

Claude 4 Opus chat on: Step 1 - extract Test inscriptions for POS

User:
NEReus /

User:
metadata: BB

Hi Claude. Today we're going to be working on some research around named entity recognition in Latin inscriptions.

To begin testing, we're going to want to evaluate how various part of speech taggers do with inscriptions.

To work with that, we're going to need to write a script to unpack the parquet and provide 20 random sources from the 500k source list.

I'll provide you the metadata. What I would like you to do is ask us one question at a time until we have a full spec for this first step.

2 / 2

Assistant:

Great! I'd be happy to help you build a script for working with Latin inscriptions and part of speech tagging evaluation.

Let me start with the first question to understand your data structure:

Question 1: What is the structure/schema of the parquet file you'll be working with? Specifically, what columns does it contain and what kind of data is in each column (e.g., inscription text, source ID, metadata fields, etc.)?

User:
metadata: BB

Attached is the csv with metadata. The LIST_v1-2.parquet file should have the exact same. Let's start by verifying it.

Assistant:

Perfect! Let me examine the metadata CSV to understand the structure of your parquet file.



metadata:

Excellent! I can see the parquet file has 66 columns with comprehensive information about Latin inscriptions. I see there are multiple text fields including `clean_text_conservative`, `clean_text_interpretive_sentence`, `clean_text_interpretive_word`, and others.

Now for my next question:

Question 2: For the 20 random sources you want to extract, which specific columns/attributes do you need in your output? For example:

- Just the inscription text (and if so, which version - conservative, interpretive, diplomatic)?
- The unique IDs (LIST-ID, EDH-ID, EDCS-ID)?
- Dating information?
- Geographic information?
- All columns?

User:

metadata: BB

LIST-ID, inscription text(conservative), inscription text (interpretive), dating information, geographic information.

We are going to want to make sure these are lengthy. They should be in the top decile of length of all inscriptions in the full dataset.

2 / 2

Assistant:

Good! So we need to filter for the longest inscriptions (top 10% by length) and extract specific fields. Let me clarify one detail:

Question 3: For the interpretive inscription text, I see there are three versions available:

- `clean_text_interpretive_sentence` (preserves interpunction between sentences)
- `clean_text_interpretive_word` (no interpunction between sentences)
- `clean_text_interpretive_word_EDCS` (EDCS-specific version)

Which interpretive version would you prefer for your POS tagging evaluation?

User:

metadata: BB

`clean_text_interpretive_word` and `clean_text_interpretive_sentence`

Assistant:

Perfect! Now I need to understand how to determine the length for filtering:

Question 4: When filtering for the top decile (10%) of inscription length, which text field should I use to calculate the length? Should I use:

- The conservative text length?

- One of the interpretive text lengths?
- The maximum length across all three text versions?
- A specific preference?

This will determine which inscriptions qualify as being in the "longest 10%".

User:

metadata: BB

clean_text_interpretive_word

Assistant:

Great! So we'll use `clean_text_interpretive_word` length to identify the top 10% longest inscriptions.

Question 5: What output format would you like for these 20 random inscriptions? Options include:

- CSV file
 - JSON file
 - Excel file
 - Python pickle/parquet
 - Multiple individual text files
 - Console output/display
 - Other format?
-

User:

metadata: BB

json, please.

Assistant:

Excellent! One more clarification about the geographic information:

Question 6: For geographic information, I see there are several geographic fields available:

- Latitude and Longitude (coordinates)
- geometry (coordinates in a specific format)
- Location names: `findspot_clean`, `findspot_modern_clean`, `findspot_ancient_clean`
- Administrative divisions: `country_clean`, `province_label_clean`, `modern_region_clean`
- Urban context: `urban_context`, `urban_context_city`

Which geographic fields would you like included in the JSON output?

User:

metadata: BB

geometry, findspot_ancient_clean, urban_context_city

Assistant:

Perfect! Now just one final question to complete the specification:

Question 7: What should be the name/path of:

- The input parquet file (you mentioned "LIST_v1-2.parquet" - is that the exact filename?)
- The output JSON file (what would you like to call it?)

