**T-tests**

T-tests acknowledge that we won’t know the population standard deviation in the real world. They use sample data. Increased degrees of freedom brings the t distribution towards the standard normal distribution.The t distribution is essentially a means of normally distributed data. When we use a t-statistic, we are checking that a mean falls within a certain α probability of the mean of means. So the t statistic returned from a t test is the number of standard deviations away from the mean. The p-value, which is also returned, is the probability that the t statistic could have happened by random chance. From this we derive the confidence level, essentially lower the p-value the better, anything about .05 means that you must throw out the results of the t-test and assume the null hypothesis.

Confidence Interval == Bounds of statistical significance for our t-test

A sample mean that falls inside of our confidence interval will "FAIL TO REJECT" our null hypothesis

A sample mean that falls outside of our confidence interval will "REJECT" our null hypothesis

Downsides

T-test assumes that the means of the data are independent, and that the outcome of one does not affect another.

Assumes that the means are distributed normally (use this to test for normal distribution from scipy.stats import normaltest)

**Chi-square test or X^2 test**

Tests the independence/dependence of categorical values

# Chi square can take any crosstab/table and test the independence of rows/cols

# The null hypothesis is that the rows/cols are independent -> low chi square

# The alternative is that there is a dependence -> high chi square

# Be aware! Chi square does \*not\* tell you direction/causation

is used to determine whether there is a statistically significant difference between the expected frequencies and the observed frequencies in one or more categories of a contingency table.

To use, make a crosstab between the variables you woul d like to compare. Then use the scipy.stats.chi2\_contingency library to execute. A high value tells us that the features are correlated.

**Confidence Intervals**

Formula is approx as follows:

X +- t(S/square root of n)

t= margin of error

S = sample standard deviation

n = degrees of freedom

A confidence interval is going to tell you within what percentage the accuracy of our mean is

In code:

data = np.array(data)

#our estimate

mean = np.mean(data)

#degrees of freeom

n = len(data)

#adding ddof tells it to use sample data not population data

stderr = stats.sem(data, ddof= 1)/np.sqrt(n)

#moe: measure of error

moe = stderr \* stats.t.ppf((1 + confidence) / 2.0, n - 1)

return (mean - moe, mean, mean + moe)

**Bayes theorem**

In words - the probability of A conditioned on B is the probability of B conditioned on A , times the probability of A and divided by the probability of B . These unconditioned probabilities are referred to as "prior beliefs", and the conditioned probabilities as "updated."

Would recommend referring to LS\_DS\_123 to get full idea.

**Linear Algebra in DS**

Vectors - Rows, columns, lists, arrays

Matrices - tables, dataframes

Linear regression

To software, an image is a matrix

Norm of vector - length of the vector, can be found using pythagorean theorem

Dot product - equal to the sum of pair-wise products of the components of vectors a and b, in other words, match up the numbers in each vector with their corresponding opposite, multiply those two numbers together, and the dot product is all those multiplications added together

Cross product - vector equivalent of multiplication, look up example for intuition

Dimensionality - number of rows and columns in a matrix, important to list rows, columns as dimension

transpose - when the the rows become the columns and the columns become the rows

Square matrix- any matrix with the same number of rows and columns

Diagonal square matrix - zeros everywhere except main diagonal

Upper triangular square matrix- values on and above main diagonal

Lower Triangular- values on and below main diagonal

Identity matrix - ones on the diagonal, zeros everywhere else. This will occur with any matrix where you multiply it by its inverse

Determinant- all square matrices have one, if you want to know the intuition, look it up

Variance- measure of the spread of numbers in a dataset, found by subtracting the mean of the points from the original points, square everything, sum it up, and divide by the number of rows

* The POPULATION VARIANCE σ2 is a PARAMETER (aspect, property, attribute, etc) of the population.
* The SAMPLE VARIANCE s2 is a STATISTIC (estimated attribute) of the sample.

