Advice for applying ML

Deciding what to try next

Debugging a learning algorithm

You've implemented **regularized linear regression** on housing prices:

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

❓ But it makes **unacceptably large errors** in predictions. What do you try next?

- Get more training examples
- Try smaller sets of features
- Try getting additional features
- Try adding polynomial features ($x_1^2, x_2^2, x_1 x_2, \text{etc}$)
- Try decreasing λ
- Try increasing λ

ML Diagnostic

Diagnostic: A test that you run to gain insight into what is/isn't working with a learning algorithm, to gain guidance into improving its performance.

- Implementing diagnostics can provide insights into the performance of learning algorithms and guide improvements.
- These tests can help determine whether investing time in collecting more training data is worthwhile.

Evaluating the Model

Model Evaluation Techniques

- Splitting the dataset: Divide the dataset into a training set (e.g., 70%) and a test set (e.g., 30%) to train the model and evaluate its performance.

- Training and test errors: Calculate the training error J_{train} and test error J_{test} to assess how well the model generalizes to new data.

Train/test procedure for Linear Regression

(with squared error cost)

The cost function is defined as follow:

$$J(\vec{w},b) = \frac{1}{2m_{train}} \sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}\left(\vec{x}^i\right) - y^i\right)^2 + \frac{\lambda}{2m_{train}} \sum_{j=1}^{n} w_j^2$$

With the test error is computed:

$$J_{test}(\vec{w},b) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right) - y_{test}^{(i)}\right)^2$$

and the training error is computed as:

$$J_{train}(\vec{w},b) = \frac{1}{2m_{train}} \sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right) - y_{train}^{(i)}\right)^2$$

We can conclude that

- If the J_{train} is too **HIGH** ⇒ the model is **underfitting**.
- If the J_{train} is too **LOW (👍 good)** but the J_{test} is too **HIGH** ⇒ the model is **overfitting**.

Train/test procedure for Classification Problems

The cost function is defined as follow:

$J(\vec{w},b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[-y^i \log\left(f_{\vec{w},b}\left(\vec{x}^i\right)\right) + \left(1-y^i\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}^i\right)\right)\right] + \frac{\lambda}{2m_{train}} \sum_{j=1}^{n} w_j^2$

With the test error is computed:

$J_{test}(\vec{w},b) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left[-y_{test}^{(i)} \log\left(f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right)\right) + \left(1-y_{test}^{(i)}\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right)\right)\right]$

and the training error is computed as:

$J_{train}(\vec{w},b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[-y_{train}^{(i)} \log\left(f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right)\right) + \left(1-y_{train}^{(i)}\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right)\right)\right]$

We can conclude that

- If the J_{train} is too **HIGH** ⇒ the model is **underfitted**.
- If the J_{train} is too **LOW (👍 good)** but the J_{test} is too **HIGH** ⇒ the model is **overfitted**.

J_{train} and J_{test}, which measure misclassification rates instead of relying solely on logistic loss.

- **Predictions**: The model predicts a class label (e.g., 0 or 1) for each test example. For instance, if the output function $f(x)$ is greater than or equal to 0.5, the prediction:

COUNT $\hat{y} \neq y$

- **Misclassification Rate**:
  - J_{test}: is the fraction of misclassified examples in the test set.
    - For example, if the model incorrectly classifies a 0 as a 1 or a 1 as a 0, these instances contribute to the misclassification rate.
  - J_{train}: Similarly, this measures the fraction of misclassified examples in the training set.

⇒ we can analyze our model in the same way as in linear regression.

Model selection and training/cross validation/test sets

Model selection (choosing a model)

Understanding Generalization Error:

- **Generalization error** refers to how well a model performs on **unseen** data.
- **Training error may be low,** but it does **not guarante**e good performance on new examples.

