**Japanese Politeness Analysis**

**1. Data Collection**

Since you don’t have data yet, your first step is to gather Japanese text with varying levels of politeness. Possible sources:

* **Japanese dialogue datasets** (e.g., OpenSubtitles, BCCWJ)
* **Crowdsourced annotations** (ask native speakers to rate politeness levels)
* **Web scraping** (extract text from sources like Twitter, news articles, and blogs)
* **Synthetic dataset** (manually construct polite, neutral, and impolite sentences)

**Libraries for Data Collection**

* **BeautifulSoup/Scrapy** (for web scraping)
* **Google Drive API** (if collecting data via shared documents)
* **Twitter API (Tweepy)** (for tweets with informal/polite language)

**2. Data Preprocessing**

Japanese text needs careful preprocessing:

* **Tokenization** → Since Japanese lacks spaces, use **MeCab**, **SudachiPy**, or **Juman++**.
* **Annotation**

**✅ Measure Inter-Annotator Agreement (Even If It’s Just You)**

Usually, **IAA (Inter-Annotator Agreement)** is measured using **Cohen’s Kappa** (for two annotators) or **Fleiss’ Kappa** (for three or more).

Since you’re working alone, you can:

1. **Annotate the same dataset twice (pass 1 vs. pass 2).**
2. **Calculate Cohen’s Kappa** to measure how much you agree with yourself.

* **Feature Engineering** → Extract politeness-related features:
  + Presence of **です/ます** forms
  + Use of **だ** vs. **です**
  + **Omission of particles** (は, を, が, etc.)
  + Sentence completeness (predicate presence)
  + Honorifics (**お〜, ご〜, 〜様, 〜さん**)
  + Verb forms (e.g., **ていただく** vs. **くれる**)
* **Sentence Embeddings** → Use a transformer model (e.g., **BERT, T5, or XLM-R**) to generate numerical embeddings.

**Libraries for Preprocessing**

* **MeCab/SudachiPy/Juman++** (morphological analysis)
* **fugashi** (wrapper for MeCab in Python)
* **transformers (Hugging Face)** (pretrained Japanese BERT)
* **pandas/numpy** (data handling)

**3. Model Selection**

You are considering a **transformer model + logistic regression**, which is a solid hybrid approach.

**Model Options**

1. **Transformer-based Feature Extractor**
   * Use a pretrained **Japanese BERT (e.g., cl-tohoku/bert-base-japanese)**
   * Convert each sentence into a vector embedding
2. **Logistic Regression / Linear Model**
   * Take transformer embeddings + handcrafted politeness features
   * Perform classification (binary or multinomial)

**Libraries for Model Training**

* **Hugging Face transformers** (for BERT-based embeddings)
* **scikit-learn** (for logistic regression)
* **PyTorch/TensorFlow** (if fine-tuning BERT)

A more **robust** model would likely come from **fine-tuning a transformer model** rather than using **frozen embeddings with logistic regression**. Here’s why:

1. Why Fine-Tuning a Transformer Is More Robust

**✅ Context Awareness**

* Japanese politeness is often **context-dependent**, relying on subtle differences in structure (e.g., omission of particles, verb forms, honorifics).
* A fine-tuned transformer **learns deeper syntactic and semantic structures**, making it better at capturing these nuances than handcrafted features.

**✅ Better Feature Extraction**

* **Pretrained BERT embeddings** are generic and may not focus on politeness-related cues.
* Fine-tuning allows the model to **specialize** in politeness detection by adjusting weights based on training data.

**✅ Adaptability**

* If new politeness trends emerge (e.g., internet slang, youth speech), a fine-tuned transformer can **adapt better** than a fixed feature-based model.

**✅ Multiclass Scalability**

* Logistic regression might work well for binary classification, but as you scale to **multinomial (polite, neutral, impolite)**, a fine-tuned transformer can model the **continuous politeness spectrum** more effectively.

2. When Logistic Regression Might Be Preferable

**✅ Smaller Datasets**

* If you don’t have **enough labeled data**, fine-tuning could lead to **overfitting**. Logistic regression can still perform well with fewer samples.

**✅ Interpretability**

* Logistic regression provides **clear feature importance**, so you can see which politeness markers contribute to classification.
* Transformers are **black boxes**, making it harder to interpret why a sentence is classified as polite/impolite.

**✅ Computational Efficiency**

* Logistic regression is much **lighter** on resources. Fine-tuning a transformer requires **GPUs** and longer training times.

3. Best Approach?

* If you have **a lot of labeled politeness data** → **Fine-tune a Japanese BERT model**.
* If you have **limited data** and need quick results → **Use frozen embeddings + logistic regression**.
* A **hybrid approach** is also possible:
  + **Fine-tune a transformer** on politeness classification.
  + Use **logistic regression** for additional explainability.

If you have fewer than 1000 labeled texts, **I strongly recommend a hybrid approach: fine-tuned transformer + feature-based logistic regression**. This strategy balances robustness (from the transformer) and explainability/generalization (from logistic regression).

**4. Model Training Process**

**(A) Feature Extraction with Transformer**

**(B) Combine with Handcrafted Features**

**(C) Train Logistic Regression Model**

**5. Evaluation & Fine-tuning**

* Use **F1-score, Precision, Recall** for performance measurement.
* If using a multinomial model, consider **confusion matrices**.
* **Hyperparameter tuning** for logistic regression (e.g., C for regularization).
* Experiment with different embeddings (e.g., **word2vec, FastText**).
* If performance is low, **fine-tune BERT** instead of using frozen embeddings.  
    
  **Baseline**: a very simple model that yields a quantity your model should certainly beat.  
  Ex) Majority class predictor (the model that always tells you “It’s positive!!” regardless of given inputs.   
  => Rule based model??

**Libraries for Evaluation**

* **scikit-learn** (classification metrics)
* **Matplotlib/Seaborn** (confusion matrices)

**6. Deployment & Application**

* **API Endpoint**: Serve the model with **Flask** or **FastAPI**.
* **Web/App Integration**: Use Streamlit for a quick UI.
* **Batch Inference**: Process multiple texts in parallel.

**Deployment Libraries**

* **Flask/FastAPI** (for API deployment)
* **Streamlit** (for interactive UI)
* **ONNX/TensorFlow Lite** (for lightweight inference)