Evaluating NLP Systems

Mostly in Python, using standard ML/NLP libraries that shows how to compute the metrics you described, organized by task:

- Classification (accuracy, precision, recall, F1, macro/micro)
- Machine Translation (BLEU, METEOR, TER-like edit distance)
- Question Answering (EM, token-level F1, MCQ accuracy)
- Language Modeling (per-token loss, perplexity)
- Text Generation (BLEU/ROUGE + simple human-ish proxies)

You can copy/paste and adapt pieces as needed.

---

1. Classification Metrics (Accuracy, Precision, Recall, F1)

Using scikit-learn for Binary / Multi-class Text Classification

```
from sklearn.metrics import accuracy_score,
        precision_recall_fscore_support, classification_report
import numpy as np

# Example: sentiment analysis or topic classification
y_true = ["pos", "neg", "pos", "neutral", "pos", "neg"]
y_pred = ["pos", "neg", "neg", "neutral", "pos", "pos"]

# 1) Basic accuracy
accuracy = accuracy_score(y_true, y_pred)
print("Accuracy:", accuracy)

# 2) Precision/Recall/F1 per class + macro/micro
# average=None -> returns metrics for each class
prec, rec, f1, support = precision_recall_fscore_support(
    y_true,
    y_pred,
    labels=["pos", "neg", "neutral"],
    average=None,
    zero_division=0
)
print("\nPer-class metrics:")
for label, p, r, f, s in zip(["pos", "neg", "neutral"], prec,
        rec, f1, support):
    print(f"Class: {label:7} | P={p:.3f} R={r:.3f} F1={f:.3f}
        Support={s}")
```

```
# 3) Macro-averaged (unweighted average over classes)
prec_macro, rec_macro, f1_macro, _ =
        precision_recall_fscore_support(
    y_true, y_pred, average="macro", zero_division=0
)
print("\nMacro-averaged: P={:.3f} R={:.3f} F1={:.3f}".format(
    prec_macro, rec_macro, f1_macro))

# 4) Micro-averaged (global counts of TP/FP/FN)
prec_micro, rec_micro, f1_micro, _ =
        precision_recall_fscore_support(
    y_true, y_pred, average="micro", zero_division=0
)
print("Micro-averaged: P={:.3f} R={:.3f} F1={:.3f}".format(
    prec_micro, rec_micro, f1_micro))

# 5) Sklearn built-in classification report
print("\nClassification Report:")
print(classification_report(y_true, y_pred, zero_division=0))
```

Key points:

- For **balanced** data, `accuracy` is often OK.
- For **imbalanced** data, look carefully at class-wise `precision`, `recall`, `F1`, and macro F1.

---

2. Machine Translation Metrics (BLEU, METEOR, TER-like)

We'll use **nltk** and **sacrebleu** as examples.

2.1 BLEU with NLTK (simple)

```
import nltk
from nltk.translate.bleu_score
        import sentence_bleu, corpus_bleu, SmoothingFunction

# make sure you did: nltk.download('punkt') if needed

references = [
    ["the", "cat", "is", "on", "the", "mat"],
]
candidate = ["the", "cat", "is", "on", "mat"]

# BLEU-4 with default weights (0.25 each)
smooth_fn = SmoothingFunction().method1
```

```
score_sentence = sentence_bleu(
    [references],  # list of reference lists (can have multiple
        refs)
    candidate,
    smoothing_function=smooth_fn
)
print("Sentence BLEU-4:", score_sentence)
```

2.2 Corpus BLEU with SacreBLEU (more realistic)

```
import sacrebleu

# Example corpus-level evaluation
refs = [
    ["The cat is on the mat.", "The cat sits on the mat."]  #
        multiple refs per source
]
sys = [
    "The cat is on mat."
]

# sacrebleu expects: list of sys outputs, and list-of-lists of
        references
bleu = sacrebleu.corpus_bleu(sys, list(zip(*refs)))  # transpose
        refs to match API
print("Corpus BLEU:", bleu.score)
```

2.3 METEOR (using NLTK as a proxy)

```
from nltk.translate.meteor_score import meteor_score

reference = "The cat is on the mat".split()
candidate = "The cat is on mat".split()

meteor = meteor_score([reference], candidate)
print("METEOR:", meteor)
```

