



Teaching Textual Analysis to Economics Majors¹

The proliferation of text-as-data methods in economics research has not yet been matched by curricular innovation at the undergraduate level. This article presents a pedagogical framework for introducing textual analysis to economics majors, using job postings as a case study in labor economics. The tutorial guides students through the complete analytical pipeline — from raw text processing (tokenization, vectorization) to economic applications (topic modelling, similarity, sentiment analysis, and prediction). By providing fully replicable teaching materials and explanations of Python code, this paper equips students with in-demand data science skills, thereby enhancing their ability to conduct novel and policy-relevant research using text data.

Cheuk Yin Ho[†]

[†]The Chinese University of Hong Kong

¹ I am grateful to the editor and two anonymous referees for their comments. Funding support from The Chinese University of Hong Kong is appreciated.

1. Introduction

Text data, such as blog posts, news articles, song lyrics, job postings, and product reviews, are ubiquitous in our daily lives. Economists have recognized the power of text data and are thus increasingly using textual analysis in academic research (Gentzkow, Kelly, & Taddy, 2019; Loughran & McDonald, 2020; Ash & Hansen, 2023; Hassan et al., 2025). Some studies have used textual analysis to measure economic policy uncertainty from newspapers (Baker, Bloom, & Davis, 2016), analyse Federal Reserve communication from FOMC transcripts (Hansen, McMahon, & Prat, 2018), extract skill demands from job postings (Deming & Kahn, 2018), and measure corporate culture from earnings call transcripts (Li et al., 2021). Working with text data, however, has not been incorporated into typical undergraduate curricula.

Although integrating data science and computational skills into economics has been emphasized in the profession (Athey & Luca, 2019), their practical implementation in undergraduate economics has only recently begun. For example, instructors use Python programming to teach microeconomic theory (Kuroki, 2021) and to simulate macroeconomic models (Jenkins, 2022). In particular, Batt et al. (2020) demonstrate data visualization using Tableau, and Kuroki (2023) incorporate data science modules directly into econometrics courses.

Economics majors develop proficiency in econometric modelling with structured data; however, they often lack training in processing and analysing unstructured text data. Consequently, undergraduate research is often confined to structured datasets, such as the Panel Studies of Income Dynamics (PSID), the World Bank's open data, and the Federal Reserve Economic Data (FRED) database. This limits students' capabilities to pursue novel and diverse research topics that require original data collection from text-based sources. For instance, measuring the impact of local news sentiment on housing market dynamics or identifying emerging skill requirements in green jobs from job postings is very difficult without the skills to process unstructured text.

To bridge the gap between modern text-as-data research and undergraduate economics education, this article offers a step-by-step tutorial on teaching foundational textual analysis to economics majors. This tutorial is suitable for students with no prior programming experience. We introduce Python programming from the ground up, integrating foundational coding concepts with textual analysis techniques. Students simultaneously learn programming fundamentals and applied textual analysis, making abstract coding concepts concrete and economically relevant.

Job postings are employed to illustrate various applications of text data in labor economics. Traditional undergraduate courses in labor economics typically require students to use cross-sectional data on wages and education to estimate the returns to schooling. However, more recent concepts—such as job similarities and job-skill matching—are rarely discussed, largely due to the lack of pre-processed job data and suitable analytical techniques. This gap underscores the need for a tutorial on textual analysis for economics majors.

2. Learning Objectives

Students will be able to learn the following skills after completing the tutorial:

- Develop foundational programming skills in Python.

- Process raw text data and transform it into tokens.

Convert tokens into numerical vectors using count-based and TF-IDF vectorization.

Discover latent themes from job postings by applying topic modelling algorithms.

Measure text similarities among job descriptions to analyse occupational mobility and skill transferability.

Analyse the sentiment expressed in job postings to infer employer tone and workplace culture.

Develop predictive models to classify job categories based on skill requirements extracted from text.

This tutorial is designed for instructors teaching undergraduate courses in econometrics, labor economics, or empirical research. Students are expected to have a basic understanding of introductory statistics, but no prior Python experience is required. The tutorial includes detailed explanations to help students understand the Python code presented. Those without any Python background can first work through Appendix B: Python Primer for Economics Students. Additionally, Appendix A provides step-by-step guidance on running a Jupyter notebook, which is the interactive environment where all tutorial code examples are presented and executed.

The datasets and Python code used in this article are available online for educational purposes. Instructors and students can download the dataset and Jupyter notebook to go through the tutorial from the link below:

<http://downloads.journalofeconomicsteaching.org/2026/2026-Ho-Textual-Appendix.zip>

3. Processing Text Data

A. Loading Libraries

This tutorial uses a sample of technology job postings from Dice.com, a leading US-based technology job board. The original dataset can be downloaded from Kaggle: <https://www.kaggle.com/PromptCloudHQ/usbased-jobs-from-dicecom>. It provides variables such as job titles, job descriptions, locations, and organizations. The original dataset is large and contains many variables. This tutorial uses an abridged version to demonstrate the essential concepts.

Libraries are collections of toolboxes to perform tasks. Figure 1 demonstrates how to load Python libraries. We present Python code in a shaded background using the **Consolas** font to distinguish it from ordinary text.

Figure 1. Loading Python Libraries

```

In [1]: # load the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import spacy
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

```

Code Explanation:

a. `import X`: brings the entire library X into our workspace.

Example: `import numpy as np` loads the NumPy (math) library and lets us call its functions with the shortcut `np`.

b. `from X import Y`: brings only a specific part (Y) of library X.

Example: `from sklearn.feature_extraction.text import TfidfVectorizer` only loads the TF-IDF tool from `scikit learn`, rather than the entire library. It can reduce memory usage and make the code more readable. We will use TF-IDF for vectorization in the latter section. Table 1 illustrates the type of libraries used in the tutorial.