Using a Test Set:

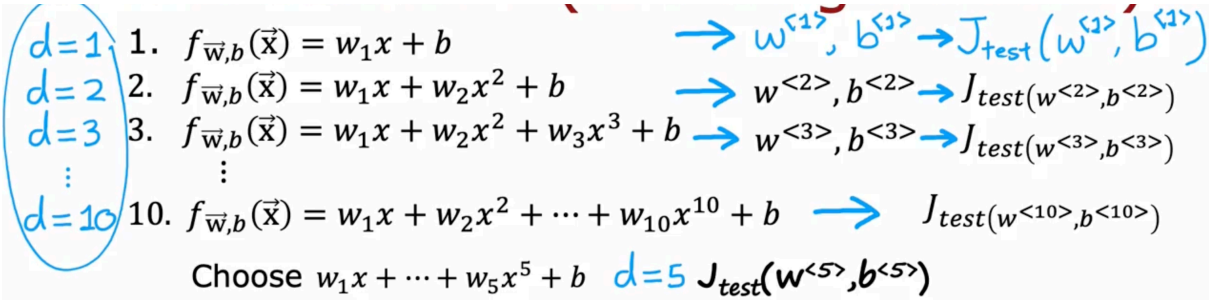
- A test set is used to evaluate the performance of a model after it has been trained.

- However, using the test set to choose model parameters can lead to an **optimistic estimate** of generalization error.

Flawed Procedure:

- If you fit a model to the training data and then select the model based on test set performance, the test error may not accurately reflect how the model will perform on new data.
- This is because the model has indirectly "seen" the test data through the selection process.

Example:



> 💡 **How well does the model perform? Report test set error $J_{test}(w^{<5>}, b^{<5>})$?**
>
> The problem: $J_{test}(w^{<5>}, b^{<5>})$ is likely to be an *optimistic estimate of generalization error* (i.e. $J_{test}(w^{<5>}, b^{<5>}) < \text{generalization error}$). Because an extra parameter d (degree of polynomial) was chosen using the **test set**. w, b are overly optimistic estimate of generalization error on training data.
>
> ⇒ When you use the test set to choose between models, **the test set is no longer truly "unseen" data**. You've essentially incorporated information from the test set into your training process, even though you didn't directly train on it.

💡 **Scenario:** You want to predict housing prices and need to choose the best polynomial degree.

The Flawed Process:

1. You train 10 different polynomial models (degrees 1-10) on your training data
2. You evaluate each model on the **same test set**
3. You find that degree 5 performs best with 85% accuracy on the test set
4. You report that your chosen model has 85% test accuracy

Why this is wrong: You've used the test set to make a decision (choosing degree 5), so the test set performance is now **optimistically biased**.

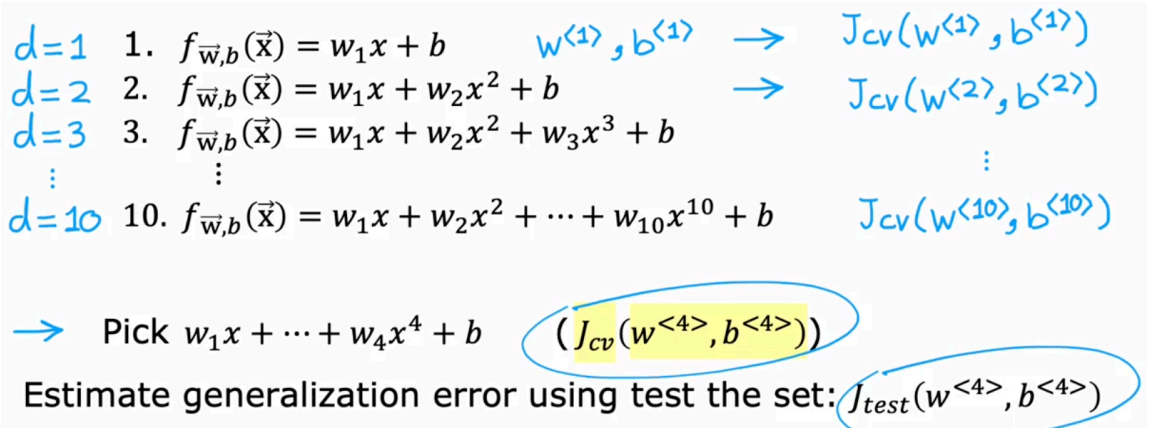
Introducing Cross-Validation

- To improve the model selection process, introduce a **cross-validation set**.
- This involves splitting the data into three subsets:
  - **Training Set**: Used to fit the model. (60%)
  - **Cross-Validation Set**: Used to evaluate different models and select the best one. (20%)
    - The cross-validation set is also called validation set, development set, or dev set.
  - **Test Set**: Used for final evaluation of the selected model. (20%)

The Cross-Validation is computed as following:

$$J_{cv}(\vec{w}, b) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} \left(f_{\vec{w},b}\left(\vec{x}_{cv}^{(i)}\right) - y_{cv}^{(i)} \right)^2$$

Procedure for Model Selection:



- Fit multiple models (e.g., different polynomial degrees) using the training set.
- Evaluate each model's performance on the cross-validation set to compute the **cross-validation error**.
- Select the model with the lowest cross-validation error.

For Neural Network Architecture Selection:

The same process applies when choosing neural network architectures:

Step 1: Train different network architectures on training data

- Small network → parameters w_1, b_1
- Medium network → parameters w_2, b_2
- Large network → parameters w_3, b_3

Step 2: Evaluate each architecture on cross-validation set

- Calculate classification error (fraction of misclassified examples)
- Compare J_{cv} for all three architectures

Step 3: Choose architecture with best cross-validation performance

- If medium network has lowest J_{cv}, select that architecture

Step 4: Final evaluation on test set

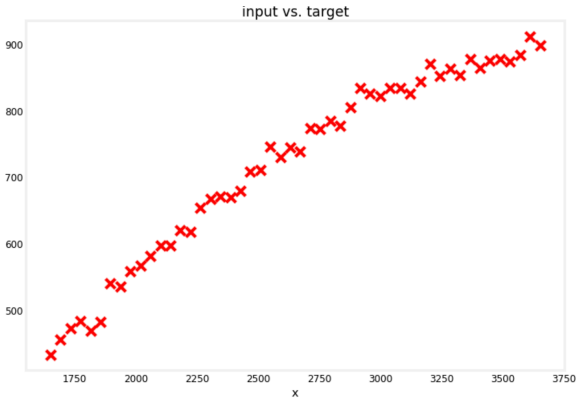
- Report J_{test} for chosen architecture as generalization estimate

Coding

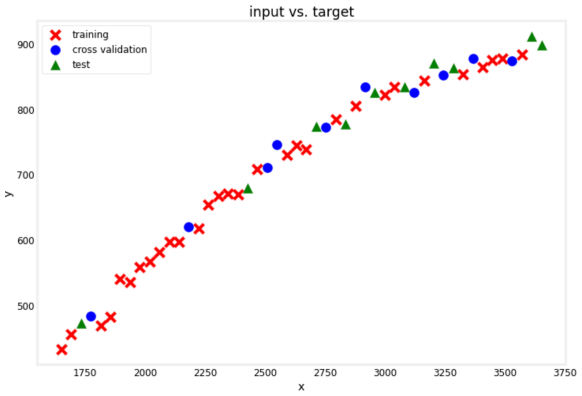
Regression

Datasets

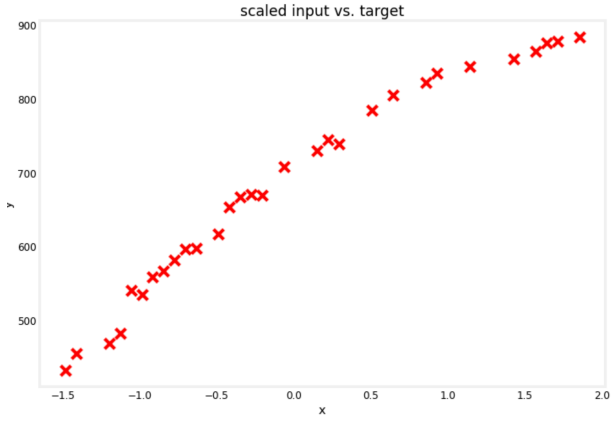
Normal:



After spliting:



Scale input of training set:



Adding Polynomial Features

You'll notice that the MSEs are significantly better for both the training and cross validation set when you added the 2nd order polynomial. You may want to introduce more polynomial terms and see which one gives the best performance. As shown in class, you can have 10 different models like this:

1. $f_{\vec{w},b}(\vec{x}) = w_1 x + b$
2. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + b$
3. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + b$

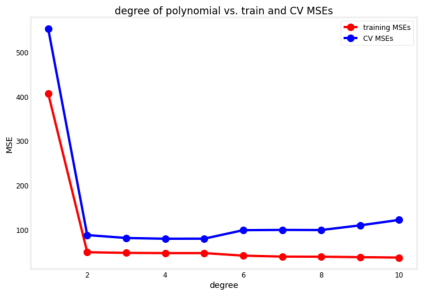
$\vdots$

10. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + \cdots + w_{10} x^{10} + b$

You can create a loop that contains all the steps in the previous code cells. Here is one implementation that adds polynomial features up to `degree=10`. We'll plot it at the end to make it easier to compare the results for each model.

```
# Initialize lists to save the errors, models, and feature transforms
train_mses = []
cv_mses = []
models = []
polys = []
scalers = []

# Loop over 10 times. Each adding one more degree of polynomial higher than the last.
for degree in range(1,11):

    # Add polynomial features to the training set
    poly = PolynomialFeatures(degree, include_bias=False)
    X_train_mapped = poly.fit_transform(x_train)
    polys.append(poly)

    # Scale the training set
    scaler_poly = StandardScaler()
    X_train_mapped_scaled = scaler_poly.fit_transform(X_train_mapped)
    scalers.append(scaler_poly)

    # Create and train the model
    model = LinearRegression()
    model.fit(X_train_mapped_scaled, y_train )
    models.append(model)

    # Compute the training MSE
    yhat = model.predict(X_train_mapped_scaled)
    train_mse = mean_squared_error(y_train, yhat) / 2
    train_mses.append(train_mse)

    # Add polynomial features and scale the cross validation set
    X_cv_mapped = poly.transform(x_cv)
    X_cv_mapped_scaled = scaler_poly.transform(X_cv_mapped)

    # Compute the cross validation MSE
    yhat = model.predict(X_cv_mapped_scaled)
    cv_mse = mean_squared_error(y_cv, yhat) / 2
```

```
    cv_mses.append(cv_mse)

# Plot the results
degrees=range(1,11)
utils.plot_train_cv_mses(degrees, train_mses, cv_mses, title="degree of polynomial vs. train and CV MSEs")
```



Choosing the best model

```
degree = np.argmin(cv_mses) + 1
X_test_mapped = polys[degree-1].transform(x_test)
X_test_mapped_scaled = scalers[degree-1].transform(X_test_mapped)
yhat = models[degree-1].predict(X_test_mapped_scaled)
test_mse = mean_squared_error(y_test, yhat) / 2
```

Neural Network

```
# Initialize lists that will contain the errors for each model
nn_train_mses = []
nn_cv_mses = []

# Build the models
nn_models = utils.build_models()

# Loop over the the models
for model in nn_models:

    # Setup the loss and optimizer
    model.compile(
    loss='mse',
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.1),
    )

    print(f"Training {model.name}...")

    # Train the model
    model.fit(
        X_train_mapped_scaled, y_train,
        epochs=300,
```

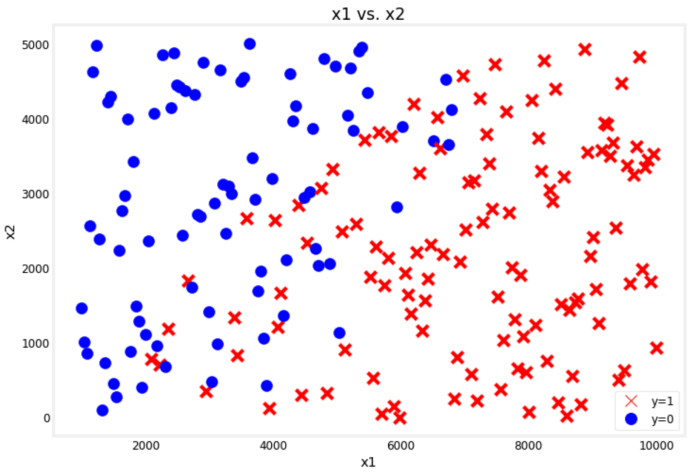
```
        verbose=0
    )

    print("Done!\n")


    # Record the training MSEs
    yhat = model.predict(X_train_mapped_scaled)
    train_mse = mean_squared_error(y_train, yhat) / 2
    nn_train_mses.append(train_mse)

    # Record the cross validation MSEs
    yhat = model.predict(X_cv_mapped_scaled)
    cv_mse = mean_squared_error(y_cv, yhat) / 2
    nn_cv_mses.append(cv_mse)
```

```
Training model_1...
Done!

Training model_2...
Done!

Training model_3...
Done!

RESULTS:
Model 1: Training MSE: 73.44, CV MSE: 113.87
Model 2: Training MSE: 73.40, CV MSE: 112.28
Model 3: Training MSE: 44.56, CV MSE: 88.51
```

```
Selected Model: 3
Training MSE: 44.56
Cross Validation MSE: 88.51
Test MSE: 87.77
```

Classification

Datasets



Evaluating the error for classification models

In the previous sections on regression models, you used the mean squared error to measure how well your model is doing. For classification, you can get a similar metric by getting the fraction of the data that the model has misclassified. For example, if your model made wrong predictions for 2 samples out of 5, then you will report an error of `40%` or `0.4`. The code below demonstrates this using a for-loop and also with Numpy's `mean()` function.

```
# Sample model output
probabilities = np.array([0.2, 0.6, 0.7, 0.3, 0.8])

# Apply a threshold to the model output. If greater than 0.5, set to 1. Else 0.
predictions = np.where(probabilities >= 0.5, 1, 0)

# Ground truth labels
ground_truth = np.array([1, 1, 1, 1, 1])

# Initialize counter for misclassified data
misclassified = 0

# Get number of predictions
num_predictions = len(predictions)

# Loop over each prediction
for i in range(num_predictions):

    # Check if it matches the ground truth
    if predictions[i] != ground_truth[i]:

        # Add one to the counter if the prediction is wrong
```

```
        misclassified += 1

# Compute the fraction of the data that the model misclassified
fraction_error = misclassified/num_predictions

print(f"probabilities: {probabilities}")
print(f"predictions with threshold=0.5: {predictions}")
print(f"targets: {ground_truth}")
print(f"fraction of misclassified data (for-loop): {fraction_error}")
print(f"fraction of misclassified data (with np.mean()): {np.mean(predictions != gro
und_truth)}")
```

```
probabilities: [0.2 0.6 0.7 0.3 0.8]
predictions with threshold=0.5: [0 1 1 0 1]
targets: [1 1 1 1 1]
fraction of misclassified data (for-loop): 0.4
fraction of misclassified data (with np.mean()): 0.4
```