Standard deviation - square root of the variance, this puts variance back into the same unit of measurement as our OG data

Covariance - how changes in one variable are associated with changes in a second variable. It does not measure the strength of a linear relationship, simply how one varies while the other varies

Correlation coefficient - found by dividing covariance by standard deviation, fixed range of -1 to 1, with 0 representing no linear relationship in the data. This measures the strength of the covariance

Span - set of all possible vectors that can be created with a linear combination of two vectors

Linearly dependent vectors - two vectors that live on the same line, meaning there is no way with math that we could ever create a vector that lies outside of that line

Linearly independent vectors - vectors that don’t lie on the same line, there should be a combination of them that could represent any vector in space

Eigenvector - vector does not change orientation during a matrix transformation (can still be scaled)

Eigenvalue - scalar that represents how the eigenvector was scaled during transformation

**PCA (Principal Component Analysis) -** lowers dimensionality of dataset while preserving as much information as possible. Reminder that this creates a whole new dataset, and the new columns are unlabelled. You must remember to standardize the data before performing PCA. Typically, you want to keep your explained variance to at least 90%, meaning only 10% of relationships were lost in the transformation.

<https://www.youtube.com/watch?v=FgakZw6K1QQ>

PCA steps (for intuition):

1. Separate data into target and features
2. Subtract the mean of each column from itself to center it at 0
3. Divide each column by its standard deviation (standardizing the data)
4. Calculate the variance-covariance matrix

(the code: cov(standardized\_data.T))

1. Calculate the eigenvectors and eigenvalues of the variance-covariance matrix
2. Sort the eigenvectors and eigenvalues
3. Use matrix transformation to project the datapoints into our eigenvector subspaces

Using libraries, as you normally will:

from numpy import array

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

# define a matrix

X = array([[1, 2], [3, 4], [5, 6]])

print("Data: \n", X)

# Standardize the Data

# Instantiate a Standard Scaler object

scaler = StandardScaler()

# Use the object to fit\_transform our data

Z = scaler.fit\_transform(X)

print("\n Standardized Data: \n", Z)

# create the PCA instance

pca = PCA(2)

# fit on data

pca.fit(Z)

# access values and vectors

print("\n Eigenvectors: \n", pca.components\_)

print("\n Eigenvalues: \n",pca.explained\_variance\_)

# transform data

B = pca.transform(Z)

print("\n Projected Data: \n", B)

**Scree plot** - shows how much explained variance each principal component has after performing PCA.

An example:

pca = PCA(15)

pca\_features = pca.fit\_transform(national\_processed)

def scree\_plot(pca):

num\_components=len(pca.explained\_variance\_ratio\_)

ind = np.arange(num\_components)

vals = pca.explained\_variance\_ratio\_

plt.figure(figsize=(18, 6))

ax = plt.subplot(111)

cumvals = np.cumsum(vals)

ax.bar(ind, vals)

ax.plot(ind, cumvals)

for i in range(num\_components):

ax.annotate(r"%s" % ((str(vals[i]\*100)[:3])), (ind[i], vals[i]), va="bottom", ha="center", fontsize=4.5)

ax.xaxis.set\_tick\_params(width=0)

ax.yaxis.set\_tick\_params(width=2, length=12)

ax.set\_xlabel("Principal Component")

ax.set\_ylabel("Variance Explained (%)")

plt.title('Explained Variance Per Principal Component')

**Supervised Learning** - ML learning where training data outputs, the prediction, is labelled. Classification (predicting a category) and Regression (predicting a continuous value) are examples of supervised learning