And should the script handle any specific cases like missing/null values in the text fields when calculating lengths?

User:

metadata: BB

POS-LIST-test1.json

And if it's null, just clean it before calculating.

Assistant:

Perfect! I now have all the specifications. Let me create a Python script that:

1. Reads the LIST_v1-2.parquet file
2. Filters for the top 10% longest inscriptions (based on clean_text_interpretive_word length, handling nulls)
3. Randomly selects 20 inscriptions from that filtered set
4. Extracts the specified fields (LIST-ID, both conservative and interpretive texts, dating info, and geographic info)
5. Saves the results to POS-LIST-test1.json

Perfect! I've created a Python script that meets all your specifications. Here's what it does:

Script Features:

1. **Reads** the LIST_v1-2.parquet file
2. **Calculates lengths** using clean_text_interpretive_word, handling nulls by treating them as empty strings
3. **Filters** for the top 10% longest inscriptions (90th percentile and above)
4. **Randomly samples** 20 inscriptions from that filtered set
5. **Extracts** the specified fields:
 - LIST-ID
 - clean_text_conservative
 - clean_text_interpretive_word
 - clean_text_interpretive_sentence
 - not_before & not_after (dating)
 - geometry, findspot_ancient_clean, urban_context_city (geography)
6. **Outputs** to POS-LIST-test1.json with clean JSON structure

Output Structure:

Each inscription in the JSON will have:

User:

json

```
{
  "LIST-ID": "...",
  "text_conservative": "...",
  "text_interpretive_word": "...",
  "text_interpretive_sentence": "...",
  "dating": {
    "not_before": ...,
    "not_after": ...
  },
  "geography": {
    "geometry": "...",
    "findspot_ancient": "...",
    "urban_context_city": "..."
  },
  "text_length": ...
}
```

Additional Features:

- Uses random seed (42) for reproducibility
- Provides detailed console output with statistics
- Reports any missing data
- Handles edge cases (missing columns, null values)

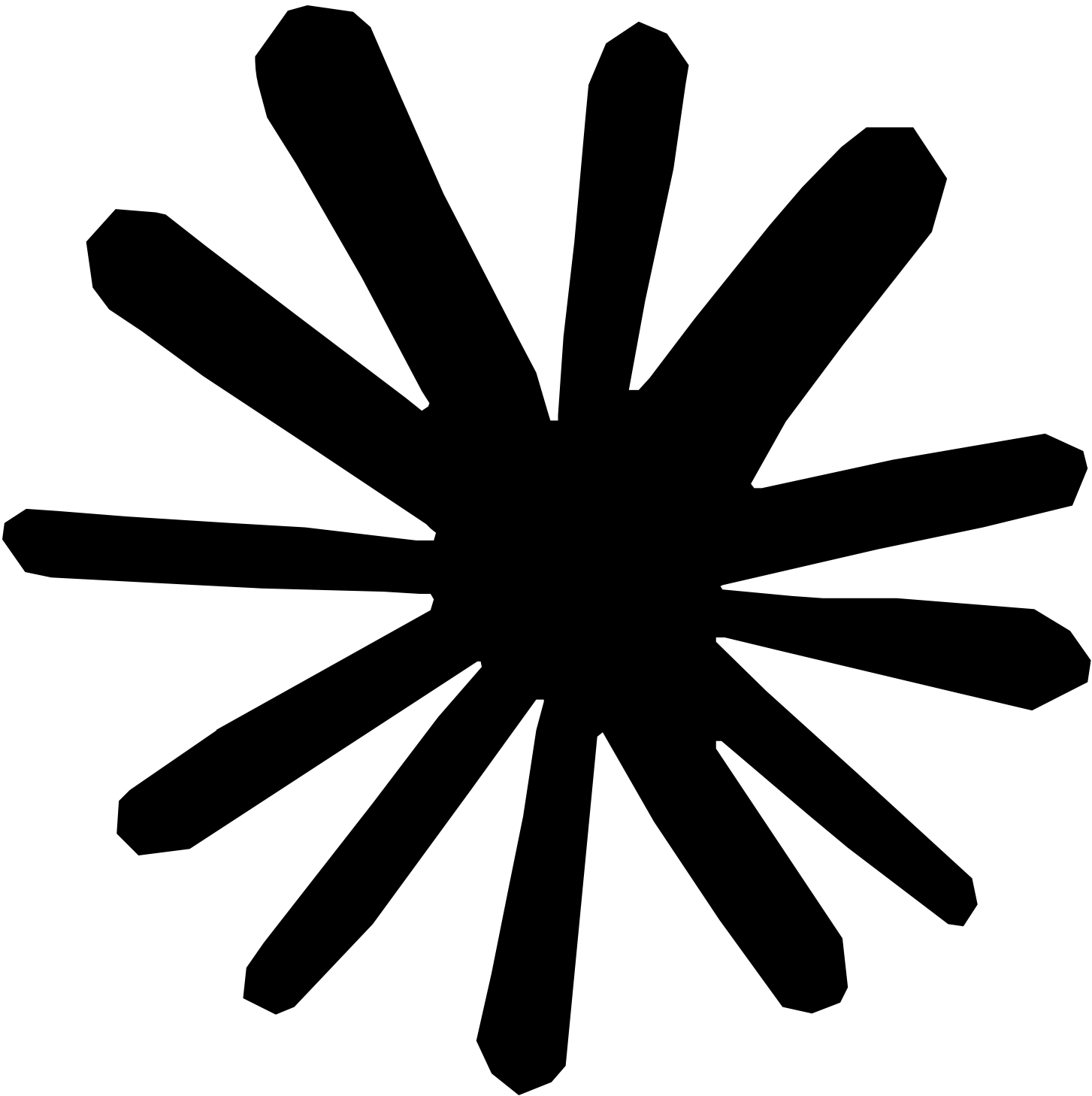
To run the script, simply execute:

User:

bash

```
python script_name.py
```

The script will automatically look for LIST_v1-2.parquet in the current directory and output POS-LIST-test1.json. Would you like me to add any additional features, such as command-line arguments for flexibility or additional validation checks?



[Claude can make mistakes.](#)
[Please double-check responses.](#)

Choose Files

No file chosen

metadata:

```
#!/usr/bin/env python3
''''''
```

Script to extract 20 random Latin inscriptions from the top decile by length

for POS tagging evaluation.

"""

```
import pandas as pd
import numpy as np
import json
import random
from pathlib import Path
```

```
def clean_text_for_length(text):
    """Clean text field for length calculation, handling nulls."""
    if pd.isna(text) or text is None:
        return ""
    return str(text).strip()
```

```
def extract_inscriptions_for_pos_testing(
    input_parquet="LIST_v1-2.parquet",
    output_json="POS-LIST-test1.json",
    n_samples=20,
    random_seed=42
):
    """
```

Extract random sample of inscriptions from top decile by length.

Parameters:

```
input_parquet : str
    Path to input parquet file
output_json : str
    Path to output JSON file
n_samples : int
    Number of random samples to extract (default: 20)
random_seed : int
    Random seed for reproducibility (default: 42)
```

"""

```
print(f"Reading parquet file: {input_parquet}")
try:
    df = pd.read_parquet(input_parquet)
    print(f"Successfully loaded {len(df):,} inscriptions")
```

```

except FileNotFoundError:
    print(f'Error: Could not find file {input_parquet}')
    return
except Exception as e:
    print(f'Error reading parquet file: {e}')
    return

# Verify required columns exist
required_cols = [
    'LIST-ID',
    'clean_text_conservative',
    'clean_text_interpretive_word',
    'clean_text_interpretive_sentence',
    'not_before',
    'not_after',
    'geometry',
    'findspot_ancient_clean',
    'urban_context_city'
]

missing_cols = [col for col in required_cols if col not in df.columns]
if missing_cols:
    print(f'Warning: Missing columns in parquet file: {missing_cols}')
    print("Available columns:", df.columns.tolist()[:10], "...")

# Calculate text lengths for filtering
print("\nCalculating text lengths...")
df['text_length'] = df['clean_text_interpretive_word'].apply(
    lambda x: len(clean_text_for_length(x))
)

# Remove entries with zero length
df_with_text = df[df['text_length'] > 0].copy()
print(f'Inscriptions with text: {len(df_with_text):,}')

# Calculate the 90th percentile threshold
length_threshold = df_with_text['text_length'].quantile(0.9)
print(f"\n90th percentile length threshold: {length_threshold:.0f} characters")

# Filter for top decile

