In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
import sklearn
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score, cl
from sklearn.metrics import confusion_matrix as cm
```

In [2]:
```
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)

X.shape
```

Out[2]:
```
(70000, 784)
```

In [3]:
```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = 0.2,
                                                     random_state=0)
```

In [4]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(56000, 784)
(56000,)
(14000, 784)
(14000,)
```

Decision Tree

In [5]:
```
from sklearn.tree import DecisionTreeClassifier
DecisionTreeClassifier?
dt = DecisionTreeClassifier(max_depth = 2, min_samples_leaf = 4)
```

```
Init signature:
DecisionTreeClassifier(
    *,
    criterion='gini',
    splitter='best',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    min_weight_fraction_leaf=0.0,
    max_features=None,
    random_state=None,
    max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    class_weight=None,
    ccp_alpha=0.0,
)
Docstring:
A decision tree classifier.

Read more in the :ref:`User Guide <tree>`.

Parameters
----------
criterion : {"gini", "entropy", "log_loss"}, default="gini"
    The function to measure the quality of a split. Supported criteria are
    "gini" for the Gini impurity and "log_loss" and "entropy" both for the
    Shannon information gain, see :ref:`tree_mathematical_formulation`.

splitter : {"best", "random"}, default="best"
    The strategy used to choose the split at each node. Supported
    strategies are "best" to choose the best split and "random" to choose
    the best random split.

max_depth : int, default=None
    The maximum depth of the tree. If None, then nodes are expanded until
    all leaves are pure or until all leaves contain less than
    min_samples_split samples.

min_samples_split : int or float, default=2
    The minimum number of samples required to split an internal node:

    - If int, then consider `min_samples_split` as the minimum number.
    - If float, then `min_samples_split` is a fraction and
      `ceil(min_samples_split * n_samples)` are the minimum
      number of samples for each split.

    .. versionchanged:: 0.18
       Added float values for fractions.

min_samples_leaf : int or float, default=1
    The minimum number of samples required to be at a leaf node.
    A split point at any depth will only be considered if it leaves at
    least ``min_samples_leaf`` training samples in each of the left and
    right branches.  This may have the effect of smoothing the model,
    especially in regression.

    - If int, then consider `min_samples_leaf` as the minimum number.
    - If float, then `min_samples_leaf` is a fraction and
      `ceil(min_samples_leaf * n_samples)` are the minimum
      number of samples for each node.
```

```
        .. versionchanged:: 0.18
           Added float values for fractions.

    min_weight_fraction_leaf : float, default=0.0
        The minimum weighted fraction of the sum total of weights (of all
        the input samples) required to be at a leaf node. Samples have
        equal weight when sample_weight is not provided.

    max_features : int, float or {"auto", "sqrt", "log2"}, default=None
        The number of features to consider when looking for the best split:

            - If int, then consider `max_features` features at each split.
            - If float, then `max_features` is a fraction and
              `max(1, int(max_features * n_features_in_))` features are considered at
              each split.
            - If "sqrt", then `max_features=sqrt(n_features)`.
            - If "log2", then `max_features=log2(n_features)`.
            - If None, then `max_features=n_features`.

        Note: the search for a split does not stop until at least one
        valid partition of the node samples is found, even if it requires to
        effectively inspect more than ``max_features`` features.

    random_state : int, RandomState instance or None, default=None
        Controls the randomness of the estimator. The features are always
        randomly permuted at each split, even if ``splitter`` is set to
        ``"best"``. When ``max_features < n_features``, the algorithm will
        select ``max_features`` at random at each split before finding the best
        split among them. But the best found split may vary across different
        runs, even if ``max_features=n_features``. That is the case, if the
        improvement of the criterion is identical for several splits and one
        split has to be selected at random. To obtain a deterministic behaviour
        during fitting, ``random_state`` has to be fixed to an integer.
        See :term:`Glossary <random_state>` for details.

    max_leaf_nodes : int, default=None
        Grow a tree with ``max_leaf_nodes`` in best-first fashion.
        Best nodes are defined as relative reduction in impurity.
        If None then unlimited number of leaf nodes.

    min_impurity_decrease : float, default=0.0
        A node will be split if this split induces a decrease of the impurity
        greater than or equal to this value.

        The weighted impurity decrease equation is the following::

            N_t / N * (impurity - N_t_R / N_t * right_impurity
                                - N_t_L / N_t * left_impurity)

        where ``N`` is the total number of samples, ``N_t`` is the number of
        samples at the current node, ``N_t_L`` is the number of samples in the
        left child, and ``N_t_R`` is the number of samples in the right child.

        ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
        if ``sample_weight`` is passed.

        .. versionadded:: 0.19

    class_weight : dict, list of dict or "balanced", default=None
```

```
    Weights associated with classes in the form ``{class_label: weight}``.
    If None, all classes are supposed to have weight one. For
    multi-output problems, a list of dicts can be provided in the same
    order as the columns of y.

    Note that for multioutput (including multilabel) weights should be
    defined for each class of every column in its own dict. For example,
    for four-class multilabel classification weights should be
    [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
    [{1:1}, {2:5}, {3:1}, {4:1}].

    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input data
    as ``n_samples / (n_classes * np.bincount(y))``

    For multi-output, the weights of each column of y will be multiplied.

    Note that these weights will be multiplied with sample_weight (passed
    through the fit method) if sample_weight is specified.

ccp_alpha : non-negative float, default=0.0
    Complexity parameter used for Minimal Cost-Complexity Pruning. The
    subtree with the largest cost complexity that is smaller than
    ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
    :ref:`minimal_cost_complexity_pruning` for details.

    .. versionadded:: 0.22

Attributes
----------
classes_ : ndarray of shape (n_classes,) or list of ndarray
    The classes labels (single output problem),
    or a list of arrays of class labels (multi-output problem).

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature.  It is also
    known as the Gini importance [4]_.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

max_features_ : int
    The inferred value of max_features.

n_classes_ : int or list of int
    The number of classes (for single output problems),
    or a list containing the number of classes for each
    output (for multi-output problems).

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
```

```
    has feature names that are all strings.

    .. versionadded:: 1.0

n_outputs_ : int
    The number of outputs when ``fit`` is performed.

tree_ : Tree instance
    The underlying Tree object. Please refer to
    ``help(sklearn.tree._tree.Tree)`` for attributes of Tree object and
    :ref:`sphx_glr_auto_examples_tree_plot_unveil_tree_structure.py`
    for basic usage of these attributes.

See Also
--------
DecisionTreeRegressor : A decision tree regressor.

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The :meth:`predict` method operates using the :func:`numpy.argmax`
function on the outputs of :meth:`predict_proba`. This means that in
case the highest predicted probabilities are tied, the classifier will
predict the tied class with the lowest index in :term:`classes_`.

References
----------

.. [1] https://en.wikipedia.org/wiki/Decision_tree_learning

.. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification
       and Regression Trees", Wadsworth, Belmont, CA, 1984.

.. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical
       Learning", Springer, 2009.

.. [4] L. Breiman, and A. Cutler, "Random Forests",
       https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Examples
--------
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
...                             # doctest: +SKIP
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
File:           c:\users\biggest\anaconda3\lib\site-packages\sklearn\tree\_classes.py
Type:           ABCMeta
Subclasses:     ExtraTreeClassifier
```