Table 1. Python Libraries Used in the Tutorial

Theme	Name	Functionality
Math and data	<code>numpy</code>	math and arrays
handling	<code>pandas</code>	data and tables
Graph	<code>matplotlib</code>	plots and visualizations
Text processing	<code>spacy</code> <code>nltk</code>	cleaning text, tokenization, and sentiment analysis
Machine learning (<code>sklearn</code> libraries)	<code>CountVectorizer</code> <code>TfidfVectorizer</code>	turns words into numbers
	<code>NMF</code>	topic modelling
	<code>cosine_similarity</code>	measure similarity between texts
	<code>LogisticRegression</code>	Prediction
	<code>Pipeline</code>	chain steps together

B. Loading the Dataset

Figure 2 shows how to load the dataset for analysis. The data file is named “job_data.csv”.

Figure 2. Loading the Dataset

```
In [2]: df = pd.read_csv('job_data.csv') # load the data and we call the data as "df"
job_code = df['job_code']
job_title = df['job_title']
job_des = df['job_description']
job_type = df['job_type']
location = df['location']
organization = df['organization']
df
```

Out[2]:

	job_code	job_title	job_description	job_type	location	organization
0	2	project manager	This nationally recognized Microsoft Gold Part...	Full Time	Toronto, Canada, ON	Nigel Frank International
1	6	software developer	Minimum Required Skills: C#, ASPNET, SQL, Java...	Full Time, Full-time, Employee	Stamford, CT	CyberCoders
2	3	java developer	Hatstand, a global financial consultancy, is s...	Full Time, C2H W2, FTE - C2H	Stamford, CT	Hatstand US
3	1	software engineer	Live IT Up at CiberAt Ciber, we believe the mo...	Full Time	Tampa, FL	Ciber
4	3	java developer	Senior Java DeveloperLocation: SunnyvaleStart ...	Contract W2	Sunnyvale, CA	TEKsystems
...	...	...	...	...	...	...
6170	20	senior java	Minimum Required Skills:Java, JavaScript, J2EE...	Full Time, Full-time, Employee	Alpharetta, GA	CyberCoders
6171	10	devops engineer	Minimum Required Skills: Amazon Web Services, ...	Full Time, Full-time, Employee	Saint Paul, MN	CyberCoders
6172	9	web developer	Minimum Required Skills:JavaScript, React.JS, ...	Full Time, Full-time, Employee	Seattle, WA	CyberCoders
6173	6	software developer	Minimum Required Skills:Ruby on Rails, HAML, S...	Full Time, Full-time, Employee	Saint Louis, MO	CyberCoders
6174	12	security engineer	Minimum Required Skills:C++/.NET/Java, Softwar...	Full Time, Full-time, Employee	Seattle, WA	CyberCoders

6175 rows x 6 columns

Code Explanation:

We use the `pd` library (pandas) to read the file. The command `df = pd.read_csv('job_data.csv')` opens the CSV file and returns it as a data frame named `df`. Typing `df` displays its contents. If the dataset is large, only the first and last few rows are shown. We can also use `df.head()` to see only the first five rows.

A data frame is like a table, where rows are job postings, and columns are variables. A name is provided for each variable. Example: `job_title = df['job_title']`.

The data frame contains six variables: job code, job title, job description, job type, location, and organization. For each job title, there is a corresponding job code. There are 6,175 job postings.

C. Tokenization: Creating Meaningful Units of Analysis

Tokenization is the process of converting text data into useful semantic units (tokens). An n-gram is simply a sequence of n consecutive words within a document. Suppose the document is “machine learning is required for this job”. There are 7 unigram tokens: [“machine”, “learning”, “is”, “required”, “for”, “this”, “job”]. To identify the meanings of two-word phrases, such as “machine learning”, bigrams are used. Therefore, additional tokens are: [“machine learning”, “learning is”, “is required”, “required for”, “for this”, “this job”].

The code in Figure 3 uses the `spaCy` library to implement a tokenizer that processes text by lowercasing, removing stop words and URLs, and retaining only alphabetic tokens. This cleaning process can reduce noise and preserve the economic meanings of tokens.

Figure 3. Custom Tokenizer Function

```
In [3]: from spacy.lang.en import English
        en = English()

        def simple_tokenizer(doc, model=en):
            tokenized_docs = []
            parsed = model(doc)
            return([t.lower_ for t in parsed if (t.is_alpha)&(not t.like_url)&(not t.is_stop)])
```

Code Explanation:

- a. `from spacy.lang.en import English`: loads spaCy's English parser, and `en = English()` creates a model that can process English sentences.
- b. `simple_tokenizer(doc, model=en)`: is a function that takes a document (a string of text) and returns a list of cleaned tokens.
- c. `parsed = model(doc)`: breaks the text into pieces (like cutting a paragraph into words).
- d. The return statement keeps only words that are:
 - Lowercased (`t.lower_`), so "Job" and "job" are treated the same.
 - Alphabetic (`t.is_alpha`), so numbers and punctuation are removed.
 - Not URLs (`not t.like_url`).
 - Not stop words (`not t.is_stop`), common filler words like "is" or "the" are excluded.

The program spaCy allows many customizations, e.g., using lemmatization (`token.lemma_` instead of `token.lower_`), or including punctuation. Readers can find additional options in the official spaCy tutorial: <https://spacy.io/usage/spacy-101>.

D. Text Vectorization: Quantifying Texts for Econometric Models

Computers cannot directly understand words. Text vectorization is the process of converting tokens into numerical vectors for econometric analysis. We introduce two methods foundational to applied economics research: (1) Count Vector and (2) TF-IDF (Term Frequency-Inverse Document Frequency) Vector.

(1) Count Vector is the count of the frequency with which each token appears in the document. Suppose there are two documents as follows:

Document A: "He is Japanese and is a student."

Document B: "She is a female student."