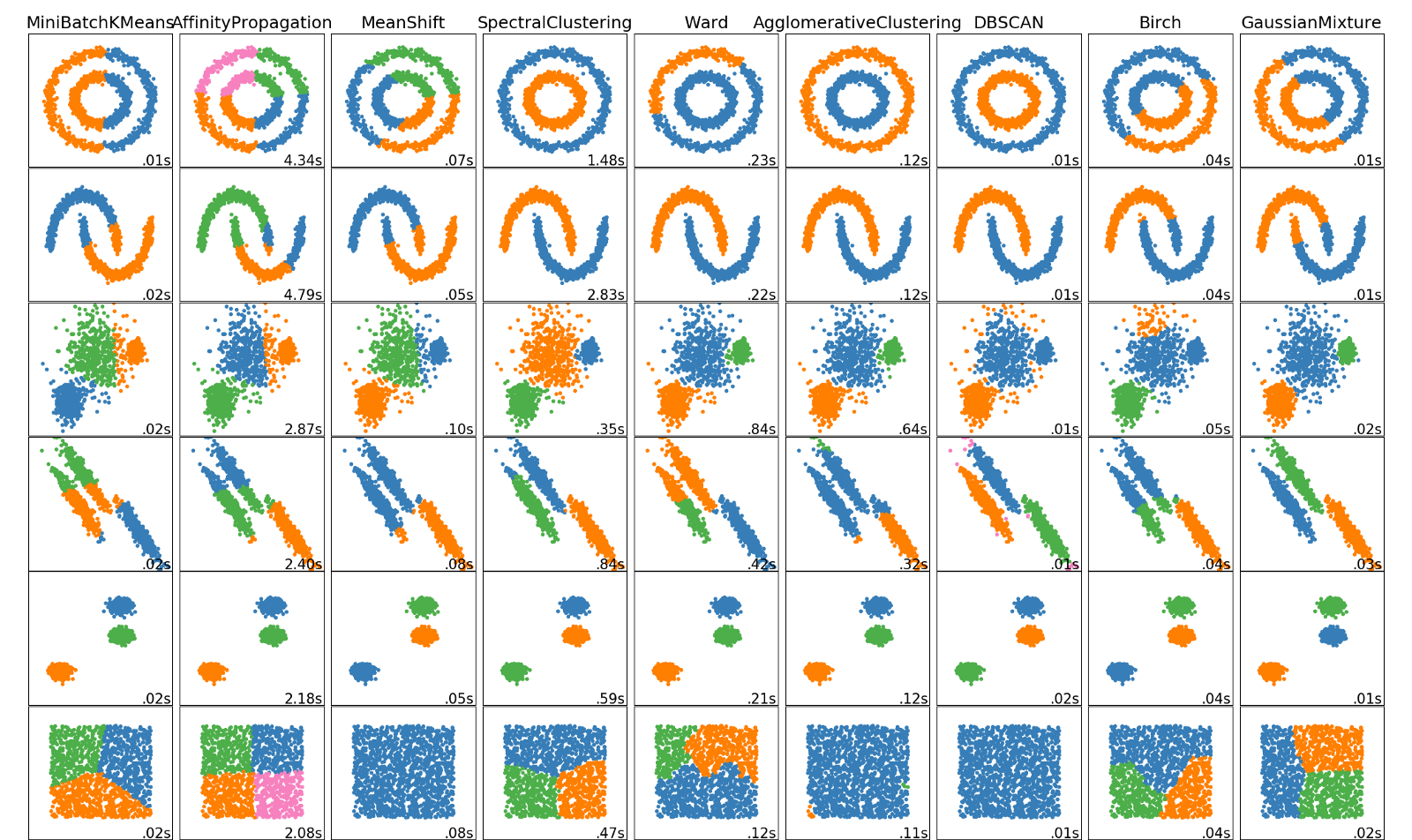
**Unsupervised Learning** - Unlabelled, perhaps entirely unable to interpret why, outputs. Clustering, dimensionality reduction, and association rule learning are examples of this

**Reinforcement Learning** - form of machine learning where “agent” interacts with its environment, and is rewarded for correct behavior and penalized for incorrect behavior

**Clustering types:**

* Hierarchical
  + Agglomerative, start with individual points and combine them into larger and larger clusters
  + Divisive, start with one cluster and divide the points into smaller clusters
* Point assignment, decide on number of clusters out the gate
* Hard clustering, assigns points to cluster
* Soft clustering, assigns each point a probability that it is in a given cluster

Display of different variances of data being clustered by different algorithms:



**Elbow plot** - essential for executing kmeans by deciding the optimal number of clusters, sample code below

sum\_of\_squared\_distances = []

K = range(1,15)

for k in K:

km = KMeans(n\_clusters=k)

km = km.fit(points)

sum\_of\_squared\_distances.append(km.inertia\_)

plt.plot(K, sum\_of\_squared\_distances, 'bx-')

plt.xlabel('k')

plt.ylabel('Sum\_of\_squared\_distances')

plt.title('Elbow Method For Optimal k')

**Heuristics -** the “rule of thumb” people use to make judgements

**Baseline** - could mean a few things:

1. Score you would get by guessing the largest group, or the mean of the target
2. Fast to make models that beat guessing
3. Complete, tuned but simple model
4. Minimum viable product
5. Human level performance