In [6]:
```
dt.fit(X_train, y_train)
```

Out[6]:
```
DecisionTreeClassifier(max_depth=2, min_samples_leaf=4)
```

In [7]:
```
# MODEL EVALUATION
y_pred = dt.predict(X_test)
accuracy_score(y_test, y_pred)
```

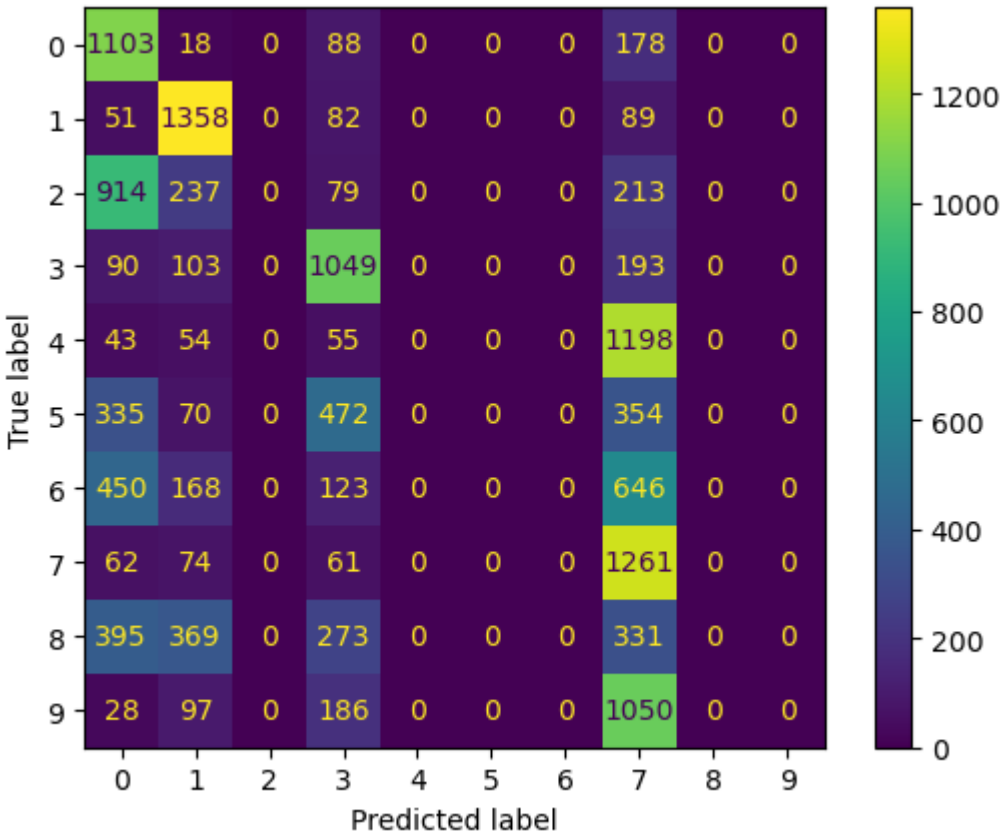
Out[7]:
```
0.3407857142857143
```

In [8]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, labels=dt.classes_.tolist()))
```

```
              precision    recall  f1-score   support

           0       0.32      0.80      0.45      1387
           1       0.53      0.86      0.66      1580
           2       0.00      0.00      0.00      1443
           3       0.43      0.73      0.54      1435
           4       0.00      0.00      0.00      1350
           5       0.00      0.00      0.00      1231
           6       0.00      0.00      0.00      1387
           7       0.23      0.86      0.36      1458
           8       0.00      0.00      0.00      1368
           9       0.00      0.00      0.00      1361

    accuracy                           0.34     14000
   macro avg       0.15      0.33      0.20     14000
weighted avg       0.16      0.34      0.21     14000
```

In [9]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
 #from sklearn.metrics import classification_report
 #assuming 'knn' is your trained model, 'X_test' are your test features
predictions = dt.predict(X_test)
cm = confusion_matrix(y_test, predictions)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=dt.classes_)
disp.plot()
plt.suptitle
```