After removing stop words (like "he", "she", "is", "a", "and") and punctuation, there are 3 unigram tokens: ["japanese", "female", "student"]. The table below shows the count vectors for the two documents:

Table 2. Count Vectors

	japanese	female	student
Document A	1	0	1
Document B	0	1	1

(2) TF-IDF Vector refines the count vector by weighting terms by their importance. $TF(t)$ is the count of a token t that appears in a document. $IDF(t)$ measures how common a token t is across documents, where $IDF(t) = \log(\text{Total Number of Documents} / \text{Number of Documents containing token } t)$. Hence, $IDF(\text{japanese}) = \log(2 / 1) = 0.693$; $IDF(\text{student}) = \log(2 / 2) = 0$; $IDF(\text{female}) = \log(2 / 1) = 0.693$. The following are TF-IDF Vectors:

Table 3. TF-IDF Vectors

	japanese	female	student
Document A	$1 * 0.693 = 0.693$	$0 * 0.693 = 0$	$1 * 0 = 0$
Document B	$0 * 0.693 = 0$	$1 * 0.693 = 0.693$	$1 * 0 = 0$

A common token has a lower IDF. The token “student” is common in both documents; its IDF is 0. It has no power to distinguish one document from another in this corpus. “japanese” and “female” are distinguishing tokens for documents A and B, respectively. Thus, each carries a higher weight.

Figure 4 shows how to use `CountVectorizer`, a text feature extraction tool provided by the `sklearn` library, to count the top ten words in job descriptions for Software Engineer (job code = 1). For example, “software” appears 3,503 times and “design” appears 1,161 times.

Figure 4. Using Count Vectorizer to Find Top Words

```
In [4]: # Word Counts
cv_job_des = CountVectorizer(tokenizer=simple_tokenizer, ngram_range=(1, 1)) # unigram
count_job_des_vecs = cv_job_des.fit_transform(job_des).toarray()
word_count_job_des = dict(zip(cv_job_des.get_feature_names_out(), count_job_des_vecs[job_code==1].sum(axis=0)))
print('Top ten words in job descriptions for Software Engineer (Counts):')
print(sorted(word_count_job_des.items(), key=lambda x: x[1], reverse=True)[:10])

Top ten words in job descriptions for Software Engineer (Counts):
[('software', 3503), ('experience', 3311), ('work', 2964), ('development', 2010), ('required', 1764), ('apply', 1571), ('engineer', 1363), ('position', 1275), ('design', 1161), ('team', 1127)]
```

Code Explanation:

- `CountVectorizer(tokenizer=simple_tokenizer, ngram_range=(1,1))` is used to obtain a count vector. The command `simple_tokenizer`, defined in Section III.C., returns a list of cleaned tokens. The command `ngram_range=(1,1)` specifies that unigrams are used.
- `fit_transform(job_des).toarray()` is used to process all job descriptions and produce a matrix where each row is a job posting and each column is a word count. `job_des` was created in Section III.B and contains all job descriptions as strings.
- `get_feature_names_out()` is used to list all the words found in the job descriptions.

- d. `count_job_des_vecs[job_code==1].sum(axis=0)` is used to select only the job postings with job code = 1 (Software Engineer) and sum their word counts.
- e. `dict(zip(...))` is used to create a dictionary linking each word to its total count. See Appendix B.4 for further explanation.
- f. `sorted(word_count_job_des.items(), key=lambda x: x[1], reverse=True)[:10]` is used to find the ten most frequent words in job descriptions.
- `word_count_job_des` is a Python dictionary created earlier in the code. Each key is a word (e.g., “software”) and its corresponding value is the total count of that word across all “Software Engineer” job descriptions. For example: {“software”: 3503, “design”: 1161, ...}
- The command `.items()` converts the dictionary into a list of key-value pairs represented as tuples.

Example: [(“software”, 3503), (‘design’, 1161), ...]

- The `key` argument tells Python what value to use for sorting.
- The function `lambda x: x[1]` is a small anonymous function. For each tuple `x` (like (‘software’, 3503)), `x[1]` selects the second element – the count. By setting `key=lambda x: x[1]`, we instruct Python to sort based on the counts, not the words.

The command `reverse=True` orders the list from largest to smallest (descending).

Figure 5 shows the results using TF-IDF vectors. The value after the token represents the TF-IDF score. TF-IDF downweights common words (e.g., “position”) and upweights rare, job-specific words (e.g., “SQL”).

Figure 5. Using TFIDF Vectorizer to Find Top Words

```
In [5]: # TF-IDF
tfidf_job_des = TfidfVectorizer(tokenizer=simple_tokenizer, ngram_range=(1,1)) # unigram
tfidf_job_des_vecs = tfidf_job_des.fit_transform(job_des).toarray()
tfidf_count_job_des = dict(zip(tfidf_job_des.get_feature_names_out(), tfidf_job_des_vecs[job_code==1].sum(axis=0)))
print('Top ten words in job descriptions for Software Engineer (TF-IDF):')
print(sorted(tfidf_count_job_des.items(), key=lambda x: x[1], reverse=True)[:10])

Top ten words in job descriptions for Software Engineer (TF-IDF):
[('software', 83.10697327029209), ('work', 59.76103318355971), ('experience', 54.81667535381762), ('apply', 45.61073034118103), ('development', 45.425372719406205), ('engineer', 44.3192695350315), ('required', 42.29488948632545), ('click', 32.621414582118625), ('receiving', 31.8610903251797), ('eligibility', 31.745017093919998)]
```

4. Applications of Text Data in Labor Economics

After cleaning text data through tokenization and vectorization, the following shows four applications of job postings to understand the labor market.

A. Topic Modelling to Discover Skill Clusters

Topic modelling is a text mining algorithm that discovers topics in a set of documents, where a topic is a cluster of words that frequently co-occur across documents. Non-Negative Matrix Factorization (NMF) is an algorithm that decomposes a document-by-word matrix into a document-by-topic matrix and a topic-by-word matrix, where the researcher determines the number of topics.

Applying NMF to job descriptions reveals “skill clusters” (Figure 6). For example, Topic 1 is associated with words such as “project,” “management,” and “plans,” indicating a role in project management. Topic 2 is defined by words such as “Java,” “web,” and “JavaScript,” indicating a Java Developer role. This enables students to move beyond predefined occupational codes and to empirically derive job types directly from the data.

Figure 6. Topic Modelling with NMF

```
In [6]: def display(model, feature_names, no_top_words):
        for topic_idx, topic in enumerate(model.components_):
            print("Topic %d:"%(topic_idx))
            print(" ".join([feature_names[i] for i in topic.argsort()[::-no_top_words - 1:-1]]))
n_components = 6      # number of topics
no_top_words = 10
nmf_job_des = NMF(n_components=n_components)
nmf_job_des_vecs = nmf_job_des.fit_transform(tfidf_job_des_vecs)
nmf_feature_names = tfidf_job_des.get_feature_names_out()
display(nmf_job_des, nmf_feature_names, no_top_words)

Topic 0:
work apply required receiving forward eligibility click inc protected employment
Topic 1:
project management manager projects managing manage experience plans scope program
Topic 2:
java experience development web software design developer applications application javascript
Topic 3:
teksystems consultants contract sincere stems targeted interests placement allowing seek
Topic 4:
network security cisco systems support experience engineer windows infrastructure hardware
Topic 5:
business data requirements analyst analysis systems technical system experience support
```

Code Explanation:

- `display(...)`: A function that displays the top words for each topic. It loops through the topics the model finds and shows the most important words for each.
- `n_components = 6`: We arbitrarily choose six topics for illustration. For advanced analysis, one needs statistical criteria to choose the optimal number of topics.
- `NMF(n_components=n_components)`: Initializes the NMF model to decompose the TF-IDF matrix into topics.
- `fit_transform(tfidf_job_des_vecs)`: Fits the model to the job descriptions and produces a document topic matrix using the TFID vectors.
- `get_feature_names_out()`: Retrieves the words in the job descriptions.

Learning Outcome: Students learn to apply topic modelling to uncover latent patterns in economic data.

B. Text Similarity to Analyse Occupational Mobility

Text similarity measures the degree of similarity between documents. TF-IDF is used to transform words into vectors for each document. Cosine similarity is then used to measure the similarity between two vectors. The TF-IDF values range from 0 to 1, where 1 indicates perfect similarity and 0 indicates no similarity.

The similarity between two job descriptions can shed light on occupational mobility and the transferability of human capital. As shown in Figure 7, a low similarity score (i.e., 0.09) between a Software Developer (job posting 1) and a Java Developer (job posting 2) suggests job-to-job switching between the two roles is low. In contrast to the job descriptions, the

geographical similarity between the two postings is perfect (score of 1.0), as both are explicitly located in Stamford, CT.

Figure 7. Applications of Cosine Similarity

```
In [7]: tfidf = TfidfVectorizer(tokenizer=simple_tokenizer, ngram_range=(1,1))
        tfidf_job_des_vecs = tfidf.fit_transform(job_des).toarray()
        tfidf_location_vecs = tfidf.fit_transform(location).toarray()

        tfidf_job_des_sims = cosine_similarity(tfidf_job_des_vecs) # cosine similarity for job descriptions
        tfidf_location_sims = cosine_similarity(tfidf_location_vecs) # cosine similarity for locations

In [8]: tfidf_job_des_sims[1][2] # software developer and java developer are NOT similar in their job descriptions
Out[8]: 0.0905348122607956

In [9]: tfidf_location_sims[1][2] # software developer and java developer are the same in their locations
Out[9]: 1.0
```

Code Explanation:

- `TfidfVectorizer(...)`: Converts job descriptions and locations into TF IDF vectors.
- `fit_transform(job_des)`: Creates a matrix of word importance for each job description.
- `fit_transform(location)`: Creates a matrix of word importance for each location.
- `cosine_similarity(...)`: Compares all pairs of vectors to measure cosine similarity.
- `tfidf_job_des_sims[1][2]`: The similarity score between the job description of posting 1 (Software Developer) and posting 2 (Java Developer).
- `tfidf_location_sims[1][2]`: Compares the location fields of the same postings.

Learning Outcome: Students learn to quantify document similarity and interpret its economic meaning.

C. Sentiment Analysis to Infer Emotions

Sentiment Analysis is used to infer the emotional tone behind documents, attempting to discern the underlying positive, neutral, or negative sentiments conveyed in the text. Figure 8 illustrates the use of the NLTK library to calculate a single score representing the overall sentiment of job descriptions. The value is between -1 (most negative) and +1 (most positive).

Aggregating these scores by state reveals geographic variation in employer tone, which might be correlated with workplace culture. Interestingly, the average score is highest in Alaska and lowest in Wyoming. While these descriptive results are suggestive, they highlight how sentiment analysis can generate hypotheses about regional labor market differences worthy of further investigation.

Figure 8. Average Sentiment Score of Job Descriptions by U.S. State.

```

In [10]: sia = SentimentIntensityAnalyzer()
df['sentiment'] = df['job_description'].apply(lambda x: sia.polarity_scores(x)['compound']) # sentiments for job descriptions

In [11]: df['state'] = df['location'].str.extract(r'\s*([A-Z]{2})$') # Define states from location
avg_sentiment = df.groupby('state')['sentiment'].mean().sort_values(ascending=False)
top_3 = avg_sentiment.head(3)
bottom_3 = avg_sentiment.tail(3)
print("Top 3 States by Average Sentiment in Job Descriptions:")
print(top_3)
print("Bottom 3 States by Average Sentiment in Job Descriptions:")
print(bottom_3)

Top 3 States by Average Sentiment in Job Descriptions:
state
AK    0.999350
MT    0.998000
BC    0.996475
Name: sentiment, dtype: float64
Bottom 3 States by Average Sentiment in Job Descriptions:
state
IA    0.860165
AR    0.839559
WY    0.496900
Name: sentiment, dtype: float64

```

Code Explanation:

- `SentimentIntensityAnalyzer()`: Loads the VADER sentiment tool.
- `df['sentiment'] = df['job_description'].apply(lambda x: sia.polarity_scores(x)['compound'])`: Applies the analyzer to each job description and stores the overall sentiment score in a new column called 'sentiment'. See Appendix B.10 for further explanation.
- `df['state'] = df['location'].str.extract(r'\s*([A-Z]{2})$')`: Extracts the two-letter state code from the location field.
- `df.groupby('state')['sentiment'].mean().sort_values(ascending=False)`: Calculates the average sentiment score for each state.
- `head(3)` and `tail(3)`: Identifies the top three and bottom three states by average sentiment.