**Linear Regression** - for continuous data, uses ordinary least squares to create a line of best fit, for data with a linear relationship. You can get the feature importances with the following code:

model = pipeline.named\_steps['logisticregression']

encoder = pipeline.named\_steps['onehotencoder']

encoded\_columns = encoder.transform(X\_val).columns

coefficients = pd.Series(model.coef\_[0], encoded\_columns)

plt.figure(figsize=(10,30))

coefficients.sort\_values().plot.barh(color='grey');

**Logistic Regression** - Like linear, but for class data.

Here are some suggested hyperparameters to look at while tuning:

* C
* class\_weight (for imbalanced classes)
* penalty

**Encoding** - transform categorical variables into numerical values

* onehotencoder makes a new columns for each
* Ordinal encoding gives each new string a number in the column, not a new column

**Univariate Feature Selection** - smarter ways of feature selection you use in a model! There are many of them, here are a few:

* SelectKBest
* Permutation importance
* <https://scikit-learn.org/stable/modules/feature_selection.html>

**Ridge Regression** - similar to linear regression, but bias is added in to avoid overfitting the data

**Profile report** - does a fantastic job looking for things you want to fix in dirty data

import pandas\_profiling

profile\_report = train.profile\_report(

check\_correlation\_pearson=False,

correlations={

'pearson': False,

'spearman': False,

'kendall': False,

'phi\_k': False,

'cramers': False,

'recoded': False,

},

plot={'histogram': {'bayesian\_blocks\_bins': False}},

)

profile\_report

**Pipelines** - chains multiple processing steps together, minimizes human error of training different transformers on the wrong version of the dataset. Basic setup below:

import category\_encoders as ce

from sklearn.impute import SimpleImputer

from sklearn.linear\_model import LogisticRegression

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

pipeline = make\_pipeline(

ce.OneHotEncoder(use\_cat\_names=True),

SimpleImputer(),

StandardScaler(),

LogisticRegression(multi\_class='auto', solver='lbfgs', n\_jobs=-1)

)

#fit model

pipeline.fit(X\_train, y\_train)

#score on validation

print('Validation Accuracy:', pipeline.score(X\_val, y\_val))

#predict on x\_test

y\_pred = pipeline.predict(X\_test)

**Decision Trees** - ML great for non-linear, non-monotonic data. The leaf size and depth are very influential for decision trees. A single decision tree, with unlimited depth, will overfit. This creates the basis for Random Forest and Gradient Boosting

To plot a decision tree, you can use the following code (you don’t want to plot a deep one):

import graphviz

from sklearn.tree import export\_graphviz

model = pipeline.named\_steps['decisiontreeclassifier']

encoder = pipeline.named\_steps['onehotencoder']

encoded\_columns = encoder.transform(X\_val).columns

dot\_data = export\_graphviz(model,

out\_file=None,

max\_depth=3, #this stops the tree viz from going on and on

feature\_names=encoded\_columns,

class\_names=model.classes\_,

impurity=False,

filled=True,

proportion=True,

rounded=True)

display(graphviz.Source(dot\_data))

To find feature importances, use the following code:

encoder= pipeline.named\_steps['onehotencoder']

encoded\_columns = encoder.transform(X\_val).columns

importances = pd.Series(model.feature\_importances\_, encoded\_columns)

plt.figure(figsize=(10, 30))

importances.sort\_values().plot.barh(color='grey')

**Random Forest** - uses decision trees (see above for further info) . Each tree trains on a random bootstrap of the data (booststrapping takes one sample at a time and returns them to the original population after they have been chosen <https://machinelearningmastery.com/a-gentle-introduction-to-the-bootstrap-method/>).

Each of these splits considers a random subset of features. You can create extra-random trees in sklearn. For further building of intuition, check out the second lecture in our kaggle challenge week of predictive modeling.

Here are some suggested hyper parameters to look at first while tuning:

* class\_weight (for imbalanced classes)
* max\_depth (usually high, can try decreasing)
* n\_estimators (too low underfits, too high wastes time)
* min\_samples\_leaf (increase if overfitting)
* max\_features (decrease for more diverse trees)

**Validation/Cross-validation** - Generally, creating a validation set is to avoid overfitting to the test set after iteratively parameter tuning. If your dataset is too small to separate into three groups, cross-validation is a good method to avoid overfitting. It can be computationally expensive, since it needs to run the same model multiple times with different sections of data selected as the validation group.