2.4 TER-like Edit Distance (illustrative)

TER itself is more complex (shift operations etc.), but you can approximate with edit distance:

```
import nltk

ref = "The cat is on the mat".split()
cand = "The cat is on mat".split()

# Simple Levenshtein distance on tokens
distance = nltk.edit_distance(ref, cand)
```

```
ter_like = distance / len(ref)
print("Edit distance:", distance)
print("TER-like (distance / len(ref)):", ter_like)
```

---

3. Question Answering Metrics

3.1 Exact Match (EM) and Token-level F1 (SQuAD-style)

```
import re
import string

def normalize_answer(s):
    """Lower text and remove punctuation, articles and extra
        whitespace."""
    def remove_articles(text):
        return re.sub(r"\b(a|an|the)\b", " ", text)

    def white_space_fix(text):
        return " ".join(text.split())

    def remove_punc(text):
        exclude = set(string.punctuation)
        return "".join(ch for ch in text if ch not in exclude)

    def lower(text):
        return text.lower()

    return
        white_space_fix(remove_articles(remove_punc(lower(s))))

def get_tokens(s):
    if not s:
        return []
    return normalize_answer(s).split()

def exact_match(prediction, ground_truth):
    return int(normalize_answer(prediction) ==
        normalize_answer(ground_truth))

def f1_score_qa(prediction, ground_truth):
    pred_tokens = get_tokens(prediction)
    gold_tokens = get_tokens(ground_truth)
    common = set(pred_tokens) & set(gold_tokens)
    num_same = sum(min(pred_tokens.count(w),
        gold_tokens.count(w)) for w in common)

    if len(pred_tokens) == 0 or len(gold_tokens) == 0:
```

```
        return int(pred_tokens == gold_tokens)
    if num_same == 0:
        return 0.0

    precision = num_same / len(pred_tokens)
    recall = num_same / len(gold_tokens)
    f1 = 2 * precision * recall / (precision + recall)
    return f1

# Example
gold_answers = [
    "Paris",
    "The capital of France is Paris",
]
predictions = [
    "Paris",
    "Paris city"
]

for pred, gold in zip(predictions, gold_answers):
    em = exact_match(pred, gold)
    f1 = f1_score_qa(pred, gold)
    print(f"Pred: {pred!r} | Gold: {gold!r} | EM={em},
        F1={f1:.3f}")
```

3.2 Multiple-Choice QA: Accuracy

```
import numpy as np

# Suppose 4-option MCQ: answers are indices 0..3
y_true = [2, 1, 0, 3]  # correct options
y_pred = [2, 0, 0, 3]  # model predictions

accuracy_mcq = np.mean([int(t == p) for t, p in zip(y_true,
        y_pred)])
print("MCQ Accuracy:", accuracy_mcq)
```

---

4. Language Modeling Metrics: Loss and Perplexity

Below is a **toy** PyTorch example for next-token prediction.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# Dummy vocabulary and model (toy)
```

```
vocab_size = 1000
embedding_dim = 64
hidden_dim = 128

class ToyLM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim,
        batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, input_ids):
        # input_ids: (batch, seq_len)
        x = self.embed(input_ids)
        outputs, _ = self.lstm(x)
        logits = self.fc(outputs)  # (batch, seq_len, vocab_size)
        return logits

model = ToyLM(vocab_size, embedding_dim, hidden_dim)

# Example batch: sequences of token ids
batch_size = 2
seq_len = 5
input_ids = torch.randint(0, vocab_size, (batch_size, seq_len))

# For next-token prediction, targets are input shifted by one
targets = input_ids[:, 1:].contiguous()      # (batch, seq_len-1)
inputs_for_lm = input_ids[:, :-1].contiguous()

logits = model(inputs_for_lm)
        # (batch, seq_len-1, vocab_size)
logits_flat = logits.view(-1, vocab_size)    #
        ((batch*(seq_len-1)), vocab_size)
targets_flat = targets.view(-1)

loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(logits_flat, targets_flat)    # average negative
        log-likelihood

# Perplexity
perplexity = torch.exp(loss)

print("Average cross-entropy loss per token:", loss.item())
print("Perplexity:", perplexity.item())
```

4.1 Manual Perplexity from Probabilities (Your Numeric Example)

```
import math

probs = [0.5, 0.4, 0.2]  # P(I), P(love), P(NLP)

nlls = [-math.log(p) for p in probs]   # natural log
L = sum(nlls) / len(nlls)
ppl = math.exp(L)

print("NLLs:", nlls)
print("Average loss L:", L)
print("Perplexity:", ppl)
```

---

5. Text Generation Metrics (BLEU, ROUGE, Simple Heuristics)

For **long-form text** (summaries, stories), we often use ROUGE and BLEU, plus human evaluation.

5.1 ROUGE with `rouge-score` (TensorFlow's implementation)