Learning Outcome: Students learn how to utilize sentiment analysis to infer the emotional tone of text data.

D. Text Classification: Predicting Jobs from Skills

In labor economics, a fundamental question is how to match workers with employers. Employers want to know which candidates fit which roles, and workers want to identify positions that match their skill sets. This section shows how to build a predictive model that utilizes job descriptions to predict the job category (e.g., Software Engineer, Data Scientist, or Java Developer). This is an example of supervised machine learning – we “teach” the computer using examples where the answer is already known, and then we test whether it learned correctly.

Step 1: What is Supervised Learning?

Studying for an exam is a good example of supervised learning. You are given many practice questions with the answers provided to learn the knowledge. Then you take an exam with new questions and predict the answers. If you learned well, you get a high score.

In our case:

Practice questions = job descriptions (the text).

Answers = job codes (the correct job category).

Exam = new job descriptions the model has never seen.

Score = prediction accuracy (percentage of correct guesses).

Step 2: Splitting the Data – Training versus Testing

We cannot test the model on the same data we used to train it, because it is like giving students the same questions on the exam. Instead, we randomly split our dataset (6,175 job postings) into two parts. First, the training set is used to train the model. Second, the test set is used only to evaluate how well the model generalizes to new data. In practice, 80% of the data is randomly split into a training set for model training and a test set for model evaluation, with 20% allocated to the test set. The procedure is illustrated in Figure 9.

Figure 9. Splitting Training and Testing Sets

```
In [12]: np.random.seed(seed=42)      # set the seed for numpy
         pct_train = 0.8              # 80% training data
         train_bool = np.random.random(len(df)) <= pct_train
         df_train = df[train_bool]
         df_test = df[~train_bool]
         x_train = df_train['job_description']
         x_test = df_test['job_description']
         y_train = df_train['job_code']
         y_test = df_test['job_code']
```

Code Explanation:

- a. `np.random.seed(42)`: Sets the random seed to ensure reproducible results. We will get the same train-test split each time they run the code.
- b. `pct_train = 0.8`: Specifies that 80% of the data is used for training.
- c. `train_bool = np.random.random(len(df)) <= pct_train`: Generates random numbers between 0 and 1 for each row in the dataset, where `len(df)` is the number of rows. `train_bool` is "True" if the random number is equal to or less than 0.8.
- d. `df_train` and `df_test`: Split the data frame into training and testing subsets.
- e. `x_train` and `x_test`: Features (job descriptions) for training and testing.
- f. `y_train` and `y_test`: Labels (job codes) for training and testing.

Step 3: From Text to Numbers – TF-IDF Vectors

The model cannot read words directly. We use the same TF-IDF vectorization introduced in Section III.D. to convert each job description into a numerical vector. The model then learns which word patterns are associated with which job code.

Step 4: Logistic Regression – A Simple Classifier

Logistic regression is a statistical classification method for predicting categories. It calculates the probability that a given job description belongs to each possible job code, then picks the code with the highest probability. We combine TF-IDF and logistic regression into a single pipeline as shown in Figure 10. A pipeline chains steps together by first vectorizing the text, then feeding the resulting vectors into the classifier.

Step 5: Measuring Accuracy

After training the logistic model, we use it to predict job codes from the test set (job descriptions it has never seen). We then compare its predictions to the true job codes. The result shows that the test accuracy is 83%. This implies that, for 83 out of 100 new job postings, the model correctly predicts the job category solely from the job descriptions.

Figure 10. Job Predictions

```
In [13]: tfidf = TfidfVectorizer(tokenizer=simple_tokenizer, ngram_range=(1,2))
lor = LogisticRegression(C = 50)
p_logit = Pipeline([('tfidf', tfidf), ('logistic', lor)])
p_logit.fit(x_train, y_train)
print('Train accuracy:', p_logit.score(x_train, y_train))
print('Test accuracy:', p_logit.score(x_test, y_test))

Train accuracy: 0.9983812221772562
Test accuracy: 0.8256285482562855
```

Code Explanation:

- `TfidfVectorizer(tokenizer=simple_tokenizer, ngram_range=(1,2))`: Converts job descriptions into numerical features using TF IDF.
- `ngram_range=(1,2)`: Both single words (unigrams) and pairs of words (bigrams) are included.
- `LogisticRegression(C=50)`: Creates a logistic regression classifier. The parameter `C` controls regularization; a higher value (50) allows the model to fit the training data more closely.
- `Pipeline([('tfidf', tfidf), ('logistic', lor)])`: Chains the TFIDF vectorizer and Logistic classifier together, so the text is automatically transformed before classification.
- `fit(x_train, y_train)`: Trains the model on the training set.
- `score(x_train, y_train)` and `score(x_test, y_test)`: Evaluate accuracy on the training and test sets.

Learning Outcome: Students learn how to develop and evaluate predictive models to understand the economic relationship between skills and jobs.

5. Teaching Implementation

The tutorial can be integrated into an undergraduate course over three laboratory sessions (approximately 3–4 weeks). Each session builds progressively from foundational skills to applied analysis, ensuring that students without prior programming experience can follow along. Instructors may adjust the pacing based on the course length and students' backgrounds.

Table 4. Teaching Implementation

Component	Activities	Estimated Time
Lab Session 1	Introduction to text data, tokenization, and vectorization. Students run starter code in Jupyter notebooks, experiment with parameters (e.g., n-gram size), and reflect on how text differs from structured data.	3 hours
Lab Session 2	Apply topic modelling and similarity analysis to job postings. Students interpret clusters of skills and discuss occupational mobility in economic terms.	3 hours
Lab Session 3	Conduct sentiment analysis and build predictive models. Students evaluate accuracy by including additional variables (e.g., location).	3 hours
Capstone Project (Optional)	Students design their own research question using alternative text data (e.g., movie reviews). They collect, clean, and analyse text, then present findings.	Flexible

6. Conclusion

This article presents a pedagogical framework for integrating textual analysis into the undergraduate economics curriculum. This bridges the gap between modern text-as-data research and conventional undergraduate economics. Textual analysis and Python programming can be taught concurrently to economics majors without prior coding experience. The tutorial uses job posting data to illustrate how to process text data and apply it to understand the labor market. It is believed that the replicable teaching resources provided will benefit both instructors and students, thereby advancing undergraduate economic education.

References

- Athey, S., & Luca, M. (2019). Economists (and economics) in tech companies. *Journal of Economic Perspectives*, 33(1), 209–230. DOI: [10.1257/jep.33.1.209](https://doi.org/10.1257/jep.33.1.209)
- Ash, E., & Hansen, S. (2023). Text algorithms in economics. *Annual Review of Economics*, 15, 659–688. DOI: [10.1146/annurev-economics-082222-074352](https://doi.org/10.1146/annurev-economics-082222-074352)
- Baker, S. R., Bloom, N., & Davis, S. J. (2016). Measuring economic policy uncertainty. *The Quarterly Journal of Economics*, 131(4), 1593–1636. DOI: [10.1093/qje/qjw024](https://doi.org/10.1093/qje/qjw024)
- Batt, S., Grealis, T., Harmon, O., & Tomolonis, P. (2020). Learning tableau: A data visualization tool. *The Journal of Economic Education*, 51(3-4), 317–328. DOI: [10.1080/00220485.2020.1804503](https://doi.org/10.1080/00220485.2020.1804503)
- Deming, D., & Kahn, L. B. (2018). Skill requirements across firms and labor markets: Evidence from job postings for professionals. *Journal of Labor Economics*, 36(S1), S337–S369. DOI: [10.3386/w23328](https://doi.org/10.3386/w23328)
- Gentzkow, M., Kelly, B., & Taddy, M. (2019). Text as data. *Journal of Economic Literature*, 57(3), 535–574. DOI: [10.1257/jel.20181020](https://doi.org/10.1257/jel.20181020)
- Hansen, S., McMahon, M., & Prat, A. (2018). Transparency and deliberation within the FOMC: A computational linguistics approach. *The Quarterly Journal of Economics*, 133(2), 801–870. DOI: [10.1093/qje/qjx045](https://doi.org/10.1093/qje/qjx045)
- Hassan, T. A., Hollander, S., Kalyani, A., van Lent, L., Schwedeler, M., & Tahoun, A. (2025). Text as data in economic analysis. *Journal of Economic Perspectives*, 39(3), 193–220. DOI: [10.1257/jep.20231365](https://doi.org/10.1257/jep.20231365)
- Jenkins, B. C. (2022). A Python-based undergraduate course in computational macroeconomics. *The Journal of Economic Education*, 53(2), 126–140. DOI: [10.1080/00220485.2022.2038322](https://doi.org/10.1080/00220485.2022.2038322)
- Kuroki, M. (2021). Using Python and Google Collab to teach undergraduate microeconomic theory. *International Review of Economics Education*, 38, Article 100225. DOI: [10.1016/j.iree.2021.100225](https://doi.org/10.1016/j.iree.2021.100225)
- Kuroki, M. (2023). Integrating data science into an econometrics course with a Kaggle competition. *The Journal of Economic Education*, 54(4), 364–378. DOI: [10.1080/00220485.2023.2220695](https://doi.org/10.1080/00220485.2023.2220695)
- Li, K., Mai, F., Shen, R., & Yan, X. (2021). Measuring corporate culture using machine learning. *The Review of Financial Studies*, 34(7), 3265–3315. DOI: [10.1093/rfs/hhaa079](https://doi.org/10.1093/rfs/hhaa079)
- Loughran, T., & McDonald, B. (2020). Textual analysis in finance. *Annual Review of Financial Economics*, 12(1), 357–375. DOI: [10.1146/annurev-financial-012820-032249](https://doi.org/10.1146/annurev-financial-012820-032249)

Appendix A. Running a Jupyter Notebook: A Student Guide

Jupyter Notebooks can combine code, results, and explanations into a single interactive document. This guide walks students through setting up and using Jupyter notebooks for the tutorial.

A.1 Install Jupyter

- Download Anaconda (recommended) from <https://www.anaconda.com/download>.

Follow the installation instructions for your operating system.

A.2 Organize Project Files

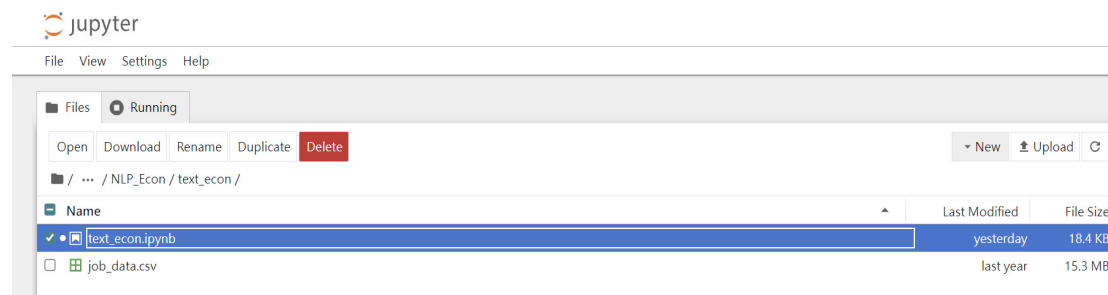
Before running any code, it is important to keep your files organized.

- Create a folder for the tutorial.
- Place both the notebook file (text_econ.ipynb) and the dataset (job_data.csv) inside this folder.