**Randomized Search CV (cross-validation)** - This allows you to iterate through different tunings of your hyperparameters. GridSearchCV tries each and every one, randomized uses a random sample to test. The n\_iter parameter limits the number of iterations of this. Below is an example of using it with a pipeline:

from sklearn.model\_selection import GridSearchCV, RandomizedSearchCV

features = train.columns.drop([target] + high\_cardinality)

X\_train = train[features]

y\_train = train[target]

pipeline = make\_pipeline(

ce.OneHotEncoder(use\_cat\_names=True),

SimpleImputer(),

StandardScaler(),

SelectKBest(f\_regression),

Ridge()

)

param\_distributions = {

'simpleimputer\_\_strategy': ['mean', 'median'],

'selectkbest\_\_k': range(1, len(X\_train.columns)+1),

'ridge\_\_alpha': [0.1, 1, 10],

}

# If you're on Colab, decrease n\_iter & cv parameters

search = RandomizedSearchCV(

pipeline,

param\_distributions=param\_distributions,

n\_iter=100,

cv=5,

scoring='neg\_mean\_absolute\_error',

verbose=10,

return\_train\_score=True,

n\_jobs=-1

)

search.fit(X\_train, y\_train);

print('Best hyperparameters', search.best\_params\_)

print('Cross-validation MAE', -search.best\_score\_)

**XGBoost** - see above random forest and decision tree for info on tree models. Here are some suggested hyperparameters to look at first while tuning:

* scale\_pos\_weight (for imbalanced classes)
* max\_depth (usually low, can try increasing)
* n\_estimators (too low underfits, too high wastes time/overfits) — *I recommend using early stopping instead of cross-validation*
* learning\_rate (too low underfits, too high overfits)

**Confusion matrix** - These can be used to easily sum up the trade off of the results in your model, and you can use it to find the precision and recall of your model.

<https://colab.research.google.com/drive/1swagZjjeBpnNywfpH94uUD-OouwSbv6D#scrollTo=Mc3QpwxkGgZJ> (this is a link to my 4th assignment of the kaggle challenge week of predictive modeling)

If you want a pretty version of the confusion matrix, this code will work well:

from sklearn.metrics import confusion\_matrix

from sklearn.utils.multiclass import unique\_labels

def plot\_confusion\_matrix(y\_true, y\_pred):

labels = unique\_labels(y\_true)

columns = [f'Predicted {label}' for label in labels]

index = [f'Actual {label}' for label in labels]

table = pd.DataFrame(confusion\_matrix(y\_true, y\_pred),

columns=columns, index=index)

return table

plot\_confusion\_matrix(y\_val, y\_pred)

**Precision** - a measure of how good your model is at finding *correct* instances of your target. This can be found by dividing your true positives by true positives + false positives. The trade off of high precision is also having many false negatives (low recall). <https://colab.research.google.com/drive/1swagZjjeBpnNywfpH94uUD-OouwSbv6D#scrollTo=Mc3QpwxkGgZJ>

**Recall** - a measure of how good your model is at finding *all* instances of your target. This can be found by dividing your true positives by true positives + false negatives. The trade off of high precision is having recall is having many false positives (low precision). <https://colab.research.google.com/drive/1swagZjjeBpnNywfpH94uUD-OouwSbv6D#scrollTo=Mc3QpwxkGgZJ>

**ROC AUC**  - This creates a plot that illustrates the ability of a binary classification model as the threshold is varied. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) while varying the threshold. The actual ROC AUC is the area that is beneath this ROC curve (0-1, ex is the area takes up 83% of the area beneath the ROC, so ROC AUC is .83). Essentially, this measures how well your classification model predicted things. The higher the better! The changing threshold reflects the tradeoff between precision and recall, the further right you go the higher the precision (lower the recall) and the further left you go the higher the recall (lower the precision).

<https://colab.research.google.com/drive/1swagZjjeBpnNywfpH94uUD-OouwSbv6D#scrollTo=Mc3QpwxkGgZJ>

One method of plotting this is as follows:

from sklearn.metrics import roc\_curve

y\_pred\_proba = pipeline.predict\_proba(X\_val)[:, 1]

fpr, tpr, thresholds = roc\_curve(y\_val, y\_pred\_proba)

plt.scatter(fpr, tpr)

plt.plot(fpr, tpr)

plt.title('ROC curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate');

To get the AUC score:

from sklearn.metrics import roc\_auc\_score

roc\_auc\_score(y\_val, y\_pred\_proba)