# Feature Engineering for Machine Learning in Python

**Creating Features**

\*One-hot encoding and dummy variables\*

---Getting to know your data---

---

# Import pandas

import pandas as pd

# Import so\_survey\_csv into so\_survey\_df

so\_survey\_df = pd.read\_csv(so\_survey\_csv)

---

# Import pandas

import pandas as pd

# Import so\_survey\_csv into so\_survey\_df

so\_survey\_df = pd.read\_csv(so\_survey\_csv)

# Print the first five rows of the DataFrame

print(so\_survey\_df.head())

---

# Import pandas

import pandas as pd

# Import so\_survey\_csv into so\_survey\_df

so\_survey\_df = pd.read\_csv(so\_survey\_csv)

# Print the first five rows of the DataFrame

print(so\_survey\_df.head())

# Print the data type of each column

print(so\_survey\_df.dtypes)

---

What type of data is the ConvertedSalary column?

Ans : Numeric

--- Selecting specific data types ---

---

# Create subset of only the numeric columns

so\_numeric\_df = so\_survey\_df.select\_dtypes(include=['int','float'])

# Print the column names contained in so\_survey\_df\_num

print(so\_numeric\_df.columns)

---

\*Dealing with categorical features\*

---One-hot encoding and dummy variables---

# Convert the Country column to a one hot encoded Data Frame

one\_hot\_encoded = pd.get\_dummies(so\_survey\_df, columns=['Country'], prefix='OH')

# Print the columns names

print(one\_hot\_encoded.columns)

---

---Dealing with uncommon categories---

# Create a series out of the Country column

countries = so\_survey\_df['Country']

# Get the counts of each category

country\_counts = countries.value\_counts()

# Print the count values for each category

print(country\_counts)

---

# Create a series out of the Country column

countries = so\_survey\_df['Country']

# Get the counts of each category

country\_counts = countries.value\_counts()

# Create a mask for only categories that occur less than 10 times

mask = countries.isin(country\_counts[country\_counts < 10].index)

# Print the top 5 rows in the mask series

print(mask.head())

---

# Create a series out of the Country column

countries = so\_survey\_df['Country']

# Get the counts of each category

country\_counts = countries.value\_counts()

# Create a mask for only categories that occur less than 10 times

mask = countries.isin(country\_counts[country\_counts < 10].index)

# Label all other categories as Other

countries[mask] = 'Other'

# Print the updated category counts

print(pd.value\_counts(countries))

---

\*Numeric variables\*

---Binarizing columns---

# Create the Paid\_Job column filled with zeros

so\_survey\_df['Paid\_Job'] = 0

# Replace all the Paid\_Job values where ConvertedSalary is > 0

so\_survey\_df.loc[so\_survey\_df['ConvertedSalary'] > 0, 'Paid\_Job'] = 1

# Print the first five rows of the columns

print(so\_survey\_df[['Paid\_Job', 'ConvertedSalary']].head())

---

---Binning values---

# Bin the continuous variable ConvertedSalary into 5 bins

so\_survey\_df['equal\_binned'] = pd.cut(so\_survey\_df['ConvertedSalary'], 5)

# Print the first 5 rows of the equal\_binned column

print(so\_survey\_df[['equal\_binned', 'ConvertedSalary']].head())

----

**Dealing with Messy Data**

\* Why do missing values exist? \*

---How sparse is my data?

# Subset the DataFrame

sub\_df = so\_survey\_df[['Age', 'Gender']]

# Print the number of non-missing values

print(sub\_df.info())

---

Question

Based on the results, how many non-missing entries are there in the Gender column?