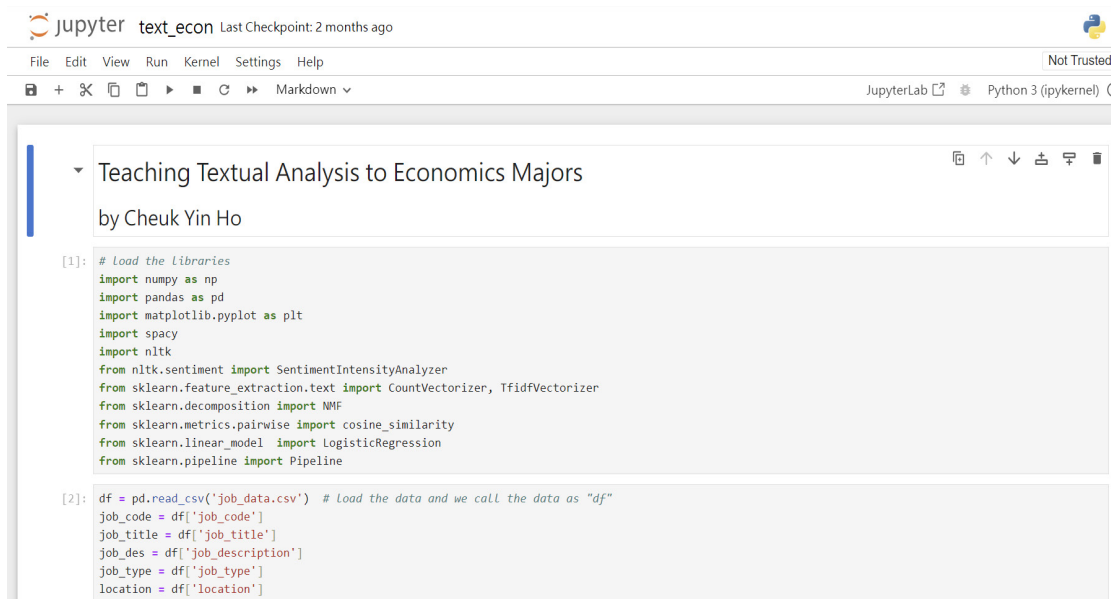
Keeping files together ensures that commands like `pd.read_csv('job_data.csv')` run smoothly.

A.3 Launch Jupyter Notebook

- Open Anaconda Navigator.
- Click “launch” under Jupyter Notebook.
- A browser window will open. Navigate to your project folder.



- Click on text_econ.ipynb to open the notebook.



The screenshot shows a JupyterLab interface with a notebook titled 'text_econ'. The top bar indicates 'Last Checkpoint: 2 months ago'. The notebook has two code cells. The first cell, labeled [1], contains the following code:

```
# Load the Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import spacy
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
```

The second cell, labeled [2], contains the following code:

```
df = pd.read_csv('job_data.csv') # Load the data and we call the data as "df"
job_code = df['job_code']
job_title = df['job_title']
job_des = df['job_description']
job_type = df['job_type']
location = df['location']
```

A.4 Run Code Cells

- Notebooks are divided into cells. Each cell can contain code or text.
- To run a cell, e.g., `print('Hello!')`.
- Press **Shift + Enter** to run the cell.

The output will appear directly below the cell.

A.5 Save and Export Work

- Save your work using **Ctrl + S** (Windows) or **Cmd + S** (Mac).
- You can export the notebook as HTML or PDF via **File → Download as**.

Appendix B. Python Primer for Economics Students

This primer covers the essential Python concepts needed to understand the tutorial. Students with no prior programming experience should work through this section before attempting the main code. For a more comprehensive introduction to Python, consult the official Python tutorial: docs.python.org/3/tutorial/.

B.1 Variables and Data Types

A variable stores a value. You assign a value using `=`.

python

```
# This is a comment - Python ignores it.
```

```
x = 5 # integer
```

```
y = 3.14 # float (decimal)
```

```
name = "economics" # string (text)
```

```
is_valid = True # boolean (True/False)
```

To see the value of a variable, use `print()`:

python

```
print(x)
```

```
print(name)
```

Output:

text

```
5
```

```
economics
```

Every value has a type. The most common ones are:

Type	What it stores	Example
Int	Integer	5, -3, 0
Float	decimal number	3.14, 0.85
Str	string (text)	"machine learning"
Bool	true or false	True, False

B.2 Basic Operations

python

```
# Arithmetic
```

```
sum_val = 10 + 3 # 13
```

```
difference = 10 - 3 # 7
```

```
product = 10 * 3 # 30
```

```
quotient = 10 / 3 # 3.333...
```

```
# Strings can be "added" (concatenated)
```

```
full = "Data " + "Scientist" # "Data Scientist"
```

B.3 Working with Text: Strings

Strings are everywhere in text analysis. You will constantly clean, split, and transform them.

Creating Strings

Use single ' or double " quotes – they work the same.

python

```
s1 = 'machine learning'
```

```
s2 = "job posting"
```

String Methods

Methods are functions attached to a string using a dot (.). They return a new string:

python

```
text = "Machine LEARNING"
```

```
# Lowercase
```

```
lower_text = text.lower()      # "machine learning"
```

```
# Uppercase
```

```
upper_text = text.upper()     # "MACHINE LEARNING"
```

The tokenizer uses `.lower()` to treat "Job" and "job" the same, and `.isalpha()` to keep letters.

python

```
"python".isalpha()           # True
```

```
"python3".isalpha()         # False (digits present)
```

B.4 Collections: Lists and Dictionaries

Lists – Ordered Sequence

A **list** holds multiple items in order. You create it with square brackets `[]`.

python

```
tokens = ["machine", "learning", "python"]
```

Common operations:

python

```
# Access by index (starts at 0)

first = tokens[0] # "machine"

second = tokens[1] # "learning"

last = tokens[-1] # "python" (negative counts from end)

# Add to the end

tokens.append("nlp") # ["machine", "learning", "python", "nlp"]

# Length

len(tokens) # 4
```

Dictionaries – Key-Value Pairs

A **dictionary** maps keys to values. Use curly braces {}.

python

```
word_counts = {"python": 3, "java": 1, "sql": 2}
```

- Keys are usually strings (words). Values can be numbers, strings, lists, etc.
- Access by key:

python

```
print(word_counts["python"]) # 3

word_counts["r"] = 0 # adds a new key 'r' with value 0
```

Creating a Dictionary from Two Lists with zip() and dict()

In the tutorial (Figure 4), you see:

python

```
dict(zip(words, counts))
```

- `zip(words, counts)` pairs each word with its count, e.g., `[("python", 3), ("java", 1), ...]`
- `dict()` turns that list of pairs into a dictionary.