```



```
df_top_decile = df_with_text[df_with_text['text_length'] >= length_threshold].copy()
print(f"Inscriptions in top decile: {len(df_top_decile):,}")

# Set random seed for reproducibility
random.seed(random_seed)
np.random.seed(random_seed)

# Sample 20 random inscriptions
n_to_sample = min(n_samples, len(df_top_decile))
if n_to_sample < n_samples:
    print(f"\nWarning: Only {n_to_sample} inscriptions available in top decile")

sampled_df = df_top_decile.sample(n=n_to_sample, random_state=random_seed)
print(f"\nSampled {len(sampled_df)} inscriptions")

# Prepare output data
output_data = []

for idx, row in sampled_df.iterrows():
    inscription_data = {
        'LIST-ID': row.get('LIST-ID'),
        'text_conservative': row.get('clean_text_conservative'),
        'text_interpretive_word': row.get('clean_text_interpretive_word'),
        'text_interpretive_sentence': row.get('clean_text_interpretive_sentence'),
        'dating': {
            'not_before': int(row['not_before']) if pd.notna(row.get('not_before')) else None,
            'not_after': int(row['not_after']) if pd.notna(row.get('not_after')) else None
        },
        'geography': {
            'geometry': row.get('geometry'),
            'findspot_ancient': row.get('findspot_ancient_clean'),
            'urban_context_city': row.get('urban_context_city')
        },
        'text_length': int(row['text_length'])
    }
    output_data.append(inscription_data)

# Sort by LIST-ID for consistent output
output_data.sort(key=lambda x: x['LIST-ID'] if x['LIST-ID'] else "")
```

```

# Write to JSON file
print(f"\nWriting output to: {output_json}")
with open(output_json, 'w', encoding='utf-8') as f:
    json.dump(output_data, f, ensure_ascii=False, indent=2)

# Print summary statistics
print("\n" + "="*50)
print("EXTRACTION COMPLETE")
print("="*50)
print(f"Output file: {output_json}")
print(f"Number of inscriptions: {len(output_data)}")

lengths = [item['text_length'] for item in output_data]
print(f"\nText length statistics:")
print(f"  Min length: {min(lengths):,} characters")
print(f"  Max length: {max(lengths):,} characters")
print(f"  Mean length: {np.mean(lengths):,.0f} characters")
print(f"  Median length: {np.median(lengths):,.0f} characters")

# Check for missing data
missing_stats = {
    'conservative_text': sum(1 for item in output_data if not item['text_conservative']),
    'interpretive_word': sum(1 for item in output_data if not item['text_interpretive_word']),
    'interpretive_sentence': sum(1 for item in output_data if not item['text_interpretive_sentence']),
    'not_before': sum(1 for item in output_data if item['dating']['not_before'] is None),
    'not_after': sum(1 for item in output_data if item['dating']['not_after'] is None),
    'geometry': sum(1 for item in output_data if not item['geography']['geometry']),
    'findspot_ancient': sum(1 for item in output_data if not item['geography']['findspot_ancient']),
    'urban_context_city': sum(1 for item in output_data if not item['geography']['urban_context_city'])
}

if any(missing_stats.values()):
    print("\nMissing data counts:")
    for field, count in missing_stats.items():
        if count > 0:
            print(f"  {field}: {count}/{len(output_data)}")

return output_data

if __name__ == "__main__":

```

```
# Run the extraction
```

```
extract_inscriptions_for_pos_testing()
```