Answer: 693

---

---Finding the missing values---

# Print the top 10 entries of the DataFrame

print(sub\_df.head(10))

---

---Dealing with missing values (I)----

---Listwise deletion---

---

# Print the number of rows and columns

print(so\_survey\_df.shape)

---

--- Replacing missing values with constants ---

# Print the count of occurrences

print(so\_survey\_df['Gender'].value\_counts())

---

---Replacing missing values with constants---

# Replace missing values

so\_survey\_df['Gender'].fillna(value='Not Given', inplace=True)

# Print the count of each value

print(so\_survey\_df['Gender'].value\_counts())

---

---Dealing with missing values (II)---

---Filling continuous missing values---

---

# Print the first five rows of StackOverflowJobsRecommend column

print(so\_survey\_df.StackOverflowJobsRecommend.head(5))

---

# Fill missing values with the mean

so\_survey\_df['StackOverflowJobsRecommend'].fillna(so\_survey\_df['StackOverflowJobsRecommend'].mean(), inplace=True)

# Print the first five rows of StackOverflowJobsRecommend column

print(so\_survey\_df['StackOverflowJobsRecommend'].head())

---

so\_survey\_df['StackOverflowJobsRecommend'].fillna(so\_survey\_df['StackOverflowJobsRecommend'].mean(), inplace=True)

# Round the StackOverflowJobsRecommend values

so\_survey\_df['StackOverflowJobsRecommend'] = round(so\_survey\_df['StackOverflowJobsRecommend'])

# Print the top 5 rows

print(so\_survey\_df['StackOverflowJobsRecommend'].head())

---

---Imputing values in predictive models---

When working with predictive models you will often have a separate train and test DataFrames. In these cases you want to ensure no information from your test set leaks into your train set. When filling missing values in data to be used in these situations how should approach the two datasets?

Answer: Apply the measures of central tendency (mean/median etc.) calculated on the train set to both the train and test sets.

---

---Apply the measures of central tendency (mean/median etc.) calculated on the train set to both the train and test sets.---

---Dealing with stray characters (I)---

---

# Remove the commas in the column

so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].str.replace(',', '')

---

---Dealing with stray characters (I)---

---

# Remove the dollar signs in the column

so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].str.replace('$', '')

---

---Dealing with stray characters (II)---

# Attempt to convert the column to numeric values

numeric\_vals = pd.to\_numeric(so\_survey\_df['RawSalary'], errors='coerce')

# Find the indexes of missing values

idx = numeric\_vals.isna()

# Print the relevant rows

print(so\_survey\_df['RawSalary'][idx])

---

# Replace the offending characters

so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].str.replace('£', '')

# Convert the column to float

so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].astype('float')

# Print the column

print(so\_survey\_df['RawSalary'])

----

---Method chaining---

# Use method chaining

so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary']\

.str.replace(',', '')\

.str.replace('$', '')\

.str.replace('£', '')\

.astype('float')

# Print the RawSalary column

print(so\_survey\_df['RawSalary'])

---

**Conforming to Statistical Assumptions**

---Data distributions---

---What does your data look like? (I)---

# Create a histogram

so\_numeric\_df.hist()

plt.show()

---

# Create a boxplot of two columns

so\_numeric\_df[['Age', 'Years Experience']].boxplot()

# plt.show()Cleaning up your text

---

# Create a boxplot of ConvertedSalary

so\_numeric\_df[['ConvertedSalary']].boxplot()

plt.show()

---

---What does your data look like? (II)---

# Import packages

import matplotlib.pyplot as plt

import seaborn as sns

# Plot pairwise relationships

sns.pairplot(so\_numeric\_df)

# Show plot

plt.show()

----

# Print summary statistics

print(so\_numeric\_df.describe())

---

---When don't you have to transform your data?---

While making sure that all of your data is on the same scale is advisable for most analyses, for which of the following machine learning models is normalizing data not always necessary?

Answer: Decision Trees

---

---Scaling and transformations---

---Normalization---

# Import MinMaxScaler

from sklearn.preprocessing import MinMaxScaler

# Instantiate MinMaxScaler

MM\_scaler = MinMaxScaler()

# Fit MM\_scaler to the data

MM\_scaler.fit(so\_numeric\_df[['Age']])

# Transform the data using the fitted scaler

so\_numeric\_df['Age\_MM'] = MM\_scaler.transform(so\_numeric\_df[['Age']])

# Compare the origional and transformed column

print(so\_numeric\_df[['Age\_MM', 'Age']].head())

----

---Standardization---

# Import StandardScaler

from sklearn.preprocessing import StandardScaler

# Instantiate StandardScaler

SS\_scaler = StandardScaler()

# Fit SS\_scaler to the data

SS\_scaler.fit(so\_numeric\_df[['Age']])

# Transform the data using the fitted scaler

so\_numeric\_df['Age\_SS'] = SS\_scaler.transform(so\_numeric\_df[['Age']])

# Compare the origional and transformed column

print(so\_numeric\_df[['Age\_SS', 'Age']].head())

----

---Log transformation---

# Import PowerTransformer

from sklearn.preprocessing import PowerTransformer

# Instantiate PowerTransformer

pow\_trans = PowerTransformer()

# Train the transform on the data

pow\_trans.fit(so\_numeric\_df[['ConvertedSalary']])

# Apply the power transform to the data

so\_numeric\_df['ConvertedSalary\_LG'] = pow\_trans.transform(so\_numeric\_df[['ConvertedSalary']])

# Plot the data before and after the transformation

so\_numeric\_df[['ConvertedSalary', 'ConvertedSalary\_LG']].hist()

plt.show()

---

---When can you use normalization?---

When could you use normalization (MinMaxScaler) when working with a dataset?

Answer: When you know the the data has a strict upper and lower bound.

---

--- Removing outliers ---

--- Percentage based outlier removal ---

# Find the 95th quantile

quantile = so\_numeric\_df['ConvertedSalary'].quantile(0.95)

# Trim the outliers

trimmed\_df = so\_numeric\_df[so\_numeric\_df['ConvertedSalary'] < quantile]

# The original histogram

so\_numeric\_df[['ConvertedSalary']].hist()

plt.show()

plt.clf()

# The trimmed histogram

trimmed\_df[['ConvertedSalary']].hist()

plt.show()

---

--- Statistical outlier removal ---

# Find the mean and standard dev

std = so\_numeric\_df['ConvertedSalary'].std()

mean = so\_numeric\_df['ConvertedSalary'].mean()

# Calculate the cutoff

cut\_off = std \* 3

lower, upper = mean - cut\_off, mean + cut\_off

# Trim the outliers

trimmed\_df = so\_numeric\_df[(so\_numeric\_df['ConvertedSalary'] < upper) \

& (so\_numeric\_df['ConvertedSalary'] > lower)]

# The trimmed box plot

trimmed\_df[['ConvertedSalary']].boxplot()

plt.show()

----

---Scaling and transforming new data---

---Train and testing transformations (I)---

# Import StandardScaler

from sklearn.preprocessing import StandardScaler

# Apply a standard scaler to the data

SS\_scaler = StandardScaler()

# Fit the standard scaler to the data

SS\_scaler.fit(so\_train\_numeric[['Age']])

# Transform the test data using the fitted scaler

so\_test\_numeric['Age\_ss'] = SS\_scaler.transform(so\_test\_numeric[['Age']])

print(so\_test\_numeric[['Age', 'Age\_ss']].head())

----

---Train and testing transformations (II)---

train\_std = so\_train\_numeric['ConvertedSalary'].std()

train\_mean = so\_train\_numeric['ConvertedSalary'].mean()

cut\_off = train\_std \* 3

train\_lower, train\_upper = train\_mean - cut\_off, train\_mean + cut\_off

# Trim the test DataFrame

trimmed\_df = so\_test\_numeric[(so\_test\_numeric['ConvertedSalary'] < train\_upper) \

& (so\_test\_numeric['ConvertedSalary'] > train\_lower)]

------

**Dealing with Text Data**

----Encoding text----

---Cleaning up your text---

# Print the first 5 rows of the text column

print(speech\_df.text.head())

----

---Cleaning up your text---

# Replace all non letter characters with a whitespace

speech\_df['text\_clean'] = speech\_df['text'].str.replace('[^a-zA-Z]', ' ')

# Change to lower case

speech\_df['text\_clean'] = speech\_df['text\_clean'].str.lower()

# Print the first 5 rows of the text\_clean column

print(speech\_df['text\_clean'].head())

---

---High level text features---

# Find the length of each text

speech\_df['char\_cnt'] = speech\_df['text\_clean'].str.len()