Out[9]:
```
<function matplotlib.pyplot.suptitle(t, **kwargs)>
```



BAGGING

In [47]:
```
from sklearn.ensemble import BaggingClassifier
BaggingClassifier?

from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5)
bag = BaggingClassifier(knn,
                        max_samples = .5,
                        max_features = 28,
                         n_estimators = 20)
```

```
Init signature:
BaggingClassifier(
    estimator=None,
    n_estimators=10,
    *,
    max_samples=1.0,
    max_features=1.0,
    bootstrap=True,
    bootstrap_features=False,
    oob_score=False,
    warm_start=False,
    n_jobs=None,
    random_state=None,
    verbose=0,
    base_estimator='deprecated',
)
Docstring:
A Bagging classifier.

A Bagging classifier is an ensemble meta-estimator that fits base
classifiers each on random subsets of the original dataset and then
aggregate their individual predictions (either by voting or by averaging)
to form a final prediction. Such a meta-estimator can typically be used as
a way to reduce the variance of a black-box estimator (e.g., a decision
tree), by introducing randomization into its construction procedure and
then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random
subsets of the dataset are drawn as random subsets of the samples, then
this algorithm is known as Pasting [1]_. If samples are drawn with
replacement, then the method is known as Bagging [2]_. When random subsets
of the dataset are drawn as random subsets of the features, then the method
is known as Random Subspaces [3]_. Finally, when base estimators are built
on subsets of both samples and features, then the method is known as
Random Patches [4]_.

Read more in the :ref:`User Guide <bagging>`.

.. versionadded:: 0.15

Parameters
----------
estimator : object, default=None
    The base estimator to fit on random subsets of the dataset.
    If None, then the base estimator is a
    :class:`~sklearn.tree.DecisionTreeClassifier`.

    .. versionadded:: 1.2
       `base_estimator` was renamed to `estimator`.

n_estimators : int, default=10
    The number of base estimators in the ensemble.

max_samples : int or float, default=1.0
    The number of samples to draw from X to train each base estimator (with
    replacement by default, see `bootstrap` for more details).

    - If int, then draw `max_samples` samples.
    - If float, then draw `max_samples * X.shape[0]` samples.
```

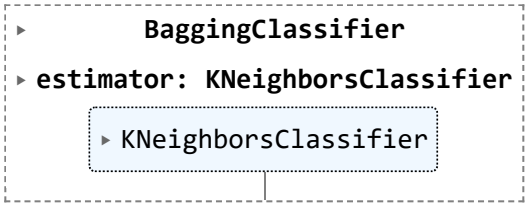
```
        max_features : int or float, default=1.0
            The number of features to draw from X to train each base estimator (
            without replacement by default, see `bootstrap_features` for more
            details).

            - If int, then draw `max_features` features.
            - If float, then draw `max(1, int(max_features * n_features_in_))` features.

        bootstrap : bool, default=True
            Whether samples are drawn with replacement. If False, sampling
            without replacement is performed.

        bootstrap_features : bool, default=False
            Whether features are drawn with replacement.

        oob_score : bool, default=False
            Whether to use out-of-bag samples to estimate
            the generalization error. Only available if bootstrap=True.

        warm_start : bool, default=False
            When set to True, reuse the solution of the previous call to fit
            and add more estimators to the ensemble, otherwise, just fit
            a whole new ensemble. See :term:`the Glossary <warm_start>`.

            .. versionadded:: 0.17
               *warm_start* constructor parameter.

        n_jobs : int, default=None
            The number of jobs to run in parallel for both :meth:`fit` and
            :meth:`predict`. ``None`` means 1 unless in a
            :obj:`joblib.parallel_backend` context. ``-1`` means using all
            processors. See :term:`Glossary <n_jobs>` for more details.

        random_state : int, RandomState instance or None, default=None
            Controls the random resampling of the original dataset
            (sample wise and feature wise).
            If the base estimator accepts a `random_state` attribute, a different
            seed is generated for each instance in the ensemble.
            Pass an int for reproducible output across multiple function calls.
            See :term:`Glossary <random_state>`.

        verbose : int, default=0
            Controls the verbosity when fitting and predicting.

        base_estimator : object, default="deprecated"
            Use `estimator` instead.

            .. deprecated:: 1.2
                `base_estimator` is deprecated and will be removed in 1.4.
                Use `estimator` instead.

        Attributes
        ----------
        estimator_ : estimator
            The base estimator from which the ensemble is grown.

            .. versionadded:: 1.2
               `base_estimator_` was renamed to `estimator_`.

        base_estimator_ : estimator
```

```
    The base estimator from which the ensemble is grown.

    .. deprecated:: 1.2
        `base_estimator_` is deprecated and will be removed in 1.4.
        Use `estimator_` instead.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

estimators_ : list of estimators
    The collection of fitted base estimators.

estimators_samples_ : list of arrays
    The subset of drawn samples (i.e., the in-bag samples) for each base
    estimator. Each subset is defined by an array of the indices selected.

estimators_features_ : list of arrays
    The subset of drawn features for each base estimator.

classes_ : ndarray of shape (n_classes,)
    The classes labels.

n_classes_ : int or list
    The number of classes.

oob_score_ : float
    Score of the training dataset obtained using an out-of-bag estimate.
    This attribute exists only when ``oob_score`` is True.

oob_decision_function_ : ndarray of shape (n_samples, n_classes)
    Decision function computed with out-of-bag estimate on the training
    set. If n_estimators is small it might be possible that a data point
    was never left out during the bootstrap. In this case,
    `oob_decision_function_` might contain NaN. This attribute exists
    only when ``oob_score`` is True.

See Also
--------
BaggingRegressor : A Bagging regressor.

References
----------

.. [1] L. Breiman, "Pasting small votes for classification in large
       databases and on-line", Machine Learning, 36(1), 85-103, 1999.

.. [2] L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140,
       1996.

.. [3] T. Ho, "The random subspace method for constructing decision
       forests", Pattern Analysis and Machine Intelligence, 20(8), 832-844,
       1998.
```

```
.. [4] G. Louppe and P. Geurts, "Ensembles on Random Patches", Machine
       Learning and Knowledge Discovery in Databases, 346-361, 2012.

Examples
--------
>>> from sklearn.svm import SVC
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=100, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = BaggingClassifier(estimator=SVC(),
...                         n_estimators=10, random_state=0).fit(X, y)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
File:           c:\users\biggest\anaconda3\lib\site-packages\sklearn\ensemble\_baggin
g.py
Type:           ABCMeta
Subclasses:
```