Example:

python

```
words = ["python", "java", "sql"]

counts = [3, 1, 2]

combined = dict(zip(words, counts))

# combined = {"python": 3, "java": 1, "sql": 2}
```

B.5. if Statements

The command `if` lets your code do different things depending on a condition.

python

```
# Example: Determine if a job posting requires a specific skill

skill = "Python"

if skill == "Python":

    print("This job requires Python.")

elif skill == "Java":

    print("This job requires Java.")

else:

    print("This job requires other skills.")
```

Output:

text

```
This job requires Python.
```

Comparison operators:

- `==` equals
- `!=` not equal
- `>` greater than, `<` less than, `>=`, `<=`
- `and`, `or`, `not` for combining conditions

In the tutorial's tokenizer, you see `if t.is_alpha and not t.is_stop` – that is a condition using `and` and `not`.

B.6 Repetition: for Loops and List Comprehensions

Loop over each item in a list.

python

```
tokens = ["machine", "learning", "python"]
```

```
for token in tokens:
```

```
    print(token.upper())
```

Output:

text

```
MACHINE
```

```
LEARNING
```

```
PYTHON
```

You can also loop over dictionary items:

python

```
word_counts = {"python": 3, "java": 1, "sql": 2}
```

```
for word, count in word_counts.items():
```

```
    print(f"{word} appears {count} times")
```

- The command `word_counts.items()` returns each key value pair as a tuple of two elements: `("python", 3)`, then `("java", 1)`, then `("sql", 2)`.
- The `for` loop unpacks each tuple into two variables: the first element goes into `word`, the second into `count`.

List Comprehension – A Shorter Way to Create Lists

python

```
upper_tokens = []
```

```
for t in tokens:
```

```
    upper_tokens.append(t.upper())
```

Alternatively:

python

```
upper_tokens = [t.upper() for t in tokens]
```

The tutorial uses this in the tokenizer:

python

```
[t for t in tokens if t.isalpha()] # keep only alphabetic tokens
```

B.7 Reusable Code: Functions

A **function** is a named block of code you can reuse. Define it with `def`, then call it.

Defining and Calling a Simple Function

python

```
def greet(name):
```

```
    return "Hello, " + name
```

```
# Call it
```

```
message = greet("Student")
```

```
print(message) # "Hello, Student"
```

- The command `def` starts the definition.
- The command `name` is a **parameter** (a variable that gets the value passed in).
- The command `return` sends the result back. Without `return`, the function returns `None`.

B.8 Using Libraries: `import`

Python has a large collection of libraries (toolkits) for additional functionality. You load them with `import`.

Two Common Ways to Import

python

```
# 1. Import the whole library and give it a short name (alias)
```

```
import pandas as pd
```

```
import numpy as np
```

```
# Now use pandas functions with pd.
```

```
df = pd.read_csv("data.csv")
```

```
arr = np.array([1,2,3])
```

```
# 2. Import only a specific function/class
```

```
from sklearn.feature_extraction.text import CountVectorizer
```

B.9 The **pandas** Library: Data Frames (Tables)

Pandas is the most important library for data work in Python. A **Data Frame** is a table with rows and columns, like an Excel sheet.

Loading a CSV File

python

```
import pandas as pd
```

```
df = pd.read_csv("job_data.csv")
```

- **df** is a Data Frame.
- **.read_csv()** reads a comma separated values file.

Exploring a Data Frame

python

```
df.head() # first 5 rows
```

```
df.shape # (rows, columns) e.g., (6175, 6)
```

```
df.columns # list of column names
```

Selecting Columns

Use square brackets with column name (string):

python

```
job_title = df["job_title"] # returns a Series (like one column)
```

```
job_des = df["job_description"]
```

Select multiple columns with a list:

python

```
subset = df[["job_title", "location"]]
```

Filtering Rows (Conditional Selection)

python

```
# Keep only rows where job code equals 1
```

```
software_jobs = df[df["job_code"] == 1]
```

The command `df["job_code"] == 1` creates a Boolean series (True/False). Then `df[...]` uses that to keep only true rows.

python

```
# This adds a new column
```

```
df["new_column"] = some values # can be a constant or a computed result
```

B.10 Useful Tools

Grouping and Aggregating

python

```
# Average sentiment by state
```

```
state_avg = df.groupby("state")["sentiment"].mean()
```

- The command `groupby("state")` splits data by unique state values.
- The command `["sentiment"]` selects that column.
- The command `.mean()` computes the average for each group.

Applying a Function to Each Element: `.apply()`

The command `.apply()` runs a function on every value in a column (or row).

python

```
def get_length(text):
```

```
    return len(text)
```

```
df["title_len"] = df["job_title"].apply(get_length)
```

The **lambda** Function

A **lambda** is a small, anonymous function you can write in one line. It works like a function but has no **def** and no **return** statement – the expression is automatically returned.

python

```
df["title_len"] = df["job_title"].apply(lambda x: len(x))
```

The tutorial uses lambda to compute sentiment scores:

python

```
df["sentiment"] = df["job_description"].apply(lambda x: sia.polarity_scores(x)["compound"])
```

x is each job description (a string). The expression calls a function that returns a dictionary, then extracts the **"compound"** score.