648029510408472
Mean Absolute Error: 0.8201980262036715
R-squared: -0.6899200719482117
```

- These give a much clearer picture of how accurate the model is.
- Some are easier to interpret than others.
- All can be used to compare the performance of different ML models.

Conclusion

- Use separate data to train and test (evaluate) models.
- When evaluating models, select an appropriate metric
- For regression common metrics include:
  - Mean squared error (MSE)
  - Root mean squared error (RMSE)
  - Mean absolute error (MAE)
  - R-squared