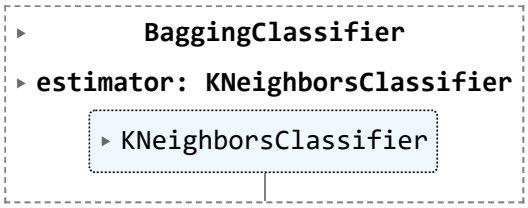
In [48]:
```
bag.fit(X_train, y_train)
```

Out[48]:



In [49]:
```
BaggingClassifier(estimator = KNeighborsClassifier(n_neighbors = 3),
                  max_features = 30,
                  max_samples = .5,
                  n_jobs = 5,
                  oob_score = True)
```

Out[49]:



In [50]:
```
# MODEL EVALUATION
y_pred = bag.predict(X_test)
accuracy_score(y_test, y_pred)
```

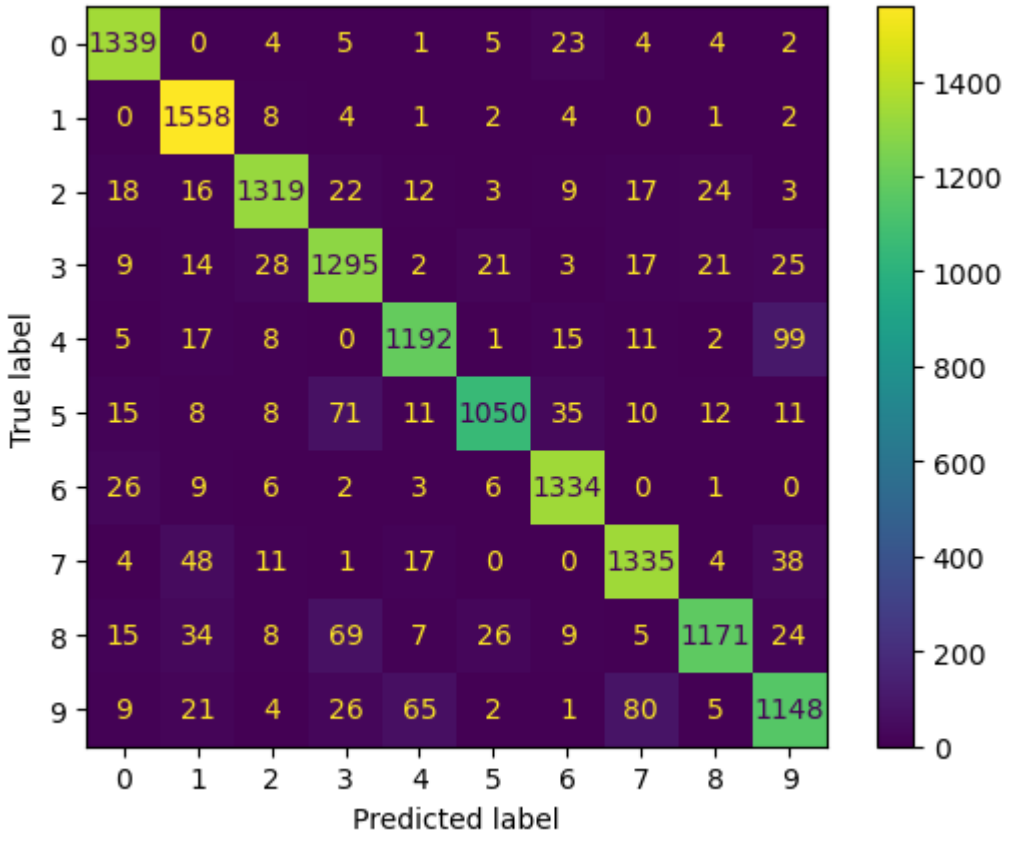
Out[50]:
```
0.9100714285714285
```

In [51]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, labels=bag.classes_.tolist()))
```

	precision	recall	f1-score	support
0	0.93	0.97	0.95	1387
1	0.90	0.99	0.94	1580
2	0.94	0.91	0.93	1443
3	0.87	0.90	0.88	1435
4	0.91	0.88	0.90	1350
5	0.94	0.85	0.89	1231
6	0.93	0.96	0.95	1387
7	0.90	0.92	0.91	1458
8	0.94	0.86	0.90	1368
9	0.85	0.84	0.85	1361
accuracy			0.91	14000
macro avg	0.91	0.91	0.91	14000
weighted avg	0.91	0.91	0.91	14000

In [52]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
 #from sklearn.metrics import classification_report
 #assuming 'knn' is your trained model, 'X_test' are your test features
predictions = bag.predict(X_test)
cm = confusion_matrix(y_test, predictions)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=bag.classes_)
disp.plot()
plt.suptitle
```