```
from rouge_score import rouge_scorer

reference = "The cat is on the mat."
candidate = "The cat sits on the mat."

scorer = rouge_scorer.RougeScorer(["rouge1", "rougeL"],
        use_stemmer=True)
scores = scorer.score(reference, candidate)

print("ROUGE-1:", scores["rouge1"])
print("ROUGE-L:", scores["rougeL"])
```

5.2 Combining Automatic Metrics + Simple Heuristics

```
import numpy as np

def repetition_ratio(text, n=3):
    """Very crude repetition measure based on repeated n-
        grams."""
    tokens = text.split()
    if len(tokens) < n:
        return 0.0
```

```
    ngrams = [" ".join(tokens[i:i+n]) for i in range(len(tokens)
        - n + 1)]
    total = len(ngrams)
    unique = len(set(ngrams))
    return 1.0 - unique / total  # higher = more repetition

generated = "The cat is on the mat. The cat is on the mat. The
        cat is on the mat."
rep = repetition_ratio(generated, n=3)
print("Repetition ratio (3-gram):", rep)
```

In real projects you might:

- Log **perplexity** for the model.
- Compute **BLEU/ROUGE** vs references (for summarization/translation).
- Sample outputs and run **human evaluation** for:
  - Fluency (1–5)
  - Coherence (1–5)
  - Relevance (1–5)
  - Diversity (e.g., repetition ratio, unique n-grams, etc.)

---

6. Connecting Metrics to Training / Fine-tuning

6.1 Tracking Pre-training Loss and Perplexity

```
def evaluate_lm(model, data_loader, device="cpu"):
    model.eval()
    total_loss = 0.0
    total_tokens = 0

    loss_fn = nn.CrossEntropyLoss(reduction="sum")
    with torch.no_grad():
        for batch in data_loader:
            # assume batch["input_ids"] shape: (batch, seq_len)
            input_ids = batch["input_ids"].to(device)
            inputs = input_ids[:, :-1]
            targets = input_ids[:, 1:]

            logits = model(inputs)
            vocab_size = logits.size(-1)
            loss = loss_fn(logits.view(-1, vocab_size),
        targets.reshape(-1))
            total_loss += loss.item()
            total_tokens += targets.numel()

    avg_loss = total_loss / total_tokens
```

```
    ppl = math.exp(avg_loss)
    return avg_loss, ppl

# Later in training loop:
# val_loss, val_ppl = evaluate_lm(model, val_loader)
# print(f"Validation loss: {val_loss:.4f}, PPL: {val_ppl:.2f}")
```

6.2 Task-specific Fine-tuning Metrics

You'd typically compute metrics like this at the end of each epoch:

```
def evaluate_classifier(model, dataloader, device="cpu"):
    model.eval()
    all_preds, all_labels = [], []

    with torch.no_grad():
        for batch in dataloader:
            input_ids = batch["input_ids"].to(device)
            labels = batch["labels"].to(device)

            logits = model(input_ids)                    # (batch,
        num_classes)
            preds = logits.argmax(dim=-1)

            all_preds.extend(preds.cpu().tolist())
            all_labels.extend(labels.cpu().tolist())

    acc = accuracy_score(all_labels, all_preds)
    prec, rec, f1, _ = precision_recall_fscore_support(
        all_labels, all_preds, average="macro", zero_division=0
    )
    return {
        "accuracy": acc,
        "precision_macro": prec,
        "recall_macro": rec,
        "f1_macro": f1,
    }

# usage:
# metrics = evaluate_classifier(model, val_loader)
# print(metrics)
```

Summary

- **Classification**: use `accuracy_score`, `precision_recall_fscore_support` / `classification_report` (macro vs micro F1).

- **Machine Translation**: `sacrebleu` for BLEU; NLTK for sentence BLEU & METEOR; edit distance as a simple TER-like measure.
- **QA**: implement SQuAD-style **EM** and **token-level F1**, plus accuracy for MCQ.
- **Language Modeling**: compute **cross-entropy loss** and **perplexity** from logits.
- **Generation**: use ROUGE/BLEU and simple repetition metrics; combine with human evaluation for fluency, coherence, relevance.