Build and train the model

```
# Initialize lists that will contain the errors for each model
nn_train_error = []
nn_cv_error = []

# Build the models
models_bc = utils.build_models()

# Loop over each model
for model in models_bc:

    # Setup the loss and optimizer
    model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
    )

    print(f"Training {model.name}...")

    # Train the model
    model.fit(
        x_bc_train_scaled, y_bc_train,
```

```
        epochs=200,
        verbose=0
    )

    print("Done!\n")

    # Set the threshold for classification
    threshold = 0.5

    # Record the fraction of misclassified examples for the training set
    yhat = model.predict(x_bc_train_scaled)
    yhat = tf.math.sigmoid(yhat)
    yhat = np.where(yhat >= threshold, 1, 0)
    train_error = np.mean(yhat != y_bc_train)
    nn_train_error.append(train_error)

    # Record the fraction of misclassified examples for the cross validation set
    yhat = model.predict(x_bc_cv_scaled)
    yhat = tf.math.sigmoid(yhat)
    yhat = np.where(yhat >= threshold, 1, 0)
    cv_error = np.mean(yhat != y_bc_cv)
    nn_cv_error.append(cv_error)

# Print the result
for model_num in range(len(nn_train_error)):
    print(
        f"Model {model_num+1}: Training Set Classification Error: {nn_train_error[mo
del_num]:.5f}, " +
        f"CV Set Classification Error: {nn_cv_error[model_num]:.5f}"
        )
```

```
Model 1: Training Set Classification Error: 0.05833, CV Set Classification Error: 0.17
500
Model 2: Training Set Classification Error: 0.06667, CV Set Classification Error: 0.1
5000
Model 3: Training Set Classification Error: 0.05000, CV Set Classification Error: 0.1
5000
```

```
Selected Model: 3
Training Set Classification Error: 0.0500
```

CV Set Classification Error: 0.1500
Test Set Classification Error: 0.1750

💡 **QUES:** In the context of machine learning, what is a diagnostic?

- An application of machine learning to medical applications, with the goal of diagnosing patients' conditions.
- **A test that you run to gain insight into what is/isn't working with a learning algorithm.** 👍🏻
- A process by which we quickly try as many different ways to improve an algorithm as possible, so as to see what works.
- This refers to the process of measuring how well a learning algorithm does on a test set (data that the algorithm was not trained on).

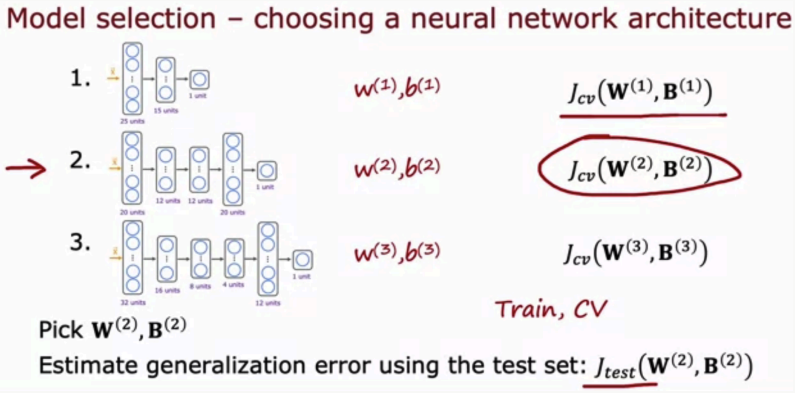
Explain: Yes! A diagnostic is a test that you run to gain insight into what is/isn't working with a learning algorithm, to gain guidance into improving its performance.

💡 **QUES:** True/False? It is always true that the better an algorithm does on the training set, the better it will do on generalizing to new data.

- True
- **False** 👍🏻

Explain: Actually, if a model overfits the training set, it may not generalize well to new data.

💡 **QUES:** For a classification task; suppose you train three different models using three different neural network architectures. Which data do you use to evaluate the three models in order to choose the best one?



- All the data -- training, cross validation and test sets put together.
- **The cross validation set** 👍🏻
- The test set
- The training set

Explain: Correct. Use the cross validation set to calculate the cross validation error on all three models in order to compare which of the three models is best.