Out[52]:
```
<function matplotlib.pyplot.suptitle(t, **kwargs)>
```



RANDOM FOREST

In [17]:
```
from sklearn.ensemble import RandomForestClassifier
RandomForestClassifier?
```

```
Init signature:
RandomForestClassifier(
    n_estimators=100,
    *,
    criterion='gini',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    min_weight_fraction_leaf=0.0,
    max_features='sqrt',
    max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    bootstrap=True,
    oob_score=False,
    n_jobs=None,
    random_state=None,
    verbose=0,
    warm_start=False,
    class_weight=None,
    ccp_alpha=0.0,
    max_samples=None,
)
Docstring:
A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree
classifiers on various sub-samples of the dataset and uses averaging to
improve the predictive accuracy and control over-fitting.
The sub-sample size is controlled with the `max_samples` parameter if
`bootstrap=True` (default), otherwise the whole dataset is used to build
each tree.

For a comparison between tree-based ensemble models see the example
:ref:`sphx_glr_auto_examples_ensemble_plot_forest_hist_grad_boosting_comparison.py`.

Read more in the :ref:`User Guide <forest>`.

Parameters
----------
n_estimators : int, default=100
    The number of trees in the forest.

    .. versionchanged:: 0.22
       The default value of ``n_estimators`` changed from 10 to 100
       in 0.22.

criterion : {"gini", "entropy", "log_loss"}, default="gini"
    The function to measure the quality of a split. Supported criteria are
    "gini" for the Gini impurity and "log_loss" and "entropy" both for the
    Shannon information gain, see :ref:`tree_mathematical_formulation`.
    Note: This parameter is tree-specific.

max_depth : int, default=None
    The maximum depth of the tree. If None, then nodes are expanded until
    all leaves are pure or until all leaves contain less than
    min_samples_split samples.

min_samples_split : int or float, default=2
    The minimum number of samples required to split an internal node:
```

```
        - If int, then consider `min_samples_split` as the minimum number.
        - If float, then `min_samples_split` is a fraction and
          `ceil(min_samples_split * n_samples)` are the minimum
          number of samples for each split.

        .. versionchanged:: 0.18
           Added float values for fractions.

    min_samples_leaf : int or float, default=1
        The minimum number of samples required to be at a leaf node.
        A split point at any depth will only be considered if it leaves at
        least ``min_samples_leaf`` training samples in each of the left and
        right branches.  This may have the effect of smoothing the model,
        especially in regression.

        - If int, then consider `min_samples_leaf` as the minimum number.
        - If float, then `min_samples_leaf` is a fraction and
          `ceil(min_samples_leaf * n_samples)` are the minimum
          number of samples for each node.

        .. versionchanged:: 0.18
           Added float values for fractions.

    min_weight_fraction_leaf : float, default=0.0
        The minimum weighted fraction of the sum total of weights (of all
        the input samples) required to be at a leaf node. Samples have
        equal weight when sample_weight is not provided.

    max_features : {"sqrt", "log2", None}, int or float, default="sqrt"
        The number of features to consider when looking for the best split:

        - If int, then consider `max_features` features at each split.
        - If float, then `max_features` is a fraction and
          `max(1, int(max_features * n_features_in_))` features are considered at each
          split.
        - If "sqrt", then `max_features=sqrt(n_features)`.
        - If "log2", then `max_features=log2(n_features)`.
        - If None, then `max_features=n_features`.

        .. versionchanged:: 1.1
            The default of `max_features` changed from `"auto"` to `"sqrt"`.

        Note: the search for a split does not stop until at least one
        valid partition of the node samples is found, even if it requires to
        effectively inspect more than ``max_features`` features.

    max_leaf_nodes : int, default=None
        Grow trees with ``max_leaf_nodes`` in best-first fashion.
        Best nodes are defined as relative reduction in impurity.
        If None then unlimited number of leaf nodes.

    min_impurity_decrease : float, default=0.0
        A node will be split if this split induces a decrease of the impurity
        greater than or equal to this value.

        The weighted impurity decrease equation is the following::

            N_t / N * (impurity - N_t_R / N_t * right_impurity
                                - N_t_L / N_t * left_impurity)
```

```
        where ``N`` is the total number of samples, ``N_t`` is the number of
        samples at the current node, ``N_t_L`` is the number of samples in the
        left child, and ``N_t_R`` is the number of samples in the right child.

        ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
        if ``sample_weight`` is passed.

        .. versionadded:: 0.19

    bootstrap : bool, default=True
        Whether bootstrap samples are used when building trees. If False, the
        whole dataset is used to build each tree.

    oob_score : bool or callable, default=False
        Whether to use out-of-bag samples to estimate the generalization score.
        By default, :func:`~sklearn.metrics.accuracy_score` is used.
        Provide a callable with signature `metric(y_true, y_pred)` to use a
        custom metric. Only available if `bootstrap=True`.

    n_jobs : int, default=None
        The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`,
        :meth:`decision_path` and :meth:`apply` are all parallelized over the
        trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`
        context. ``-1`` means using all processors. See :term:`Glossary
        <n_jobs>` for more details.

    random_state : int, RandomState instance or None, default=None
        Controls both the randomness of the bootstrapping of the samples used
        when building trees (if ``bootstrap=True``) and the sampling of the
        features to consider when looking for the best split at each node
        (if ``max_features < n_features``).
        See :term:`Glossary <random_state>` for details.

    verbose : int, default=0
        Controls the verbosity when fitting and predicting.

    warm_start : bool, default=False
        When set to ``True``, reuse the solution of the previous call to fit
        and add more estimators to the ensemble, otherwise, just fit a whole
        new forest. See :term:`Glossary <warm_start>` and
        :ref:`gradient_boosting_warm_start` for details.

    class_weight : {"balanced", "balanced_subsample"}, dict or list of dicts,
    default=None
        Weights associated with classes in the form ``{class_label: weight}``.
        If not given, all classes are supposed to have weight one. For
        multi-output problems, a list of dicts can be provided in the same
        order as the columns of y.

        Note that for multioutput (including multilabel) weights should be
        defined for each class of every column in its own dict. For example,
        for four-class multilabel classification weights should be
        [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
        [{1:1}, {2:5}, {3:1}, {4:1}].

        The "balanced" mode uses the values of y to automatically adjust
        weights inversely proportional to class frequencies in the input data
        as ``n_samples / (n_classes * np.bincount(y))``

        The "balanced_subsample" mode is the same as "balanced" except that
```

```
        weights are computed based on the bootstrap sample for every tree
        grown.

        For multi-output, the weights of each column of y will be multiplied.

        Note that these weights will be multiplied with sample_weight (passed
        through the fit method) if sample_weight is specified.

    ccp_alpha : non-negative float, default=0.0
        Complexity parameter used for Minimal Cost-Complexity Pruning. The
        subtree with the largest cost complexity that is smaller than
        ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
        :ref:`minimal_cost_complexity_pruning` for details.

        .. versionadded:: 0.22

    max_samples : int or float, default=None
        If bootstrap is True, the number of samples to draw from X
        to train each base estimator.

        - If None (default), then draw `X.shape[0]` samples.
        - If int, then draw `max_samples` samples.
        - If float, then draw `max(round(n_samples * max_samples), 1)` samples. Thus,
          `max_samples` should be in the interval `(0.0, 1.0]`.

        .. versionadded:: 0.22

    Attributes
    ----------
    estimator_ : :class:`~sklearn.tree.DecisionTreeClassifier`
        The child estimator template used to create the collection of fitted
        sub-estimators.

        .. versionadded:: 1.2
           `base_estimator_` was renamed to `estimator_`.

    base_estimator_ : DecisionTreeClassifier
        The child estimator template used to create the collection of fitted
        sub-estimators.

        .. deprecated:: 1.2
            `base_estimator_` is deprecated and will be removed in 1.4.
            Use `estimator_` instead.

    estimators_ : list of DecisionTreeClassifier
        The collection of fitted sub-estimators.

    classes_ : ndarray of shape (n_classes,) or a list of such arrays
        The classes labels (single output problem), or a list of arrays of
        class labels (multi-output problem).

    n_classes_ : int or list
        The number of classes (single output problem), or a list containing the
        number of classes for each output (multi-output problem).

    n_features_in_ : int
        Number of features seen during :term:`fit`.

        .. versionadded:: 0.24
```

```
feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

n_outputs_ : int
    The number of outputs when ``fit`` is performed.

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature.  It is also
    known as the Gini importance.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

oob_score_ : float
    Score of the training dataset obtained using an out-of-bag estimate.
    This attribute exists only when ``oob_score`` is True.

oob_decision_function_ : ndarray of shape (n_samples, n_classes) or            (n_sa
mples, n_classes, n_outputs)
    Decision function computed with out-of-bag estimate on the training
    set. If n_estimators is small it might be possible that a data point
    was never left out during the bootstrap. In this case,
    `oob_decision_function_` might contain NaN. This attribute exists
    only when ``oob_score`` is True.

See Also
--------
sklearn.tree.DecisionTreeClassifier : A decision tree classifier.
sklearn.ensemble.ExtraTreesClassifier : Ensemble of extremely randomized
    tree classifiers.
sklearn.ensemble.HistGradientBoostingClassifier : A Histogram-based Gradient
    Boosting Classification Tree, very fast for big datasets (n_samples >=
    10_000).

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore,
the best found split may vary, even with the same training data,
``max_features=n_features`` and ``bootstrap=False``, if the improvement
of the criterion is identical for several splits enumerated during the
search of the best split. To obtain a deterministic behaviour during
fitting, ``random_state`` has to be fixed.

References
----------
.. [1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.
```

```
Examples
--------
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = RandomForestClassifier(max_depth=2, random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(...)
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
File:           c:\users\biggest\anaconda3\lib\site-packages\sklearn\ensemble\_forest.py
Type:           ABCMeta
Subclasses:
```