# Count the number of words in each text

speech\_df['word\_cnt'] = speech\_df['text\_clean'].str.split().str.len()

# Find the average length of word

speech\_df['avg\_word\_length'] = speech\_df['char\_cnt'] / speech\_df['word\_cnt']

# Print the first 5 rows of these columns

print(speech\_df[['text\_clean', 'char\_cnt', 'word\_cnt', 'avg\_word\_length']])

---

---Word counts----

---Counting words (I)---

# Import CountVectorizer

from sklearn.feature\_extraction.text import CountVectorizer

# Instantiate CountVectorizer

cv = CountVectorizer()

# Fit the vectorizer

cv.fit(speech\_df['text\_clean'])

# Print feature names

print(cv.get\_feature\_names())

---

---Counting words (II)---

# Apply the vectorizer

cv\_transformed = cv.transform(speech\_df['text\_clean'])

# Print the full array

cv\_array = cv\_transformed.toarray()

print(cv\_array)

---

# Apply the vectorizer

cv\_transformed = cv.transform(speech\_df['text\_clean'])

# Print the full array

cv\_array = cv\_transformed.toarray()

# Print the shape of cv\_array

print(cv\_array.shape)

---

---Limiting your features---

# Import CountVectorizer

from sklearn.feature\_extraction.text import CountVectorizer

# Specify arguements to limit the number of features generated

cv = CountVectorizer(min\_df=0.2, max\_df=0.8)

# Fit, transform, and convert into array

cv\_transformed = cv.fit\_transform(speech\_df['text\_clean'])

cv\_array = cv\_transformed.toarray()

# Print the array shape

print(cv\_array.shape)

---

---Text to DataFrame---

# Create a DataFrame with these features

cv\_df = pd.DataFrame(cv\_array,

columns=cv.get\_feature\_names()).add\_prefix('Counts\_')

# Add the new columns to the original DataFrame

speech\_df\_new = pd.concat([speech\_df, cv\_df], axis=1, sort=False)

print(speech\_df\_new.head())

----

---Term frequency-inverse document frequency---

---Tf-idf---

# Import TfidfVectorizer

from sklearn.feature\_extraction.text import TfidfVectorizer

# Instantiate TfidfVectorizer

tv = TfidfVectorizer(max\_features=100, stop\_words='english')

# Fit the vectroizer and transform the data

tv\_transformed = tv.fit\_transform(speech\_df['text\_clean'])

# Create a DataFrame with these features

tv\_df = pd.DataFrame(tv\_transformed.toarray(),

columns=tv.get\_feature\_names()).add\_prefix('TFIDF\_')

print(tv\_df.head())

----

---Transforming unseen data---

# Instantiate TfidfVectorizer

tv = TfidfVectorizer(max\_features=100, stop\_words='english')

# Fit the vectroizer and transform the data

tv\_transformed = tv.fit\_transform(train\_speech\_df['text\_clean'])

# Transform test data

test\_tv\_transformed = tv.transform(test\_speech\_df['text\_clean'])

# Create new features for the test set

test\_tv\_df = pd.DataFrame(test\_tv\_transformed.toarray(),

columns=tv.get\_feature\_names()).add\_prefix('TFIDF\_')

print(test\_tv\_df.head())

---

---N-grams----

---Using longer n-grams---

# Import CountVectorizer

from sklearn.feature\_extraction.text import CountVectorizer

# Instantiate a trigram vectorizer

cv\_trigram\_vec = CountVectorizer(max\_features=100,

stop\_words='english',

ngram\_range = (3,3))

# Fit and apply trigram vectorizer

cv\_trigram = cv\_trigram\_vec.fit\_transform(speech\_df['text\_clean'])

# Print the trigram features

print(cv\_trigram\_vec.get\_feature\_names())

---

---Finding the most common words---

# Create a DataFrame of the features

cv\_tri\_df = pd.DataFrame(cv\_trigram.toarray(),

columns=cv\_trigram\_vec.get\_feature\_names()).add\_prefix('Counts\_')

# Print the top 5 words in the sorted output

print(cv\_tri\_df.sum().sort\_values(ascending=False).head())

----

**--- Thank You ---**