In [18]:
```
rf = RandomForestClassifier(n_estimators = 20)
```

In [19]:
```
rf.fit(X_train, y_train)
```

Out[19]:
```
RandomForestClassifier
RandomForestClassifier(n_estimators=20)
```

In [20]:
```
# MODEL EVALUATION
y_pred = rf.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[20]:
```
0.9563571428571429
```

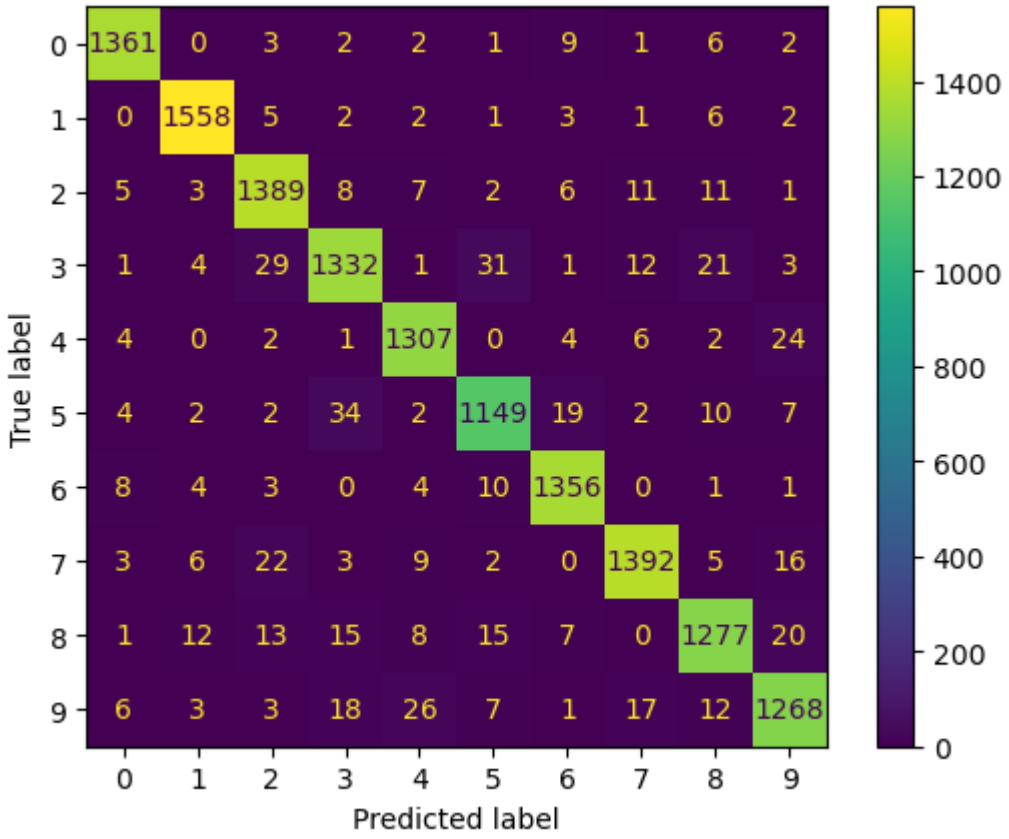
In [21]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, labels=rf.classes_.tolist()))
```

```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      1387
           1       0.98      0.99      0.98      1580
           2       0.94      0.96      0.95      1443
           3       0.94      0.93      0.93      1435
           4       0.96      0.97      0.96      1350
           5       0.94      0.93      0.94      1231
           6       0.96      0.98      0.97      1387
           7       0.97      0.95      0.96      1458
           8       0.95      0.93      0.94      1368
           9       0.94      0.93      0.94      1361

    accuracy                           0.96     14000
   macro avg       0.96      0.96      0.96     14000
weighted avg       0.96      0.96      0.96     14000
```

In [22]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
 #from sklearn.metrics import classification_report
 #assuming 'knn' is your trained model, 'X_test' are your test features
predictions = rf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf.classes_)
disp.plot()
plt.suptitle
```

Out[22]:
```
<function matplotlib.pyplot.suptitle(t, **kwargs)>
```



ADA BOOST CLASSIFIER

In [23]:
```
from sklearn.ensemble import AdaBoostClassifier
AdaBoostClassifier?
```

```
Init signature:
AdaBoostClassifier(
    estimator=None,
    *,
    n_estimators=50,
    learning_rate=1.0,
    algorithm='SAMME.R',
    random_state=None,
    base_estimator='deprecated',
)
Docstring:
An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a
classifier on the original dataset and then fits additional copies of the
classifier on the same dataset but where the weights of incorrectly
classified instances are adjusted such that subsequent classifiers focus
more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

Read more in the :ref:`User Guide <adaboost>`.

.. versionadded:: 0.14

Parameters
----------
estimator : object, default=None
    The base estimator from which the boosted ensemble is built.
    Support for sample weighting is required, as well as proper
    ``classes_`` and ``n_classes_`` attributes. If ``None``, then
    the base estimator is :class:`~sklearn.tree.DecisionTreeClassifier`
    initialized with `max_depth=1`.

    .. versionadded:: 1.2
       `base_estimator` was renamed to `estimator`.

n_estimators : int, default=50
    The maximum number of estimators at which boosting is terminated.
    In case of perfect fit, the learning procedure is stopped early.
    Values must be in the range `[1, inf)`.

learning_rate : float, default=1.0
    Weight applied to each classifier at each boosting iteration. A higher
    learning rate increases the contribution of each classifier. There is
    a trade-off between the `learning_rate` and `n_estimators` parameters.
    Values must be in the range `(0.0, inf)`.

algorithm : {'SAMME', 'SAMME.R'}, default='SAMME.R'
    If 'SAMME.R' then use the SAMME.R real boosting algorithm.
    ``estimator`` must support calculation of class probabilities.
    If 'SAMME' then use the SAMME discrete boosting algorithm.
    The SAMME.R algorithm typically converges faster than SAMME,
    achieving a lower test error with fewer boosting iterations.

random_state : int, RandomState instance or None, default=None
    Controls the random seed given at each `estimator` at each
    boosting iteration.
    Thus, it is only used when `estimator` exposes a `random_state`.
    Pass an int for reproducible output across multiple function calls.
```

```
    See :term:`Glossary <random_state>`.

base_estimator : object, default=None
    The base estimator from which the boosted ensemble is built.
    Support for sample weighting is required, as well as proper
    ``classes_`` and ``n_classes_`` attributes. If ``None``, then
    the base estimator is :class:`~sklearn.tree.DecisionTreeClassifier`
    initialized with `max_depth=1`.

    .. deprecated:: 1.2
        `base_estimator` is deprecated and will be removed in 1.4.
        Use `estimator` instead.

Attributes
----------
estimator_ : estimator
    The base estimator from which the ensemble is grown.

    .. versionadded:: 1.2
       `base_estimator_` was renamed to `estimator_`.

base_estimator_ : estimator
    The base estimator from which the ensemble is grown.

    .. deprecated:: 1.2
        `base_estimator_` is deprecated and will be removed in 1.4.
        Use `estimator_` instead.

estimators_ : list of classifiers
    The collection of fitted sub-estimators.

classes_ : ndarray of shape (n_classes,)
    The classes labels.

n_classes_ : int
    The number of classes.

estimator_weights_ : ndarray of floats
    Weights for each estimator in the boosted ensemble.

estimator_errors_ : ndarray of floats
    Classification error for each estimator in the boosted
    ensemble.

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances if supported by the
    ``estimator`` (when based on decision trees).

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.
```

```
    .. versionadded:: 1.0

See Also
--------
AdaBoostRegressor : An AdaBoost regressor that begins by fitting a
    regressor on the original dataset and then fits additional copies of
    the regressor on the same dataset but where the weights of instances
    are adjusted according to the error of the current prediction.

GradientBoostingClassifier : GB builds an additive model in a forward
    stage-wise fashion. Regression trees are fit on the negative gradient
    of the binomial or multinomial deviance loss function. Binary
    classification is a special case where only a single regression tree is
    induced.

sklearn.tree.DecisionTreeClassifier : A non-parametric supervised learning
    method used for classification.
    Creates a model that predicts the value of a target variable by
    learning simple decision rules inferred from the data features.

References
----------
.. [1] Y. Freund, R. Schapire, "A Decision-Theoretic Generalization of
       on-Line Learning and an Application to Boosting", 1995.

.. [2] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.

Examples
--------
>>> from sklearn.ensemble import AdaBoostClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
>>> clf.score(X, y)
0.983...
File:           c:\users\biggest\anaconda3\lib\site-packages\sklearn\ensemble\_weight
_boosting.py
Type:           ABCMeta
Subclasses:
```

In [60]:
```
ada = AdaBoostClassifier(
                        n_estimators = 100,
                      random_state=4,
                      algorithm='SAMME.R' )
ada.fit(X_train, y_train)
```

Out[60]:
```
AdaBoostClassifier(n_estimators=100, random_state=4)
```

In [61]:
```
# MODEL EVALUATION
y_pred = ada.predict(X_test)
```

```
accuracy_score(y_test, y_pred)
```

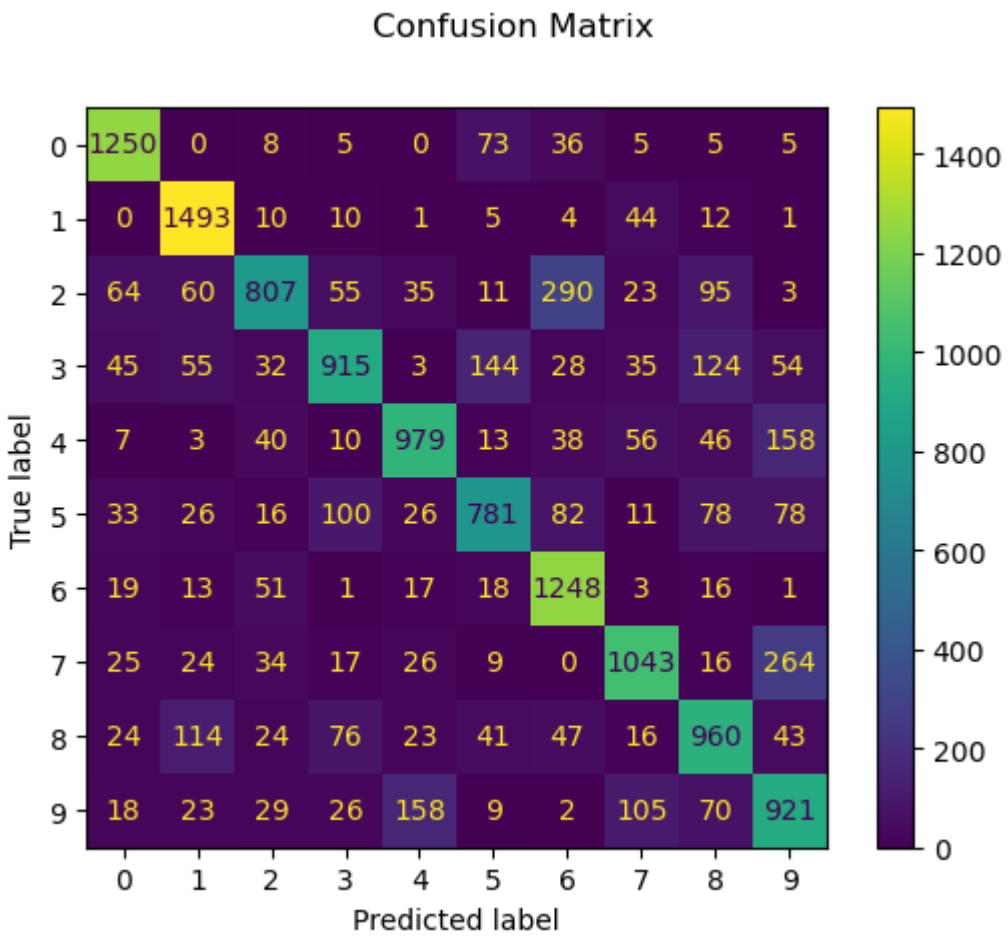
Out[61]:
```
0.7426428571428572
```

In [62]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, labels=ada.classes_.tolist()))
```

```
              precision    recall  f1-score   support

           0       0.84      0.90      0.87      1387
           1       0.82      0.94      0.88      1580
           2       0.77      0.56      0.65      1443
           3       0.75      0.64      0.69      1435
           4       0.77      0.73      0.75      1350
           5       0.71      0.63      0.67      1231
           6       0.70      0.90      0.79      1387
           7       0.78      0.72      0.75      1458
           8       0.68      0.70      0.69      1368
           9       0.60      0.68      0.64      1361

    accuracy                           0.74     14000
   macro avg       0.74      0.74      0.74     14000
weighted avg       0.74      0.74      0.74     14000
```

In [57]:
```
predictions = ada.predict(X_test)
cm = confusion_matrix(y_test, predictions)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=ada.classes_)
disp.plot()
plt.suptitle('Confusion Matrix')
```

Out[57]:
```
Text(0.5, 0.98, 'Confusion Matrix')
```


Confusion Matrix
True label
Predicted label
0 1250 0 8 5 0 73 36 5 5 5
1 0 1493 10 10 1 5 4 44 12 1
2 64 60 807 55 35 11 290 23 95 3
3 45 55 32 915 3 144 28 35 124 54
4 7 3 40 10 979 13 38 56 46 158
5 33 26 16 100 26 781 82 11 78 78
6 19 13 51 1 17 18 1248 3 16 1
7 25 24 34 17 26 9 0 1043 16 264
8 24 114 24 76 23 41 47 16 960 43
9 18 23 29 26 158 9 2 105 70 921
0 1 2 3 4 5 6 7 8 9
1400
1200
1000
800
600
400
200
0

In [ ]:

In [ ]: